COPYRIGHT NOTICE
for
An Enhanced FORTRAN Compiler

Version 9.38 Released May, 2007

Published by:

Great Migrations LLC
7453 Katesbridge Ct
Dublin, Ohio 43017
(614) 761-9816

DISCLAIMER OF WARRANTIES AND LIMITATIONS OF LIABILITIES

TRADEMARK

DEFINITION OF PURCHASE

The definition of your particular purchase is specified in the Great Migrations LLC License Agreement which came with the PROMULA FORTRAN Compiler product and which you completed and returned to the Great Migrations LLC. If you have any questions about your rights or obligations as a PROMULA FORTRAN Compiler user or believe that you have not received the complete PROMULA FORTRAN Compiler package that you purchased, please contact:

Great Migrations LLC
7453 Katesbridge Ct
Dublin, Ohio 43017
(614) 761-9816

Table Of Contents

1. INTRODUCTION

The PROMULA FORTRAN Compiler is a general-purpose, multi-dialect, and portable FORTRAN compiler. It runs on multiple platforms and supports both the ANSI FORTRAN 66 and ANSI FORTRAN 77 standard dialects, as well as a large number of common extensions such as those found in the following commercial compilers: VAX FORTRAN, PDP FORTRAN, PRIME FORTRAN, Data General FORTRAN, and Sun FORTRAN. Some Fortran 90 extensions are also supported. In cases where different versions of FORTRAN have conflicting features or conventions, a dialect selection command switch can be used to select the desired set. Extended FORTRAN applications will compile with this product on 'open' platforms as is — i.e. without making changes to their FORTRAN source code.

Validated by the GSA FORTRAN Compiler Validation Test Suite on every platform, this compiler works by producing intermediate C source code. It requires a C compiler to produce executable code and is the ideal processor for hybrid FORTRAN/C applications. FORTRAN debugging is supported by the debugger of the host C platform (e.g., the dbx tool for UNIX platforms).

The product includes both the compiler and an extensive FORTRAN runtime library (about 300 functions) in object form. The runtime library may be linked into your executables for royalty-free distribution.

1.1 User Support

If you are a licensed and registered user of the PROMULA FORTRAN compiler, then you are entitled to user support from the Great Migrations LLC.

If you encounter a problem that you cannot resolve on your own by referring to this User's Manual, you may call or write us:

Great Migrations LLC
PROMULA FORTRAN Support
7453 Katesbridge Ct
Dublin, Ohio 43017
(614) 761-9816

Your comments and suggestions about this product are always welcome.

If possible, we will provide support over the telephone. However, if the problem involves an apparent compilation or runtime library problem, we will probably need a copy of your source FORTRAN code. We will protect the full confidentiality of any sample codes that you send to us.

1.2 The Design of the Compiler

The PROMULA FORTRAN Compiler (PF) is a comprehensive FORTRAN compiler for C-based platforms. It works by converting FORTRAN source codes into intermediate C source codes. These are then processed by the C compiler of the host platform to produce object code. There are four fundamental reasons for this design: portability, integration, efficiency, and completeness.

By using C as an intermediate step, applications using this compiler achieve a high degree of portability. Our compiler is highly portable. Once your code compiles and runs under PF on any C-based platform, it will compile and run on almost

any other C-base platform. Not only are your FORTRAN source codes processed via C, but our entire FORTRAN runtime library is also written in C.

Since C is used as an intermediate language, it is trivial to integrate FORTRAN programs with other C based software such as GUIs, operating systems, data management systems, etc. We know of no major software library available today which is not usable via C. External C functions can be referenced very naturally using FORTRAN CALL statements and function references. In addition, prototype files can be supplied to ensure that the compiler properly handles call-by-value versus call-by-name.

Runtime efficiency is always a concern. At one time the C language was intended for small integer-based applications only. Early C compilers did not even support floating-point arithmetic. This is no longer true. Contemporary C compilers are fully featured, run very quickly, and produce efficient code. Benchmarks between PROMULA FORTRAN produced executables and those produced by contemporary native FORTRAN compilers, when available, typically favor PROMULA. In fact, it is typical for contemporary FORTRAN compilers to share the optimizer back-end of the native C-compiler.

PROMULA FORTRAN covers the complete current FORTRAN language as currently used and not simply as defined by the various standards. PF passes the GSA FORTRAN Compiler Validation test on every platform that it operates on. Other contemporary FORTRAN compilers cover only the FORTRAN standard along with selected extensions from other dialects. At PROMULA, our primary concern is the dialect coverage of our compiler and its runtime library. We not only support the VAX, PDP, PRIME, DG, and SUN extensions, but we also provide a dialect selection switch to allow for conflicting features.

The PROMULA FORTRAN Compiler is completely compatible with and shares its front-end with PROMULA.FORTRAN — our FORTRAN to C translator and portation support tool. In addition the compiler is compatible with our Application Management system — PROMULA. Via a simple configuration file, PF will add PROMULA data base access instructions to your program during compilation; thus, unlocking the information within those programs for external management.

It is the dream of Great Migrations LLC that ultimately there will be only one compiled language — be it ANSI C, C++, or some other language yet to be designed or discovered. All compiler technology will be focused on making this language as efficient as possible on every platform supporting it. Operating systems and language processors will be written in this language and will work through this single language, thus maximizing portability and minimizing startup time on new platforms. Variation and experimentation in hardware design and in programming languages, tools, and techniques should be encouraged. Our design makes this possible.

1.3 Notation Used

The notation used to describe FORTRAN statements throughout this manual is as follows:

UPPERCASE	indicates a statement keyword or character that is to be written as shown.
lowercase	indicates a name, number, symbol, or entity that is to be supplied by the programmer.
[ ]	indicate an optional item that can be used or omitted.
{ }	indicate that only one of the enclosed items can be used.
...	indicates that the preceding optional item in brackets can be repeated as necessary.

In program examples, a vertical ellipsis indicates that other FORTRAN statements or parts of the program have not been shown because they are not relevant to the example.

Finally, the symbol ^ indicates a blank character in cases where such is not obvious.

2. USING THE COMPILER

PF is the program that controls the compilation of FORTRAN programs. It guides files of source and object code through each phase of compilation and linking. PF has many options to assist in the compilation of FORTRAN programs; in essence, however, all you need do to produce an executable file from your FORTRAN program is to type PF followed by the name of the file or files that hold your program. PF checks whether the file names you give are reasonable, selects the right phase for each file, and performs other tasks that ease the compilation, debugging, and analysis of your programs.

The summary syntax of the PF command is as follows:

```
pf [options] filenames [options]
```

The remainder of this chapter discusses in detail the options and filenames that are accepted.

2.1 How PF Works

PF works as follows. First, it processes all entries on its command line to determine the type of tasks to be performed and the resources required to perform them. If there is any entry on the command line which PF cannot interpret, PF exits with an error message. The chapter on error messages describes these in detail.

Next PF processes each FORTRAN source file specified, along with any prototype and global files, to form intermediate C source files. These C source files are then converted to compiled form via the native C compiler. During this conversion you may get some messages from the host C compiler. These depend upon the particular platform.

Finally, if an executable is to be formed, all compiled files formed and any additional compiled and library files included on the command line are passed to the linker along with any specified linker options.

2.2 File Names

PF classifies files by their extension — i.e., by the final 1, 2, or 3 characters of the file name that are preceded by a period. Note that the PROMULA compiler is not case sensitive; however, many operating environments are. In the following discussion, the letters defining the various file types may be either in upper or lower case insofar as the compiler is concerned.

If the extension of the file name begins with an 'f', then PF assumes that it is a FORTRAN source code and passes it to the FORTRAN preprocessor and then to the C compiler for compilation. Upon completion a file with the extension 'o', or 'obj' is formed. This file may then be passed on to the linker.

If the file name extension is 'o', or 'obj' on some platforms, then PF assumes that the file contains compiled code and passes it to the linker unchanged.

If the file name extension is 'a', or 'lib' on some platforms, then PF assumes that the file is a library of compiled codes and passes it to the linker unchanged.

If the file name extension is 'pro', then PF assumes that the file contains a list of function prototypes to be used by the FORTRAN preprocessor. This file is passed to that preprocessor along with the names of the FORTRAN source codes. Multiple prototype files may be supplied. The content of prototype type files is described in a later section of this chapter.

If the file name extension is 'glb', then PF assumes that the file contains a list of the program variables that are to be made "global" for use by the PROMULA Application Development System. The content of this file and the general topic of the PROMULA interface are described in a later chapter in this manual.

2.3 Summary of Options

Each option on the PF command line may be preceded by a '-' so that it will not be interpreted as a file name. This is not normally a problem since options do not usually end with extensions; while all valid filenames must. The following is a complete list of the options. Note that, with one exception, the PF command line options are NOT case sensitive — i.e., the characters 'a' and 'A' are considered to be the same on the command line.

Option	Information	Sec	Characteristic Affected by Switch
C		2.7	Compile only
C	l,s	2.5	Treatment of short arithmetic
D	vax,pdp,p77,piv	2.5	FORTRAN dialect conventions
El	1,2,3	2.4	Echo errors at specified level
Es		2.4	Echo source code
Ex		2.4	Echo symbol cross-reference information
F	Snum,t,f,v,9	2.5	FORTRAN source format used
FI	s,l	2.5	FORTRAN INTEGER type
G		2.7	Debugging flag
I	name	2.7	C compiler include file pathname
L,l	name	2.7	Library file name or pathname for linker
O	name	2.7	Name of executable
PH	numb	2.4	Output page height
PW	numb	2.4	Output page width
QE	numb	2.6	Quantity of line number storage
QI	numb	2.6	Quantity of input record storage
QH	numb	2.6	Quantity of include file storage
QX	numb	2.6	Quantity unresolved external symbol storage
S	a,s	2.5	Storage auto or static
T	0,1	2.5	Doloop trip count assumption
UR	numb	2.5	Input unit number
UW	numb	2.5	Output unit number
UP	numb	2.5	Punch unit number
V	1,2	2.4	Verbose flag
Z	name	2.7	Include file pathname for FORTRAN processor

2.4 Echo Control Options

PF is normally silent, unless a fatal error is encountered. Messages are sent to standard output and may be redirected. Note that error messages themselves are discussed in a later chapter in this manual. The echo control options may be used to send additional information to the standard output file. This information includes the following:

Option	Information sent to standard output
EL1	Warnings about potentially serious inconsistencies or usages in the FORTRAN source code
EL2	Comments about possibly nonportable code or about code that may be incorrect and the warnings from above

EL3 Notes about standard FORTRAN violations and other miscellaneous observations and comments and warnings from above.

ES An annotated listing of the source code.

EX An alphabetical listing of the symbols used in the source along with a summary description of each and a cross-reference listing of symbol references by line number.

V1 A message each time the preprocessor opens a new source file or include file.

V2 A message each time the preprocessor starts a new subprogram as well as the messages from above.

2.4.1 Warnings, Notes, and Comments

The form and meaning of the warnings, notes, and comments are discussed in a later chapter on error messages. Suffice to say that the PF does extensive syntax checks while it is processing the source code and extensive consistency checks after it has processed each subprogram.

The following is a sample listing produced which shows the type of messages that might be produced at the EL1 level:

```
390: utest.for: W818: The argument "ia" is being defined with type integer*4 when it has
been passed an argument of type character.
```

The "W" appended to the error number indicates a warning. Messages at this level can be ignored, but they typically indicate potentially serious problems.

At the EL2 level the following additional types of messages are issued:

```
54: utest.for: C870:  The array OBUF is being subscripted with less than 1 expressions.
380: utest.for: C815: A data value of type character is being assigned to the variable IA
      of type integer*4.
520: utest.for: C816: The binary type character is being used where type integer*2 is
      expected.
558: utest.for: C861: Data is being allocated to common storage via the variable SEATRD.
820: utest.for: C866: The real*4 type has previously been assigned to UC_.
```

At this level, in addition to warnings, other usages are isolated that should either be checked or that represent potentially serious portation problems.

Finally, at the EL3 level the following additional types of messages are issued:

```
37: utest.for: N858: The identifier THERMA1 with more than 6 characters is nonstandard.
184: utest.for: N864:Declarative statements following executable statements is nonstandard.
223: utest.for: N851:The use of inline comments is nonstandard.
295: utest.for: N872:Equivalencing CBUF of type character*80 with IBUF of type INTEGER*2 is
nonstandard.
343: utest.for: N853:The standard delimeter for a character constant is the single quote.
431: utest.for: N858:The identifier PRANDOM_INDX$ with more than 6 characters is
nonstandard.
474: utest.for: N806:Omitting the comma after the I FORMAT specification is nonstandard.
545: utest.for: N823:The COMMON block DIR1 has character*1 variable VFORMS and an
unspecified variable OPTEV.
722: utest.for: N801:The INCLUDE statement is nonstandard.
1: STRUC6.INC: N801: The STRUCTURE statement is nonstandard.
800: utest.for: N833:The nonparenthetical form of the PARAMETER statement is nonstandard.
```

As can be seen, the EL3 level generates many messages and is primarily intended for those who are trying to write pure standard conforming code.

2.4.2 Monitoring Internal Operations

The V1 and V2 flags are used to monitor the internal operations of the FORTRAN preprocessor. Only simple messages are displayed. These options are generally not used in conjunction with the ES and EX flags which produce structured reports.

The following is typical output produced at the V1 level:

```
Compiling source file: utest.for
Including source file: STRUC6.INC
Including source file: STRUC6.INC
Including source file: STRUC7.INC
Including source file: STRUC7.INC
```

In essence, a message is generated each time a source file is opened.

At the V2 level, the following output might be produced:

```
Compiling source file: utest.for
Processing subprogram: utest
Processing subprogram: ana1
Processing subprogram: ana2
Processing subprogram: ana3
Including source file: STRUC6.INC
Processing subprogram: get_date
Including source file: STRUC6.INC
Processing subprogram: struc7
Including source file: STRUC7.INC
Processing subprogram: get_d
Including source file: STRUC7.INC
Processing subprogram: struc8
Processing subprogram: get_da
Processing subprogram: struc9
Processing subprogram: therma1
```

In addition to source files, as each new subprogram is compiled a message is generated.

2.4.3 Annotated Listing of Source Code

The following listing was produced via the Es option for a simple subprogram referencing a single include file:

```
PROMULA FORTRAN Compiler V4.00       Date: 08/11/92  Time: 09:35  Page: 1
File: utest.for

If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE STRUC6
       2          INCLUDE 'STRUC6.INC'
 1     1          STRUCTURE /DATE/
 1     2  1           INTEGER*4 DAY,MONTH
 1     3  1           INTEGER*4 YEAR
 1     4  1       END STRUCTURE
 1     5    C
 1     6    C          STRUCTURE /TIME/ APP_TIME(2)
 1     7    C              LOGICAL*1 HOUR,MINUTE
 1     8    C          END STRUCTURE
```

```
1     9    C
1    10    C     This is the same as:
1    11    C
1    12    C          STRUCTURE /TIME/
1    13    C              LOGICAL*1 HOUR,MINUTE
1    14    C          END STRUCTURE
1    15    C          RECORD /TIME/ APP_TIME(2)
1    16    C
1    17              STRUCTURE /TIME/
1    18  1               LOGICAL*1 HOUR,MINUTE
1    19  1           END STRUCTURE
1    20
1    21          STRUCTURE /APPOINTMENT/
1    22  1           RECORD /DATE/ APP_DATE
1    23  1
1    24  1           RECORD /TIME/ APP_TIME(2)
1    25  1
1    26  1           CHARACTER*20 APP_MEMO(4)
1    27  1           LOGICAL*1 APP_FLAG
1    28  1       END STRUCTURE
1    29           RECORD /APPOINTMENT/ NEXT_APP,APP_LIST(7)
1    30           RECORD /DATE/ TODAY
      3           WRITE(6,*) '********* STRUC6.OUT'
      4           DO 10 I = 1,7
      5  1            CALL GET_DATE(I,TODAY)
      6  1            WRITE(6,*) TODAY.DAY,TODAY.MONTH,TODAY.YEAR
      7  1            APP_LIST(I).APP_DATE = TODAY
      8  1            TODAY.DAY = TODAY.DAY + 1
      9  1   10   END DO
     10        5   FORMAT(3I5)
     11            NEXT_APP = APP_LIST(1)
     12            OPEN(1,FILE='STRUC6.BIN',FORM ='UNFORMATTED',STATUS='UNKNOWN')
     13            WRITE(1) NEXT_APP
     14            WRITE(1) APP_LIST
     15            END
```

The heading which is printed at the top of each page contains the name of this compiler along with its current version number on the left-hand-side of the page. The right-hand-side normally contains the name of the file being compiled, the date, the time, and the page number relative to the file. In this case the page width was set to be narrow, so the name of the file is placed on a second line.

The annotated listing itself contains the include file number (If), the line number within the source file, the nesting level of the statement, and the actual source code statement. The include file number is left blank for statements in the original source file. The nesting level indicator is used with declaration statements when structures are being defined. It indicates the level of nesting within the structure. For executable statements the nesting level indicator indicates the degree of nesting within DO and/or IF statements. If there is no current nesting, the nesting level is left blank.

2.4.4 Symbol Listing and Cross Reference Table

The following symbol listing and cross reference table was produced using the Ex option. Not counting the heading, which was described above, the table consists of 4 sections: include files, symbols referenced, symbol references by line number, and statement label types and references. In general, in these tables all user-defined symbols are shown in uppercase, while all descriptive symbols are shown in lowercase.

```
PROMULA FORTRAN Compiler V4.00       Date: 08/11/92  Time: 09:35  Page: 2
File: utest.for

Include files used in unit:
Seq Filename
```

```
--- --------
  1 STRUC6.INC

Symbols referenced in SUBROUTINE STRUC6

Identifier       Object     Storage         Type             Comment
----------       ------     -------         ----             -------
APP_DATE         record     APPOINTMENT     DATE             scalar
APP_FLAG         variable   APPOINTMENT     logical*1        scalar
APP_LIST         record     static          APPOINTMENT      1d-array(679)
APP_MEMO         variable   APPOINTMENT     character*20     1d-array(80)
APP_TIME         record     APPOINTMENT     TIME             1d-array(4)
APPOINTMENT      structure
DATE             structure
DAY              variable   DATE            integer*4        scalar
GET_DATE         subroutine                                  2 args(integer*4,
                                                            unspecified)
HOUR             variable   TIME            logical*1        scalar
I                variable   static          integer*4        scalar
MINUTE           variable   TIME            logical*1        scalar
MONTH            variable   DATE            integer*4        scalar
NEXT_APP         record     static          APPOINTMENT      scalar
TIME             structure
TODAY            record     static          DATE             scalar
YEAR             variable   DATE            integer*4        scalar

Symbol references by line number in SUBROUTINE STRUC6

Identifier           Line.If:u (D=defined, M=modified, U=used, P=passed)
----------           --------- ----------- ---------- ------- ---------
APP_DATE               22.01:D       7.00:U APP_FLAG              27.01:D
APP_LIST               29.01:D       7.00:M      11.00:U      14.00:U
APP_MEMO               26.01:D
APP_TIME               24.01:D
APPOINTMENT            21.01:D
DATE                    1.01:D
DAY                     2.01:D       6.00:U       8.00:U
GET_DATE                5.00:U
HOUR                   18.01:D
I                       4.00:M       5.00:P       7.00:U
MINUTE                 18.01:D
MONTH                   2.01:D       6.00:U
NEXT_APP               29.01:D      11.00:M      13.00:U
TIME                   17.01:D
TODAY                  30.00:D       5.00:P       6.00:U       7.00:U
                        8.00:M
YEAR                    3.01:D       6.00:U

Statement label types and references by line number in SUBROUTINE STRUC6

Label  Type          Line.If:u (D=defined, U=used, A=assigned)
-----  ----          --------- ----------- ------- -----------
5      format          10.00:D
10     statement        4.00:U       9.00:D
```

The include files section simply lists all include files encountered to date in the compilation along with the sequence number assigned to them. Note that the base source file has a number of zero. If there are no include files referenced in the current subprogram, then this section is omitted.

The philosophy behind the design of the symbols reference table is that the user will use this table when he wants information about a particular symbol, whose status he may not be familiar with but whose identifier he knows or has seen

somewhere in the listing. The report consists of a single alphabetical list of each symbol referenced in the subprogram. An object type, storage status, binary type, and comment are provided for each symbol.

The object type is straightforward. There are thirteen possible entries: "constant", "parameter", "variable", "subroutine", "function", "intrinsic", "namelist","entry", "statefunc", "structure", "record", "pointer", and "common". These names correspond directly to the possible FORTRAN object types. It should be mentioned that members within structure definitions are treated simply as variables or records. This convention is compatible with the approach of using a single alphabetized list of all symbols.

The storage status of a symbol can be one of four different things.

For subprogram arguments it is specified as "argument".
For variables in common blocks, it is the name of the common block containing the variable.
For members of structures, it is the name of structure containing the member.
For simple variables it is one of the following: "static", "auto", "dynamic", or "virtual".

See the Sa and Ss switches for a description of static versus auto storage. Dynamic and virtual variables can be created via the PROMULA interface described in a later chapter.

The type of a variable is simply its type specification. For records it is the structure type of the record.

The comment associated with the symbol is a function of its object type. For constant integer parameters — i.e., those that may be used in other declaration statements — the comment specifies the value of the parameter as specified or computed. For variables or records, the dimensionality is given; for arrays, the total size in bytes is also given. For subprograms the number of arguments is given along with the assumed type of each argument. Note that in the C tradition, the PROMULA FORTRAN compiler makes extensive use of subprogram prototypes and always makes certain that argument types are consistent. If they are not, it issues a warning.

The symbol reference by line number table is simply that. Along with the line number, a use type code is also specified — 'D' means 'defined', 'M' means 'modified', 'U' means 'used', and 'P' means 'passed to a subprogram'. If a symbol has multiple references in a single statement, then only one reference is reported.

Note that if include files are involved, the line number is followed by the include file sequence number. If include files are not involved, a simple sequence number is used.

The statement label types and references table is a numerical listing of the statement labels, along with their type, "statement" or "format", and a listing of the lines where they are referenced.

If the subprogram contains EQUIVALENCE statements, then a fifth table type is generated: the equivalence pairs table. Given the code fragment below:

```
48          SUBROUTINE ANA1
49          INTEGER BUF1(2048), BUF2(2048), BUF3(2048)
50          BYTE OBUF(32767)
51          EQUIVALENCE (BUF2(1), OBUF(1))
52          EQUIVALENCE (BUF3, OBUF)
62          END
```

The following equivalence pairs table is produced:

```
Equivalence pairs:

Dependent        Base               Offset
---------        ----               ------
BUF2             OBUF                    0
BUF3             OBUF                    0
```

The base variable is the variable whose storage is being used to contain the dependent variable. The offset is the byte offset within the base variable of the start of the dependent variable.

2.4.5 Controlling Page Size

The PH and PW flags are used in conjunction with the ES and EX flags to specify the desired output page height and width. The default setting for PH is 80 lines and that for PW is 132 characters. The reports produced are quite large, and the minimum setting allowed for PW is 80 characters.

Thus a command like the following:

```
pf -ph56 -pw120 -es -ex demo.for
```

would compile a FORTRAN program `'demo.for'` and would produce tables 120 characters wide with 56 lines per page.

Note that the symbol table widths are adjusted to fit into the specified page size; however, the source code listing is truncated if the source line display becomes too wide.

2.5 FORTRAN Dialect Convention Flags

Though the FORTRAN language (or Fortran as it is called in the newest standard) has been three times standardized — in 1966, 1977, and 1990 — it remains a highly nonstandard language. The dialect convention flags allow you to specify the characteristics of your particular version of FORTRAN.

2.5.1 The FORTRAN Integer Type

There is variation between FORTRAN compilers as to whether the default type of the INTEGER specification should be INTEGER*2 or INTEGER*4. In fact, an interesting aspect of many modern FORTRAN compilers is that the user may specify whether the default integer type is to be a short 16 bit representation or a long 32 bit representation on the command line.

The FIs and FIl command line options allow for this specification. The FIs specification says that the default integer type is short, INTEGER*2; while FIl specifies that it is long, INTEGER*4. The default setting for this switch is FIl.

Note that this switch also effects the LOGICAL type. Thus, if default integers are short, then so are default logicals.

2.5.2 The Treatment of Short Integer Arithmetic

In moving from one environment to another, one must always be concerned with the accuracy of floating point arithmetic; however, there is also a real problem with fixed point arithmetic even when dealing with identical word sizes. This section concerns itself with short integer arithmetic in non-short integer environments. There is a real semantics issue here: the same program compiled in different environments behaves differently even though these environments have the same word size. Consider the following FORTRAN program which does short integer additions, multiplications, and divisions in a variety of contexts.

```
PROGRAM VARI2
INTEGER*2 I1,I2,I3,I4
INTEGER*2 ISUM,IPROD,IQUOT
INTEGER*4 JSUM,JPROD,JQUOT
REAL*4 RSUM,RPROD,RQUOT
REAL*8 DSUM,DPROD,DQUOT
I1 = 20000
I2 = 30000
I3 = 200
```

```
 I4 = 300
 ISUM = I1 + I2         <— Note addition overflow
 IPROD = I3 * I4        <— Note multiplication overflow
 IQUOT = (I1 + I2) / I4 <— Note intermediate overflow
 JSUM = I1 + I2
 JPROD = I3 * I4
 JQUOT = (I1 + I2) / I4
 RSUM = I1 + I2
 RPROD = I3 * I4
 RQUOT = (I1 + I2) / I4
 DSUM = I1 + I2
 DPROD = I3 * I4
 DQUOT = (I1 + I2) / I4
 WRITE(*,'(24H Short Integer Results: ,3I13)')
+   ISUM,IPROD,IQUOT
 WRITE(*,'(24H Long Integer Results:  ,3I13)')
+   JSUM,JPROD,JQUOT
 WRITE(*,'(24H Short Real Results:    ,3F13.5)')
+   RSUM,RPROD,RQUOT
 WRITE(*,'(24H Long Real Results:     ,3F13.5)')
+   DSUM,DPROD,DQUOT
 STOP
 END
```

In running this example with various FORTRAN compilers, always on machines with 16-bit short integers (VAX, IBM mainframe, IBM PC), we have obtained the following three results:

(1) Universal promotion to long

```
Short Integer Results:        -15536        -5536          166
Long Integer Results:          50000        60000          166
Short Real Results:      50000.00000  60000.00000    166.00000
Long Real Results:       50000.00000  60000.00000    166.00000
```

(2) No automatic promotion to long

```
Short Integer Results:        -15536        -5536          -51
Long Integer Results:         -15536        -5536          -51
Short Real Results:     -15536.00000  -5536.00000    -51.00000
Long Real Results:      -15536.00000  -5536.00000    -51.00000
```

(3) Selective promotion to long

```
Short Integer Results:        -15536        -5536          166
Long Integer Results:          50000        60000          166
Short Real Results:      50000.00000  -5536.00000    166.00000
Long Real Results:       50000.00000  -5536.00000    166.00000
```

In reviewing these results, a negative number means that a short integer overflow has occurred. The typical FORTRAN result is the first one. In this instance, the output of all integer calculations is a long. That result is then converted to the desired result type. Notice that even the intermediate addition in the division example is calculated as a long.

In the second case, the result of any short integer calculation is always also short, regardless of the surrounding context. This type of result is unusual for mainframe FORTRANs and is common for PC FORTRANs. Note that Microsoft FORTRAN allows the user to select which type of convention is to be followed as a side-effect of the "WORDSIZE" metacommand.

The third case is strange and difficult to deal with. The particular result above can be gotten from VS FORTRAN. Note that in an integer context, universal promotion to integer is followed. Also in all cases, the intermediate addition result is promoted to long. But for some reason the multiplication result is allowed to overflow while the addition result is not.

Using the Cl and Cs options you can select whether you want universal promotion to long or no automatic promotion to long. There is no provision for selective promotions. Note that selective and universal promotion differ only in overflow conditions, so users from such environments should use the universal promotion to long convention. The default convention is no universal promotion — Cs. Automatic promotion is selected via the "Cl" option.

2.5.3 FORTRAN Source Format Used

In the good old days the one thing that was always the same was the basic line format used to enter FORTRAN programs. But all good things must end. Now there are at least 5 different major variations of the FORTRAN entry format that we know of. The F command line switch allows you to specify the entry format that you are using. There is an extensive discussion of the different formats in the chapter on the FORTRAN language elements in this manual. That discussion will not be repeated here. The individual settings associated with this flag are mutually exclusive and are as follows:

FSnum Selects the standard fixed format with an ending column of n. The default setting is Fs72, which is that good-old format referred to above.

Ft Selects tab format which comes from the VAX FORTRANs.

Ff Selects the free-form format which is relatively typical of those FORTRANs that accepted "terminal input".

Fv Selects the VS FORTRAN free-form format.

F9 Selects the Fortran 90 free-format

2.5.4 Default Local Variable Storage Type

There are two fundamentally different ways in which local subprogram variables can be allocated to memory. First, there is static storage. This is the default storage for local subprogram variables. It is selected via the Ss option. It is equivalent to the normal FORTRAN memory allocation. It is wasteful in that local variables all have unique memory locations, but simple to implement. Also, static storage has memory between calls. This means that local variables can retain their values between calls to their code.

Second, there is auto storage. This storage is selected vi the Sa option. Auto storage is allocated to the stack each time a subprogram is called. When the subprogram exits, the storage is returned to the stack for use by other subprograms. The advantages of auto storage are that it is fast, easy to use, and economizes on total storage used. The disadvantages are that it has no memory and that it is very limited on PC platforms where program stacks are typically quite short and always less than 64K.

Note that local variables that are initialized via a DATA statement have static storage as their default storage type, even if the Sa option is selected.

2.5.5 The Doloop trip count assumption

Below is the syntax for the FORTRAN DO statement. It looks very simple; however, what you see is unfortunately not what you get. FORTRAN compilers vary widely on how the DO statement is executed.

Syntax:

```
DO [slab[,]] v=e1,e2[,e3]
```

Where:

`slab` is the label of an executable statement called the terminal statement of the DO loop. If slab is omitted, the do loop is terminated via an END DO statement.
`v` is an integer, real, or double precision control variable.
`e1` is an initial parameter.
`e2` is a terminal parameter.
`e3` is an optional increment parameter; default is 1.

`e1`, `e2`, and `e3` are called indexing parameters; they can be integer, real, double precision, or symbolic constants, variables, or expressions.

The DO statement differs widely between FORTRAN 66 and FORTRAN 77. In FORTRAN 66, the value of `v` is not compared with that of `e2` until the bottom of the loop; therefore, the loop is always executed once. In FORTRAN 77, the comparison of `v` is performed at the top of the loop; therefore, if `e1` exceeds `e2` initially the loop is never incremented. Officially, do loops are to be executed as follows:

(1) The expressions `e1`, `e2`, and `e3` are evaluated and then converted to the type of the control variable `v`, if necessary, yielding the values `m1`, `m2`, and `m3`.

(2) The control variable `v` is assigned the value of `m1`.

(3) The iteration count is established as follows:

```
ic = MAX(INT((ms-m1+m3)/m3),mtc)
```

where `mtc` is the minimum iteration count and equals 0 if FORTRAN 66 conventions are desired and 1 if FORTRAN 77 conventions are desired.

(4) If `ic` is not zero, the loop is executed, else execution continues beyond the end of the loop.

(5) The control variable is incremented by the value `m3`, `ic` is decremented by 1, and execution loops back to step 4.

The T0 and T1 options set the value of `mtc` in the above description. Selecting T0, which is the default, selects the FORTRAN 77 convention. Selecting T1, which forces DO loops to be executed at least once, selects the FORTRAN 66 convention.

2.5.6 Specifying Unit Numbers

FORTRAN has a variety of contexts with I/O statements in which no explicit unit number is provided: PRINT, READ(*, WRITE(*, PUNCH. There are a variety of conventions as to what actual units to associate with these statement forms. The U command line switch allows you to control this number. Note that the PROMULA FORTRAN runtime library associates no significance to any particular unit number value. A unit number may be any integer value.

The URnum or Ur option tells the compiler to use unit number "num" with READ statements for which no unit number is explicitly shown. The default setting of UR specifies that READ statements for which no unit is explicitly shown should be directed to standard input.

The UWnum or UW option tells the compiler to use unit number "num" with WRITE or PRINT statements for which no unit number is explicitly shown. The default setting UW specifies that WRITE or PRINT statements for which no unit number is explicitly shown should be directed to standard output.

The UPnum or UP option tells the compiler to use unit number "num" with PUNCH statements for which no unit number is explicitly shown. The default setting UP specifies that PUNCH statements for which no unit number is explicitly shown should be directed to standard output.

Note: See the chapter on controlling runtime behavior for more discussion or the relationship between particular FORTRAN units and the standard input and output streams.

2.5.7 Selecting Dialect Conventions

The PROMULA FORTRAN Compiler is a general-purpose, multi-dialect, and portable FORTRAN compiler. It runs on multiple platforms and supports both the ANSI FORTRAN 66 and ANSI FORTRAN 77 standard dialects, as well as a large number of common extensions such as those found in the following commercial compilers: VAX FORTRAN, PDP FORTRAN, PRIME FORTRAN, Data General FORTRAN, and Sun FORTRAN. Some FORTRAN 90 extensions are also supported. In cases where different versions of FORTRAN have conflicting features or conventions a dialect selection option switch can be used to select the desired set. The particular dialect option which the compiler supports are as follows:

Option	Dialect
DVAX	Vax FORTRAN
DPDP	PDP FORTRAN
DP77	Prime FORTRAN 77
DPIV	Prime FORTRAN IV

This particular option should only be used if code is being moved directly from one of these compilers to the PROMULA compiler.

2.6 Storage Quantity Values

The PROMULA compiler is very conservative in its memory use. Extremely large programs — even those that exceed the size constraints of some mainframe compilers — can be compiled even on the traditional MS-DOS platforms. There are, however, a few miscellaneous storage areas which need to be preallocated for efficiency reasons:

(1) The table of statement line numbers, controlled by the QEn option;

(2) The current input statement, controlled by the QIn option;

(3) The control table of all include files processed, controlled by the QHn option; and

(4) The unresolved externals tables, controlled by the QXn option.

The details of these tables will not be discussed here. If one of these areas is exceeded, you will get a message telling you explicitly which has been exceeded and which option to use.

2.7 Miscellaneous Options

The miscellaneous options are used to control the behavior of the compiler and the various other tools used by it.

2.7.1 The Compile Only Option

The "c" option tells the compiler, that FORTRAN source codes are to be compiled to object form, but that no link step should be attempted.

2.7.2 The Debugging Flag

The "g" flag tells the compiler, that line number information and symbol information is to be left in the object forms so that the standard debugger can be used. See the information on the debugger for your platform for details on its use.

2.7.3 C Compiler Include File Pathname

The "Iname" option specifies the path to be used by the C compiler to find its standard include files. This option should only be used if your C compiler has been installed in a nonstandard way.

2.7.4 Library File Name or Pathname for Linker

The "Lname" uppercase version specifies the pathname name to be used in locating the runtime libraries specified via the "lname" option. The lowercase version of this option specifies additional libraries to be added into the link step, via a shorthand name. See the installation instructions for the conventions for your platform.

2.7.5 Name of executable

The "Oname" option specifies the name of the executable to be produced. No extension may be specified for this name. If this option is omitted, the name of the executable is taken from the first source or object file name encountered on the command line.

2.7.6 Location of FORTRAN Files To Be Included

It is often desirable to have include files that are referenced from within FORTRAN programs stored in some other directories. The "Zname" option can be used to specify directories to be searched for include files. The syntax of name varies for operating system to operating system. See the installation instructions for details.

If the first character of "name" is a "#" or an "@" then the remainder of name is assumed to be the name of an environment variable which contains the include file search paths.

2.8 Prototype Files

As has been discussed in several other places, FORTRAN passes all function arguments by name — i.e., it passes the address of an argument to a subprogram as opposed to its value. C, on the other hand, allows argument values to be passed directly as well as allowing the passing of argument addresses. In instances where the value of the argument is scalar and where its value is not being changed by the subprogram, passing the value of that argument is far more efficient than passing its address. Whenever possible, the FORTRAN compiler should use call-by-value. The problem is that it is not always possible to tell whether call-by-value is valid simply from the source code. Some other device is needed to tell the compiler which arguments can be passed by value.

Another problem with FORTRAN is that it is weak typing. By this is meant that it is sometimes valid to pass data of differing binary types via the same subprogram argument. The compiler needs to know when this is valid; and if it is not

valid, it needs to know which binary type is the expected one. Many perfectly valid FORTRAN programs fail either at compilation time or at execution time because of differing typing conventions and differing internal representations. There is no general solution to the compilation of "weak-typing" FORTRAN programs on multiple platforms. Such programs may work on one platform, but not on another. The compiler must be told what to do. Some device is needed to describe subprogram arguments.

A similar problem has faced C programmers as well. Consequently, the new ANSI C has introduced the notion of a "function prototype" which describes the arguments of functions in terms of their binary type and in terms of their pointer status. The conventions developed there are exactly those needed by the FORTRAN compiler as well, although they need to be extended slightly to deal with virtual variables, multiple forms, and "external name clash".

The C prototype system and its extensions to FORTRAN are discussed in this section. It is strongly recommended that you take the time to develop a set of prototypes for the subprograms within a FORTRAN program once you have operationalized it to obtain optimal runtime efficiency.

It should be pointed out that the storage of prototype information read from a configuration file has been carefully optimized, which means that there are minimal performance penalties for using large prototype files.

2.8.1 Formation of External Symbols

Before moving into the discussion of the prototype syntax, a word about how the compiler forms its own external symbols is needed. Each COMMON block name and each subprogram name are passed to the linker in a slightly modified form — an underscore character is added to the back or each name. This has pretty much become the defacto standard for FORTRAN compilers. The additional underscore reduces the likelihood of name clash with standard system resources.

2.8.2 Function Prototype Syntax

In C, a "function prototype" declaration defines the name, return type, and storage class of a function. In addition, it can define the types of some or all of the arguments for that function. The prototype declaration has the same format as the function declaration, except that it is terminated by a semicolon, and the argument identifiers are optional. If used, argument identifiers have scope only within the prototype declaration and serve only as place holders.

The syntax used for prototypes by the FORTRAN compiler is similar to that used by C; however, it is unfortunately not the same. The reason is that additional information must be supplied to the compiler, since additional problems are introduced by the weak-typing conventions of FORTRAN.

The syntax of the PROMULA prototype definition is as follows:

```
[fname] type name(type[c],type[c][,...])
      [,type name(type[c],type[c][,...])[,...]
```

Where:

`fname` is the name of the subprogram used in the FORTRAN code (without the added underscore).

`type` is one of the following C binary type specifiers (the corresponding FORTRAN type is also shown):

void	Any type or a record type
short	integer*2
double	real*8
unsigned short	logical*2
char	integer*1 or byte
long	integer*4

float	real*4
unsigned long	logical*4
dcomplex	complex*16
complex	complex*8
string	character

`name` is any valid identifier to be used in the C output

`c` is one of the following special characters:

- `*` indicates a memory pointer to the indicated type
- `+` indicates a virtual pointer to the indicated type
- `!` indicates that a value conversion should be made to the indicated type

Only a single prototype definition may occur on each record and the notation below

```
type name()
```

means that the function has no arguments. It does not mean that the function has some unspecified number of unspecified arguments.

The type specifiers have their traditional C interpretation. The "complex" and "dcomplex" types refer to single-precision complex and double-precision complex respectively. The "string" type refers to the FORTRAN style character string. A string argument actually consists of a pointer and a length specification.

PROTOTYPE definitions affect both the compilation of references to subprograms and the definitions of subprograms. The prototype file must be supplied both when subprograms are being compiled and when they are being referenced.

2.8.3 Value Parameters

If no "c" specifier follows the type specification, a value parameter is being defined. When a reference to this argument is processed, the value of the argument is passed and not its address. If the actual argument does not have the proper type then an error occurs. The only exception to this occurs when processing numeric constants. A constant with a decimal point or exponent specified may be used as either a float or double value parameter, and a numeric constant without a decimal indication with a value in the range -32767 to + 32767 may be used in either a "long" or "short" environment.

2.8.4 External Name Clash

A problem that pervades C, in particular, is the external name clash problem. One tends to use many different libraries with C. It is not at all unusual to have the same names used by different libraries. As FORTRAN moves into contemporary environments, it must face the same problem.

With PROMULA a replacement name may be specified in the prototype. Thus, the following two prototypes

```
void fread(short,long*,short)
void fwrite(short,long*,short)
```

which specify argument types for two functions could also be written

```
fread void ftread(short,long*,short)
fwrite void ftwrite(short,long*,short)
```

This second form specifies not only the argument types but also changes the external names of the subprograms. The decision to change these names is made entirely in the prototype file, no changes are needed in the actual FORTRAN

source. Note that this renaming overrides the normal addition of an underscore to the back of the name. See the section on global symbols below for more discussion on this topic.

2.8.5 Multiple Forms

Another problem that comes up has to do with multiple forms. Here, a single function in FORTRAN might have to be compiled in different ways either because of some weak typing convention or because the function is to be used in both virtual and non-virtual mode. See the chapter on the PROMULA interface for a discussion of virtual mode.

As an example, consider that you have a statistical analysis function which computes the mean and variance of a vector of values. You have compiled it twice, once using virtual conventions and once using memory conventions. Let us first see how these two versions of the following utility can be produced.

```
      SUBROUTINE ANADAT(VAL,N,XBAR,VAR)
      DIMENSION VAL(N)
      XBAR=0.0
      VAR=0.0
      DO 10 J = 1,N
      XBAR = XBAR + VAL(J)
   10 CONTINUE
      XBAR = XBAR/N
      DO 15 J = 1,N
      S = VAL(J) - XBAR
      VAR = VAR + S*S
   15 CONTINUE
      VAR = VAR/(N-1)
      RETURN
      END
```

Compiling it with the following prototype produces a memory version.

```
anadat void manadat(float*,short,double*,double*)
```

This version is like the FORTRAN original — except that its external name is `manadat` and not `anadat`.

Now compiling the identical FORTRAN code with the following prototype produces a virtual version.

```
anadat void vanadat(float+,short,double*,double*)
```

This version is called `vanadat` and assumes that the vector to be analysed is on disk and is not directly stored in memory. To achieve these two different versions two compilations have been made, but no changes have been made to the FORTRAN source.

Now, the following prototype and FORTRAN code can be compiled. For the compilation we will also read a globals file which will request that the variable "C" resides on disk; while the remaining ones reside in memory. Again see the chapter on the PROMULA interface for more information on global files.

```
anadat void vanadat(float+,short,double*,double*),
void manadat(float*,short,double*,double*)

SUBROUTINE TEST
DIMENSION A(10),B(20),C(100)
REAL*8 ABAR,AVAR,BBAR,BVAR,CBAR,CVAR
CALL ANADAT(A,10,ABAR,AVAR)
CALL ANADAT(B,20,BBAR,BVAR)
CALL ANADAT(C,100,CBAR,CVAR)
RETURN
END
```

In this function ANADAT will be compiled to as manadat when the vector is in memory and vanadat when the vector is virtual.

2.8.6 Global Symbols and Prototypes

An additional problem that comes up is that some of the runtime libraries have been implemented via FORTRAN and others via C. Some FORTRANs and/or some Cs append additional characters to each external symbol — be it a function or a subroutine. This requires that groups of global symbols, but not all, use a modified naming convention. These modifications are achieved via GLOBAL strings, which may be entered into prototype files.

The GLOBALS string consists of two characters only — the function or subroutine prefix and suffix characters. As an example, consider the following piece of a prototype file for a FORTRAN based system in which the FORTRAN compiler appends a underscore character.

```
GLOBALS " _"
void actday(long*,long*,long*,long*);
void valdt(long*,long*,long*,short*);
void fixdt(long*,long*,long*,long*,long*,long*,long*,short*);
void weeknd(long*,long*,long*,long*,long*,long*,long*,short*);
```

The actual public names are `actdat_`, `valdt_`, `fixdt_`, and `weeknd_`. The GLOBALS string preceding these tells the compiler to make this change in the external names.

2.8.7 Renaming Identifiers Only

A final capability of the prototype file is to allow the user to enter simple identifier renaming requests. The following notation

```
oldname * newname
```

in a prototype input file will rename all occurrences of `oldname` with `newname`.

3. LANGUAGE ELEMENTS

This chapter discusses the language elements of FORTRAN: FORTRAN statements, symbolic names, constants, variables, arrays, character substrings, statement order, and user-written program units.

3.1 FORTRAN Statements

FORTRAN statements are written using the FORTRAN character set which consists of 26 letters, 10 digits, and 16 special characters as follows:

	blank
)	right parenthesis
.	period or decimal point
_	underline
+	plus sign
$	dollar sign
-	minus sign
/	slash
*	asterisk
:	colon
'	single quote
&	ampersand
"	double quote
=	equals sign
(	left parenthesis
!	exclamation mark

To represent the letters, uppercase or lowercase symbols may be used. There is no distinction made between them. Remember that in the language syntax descriptions uppercase is always used for reserved words; while lowercase is always used for user supplied identifiers.

Blanks may be inserted anywhere in a statement except within a Hollerith constant or a quoted string. Such blanks are simply ignored. Alternatively, all blanks except within Hollerith constants and quoted strings may be omitted. Blanks within Hollerith constants and quoted strings are simply treated as any other character.

Characters that are not included in the character set described above can be used in Hollerith constants, in quoted strings and in comment lines. Users should consult with their local system manager for a list of these other locally available characters and their representation. Users are warned, however, that use of such other characters can reduce the portability of their programs.

A FORTRAN statement is a sequence of characters as described above. Four general statement formats are available: fixed, tab, free, and continuation.

To describe these formats, the notions of a "source line" and a "statement field" are needed. A source line in a FORTRAN program can be one of three things:

(1) A blank line

(2) A comment line

(3) A part of a statement

A FORTRAN statement is composed of four different types of fields:

(1) A statement label field

(2) A statement content field

(3) A comment field

(4) A continuation indicator field

where the sequence of content-comment-continuation fields may repeat. The line containing the statement label of the first statement field is the "initial" line and all other lines are "continuation" lines.

There are four distinctive formats that may be used to enter FORTRAN programs:

(1) Standard fixed-format lines

(2) Tab-format lines

(3) Normal free-format lines

(4) Continuation free-format lines.

Blank lines are the same for all formats. They may appear anywhere and are always ignored. Comment lines have slightly different formats depending upon the format type; however, they may appear anywhere within a program unit, including before or within a continuation line sequence.

For all formats the maximum length of the concatenation of the content fields for a given statement is 4096 characters, excluding blanks and inline comments.

3.1.1 Standard Fixed-Format Lines

The typical FORTRAN program uses standard fixed-format. Under this format a comment line is any line with a nonblank, non-numeric character in column 1. Statement lines have fields as follows:

Field	Column(s)
Statement label	1-5
Continuation	6
Content	7 - `endcol`
Comment	`endcol` + 1

The only variable aspect of this format is the value of `endcol`. When the standard format is selected there is a fixed statement ending column specified. This is usually 72, though 132 is not unusual. Any characters beyond this column are always ignored. In addition, there is an "inline comment" character (!). If this character appears anywhere on the line, not in a quoted string or a Hollerith constant, then `endcol` is one position to the left of this character and all characters to the right are part of a comment field.

The continuation character is any nonblank character in column 6, other than the character '0'.

When the default format is in effect, with a maximum ending column of 72, then the maximum number of continuation lines is 61. This assumes that all lines are completely filled — each with 66 characters.

3.1.2 Tab Format Lines

With tab format the statement label field consists of the characters which precede the first tab character, within position 1 - 8. After the first tab character, or beyond position 8 there is either a continuation indicator field or a content field. The continuation indicator field is either a numeric character or an ampersand (&). The content field continues until an end-of-line is encountered or until the inline comment character (!) is found, not within a quoted string or a Hollerith constant.

Comment lines are any lines with a nonnumeric character in the first column.

3.1.3 Normal Free-Format Lines

A free-format line is a sequence of variable length lines with no fixed field boundaries. If the first nonblank character on a line is a numeric digit, then it is considered an initial line with a statement label. If the first nonblank character is an ampersand (&), then the line is a continuation line. Otherwise, the line is initial with no statement label. The statement fields continue until the end-of-line is encountered, or until the inline comment character is encountered.

3.1.4 Continuation Free-Format Lines

The continuation free-format is unique in that the continuation indication field occurs on the line being continued and not on the continuation line. As a result, inline comments and comments within continuation sequences are not allowed. In this format, a comment line begins with a quotation mark (") in position 1. A statement begins if there is no comment. If the last character on the line is a minus sign (-) then the following line continues the present one. The continuation minus sign is not part of the statement.

3.2 Symbolic Names

Symbolic names are assigned by the user. They consist of from 1 to 255 letters, digits, the underscore (_) and the dollar sign ($). The first character may not be a digit. Letters may be uppercase or lowercase; however, the case is not significant in establishing the uniqueness of a symbolic name.

Names that are FORTRAN keywords can be used as user-assigned symbolic names without conflict. In general, however, it is good programming practice to avoid naming conflicts by assigning unique names to program entities. Certain of these conflicts are illegal and are diagnosed.

3.3 Constants

A constant is a fixed quantity. There are nine types of constants supported:

(1) integer
(2) real
(3) double precision
(4) complex
(5) double complex
(6) logical
(7) character
(8) Hollerith
(9) exact representation

Integer, real, double precision, complex, and double complex constants are considered arithmetic constants.

3.3.1 Integer Constant

An integer constant is a string of 1 to 16 decimal digits written without a decimal point. It can be positive, negative, or zero. If the integer is positive, the plus sign can be omitted; if it is negative the minus sign must be present. An integer constant must not contain a comma. Its syntax is as follows:

```
[ { + - } ]  d[d]...
```

Where:

`d` is a decimal digit.

The range of an integer constant is -2147483648 to 2147483647.

Examples of valid integer constants are as follows:

237
-74
+136772
-0024

Examples of invalid integer constants are as follows:

46.	Decimal point not allowed
23A	Letter not allowed
7,200	Comma not allowed

3.3.2 Real Constant

A real constant consists of a string of decimal digits, "coefficient", written with a decimal point or with an exponent, or with both. Commas are not allowed. The plus sign can be omitted from either the coefficient or the exponent if they are positive, but the minus sign must be present with either or both if either or both are negative. Its syntax is as follows:

```
[ { + - } ] coeff
[ { + - } ] coeff E [ { + - } ] exp
[ { + - } ] n E [ { + - } ] exp
```

Where:

`n` is an unsigned integer constant

`coeff` is a coefficient in one of the following forms:

```
n.
n.n
.n
```

`exp` is an unsigned integer constant

The range of the exponent is -307 to 308. The precision for a real constant is approximately 7 significant digits.

Optionally, a real constant can be followed by a decimal exponent, written as the letter E and an integer constant that indicates the power of ten by which the number is to be multiplied. If the E is present, the integer constant following the letter E must not be omitted. The plus sign can be omitted if the exponent is positive, but the minus sign must be present if the exponent is negative.

Examples of valid real constants:

7.5
-3.22
+4000.
.5

Examples of invalid real constants:

33,500.	Comma not allowed
2.5A	Letter not allowed

Examples of valid real constants with exponents:

42.E1	Value	42. x 10^{1}	= 420.0
.00028E+5	Value	.00028 x 10^{5}	= 28.0
6.205E6	Value	6.205 x 10^{6}	= 6205000.0
700.E-2	Value	700. x 10^{-2}	= 7.0

3.3.3 Double Precision Constant

A double precision constant is written in the same way as a real constant with exponent, except that the exponent is prefixed by the letter D instead of E. It consists of a string of decimal digits, "coefficient", written with an optional decimal point and with an exponent. Commas are not allowed. The plus sign can be omitted from either the coefficient or the exponent if they are positive, but the minus sign must be present with either or both if either or both are negative. Its syntax is as follows:

```
[ { + - } ] coeff D [ { + - } ] exp
[ { + - } ] n D [ { + - } ] exp
```

Where:

`n` is an unsigned integer constant

`coeff` is a coefficient in one of the following forms:

```
n.
n.n
.n
```

`exp` is an unsigned integer constant

The range of the exponent is -307 to 308. The precision for a real constant is approximately 13 significant digits.

Examples of valid double precision constants:

5.834D2	Value	5.834 x 10^{2}	= 583.4
14.D-5	Value	14. x 10^{-5}	= .00014
9.2D03	Value	9.2 x 10^{3}	= 9200.0
3120D4	Value	3120. x 10^{4}	= 31200000.0

Examples of invalid double precision constants:

7.2D	Exponent missing
D5	Exponent alone not allowed

2,001.3D2 Comma illegal
3.14159265 D and exponent missing

3.3.4 Complex Constant

Complex constants are written as a pair of real or integer constants, separated by a comma and enclosed in parentheses. The first constant represents the real part of the complex number, and the second constant represents the imaginary part. The parentheses are part of the constant and must always appear. Either constant can be preceded by a plus or minus sign. The syntax of a complex constant is as follows:

```
(real,imag)
```

Where:

`real` is a real or integer constant for the real part.
`imag` is a real or integer constant for the imaginary part.

Examples of valid complex constants:

(1, 7.54)	Value	1. + 7.54i, where i =$\sqrt{-1}$
(-2.1E1, 3.24)	Value	-21. + 3.24i
(4, 5)	Value	4.0 + 5.0i
0.0 - 1.0i	Value	0.0 - 1.0i

Examples of invalid complex constants:

(12.7D-4 16.1) Comma missing and double precision not allowed.
4.7E + 2,1.942 Parentheses missing

3.3.5 Double Complex Constant

Double complex constants are written as a real, integer, or double precision constant paired with a double precision constant separated by a comma and enclosed in parentheses. The first constant represents the real part of the double complex number, and the second constant represents the imaginary part. The parentheses are part of the constant and must always appear. Either constant can be preceded by a plus or minus sign. The syntax of a double complex constant is as follows:

```
(real,imag)
```

Where:

`real` is a real, integer, or double precision constant for the real part.
`imag` is a real, integer, or double precision constant for the imaginary part.

Either `real` or `imag` must be double precision.

Examples of valid double complex constants:

(1, 7.54D0)	Value 1. + 7.54i, where i =$\sqrt{-1}$
(-2.1D1, 3.24)	Value -21. + 3.24i
(4, 5D0)	Value 4.0 + 5.0i
(0D0, -1.)	Value 0.0 - 1.0i

Examples of invalid double complex constants:

(12.7E-4 16.1)	Comma missing and single precision not allowed.
4.7D0 + 2,1.942	Parentheses missing

3.3.6 Logical Constant

A logical constant takes the form of .TRUE. or .FALSE.. The periods are part of the constant and must appear. Its syntax is as follows:

```
{ .TRUE.   .FALSE. }
```

Where:

.TRUE.	Represents the logical value true.
.FALSE.	Represents the logical value false.

3.3.7 Character Constant

A character constant is a string of characters enclosed in single or double quotes. Within a character string, the delimiter is represented by two consecutive occurrences of that delimiter. Its syntax is as follows:

```
'S'
"S"
```

Where:

S	is a string of characters.

The minimum number of characters in a character constant is one, and the maximum number of characters in a character constant is 32755. The length is the number of characters in the string. Blanks are significant in a character constant. Any characters in the platform system character set can be used.

Character positions in a character constant are numbered consecutively as 1, 2, 3, and so forth, up to the length of the constant. The length of the character constant is significant in all operations in which the constant is used. The length must be greater than zero.

Examples of valid character constants:

```
'ABC'
"123"
'YEAR"S'
```

Examples of invalid character constants:

'ABC	Terminating quote is missing
'YEAR'S'	Invalid number of quotes
""	Zero length character constant

3.3.8 Hollerith Constant

A Hollerith constant is an unsigned integer constant followed by the letter H followed by the specified number of characters. Its syntax is as follows:

```
nHs
```

Where:

- `n` is an unsigned nonzero integer constant
- `s` is a string of exactly n characters.

The minimum number of characters in a Hollerith constant is one, and the maximum number of characters in a Hollerith constant is 32755. The length is the number of characters in the constant and must always be equal to the specified value of n. Blanks are significant in a Hollerith constant. Any characters in the platform system character set can be used.

Character positions in a character constant are numbered consecutively as 1, 2, 3, and so forth, up to the length of the constant. The length of the Hollerith constant is significant in all operations in which the constant is used. The length must be greater than zero.

Examples of valid Hollerith constants:

```
3HABC
3H123
6HYEAR'S
```

Examples of invalid Hollerith constants:

HABC	Missing count
0H	Zero length

3.3.9 Exact Representation Constants

Exact representation constants specify an exact sequence of bits using either octal or hexadecimal notation. This use of these constants is highly nonportable. They are alternative ways to represent numeric values and may not be assigned to character variables. Exact representation constants have the following syntax.

```
's' { O X }
```

Where:

- `s` is a sequence of octal or hexidecimal digits, depending upon whether an `O` for octal or an `X` for hexidecimal is specified.

The length of these constants depends upon the context in which they are used. They are considered to be the "typeless" numeric constants. They assume a type based upon the way in which they are used, as follows:

(1) When used in a binary operation, including assignment, the constant takes on the type of the other operand.

(2) When used in some context where a specific data type is required, such as in a subscript context or as an argument to an intrinsic function, then the constant takes on that data type.

(3) When used in a context for which no type can be determined, then the exact representation constant is assumed to be an integer constant.

Examples of valid octal constants:

`'02247'O`	Value 1191
`'10'O`	Value 8

Examples of invalid octal constants:

`'7782'O`	Invalid character 8
`'0747'`	No O after second quote

Examples of valid hexadecimal constants:

`'BF342'X`	Value 783170
`'FFB'X`	Value 4091

Examples of invalid hexadecimal constants:

`'82.8'X`	Invalid Character .
`'A9X`	Missing second quote

3.4 Variables

A variable represents a quantity with a value that can be changed repeatedly during program execution. Variables are identified by a symbolic name of 1 to 255 letters, digits, underscore (_), or dollar sign ($), beginning with a nondigit. A variable is associated with a storage location. Whenever a variable is used, it references the value currently in that location. A variable does not have be defined before being referenced for its value.

The possible types of variables are as follows:

(1) Integer
(2) Short integer
(3) Byte
(4) Real
(5) Double precision
(6) Complex
(7) Double complex
(8) Logical
(9) Short logical
(10) Logical byte
(11) Character

When the type of a variable is not explicitly specified, its type is determined by its first character. It is integer if the first letter is I, J, K, L, M, or N, and is real if the first letter is any other character. Note that the IMPLICIT statement is used to override this default convention.

3.4.1 Integer Variable

An integer variable is a variable that is typed explicitly, implicitly, or by default as integer. The range of an integer variable is from -2147483648 to 2147483647.

3.4.2 Short Integer Variable

A short integer variable is a variable that is typed explicitly as a short integer. The range of a short integer variable is from - 32768 to 32767.

3.4.3 Byte Variable

A byte variable is a variable that is typed explicitly as a byte. It behaves like an integer variable; however, its range is from 0 to 255.

3.4.4 Real Variable

A real variable is a variable that is typed explicitly, implicitly, or by default as real. The range of a real variable is from 10^{-307} to 10^{308}. The precision for a real constant is approximately 7 significant digits.

3.4.5 Double Precision Variable

A double precision variable is a variable that is typed explicitly as double precision. The value of a double precision variable can range from 10^{-307} to 10^{308} with approximately 17 significant digits of precision.

3.4.6 Complex Variable

A complex variable is a variable that is typed explicitly as complex. A complex variable may be thought of as a sequence of two real values, with the first being the real part of the complex number and the second being the imaginary part of the number.

3.4.7 Double Complex Variable

A double complex variable is a variable that is typed explicitly as double complex. A double complex variable may be thought of as a sequence of two double precision values, with the first being the real part of the double complex number and the second being the imaginary part of the number.

3.4.8 Logical Variable

A logical variable is a variable that is typed explicitly as logical. It may contain only a value of TRUE or FALSE. The storage allocated to a logical variable is the same as that allocated to an integer variable.

3.4.9 Short Logical Variable

A short logical variable is a variable that is typed explicitly as short logical. It may contain only a value of TRUE or FALSE. The storage allocated to a short logical variable is the same as that allocated to a short integer variable.

3.4.10 Logical Byte Variable

A logical byte variable is a variable that is typed explicitly as logical byte. It may contain only a value of TRUE or FALSE. The storage allocated to a logical byte variable is the same as that allocated to an integer byte variable.

3.4.11 Character Variable

A character variable is a variable that is typed explicitly as character. The length of the character variable is specified when the variable is typed as character. There is no storage distinction between a character variable of length n and an n element array of single characters. The maximum length of a character variable is 32767.

3.5 Arrays

A FORTRAN array is a set of elements identified by a single name. The name is composed of 1 to 255 letters, digits, underscore (_), or dollar sign ($), beginning with a nondigit. Each array element is referenced by the array name and a subscript.

The type of the array elements is determined by the array name in the same manner as the type of a variable is determined by the variable name. The array name can be typed explicitly with a type statement, implicitly with an IMPLICIT statement, or by default typing.

The array name and its dimensions must be declared in a DIMENSION, COMMON, or type statement. When an array is declared, the declaration of array dimensions takes the form shown below.

```
array (d[,d]...)
```

Where:

`array` is the symbolic name of the array.

`d` specifies the bounds of an array dimension and takes the form:

```
[lower:] upper
```

Where:

`lower` specifies the lower bound of the dimension. The lower bound is an integer expression with a positive, zero, or negative value. If omitted, the lower bound is assumed to be 1.

`upper` specifies the upper bound of the dimension. The upper bound is an integer expression with a positive, zero, or negative value. The upper bound must be greater than or equal to the lower bound. In the case of an assumed size array, the upper bound of the last dimension can be specified as *.

There is no limit on the number of dimensions that an array can have. The dimension bounds can be positive, negative, or zero. If the lower bound is omitted, the lower bound is assumed to be one. In this case, the upper bound must be positive. The general rule is that the upper bound must always be greater than or equal to the lower bound. The size of each dimension is indicated by the distance between the lower bound and upper bound — i.e., the span of an array dimension is given by `(upper-lower+1)`, where `upper` is the upper dimension bound and `lower` is the lower dimension bound.

For example,

```
DIMENSION RX(0:5)
```

declares a 1-dimensional array of six elements such as shown below. The values are the element locations relative to the location of the first element.

Row 0	0
Row 1	1
Row 2	2
Row 3	3
Row 4	4
Row 5	5

Alternatively,

```
DIMENSION TABLE(4,3)
```

declares a 2-dimensional array of four rows and three columns, for a total of twelve elements as shown below with the values being the element locations relative to the location of the first element.

	Column 1	Column 2	Column 3
Row 1	0	4	8
Row 2	1	5	9
Row 3	2	6	10
Row 4	3	7	11

Finally,

```
INTEGER STOR(3,4,2)
```

declares a 3-dimensional array of three rows, four columns and two pages (or planes) for a total of 24 elements. It can be viewed as follows, again with the values being the element locations relative to the location of the first element.

Plane 1

	Column 1	Column 2	Column 3	Column 4
Row 1	0	3	6	9
Row 2	1	4	7	10
Row 3	2	5	8	11

Plane 2

	Column 1	Column 2	Column 3	Column 4
Row 1	12	15	18	21
Row 2	13	16	19	22
Row 3	14	17	20	23

3.5.1 Array Storage

The elements of an array have a specific storage order, with elements of any array stored as a linear sequence of storage locations. The first element of the array begins with the first storage location or character storage position, and the last element ends with the last storage location or character storage position.

The number of storage words reserved for an array is determined by the type of the array and its size. Integer, real, and logical arrays all occupy the same amount of storage — referred to as a "storage word". Thus, for these types the number of storage words in an array equals the array size. For complex and double precision arrays, the number of storage words reserved is twice the array size. For double complex the number of storage words per element is 8. For short arrays, logical or integer, two elements occupy a storage word; while for character arrays and byte arrays four elements occupy a storage word.

Though assumptions about the relative locations of the standard numeric types can be made based upon the above notion of "storage unit" in a transportable manner, great care must be taken when making assumptions about the sizes of boundaries between these types, especially in COMMON blocks.

Storage patterns for 1-dimensional, 2-dimensional and 3-dimensional arrays were shown above. In general, array elements are stored in ascending locations by columns. The first subscript value increases most rapidly, and the last subscript value increases least rapidly.

3.5.2 Array References

Array references can be references to complete arrays or to specific array elements. A reference to a complete array is simply the array name. A reference to a specific element involves the array name followed by a subscript specification. An array element reference is also called a subscripted array name.

A reference to the complete array references all elements of the array in the order in which they are stored. For example,

```
DIMENSION XT(3)
DATA XT/1.,2.,3./
CALL CALC(XT)
```

uses the array reference `XT` in the DATA statement and the CALL statement.

A reference to an array element references a specific element and takes the form shown below:

```
array (e[,e]...)
```

Where:

`array` is the symbolic name of the array.

`e` is a subscript expression that is an integer, real, or double precision. Each subscript expression has a value that is within the bounds of the corresponding dimension. Non integer subscripts are converted to integer by truncation prior to their use.

An array element reference must specify a value for each dimension in the array. Array element references are not legal unless a value is supplied for each dimension. Each subscript value after conversion to integer must not be less than the lower bound or greater than the upper bound of the dimension. If the array is an assumed-size array with the upper bound of the last dimension specified as asterisk, the value of the subscript expression must not exceed the actual size of the dimension. The results are unpredictable if an array element reference exceeds the size of an array. For each array element reference, evaluation of the subscript expressions yields a value for each dimension and a position relative to the beginning of the complete array.

The position of an array element is calculated as shown below for an array with from 1 to 7 dimensions. The position indicates the storage location of an array element relative to the first.

```
Dimensions    Position of Array Element

1     (s1-j1)
2     (s1-j1) + (s2-j2)*n1
3     (s1-j1) + (s2-j2)*n1 + (s3-j3)*n2*n1
4     (s1-j1) + (s2-j2)*n1 + (s3-j3)*n2*n1 + (s4-j4)*n3*n2*n1
5     (s1-j1) + (s2-j2)*n1 + (s3-j3)*n2*n1 + (s4-j4)*n3*n2*n1 + (s5-j5)*n4*n3*n2*n1
6     (s1-j1) + (s2-j2)*n1 + (s3-j3)*n2*n1 + (s4-j4)*n3*n2*n1 + (s5-j5)*n4*n3*n2*n1
      + (s6-j6)*n5*n4*n3*n2*n1
7     (s1-j1) + (s2-j2)*n1 + (s3-j3)*n2*n1 + (s4-j4)*n3*n2*n1 + (s5-j5)*n4*n3*n2*n1
      + (s6-j6)*n5*n4*n3*n2*n1 + (s7-j7)*n6*n5*n4*n3*n2*n1

Where:

      ji    Lower bound of dimension i
      ki    Upper bound of dimension i
      ni    Size of dimension i, ni=(ki-ji+1)
```

`si` Value of the subscript expression specified for dimension `i`.

3.6 Character Substrings

When a character variable or character entity is declared, the entire character string can be defined and referenced. Specific parts of the character string can also be defined or referenced with character substring references. A character entity must be declared with the CHARACTER statement. The declaration of a character entity specifies the length in characters.

3.6.1 Substring References

If the name of a character entity is used in a reference, the value is the current value of the entire string. A reference to part of a string is written as a character substring whose syntax is shown below:

```
char([first]:[last])
```

Where:

`char` is the name of a character variable or array and can be an array element reference.

`first` specifies an integer, real, or double precision expression for the position of the first character of the substring. If `first` is omitted, the value is one.

`last` specifies an integer, real, or double precision expression for the position of the last character in the substring. If `last` is omitted, the value is the length of the string.

The specification of the first character in the substring is an integer, real, or double precision expression that is evaluated and converted as necessary to integer via truncation. The expression can contain array element references and function references, but evaluation of a function reference must not alter the value of the other expression in the substring reference. If the specification of `first` is omitted, the value is 1 and all characters from 1 to the value of the specification of `last` are included in the substring. The specification of `last` in the substring is an expression subject to the same rules as the specification of `first`. If `last` is omitted, the value is the length of the string and all characters from the specified first position to the end of the string are included in the substring. For a string length `len`, the value of `first` must be at least 1 and must not exceed `last`; the value of `last` must not exceed the value of `len`.

The following is an example of a string reference:

```
CHARACTER*6 S1,S2
DATA S1/'STRING'/
S2 = S1
```

Reference to `S1` is a reference to the full string.

```
S1(1:3)    Value 'STR'
S1(3:4)    Value 'RI'
S1(4:)     Value 'ING'
S1(:4)     Value 'STRI'
S1(:)      Value 'STRING'
```

Note that the substring reference `S1(:)` has the same effect as the reference `S1`, since all characters in the string are referenced.

3.6.2 Substrings and Arrays

If a substring reference is used to select a substring from an array element of a character array, the combined reference includes specification of the array element followed by specification of the substring. For example

```
CHARACTER*8 ZS(5)
CHARACTER*4 RSEN
ZS(4)(5:6)='FG'
RSEN=ZS(1)(:4)
```

The first reference refers to characters 5 and 6 in element 4 of array ZS. The second reference refers to the first four characters of the first element of array ZS.

3.7 Statement Order

The order of various statements within the program unit is relatively free, given the rules of FORTRAN.

A PROGRAM statement can appear only as the first statement in a main program. The first statement of a subroutine, function, or block data subroutine is respectively a SUBROUTINE statement, FUNCTION statement, or BLOCK DATA statement. The END statement is the last statement of each of the preceding program units.

If a variable is to be explicitly typed, then that typing must precede the first reference to that variable either in a DATA statement or in an executable statement.

Statement function definitions, PARAMETER definitions, and NAMELIST specifications must precede their first reference.

Comments can appear anywhere within the program unit. Note that any comment following the END statement is considered part of the next program unit.

3.8 User-Written Program Units

An executable program consists of one main program and optional subprograms. Both main programs and subprograms are known as program units. A program unit contains a group of FORTRAN statements, including optional comments; it is terminated by an END statement. Program units can be compiled independently of each other, but a subprogram cannot be executed except through a main program.

There are two types of subprograms: a specification subprogram and a procedure subprogram. A subprogram that begins with a BLOCK DATA statement is a specification subprogram. It is used to enter initial values for variables and array elements in named common blocks. A subprogram that begins with a SUBROUTINE statement or a FUNCTION statement is a procedure subprogram known as a subroutine subprogram or a function subprogram, respectively. It can accept one or more values through a list of arguments, common blocks, or both.

A procedure is a function or a subroutine subprogram that can be executed many times. A subroutine subprogram begins with a SUBROUTINE statement and terminates with an END statement; it can return one or more values to the referencing program unit.

A function is used only in expressions to supply a value to the expression. Functions can occur in two forms: as a user-written function subprogram beginning with a FUNCTION statement, terminating with an END statement, and containing other statements; as a single statement written by the user.

A main program is a program unit that does not begin with a FUNCTION, SUBROUTINE, BLOCK DATA, or ENTRY statement. The main program should have a PROGRAM statement (optional) and at least one executable statement followed by an END statement. The execution of any program begins with the main program unit. No executable program can have more than one main program unit.

The main program can be compiled independently of any subprograms. However, when a main program is loaded into memory for execution, all the required subprograms must be loaded with it prior to its execution.

3.8.1 Program Unit and Procedure Communication

Communication between the referencing program unit and the referenced procedure is accomplished by passing actual arguments and by using common blocks. Common blocks can be used to pass data to a subprogram, but not to an intrinsic function or a statement function. Data must be passed to these functions through an argument list.

When passing arguments, actual arguments in the referencing program unit are associated with the referenced procedure through dummy arguments. Actual arguments appear in the argument list of the referencing program unit. The referencing program unit passes actual arguments to the referenced procedure. The procedure receives values from the actual arguments and returns values to the referencing program unit. Actual arguments can be constants, symbolic names of constants, variables, array names, array elements, function references, and expressions. An actual argument cannot be the name of a statement function within the referencing program unit.

Dummy arguments appear in the argument list of the referenced procedure. Within the referenced procedure, the dummy arguments are associated with the actual arguments passed. Procedures use dummy arguments to indicate the types of actual arguments, the number of arguments, and whether each argument is a variable, array, procedure, or statement label. Dummy arguments for statement functions can only be variables. Since all names are local to the program unit, the same dummy argument name can be used in more than one procedure. A dummy argument appearing in a SUBROUTINE, FUNCTION, or ENTRY statement must not appear in EQUIVALENCE, DATA, PARAMETER, SAVE, INTRINSIC, or COMMON statements except as a common block name. Dummy arguments used in array declarations for adjustable dimensions must be type integer. Dummy arguments representing array names must be dimensioned.

When a procedure is executed, the actual arguments and dummy arguments are matched up and each actual argument replaces each dummy argument. The type of the actual argument and the dummy argument must be the same. The

actual arguments must be in the same order and there must be the same number as the dummy arguments in the referenced procedure. The actual arguments that are evaluated before the association of arguments include: expressions, substring expressions, and array subscripts. If the actual argument is a procedure name, the procedure must be available for execution at the time of the reference to the procedure.

A dummy argument is undefined unless it is associated with an actual argument. Argument association can exist at more than one level of procedure reference, and terminates within a program unit at the execution of a RETURN or END statement. A subprogram reference can cause a dummy argument to be associated with another dummy argument in the referenced procedure.

For type character, both the dummy and actual arguments must be of type character, and the length of the actual argument must be greater than or equal to the length of the dummy argument. If the length of the actual argument of type character is greater than the length of the dummy argument, only the leftmost characters of the actual argument, up to the length of the dummy argument, are used as the dummy argument.

If a dummy argument is an array name, length applies to the entire array and not to each array element. Length of array elements in the dummy argument can be different from length of array elements in the actual argument. The total length of the actual argument array must be greater than or equal to the total length of the dummy argument array.

When an actual argument is a character substring, the length of the actual argument is the length of the substring. If the actual argument expression involves concatenation, the sum of the lengths of the operands is the length of the actual argument.

A variable in a dummy argument can be associated with a variable, array element, substring, or expression in the actual argument. A procedure can define or redefine the associated dummy argument if the actual argument is a variable name, array element name, or substring name. The procedure cannot redefine the dummy argument if the actual argument is a constant, a symbolic constant, a function reference, an expression using operators, or an expression enclosed in parentheses.

The array declaration in a type, COMMON, or DIMENSION statement provides the information needed for the array during the execution of the program unit. The actual argument array and the dummy argument array can differ in the number of the dimension and size of the array. A dummy argument array can be associated with an actual argument that is an array, array element, or array element substring.

If the actual argument is a noncharacter array name, the size of the actual argument array cannot be less than the size of the dummy argument array. Each actual argument array element is associated with the dummy argument array element that has the corresponding subscript value.

An association exists for array elements in a character array. Note that unless the lengths of the elements in the dummy and actual argument agree, the dummy and actual argument array elements might consist of different characters. For example, if a program unit has the following statements:

```
DIMENSION A(2)
CHARACTER A*2
.
.
.
CALL SUB(A)
```

and the subroutine has the following statements:

```
SUBROUTINE SUB(B)
DIMENSION B(2)
CHARACTER B*1
```

then the first character of `A(1)` corresponds to `B(1)` and the second character of `A(1)` corresponds to `B(2)`.

If the actual argument is a noncharacter array element name, the size of the dummy argument cannot exceed `(as+1-av)`, where `as` is the size of the actual argument array and `av` is the subscript value of the array element. For example, if the program unit has the following statements:

```
DIMENSION ARRAY(20)
.
.
.
CALL CHECK(ARRAY(3))
```

then the value of `as` is 20, and `av` is 3. The maximum dummy array size is 18 for the subroutine:

```
SUBROUTINE CHECK (DUMMY)
DIMENSION DUMMY(18)
.
.
.
SWAP=DUMMY(2)
```

Actual argument array elements are associated with dummy argument array elements, starting with the first element passed. In the example, `DUMMY(2)` is associated with `ARRAY(4)`, and `DUMMY(18)` is associated with `ARRAY(20)`.

The association for characters is basically the same as for noncharacter array elements. The actual argument for characters can be an array name, array element name, or array element substring name. If the actual argument begins at character

storage position `acu` of an array, then the first character storage position of the dummy argument array becomes associated with character storage position `acu` of the actual argument array, and so forth to the end of the dummy argument array.

A dummy argument that is a dummy procedure can be associated only with an actual argument that is an intrinsic function, external function, subroutine, or another dummy procedure. If the dummy argument is used as an external function, the actual argument that is passed must be a function or dummy procedure. The type of the dummy argument must agree with the type of result of all specific actual arguments that become associated with the dummy argument. When a dummy argument is used as an external function and is the name of an intrinsic function, the intrinsic function name corresponding to the dummy argument name is not available. If the dummy argument is referenced as a subroutine, the actual argument must be the name of a subroutine or dummy procedure, and the dummy argument must not appear in a type statement or be referenced as a function.

A dummy argument that is an asterisk can only appear in the argument list of a SUBROUTINE or ENTRY statement in a subroutine subprogram. The actual argument is an alternate return specifier in the CALL statement.

3.8.2 Adjustable Dimensions

Adjustable dimensions enable creation of a more general subprogram that can accept varying sizes of array arguments. For example, a subroutine with a fixed array can be declared as:

```
SUBROUTINE SUM(A)
DIMENSION A(10)
```

The maximum array size subroutine `SUM` can accept is 10 elements. If the same subroutine is to accept an array of any size, it can be written as:

```
SUBROUTINE SUM(A, N)
DIMENSION A(N)
```

Value `N` is passed as an actual argument. Adjustable dimensions can also be passed through common variables. For example,

```
SUBROUTINE SUB(A)
COMMON/B/M,N
DIMENSION A(M,N)
```

Dimension of array `A`, in subroutine `SUB`, is specified by the values `M` and `N` passed through the common block `B`.

Character strings and arrays can also be adjustable. For example,

```
SUBROUTINE MESSAG(X)
CHARACTER X*(*)
PRINT *, X
```

The subroutine declares `X` with a length of `(*)` to accept strings of varying size. Note that the length of the string is not passed explicitly as an actual argument.

Another form of adjustable dimension is the assumed-size array. In this case, the upper bound of the last dimension of the array is specified by an asterisk. The value of the dimension is not passed as an argument, but is determined by the number of elements stored in the array. If an array is dimensioned `*`, the array in the calling program must be large enough to contain all the elements stored in it in the subprogram.

Use of the asterisk form of the adjustable dimension prevents subscript checking for the array, so the user must be careful not to reference outside the array bounds. Use of this form is preferable to the common practice of declaring arrays to have dimension 1.

3.8.3 Using COMMON Blocks

Common blocks can be used to transfer values between a referencing program unit and a subprogram. Common blocks can reduce the number of storage units required for a program by enabling two or more subprograms to share some of the same storage units. The variables and arrays in a common block can be defined and referenced in all subprograms that contain a declaration of that common block. The names of the variables and arrays in the common block can be different for each subprogram. The association is by storage and not by name.

Common blocks cannot be used to pass data to intrinsic functions or statement functions; the method used to pass data to these procedures is through an argument list.

A reference to data in a common block is valid if the data is defined and is the same type as the type of the name used in the main program or subprogram. The exceptions to agreement between the type in common and the type of the reference are:

(1) either part of a complex entity can be referenced as real;

(2) character arrays may have different lengths and/or dimensionality.

4. EXPRESSIONS, LVALUES, ASSIGNMENTS, AND STATEMENT FUNCTIONS

Expressions are formed from a combination of operators, operands, and parentheses. Assignment statements are executable statements that use expressions to define or redefine the values of variables.

4.1 Expressions

Expressions are classified in two ways: by whether or not they are constant and by their type. A constant expression is an expression in which only constants (or symbolic constants) and operators are used. If an arithmetic expression is written using only constants and operators, the expression is an arithmetic constant expression. If a character or logical expression is written using only constants and operators, the expression is, respectively, a character constant expression, or logical constant expression.

The types of expressions are: arithmetic, character, logical, and relational. The relational expressions are not fully independent and are used as parts of logical expressions. Each of these types is discussed.

4.1.1 Arithmetic Expression

An arithmetic expression is a sequence of unsigned constants, symbolic constants, variables, array elements, and function references separated by operators and parentheses. For example,

```
(A-B)*F + C/D**E
```

is a valid arithmetic expression. Formally, the syntax of an arithmetic expression is as follows:

```
term
+ term
- term
arithexp + term
arithexp - term
```

Where:

`term` is an arithmetic term in one of the forms:

```
fact
term * fact
term / fact
```

`fact` is an arithmetic factor in one of the forms:

```
prim
prim ** fact
```

`prim` is an arithmetic primary, which can be an arithmetic expression enclosed in parentheses, or any of the following:

Unsigned arithmetic constant
Arithmetic symbolic constant

Arithmetic variable
Arithmetic array element reference
Arithmetic function reference

An arithmetic expression can be an unsigned arithmetic constant, symbolic name of an arithmetic constant, arithmetic variable reference, arithmetic array element reference, or arithmetic function reference. More complicated arithmetic expressions can be formed by using one or more arithmetic operands together with arithmetic operators and parentheses. Arithmetic operands identify values of type integer, real, complex, short integer, byte, double precision, or double complex.

The arithmetic operators are shown below.

Operator	Representing	Use	Meaning
**	Exponentiation	x1 ** x2	Exponentiate x1 to the power x2
*	Multiplication	x1 * x2	Multiply x1 and x2.
/	Division	x1 / x2	Divide x1 by x2.
+	Addition	x1 + x2	Add x1 and x2.
+	Identity	+ x2	Same as x2.
-	Subtraction	x1 - x2	Subtract x2 from x1.
-	Negation	- x2	Negate x2.

Each of the operators **, /, and * operates on a pair of operands and is written between the two operands. Each of the operators + and - either operates on a pair of operands and is written between the two operands, or operates on a single operand and is written preceding that operand.

The interpretation of a division can depend on the data types of the operands.

A set of rules establishes the interpretation of an arithmetic expression that contains two or more operators. A precedence among the arithmetic operators determines the order in which the operands are to be combined:

**	Highest
* and /	Intermediate
+ and -	Lowest

For example, in the expression

```
-A**2
```

the exponentiating operator `(**)` has precedence over the negation operator `(-)`. The operands of the exponentiation operator are combined to form an expression used as the operand of the negation operator. The expression is the same as the expression `-(A**2)`.

Successive exponentiations are combined from right to left. For example,

```
2**3**2
```

is interpreted as

```
2**(3**2)
```

Two or more multiplication or division operators are combined from left to right.

Two or more addition or subtraction operators are combined from left to right. Note that arithmetic expressions containing two consecutive arithmetic operators, such as `A**-B` or `A+-B` are not permitted. However, expressions such as `A**(-B)` and `A+(-B)` are permitted.

Subexpressions containing operators of equal precedence are evaluated from left to right. The compiler may reorder individual operations that are mathematically associative and/or communative to perform optimizations such as removal of repeated subexpressions. The mathematical results of the reordering are correct but the specific order of evaluation is indeterminate. For example, the expression `A/B*C` is guaranteed to equal algebraically `(AC/B)`, not `A/(BC)`, but the specific order of evaluation by the compiler is indeterminate.

An arithmetic constant expression contains only arithmetic constants, symbolic names of arithmetic constants, or arithmetic constant expressions enclosed in parentheses. The exponentiation operator is not permitted unless the exponent is of type integer.

An integer constant expression is an arithmetic constant expression in which each constant or symbolic name of a constant is of type integer, short integer, or byte.

The data type of an arithmetic expression containing one or more arithmetic operators is determined from the data types of the operands. When an arithmetic operator combines operands of the same type, then the result is of that type. When the operands are of different types, the lower type is promoted to the higher type prior to the evaluation of the operator. Using this terminology the operand types from highest to lowest are double complex, complex, double precision, real, integer, short integer, and byte.

4.1.2 Character Expression

A character expression is used to express a character string. Evaluation of a character expression produces a result of type character. The simplest form of a character expression is a character constant, symbolic name of a character constant, character variable reference, character array element reference, character substring reference, or character function reference. More complicated character expressions can be formed by using one or more character operands together with character operators and parentheses. The only character operator available is // which performs a concatenation.

The result of a concatenation operation is a character string concatenated on the right with another string and whose length is the sum of the lengths of the strings. For example, the value of 'AB' // 'CDE' is the string 'ABCDE'. A character expression and the operands of a character expression must identify values of type character.

Two or more concatenation operators are combined from left to right to interpret the expression. For example, the interpretation of the character expression

```
'AB' // 'CD' // 'EF'
```

is the same as the interpretation of the character expression

```
('AB' // 'CD') // 'EF'
```

The value of the preceding expression is the same as that of the constant 'ABCDEF'.

Note that parentheses have no effect on the value of a character expression. Thus, the expression

```
'AB'//('CD'//'EF')
```

has the same value as the preceding expressions.

A character constant expression is a character expression in which each operand is a character constant, the symbolic name of a character constant, or a character constant expression enclosed in parentheses.

4.1.3 Logical Expression

A logical expression is used to express a logical computation. Evaluation of a logical expression, `logexp`, produces a result of type logical, with a value of TRUE or FALSE. It has the following syntax:

```
logdis
logexp .EQV. logdis
logexp .NEQV. logdis
logexp .XOR. logdis
```

Where:

`logdis` is a logical disjunction in either form:

```
logterm
logdis .OR. logterm
```

`logterm` is a logical term in either form:

```
logfact
logterm .AND. logfact
```

`logfact` is a logical factor in either form:

```
logprim
.NOT. logprim
```

`logprim` is a logical primary. A logical primary can be a logical expression enclosed in parentheses, a relational expression, or any of the following:

Logical constant
Logical symbolic constant
Logical, short logical, or logical byte variable
Logical, short logical, of logical byte array element reference
Logical function reference

The simplest form of a logical expression is a logical constant, symbolic name of a logical constant, logical variable reference, logical array element reference, logical function reference, or relational expression. More complicated logical expressions can be formed by using one or more logical operands together with logical operators and parentheses.

The logical operators are shown below.

Operator	Representing	Meaning
.NOT.	Negation	Complement x
.AND.	Conjunction	Product of x1 and x2
.OR.	Inclusive disjunction	Sum of x1 and x2
.EQV.	Equivalence	Is x1 equivalent to x2?
.NEQV.	Nonequivalence	Is x1 not equivalent to x2?
.XOR.	Exclusive disjunction	Difference of x1 and x2

A set of rules establishes the interpretation of a logical expression that contains two or more logical operators. A precedence among the logical operators determines the order in which the operands are to be combined, unless the order is changed by the use of parentheses. The precedence of the logical operators is:

.NOT. Highest
.AND.
.OR.

.EQV. or .NEQV. or .XOR.	Lowest

For example, in the expression

```
A .OR. B .AND. C
```

the `.AND.` operator has higher precedence than the `.OR.` operator; therefore, the interpretation is the same as

```
A .OR. (B .AND. C)
```

Logical quantities are combined from left to right when a logical expression contains two or more `.AND.` operators, two or more `.OR.` operators, or two or more `.EQV.`, `.NEQV.`, or `.XOR.` operators.

The value of a logical factor involving any logical operator is shown below:

x1	x2	.NOT.x2	x1.AND.x2	x1.OR.x2	x2.EQV.x2	x1.NEQV.x2	x1.XOR.x2
T	T	F	T	T	T	F	F
T	F	T	F	T	F	T	T
F	T	F	F	T	F	T	T
F	F	T	F	F	T	F	F

A logical constant expression contains only logical constants, symbolic names of logical constants, relational expressions which contain only constant expressions, or logical constant expressions enclosed in parentheses.

4.1.4 Relational Expression

A relational expression can appear only within logical expressions. Evaluation of a relational expression produces a logical result with a `TRUE` or `FALSE` value. A relational expression used as a primary in a logical expression is in one of the forms:

```
arithexp rop arithrexp
charexp rop charexp
```

Where:

`rop`	is one of the relational operators: .LT. .LE. .EQ. .NE. .GT. .GE.
`arithexp`	is an arithmetic expression.
`charexp`	is a character expression.

A relational expression is used to compare the values of two arithmetic or two character expressions. A relational expression cannot be used to compare the value of an arithmetic expression with the value of a character expression.

The relational operators are shown below:

Operator	Representing	Meaning
.LT.	Less than	Is x1 less than x2?
.LE.	Less than or equal to	Is x1 less than or equal to x2?
.EQ.	Equal to	Is x1 equal to x2?
.NE.	Not equal to	Is x1 not equal to x2?
.GT	Greater than	Is x1 greater than x2?
.GE	Greater than or equal to	Is x1 greater than or equal to x2?

An operand of type complex or double complex is permitted only when the relational operator is .EQ. or .NE.

An arithmetic relational expression has the logical value TRUE only if the values of the operands satisfy the relation specified by the operator. If the two arithmetic expressions are of different types, then the operand of the lower type is promoted to the higher type using the same rules as are used for arithmetic expressions, prior to the comparison.

A character relational expression has the logical value TRUE only if the values of the operands satisfy the relation specified by the operator. The character expression `X1` is considered to be less than `X2` if the value of `X1` precedes the value of `X2` in the collating sequence; `X1` is greater than `X2` if the value of `X1` follows the value of `X2` in the collating sequence. Note that the collating sequence in use determines the result of the comparison. If the operands are of unequal length, the shorter operand is extended on the right with blanks to the length of the longer operand.

4.1.5 General Rules for Expressions

The order in which operands are combined using operators is determined by:

1. Use of parentheses
2. Precedence of the operators
3. Right-to-left interpretation of exponentiations
4. Left-to-right interpretation of multiplications and divisions
5. Left-to-right interpretation of additions and subtractions in an arithmetic expression
6. Left-to-right interpretation of concatenations in a character expression
7. Left-to-right interpretation of .NOT. operators
8. Left-to-right interpretation of .AND. operators
9. Left-to-right interpretation of .OR. operators
10. Left-to-right interpretation of .EQV., .NEQV., and .XOR. operators in a logical expression or Boolean expression

Precedences exist among the arithmetic and logical operators. There is only one character operator. No precedence exists among the relational operators. The precedences among the operators are:

Arithmetic	Highest
Character	
Relational	
Logical	Lowest

An expression can contain more than one kind of operator. For example, the logical expression

```
L .OR. A + B .GE. C
```

where `A`, `B`, and `C` are of type real, and `L` is of type logical, contains an arithmetic operator, a relational operator, and a logical operator. This expression would be interpreted as

```
L .OR. ((A + B) .GE. C)
```

Any arithmetic operation whose result is not mathematically defined is prohibited: for example, neither dividing by zero nor raising a zero-valued primary to a zero-valued or negative-valued power is allowed.

4.2 LVALUES

An lvalue is the name of a variable or array element of type integer, short integer, byte, real, double precision, complex, or double complex. The type of the lvalue is the same as the type of the variable or array element which forms it. The term "lvalue" is an abbreviation of the term "left-hand-value". The name is derived from the fact that only lvalues may appear on the left-hand-side of an assignment statement. Lvalues are contrasted with expressions in that expressions produce a value; while lvalues can receive a value.

4.3 Assignment Statements

There are four types of assignment statements:

Arithmetic
Character
Logical
Statement label with the ASSIGN statement

The first three types of assignment are discussed below. The ASSIGN statement is discussed in a later chapter. As is the normal convention, the action of the ASSIGN statement is not considered to be "assignment" as that term will be used in this discussion.

4.3.1 Arithmetic Assignment

The arithmetic assignment statement is shown below

```
v = e
```

Where:

v is an lvalue of type integer, short integer, byte, real, double precision, complex, or double complex.
e is an arithmetic expression.

After evaluation of arithmetic expression e, the result is converted to the type of v using the standard intrinsic conversion functions. The result is then assigned to v, and v is defined or redefined with that value.

4.3.2 Character Assignment

The character assignment statement is shown below.

```
v = e
```

Where:

v is the name of a character variable, character array element, character substring.
e is a character expression.

The character expression e is evaluated, and the result is then assigned to v. The variable v and expression e can have different lengths. If the length of v is greater than the length of e, e is extended to the right with blank characters until it is the same length as v. If the length of v is less than the length of e, e is truncated from the right until it is the same length as v.

Only as much of the value of e must be defined as is needed to define v.

In the example

```
CHARACTER A*2, B*4
A=B
```

the assignment `A=B` requires that the substring `B(1:2)` be defined. It does not require that the substring `B(3:4)` be defined. If v is a substring, e is assigned only to the substring. The definition status of substrings not specified by v is unchanged.

4.3.3 Logical Assignment

The logical assignment statement is shown below.

```
v = e
```

Where:

`v` is the name of a logical, short logical, or logical byte variable or logical array element.
`e` is a logical expression.

The logical expression is evaluated and the result is then assigned to `v`. Note that `e` must have a value of either `.TRUE.` or `.FALSE.`.

4.4 Statement Functions

A statement function is a user-defined procedure which has the same basic syntax as an assignment statement. It is a nonexecutable, single-statement computation that applies only to the program unit containing the definition.

Within a program unit, a statement function must appear after the specification statements and before the first executable statement in the unit. A statement function must not directly or indirectly reference itself.

Syntax:

```
fun([d[,d]...])) = expr
```

Where:

`fun` is a symbolic name which identifies the statement function

`d` is a statement function dummy argument. There must be at least one dummy argument.

`expr` is an expression in which each primary is one of the following:

an `expr` enclosed in parentheses
a constant
a symbolic constant
a variable reference
an array element reference
an intrinsic function reference
a reference to a statement function which appears in the same program unit
an external function reference
a substring reference

The symbolic name of the function is a variable and contains the value of the expression after execution. During execution, the actual argument expressions are evaluated, converted if necessary to the types of the corresponding dummy arguments according to the rules for assignment, and passed to the function. Thus, an actual argument cannot be an array name or a function name. In addition, if a character variable or array element is used as an actual argument, a substring reference to the corresponding dummy argument must not be specified in the statement function expression. The expression of the function is evaluated, and the resulting value is converted as necessary to the data type of the function.

The symbolic name of a statement function is local and must not be the same as any other local name in the program unit, except a common block name. The name of a statement function cannot be an actual argument and must not appear in an INTRINSIC or EXTERNAL statement. If the statement function is used in a function subprogram, then the statement

function can contain a reference to the name of the function subprogram or any of its entry names as a variable, but not as a function.

Each variable reference in the expression can be either a reference to a variable within the same program unit or to a dummy argument of the statement function. Statement functions can reference dummy variables that appear in a SUBROUTINE, FUNCTION, or ENTRY statement, but that statement must precede the statement function. Statement function dummy arguments can have the same names as variables defined elsewhere in the same program unit without conflict. Any reference to the name inside the function refers to the dummy argument, and any reference to the name outside the function definition refers to the variable.

A statement function is referenced through its statement function name. When the statement function name is referenced in an expression, the statement function is evaluated. The actual arguments are evaluated and converted to the type of the corresponding dummy argument; the resulting values are used in place of the corresponding dummy arguments in evaluation of the statement function expression. The definition of a statement function must not directly or indirectly reference itself. The statement function name can appear anywhere in an expression where an operand of the same type can be used.

The type of the statement function result is the type of the statement function name. The arguments must agree in order and number with the corresponding dummy arguments.

A statement function can be referenced only in the program unit where the statement function appears.

5. STATEMENTS SUPPORTED

This chapter describes the statements of FORTRAN other than assignment statements. Each statement is identified by the keyword which introduces it. The statement descriptions themselves are organized alphabetically to simplify using this manual as a reference guide.

5.1 ASSIGN Statement

The ASSIGN statement assigns a statement label to an integer variable.

Syntax:

```
ASSIGN label TO identifier
```

Where:

`label`	is the label of an executable statement or a FORMAT statement
`identifier`	is the identifier of a scalar integer variable

Description:

The ASSIGN statement assigns a statement label to an integer variable. The value assigned represents the label of an executable statement or a FORMAT statement. It is not the value of the `label` itself. The labeled statement must appear in the same program unit as the ASSIGN statement. While the variable contains an assigned value, it cannot be used in any statement other than an assigned GOTO statement, or in the FORMAT position of an input/output statement.

The ASSIGN statement is an executable statement; therefore, a variable must be ASSIGNed a label to execution of the assigned GOTO statement or the input/output statement that references the assigned label.

Examples:

The following assigns the label of an executable statement to a variable for later use in an ASSIGNed GOTO statement.

```
        ASSIGN 10 TO iswit
        GOTO iswit (5,10,15,20)
10      STOP
```

Execution of the ASSIGN statement prior to the execution of the GOTO statement will cause the program unit to branch to the statement labeled 10. The following assigns the label of a FORMAT statement to a variable for later use in a FORMATted input or output statement.

```
15      FORMAT(1X,2F10.5)
        ASSIGN 15 TO ifmt
        WRITE(*,ifmt) a, b
```

Execution of the ASSIGN statement prior to the execution of the WRITE statement will write the values of `a` and `b` in accordance with FORMAT 15.

5.2 BACKSPACE Statement

The BACKSPACE statement positions a file to the start of the preceding record.

Syntax:

```
BACKSPACE unit

BACKSPACE ( [ UNIT= ] unit [ ,IOSTAT= status ] [ ,ERR= err ] )
```

Where:

`unit` is an integer expression

`status` is an integer lvalue

`err` is the label of an executable statement

Notes:

The simplest form of the BACKSPACE statement consists of a single unit specification not enclosed in parentheses. With this form no additional parameters can be specified. If the parenthetical form of BACKSPACE is used, then the "UNIT=" specification is optional; however, when omitted the unit specification must be the first specification. Other than the above, the order of the parameters within the parenthetical version of the BACKSPACE statement is free.

Description:

The BACKSPACE statement backspaces the file currently under the specified unit number one record. Any file, regardless of its type or open status, may be backspaced. When the file is positioned at beginning-of-information, this statement acts as a do-nothing statement.

If an error occurs as a result of the backspace, and if neither `err` nor `status` are supplied then execution will terminate with an error code set. If either or both are supplied, the execution continues despite any errors.

If supplied, the `status` lvalue receives the runtime error code for the error condition encountered or a zero, if no error occurred. If `err` is supplied, then execution will branch to the statement labeled by it if an error occurs.

Examples:

The files associated with units 1 through 4 are backspaced one record via the following. If any errors occur, the program will terminate abnormally.

```
      DO 1 lun = 1,4
1     BACKSPACE lun
```

The following also attempts to backspace units 1 through 4; however, an error reports the error code and the unit number. Note that the unit number need not be the first specification when it is preceded by the "UNIT=" symbol.

```
      DO 2 lun = 1,4
2     BACKSPACE(IOSTAT=j,UNIT=lun,ERR=15)
      WRITE(*,*) "Backspace successful"
      STOP
15    WRITE(*,*) "Backspace error ",j," on unit ",lun
```

See also:

The discussion of the OPEN statement describes the FORTRAN file system in general, including the different file types.

5.3 BLOCK DATA Statement

The BLOCK DATA statement introduces a BLOCKDATA subprogram.

Syntax:

```
BLOCK DATA [bdname]
```

Where:

`bdname` is a symbolic name identifying the block data subprogram

Description:

The block data subprogram is a nonexecutable specification subprogram that can be used to enter initial values for variables and array elements in common blocks. Both named and blank common may be initialized. A program can have more than one block data subprogram. Only one block data subprogram can be unnamed.

The BLOCK DATA statement must appear as the first statement of the block data subprogram. The name used for the block data subprogram must not be the same as any local variables in the subprogram. The name must not be the same as any other program unit or entry name in the program.

Block data subprograms can contain IMPLICIT, PARAMETER, DIMENSION, TYPE, COMMON, SAVE, EQUIVALENCE, or DATA statements. A block data subprogram ends with an END statement. Data can be entered into more than one common block in a block data program. All variables having storage in the named common must be specified even if they are not all initially defined.

5.4 BYTE Statement

The BYTE statement defines some user defined entity to be of type byte.

Syntax:

```
BYTE name[,name]...
```

Where:

`name` has one of the forms:

```
var [/ c /]
array [(d[,d]...)]
[/ clist /]
```

`var` is a variable, function name, symbolic constant, or dummy procedure

`array` is an array name

`d` specifies the bounds of a dimension.

`clist` is a list of constants or symbolic constants specifying the initial values.

Each item in the list can take the form:

```
c
r*c
```

`c` is a constant or symbolic constant.

`r` is a repeat count that is an unsigned nonzero integer constant or the symbolic name of such a constant.

Notes:

The BYTE statement performs the same action as the INTEGER*1 statement.

Description:

The BYTE statement is used to define a variable, array, symbolic constant, function name, or dummy procedure name as type byte. The symbol may already have been defined in another declaration statement. A byte entity behaves like an integer entity; however, its range is 0 to 255.

See also:

See the discussion of the DIMENSION statement for a description of how dimension bounds are defined.

See the discussion of the DATA statement for a description of how initial values are defined.

5.5 CALL Statement

The CALL statement transfers control to a subroutine subprogram.

Syntax:

```
CALL sub[([a[,a]...])]
```

Where:

`sub` is the name of a subroutine or dummy procedure.

`a` is an actual argument that can be one of the following:

An expression
An array name
An intrinsic function name
An external procedure name
An alternate return specifier of the form `*s`

`s` is the statement label of an executable statement that appears in the same program unit as the CALL statement.

Description:

The CALL statement can contain actual arguments and statement labels which must correspond in order, number, and type to those in the subroutine definition.

An actual argument in a CALL statement can be a dummy argument name that appears in the dummy argument list of the subprogram containing the CALL statement. An asterisk dummy argument cannot be used as an actual argument.

5.6 CLOSE Statement

The CLOSE statement disconnects a file from a specified unit.

Syntax:

```
CLOSE unit
```

or

```
CLOSE( [ UNIT= ] unit [,STATUS=clssta ] [,IOSTAT=status ] [,ERR=err ] )
```

Where:

`unit` is an integer expression

`clssta` is a character expression

`status` is an integer lvalue

`err` the label of an executable statement

Notes:

The simplest form of the CLOSE statement consists of a single unit specification not enclosed in parentheses. With this form no additional parameters can be specified. If the parenthetical form of CLOSE is used, then the "UNIT=" specification is optional; however, when omitted the unit specification must be the first specification. Other than the above, the order of the parameters within the parenthetical version of the CLOSE statement is free.

Description:

The CLOSE statement disconnects a file from a specified unit and specifies whether the file connected to that unit is to be kept or released. A CLOSE statement can appear in any program unit in the program; it need not appear in the same program unit as the OPEN statement specifying the same unit. A CLOSE statement that references a unit that does not have a file connected to it has no effect.

After a unit has been disconnected by a CLOSE statement, it can be connected again within the same program to the same file or to a different file. A file connected to a unit specified in a CLOSE statement can be connected again to the same or to another unit, provided the file still exists.

The `clssta` variable is a character expression that determines the disposition of the file associated with the specified unit. Valid values are as follows:

'KEEP' The file is kept after execution of the CLOSE statement.

'DELETE' The file is unloaded after execution of the CLOSE statement.

The default, if `clssta` is not specified is STATUS='DELETE' if the file status was 'SCRATCH' when it was opened; otherwise, the default is STATUS='KEEP'.

If an error occurs as a result of the close, and if neither `err` nor `status` are supplied then execution will terminate with an error code set. If either or both are supplied, the execution continues despite any errors.

If supplied, the `status` lvalue receives the runtime error code for the error condition encountered or a zero, if no error occurred.

If `err` is supplied, then execution will branch to the statement labeled by it if an error occurs.

Example:

```
CLOSE (2,ERR=25,STATUS='DELETE')
```

When this statement is executed, the file connected to unit 2 will be closed and disconnected. If an error occurs, execution will branch to statement 25.

See also:

The discussion of the OPEN statement describes the FORTRAN file system in general, including the different file types.

5.7 CHARACTER Statement

The CHARACTER statement defines some user defined entity to be of type character.

Syntax:

```
CHARACTER[*len][,]name[,name]...
```

Where:

`name` has one of the forms

```
var [*len] [/ c /]

array [(d[,d]...)] [*len] [/ clist /]
```

`len` specifies the length and can be an unsigned nonzero integer constant; an integer constant expression, enclosed in parentheses, with a positive value; or an asterisk enclosed in parentheses.

`var` is a variable, function name, symbolic constant, or dummy procedure

`array` is an array name

`d` specifies the bounds of a dimension.

`clist` is a list of constants or symbolic constants specifying the initial values. Each item in the list can take the form:

```
c
r*c
```

`c` is a constant or symbolic constant.

`r` is a repeat count that is an unsigned nonzero integer constant or the symbolic name of such a constant.

The CHARACTER statement is used to define a variable, array, symbolic constant, function name, or dummy procedure name as type character. A length specification immediately following the word CHARACTER applies to each entity not having its own length specification. A length specification immediately following an entity is the length specification only for that entity. Note that for an array, the length specified is for each array element. If a length is not specified for an entity the length is 1.

If a dummy argument has the length (*) specified, the dummy argument assumes the length of the associated actual argument for each reference to the subroutine or function. If the associated actual argument is an array name, the length assumed by the dummy argument is the length of each array element in the associated actual argument.

If a symbolic constant of type character has the length (*) specified, the constant has the length of its corresponding constant expression in a PARAMETER statement.

See also:

See the discussion of the DIMENSION statement for a description of how dimension bounds are defined.

See the discussion of the DATA statement for a description of how initial values are defined.

5.8 COMMON Statement

The COMMON statement provides a means whereby different program units can share the same information.

Syntax:

```
COMMON [[/[cb]/]nlist[[,]/[cb]/nlist] ...
```

Where:

`cb` is a symbol name

`nlist` is a list of entities separated by commas which can take the following forms:

```
var [/ c /]
array [(d[,d]...)] [/ clist /]
```

`var` is a variable.

`array` is an array name.

`d` specifies the bounds of a dimension.

`clist` is a list of constants or symbolic constants specifying the initial values. Each item in the list can take the form:

```
c
r*c
```

`c` is a constant or symbolic constant.

`r` is a repeat count that is an unsigned nonzero integer constant or the symbolic name of such a constant.

Description:

The COMMON statement provides a means of associating entities in different program units. The use of common blocks enables different program units to define and reference the same data without using arguments, and to share storage units. Within one program unit, any entity in a common block is known by a specific name. Within another program unit, the same data can be known by a different symbolic name that is valid only within the scope of that program unit.

The `cb` parameter is a common block name identifying a named common block containing the entities in `nlist`. If the name is omitted, the `nlist` entities are in blank common. Other than having no name, blank common has no characteristics different from those of labeled common blocks.

A single variable name or array name can appear only once in any COMMON statement within the program unit. Function or entry names cannot be included in common blocks. In a subprogram, names of dummy arguments cannot be included in common blocks.

If the common block name is omitted, the common block is blank common. When the first specification in the COMMON statement is for blank common, the slashes can also be omitted. If a common block name is specified, the common block is a named common block. Within a program unit, declarations of common blocks are cumulative. The `nlist` following each successive appearance of the common block name (or no name for blank common) adds more entities to the common block and is treated as a continuation of the specification. Variables and arrays are stored in the order in which they appear in the specification. Alignment bytes are often inserted between members of different types within common blocks; therefore, great care must be taken when the types of variables within COMMON blocks vary from subprogram to subprogram.

The actual size of any common block is the number of storage words required for the entities in the common block, plus any extensions associated with the common block by EQUIVALENCE statements. Extensions can only be made by adding storage words at the end of the common block.

Entities in common blocks can be initially defined by a DATA statement in a block data subprogram, or by a DATA statement in any program unit. If data is assigned to a common block in a subprogram other than a BLOCK DATA subprogram, then that data may not be defined until that subprogram has been executed. If data is assigned to the same storage location in a COMMON block in multiple non BLOCK DATA subprograms, then the content of that location may vary as the different subprograms are initially executed. The initialization of COMMON variables in non BLOCK DATA subprograms is highly discouraged.

See also:

See the discussion of the DIMENSION statement for a description of how dimension bounds are defined.

See the discussion of the DATA statement for a description of how initial values are defined.

See the description of the EQUIVALENCE statement for a discussion of how COMMON areas can be extended.

5.9 COMPLEX Statement

The COMPLEX statement defines some user defined entity to be of type complex or double complex.

Syntax:

```
COMPLEX[*len][,]name[,name]...
```

Where:

`name` has one of the forms

```
var [*len] [/ c /]

array [(d[,d]...)] [*len] [/ clist /]
```

`len` specifies the complex subtype and can be an unsigned nonzero integer constant whose value is 8 or 16.

`var` is a variable, function name, symbolic constant, or dummy procedure

`array` is an array name

`d` specifies the bounds of a dimension.

`clist` is a list of constants or symbolic constants specifying the initial values. Each item in the list can take the form:

```
c
r*c
```

`c` is a constant or symbolic constant.

`r` is a repeat count that is an unsigned nonzero integer constant or the symbolic name of such a constant.

Description:

The COMPLEX statement is used to define a variable, array, symbolic constant, function name, or dummy procedure name as type complex or double complex. A length specification immediately following the word COMPLEX applies to each entity not having its own length specification. A length specification immediately following an entity is the length specification only for that entity. If the length specification for a given entity is 8 or if it is omitted, then the type defined is complex. If the value is 16, then the type is double complex.

See also:

See the discussion of the DIMENSION statement for a description of how dimension bounds are defined.

See the discussion of the DATA statement for a description of how initial values are defined.

5.10 CONTINUE Statement

The CONTINUE statement performs no operation.

Syntax:

```
CONTINUE
```

Description:

The CONTINUE statement performs no operation. It is an executable statement that can be placed anywhere in the executable statement portion of a source program without affecting the sequence of execution. The CONTINUE statement is most frequently used as the last statement of a DO loop. It can provide loop termination when a GOTO or IF would normally be the last statement of the loop.

5.11 DATA Statement

The DATA statement provides initial values for storage elements.

Syntax:

```
DATA nlist/clist/ [[,]nlist/clist/]...
```

Where:

`nlist` is a list of names to be initially defined. Each name in the list can take the form:

```
var
```

```
array
element
substr
dolist
```

`var` is a variable name.

`array` is an array name.

`element` is a subscripted array element name. Subscript expressions must consist of integer constants and active control variables from DO lists.

`substr` is a substring of a character variable or array element.

`dolist` is an implied-DO list of the form:

```
(dlist, i = init, term [,incr])
```

`dlist` is a list of array element names and implied-DO lists.

`i` is an integer variable called the implied-DO variable.

`init` is an integer constant, symbolic constant, or expression specifying the initial value, as for DO loops.

`term` is an integer constant, symbolic constant, or expression specifying the terminal value, as for DO loops.

`incr` is an integer constant, symbolic constant, or expression specifying the increment, as for DO loops.

`clist` is a list of constants or symbolic constants specifying the initial values. Each item in the list can take the form:

```
c
r*c
```

`c` is a constant or symbolic constant.

`r` is a repeat count that is an unsigned nonzero integer constant or the symbolic name of such a constant.

Description:

The DATA statement is used to provide initial values for variables, arrays, array elements, and substrings. The DATA statement is nonexecutable and can appear anywhere in a program unit after any statements explicitly typing the elements or dimensioning them. Usually, DATA statements are placed after the specification statements but before the statement function definitions and executable statements.

Example:

The following shows a simple example of a DATA statement.

```
INTEGER K(6)
DATA JR/4/
DATA AT/5.0/,AQ/7.5/
DATA NRX,SRX/17.0,5.2/
```

```
DATA K/1,2,3,3,2,1/
```

The variables JR, AT, AQ, and SRX are initially defined with the values 4, 5.0, 7.5, and 5.2, respectively. Variable NRX is initially defined with the value 17, after type conversion of the real 17.0 to the integer 17. Array K with 6 elements is initially defined with a value for each array element.

This second example shows the use of a repeat count.

```
REAL R(10,10)
DATA R/50*5.0,50*75.0/
```

The array R is initially defined with the first 50 elements set to the value 5.0 and the remaining 50 elements set to the value 75.0.

Entities that are initially defined by the DATA statement are defined when the program begins execution. Entities that are not initially defined, and not associated with an initially defined entity, are undefined at the beginning of execution of the program.

A variable, array element, or substring must not be initially defined more than once in the program. If two entities are associated, only one can be initially defined by a DATA statement.

Names of dummy arguments and functions cannot be initially defined. Entities in a common block can be initially defined within a block data subprogram, or within any program unit in which the common block appears; however, the DATA initialization of COMMON variables with DATA statements in non BLOCK DATA subprograms is discouraged.

Within the DATA statement, each list `nlist` must have the same number of items as the corresponding list `clist`. A one-to-one correspondence exists between the items specified by `nlist` and the constants specified by `clist`. The first item of `nlist` corresponds to the first constant of `clist`, the second item to the second constant, and so forth.

If an unsubscripted array name appears as an item in `nlist`, a constant in `clist` must be specified for each element of the array. The values of the constants are assigned according to the storage order of the array.

For arithmetic data types, the constant is converted to the type of the associated `nlist` item if the types differ. For all other types, the data type of each constant in `clist` must be compatible with the data type of the `nlist` item.

Each subscript expression used in an array element name in `nlist` must be an integer constant expression, except that implied-DO variables can be used if the array element name is in `dlist`. Each substring expression used for an item in `nlist` must be an integer constant expression.

5.11.1 Implied DO List Use in DATA Statement

An implied DO list can be used as an item in `nlist` as the following example shows.

```
REAL X(5,5)
DATA ((X(J,I),I=1,J),J=1,5)/15*1.0/
```

The elements of array X are initially defined with the DATA statement. Elements in the lower diagonal part of the matrix are set to the value 1.0. The elements initialized are (1,1), (2,1), (2,2), (3,1), (3,2), (3,3), (4,1), (4,2), (4,3), (4,4), (5,1), (5,2), (5,3), (5,4), and (5,5) as shown below:

Array X

	I=1	I=2	I=3	I=4	I=5
J=1	1				
J=2	1	1			

J=3	1	1	1		
J=4	1	1	1	1	
J=5	1	1	1	1	1

As a second example consider the following.

```
PARAMETER (PI=3.14159)
REAL Y(5,5)
DATA ((Y(J+1,I),J=I+1,4),I=1,3)/6*PI/
```

The elements of array Y initialized to 3.14159 are (3,1), (4,1), (4,2), (5,1), (5,2), and (5,3) as shown below:

Array Y

	I=1	I=2	I=3	I=4	I=5
J=1					
J=2					
J=3	3.14159				
J=4	3.14159	3.14159			
J=5	3.14159	3.14159	3.14159		

An iteration count and the values of the implied DO variable are established from init, term, and the optional incr just as for DO loops, except that the iteration count must be positive. When the implied DO list appears in a DATA statement, the list items in `dlist` are specified once for each iteration of the implied DO list, with appropriate substitution of values for each occurrence of the implied DO variable `i`.

The appearance of a name as an implied DO variable in a DATA statement does not affect the value or definition status of a variable with the same name in the program unit. An implied DO variable has the scope of the implied DO list only.

Each subscript expression used in dlist must be an integer constant expression, except that any expression can contain an implied DO variable if the subscript expression is within the corresponding implied DO list.

5.11.2 Character Data Initialization

For initialization by DATA statement, a character item in `nlist` must correspond to a character constant in `clist`. The initial value is assigned according to the following rules:

(1) If the length of the character item in `nlist` is greater than the length of the corresponding character constant, the additional character positions in the item are initially defined as blanks;

(2) if the length of the character item in `nlist` is less than the length of the corresponding character constant, the additional characters in the constant are ignored.

Note that initial definition of a character item causes definition of all character positions. Each character constant initially defines exactly one character variable, array element, or substring.

The following is a character data initialization example.

```
CHARACTER STR1*6,STR2*3
DATA STR1/'ABCDE'/
DATA STR2/'FKGJK'/
```

The character variables `STR1` and `STR2` are initially defined. Variable `STR1` is set to 'ABCDE ', with the sixth character position defined as blank. Variable `STR2` is set to 'FKG', with the fourth and fifth characters of the constant ignored.

5.12 DECODE Statement

The DECODE statement is the extended internal file input statement.

Syntax:

```
DECODE(c, fn, u) iolist
```

Where:

`c` is an unsigned integer constant or variable having a value greater than zero

`fn` is a statement label of a FORMAT statement, or a character expression whose value is a format specification

`u` is a variable, array element or array name

`iolist` is a list of character variables, arrays, or array elements

Description:

The DECODE statement is the extended internal file input statement. It performs a memory-to-memory transfer of data similar to an internal file formatted READ. Starting at location `u`, display code characters in memory are converted according to the specified format and stored in the variables specified in `iolist`. The parameter `c` specifies the number of characters per record in the internal input file.

When DECODE processes an illegal character for a given conversion specification a fatal error results. DECODE can be used to pack the partial contents of two words into one.

See also:

The READ statement for a discussion of internal files and of formatted input processing.

5.13 The DIMENSION Statement

The DIMENSION statement defines arrays and their bounds.

Syntax:

```
DIMENSION array(d[,d]...) [ / clist / ] ...
```

Where:

`array` is an array name.

`d` specifies the bounds of a dimension in one of the forms:

```
upper
lower:upper
```

`upper` is an expression in which all constants, symbolic constants, and variables are type integer or an asterisk (*)

`lower` is an expression in which all constants, and symbolic constants, and variables are of type integer.

`clist` is a list of constants or symbolic constants specifying the initial values. Each item in the list can take the form:

```
c
r*c
```

`c` is a constant or symbolic constant.

`r` is a repeat count that is an unsigned nonzero integer constant or the symbolic name of such a constant.

Description:

The DIMENSION statement defines symbolic names as array names and specifies the bounds of each array. More than one array can be declared in a single DIMENSION statement. Dummy argument arrays specified within a procedure subprogram can have adjustable dimension specifications. A further explanation of adjustable dimension specifications appears below.

The parameter `upper` is the upper bound of the dimension and `lower` is the lower bound of the dimension. If only the upper bound is specified, the value of the lower bound is one. The earlier discussion of arrays in Chapter 2 presented the exact form of a dimension bound expression.

The following shows the relation between the DIMENSION statement and a type declaration statement.

```
REAL NIL
DIMENSION NIL(6,2,2)
```

is equivalent to

```
REAL NIL(6,2,2)
```

and both are equivalent to

```
DIMENSION NIL(6,2,2)
REAL NIL
```

In the following example

```
CHARACTER*8 XR
DIMENSION XR(0:4)
```

the array `XR` contains 5 character elements, with each element having a length of 8 characters. A reference to the third and fourth characters of the second element would be `XR(1)(3:4).`

The following example shows an expression in a DIMENSION bound specification:

```
PARAMETER(N=100)
DIMENSION ARR(1:N*3,0:5)
```

Array `ARR` is a two-dimensional array that contains 1800 elements. The value of `N` in the dimension bound expression for the first dimension is defined in the PARAMETER statement; thus the first bound goes from 1 to 300, while the second goes from 0 to 5.

Within the same program unit, only one definition of an array is permitted. Note that dimension information can be specified in COMMON statements and type statements. The dimension information defines the array dimensions and the bounds for each dimension.

5.13.1 Adjustable Dimensions

Adjustable dimensions enable creation of a more general subprogram that can accept varying sizes of array arguments. For example, a subroutine with a fixed array can be declared as:

```
SUBROUTINE SUM(A)
DIMENSION A(10)
```

The maximum array size subroutine `SUM` can accept is 10 elements.

If the same subroutine is to accept an array of any size, it can be written as:

```
SUBROUTINE SUM(A, N)
DIMENSION A(N)
```

Value `N` is passed as an actual argument.

Adjustable dimensions can be passed through common variables. For example,

```
SUBROUTINE SUB(A)
COMMON/B/M,N
DIMENSION A(M,N)
```

Dimension of array `A`, in subroutine `SUB`, is specified by the values `M` and `N` passed through the common block `B`.

Character strings and arrays can also be adjustable. For example,

```
SUBROUTINE MESSAG(X)
CHARACTER X*(*)
PRINT *, X
```

The subroutine declares `X` with a length of `(*)` to accept strings of varying size. Note that the length of the string is not passed explicitly as an actual argument.

Another form of adjustable dimension is the assumed-size array. In this case, the upper bound of the last dimension of the array is specified by an asterisk. The value of the dimension is not passed as an argument, but is determined by the number of elements stored in the array. If an array is dimensioned *, the array in the calling program must be large enough to contain all the elements stored into it in the subprogram. See the following for example.

```
      SUBROUTINE CAT (A,M,N,B,C)
      REAL A(M), B(N), C(*)
      DO 10 I=1, M
10    C(I)=A(I)
      DO 20 I=1, N
20    C(I+M)=B(I)
      RETURN
      END
```

Subroutine `CAT` places the contents of array `A` followed by the contents of array `B` into array `C`. The dimension of `C` in the calling program must be greater than or equal to `M+N`. Use of the asterisk form of the adjustable dimension to prevent subscript checking for the array is preferable to the common practice of declaring arrays to have dimension 1.

See also:

The description of arrays is in chapter 2. This description covers the properties of arrays, the storage of arrays, and array references.

5.14 DO Statement

The DO statement repeats a group of statements.

Syntax:

```
       DO sl [,] v=e1,e2[,e3]
       [block]
    sl exec
or
       DO v=e1,e2,[,e3]
       [block]
       ENDDO
```

Where:

`sl` is the label of an executable statement.

`v` is an integer, real, or double precision variable.

`e1` is an arithmetic expression.

`e2` is an arithmetic expression.

`e3` is an arithmetic expression.

`exec` is an executable statement.

`block` is a block of executable statements.

Description:

The DO statement is used to specify a loop, called a DO loop, that repeats a group of statements. In the first form `sl` is the label of an executable statement called the terminal statement of the DO loop. In the second form the ENDDO behaves like a CONTINUE statement and as such may be treated as the terminal statement. The parameter `v` is the control variable for the loop. It may be any noncomplex arithmetic type. The parameters `e1`, `e2`, and `e3` are called indexing parameters; they can be integer, real, or double precision. The types of these expressions are always converted to the type of the control variable prior to their use. The parameter `e1` is the initial parameter, `e2` is the terminal parameter, and `e3` is an optional increment parameter — if omitted its default is 1.

The terminal statement of a DO loop is an executable statement that must physically follow and reside in the same program unit as its associated DO statement. The terminal statement must not be an unconditional GO TO, assigned GO TO, arithmetic IF, or block IF, ELSE IF, ELSE, END IF, RETURN, STOP, END, or DO statement. If the terminal statement is a logical IF statement, it can contain any statement except a DO, block IF, ELSE IF, ELSE, END IF, END, or another logical IF.

The range of a DO loop consists of all the executable statements following the DO statement up to and including the terminal statement. The DO statement execution is as follows:

1. The expressions `e1`, `e2`, and `e3` are evaluated and, if necessary, converted to the type of the control variable `v`.

2. Control variable `v` is assigned the value of `e1`.

3. The iteration count is established; this value is determined by the following expression:

```
MAX(INT((m2-m1+m3)/m3), mtc)
```

Where:

`m1, m2, m3` are the values of the expressions `e1`, `e2`, and `e3`, respectively, after conversion to the type of `v`.

`mtc` is the minimum trip count; `mtc` normally has a value of 0. It may be set to 1 if a very old dialect of FORTRAN is being processed.

4. If the iteration count is not 0, the range of the DO loop is executed. If the iteration count is 0, execution continues with the statement following the terminal statement of the DO loop; the control variable retains its most recent value, which is typically `m2+m3`.

5. Control variable `v` is incremented by the value of `e3`.

6. The iteration count is decremented by one.

Steps 4 through 6 are repeated until the iteration count has a value of 0.

If a DO loop appears within an IF-block, the range of the DO loop must be entirely contained within the IF-block. If a block IF statement appears within the range of a DO loop, the corresponding END IF statement must also appear within the range of that DO loop.

A DO loop can be active, inactive, or nested. Each is discussed below.

5.14.1. Active and Inactive DO Loops

Initially, a DO loop is inactive. A DO loop becomes active only when its DO statement is executed. Once active, a loop becomes inactive when any of the following occurs:

1. The iteration count is determined to be 0.

2. A RETURN, STOP, or END statement is executed within the program unit containing the loop.

3. The control variable becomes undefined or is redefined (by a process other than loop incrementation).

4. It is in the range of another loop that becomes inactive.

5. It is in the range of another loop whose DO statement is executed. Transfer of control out of the range of a DO loop does not deactivate the loop. When such a transfer occurs, the control variable retains its most recent value in the loop. Control can be returned to the range of the loop provided that the control variable is not redefined outside the range or the program unit containing the loop has not been exited by a RETURN, STOP, or END statement. The loop becomes inactive once the control variable is redefined and should not be reentered except through its DO statement.

If a DO loop executes zero times, the control variable value equals `m1`. Otherwise, if the index variable iterates through the terminal parameter value, the control value is the most recent value of the control variable plus the increment parameter value.

If a DO loop becomes inactive but has not executed to completion (iteration count does not equal 0), its control variable retains its most recent value unless it has become undefined.

Transfer into the range of an inactive DO loop from outside the range is strongly discouraged.

5.14.2 Nested DO Loops

When a DO loop entirely contains another DO loop, the grouping is called a DO nest. The range of a DO statement can include other DO statements providing the range of each inner DO is entirely within the range of the containing DO statement.

The last statement of an inner DO loop must be either the same as the last statement of the outer DO loop or must occur before it. A terminal statement that is shared by more than one DO loop can be referenced in a GO TO or IF statement from within the range of the innermost loop only. If the terminal statement is referenced from any loop other than the innermost loop, results are undefined.

5.15 DOUBLE COMPLEX Statement

The DOUBLE COMPLEX statement defines some user defined entity to be of type double complex.

Syntax:

```
DOUBLE COMPLEX name[,name]...
```

Where:

`name` has one of the forms:

```
var [/ c /]
array [(d[,d]...)] [/ clist /]
```

`var` is a variable, function name, symbolic constant, or dummy procedure.

`array` is an array name.

`d` specifies the bounds of a dimension.

`clist` is a list of constants or symbolic constants specifying the initial values. Each item in the list can take the form:

```
c
r*c
```

`c` is a constant or symbolic constant.

`r` is a repeat count that is an unsigned nonzero integer constant or the symbolic name of such a constant.

Notes:

The DOUBLE COMPLEX statement performs the same action as the COMPLEX*16 statement.

Description:

The DOUBLE COMPLEX statement is used to define a variable, array, symbolic constant, function name, or dummy procedure name as type double complex. The symbol may already have been defined in another declaration statement.

See also:

See the discussion of the DIMENSION statement for a description of how dimension bounds are defined.

See the discussion of the DATA statement for a description of how initial values are defined.

5.16 DOUBLE PRECISION Statement

The DOUBLE PRECISION statement defines some user defined entity to be of type double precision.

Syntax:

```
DOUBLE PRECISION name[,name]...
```

Where:

`name` has one of the forms:

```
var [/ c /]
array [(d[,d]...)] [/ clist /]
```

`var` is a variable, function name, symbolic constant, or dummy procedure.

`array` is an array name.

`d` specifies the bounds of a dimension.

`clist` is a list of constants or symbolic constants specifying the initial values. Each item in the list can take the form:

```
c
r*c
```

`c` is a constant or symbolic constant.

`r` is a repeat count that is an unsigned nonzero integer constant or the symbolic name of such a constant.

Notes:

The DOUBLE PRECISION statement performs the same action as the REAL*8 statement.

Description:

The DOUBLE PRECISION statement is used to define a variable, array, symbolic constant, function name, or dummy procedure name as type double precision. The symbol may already have been defined in another declaration statement.

See also:

See the discussion of the DIMENSION statement for a description of how dimension bounds are defined.

See the discussion of the DATA statement for a description of how initial values are defined.

5.17 DO WHILE Statement

The DO WHILE statement repeats a group of statements while a condition is true.

Syntax:

```
      DO sl WHILE(exp)
      [block]
   sl exec
```

or

```
      DO WHILE(exp)
      [block]
      END DO
```

Where:

`sl` is the label of an executable statement.

`exp` is a logical expression.

`exec` is an executable statement.

`block` is a block of executable statements.

Description:

The DO WHILE statement is similar to the DO statement. The difference is that DO WHILE executes for as long as a logical expression is true; while the simple DO executes for a fixed number of iterations.

The DO WHILE statement tests the logical expression at the beginning of each execution of the loop, including the first. If the value of the expression is true, the statements in the body of the loop are executed; if the expression is false, control transfers to the statement following the loop.

If no label appears in a DO WHILE statement, the DO WHILE loop must be terminated with an END DO statement.

Example:

The loop below:

```
      DO WHILE (I .GT. J)
            A(I,J) =1 0
            I = I - 1
      END DO
```

is equivalent to

```
      DO 10 WHILE (I .GT. J)
            A(I,J) = 10
            I = I - 1
   10 CONTINUE
```

5.18 ELSE Statement

The ELSE statement specifies an optional statement block to be executed within a block IF statement.

Syntax:

```
            ELSE [ IF (e1) THEN ]
            [ block ]
            END IF
```

Where:

`e1` is a logical expression.

`block` is a block of executable statements.

Description:

If no preceding logical expression with a block IF statement has been true, and if the expression `e1` is true or if `e1` is omitted, then the executable statements in the block are executed and control is transferred to the first statement immediately after the END IF statement.

If the `e1` expression is omitted, then it must be the last ELSE within the block IF.

See also:

See the block IF statement for a complete discussion of the various component statements within the block IF construct.

5.19 ENCODE Statement

The ENCODE statement is an internal file output statement.

Syntax:

```
ENCODE (c, fn, u) iolist
```

Where:

`c` is an unsigned integer constant or variable having a value greater than zero.

`fn` is a statement label of a FORMAT statement, or a character expression whose value is a format specification.

`u` is an extended internal file (variable, array element or array name) in which the record is to be encoded.

`iolist` is a list of variables, arrays or array elements to be transmitted to the location specified by `u`.

Description:

ENCODE is similar to an internal file formatted WRITE. Values are transferred to the receiving storage area from the variables specified in `iolist` under the specified format. The parameter `c` specifies the number of characters to be transferred per record. The record length is calculated from `c`. The parameter `fn` describes the format to be used to encode the values. The parameter `u` specifies the area to receive the encoded information.

The first record starts with the leftmost character of the location specified by `u`. The internal file must be large enough to contain the total number of characters transmitted by the ENCODE statement.

If the list and the format specification transmit more than the number of characters specified per record, the excess characters are ignored. If the number of characters transmitted is less than the record length, remaining characters in the record are blank filled.

See also:

See the WRITE statement for a discussion of internal files and of formatted output processing.

5.20 END Statement

The END statement indicates the end of the program unit to the compiler.

Syntax:

```
END
```

Description:

Every program unit must physically terminate with an END statement. The END statement can be labeled. If control flows into or branches to an END statement in a main program, execution terminates. If control flows into or branches to an END statement in a function or subroutine, it is treated as if a RETURN statement has preceded the END statement.

An END statement cannot be continued; it must be completely contained on an initial line. A line following an END statement is considered to be the first line of the next program unit, even if it has a continuation indicator.

5.21 END DO Statement

The END DO statement terminates the range of a DO loop.

Syntax:

```
END DO
```

Description:

The END DO statement terminates the range of a DO or DO WHILE statement. It must be used to terminate such a statement if the statement did not contain a terminal-statement label.

See also:

See the DO and DO WHILE statements for a detailed discussion of loops.

5.22 The ENDFILE Statement

The ENDFILE statement writes an end-of-file to the designated unit.

Syntax:

```
ENDFILE unit
```

or

```
ENDFILE ( [ UNIT= ] unit [ ,IOSTAT= status ] [ ,ERR= err ] )
```

Where:

`unit` is an integer expression.

`status` is an integer lvalue.

`err` the label of an executable statement.

Notes:

The simplest form of the ENDFILE statement consists of a single unit specification not enclosed in parentheses. With this form no additional parameters can be specified. If the parenthetical form of ENDFILE is used, then the "UNIT=" specification is optional; however, when omitted the unit specification must be the first specification. Other than the above, the order of the parameters within the parenthetical version of the ENDFILE statement is free.

Description:

The ENDFILE statement writes end-of-file to the designated unit. Depending upon the file type, this end-of-file may or may not be a physical record. ENDFILE is not permitted on units opened for direct access.

The end-of-file can be later detected by the END= and IOSTAT= parameters on other input statements.

See also:

The discussion of the OPEN statement describes the FORTRAN file system in general, including the different file types.

The discussion of the READ statement describes the END= parameter for detecting the end-of-file condition.

5.23 The END IF Statement

The END IF statement terminates a block IF construct.

Syntax:

```
END IF
```

Description:

The END IF statement is always the last statement of a block IF construct. Whenever a block of statements below the block IF itself or below one of the ELSE statements has been executed, control is transferred to the first statement immediately following the END IF statement. Alternatively, if none of the conditional expressions within the block IF are true and if there is no ELSE with no conditional expression, then control passes directly from the start of the block IF to the statement immediately following the END IF.

See also:

See the block IF statement for a complete discussion of the various component statements within the block IF construct.

5.24 ENTRY Statement

The ENTRY statement defines additional entry points for a procedure established by the SUBROUTINE or FUNCTION statement.

Syntax:

```
ENTRY ep[([d[,d]...])]
```

Where:

`ep` is a symbolic name which identifies the entry point

`d` is a dummy argument that can be one of the following:

a variable name
an array name
a dummy procedure name
an asterisk, only if in a subroutine subprogram

Description:

The ENTRY statement can be used to define additional entry points for a procedure established by the SUBROUTINE or FUNCTION statement. Each procedure subprogram has a primary entry point established by the statement that begins the program unit. Usually, a subroutine call or function reference invokes the procedure at the primary entry point, and the first statement executed is the first executable statement in the program unit. ENTRY statements are used to define other entry points. A procedure that contains one or more ENTRY statements is said to have "multiple entry points".

An ENTRY statement can appear anywhere after the SUBROUTINE or FUNCTION statement in the subprogram. When an entry name is used to reference a procedure, execution begins with the first executable statement that follows the referenced entry point. An entry name is available for reference in any program unit, except in the procedure that contains the entry name. The entry name can appear in an EXTERNAL statement and for a function entry name in a type statement.

Each reference to a procedure must use an actual argument list that corresponds in number of arguments and type of arguments with the dummy argument list in the corresponding SUBROUTINE, FUNCTION, or ENTRY statement. Type agreement is not required for actual arguments that have no type, such as a dummy subroutine name. The dummy arguments for an entry point can therefore be different from the dummy arguments for the primary entry point or another entry point. No dummy argument can be used in an executable statement of a procedure unless it has already appeared in a FUNCTION, SUBROUTINE, or ENTRY statement.

5.25 The EQUIVALENCE Statement

The EQUIVALENCE statement specifies the sharing of storage by two or more entities in a program unit.

Syntax:

```
EQUIVALENCE (nlist) [,(nlist)]...
```

Where:

`nlist` is a list of variable names, array names, array element names, or character substring names separated by commas.

Description:

The EQUIVALENCE statement is used to specify the sharing of storage by two or more entities in a program unit. Equivalencing causes association of the entities that share the storage. Equivalencing associates entities within a program unit, and common blocks associate entities across program units. Equivalencing and common can interact.

The behavior of the EQUIVALENCE statement can best be described via an example.

```
DIMENSION Y(4),B(3,2)
EQUIVALENCE (Y(1),B(3,1)
EQUIVALENCE (X,Y(2))
```

In this example storage is shared so that 6 storage words are needed for `Y`, `B`, and `X`. The associations are:

```
B(1,1)
B(2,1)
B(3,1)    Y(1)
B(1,2)    Y(2)      X
B(2,2)    Y(3)
B(3,2)    Y(4)
```

In any equivalence set — i.e., set of variables which occupy the same storage because of EQUIVALENCE statements — one variable is always the "base" variable; while other variables are the "derived" variables. The base variable is the

variable which actually allocates storage, while the storage locations of the derived variables are computed from the storage location of the base variable. In the above example, `B` is the base variable because it is the "largest" — i.e., occupies the most memory locations — variable in the equivalence set `(B,Y,X)`.

The following example shows an equivalence set of character variables.

```
CHARACTER A*5,C*4,D(2)*2
EQUIVALENCE (A,D(1)),(C,D(2))
```

In this example, storage is shared so that 5 character storage positions are needed for `A`, `C`, and `D`. The associations are:

```
A(1:1)      D(1)(1:1)
A(2:2)      D(1)(2:2)
A(3:3)      D(2)(1:1)    C(1:1)
A(4:4)      D(2)(2:2)    C(2:2)
A(5:5)                   C(3:3)
                         C(4:4)
```

Again `A` is the base variable in the equivalence set `(A,D,C)` because `A` occupies the most memory. Notice, however, that the total memory allocated by the equivalence set is larger than the storage allocated to `A` in the CHARACTER statement, because the fourth element of `C` extends beyond the fifth member of `A`. In such cases, the storage of the base variable is expanded to allow for the extra storage needed for the equivalence set. Thus, though `A` will be treated as though it were a CHARACTER*5 throughout the subprogram, storage is allocated to `A` as though it were a CHARACTER*6.

Different data types are associated with the equivalencing of the first storage word of each entity:

```
REAL TR(4)
COMPLEX TS(2)
EQUIVALENCE (TR,TS)

TR(1)  TS(1)  real part
TR(2)  TS(1)  imaginary part
TR(3)  TS(2)  real part
TR(4)  TS(2)  imaginary part
```

If the equivalenced entities are of different data types, equivalencing does not cause type conversion. If a variable and an array are equivalenced, the variable does not acquire array properties and the array does not lose the properties of an array. There are no restrictions about equivalencing character and noncharacter variables and the lengths of the equivalenced character entities can be different.

Each `nlist` specification must contain at least two names of entities to be equivalenced. In a subprogram, names of dummy arguments cannot appear in the list. Function and entry names cannot be included in the list. Equivalencing specifies that all entities in the list share the same first storage word. For character entities, equivalencing specifies that all entities in the list share the same first character storage position. Equivalencing can indirectly cause the association of other entities: for instance, when an EQUIVALENCE statement interacts with a COMMON statement.

If an array element is included in `nlist`, the number of subscript expressions must match the number of dimensions declared for the array name. If an array name appears in the list, the effect is as if the first element of the array had been included in the list. Any subscript expression must be an integer constant expression. For character entities, any substring expression must be an integer constant expression.

Equivalencing must not reference array elements in such a way that the storage sequence of the array would be altered. The same storage unit cannot be specified as occurring more than once in the storage sequence. For example,

```
REAL FA(3)
EQUIVALENCE (FA(1), B), (FA(3), B)
```

would be illegal.

Also, the normal storage sequence of array elements cannot be interrupted to make consecutive storage words no longer consecutive. For example,

```
REAL BZ(7), CZ(5)
EQUIVALENCE (BZ,  CZ), (BZ(3), CZ(4))
```

would also be illegal.

The interaction of COMMON and EQUIVALENCE statements is restricted in two ways. First, an EQUIVALENCE statement must not attempt the association of two different common blocks in the same program unit. For example,

```
COMMON  /LT/  A,  T
COMMON  /LX/  S,  R
EQUIVALENCE  (T,  S)
```

is not legal. Second, an EQUIVALENCE statement must not cause a common block to be extended by adding storage words before the first storage word of the common block. On the other hand, common storage words are added at the end of the common block. For example,

```
COMMON  /X/  A
REAL  B(5)
EQUIVALENCE  (A, B(4))
```

is not legal, whereas:

```
COMMON  /X/  A
REAL  B(5)
EQUIVALENCE  (A,  B(1))
```

can be used to extend the common block.

5.26 EXTERNAL Statement

The EXTERNAL statement defines a symbolic identifier as referring to an external procedure.

Syntax:

```
EXTERNAL proc[,proc]...
```

Where:

`proc` is a symbolic name

Description:

The EXTERNAL statement is used to identify a symbolic name as representing an external procedure and to permit such a name to be used as an actual argument. The following shows an example of this.

```
SUBROUTINE CHECK
EXTERNAL LOW,HIGH
CALL AR(LOW,VAL)
CALL AR(HIGH,VAL)
END
SUBROUTINE AR(FUNC,VAL)
VAL= FUNC(VAL)
END
REAL FUNCTION LOW(X)
REAL FUNCTION HIGH(X)
```

The names `LOW` and `HIGH` are declared as external. In the first call to subroutine `AR`, `LOW` is passed as an actual argument and the function reference `FUNC(VAL)` is equivalent to `LOW(VAL)`. In the second call to subroutine `AR`, the function reference `FUNC(VAL)` is equivalent to `HIGH(VAL)`.

The EXTERNAL statement also specifies that the procedure is a user-written rather than an intrinsic function. Consider the following example.

```
SUBROUTINE ARGR
EXTERNAL SQRT
Y= SQRT(X)
END
FUNCTION SQRT(XVAL)
```

Since the name `SQRT` is declared external, the function reference `SQRT(X)` references the user-written function `SQRT` rather than the intrinsic function `SQRT`.

If an external procedure name is an actual argument in a program unit, it must appear in an EXTERNAL statement in the program unit. A statement function name must not appear in an EXTERNAL statement.

If an intrinsic function name appears in an EXTERNAL statement in a program unit, the name becomes the name of some external procedure. The intrinsic function with the same name cannot be referenced in the program unit.

5.27 FORMAT Statement

The FORMAT statement is a nonexecutable statement which specifies the formatting of data to be read or written with formatted I/O.

Syntax:

```
sl FORMAT (flist)
```

Where:

`sl` is a statement label

`flist` is a list of items, separated by commas, having the following forms:

```
[r]ed
ned
[r](flist)
```

`ed` is a repeatable edit descriptor.

`ned` is a nonrepeatable edit descriptor.

`r` is a nonzero unsigned integer constant repeat specification.

Format specifications are used in conjunction with formatted input/output statements to produce output or read input that consists of strings of display code characters. On input, data is converted from a specified format to its internal binary representation. On output, data is converted from its internal binary representation to the specified format before it is transmitted. Formats can be specified by:

(1) the statement label of a FORMAT statement.

(2) an integer variable which has been assigned the statement label of a FORMAT statement (see ASSIGN statement).

(3) a character array name or any character expression, except one involving assumed-length character entities.

(4) a noncharacter array name.

The FORMAT statement is used in conjunction with formatted input and output statements. It can appear anywhere in the program after the PROGRAM, FUNCTION or SUBROUTINE statement. An example of a FORMAT statement and its associated READ statement is as follows:

```
    READ (5,10) INK,NAME,AREA
10  FORMAT (10X,I4,I2,F7.2)
```

The format specification consists of edit descriptors in parentheses. Blanks are not significant except in H, quote, and apostrophe descriptors.

Generally each item in an input/output list is associated with a corresponding edit descriptor in a FORMAT statement. The FORMAT statement specifies the external format of the data and the type of conversion to be used. Complex variables always correspond to two edit descriptors. Double precision variables correspond to one edit descriptor when using D, E, F, or G. The D edit descriptor corresponds to exactly one list item. Complex editing requires two (D, E, F, G) descriptors; the two descriptors can be different.

The type of conversion should correspond to the type of the variable in the input/output list. The FORMAT statement specifies the type of conversion for the input data, with no regard to the type of the variable which receives the value when reading is complete. For example, the statements

```
    INTEGER N
    READ (5,10) N
10  FORMAT (F10.2)
```

will assign a floating point number to the variable `N` which could cause unpredictable results if `N` is referenced later as an integer.

5.27.1 Character FORMAT Specifications

A format specification can also be specified as a character expression or as the name of a character variable or array containing a format specification. The form of these format specifications is the same as for FORMAT statements without the keyword FORMAT. Any character information beyond the terminating parenthesis is ignored. The initial left parenthesis can be preceded by blanks. For example

```
CHARACTER FORM*11
DATA FORM/'(I3,2E14.4)'/
READ (2,FMT=FORM,END=50) N,A,B
```

is equivalent to

```
    READ (2,FMT=10,END=50) N,A,B
10  FORMAT (I3,2E14.4)
```

The examples above can also be expressed as:

```
READ 92,FMT='(I3, 2E14.4)',END=50) N,A,B
```

or

```
CHARACTER FORM*(*)
PARAMETER (FORM='(I3,2E14.4)')
READ (2,FMT=FORM,END=50) N,A,B
```

As a second example consider that

```
CHARACTER AR(2)*10
DATA AR/'(10X,2I2,1','0X,F6.2)'/
READ (5,AR) I,J,X
```

is equivalent to

```
    READ (5,100) I,J,X
10  FORMAT (10X,2I2,10X,F6.2)
```

If a format specification is contained in a character array, the specification may cross element boundaries. Only the array name need be specified in the input/output statement; all information up to the closing parenthesis is considered to be part of the format specification.

5.27.2 Noncharacter FORMAT Specifications

Format specifications can be contained in a noncharacter array. The rules for noncharacter format specifications are the same as for character format specifications.

5.27.3 Edit Descriptors

Edit descriptors specify the data conversions to be performed. Below are the repeatable edit descriptors.

Type	Descriptor	Description
Character	A	Character with data-dependent length
	Aw	Character with specified length
Numeric	Dw.d	Double precision floating-point with exponent
	Ew.d	Single precision floating-point with exponent
	Ew.dEe	Single precision floating-point with exponent length
	Fw.d	Single precision floating-point without exponent
	Gw.d	Single precision floating-point with or without exponent
	Gw.dEe	Single precision floating-point with or without explicitly specified exponent length
	Iw	Decimal integer
	Iw.m	Decimal integer with minimum number of digits
Logical	Lw	Logical
Typeless	Ow	Octal integer
	Ow.m	Octal integer with leading zeros and minimum number of digits
	Zw	Hexadecimal integer
	Zw.m	Hexadecimal with leading zeros and minimum number of digits

Below are the nonrepeatable edit descriptors.

Type	Descriptor	Description
Input control	BN	Blanks ignored in numerics
	BZ	Blanks treated as zeros in numerics
Scale factor	kP	Scaling for numeric editing

Hollerith	nH	Output Hollerith string
Character output	"	Output character string
	'	Output character string
Skip spaces	nX	Position forward
Numeric output	SP	Plus signs (+) produced
	SS	Plus signs (+) suppressed
	S	Plus signs (+) suppressed
Tabulation	Tn	Position forward or backward
	TRn	Position forward
	TLn	Position backward
Format control	:	Terminate format control
End of record	/	Indicates end of current input or output record
Output control	$	Suppress end-of-record on output

In both tables, uppercase letters indicate the type of conversion. Lowercase letters indicate user-supplied information that has the following meaning:

`w` Nonzero unsigned integer constant specifying the field width in number of character positions in the external record. This width includes any leading blanks, + or - signs, decimal point, and exponent.

`d` Unsigned integer constant specifying the number of digits to the right of the decimal point within the field. On output all numbers are rounded.

`e` Nonzero unsigned integer constant specifying the number of digits in the exponent.

`m` Unsigned integer constant specifying the minimum number of digits to be output.

`k` Integer constant scale factor.

`n` Positive nonzero decimal integer.

The following paragraphs discuss input/output conversions, field separators, repeatable and nonrepeatable edit descriptors, and repetition factors.

5.27.4 Input/Output Conversions

For the D, E, F, and G input conversions, a decimal point in the input field overrides the decimal point specification of the field descriptor.

Leading blanks are not significant in numeric input conversions; other blanks in numeric conversions are ignored unless BLANK='ZERO' is specified for the file on an OPEN statement or a BZ edit descriptor is in effect. Plus signs can be omitted. An all-blank field is considered to be zero, except for logical input, where an all-blank field is considered to be FALSE.

The output field is right-justified for all output conversions. If the number of characters produced by the conversion is less than the field width, leading blanks are inserted in the output field unless `w.m` is specified, in which case leading zeros are produced as necessary. The number of characters produced by an output conversion must not be greater than the field width. If the field width is exceeded, asterisks are inserted throughout the field.

Complex data items are converted on input/output as two independent floating-point quantities. The format specification uses two conversion elements.

```
      COMPLEX A,B,C,D
      WRITE (6,10) A
10    FORMAT (F7.2,E8.2)
      READ (5,11) B,C,D
11    FORMAT (2E10.3,2(F8.3,F4.1))
```

Different types of data can be read by the same FORMAT specification. For example,

```
10    FORMAT (I5,F15.2)
```

specifies two values: the first of type integer, the second of type real.

Example:

```
      CHARACTER R*4
      READ (5,15) NO,NONE,INK,A,B,R
15    FORMAT (3I5,2F7.2,A4)
```

reads three integer values, two real values, and one character string.

5.27.5 Field Separators

Field separators are used to separate descriptors and groups of descriptors. The format field separators are the slash (/), the comma, the colon, and the dollar sign ($). The slash is also used to specify demarcation of formatted records; while the dollar sign blocks the formation of a new record when the end of the format specification is encountered.

5.27.6 Repeatable and Nonrepeatable Edit Descriptors

The repeatable edit descriptors are used to specify numeric, logical, or character data conversions. The repeatable edit descriptors can be repeated by prefixing the descriptor with a nonzero unsigned integer constant specifying the number of repetitions required. The repeatable edit descriptors are A, D, E, F, G, I, L, O, and Z.

The nonrepeatable edit descriptors are used for numeric input/output control, tabulation control, character output control, format control, end-of-record designation, and scaling for numeric editing. The nonrepeatable edit descriptors cannot be repeated. The nonrepeatable edit descriptors are single quote ('), double quote ("), BN, BZ, colon (:), slash (/), nH, kP, S, SP, SS, Tn, TLn, TRn, and dollar sign ($).

5.27.7 A Descriptor

The A descriptor is used with an input/output list item of type character or noncharacter. The following paragraphs discuss the A descriptor for input/output list items of type character and noncharacter.

The form of the A descriptor for character list items is:

```
      A
```

or

```
      Aw
```

On input, if w is less than the length of the list item, the input quantity is stored left-justified in the item; the remainder of the item is filled with blanks. If w is greater than the length of the item, the rightmost characters are stored and the remaining characters are ignored. If w is omitted, the length of the field is equal to the length of the list item.

This example shows a character list item:

```
      CHARACTER A*9
      READ (5,10) A
10 FORMAT (A7)
```

Input record:

```
EXAMPLE
```

In location A:

```
EXAMPLE^^
```

The second example shows truncation when input is wider than the list item.

```
      CHARACTER B*10
      READ (T,20) B
20  FORMAT (A13)
```

Input record:

```
1         13
SPECIFICATION
```

In location B:

```
1      10
CIFICATION
```

The following example shows that if no length is specified for an A edit descriptor, then the length of the list item is used.

```
CHARACTER NAME*30,PHONE*7
READ (5,'(A,A)') NAME,PHONE
```

On output, if w is less than the length of the list item, the leftmost characters in the item are output. For example, if a variable A, declared CHARACTER A*8, contains

```
SAMPLE^^
```

and A is output with the statement

```
WRITE (6,'(1X,A4)')A
```

then the characters SAMP are output.

If w is greater than the length of the list item, the characters are output right-justified in the field, with blanks on the left. For example, if A in the previous example is output with the statements

```
      WRITE (6,40)A
40  FORMAT (1X,A12)
```

output is as follows:

```
^^^^SAMPLE^^
```

If w is omitted, the length of the character list item determines the length of the output field.

5.27.8 Single and Double Quote Descriptors

Character strings delimited by a pair of single quote (') or double quote (") symbols can be used as alternate forms of the H specification for output. The paired symbols delineate the string. If the string is empty or invalidly delimited, a fatal compilation error occurs and an error message is printed. The single and double quote descriptors are ignored on input.

A single or double quote within a string delimited by the same symbol can be represented by two consecutive occurrences of the symbol. Alternatively, if a single quote or double quote appears within a string, the other symbol can be used as the delimiter.

5.27.9 BN and BZ Blank Interpretation

The nonrepeatable BN and BZ edit descriptors can be used with the repeatable D, E, F, G, and I edit descriptors, on input, to specify the interpretation of blanks (other than leading blanks). In the absence of a BN or BZ descriptor, blanks in input fields are interpreted as zeros or are ignored. Their interpretation depends on the value of the BLANK= parameter in the OPEN statement that is currently in effect for the input/output unit. BLANK='NULL' (blanks ignored) is the default for input. If a BN descriptor is encountered in a format specification, all blank characters in succeeding numeric input fields are ignored; that is, the field is treated as if blanks had been removed, the remaining portion of the field right-justified, and the field padded with leading blanks. A field of all blanks has a value of zero.

If a BZ descriptor is encountered in a format specification, all blank characters in succeeding numeric input fields are interpreted as zeros.

For example, assuming BLANK='NULL', if the statement

```
READ (6,'(I13, BZ, I3, BN, I3)')I,J,K
```

reads the input record

```
1^^2^^3^^
```

`I`, `J`, and `K` are assigned the following values:

```
I = 1
J = 200
K = 3
```

5.27.10 Carriage Control Character

The carriage control character is the first character of a printer output record and is not printed. It appears in other forms of output as data.

The carriage control characters are shown below:

Character	Action
Blank	Space vertically one line, then print
0	Space vertically two lines, then print
1	Eject to the first line of the next page before printing.
+	No advance before printing; allows overprinting
Any other character	Refer to the operating system reference manual.

Carriage control characters are required at the beginning of every record to be printed, including new records introduced by means of a slash. Carriage control characters can be generated by any means.

5.27.11 D Descriptor

The D descriptor specifies conversion between an internal double precision real number and an external floating-point number written with an exponent. This descriptor has the form:

```
Dw.d
```

On input, D editing corresponds to E editing and can be used to input all the same forms as E.

The diagram below illustrates the structure of the input field. It shows the characters allowed to start a subfield.

+	.	+
-		-
digit		D or E
integer	fraction	exponent
subfield	subfield	

On output, type D conversion is used to output double precision values. D conversion corresponds to E conversion except that D replaces E at the beginning of the exponent subfield. For example

```
      DOUBLE PRECISION A,B,C
      A = 111111.11111D0
      B = 222222.22222D0
      C = A + B
      WRITE (2,10) A,B,C
10    FORMAT (3D23.11)
```

produces output of:

```
.11111111111D+06
.22222222222D+06
.33333333333D+06
```

In general, the specification `Dw.d` produces output in the following format:

```
s.a±eee
```

for values where the magnitude of the exponent is greater than or equal to 100, and

```
s.aD±ee
```

for values where the magnitude of the exponent is less than 100

where:

`s`	Minus sign if the number is negative, or blank if the number is positive
`a`	One or more most significant digits
`ee[e]`	Digits in the exponent

5.27.12 E Descriptor

The E descriptor specifies conversion between an internal real or double precision value and an external number written with an exponent. This descriptor has the forms:

```
Ew.d
Ew.dEe
```

On input, the width `w` includes plus or minus signs, digits, decimal point, E, and exponent. If an external decimal point is not provided, `d` acts as a negative power-of-10 scaling factor. The internal representation of the input quantity is:

```
(integer subfield) X 10-d X 10 (exponent subfield)
```

For example, if the specification is E10.8, the input quantity 3267E+05 is converted and stored as:

```
3267 X 10-8 X 10^5 = 3.267.
```

If an external decimal point is provided, it overrides `d; e,` if specified, has no effect on input. An input field consisting entirely of blanks is interpreted as zero.

The following diagram illustrates the structure of the E input field. It shows the characters allowed to start a subfield.

```
+            .            +
-                         -
digit                     E or D

integer      fraction     exponent
subfield     subfield
```

The integer subfield begins with a + or - sign, a digit, or a blank; and it can contain a string of digits. The integer field is terminated by a decimal point, E, +, - or the end of the input field.

The fraction subfield begins with a decimal point and terminates with an E, +, -, or the end of the input field. It can contain a string of digits.

The exponent subfield can begin with E, + or -. When it begins with E, the + is optional between E and the string of digits in the subfield. For example, the following are valid equivalent forms for the exponent 3:

```
E+ 03  E 03   E03    E3     +3
```

Valid subfield combinations are as follows:

```
+1.6327E-04         Integer-fraction-exponent
-32.7216            Integer-fraction
+328+5              Integer-exponent
 629E-1             Fraction-exponent
+136                Integer only
.07628431           Fraction only
E-06                Exponent only
```

If the field length specified by `w` in `Ew.d` is not the same as the length of the field containing the input number, incorrect numbers might be read, converted, and stored.

The example below illustrates a situation where numbers are read incorrectly, converted, and stored; yet there is no immediate indication that an error has occurred.

```
   OPEN (3,BLANK='ZERO')
   READ (3,20) A,B,C
20 FORMAT (E9.3,E7.2,E10.3)
```

On the input record, quantities are in three adjacent fields, columns 1 through 24:

```
+6.47E-01 -2.36 +5.321E+02
       9      5      10
```

would be read as:

```
    9        7 10
+6.47E-01  -2.36+5  .321E+02^^
```

First, `+647E-01` is read, converted and placed in location `A`. The second specification `E7.2` exceeds the width of the second field by two characters. The number `-2.36+5` is read instead of `-2.36.` The specification error (`E7.2` instead of `E5.2`) caused the two extra characters to be read. The number read (`-2.36+5`) is a legitimate input number. Since the second specification incorrectly took two digits from this number, the specification for the third number is now incorrect. The field `.321E+02^^` is read. The OPEN statement specifies that trailing blanks are to be treated as zeros; therefore the number `.321E+0200` is read converted and placed in location `C`. Here again, this is a legitimate input number which is converted and stored, even though it is not the number desired.

Some additional examples of `Ew.d` input specifications are shown below.

Input Field	Specification	Value	Remarks
+143.26E-03	E11.2	0.14326	All subfields present.
327.625	E7.3	327.625	No exponent subfield.
-.0003627+5	E11.7	-36.27	Integer subfield only a minus sign and a plus sign appears instead of E.
-.0003627E5	E11.7	-36.27	Integer subfield left of decimal contains minus sign only.
^^^^^^^	E4.1	0.	All subfields empty.
E+06	E10.6	0.	No integer or fraction subfield: zero stored regardless of exponent field contents.

On output, the width `w` must be sufficient to contain digits, plus or minus signs, decimal point, E, the exponent, and blanks. Generally, `w` must be at least `(d+6)` or `(d+e+4)` for negative numbers, and `w` must be at least `(d+5)` or `(d+e+3)` for positive numbers. Positive numbers need not reserve a space for the sign of the number unless an SP specification is in effect. If the field is not wide enough to contain the output value, asterisks are inserted throughout the field. If the field is longer than the output value, the quantity is right-justified with blanks on the left. If the value being converted is indefinite, an I is printed in the field; if it is out of range, an R is printed.

The `Ew.d` specification produces output in the following formats:

```
s.a...aE ± ee
```

for values where the magnitude of the exponent is less than 100, and

```
s.a...a ± eee
```

for values where the magnitude of the exponent exceeds 100.

Where:

`s` is a minus sign if the number is negative, and omitted if the number is positive.

`a...a` are the most significant digits of the value correctly rounded.

`ee[e]` digits in the exponent.

When the specification `Ew.dEe` is used, the exponent is preceded by `E`, and the number of digits used for the exponent field not counting the letter and sign is determined by `e`. If `e` is specified too small for the value being output, the entire field width as specified by `w` will be filled with asterisks.

5.27.13 End-of-Record Slash

The slash indicates the end of a record anywhere in the FORMAT specification. When a slash is used to separate edit descriptors, a comma is allowed, but not required. Consecutive slashes can be used and need not be separated from other elements by commas. When a slash is the last format specification to be processed, it causes a blank record to be written on output or an input record to be skipped. Normally, the slash indicates the end of a record during output and specifies that further data comes from the next record during input.

5.27.14 F Descriptor

The F descriptor specifies conversion between an internal real or double precision number and an external floating-point number without an exponent. This descriptor has the form:

```
Fw.d
```

On input, the F specification is treated the same as the E specification. Examples of F input are shown below

Input Field	Specification	Value	Remarks
367.2593	F8.4	367.2593	Integer and fraction field.
.62543	F6.5	.62543	No integer subfield.
.62543	F6.2	.62543	Decimal point overrides d of specification.
+144.15E-03	F11.2	.14415	Exponents are allowed in F in-put.
50000	F5.2	500.00	No fraction subfield; input number converted as $50000x10^{-2}$
^^^^	F5.2	0	Blanks in input field interpreted as 0.

On output, the F descriptor outputs a real number without a decimal exponent. The plus sign is suppressed for positive numbers. If the field is too short, all asterisks appear in the output field. If the field is longer than required, the number is right-justified with blanks on the left.

The specification `Fw.d` outputs a number in the following format:

```
sn.n
```

where:

`n` is a field of decimal digits

s is a minus sign if the number is negative, or omitted if the number is positive

The following shows some examples of F output:

Value of A	FORMAT	Output (Before Printing)
+32.694	F6.3	^32.694
+32.694	F10.3	^^^^^32.694
-32.694	F6.3	^
**.32694	F4.3,F6.3	^.327^^.327
32.694	F6.0	^^^^33.

5.27.15 G Descriptor

The G descriptor specifies conversion between an internal real or double precision number and an external floating-point number written either with or without an exponent, depending on the magnitude of the number. This descriptor has the forms:

```
Gw.d
Gw.dEe
```

On input, the G specification is treated the same as the E specification. The rules which apply to the E specification also apply to the G specification. For example,

```
    READ (5,11) A,B,C
11  FORMAT (G13.6,2G12.4)
```

On output, results depend on the size of the floating-point number being edited. For values in the range greater than or equal to .1 and less than 10`d` the number is output under F format. For values outside this range, `Gw.d` output is identical to `Ew.d` and `Gw.dEe` is identical to `Ew.dEe`.

If a number is output under the `Gw.d` specification without an exponent, four spaces are inserted to the right of the field (these are reserved for the exponent field `E±ee`). Therefore, for output under G conversion, `w` must be greater than or equal to `d+6`. The 6 extra spaces are required for sign and decimal point plus four spaces for the exponent field. If the `Gw.dEe` form is used for a number output without an exponent, then `e+2` spaces are inserted to the right of the field. See the examples below:

```
    Y=77.132
    WRITE (7,20) Y
20  FORMAT (G10.3)
```

Output (before printing): `^^77.1^^^^`

```
    EXIT=1214635.1
    WRITE (4,10) EXIT
20  FORMAT (G10.3)
```

Output (before printing): `^^.121E+07`

```
      READ (5,50) SAMPLE
50    FORMAT (E20.5)
      .
      .
      .
      WRITE (6,60) SAMPLE
60    FORMAT (1X,G14.8)
```

Data Read By READ Statement	Output (before printing)	Format Option
.1415926535 E-10	^^.14159265E-10	E conversion
^^^.8979323846	^^.89793238	F conversion
^^^.2643383279.	^^.26433833E+10	E conversion
^^^-693.9937510	^-693.99375	F conversion

5.27.16 H Descriptor

The H descriptor is used to output strings of characters. This descriptor is not associated with a variable in the output list. The H descriptor has the form:

```
nHstring
```

Where:

`n` is the number of characters in the string including blanks.

`string` is a string of characters.

The H descriptor cannot be used on input.

Note that although using quotes to designate a character string precludes the need to count characters, the H descriptor may be more convenient if the string contains quotes.

5.27.17 I Descriptor

The I descriptor specifies integer conversion. This descriptor has the forms:

```
Iw
Iw.m
```

On input, the plus sign can be omitted for positive integers. When a sign appears, it must precede the first digit in the field. The `Iw` and `Iw.m` specifications have the same effect on input. An all blank field is considered to be zero. Decimal points are not permitted. The value is stored in the specified variable. Any character other than a decimal digit, blank, or the leading plus or minus sign in an integer field on input will cause an error. See example of I input below.

```
      OPEN (2,BLANK='NULL')
      READ (2,10) I,J,K,L,M,N
10    FORMAT (I3,I7,I2,I3,I2,I4)
```

Input Record:

```
139^^^^-1518^^7^^^1^4
```

In storage:

```
I    contains 139
J    contains -15
K    contains 18
L    contains 7
M    contains 0
N    contains 14
```

If BLANK='ZERO' had been specified on the OPEN statement, `J` would contain -1500 and `N` would contain 104. Other values would not be affected.

On output, if the integer is positive, the plus sign is suppressed unless an `SP` specification is in effect. Leading zeros are suppressed.

If `Iw.m` is used and the output value occupies fewer than `m` positions, leading zeros are generated to fill up to `m` digits. If `m=0`, a zero value will produce all blanks. If `m=w`, no blanks will occur in the field when the value is positive, and the field will be too short for any negative value. If the field is too short, asterisks occupy the field.

5.27.18 L Descriptor

The L descriptor is used to input or output logical items. This descriptor has the form:

```
Lw
```

On input, if the first nonblank characters in the field are T or .T., the logical value .TRUE. is stored in the corresponding list item, which should be of type logical. If the first nonblank characters are F or .F., the value .FALSE. is stored. If the first nonblank characters are not T, .T., F, or .F., a diagnostic is printed. An all blank field has the value .FALSE.

On output, variables under the L specification should be of type logical. A value of .TRUE. or .FALSE. in memory is output as a right-justified T or F with blanks on the left.

5.27.19 O Descriptor

The O descriptor (letter O) is used to input or output items in octal format. This descriptor has the forms:

```
Ow
Ow.m
```

The form `Ow.m` means the same as `Ow` on input. The octal digits are the digits 0 through 7.

On input blanks are allowed, and a plus or minus sign can precede the first octal digit. Blanks are interpreted as zeros, and an all blank field is interpreted as zero. A decimal point is not allowed.

On output the rightmost `w` digits are output. If `m` is specified, the number is printed with leading zeros so that at least `m` digits are printed.

5.27.20 P Descriptor

The P descriptor is used to change the position of a decimal point of a real number when it is input or output. The P descriptor has the form:

```
kP
```

where k is a signed or unsigned integer constant called the scale factor. Scale factors can either precede D, E, F, and G format specifications or appear independently. Forms are as follows:

```
kPDw.d
kPEw.dEe
kPEw.d
kPFw.d
kPGw.d
kP
```

A scale factor of zero is established when each FORMAT specification is first referenced; it holds for all F, E, G, and D field descriptors until another scale factor is encountered.

Once a scale factor is specified, it holds for all D, E, F, and G descriptors in that FORMAT specification until another scale factor is encountered. To nullify this effect for subsequent D, E, F, and G descriptors a zero scale factor (0P) must be specified. For example,

```
15  FORMAT(2P,E14.3,F10.2,G16.2,0P,4F13.2)
```

The 2P scale factor applies to the E14.3 format specification and also to the F10.2 and G16.2 format specifications. The 0P scale factor restores normal scaling ($10^0 = 1$) for the subsequent specification 4F13.2.

Example:

```
20 FORMAT(3P,5X,E12.6,F10.3, 0PD18.7,-1P,F5.2)
```

E12.6 and F10.3 specifications are scaled by 10^3. The D18.7 specification is not scaled, and the F5.2 specification is scaled by 10^{-1}.

The specification (3P,3I9,F10.2) is the same as the specification (3I9,3PF10.2).

On input, for F, E, D, and G editing, the number is divided by 10^K and stored, provided that the number in the input field does not have an exponent. For example, if the input quantity 314.1592 is read under the specification 2PF8.4, the internal number is $314.1592 \times 10^{-2} = 3.141592$. However, if an exponent is read the scale factor is ignored.

On output, for F editing, the number in the output field is the internal number multiplied by 10^K. In the output representation, the decimal point is fixed; the number is adjusted to the left or right, depending on whether the scale factor is plus or minus. For example, the internal number -3.1415926536 can be represented on output under scaled F specifications as shown below

```
(-1PF13.6)    -.314159
(   F13.6)    -3.141593
( 1PF13.6)    -31.415927
( 3PF13.6)    -3141.592654
```

For E and D editing, the effect of the scale factor kP is to shift the output coefficient left k places and reduce the exponent by k. In addition, the scale factor controls the decimal normalization between the coefficient and the exponent such that: if k is less than or equal to 0, there will be exactly -k leading zeros and d+k significant digits after the decimal point; if k is greater than 0, there will be exactly k significant digits to the left of the decimal point and d-k+1 significant digits to the right of the decimal point. For example, the number -3.1415926536 is represented on output under the indicated Ew.d scaling as shown below.

```
(-3PE20.4)    -.0003E+04
(-1PE20.4)    -.0314E+02
(   E20.4)    -.3142E+01
( 1PE20.4)    -3.1416E+00
( 3PE20.4)    -314.16E-02
```

For G editing, the effect of the scale factor is nullified unless the magnitude of the number to be output is outside the range that permits effective use of F conversion (namely, unless the number N is less than 10^{-1} or greater than or equal to 10^{d}). In these cases, the scale factor has the same effect as described for `Ew.d` and `Dw.d` scaling. For example, the numbers -3.1415926536 and -.00031415926536 are represented on output under the indicated `Gw.d` scaling as shown below.

```
(-3PG20.6)    -3.14159
(-1PG20.6)    -3.14159
(   G20.6)    -3.14159
( 1PG20.6)    -3.14159
( 3PG20.6)    -3.14159
( 5PG20.6)    -3.14159
( 7PG20.6)    -3.14159

(-3PG20.6)    000314E+00
(-1PG20.6)    -.031416E-02
(   G20.6)    -.314159E-03
( 1PG20.6)    -3.141593E-04
( 3PG20.6)    -314.1593E-06
( 5PG20.6)    -31415.93E-08
( 7PG20.6)    -3141593.E-10
```

5.27.21 S, SP, SS Plus Sign Control

The nonrepeatable S, SP and SS edit descriptors can be used on output with the repeatable D, E, F, G, and I edit descriptors to control the printing of plus (+) characters. S, SP and SS have no effect on input.

Normally, FORTRAN does not precede positive numbers by a plus sign on output. If an SP descriptor is encountered in a format specification, all succeeding positive numeric fields will contain the plus sign (w must be of sufficient length to include the sign). If an SS or S descriptor is encountered, the optional plus signs will not appear.

S, SP, and SS have no effect on plus signs preceding exponents, since those signs are always provided.

5.27.22 T, TL, TR Descriptors

The T, TL, and TR descriptors provide for tabulation control. These descriptors have the forms:

```
Tn
TLn
TRn
```

Where:

`n` is a nonzero unsigned decimal integer

When a `Tn` descriptor is encountered in a format specification, input or output control skips right or left to column `n`; the next edit descriptor is then processed.

When a `TLn` descriptor is encountered, control skips backward (left) `n` columns. If `n` is greater than or equal to the current character position, control skips to the first character position.

When a `TRn` descriptor is encountered, control skips forward (right) `n` characters. `TRn` is the same as `nX`.

On input, control can be positioned beyond the end-of-record, but a succeeding descriptor would read only blanks.

With a T, TR, or TL specification, the order of a list need not be the same as that of the input or output record. The same information can be read more than once.

When a T, TR, TL specification causes control to pass over character positions on output, positions not previously filled during record generation are set to blanks; those already filled are left unchanged.

5.27.23 Termination of Format Control

A colon (:) in a format specification terminates format control if there are no more items in the input/output list. The colon has no effect if there are more items in the input/output list. This descriptor is useful in forms where nonlist item edit descriptors follow list item edit descriptors; when the iolist is exhausted, the subsequent edit descriptors are not processed.

5.27.24 X Descriptor

The X descriptor is used to skip character positions in an input line or output line. X is not associated with a variable in the input/output list. The X descriptor has the form:

```
nX
```

Where:

- n is the number of character positions to be skipped from the current character position; n is a nonzero unsigned integer.

The specification nX indicates that transmission of the next character to or from a record is to occur at the position n characters forward from the current position.

When an X specification causes control to pass over character positions on output, positions not previously filled during record generation are set to blanks; however, positions already filled are left unchanged.

5.27.25 Z Descriptor

The Z descriptor is used for hexadecimal conversion. This descriptor has the forms:

```
Zw
Zw.m
```

The form Zw.m is meaningful for output only. Hexadecimal digits include the digits 0 through 9 and the letters A through F. A hexadecimal digit is represented by 4 bits.

On input embedded blanks are interpreted as zero, and an all blank field is equivalent to zero. The string is stored right-justified with zeros on the left.

On output the rightmost w*4 bits are converted to hexadecimal and written. If m is specified, the number is printed with leading zeros so that at least m digits are output. If the number of hexadecimal digits exceeds w, a field of asterisks is written.

5.27.26 Repetition Factors

The repeatable edit descriptors can be repeated by prefixing the descriptor with a nonzero unsigned integer constant specifying the number of repetitions required. For example,

```
10  FORMAT (3I4,2E7.3)
```

is equivalent to:

```
10  FORMAT (I4,I4,I4,E7.3,E7.3)
```

Also,

```
50  FORMAT (4G12.6)
```

is equivalent to:

```
50  FORMAT (G12.6,G12.6,G12.6,G12.6)
```

A group of descriptors can be repeated by enclosing the group in parentheses and prefixing it with the repetition factor. If no integer precedes the left parenthesis, the repetition factor is 1. For example,

```
1   FORMAT (I3,2(E15.3,F6.1,2I4))
```

is equivalent to the following specification if the number of items in the input/output list does not exceed the number of format conversion codes:

```
1   FORMAT(IE,E15.3,F6.1,I4,I4,E15.3,
   + F6.1,I4,I4)
```

A maximum of five levels of parentheses is allowed in addition to the parentheses required by the FORMAT statement.

If there are fewer items in the input/output list than indicated by the format conversions in the FORMAT specification, the excess conversions are ignored.

If the number of items in the input/output list exceeds the number of format conversions when the final right parenthesis in the FORMAT statement is reached, the line formed internally is output. The format control then scans to the left looking for a right parenthesis within the FORMAT statement. If none is found, the scan stops when it reaches the beginning of the format specification. If a right parenthesis is found, however, the scan continues to the left until it reaches the field separator which precedes the left parenthesis pairing the right parenthesis. Output resumes with the format control moving right until either the output list is exhausted or the final right parenthesis of the FORMAT statement is encountered.

If n slashes are indicated, a repetition factor can be used to indicate multiple slashes; n-1 lines are skipped on output.

5.27.27 Execution Time FORMAT Specification

Variable format specifications can be read in as part of the data at execution time and used wherever a normal format can be used. The format can be read in under the A specification and stored in a character array, variable, or array element; or it can be included in a DATA statement. Formats can also be generated by the program at execution time.

If an array or array element is used, its type can be other than character, although character is the preferred type. In either case, the format must consist of a list of descriptors and editing characters enclosed in parentheses, but without the keyword FORMAT and the statement label.

The name of the entity containing the specifications is used in place of the FORMAT statement number in the associated input/output statement. The name specifies the location of the first word of the format information.

5.28 FUNCTION Statement

The FUNCTION statement introduces and specifies the name of the main entry point of a function subprogram.

Syntax:

```
[typ[*len]] FUNCTION fun([d[,d]...])
```

Where:

`typ` is any valid type keyword: BYTE, INTEGER, REAL, DOUBLECOMPLEX, DOUBLEPRECISION, LOGICAL, COMPLEX, CHARACTER

`len` is an integer constant, which specifies additional type information

`fun` is the symbolic name which identifies the function subprogram

`d` is a dummy argument that can be a variable name, array name, or dummy procedure name.

Notes: If there are no dummy arguments, either `fun` or `fun( )` can be used.

Description:

A function subprogram is a procedure that communicates with the referencing program unit through a list of arguments or common blocks. It is usually referred to as an external function. A function subprogram performs a set of calculations when the name appears in an expression in the referencing program unit. A function subprogram begins with a FUNCTION statement and ends with an END statement. Control is returned to the referencing program unit when a RETURN or END is encountered; a RETURN statement of the form RETURN exp is not allowed in a function subprogram.

A function subprogram can contain any statements except PROGRAM, BLOCK DATA, SUBROUTINE, or another FUNCTION statement. A function must not directly or indirectly reference itself.

The symbolic name of a function subprogram, or an associated entry name of the same type, is a variable name in the function. The symbolic name specified in a FUNCTION or ENTRY statement must not appear in any other nonexecutable statement, except a type statement. If the type of a function is specified in a FUNCTION statement, then the function name cannot appear in a type statement. In an executable statement, the symbolic name can appear only as a variable. During execution, this variable becomes defined and can be referenced or redefined. The value of the function is the value of this variable when control returns to the referencing program unit.

The type of the function name must be the same in the referencing program unit and the referenced function subprogram. When type is omitted, the type of the function is determined by the first character of the function name. Implicit typing by the IMPLICIT statement takes effect only when the function name is not explicitly typed. The name cannot have its type explicitly specified more than once.

The symbolic name of a function subprogram must not be the same as any other name, except a variable name or common block name. The function subprogram can have more than one entry point, although alternate returns are prohibited. Multiple entry points are established through the ENTRY statement.

In a function subprogram, the symbolic name of a dummy argument is unique to the program unit and must not appear in an EQUIVALENCE, PARAMETER, SAVE, INTRINSIC, DATA, or COMMON statement, except as a common block name. The dummy arguments are replaced with the actual arguments during a function reference.

Dummy arguments that represent array names must be dimensioned by a DIMENSION or type statement. Adjustable dimensions are permitted in function subprograms.

The type of the function result is the type of the function name. The arguments must agree in order, number, and type with the corresponding dummy arguments. Function subprograms can be referenced in any procedure subprogram.

5.29 GENERIC Statement

The GENERIC statement identifies a name as representing a generic function.

Syntax:

```
GENERIC fun[,fun]...
```

Where:

`fun` is a generic function name.

Notes:

The GENERIC statement performs the same action as the INTRINSIC statement.

Description:

The GENERIC statement is used to identify a name as representing a generic or intrinsic function. This statement enables use of a generic function name as an actual argument. As an example

```
SUBROUTINE DC
GENERIC SQRT
CALL SUBA(X,Y,SQRT)
END
SUBROUTINE SUBA(A,B,FNC)
B=FNC(A)
END
```

The name `SQRT` is declared generic in subroutine `DC` and passed as an argument to subroutine `SUBA`. Within `SUBA`, the reference `FNC(A)` references the generic function `SQRT`. Note that if `SQRT` were declared EXTERNAL, then a user function would be assumed.

The appearance of a name in a GENERIC statement declares the name as a generic, intrinsic function name. If an intrinsic function name is used as an actual argument in a program unit, it must appear in a GENERIC or INTRINSIC statement in the program unit. The following intrinsic function names must not be used as actual arguments:

Type conversion functions BOOL, CHAR, CMPLX, DBLE, FLOAT, ICHAR, IDINT, IFIX, INT, REAL, and SINGL

Lexical relationship functions LGE, LGT, LLE, and LLT

Largest/smallest value functions AMAX0, AMAX1, AMIN0, AMIN1, DMAX1, DMIN1, MAX, MAX0, MAX1, MIN, MIN0, MIN0

Logical and masking functions AND, OR, XOR, NEQV, EQV

The appearance of a generic intrinsic function name in a GENERIC statement does not remove the generic properties of the name.

See Also:

The EXTERNAL statement which allows user subprograms to be used as parameters.

The INTRINSIC statement which is equivalent to this statement.

5.30 GOTO Statement

The GOTO statement transfers control to a labeled statement in a program unit.

Description:

The three types of GOTO statements are the unconditional GOTO, the computed GOTO, and the assigned GOTO. Each is discussed separately below.

See Also:

The ASSIGN statement which assigns labels to variables for use in the assigned GOTO statement.

5.30.1 Unconditional GOTO

The unconditional GOTO branches directly to a statement whose label is specified.

Syntax:

```
GOTO slab
```

Where:

`slab` is the label of an executable statement.

Description:

As an example, in the following

```
10  AAA=B+Z
    B=X+Y
    IF(A-B)20,20,30
20  Z=A
    GOTO 10
30  Z=B
```

control transfers to statement 10 when the GOTO statement executes.

5.30.2 Computed GOTO Statement

The computed GOTO statement transfers control to the statement identified by one of the specified labels.

Syntax:

```
GOTO (slab[,slab]...)[,]exp
```

Where:

`slab` is the label of an executable statement.

`exp` is an arithmetic expression.

Description:

The label selected is determined by the value of the expression. If `exp` has a value of 1, control transfers to the statement identified by the first label in the list; if `exp` has a value of 4, control transfers to the statement identified by the fourth label in the list, and so forth. The value of `exp` is truncated and converted to integer, if necessary.

If the value of `exp` is less than 1 or greater than the number of labels in the list, execution continues with the statement following the computed GOTO. As an example, in the following,

```
GOTO(10,20,30,20) L
```

the next statement executed is 10 if `L` equals 1, 20 if `L` equals 2 or 4, and 30 if `L` equals 3.

5.30.3 Assigned GOTO Statement

The assigned GOTO statement transfers control to the executable statement last assigned to an integer variable by the execution of a prior ASSIGN statement.

Syntax:

```
GOTO iv [[,] (slab[,slab]...)]
```

Where:

`slab` is the label of an executable statement.

`iv` is an integer variable.

Description:

The variable `iv` must not be defined by any statement other than an ASSIGN statement. The list of statement labels is optional. The possible branches taken by the GOTO are determined entirely by the assignments made to the variable within the program unit.

In the following example,

```
   ASSIGN 50 TO JUMP
10 GOTO JUMP,(20,30,40,50)
20 CONTINUE
30 CAT=ZERO+HAT
40 CAT=10.1-3.
50 CAT=25.2+7.3
```

statement 50 is executed immediately after statement 10.

5.31 IF Statement

The IF statement evaluates an expression and conditionally transfers control or executes another statement, depending on the outcome of the test.

Syntax:

```
IF(aexp)slab1,slab2,slab3
```

```
IF(lexp) stat
IF(lexp) THEN
    .
    .
    .
[ELSEIF(lexp) THEN
    .
    .
    . ]
[ ELSE
    .
    .
    . ]
ENDIF
```

Where:

`aexp` is an arithmetic expression.

`lexp` is a logical expression.

`slab1` are labels of executable statements.
`slab2`
`slab3`

`stat` is any executable statement except a DO, block IF, ELSE, ELSE IF, END, END IF, or another logical IF statement.

Description:

The IF statement evaluates an expression and conditionally transfers control or executes another statement, depending on the outcome of the test. The kinds of IF statements are arithmetic IF, logical IF, and block IF.

See also:

The ELSEIF, ELSE, and ENDIF statements which discuss the other components of the block IF statement.

5.31.1 Arithmetic IF Statement

The arithmetic IF statement transfers control to one of three labeled statements, depending on the value of an expression.

Syntax:

```
IF(exp)slab1,slab2,slab3
```

Where:

`exp` is an arithmetic expression.

`slab1` are labels of executable statements.
`slab2`
`slab3`

Description:

If the value of `exp` is negative, control transfers to the first statement label; if `exp` is 0, control transfers to the second statement label; if `exp` is greater than 0, control transfers to the third statement label.

As an example, in the following,

```
      READ (5,10) I,J,K,N
10    FORMAT (10X,4I4)
      IF(I-N) 3,4,6
3     ISUM=J+K
6     CALL ERROR1
      WRITE (6,2) ISUM
2     FORMAT (110)
4     END
```

If `I` is less than `N`, control transfers to statement 3, else if `I` equals `N` control transfers to statement 4, else control transfers to statement 6.

5.31.2 Logical IF Statement

The logical IF statement allows for conditional execution of a statement.

Syntax:

```
IF(exp) stat
```

Where:

`exp` is a logical expression.

`stat` is any executable statement except a DO, block IF, ELSE, ELSE IF, END, END IF, or another logical IF statement.

Description:

If the value of `exp` is true, statement `stat` is executed. If the value of `exp` is false, `stat` is not executed; execution continues with the next statement.

As an example

```
IF(P.AND.Q) RES=7.2
TEMP=RES*Z
```

if `P` and `Q` are both true, the value of the variable `RES` is replaced by 7.2; otherwise, the value of `RES` is unchanged. In either case, `TEMP` is set equal to `RES` times `Z`.

As another example,

```
      IF (A.LT.B) CALL SUB1
20    ZETA=TEMP+RES4
```

if `A` is less than `B`, the subroutine `SUB1` is called. Upon return from this subroutine, statement 20 is executed. If `A` is greater than or equal to `B`, statement 20 is executed and `SUB1` is not called.

5.31.3 Block IF Statement

The block IF statement allows conditional execution of a block of executable statements.

Syntax:

```
IF(exp1) THEN
    .
    .
    .
[ ELSEIF(exp2) THEN
    .
    .
    . ]
[ ELSE
    .
    .
    . ]
ENDIF
```

Where:

`exp1` are logical expressions.
`exp2`

Description:

The block IF statement is used with the END IF and, optionally, the ELSE and ELSE IF statements to form block IF structures. If the logical expression is true, execution continues with the next executable statement. If `exp1` or `exp2` is false, control transfers to an ELSE or ELSE IF statement; or if none are present, to an END IF statement. If control is transferred to an ELSEIF, it behaves exactly as the original IF, except that its logical expression is tested. If control is transferred to an ELSE, then the statements immediately following it are executed.

In general, block IF structures provide for alternative execution of blocks of statements. A block IF structure begins with a block IF statement, ends with an END IF statement and, optionally, includes one ELSE or one or more ELSE IF statements. Each block IF, ELSE, and ELSE IF statement is followed by an associated block of executable statements called if-block.

The simplest form of a block IF structure is shown below.

```
IF(exp) THEN
if-block
ENDIF
```

If expression `exp` is true, execution continues with the first statement in the if-block. If `exp` is false, control transfers to the statement following the END IF statement. The if-block can contain any number of executable statements, including block IF statements. An actual example of this structure is as follows:

```
IF (I.EQ.0) THEN
X=X+DX
Y=Y+DY
END IF
```

If `I` is zero, the subsequent statements are executed. If not, control transfers to the statement following END IF.

Control can be transferred out of an if-block from inside the if-block. Control can be transferred into an if-block from outside the if-block; however, this practice is not recommended. It is not permissible to branch directly to an ELSE, ELSE IF, or END IF statement. However, it is permissible to branch directly to a block IF statement.

When execution of the statements in an if-block has completed, and if control has not been transferred outside an if-block, execution continues with the statement following ENDIF.

A block IF structure can contain one ELSE statement to provide an alternative path of execution within the structure. This structure is shown below.

```
IF (exp) THEN
if-block-1
ELSE
if-block-2
END IF
```

In the structure with an ELSE statement, execution continues with the first statement in `if-block-1` if `exp` is true. If the last statement of `if-block-1` does not transfer control, control transfers to the statement following ENDIF. However, if `exp` is false, control transfers to the first statement in `if-block-2`. If the last statement in `if-block-2` does not transfer control, execution continues with the statement following END IF. A block IF statement can have at most one associated ELSE statement.

An actual example is as follows:

```
READ (2,12) A,B
IF(XSUM.LT.XLIM) THEN
X(I)=A/2.0+B/2.0
XSUM=XSUM+X(I)
WRITE (3,14) X(I),XSUM
ELSE
Y(I)=A*B
YSUM=Y(I)
WRITE (3,16) YSUM,Y(I)
ENDIF
```

An IF structure can contain one or more ELSEIF statements to provide for alternative execution of additional block IF statements. An example of this structure is as follows:

```
IF(exp1) THEN
   if-block-1
ELSEIF(exp2) THEN
   if-block-2
ELSEIF(exp3) THEN
   if-block-3
ELSE
   if-block-4
ENDIF
```

This capability allows formation of IF structures containing a number of possible execution paths depending on the outcome of the associated IF tests. In this structure, the initial block IF statement and each ELSEIF or ELSE statement has an associated if-block. Only one if-block in this structure is executed (if no nested levels appear). Each logical expression is evaluated until one is found that is true. Control then transfers to the first statement of the associated if-block. When execution of the if-block has completed, and if control has not been transferred, control transfers to the statement following ENDIF. If none of the logical expressions are true and no ELSE statement appears, no if-blocks are executed; control transfers to the statement following ENDIF. In this structure, at most one if-block is executed.

If an ELSE statement appears, it must follow the last ELSE IF statement. If no logical expression is true, control transfers to the statement following ELSE.

There are no restrictions about branching in or out of if-blocks. If a statement branches into an if-block, then execution continues from that point in exactly the same manner as if control had normally flowed to that statement via the logic of the block IF.

Any given if-block may be empty — i.e., contain no statements.

Finally, block IF structures can be nested: that is, any if-block within a structure can itself contain block IF structures. Within a nesting hierarchy, control can transfer from a lower level structure into a higher level structure; however, control cannot transfer from a higher level structure into a lower level structure without the use of explicit GOTO statements.

5.32 IMPLICIT Statement

The IMPLICIT statement specifies the default typing that occurs according to the first letters of the names.

Syntax:

```
IMPLICIT type(ac[,ac]...) [,type(ac[,ac]...)]...
```

or

```
IMPLICIT NONE
```

Where:

`type` is any valid type specification.

`ac` is a single letter, or range of letters represented by the first and last letter separated by a hyphen.

Description:

The IMPLICIT statement changes or confirms the default typing that occurs according to the first letters of the names. The valid type specifications are as follows: BYTE, INTEGER*1, INTEGER*2, INTEGER*4, INTEGER, REAL*4, REAL*8, REAL, DOUBLECOMPLEX, DOUBLEPRECISION, LOGICAL*1, LOGICAL*2, LOGICAL*4, LOGICAL, COMPLEX*8, COMPLEX*16, COMPLEX, CHARACTER[*len].

The letters `ac` indicate which variables are implicitly typed to the specified type. As an example

```
IMPLICIT CHARACTER*20 (M,X-Z)
```

specifies that variables whose names begin with the letter `M` are typed as character rather than integer; and those beginning with `X`, `Y`, or `Z` are character rather than real. Default typing is effective in all other cases. In this second example,

```
IMPLICIT LOGICAL (L)
INTEGER L,LX,TT
```

variable `L` is integer, rather than logical, because it is explicitly typed. `LX` is integer. The name `TT` is integer, because an explicit type overrides the default typing.

The statement specifies the type of variables, arrays, symbolic constants, and functions beginning with the letters `ac`. The IMPLICIT statements in a program unit must precede all other specification statements except PARAMETER statements. An IMPLICIT statement in a function or subroutine subprogram affects the type associated with dummy arguments and the function name, as well as other variables in the subprogram. Explicit typing of a variable name or array element in a type statement or FUNCTION statement overrides an IMPLICIT specification.

The specified single letters or ranges of letters specify the entities to be typed. A range of letters has the same effect as writing a list of the single letters within the range. The same letter can appear as a single letter, or be within a range of letters, only once in all IMPLICIT statements in a program unit.

The length can be specified implicitly for entities of type character. If length is not specified, the length is one. The length can be specified as an unsigned nonzero integer constant, or an integer constant expression, enclosed in parentheses, with a positive value. The specified length applies to all entities implicitly typed as character.

Note that any explicit typing with a type statement is effective in overriding both the default typing and any implicit typing.

The IMPLICIT NONE statement overrides all implicit defaults. All data types in the program unit must be explicitly declared. If you specify IMPLICIT NONE, you must not include any other IMPLICIT statement in the program unit.

The IMPLICIT statement has no effect on the default types of intrinsic functions.

See also:

The explicit TYPE statements BYTE, INTEGER, DOUBLECOMPLEX, DOUBLEPRECISION, REAL, LOGICAL, and CHARACTER which describe the actual meanings of the various types.

5.33 INCLUDE Statement

The INCLUDE statement specifies that a given source file be included as part of a source file.

Syntax:

```
INCLUDE 'file'
```

Where:

`file` is the name of a file

Description:

When the compiler encounters an INCLUDE statement, it stops reading from the current file and reads the statements in the included file. When it reaches the end of the included file, the compiler resumes reading the statement immediately after the INCLUDE statement. An INCLUDE statement may appear within an included file; however, nesting may only be five deep.

The INCLUDE statement can appear anywhere in a program unit; however, it may not begin with a continuation line or with a comment within a continued statement. Statements within an included file are treated in exactly the same manner as if they had been physically contained within the file containing the INCLUDE.

5.34 INTEGER Statement

The INTEGER statement defines some user defined entity to be of type integer, short integer, or byte.

Syntax:

```
INTEGER[*len][,]name[,name]...
```

Where:

`name` has one of the forms

```
var [*len] [/ c /]
array [(d[,d]...)] [*len] [/ clist /]
```

`len` specifies the integer subtype and can be an unsigned nonzero integer constant whose value is 1, 2 or 4.

`var` is a variable, function name, symbolic constant, or dummy procedure.

`array` is an array name.

`d` specifies the bounds of a dimension.

`clist` is a list of constants or symbolic constants specifying the initial values. Each item in the list can take the form:

```
c
r*c
```

`c` is a constant or symbolic constant.

`r` is a repeat count that is an unsigned nonzero integer constant or the symbolic name of such a constant.

Description:

The INTEGER statement is used to define a variable, array, symbolic constant, function name, or dummy procedure name as type integer, short integer, or byte. A length specification immediately following the word INTEGER applies to each entity not having its own length specification. A length specification immediately following an entity is the length specification only for that entity. If the length specification for a given entity is 4 or if it is omitted, then the type defined is integer. If the value is 2, the type is short integer, and if the value is 1, the type is byte.

See also:

See the discussion of the DIMENSION statement for a description of how dimension bounds are defined.

See the discussion of the DATA statement for a description of how initial values are defined.

5.35 INQUIRE Statement

The INQUIRE statement obtains information about the current status of a file or unit.

Syntax:

```
INQUIRE([UNIT=]u[,IOSTAT=ios][,ERR=sl][,OPENED=od][,NUMBER=num]
   [,NAMED=nmd][,NAME=fn][,ACCESS=acc][,SEQUENTIAL=seq][,DIRECT=dir]
   [,FORM=fm][,FORMATTED=fmt][,UNFORMATTED=unf][,RECL=rcl][,NEXTREC=nr]
   [,BLANK=blnk][,EXIST=ex])

INQUIRE(FILE=fin [,IOSTAT=ios][,ERR=sl][,OPENED=od][,NUMBER=num]
   [,NAMED=nmd][,NAME=fn][,ACCESS=acc][,SEQUENTIAL=seq][,DIRECT=dir]
   [,FORM=fm][,FORMATTED=fmt][,UNFORMATTED=unf][,RECL=rcl][,NEXTREC=nr]
   [,BLANK=blnk][,EXIST=ex])
```

Where:

`u` is an integer expression.

`fin` is a character expression.

`ios` is an integer lvalue.

`sl` is the label of an executable statement.

`ex`
`od`
`nmd` are logical lvalues.

`num`
`rcl`
`nr` are integer lvalues.

`fn` are character lvalues.
`acc`
`seq`
`fm`
`fmt`
`unf`
`blnk`
`dir`

Notes:

The "UNIT=" specification is optional; however, when omitted the unit specification must be the first specification. Other than the above, the order of the parameters within the INQUIRE statement is free.

Description:

There are two forms of the INQUIRE statement. Inquire by unit is used to obtain information about the current status of a specified unit. Inquire by file is used to obtain information about the current status of a file. Either a file name (inquire by file) or a unit specifier (inquire by unit), but not both, must be specified in an INQUIRE statement. The file or unit need not exist when INQUIRE is executed.

Following execution of an INQUIRE statement, the specified parameters contain values that are current at the time the statement is executed. The actual information returned in the parameters is as follows:

`u` is the unit number of the file.

`ex` is a logical variable which returns .TRUE. if the file (unit) exists, and .FALSE. if the file (unit) does not exist.

`ios` is the status code for the file.

`sl` is the statement label where execution branches if an error occurs.

`od` is a logical variable which returns .TRUE. if the file (unit) is connected to a unit (file), and .FALSE. if the file (unit) is not connected to a unit (file).

`num` is an integer variable which returns the external unit number of the unit currently associated with the file; undefined if the file is not associated with a unit.

`nmd` is a logical variable which returns .TRUE. if the file has a name, and .FALSE. if the file does not have a name.

`fn` is a character variable which returns the name of the file associated with unit u.

`acc` is a character variable which returns the access method of the file: 'SEQUENTIAL' if the file is opened for sequential access input/output, and 'DIRECT' if the file is opened for direct access input/output. If the file is not opened, acc is undefined.

`seq` is a character variable which returns whether or not the file can be opened for sequential access input/output. It returns 'YES' if the file can be opened for sequential access input/output, 'NO' if the file cannot be opened for sequential access input/output, and 'UNKNOWN' if this cannot be determined.

`dir` is a character variable which returns whether or not the file can be opened for direct access input/output. It returns 'YES' if the file can be opened for direct access input/output, 'NO' if the file cannot be opened for direct access input/output, and 'UNKNOWN' if this cannot be determined.

`fm` is a character variable which returns whether or not the file is formatted. It returns 'FORMATTED' if the file is opened for formatted input/output, and 'UNFORMATTED' if the file is opened for unformatted input/output. If the file has not been opened, fm is undefined.

`fmt` is a character variable specifying whether the file can be opened for formatted input/output. It returns 'YES' if the file can be opened for formatted input/output, 'NO' if the file cannot be opened for formatted input/output, and 'UNKNOWN' if it cannot be determined if the file can be opened for formatted input/output.

`unf` is a character variable specifying whether the file can be opened for unformatted input/output. It returns 'YES' if the file can be opened for unformatted input/output, 'NO' if the file cannot be opened for unformatted input/output, and 'UNKNOWN' if it cannot be determined if the file can be opened for unformatted input/output.

`rcl` is an integer variable which returns the record length of a file opened for direct access. If the file is 'FORMATTED', `rcl` contains the record length in characters; if 'UNFORMATTED', the record length is in words. It is undefined if the file is not opened for direct access.

`nr` is an integer variable which, for a direct access file, returns the record number of the last record read or written plus one. If no records have been read or written, `nr` contains 1. For sequential files it is undefined.

`blnk` is a character variable which specifies the treatment of blanks. It returns 'NULL' if null blank control is in effect for a file opened for formatted input/output. It returns 'ZERO' if zero blank control is in effect for a file opened for formatted input/output. It is undefined if the file is not opened for formatted input/output.

If an error occurs as a result of the inquire, and if neither `sl` nor `ios` are supplied, then execution will terminate with an error code set. If either or both are supplied, the execution continues despite any errors.

If supplied, the `ios` lvalue receives the runtime error code for the error condition encountered or a zero, if no error occurred.

If `sl` is supplied, then execution will branch to the statement labeled by it if an error occurs.

If a unit number is specified and the unit is opened, the NAMED, NAME, ACCESS, SEQUENTIAL, DIRECT, FORM, FORMATTED, UNFORMATTED, RECL, NEXTREC, OPENED, EXIST, NUMBER, ACCESS, and BLANK variables will contain information about the file associated with the unit.

If a file name is specified, the NAMED, NAME, SEQUENTIAL, DIRECT, FORMATTED, UNFORMATTED, OPENED, EXIST, NUMBER, ACCESS, FORM, RECL, NEXTREC, and BLANK variables will contain information about the file and the unit it is associated with. EXIST returns a TRUE value only if a non-empty local file by this name exists or if an empty local file by this name is currently open.

When EXIST returns a FALSE value, the NUMBER, NAMED, NAME, ACCESS, SEQUENTIAL, DIRECT, FORM, FORMATTED, UNFORMATTED, RECL, NEXTREC, and BLANK variables will contain undefined values. This does not result in an error.

If a file is specified that is associated with more than one unit, the NUMBER variable will contain one of the unit numbers or names.

Note that if a unit that is not associated with a file is specified, only the IOSTAT and EXIST variables contain values.

See also:

The discussion of the OPEN statement describes the FORTRAN file system in general, including the different file types.

5.36 INTRINSIC Statement

The INTRINSIC statement identifies a name as representing an intrinsic function.

Syntax:

```
INTRINSIC fun[,fun]...
```

Where:

`fun` is an intrinsic function name.

Notes:

The INTRINSIC statement performs the same action as the GENERIC statement.

Description:

The INTRINSIC statement is used to identify a name as representing a generic or intrinsic function. This statement enables use of a generic function name as an actual argument. As an example

```
SUBROUTINE DC
INTRINSIC SQRT
CALL SUBA(X,Y,SQRT)
END
SUBROUTINE SUBA(A,B,FNC)
B=FNC(A)
END
```

The name `SQRT` is declared generic in subroutine `DC` and passed as an argument to subroutine `SUBA`. Within `SUBA`, the reference `FNC(A)` references the generic function `SQRT`. Note that if `SQRT` were declared EXTERNAL, then a user function would be assumed.

The appearance of a name in an INTRINSIC statement declares the name as a generic, intrinsic function name. If an intrinsic function name is used as an actual argument in a program unit, it must appear in a GENERIC or INTRINSIC statement in the program unit. The following intrinsic function names must not be used as actual arguments:

Type conversion functions BOOL, CHAR, CMPLX, DBLE, FLOAT, ICHAR, IDINT, IFIX, INT, REAL, and SINGL

Lexical relationship functions LGE, LGT, LLE, and LLT

Largest/smallest value functions AMAX0, AMAX1, AMIN0, AMIN1, DMAX1, DMIN1, MAX, MAX0, MAX1, MIN, MIN0, MIN1

Logical and masking functions AND, OR, XOR, NEQV, EQV

The appearance of a generic intrinsic function name in an INTRINSIC statement does not remove the generic properties of the name.

See Also:

The EXTERNAL statement which allows user subprograms to be used as parameters.

The GENERIC statement which is equivalent to this statement.

5.37 LOGICAL Statement

The LOGICAL statement defines some user defined entity to be of type logical, short logical, or logical byte.

Syntax:

```
LOGICAL[*len][,]name[,name]...
```

Where:

`name` has one of the forms

```
var [*len] [/ c /]

array [(d[,d]...)] [*len] [/ clist /]
```

`len` specifies the integer subtype and can be an unsigned nonzero integer constant whose value is 1, 2 or 4.

`var` is a variable, function name, symbolic constant, or dummy procedure.

`array` is an array name.

`d` specifies the bounds of a dimension.

`clist` is a list of constants or symbolic constants specifying the initial values. Each item in the list can take the form:

```
c
r*c
```

`c` is a constant or symbolic constant.

`r` is a repeat count that is an unsigned nonzero integer constant or the symbolic name of such a constant.

Description:

The LOGICAL statement is used to define a variable, array, symbolic constant, function name, or dummy procedure name as type logical, short logical, or logical byte. A length specification immediately following the word LOGICAL applies to each entity not having its own length specification. A length specification immediately following an entity is the length specification only for that entity. If the length specification for a given entity is 4 or if it is omitted, then the type defined is logical. If the value is 2, the type is short logical, and if the value is 1, the type is logical byte.

See also:

See the discussion of the DIMENSION statement for a description of how dimension bounds are defined.

See the discussion of the DATA statement for a description of how initial values are defined.

5.38 NAMELIST Statement

The NAMELIST statement defines a group of variables and/or arrays for later input or output.

Syntax:

```
NAMELIST/name/a[,a]...[/name/a[,a]...]...
```

Where:

`name` is a symbolic name.

`a` is a variable or array name.

Description:

The NAMELIST statement permits the input and output of groups of variables and arrays with an identifying name. No format specification is used. The NAMELIST statement is a nonexecutable statement that appears in the program following the declarative portion. The symbolic group name must be enclosed in slashes and must be unique within the program unit.

The namelist group name identifies the succeeding list of variables or array names. It must be declared in a NAMELIST statement before it is used in an input/output statement. The group name can be declared only once, and it cannot be used for any purpose other than a namelist name in the program unit. It can appear in READ, WRITE, PRINT, and PUNCH statements in place of the format designator. When a namelist group name is used, the list must be omitted from the input/output statement.

A variable or array name can belong to one or more namelist groups. Assumed size arrays cannot appear in a namelist group.

Data read by a single namelist name READ statement must contain only names listed in the referenced namelist group. All items in the namelist group, or any subset of the group, can be input. Values are unchanged for items not input. Variables need not be in the order in which they appear in the defining NAMELIST statement.

The program in the following example

```
      PROGRAM NMLIST
      NAMELIST /SHIP/ A,B,C,I1,I2
      READ(*, SHIP,END=10)
      IF(C .GT. 0.0) THEN
      A=B+C
      I1=I1+I2
      WRITE(*, SHIP)
      ENDIF
      STOP
10    PRINT *, 'NO DATA FOUND'
      STOP
      END
```

can read the following input record

```
$SHIP A=14.7,B=12.3,C=3.4,I1=58,I2=8$END
```

and if the above were read would produce the following output:

```
$SHIP
A = .157E+02,
B = .123E+02,
C = .34E+01
I1 = 66,
I2 = 8,
$END
```

The following sections discuss namelist input and output and arrays in namelist. The syntax descriptions are simplified to focus the discussion on the namelist. See the sections on READ, WRITE, PRINT, and PUNCH for a general discussion of these statements.

5.38.1 NAMELIST READ Statement

The namelist READ statement reads input data from a designated file. Its simplified syntax is shown below.

Syntax:

```
READ(u,name)
```

Where:

`u` is an integer expression.

`name` is the identifier of a NAMELIST group.

Description:

When a READ statement references a namelist group name, input data in the format described below is read from the designated file.

Syntax:

```
$name entity = value [, entity = value, ...] $END
```

Where:

`name` is the name of the namelist that contains the entity or entities whose values are being specified.

`entity` is a namelist-defined entity. It can be a variable, array name, subscripted variable, variable with a substring, or a subscripted variable with a substring.

`value` is a constant, a list of constants, a repetition of constant in the form `r*c`, or a repetition of values in the form `r*`.

Where

`c` is a constant or symbolic constant.

`r` is a repeat count that is an unsigned nonzero integer constant or the symbolic name of such a constant.

As an example, consider the following:

```
$AGRP                 Group name
XVAL=5.0,
ARR=5*(1.7,-2.4),     Five complex numbers
CHAR='HI THERE',
$END                  Group terminator
```

In each record of a namelist group, column one is reserved for carriage control and must be left blank. Data items following `$name` are read until another `$` is encountered.

Blanks must not appear in the following locations:

(1) Between $ and namelist group name,
(2) Between $ and END,
(3) Within array names and variable names.

Blanks can be used freely elsewhere.

Complex constants can be broken across records between the real part and the comma, and between the comma and the imaginary part.

More than one record can be used as input data in a namelist group. The first column of each input record is ignored. All input records containing data should end with a constant followed by a comma; however, the last record can be terminated by a $ without the final comma. Constants can be preceded by a repetition factor followed by an asterisk. Omitting a constant constitutes a fatal error.

Constants can be integer, real, double precision, complex, logical, exact representation or character. Each constant must agree with the type of the corresponding input list item as follows:

(1) Character constants and exact representation constants may be associated with any input list item type. A character constant is truncated from the right, or extended on the right with blanks, if necessary, to yield a constant of the same length as the variable, array element, or substring.

(2) A logical or complex constant must be of the same type as the corresponding input list item.

(3) An integer, real, or double precision constant can be used for an integer, real, or double precision input list item. The constant is converted to the type of the list item.

Logical constants have the following forms:

```
.TRUE.        .FALSE.
.T.           .F.
Tccc          Fccc
```

If the third form is used, any words starting with T or F may be used. Note that either upper or lower case may be used.

A character constant must have delimiting single or double quotes. If a character constant occupies more than one record, each continuation of the constant must begin in column two.

A complex constant has the form (real constant, real constant).

Blank characters appearing within noncharacter constants are ignored. The BLANK= parameter in an OPEN statement has no effect on namelist. If a constant, other than a character constant, contains no characters other than blanks, a fatal error results.

5.38.2 NAMELIST WRITE, PRINT, PUNCH Statements

The three namelist output statements are WRITE, PRINT, and PUNCH. Their simplified syntax is shown below.

Syntax:

```
WRITE(u, name)

PRINT name

PUNCH name
```

Where:

u is an integer expression.

name is the identifier of a NAMELIST group.

Description:

All variables and arrays and their values in the list associated with the namelist group name are output on the file associated with unit u, OUTPUT, or PUNCH. They are output in the order of specification in the NAMELIST statement. Output consists of at least three records. The first record is a $ in column 2 followed by the group name; the last record is a $ in column 2 followed by the characters END. Each group begins with a new record.

As an example, the following

```
PROGRAM NAME
NAMELIST /VALUES/ TOTAL,QUANT,COST
DATA QUANT,COST /15.,3.02/
TOTAL = QUANT*COST*1.3
WRITE (6,VALUES)
STOP
END
```

produces the following output:

```
$VALUES
TOTAL     =   .5889E+02,
QUANT     =   .15E+02,
COST      =   .302E+01,
$END
```

No data appears in column 1 of any record. Logical constants appear as T or F. Elements of an array are output in the order in which they are stored. Character constants are written with delimiting single quotes.

Records output by a namelist WRITE statement can be read later in the same program by a namelist READ statement specifying the same group name.

5.38.3 Arrays in NAMELIST

In input data the number of constants, including repetitions, given for an array name should not exceed the number of elements in the array. As a simple example of array use, consider the following code fragment

```
INTEGER BAT(10)
NAMELIST /HAT/ BAT,DOT
READ (5,HAT)
```

processing the input record

```
$HAT BAT=2,3,8*4,DOT=1.05$END
```

The value of DOT becomes 1.05; the array BAT is as follows:

```
BAT(1)       2
BAT(2)       3
BAT(3)       4
BAT(4)       4
```

```
BAT(5)       4
BAT(6)       4
BAT(7)       4
BAT(8)       4
BAT(9)       4
BAT(10)      4
```

As a second example, consider the code fragment

```
DIMENSION GAY(5)
NAMELIST /DAY/ GAY,BAY,RAY
READ (5,DAY)
```

when it processes the input record:

```
$DAY GAY(3)=7.2,8.3,GAY(5)=3.0,BAY=2.3,RAY=77.2$
```

When data is input in the form

```
array element=constant,...,constant
```

the constants are stored consecutively beginning with the location given by the array element. The number of constants need not be equal, but must not exceed, the remaining number of elements in the array. As a final example

```
DIMENSION Y(3,5)
LOGICAL L
COMPLEX Z
NAMELIST /HURRY/ I1,I2,I3,K,M,Y,Z,L
READ(5, HURRY)
```

can process the following input record:

```
$HURRY I1=1,L=.TRUE.,I2=2,I3=3.5,Y(3,5)=26,Y(1,1)=11,
12.0E1,13,4*14,Z=(1.,2.),K=16,M=17$
```

The final values stored are as follows:

```
I1=1             Y(1,2)=14.0
I2=2             Y(2,2)=14.0
I3=3             Y(3,2)=14.0
Y(3,5)=26.0      Y(1,3)=14.0
Y(1,1)=11.0      K=16
Y(2,1)=120.0     M=17
Y(3,1)=13.0      Z=(1.,2.)
L=.TRUE.
```

The rest of `Y` is unchanged.

5.39 OPEN Statement

The OPEN statement can be used to associate an existing file with a unit number to create a new file and associate it with a unit number, or to change certain attributes of an existing file.

Syntax:

```
OPEN unit

OPEN ([UNIT=]u[,IOSTAT=ios][,ERR=sl][,FILE=fin][,STATUS=sta][,ACCESS=acc]
     [,FORM=fm][,RECL=rl][,BLANK=blnk]
```

Where:

`u` is an integer expression.
`unit`

`ios` is an integer lvalue.

`sl` is the label of an executable statement.

`fin` is a character expression.
`sta`
`acc`
`fm`
`blnk`

`r` is an integer expression.

Notes:

The simplest form of the OPEN statement consists of a single unit specification not enclosed in parentheses. With this form no additional parameters can be specified. If the parenthetical form of OPEN is used, then the "UNIT=" specification is optional; however, when omitted the unit specification must be the first specification. Other than the above, the order of the parameters within the parenthetical version of the OPEN statement is free.

Description:

The OPEN statement can be used to associate an existing file with a unit number, to create a new file and associate it with a unit number, or to change certain attributes of an existing file. The UNIT= parameter is required; all other parameters are optional, except for the RECL parameter, which must be specified if a file is being opened for direct access.

If an error occurs as a result of the open, and if neither ERR= nor STATUS= are supplied, then execution will terminate with an error code set. If either or both are supplied, the execution continues despite any errors.

If supplied, the STATUS= `sta` lvalue receives the runtime error code for the error condition encountered or a zero, if no error occurred.

If `ERR=sl` is supplied, then execution will branch to `sl`, the statement labeled by it if an error occurs.

The meaning of the other individual parameters on the OPEN statement is as given below.

UNIT=`u` specifies the unit number of the file to be opened.

FILE=`fin` is a character expression whose value is the name of the file to be opened. Trailing blanks are removed. This file becomes associated with unit `u`. If FILE is omitted, then a file name is created. See your system manager for the conventions used for creating this name at your site.

STATUS=`sta` is a character expression specifying file status. Valid values are:

'OLD' File `fin` currently exists.

'NEW' File `fin` does not currently exist.

'SCRATCH' Delete the file associated with unit `u` on program termination or execution of CLOSE that specifies unit `u`.

'UNKNOWN' File status is unknown.

If STATUS is omitted, then an unknown status is assumed.

ACCESS=acc is a character expression specifying the access method of the file. Valid values are:

'SEQUENTIAL' File is to be opened for sequential access.

'DIRECT' File is to be opened for direct access.

If ACCESS is omitted, then sequential access is assumed. If the file exists, the access method must be valid for the existing file. Direct access files are discussed further in a following subsection.

FORM=fm is a character expression having one of the following values:

'FORMATTED' File is being opened for formatted input/output.

'UNFORMATTED' File is being opened for unformatted input/output.

If FORM is omitted, FORMATTED is assumed for sequential access files and UNFORMATTED is assumed for direct access files. For an existing file, the specified form must be valid for that file.

RECL=rl is an integer expression specifying the record length for a direct or sequential access file. RECL is required for direct access files. If omitted for sequential files, variable length records are assumed.

BLANK=blnk is a character expression having one of the following values:

'NULL' Blank values in numeric formatted input fields are ignored, except that a field of all blanks is treated as zeros.

'ZERO' Blanks, other than leading blanks, are treated as zeros.

If omitted, blanks are ignored in numeric input fields.

Once properties of a file have been established in an OPEN statement, only the BLANK= parameter can be changed in a subsequent OPEN statement for that file, unless the file is first closed in a CLOSE statement.

Once a file has been associated with a particular unit, the file can be associated with another unit in a subsequent OPEN statement. The file is then associated with more than one unit. In this case the unit numbers refer to the same file. Actions taken on one unit also affect the other unit. For example, closing a unit closes all other units associated with the same file.

If a file is associated with a unit and a succeeding OPEN statement associates a different file with the same unit, the effect is the same as performing a CLOSE without a STATUS= parameter on the currently associated file before associating the new file with the unit.

5.39.1 Direct Access Files

Direct access file manipulations differ from conventional sequential file manipulations. In a sequential file, records are stored in the order in which they are written, and they normally can be read back only in the same order. This can be slow and inconvenient in applications where the order of writing is not the same as the retrieval order. In addition, such processing requires a continuous awareness of the current file position and the position of the required record. To remove these limitations, a direct access file capability is provided by the FORTRAN input/output statements.

In a direct access file, any record can be read, written, or rewritten directly, without concern for the position or structure of the file. This is possible because the file resides on a random access mass storage device that can be positioned to any portion of a file. Thus, the entire concept of file position does not apply to a direct access file. The notion of rewinding a direct access file is, for instance, without meaning.

To create a direct access file the user must specify an OPEN statement with ACCESS='DIRECT' and include the RECL (record length) parameter. For example,

```
OPEN(2,FILE='DAFL',ACCESS='DIRECT',RECL=120)
```

opens an unformatted file `DAFL` for direct access. The file is associated with unit 2 and has a record length of 120 words.

All records in a direct access file must have the same length.

The record length for a formatted direct access file is specified in characters. The record length for an unformatted direct access file is specified in words. If the iolist for an unformatted WRITE contains character data, the record length to be written must still be specified in words. For all systems integer, logical, and real numeric items count as one word. Double precision and complex count as two words, and double complex count as four words. The method for counting character items, and short or byte length items unfortunately varies from system to system. You should consult with your system manager if you need to write records containing such items to an unformatted direct access file.

A record number identifies a record in a direct access file. The record number is a positive decimal integer that is assigned when the record is written. Once a record number is assigned to a record, the record can always be accessed by that record number. The order of record numbers is the order of records on a direct access file.

Records can be written, rewritten, or read by specifying the record number in a READ or WRITE statement. Records can be read or written in any order; they need not be referenced in the order of their record numbers. The number of the record to be read or written is specified in a READ or WRITE statement with the REC= parameter.

The REC= parameter, on a direct access READ statement, must not be set to a record number greater than the highest record number written in the file. An attempt to read record numbers greater than the highest in the file can return unpredictable data without any error being reported.

If the length of the iolist in a direct access formatted WRITE statement is less than the record length of the direct access file, the unused portion of the record is blank filled. A direct access WRITE statement must not write a record longer than the record length.

A direct access file can be opened for formatted or unformatted input/output. However, neither list directed nor namelist input/output can be used with direct access files.

5.40 PARAMETER Statement

The PARAMETER statement gives a symbolic name to a constant.

Syntax:

```
PARAMETER (p=e [,p=e]...)
```

Where:

`p` is a symbolic name.

`e` is a constant or a constant expression

Description:

The PARAMETER statement is used to give a symbolic name to a constant. PARAMETER statements can be used anywhere among the specification statements, but each symbolic constant must be defined in a PARAMETER statement before the first reference to the symbolic constant.

As an example consider the following:

```
      PARAMETER (ITER=20,START=5)
      CHARACTER CC*(*)
      PARAMETER (CC='(I4,F10.5)')
      DATA COUNT/START/
      DO 10 J=1,ITER
      READ CC,IX,RX
10    CONTINUE
```

The symbolic constant `START` is used to assign an initial value to variable `COUNT`, the symbolic constant `ITER` is used to control the DO loop, and the symbolic constant `CC` is used to specify a character constant format specification.

A constant expression can contain: a constant, a previously-defined symbolic constant, the arithmetic operators, an extended constant expression enclosed in parentheses.

If a symbolic name is of type integer, real, double precision, complex, or double complex, the corresponding expression must be an arithmetic constant expression. If the symbolic name is of type character or logical, the corresponding expression must be a character constant expression or logical constant expression. Each symbolic name becomes defined with the value of the expression that appears to the right of the equals, according to the rules for assignment. Any symbolic constant that appears in an expression must have been previously defined in the same or a different PARAMETER statement in the program unit.

A symbolic name of a constant can be defined only once in a program unit, and can identify only the corresponding constant. The type of a symbolic constant can be specified by an IMPLICIT statement or type statement before the first appearance of the symbolic constant in a PARAMETER statement. If the length of a symbolic character constant is not the default length of one, the length must be specified in an IMPLICIT statement or type statement before the first appearance of the symbolic constant. The easiest way to do this is to explicitly type the symbolic constant as character with length (*). The actual length of the constant is determined by the length of the string defining it in the PARAMETER statement. The length must not be changed by another IMPLICIT statement or by subsequent statements.

Once defined, a symbolic constant can appear in the program unit in the following ways:

In an expression in any subsequent statement

In a DATA statement as an initial value or a repeat count

In a complex constant as the real or imaginary part

A symbolic constant cannot appear in a FORMAT statement.

5.41 PAUSE Statement

The PAUSE statement temporarily suspends execution of a program.

Syntax:

```
PAUSE [n]
```

Where:

`n` is an integer or string expression.

Description:

The PAUSE statement causes the program to temporarily suspend execution. At the same time, the message PAUSE `n` is written to the CONSOLE file. The manner in which the program is restarted varies from system to system. Consult your system manager.

5.42 PRINT Statement

The PRINT statement transfers formatted information from the values named in the input/output list to the OUTPUT file.

Syntax:

```
PRINT fn [,iolist]
```

Where:

`fn` is a format specification.

`iolist` is an output list.

Notes:

The PRINT statement is a special case of the formatted WRITE statement.

Description:

The PRINT statement transfers formatted information from the values specified in the input/output list to the OUTPUT file according to the specified format. This statement is identical to the statement

```
WRITE(*,fn) iolist
```

As a simple example consider

```
    PROGRAM PRINT
    CHARACTER B*3
    A=1.2
    B='YES'
    N=19
    PRINT 4,A,B,N
4   FORMAT (G20.6,A,I5)
    PRINT 50
50  FORMAT (' END OF FILE')
    STOP
    END
```

which will convert the values of `A`, `B`, and `N` into a coded record using format 4 and will then write that record to the OUTPUT file, followed by a record containing " END OF FILE".

See also:

The WRITE statement describes format specifications and `iolists` in detail. See that statement for details on these.

5.43 PROGRAM Statement

The PROGRAM statement defines the program name and begins a new program definition.

Syntax:

```
PROGRAM name[(fpar[,fpar]...)]
```

Where:

`name` is a symbolic name.

`fpar` is a token of any type.

Description:

The PROGRAM statement defines a program name and begins a new program definition. The name of the program and any parameters associated with it are purely for documentation and are not used by the system in any way.

5.44 PUNCH Statement

The PUNCH statement transfers formatted information from the values named in the input/output list to the PUNCH file.

Syntax:

```
PUNCH fn [,iolist]
```

Where:

`fn` is a format specification.

`iolist` is an output list.

Notes:

The PUNCH statement is a special case of the formatted WRITE statement.

Description:

The PUNCH statement transfers formatted information from the values specified in the input/output list to the PUNCH file according to the specified format. This statement is identical to the statement

```
WRITE(u,fn) iolist
```

where `u` is a unit number assigned to the PUNCH file.

As a simple example consider

```
      PROGRAM PUNCH
      CHARACTER B*3
      A=1.2
      B='YES'
      N=19
      PUNCH 4,A,B,N
4     FORMAT (G20.6,A,I5)
      PUNCH 50
50    FORMAT (' END OF FILE')
      STOP
      END
```

which will convert the values of `A`, `B`, and `N` into a coded record using format `4` and will then write that record to the PUNCH file, followed by a record containing " END OF FILE".

See also:

The WRITE statement describes format specifications and iolists in detail. See that statement for details on these.

5.45 READ Statement

The READ statement transmits data from an external unit or internal file to memory storage locations.

Syntax:

```
READ fn[,ilist]

READ *,[ilist]

READ nl,[ilist]

READ([UNIT=u],[FMT=]fn [,REC=rn][,IOSTAT=ios][,ERR=sl][,END=sl]) [ilist]

READ([UNIT=u],[FMT=]nl [,REC=rn][,IOSTAT=ios][,ERR=sl][,END=sl]) [ilist]

READ([UNIT=u],[FMT=]*[,IOSTAT=ios][,ERR=sl][,END=sl]) [ilist]

READ([UNIT=]u[,IOSTAT=ios][,REC=rn][,ERR=sl][,END=sl] [ilist]
```

Where:

`u`	is an integer expression or a character variable identifier or an asterisk.
`fn`	is integer variable, the label of a FORMAT statement, a character expression, or an array identifier.
`rn`	is an integer expression.
`nl`	is the identifier of a NAMELIST group.
`ios`	is an integer lvalue.
`sl`	is the label of an executable statement.
`ilist`	is an input item list.

Notes:

The simplest form of the READ statement consists of a single format specification not enclosed in parentheses, possibly followed by an input item list. With this form no additional parameters can be specified. If one of the parenthetical forms of READ is used, then the "UNIT=" and "FMT=" specifications are optional; however, when omitted the unit specification must be the first specification and the `fn` specification must be second. Other than the above, the order of the parameters within the parenthetical versions of the READ statement is free.

Description:

The READ statement transfers input data to internal storage from records contained in external logical units or internal files. There are four basic types of READ statements:

(1) formatted read statements — those which contain an `fn` parameter;

(2) list directed read statements — those which contain an asterisk where an `fn` parameter would otherwise appear;

(3) namelist read statements — those which contain a NAMELIST group identifier where an `fn` parameter would otherwise appear;

(4) unformatted read statements — those which contain no `fn` parameter or asterisk or NAMELIST identifier in the `fn` position.

Formatted, list directed, and namelist read statements decode information from character records which may either be contained in external logical units or internal files. Unformatted read statements perform no decoding operations and may transfer information from external logical units only.

Formatted and unformatted read statements may operate with either sequential or direct access files, while list directed and namelist read statements may operate only with sequential files.

In general, the parameters on the various READ statements are as follows.

UNIT=`u` is an integer expression, a character variable identifier, or an asterisk. If it is an integer expression, then it refers to a specific external logical unit. If it is a character variable, then that variable is treated as an internal file of character records. If it is an *, then the read is performed from the standard INPUT device. If one of the simple forms of the READ is used, which contain no parentheses and thus no unit number, then these also use the standard input device.

FMT=`fn` is an integer variable, the label of a FORMAT statement, a character expression or an array identifier. It specifies a format to be used for formatted input. If `fn` is a statement label, then it identifies a FORMAT statement in the program unit containing the input statement, which specifies how the input records are formatted. If it is a character expression, then that expression specifies how the input records are formatted. If it is an array identifier, then that array is assumed to contain the format information. Finally, if it is an integer variable, the variable is assumed to have been assigned the statement number of a FORMAT statement by an ASSIGN statement.

REC=`rn` is an integer expression which specifies the number of the record to be read or written in the file. It must be greater than zero, and is valid for files opened for direct access only.

FMT=`nl` is the identifier of a NAMELIST group. The use of this parameter indicates a namelist read. See the NAMELIST statement for a description of the conventions used for the namelist read.

END=`sl` specifies the label of an executable statement to which control transfers when an end-of-file is encountered during an input operation. `END=sl` is ignored for direct access input operations.

ERR=`sl` specifies the label of an executable statement to which control transfers if an error condition is encountered during input processing.

IOSTAT=`ios` specifies an integer variable into which an error code is stored as follows:

-1 means that an end-of-file was encountered;

0 means that the input operation completed normally, and a value greater than zero indicates some other error condition.

`ilist` is an input item list described below.

Regardless of the type of the read, if an error occurs as a result, and if neither ERR=`sl` nor IOSTAT=`ios` is supplied, then execution will terminate with an error code set. If either or both are supplied, the execution continues despite any errors. If supplied, the IOSTAT=`ios` lvalue receives the runtime error code for the error condition encountered or a zero, if no error occurred. If ERR=`sl` is supplied, then execution will branch to the statement `sl` if an error occurs.

If END=`sl` is not supplied, then encountering an end-of-file is treated as any other runtime error, except that the error condition code is negative. If END=`sl` is supplied, then execution continues at the statement labeled, with IOSTAT=`ios` receiving a negative error condition code.

The input item list portion of an input statement specifies the items to be read and the order of transmission. The input item list can contain any number of items. List items are read sequentially from left to right. If no list appears one or more records are skipped.

A list item consists of a variable name, an array name, an array element name, a character substring name, or an implied DO list. List items are separated by commas. Subscripts in an input item list can be written as any valid subscript form. An array name without subscripts in an input list specifies the entire array in the order in which it is stored. The entire array (not just the first word of the array) is read. Assumed-size array names are legal in input lists.

On formatted input the iolist is scanned and each item in the list is paired with the field specification provided by the FORMAT statement. After one item has been input, the next format specification is taken together with the next element of the list; and so on, until the end of the list.

An implied DO specification has the following form:

```
(dlist [,i=e1,e2 [,e3]])
```

The elements `i`, `e1`, `e2`, and `e3` have the same meaning as in the DO statement, and `dlist` is an input item list. Redundant parentheses are allowed and are ignored. The range of an implied DO specification is that of `dlist`. The value of `i` must not be changed within the range of the implied DO list by the READ statement. Changes to the values of `e1, e2`, and `e3` have no effect upon the execution of the implied DO. However, their values can be changed in the READ statement if they are outside the range of the implied DO, and the change does have effect. For example,

```
READ 100, K, (A(I),I=1,K)
```

reads a value into `K` and uses that value as the terminal parameter of the implied DO.

The statements:

```
    K=2
    READ 10, (A(I),I=1,K)
10  FORMAT (F10.3)
```

read two records, each containing a value for `A`.

An implied DO can be used to obtain a single value more than one time. For example, the list `(A(K),B,K=1,5)` causes the value of variable `B` to be read five times. Obviously, the last value read will be the one stored in `B` when the READ is completed.

Input of array elements can be accomplished by using an implied DO. The list of variables followed by the DO index is enclosed in parentheses to form a single element of the input item list. For example,

```
READ(5,100) (A(I),I=1,3)
```

has the same effect as the statement:

```
READ(5,100) A(1),A(2),A(3)
```

A variable cannot be used as a control variable more than once in the same implied DO nest, but `ilist` items can appear more than once. The value of a control variable within an implied DO specification is defined within that specification. On exit from the implied DO specification the control variable retains the first value to exceed the upper limit `(e2)`.

The implied DO can be nested: that is, the `ilist` in an implied DO can itself contain an implied DO. The first (innermost) control variable varies most rapidly, and the last (outermost) control variable varies least rapidly. For example, a nested implied DO with two levels has the form:

```
((list,v1=e1,e2,e3),v2=ee1,ee2,ee3)
```

Nested implied DO loops are executed in the same manner as nested DO statements. The nested form can be used to read values into arrays.

Each execution of a READ statement processes at least one record. The formatted READ statement transmits data to storage locations named in `ilist` according to FORMAT specification `fn`. Once a READ is initiated, the FORMAT statement determines when a new record will be transmitted. The unformatted READ statement transmits one record from the specified unit `u` to the storage locations named in `ilist`. Records are not converted; no FORMAT statement is used. The information is transmitted from the designated file in the form in which it exists on the file without any conversion. If the number of words in the list exceeds the number of words in the record, an execution diagnostic results. If the number of locations specified in `ilist` is less than the number of words in the record, the excess data is ignored. If `ilist` is omitted, the unformatted READ skips one record.

The list directed READ statement reads data into the storage locations named in `ilist`, with the input data items being free-form with separators rather than in fixed-size fields. A list directed READ following a list directed READ that terminated in the middle of a record starts with the next data record.

Input data consists of a string of values separated by one or more blanks, or by a comma or slash, either of which can be preceded or followed by any number of blanks. Also, a line boundary, such as end-of-record or end-of-card, serves as a value separator; however, a separator adjacent to a line boundary does not indicate a null value.

Embedded blanks are not allowed in input values, except character values and complex numbers. The format of values in the input record is as follows. Integers use the same format as for integer constants. For real numbers any valid FORTRAN format for real or double precision numbers may be used. In addition, the decimal point can be omitted; it is assumed to be to the right of the mantissa. Complex numbers consist of two real values, separated by a comma, and enclosed by parentheses. The parentheses are not considered to be a separator. The decimal point can be omitted from either of the real constants. Each of the real values can be preceded or followed by blanks.

Character values consist of a string of characters (which can include blanks) enclosed by single or double quotes. A delimiting quote can be represented within a string by two successive occurrences. Character values can only be read into variables of any type and character substrings. If the string length exceeds the length of the list item, the string is truncated. If the string is shorter than the list item, the string is left-justified and remaining character positions are blank filled.

Logical values consist of an optional period, followed by a T or F, followed by optional characters which do not include separators (slashes or commas).

Exact representation constants may be read into variables of any type.

To repeat a value, an integer repeat constant is followed by an asterisk and the constant to be repeated. Blanks cannot be embedded in the repeat part of the specification.

A null can be input in place of a constant when the value of the corresponding list entity is not to be changed. A null is indicated by the first character in the input string being a comma or by two commas separated by an arbitrary number of blanks. Nulls can be repeated by specifying an integer repeat count followed by an asterisk and any value separator. The next value begins immediately after a repeated null. A null cannot be used for either the real or imaginary part of a complex constant; however, a null can represent an entire complex constant.

When the value separator is a slash, remaining list elements are treated as nulls and the remainder of the current record is discarded.

Input values must correspond in type to variables in the input/output list. Note that the form of a real value can be the same as that of an integer value.

See also:

The discussion of the OPEN statement describes the FORTRAN file system in general, including the different file types. The discussion of the FORMAT statement describes how format information is interpreted. The NAMELIST statement describes the namelist read statement.

5.46 REAL Statement

The REAL statement defines some user defined entity to be of type real or double precision.

Syntax:

```
REAL[*len][,]name[,name]...
```

Where:

`name` has one of the forms

```
var [*len] [/ c /]
array [(d[,d]...)] [*len] [/ clist /]
```

`len` specifies the real subtype and can be an unsigned nonzero integer constant whose value is 4 or 8.

`var` is a variable, function name, symbolic constant, or dummy procedure

`array` is an array name

`d` specifies the bounds of a dimension.

`clist` is a list of constants or symbolic constants specifying the initial values. Each item in the list can take the form:

```
c
r*c
```

`c` is a constant or symbolic constant.

`r` is a repeat count that is an unsigned nonzero integer constant or the symbolic name of such a constant.

Description:

The REAL statement is used to define a variable, array, symbolic constant, function name, or dummy procedure name as type real or double precision. A length specification immediately following the word REAL applies to each entity not having its own length specification. A length specification immediately following an entity is the length specification only for that entity. If the length specification for a given entity is 4 or if it is omitted, then the type defined is real. If the value is 8, then the type is double precision.

See also:

See the discussion of the DIMENSION statement for a description of how dimension bounds are defined.

See the discussion of the DATA statement for a description of how initial values are defined.

5.47 RETURN Statement

The RETURN statement terminates a subroutine or function subprogram.

Syntax:

```
RETURN[exp]
```

Where:

`exp` is an arithmetic expression.

Description:

The RETURN statement terminates the currently executing subroutine or function subprogram and returns control to the calling program unit. If the RETURN is used without the optional expression, then it is a "simple" return. If it is used with an expression, then it is an "alternate" return. Both subroutine subprograms and function subprograms can have simple returns. A simple return always begins execution in the calling program unit, at the statement immediately following the CALL statement. A multiple return exists when the subprogram has more than one RETURN statement, or when a single RETURN statement is separated from the END statement by other statements.

An alternate return is used only within a subroutine subprogram. It returns control to the referencing program unit at a place other than the next executable statement after the CALL statement. The RETURN statement in the form RETURN `exp` is used for an alternate return.

Control is returned to a specified point in the referencing program unit. The specified point is a statement label in the referencing program unit. The statement labels must be included in the actual argument list, each preceded by an asterisk. Control returns to the statement label determined by the integer value of the alternate return expression. If the value of the expression is less than one, or greater than the number of asterisks in the SUBROUTINE statement or ENTRY statement that is the current entry point, control returns to the statement following the CALL statement. For example, if a CALL statement contains five statement labels and if the alternate return expression evaluates to three, control returns to the third statement label specified in the actual argument in the alternate return list.

5.48 RECORD Statement

The RECORD statement creates a record of the form specified in a previously declared structure.

Syntax:

```
RECORD /sname/rlist
[,/sname/rlist]
    .
    .
    .
[,/sname/rlist]
```

Where:

`sname` is the name of a previously declared structure

`rlist` is a list of one or more variable names, array names, or array declarators, separated by commas.

Description:

The RECORD statement creates records of the form specified in a previously declared structure. All of the records named in any given `rlist` have the same structure and are allocated separately in memory. The RECORD statement is comparable to that of an ordinary type declaration except that aggregate data items are declared instead of scalar data items.

Record names can be used in COMMON and DIMENSION statements. They cannot be used in DATA, EQUIVALENCE, NAMELIST, or SAVE statements. Records initially have undefined values unless you have defined their values in their structure declarations.

See also:

The STRUCTURE statement describes how record structures are defined.

5.49 REWIND Statement

Syntax:

```
REWIND unit

REWIND ( [ UNIT= ] unit [ ,IOSTAT= status ] [ ,ERR= err ] )
```

Where:

`unit` is an integer expression

`status` is an integer lvalue

`err` the label of an executable statement

Notes:

The simplest form of the REWIND statement consists of a single unit specification not enclosed in parentheses. With this form no additional parameters can be specified. If the parenthetical form of REWIND is used, then the "UNIT=" specification is optional; however, when omitted the unit specification must be the first specification. Other than the above, the order of the parameters within the parenthetical version of the REWIND statement is free.

Description:

The REWIND statement positions a file at beginning-of-information so that the next input/output operation references the first record in the file. If the file is already at beginning-of-information, no action is taken.

If an error occurs as a result of the rewind, and if neither `err` nor `status` are supplied, execution will terminate with an error code set. If either or both are supplied, the execution continues despite any errors.

If supplied, the `status` lvalue receives the runtime error code for the error condition encountered or a zero, if no error occurred.

If `err` is supplied, then execution will branch to the statement `err` if an error occurs.

5.50 SAVE Statement

The SAVE statement is used to retain the definition status of entities within a subprogram after that subprogram has executed.

Syntax:

```
SAVE [a[,a]...]
```

Where:

a is a variable name, array name, or common block name enclosed in slashes.

Description:

The SAVE statement is used to retain the definition status of entities after the execution of a RETURN or END statement in a subprogram. A SAVE statement in a main program is optional and has no effect.

Dummy argument names, procedure names, and names of entities in a common block must not appear in the SAVE statement. A common block name (or // indicating blank common) has the effect of specifying all of the entities in the common block. A SAVE statement with no list is treated as though it contained the names of all allowable items in the program unit. If a common block name is specified in a SAVE statement in a subprogram, the common block name must be specified by a SAVE statement in every subprogram in which the common block appears.

Execution of a RETURN statement or an END statement within a subprogram may cause the entities within the subprogram to become undefined, except in the following cases:

(1) entities specified by SAVE statements

(2) entities that have been initially defined in a DATA declaration.

If a local variable or array that is specified in a SAVE statement and is not in a common block is defined in a subprogram at the time a RETURN or END statement is executed, that variable or array remains defined with the same value at the next reference to the subprogram.

5.51 STOP Statement

The STOP statement terminates program execution.

Syntax:

```
STOP[n]
```

Where:

n is an integer or string expression

The STOP statement writes the message STOP n in the CONSOLE file, and then terminates program execution. The actual value of n at the time of termination is shown in the STOP message.

5.52 STRUCTURE Statement

The STRUCTURE statement introduces a multistatement structure declaration block which is used to define the form and possibly the content of records to be defined later in the program unit.

Syntax:

```
STRUCTURE /sname/
fdecl
  .
  .
  .
END STRUCTURE
```

Where:

`sname` is a symbolic name

`fdecl` has one of the following forms

```
data
substruc
union
```

`data` is any valid type statement: BYTE, INTEGER, REAL, DOUBLECOMPLEX, DOUBLEPRECISION, LOGICAL, COMPLEX, CHARACTER

`substruc` has the form

```
STRUCTURE [/sname/][flist]
fdecl
  .
  .
  .
END STRUCTURE
```

`flist` is a list of symbolic names

`union` has the form

```
UNION
mapdecl
  .
  .
  .
END UNION
```

`mapdecl` has the form

```
MAP
fdecl
  .
  .
  .
END MAP
```

Description:

The form of a record is defined by the multistatement STRUCTURE declaration. This declaration is composed of the STRUCTURE statement itself which provides a symbolic name for the STRUCTURE being defined, followed by the body of the declaration which is composed of one or more field declarations. The order of the declarations determines the order of the fields within a structure. The STRUCTURE declaration is terminated by an END STRUCTURE statement.

Field declarations within structure declarations consist of the following:

1. Typed data declaration statements which are simply standard FORTRAN type declarations. Fields can be any valid type and are dimensioned in the normal way. The only extension is that pseudo-name %FILL can be specified in place of a field name to create empty space in a record for purposes such as alignment. This creates an unnamed field.

2. Substructure declarations. A field within a structure can be a substructure composed of atomic fields and/or other substructures. These substructures are exactly like STRUCTURE declaration blocks except that they include a list of field names.

3. Union declarations. A union declaration declares groups of fields that logically share a common location within a structure. Each group of fields within a union declaration is declared by a map declaration, with one or more fields per map declaration. Union declarations use the same area of memory to alternately contain two or more groups of fields.

The names specified in these statements are not the names of variables and the statements in a structure declaration do not create variables. The names are field names, and the information provided in the statements describes the layout, or form, of the structure. The ordering of both the statements and the field names within the statements is important because this ordering determines the order of the fields in records.

See also:

The explicit TYPE statements BYTE, INTEGER, DOUBLECOMPLEX, DOUBLEPRECISION, REAL, LOGICAL, and CHARACTER which describe the actual meanings of the various types.

5.53 SUBROUTINE Statement

The SUBROUTINE statement introduces and specifies the name of the main entry point of a subroutine subprogram.

Syntax:

```
SUBROUTINE sub[([d[,d]...])]
```

Where:

`sub` is a symbolic name identifying the subroutine subprogram.

`d` is a dummy argument that can be a variable name, array name, dummy procedure name, or *.

Notes:

If there are no dummy arguments, either `sub` or `sub( )` can be used.

Description:

A subroutine subprogram is a procedure that communicates with the calling program unit either through a list of arguments passed with the CALL statement or through common blocks. A subroutine subprogram is executed when a CALL statement naming the subroutine is encountered in a program unit. Subroutines begin with a SUBROUTINE statement and end with an END statement. Control is returned to the calling program unit when a RETURN statement is encountered. If control flows into the END statement, then a RETURN is implied. Subroutines differ from functions in that the subprogram name is not used to return results to the calling program.

Subroutines can contain any statement except a PROGRAM, BLOCK DATA, FUNCTION, or another SUBROUTINE statement. A subroutine subprogram must not directly or indirectly call itself.

In a subroutine subprogram, the symbolic name of a dummy argument is unique to the program unit and must not appear in an EQUIVALENCE, PARAMETER, SAVE, INTRINSIC, DATA, or COMMON statement, except as a common block name. The dummy arguments are replaced with the actual arguments during a subroutine call. The SUBROUTINE statement can also have dummy arguments for statement labels; these arguments are represented by asterisks.

Dummy arguments that represent array names must be dimensioned by a DIMENSION or type statement. Adjustable dimensions are permitted in subroutine subprograms.

5.54 TYPE Statement

The TYPE statement transfers formatted information from the values named in the input/output list to the OUTPUT file.

Syntax:

```
TYPE fn [,iolist]
```

Where:

`fn` is a format specification

`iolist` is an output list

Notes:

The TYPE statement is a special case of the formatted WRITE statement.

Description:

The TYPE statement transfers formatted information from the values specified in the output item list to the OUTPUT file according to the specified format. This statement is identical to the statement

```
WRITE(*,fn) iolist
```

As a simple example consider

```
    PROGRAM TYPE
    CHARACTER B*3
    A=1.2
    B='YES'
    N=19
    TYPE 4,A,B,N
4   FORMAT (G20.6,A,I5)
    TYPE 50
50  FORMAT (' END OF FILE')
    STOP
    END
```

which will convert the values of `A`, `B`, and `N` into a coded record using format 4 and will then write that record to the OUTPUT file, followed by a record containing " END OF FILE".

See also:

The WRITE statement describes format specifications and output lists in detail. See that statement for details on these.

5.55 WRITE Statement

The WRITE statement transmits data from memory storage location or values of expressions to an external or internal unit.

Syntax:

```
WRITE fn[,olist]

WRITE *,[olist]
```

```
WRITE nl

WRITE([UNIT=u],[FMT=]fn [,REC=rn][,IOSTAT=ios][,ERR=sl][,END=sl]) [olist]

WRITE([UNIT=u],[FMT=]nl [,REC=rn][,IOSTAT=ios][,ERR=sl][,END=sl])

WRITE([UNIT=u],[FMT=]*[,IOSTAT=ios][,ERR=sl][,END=sl]) [olist]

WRITE([UNIT=]u[,IOSTAT=ios][,REC=rn][,ERR=sl][,END=sl] [olist]
```

Where:

`u`	is an integer expression or a character variable identifier or an asterisk
`fn`	is an integer variable, the label of a FORMAT statement, a character expression, or an array identifier
`rn`	is an integer expression
`nl`	is the identifier of a NAMELIST group
`ios`	is an integer lvalue
`sl`	is the label of an executable statement
`olist`	is an output item list

Notes:

The simplest form of the WRITE statement consists of a single format specification not enclosed in parentheses, possibly followed by an output item list. With this form no additional parameters can be specified. If one of the parenthetical forms of WRITE is used, then the "UNIT=" and "FMT=" specifications are optional; however, when omitted the unit specification must be the first specification and the `fn` specification must be second. Other than the above, the order of the parameters within the parenthetical versions of the WRITE statement is free.

Description:

The WRITE statement transfers output data from internal storage to records contained in external logical units or internal files. There are four basic types of WRITE statements:

(1) formatted write statements — those which contain an `fn` parameter;

(2) list directed write statements — those which contain an asterisk where an `fn` parameter would otherwise appear;

(3) namelist write statements — those which contain a NAMELIST group identifier where an `fn` parameter would otherwise appear;

(4) unformatted write statements — those which contain no `fn` parameter or asterisk or NAMELIST identifier in the `fn` position.

Formatted, list directed, and namelist write statements encode information into character records which may either be contained in external logical units or internal files. Unformatted write statements perform no encoding operations and may transfer information to external logical units only.

Formatted and unformatted write statements may operate with either sequential or direct access files, while list directed and namelist write statements may operate only with sequential files.

In general, the parameters on the various WRITE statements are as follows.

UNIT=u — is an integer expression, a character variable identifier, or an asterisk. If it is an integer expression, then it refers to a specific external logical unit. If it is a character variable, then that variable is treated as an internal file of character records. If it is an *, then the write is performed to the standard OUTPUT device. If one of the simple forms of the WRITE is used, which contain no parentheses and thus no unit number, then these also use the standard OUTPUT device.

FMT=fn — is integer variable, the label of a FORMAT statement, a character expression or an array identifier. It specifies a format to be used for formatted output. If `fn` is a statement label, then it identifies a FORMAT statement in the program unit containing the output statement, which specifies how the output records are to be formatted. If it is a character expression, then that expression specifies how the output records are to be formatted. If it is an array identifier, then that array is assumed to contain the format information. Finally, if it is an integer variable, then that variable is assumed to have been assigned the statement number of a FORMAT statement by an ASSIGN statement.

REC=rn — is an integer expression which specifies the number of the record to be written to the file. It must be greater than zero, and is valid for files opened for direct access only.

FMT=nl — is the identifier of a NAMELIST group. The use of this parameter indicates a namelist write. See the NAMELIST statement for a description of the conventions used for the namelist write.

ERR=sl — specifies the label of an executable statement to which control transfers if an error condition is encountered during output processing.

IOSTAT=ios — specifies an integer variable into which an error code is stored as follows: 0 means that the output operation completed normally, and a value greater than zero indicates some other error condition.

olist — is an output item list described below.

Regardless of the type of the write, if an error occurs as a result, and if neither ERR=`sl` nor IOSTAT=`ios` is supplied then execution will terminate with an error code set. If either or both are supplied, the execution continues despite any errors. If supplied the IOSTAT=`ios` lvalue receives the runtime error code for the error condition encountered or a zero, if no error occurred. If ERR=`sl` is supplied, then execution will branch to the statement `sl` if an error occurs.

The output item list portion of a WRITE statement specifies the items and/or values to be written and the order of transmission. The output item list can contain any number of items. List items are written sequentially from left to right. If no list appears one or more empty records are written.

A list item consists of a variable name, an array name, an array element name, a character substring name, an rvalue of any type, or an implied DO list. List items are separated by commas. Subscripts in an output item list can be written as any valid subscript form. An array name without subscripts in an output list specifies the entire array in the order in which it is stored. The entire array (not just the first word of the array) is written. Assumed-size array names are legal in output lists.

On formatted output, the `olist` is scanned and each item in the list is paired with the field specification provided by the FORMAT statement. After one item has been output, the next format specification is taken together with the next element of the list; and so on, until the end of the list.

An implied DO specification has the following form:

```
(dlist [,i=e1,e2 [,e3]])
```

The elements `i`, `e1`, `e2`, and `e3` have the same meaning as in the DO statement, and `dlist` is an output item list. Redundant parentheses are allowed and are ignored. The range of an implied DO specification is that of `dlist`. An implied DO can be used to output a single value more than one time. For example, the list `(A(K),B,K=1,5)` causes the value of variable `B` to be written five times.

Output of array elements can be accomplished by using an implied DO. The list of variables followed by the DO index is enclosed in parentheses to form a single element of the output item list. For example,

```
WRITE(5,100) (A(I),I=1,3)
```

has the same effect as the statement:

```
WRITE(5,100) A(1),A(2),A(3)
```

A variable cannot be used as a control variable more than once in the same implied DO nest, but `olist` items can appear more than once. The value of a control variable within an implied DO specification is defined within that specification. On exit from the implied DO specification the control variable retains the first value to exceed the upper limit `(e2).`

The implied DO can be nested: that is, the `ilist` in an implied DO can itself contain an implied DO. The first (innermost) control variable varies most rapidly, and the last (outermost) control variable varies least rapidly. For example, a nested implied DO with two levels has the form:

```
((list,v1=e1,e2,e3),v2=ee1,ee2,ee3)
```

Nested implied DO loops are executed in the same manner as nested DO statements. The nested form can be used to write from arrays.

Each execution of a WRITE statement writes at least one record. The unformatted WRITE statement transmits one record to the specified unit u from the storage locations or values named in `olist`. Records are not converted; no FORMAT statement is used. The information is transmitted to the designated file in the form in which it exists in the memory without any conversion.

The list directed WRITE statement writes data using a free-form notation, with items separated by commas. The actual format used varies from platform to platform. In all cases, list-directed output records can be read by the list-directed READ statement.

See also:

The discussion of the OPEN statement describes the FORTRAN file system in general, including the different file types. The discussion of the FORMAT statement describes how format information is interpreted. The NAMELIST statement describes the namelist read statement.

6. FORTRAN INTRINSIC FUNCTIONS

Certain procedures that are of general utility or that are difficult to express in FORTRAN statements are contained within the FORTRAN library. In general, these include the intrinsic functions and various subprograms that interface with different aspects of the operating system.

An intrinsic function is a procedure that performs a set of calculations when its name appears in an expression in the referencing program unit. Intrinsic functions communicate with the referencing program unit through a single value associated with the function symbolic name.

When the name of an intrinsic function duplicates another element in a program, the result depends on the element and the references. If a variable, array, or statement function is defined with the same name as an intrinsic function, the name is a local name that no longer refers to the intrinsic function. If an external function subprogram is written with the same name as an intrinsic function, use of the name references the intrinsic function, unless the name is declared as the name of an external function with the EXTERNAL statement.

Intrinsic functions are typed by default and need not appear in any explicit type statement in the program. Explicitly typing a generic intrinsic function name does not remove the generic properties of the name.

Certain intrinsic functions are generic, but have related specific functions. For example, the generic function name LOG computes the natural logarithm of an argument. Its argument can be real, double precision, or complex, and the type of the result is the same as the type of the argument. Specific function names ALOG, DLOG, and CLOG also compute the natural logarithm. The specific function name ALOG computes the log of a real argument and returns a real result. Likewise, the specific name DLOG is for double precision arguments and results, and the specific name CLOG is for complex arguments and results.

If a generic name and specific names exist, a generic name can be used in place of a specific name and is more flexible than a specific name. Except for type conversion generic functions, the type of the argument determines the type of the result.

Only a specific name can be used as an actual argument when passing the function name to a user-defined procedure or function.

The following table lists the intrinsic functions in alphabetical order by name.

Table 6-1. Intrinsic Functions

Name	Type	Arguments	Description
ABS	INTEGER*2	INTEGER*2	Absolute Value
	INTEGER	INTEGER	
	REAL	REAL	
	REAL*8	REAL*8	
	REAL	COMPLEX	
	REAL*8	COMPLEX*16	
ACOS	REAL*8	REAL*8	Arccosine
	REAL	REAL	
AIMAG	REAL	COMPLEX	Imaginary part
	REAL*8	COMPLEX*16	
AINT	REAL	REAL	Truncation
	REAL*8	REAL*8	
ALOG10	REAL	REAL	Logarithm base 10
ALOG	REAL	REAL	Natural logarithm

Table 6-1. Intrinsic Functions

Name	Type	Arguments	Description
AMAX0	REAL	INTEGER,...	Maximum value
AMAX1	REAL	REAL,...	Maximum value
AMIN0	REAL	INTEGER,...	Minimum value
AMIN1	REAL	REAL,...	Minimum value
AMOD	REAL	REAL,REAL	Remaindering
AND	LOGICAL*2	LOGICAL*2,...	Logical and
	LOGICAL	LOGICAL,...	
ANINT	REAL	REAL	Nearest whole number
	REAL*8	REAL*8	
ASIN	REAL	REAL	Arcsine
	REAL*8	REAL*8	
ATAN2	REAL	REAL,REAL	Arctangent of quotient
	REAL*8	REAL*8,REAL*8	
	REAL*8	REAL*8,REAL	
	REAL*8	REAL,REAL*8	
ATAN	REAL	REAL	Arctangent
	REAL*8	REAL*8	
CABS	REAL	COMPLEX	Absolute value
CCOS	COMPLEX	COMPLEX	Cosine
CDABS	REAL*8	COMPLEX*16	Absolute value
CDCOS	COMPLEX*16	COMPLEX*16	Cosine
CDEXP	COMPLEX*16	COMPLEX*16	Exponential
CDLOG10	COMPLEX*16	COMPLEX*16	Logarithm base 10
CDLOG	COMPLEX*16	COMPLEX*16	Natural logarithm
CDSIN	COMPLEX*16	COMPLEX*16	Sine
CDSQRT	COMPLEX*16	COMPLEX*16	Square root
CEXP	COMPLEX	COMPLEX	Exponential
CHAR	CHARACTER*1	NTEGER*2	Character value
	CHARACTER*1	INTEGER	
CLOG10	COMPLEX	COMPLEX	Logarithm base 10
CLOG	COMPLEX	COMPLEX	Natural logarithm
CMPLX	COMPLEX	INTEGER*2	Complex value
	COMPLEX	INTEGER	
	COMPLEX	REAL	
	COMPLEX	REAL*8	
	COMPLEX	COMPLEX	
	COMPLEX	COMPLEX*16	
	COMPLEX	REAL,REAL	
	COMPLEX	REAL*8,REAL*8	
	COMPLEX	REAL*8,REAL	
	COMPLEX	REAL,REAL*8	
CONJG	COMPLEX	COMPLEX	Conjugate
	COMPLEX*16	COMPLEX*16	
COSH	REAL	REAL	Hyperbolic cosine
	REAL*8	REAL*8	
COS	REAL	REAL	Cosine
	REAL*8	REAL*8	
	COMPLEX	COMPLEX	
	COMPLEX*16	COMPLEX*16	
CSIN	COMPLEX	COMPLEX	Sine
CSQRT	COMPLEX	COMPLEX	Square root
DABS	REAL*8	REAL*8	Absolute value
DACOS	REAL*8	REAL*8	Arccosine

Table 6-1. Intrinsic Functions

Name	Type	Arguments	Description
DASIN	REAL*8	REAL*8	Arcsin
DATE	---	CHARACTER*(*)	Current date
DATAN2	REAL*8	REAL*8,REAL*8	Arctangent of quotient
DATAN	REAL*8	REAL*8	Arctangent
DBLE	REAL*8	INTEGER*2	Double precision value
	REAL*8	INTEGER	
	REAL*8	REAL	
	REAL*8	REAL*8	
	REAL*8	COMPLEX	
	REAL*8	COMPLEX*16	
DCMPLX	COMPLEX*16	INTEGER*2	Double complex value
	COMPLEX*16	INTEGER	
	COMPLEX*16	REAL	
	COMPLEX*16	REAL*8	
	COMPLEX*16	COMPLEX*16	
	COMPLEX*16	COMPLEX	
	COMPLEX*16	REAL*8,REAL*8	
DCONJG	COMPLEX*16	COMPLEX*16	Conjugate
DCOSH	REAL*8	REAL*8	Hyperbolic cosine
DCOS	REAL*8	REAL*8	Cosine
DDIM	REAL*8	REAL*8,REAL*8	Positive difference
DEXP	REAL*8	REAL*8	Exponential
DIM	REAL	REAL,REAL	Positive difference
	REAL*8	REAL*8,REAL*8	
	REAL*8	REAL,REAL*8	
	REAL*8	REAL*8,REAL	
	INTEGER*2	INTEGER*2,INTEGER*2	
	INTEGER	INTEGER,INTEGER	
	INTEGER	INTEGER,INTEGER	
DIMAG	REAL*8	COMPLEX*16	Imaginary part
DINT	REAL*8	REAL*8	Truncation
DLOG10	REAL*8	REAL*8	Logarithm base 10
DLOG	REAL*8	REAL*8	Natural logarithm
DMAX1	REAL*8	REAL*8,...	Maximum value
DMIN1	REAL*8	REAL*8,...	Minimum value
DMOD	REAL*8	REAL*8,REAL*8	Remainder
DNINT	REAL*8	REAL*8	Nearest integer
DPROD	REAL*8	REAL,REAL	Product
DSIGN	REAL*8	REAL*8,REAL*8	Transfer of sign
DSINH	REAL*8	REAL*8	Hyperbolic sine
DSIN	REAL*8	REAL*8	Sine
DSQRT	REAL*8	REAL*8	Square root
DTANH	REAL*8	REAL*8	Hyperbolic tangent
DTAN	REAL*8	REAL*8	Tangent
EXIT	---	INTEGER	Stop program execution
EXP	REAL	REAL	Exponential
	REAL*8	REAL*8	
	COMPLEX	COMPLEX	
	COMPLEX*16	COMPLEX*16	
FLOAT	REAL	INTEGER*2	Real value
	REAL	INTEGER	
GETCL	---	CHARACTER*(*)	Get command line
IABS	INTEGER	INTEGER	Absolute value

Table 6-1. Intrinsic Functions

Name	Type	Arguments	Description
	INTEGER*2	INTEGER*2	
IAND	INTEGER*2	INTEGER*2,...	Bitwise and
	INTEGER	INTEGER,...	
ICHAR	INTEGER*2	CHARACTER*(*)	Integer value of character
IDIM	INTEGER	INTEGER,INTEGER	Positive difference
IDINT	INTEGER	REAL*8	Integer value
IDNINT	INTEGER	REAL	Nearest integer
IFIX	INTEGER	INTEGER*2	Integer value
	INTEGER	INTEGER	
	INTEGER	REAL	
	INTEGER	REAL*8	
	INTEGER	COMPLEX	
	INTEGER	COMPLEX*16	
INDEX	INTEGER	CHARACTER*(*),CHARACTER*(*)	Location of substring
INT2	INTEGER*2	INTEGER	Integer value
	INTEGER*2	INTEGER*2	
	INTEGER*2	REAL	
	INTEGER*2	REAL*8	
	INTEGER*2	COMPLEX	
	INTEGER*2	COMPLEX*16	
INT4	INTEGER	INTEGER*2	Integer value
	INTEGER	INTEGER	
	INTEGER	REAL	
	INTEGER	REAL*8	
	INTEGER	COMPLEX	
	INTEGER	COMPLEX*16	
INT	INTEGER	INTEGER*2	Integer value
	INTEGER	INTEGER	
	INTEGER	REAL	
	INTEGER	REAL*8	
	INTEGER	COMPLEX	
	INTEGER	COMPLEX*16	
ISIGN	INTEGER	INTEGER,INTEGER	Transfer of sign
I2ABS	INTEGER*2	INTEGER*2	Absolute value
I2DIM	INTEGER*2	INTEGER*2,INTEGER*2	Positive difference
I2MAX0	INTEGER*2	INTEGER*2,...	Maximum value
I2MIN0	INTEGER*2	INTEGER*2,...	Minimum value
I2MOD	INTEGER*2	INTEGER*2,INTEGER*2	Remainder
I2NINT	INTEGER*2	REAL	Nearest integer
I2SIGN	INTEGER*2	INTEGER*2,INTEGER*2	Transfer of sign
LEN	INTEGER	CHARACTER*(*)	Number of characters
LGE	LOGICAL*2	CHARACTER*(*),CHARACTER*(*)	Lexically greater or equal
LGT	LOGICAL*2	CHARACTER*(*),CHARACTER*(*)	Lexically greater
LLE	LOGICAL*2	CHARACTER*(*),CHARACTER*(*)	Lexically less or equal
LLT	LOGICAL*2	CHARACTER*(*),CHARACTER*(*)	Lexically less
LOG10	REAL	REAL	Logarithm base 10
	REAL*8	REAL*8	
	COMPLEX	COMPLEX	
	COMPLEX*16	COMPLEX*16)	
LOG	REAL	REAL	Natural log
	REAL*8	REAL*8	
	COMPLEX	COMPLEX	
	COMPLEX*16	COMPLEX*16	

Table 6-1. Intrinsic Functions

Name	Type	Arguments	Description
MAX0	INTEGER	INTEGER,...	Maximum value
	INTEGER*2	INTEGER*2,...	
MAX1	INTEGER	REAL,...	Maximum value
MAX	INTEGER*2	INTEGER*2,...	Maximum value
	INTEGER	INTEGER,...	
	REAL	REAL,...	
	REAL*8	REAL*8,...	
MIN0	INTEGER	INTEGER,...	Minimum value
	INTEGER*2	INTEGER*2,...	
MIN1	INTEGER	REAL,...	Minimum value
MIN	INTEGER*2	INTEGER*2,...	Minimum value
	INTEGER	INTEGER,...	
	REAL	REAL,...	
	REAL*8	REAL*8,...	
MOD	INTEGER*2	INTEGER*2,INTEGER*2	Remainder
	INTEGER	INTEGER,INTEGER	
	REAL	REAL,REAL	
	REAL*8	REAL*8,REAL*8	
	REAL*8	REAL,REAL*8	
	REAL*8	REAL*8,REAL	
	INTEGER	INTEGER,INTEGER	
NINT	INTEGER	INTEGER*2	Nearest integer
	INTEGER	INTEGER	
	INTEGER	REAL	
	INTEGER	REAL*8	
REAL	REAL	INTEGER*2	Real value or part
	REAL	INTEGER	
	REAL	REAL	
	REAL	REAL*8	
	REAL	COMPLEX	
	REAL	COMPLEX*16	
SIGN	NTEGER*2	INTEGER*2,INTEGER*2	Transfer of sign
	INTEGER	INTEGER,INTEGER	
	REAL	REAL,REAL	
	REAL*8	REAL*8,REAL*8	
	REAL*8	REAL,REAL*8	
	REAL*8	REAL*8,REAL	
SINH	REAL	REAL	Hyperbolic sine
	REAL*8	REAL*8	
SIN	REAL	REAL	Sine
	REAL*8	REAL*8	
	COMPLEX	COMPLEX	
	COMPLEX*16	COMPLEX*16	
SNGL	REAL	REAL*8	Real value
SQRT	REAL	REAL	Square root
	REAL*8	REAL*8	
	COMPLEX	COMPLEX	
	COMPLEX*16	COMPLEX*16	
TANH	REAL	REAL	Hyperbolic tangent
	REAL*8	REAL*8	
TAN	REAL	REAL	Tangent
	REAL*8	REAL	
TIME	---	CHARACTER*(*)	Current time

The description of each intrinsic function is given in the following sections.

6.1 ABS: Absolute Value

Synopsis:

```
INTEGER*2 FUNCTION ABS(INTEGER*2)

INTEGER FUNCTION ABS(INTEGER)

REAL FUNCTION ABS(REAL)

REAL*8 FUNCTION ABS(REAL*8)

REAL FUNCTION ABS(COMPLEX)

REAL*8 FUNCTION ABS(COMPLEX*16)
```

Description:

`ABS(a)` is a generic function that returns an absolute value of its argument. The result is integer, real, or double precision, depending on the argument type. For noncomplex arguments, the result is `a`, if `a` is greater than or equal to zero, else it is `-a`. For a complex argument, the result is the noncomplex square root of the sum of the squares of the two components of the complex number.

See Also:

IABS, DABS, CABS, CDABS and I2ABS.

6.2 ACOS: Arccosine

Synopsis:

```
REAL*8 FUNCTION ACOS(REAL*8)

REAL FUNCTION ACOS(REAL)
```

Description:

`ACOS(a)` is a generic function that returns an arccosine. The result is expressed in radians. The result is real or double precision, depending on the argument type.

See also:

DACOS

6.3 AIMAG: Imaginary Part

Synopsis:

```
REAL FUNCTION AIMAG(COMPLEX)

REAL*8 FUNCTION AIMAG(COMPLEX*16)
```

Description:

`AIMAG(a)` returns the imaginary part of a complex argument. The result is real if the argument is complex and double precision if the argument is double complex.

See also:

DIMAG

6.4 AINT: Truncation

Synopsis:

```
REAL FUNCTION AINT(REAL)

REAL*8 FUNCTION AINT(REAL*8)
```

Description:

`AINT(a)` is a generic function that returns an integer after truncation. The result is real or double precision. For a real or double precision argument, the result is 0 if the absolute value of `a` is less than 1. If the absolute value of `a` is greater than or equal to 1, the result is the largest integer with the same sign as argument `a` that does not exceed the magnitude of `a`.

See also:

DINT

6.5 ALOG10: Logarithm Base 10

Synopsis:

```
REAL FUNCTION ALOG10(REAL)
```

Description:

`ALOG10(a)` is a specific function that returns the logarithm base 10 of the argument. The argument is real and the result is real.

See also:

CDLOG10, CLOG10, DLOG10, and LOG10

6.6 ALOG: Natural Logarithm

Synopsis:

```
REAL FUNCTION ALOG(REAL)
```

Description:

`ALOG(a)` is a specific function that returns the natural logarithm of the argument. The argument is real and the result is real.

See also:

CDLOG, CLOG, DLOG, LOG

6.7 AMAX0: Maximum Value

Synopsis:

```
REAL FUNCTION AMAX0(INTEGER,...)
```

Description:

`AMAX0(a1,a2[,an]...)` is a specific function that returns the value of the largest argument. All arguments are integer, and the result is real.

See also:

AMAX1, MAX, DMAX1, I2MAX0, MAX0, and MAX1

6.8 AMAX1: Maximum Value

Synopsis:

```
REAL FUNCTION AMAX1(REAL,...)
```

Description:

`AMAX1(a1,a2[,an]...)` is a specific function that returns the value of the largest argument. All arguments are real, and the result is real.

See also:

AMAX0, MAX, DMAX1, I2MAX0, MAX0, and MAX1

6.9 AMIN0: Minimum Value

Synopsis:

```
REAL FUNCTION AMIN0(INTEGER,...)
```

Description:

`AMIN0(a1,a2[,an]...)` is a specific function that returns the value of the smallest argument. All arguments are integer, and the result is real.

See also:

AMIN1, DMIN1, I2MIN0, MIN0, MIN1, and MIN

6.10 AMIN1: Minimum Value

Synopsis:

```
REAL FUNCTION AMIN1(REAL,...)
```

Description:

`AMIN1(a1,a2[,an]...)` is a specific function that returns the value of the smallest argument. All arguments are real, and the result is real.

See also:

AMIN0, DMIN1, I2MIN0, MIN0, MIN1, and MIN

6.11 AMOD: Remaindering

Synopsis:

```
REAL FUNCTION AMOD(REAL,REAL)
```

Description:

`AMOD(a1,a2)` is a specific function that returns the remainder of `a1` divided by `a2`. Both arguments and the result are real. The result is

```
a1-(int(a1/a2)*a2).
```

If `a2` equals zero, then the result is zero.

See also:

DMOD, I2MOD, and MOD

6.12 AND: Logical And

Synopsis:

```
LOGICAL*2 FUNCTION AND(LOGICAL*2,...)

LOGICAL FUNCTION AND(LOGICAL,...)
```

Description:

`AND(a1,a2[,an]...)` is a generic function that returns the logical product of its arguments.

See also:

None

6.13 ANINT: Nearest Whole Number

Synopsis:

```
REAL FUNCTION ANINT(REAL)

REAL*8 FUNCTION ANINT(REAL*8)
```

Description:

`ANINT(a)` is a generic function that returns a real or double precision result from a real or double precision argument. It computes the nearest integer to its argument. In particular the result is

```
dint(a+0.5) if a >= 0.0
```

and

```
dint(a-0.5) if a <= 0.0
```

where `dint` is the FORTRAN intrinsic function `DINT`.

See also:

DNINT, IDNINT, I2NINT, and NINT

6.14 ASIN: Arcsine

Synopsis:

```
REAL FUNCTION ASIN(REAL)

REAL*8 FUNCTION ASIN(REAL*8)
```

Description:

`ASIN(a)` is a generic function that returns an arcsine. The result is expressed in radians. The result is real or double precision, depending on the argument type.

See also:

DASIN

6.15 ATAN2: Arctangent of Quotient

Synopsis:

```
REAL FUNCTION ATAN2(REAL,REAL)

REAL*8 FUNCTION ATAN2(REAL*8,REAL*8)

REAL*8 FUNCTION ATAN2(REAL*8,REAL)

REAL*8 FUNCTION ATAN2(REAL,REAL*8)
```

Description:

`ATAN2(a1,a2)` is a generic function that returns the arctangent of `a2` divided by `a1`. The result is expressed in radians. The result is real or double precision, depending on the type of the arguments. If both arguments are zero, the result is zero.

See also:

DATAN2

6.16 ATAN: Arctangent

Synopsis:

```
REAL FUNCTION ATAN(REAL)

REAL*8 FUNCTION ATAN(REAL*8)
```

Description:

`ATAN(a)` is a generic function that returns an arctangent. The result is expressed in radians. The result is real or double precision, depending on the argument type.

See also:

DATAN

6.17 CABS: Absolute Value

Synopsis:

```
REAL FUNCTION CABS(COMPLEX)
```

Description:

`CABS(a)` returns the real absolute value of its argument. The result is the noncomplex square root of the sum of the squares of the two components of the complex number.

See Also:

ABS, IABS, DABS, CDABS and I2ABS.

6.18 CCOS: Cosine

Synopsis:

```
COMPLEX FUNCTION CCOS(COMPLEX)
```

Description:

`CCOS(a)` is a specific function that returns a cosine. The argument is assumed to be in radians. Both the argument and the result are complex.

See also:

CDCOS, COS, and DCOS

6.19 CDABS: Absolute Value

Synopsis:

```
REAL*8 FUNCTION CDABS(COMPLEX*16)
```

Description:

`CDABS(a)` returns the absolute value of its argument. The result is the double precision square root of the sum of the squares of the two components of the complex number.

See Also:

ABS, IABS, DABS, CABS, and I2ABS.

6.20 CDCOS: Cosine

Synopsis:

```
COMPLEX*16 FUNCTION CDCOS(COMPLEX*16)
```

Description:

`CDCOS(a)` is a specific function that returns a cosine. The argument is assumed to be in radians. The result and argument are both double complex.

See also:

CCOS, COS, and DCOS

6.21 CDEXP: Exponential

Synopsis:

```
COMPLEX*16 FUNCTION CDEXP(COMPLEX*16)
```

Description:

`CDEXP(a)` is a specific function that returns a double complex result from a double complex argument. It returns the exponential of its argument.

See also:

CEXP, DEXP, and EXP

6.22 CDLOG10: Logarithm Base 10

Synopsis:

```
COMPLEX*16 FUNCTION CDLOG10(COMPLEX*16)
```

Description:

`CDLOG10(a)` is a specific function that returns the logarithm base 10 of the argument. The argument is double complex and the result is double complex.

See also:

ALOG10, CLOG10, DLOG10, and LOG10

6.23 CDLOG: Natural Logarithm

Synopsis:

```
COMPLEX*16 FUNCTION CDLOG(COMPLEX*16)
```

Description:

`CDLOG(a)` is a specific function that returns the natural logarithm of the argument. The argument is double complex and the result is double complex.

See also:

ALOG, CLOG, DLOG, and LOG

6.24 CDSIN: Sine

Synopsis:

```
COMPLEX*16 FUNCTION CDSIN(COMPLEX*16)
```

Description:

`CDSIN(a)` is a specific function that returns a sine. The argument is assumed to be in radians. The argument and result are both double complex.

See also:

CSIN, DSIN, and SIN

6.25 CDSQRT: Square Root

Synopsis:

```
COMPLEX*16 FUNCTION CDSQRT(COMPLEX*16)
```

Description:

`CDSQRT(a)` is a specific function that returns the principal square root of its argument. The argument and result are double complex.

See also:

CSQRT, DSQRT, and SQRT

6.26 CEXP: Exponential

Synopsis:

```
COMPLEX FUNCTION CEXP(COMPLEX)
```

Description:

`CEXP(a)` is a specific function that returns a complex result from a complex argument. It returns the exponential of its argument.

See also:

CDEXP, DEXP, and EXP

6.27 CHAR: Character Value

Synopsis:

```
CHARACTER*1 FUNCTION CHAR(INTEGER*2)

CHARACTER*1 FUNCTION CHAR(INTEGER)
```

Description:

`CHAR(a)` is a generic function that converts a numeric display code, or "lexical value" or "collating weight" into its character code. The point of this function is that character values on the host processor are not necessarily the same as those on the machine for which a given FORTRAN program was written. All numeric display code references in a source FORTRAN program are passed through this function either by the system directly or by the other runtime functions included in this library. Note that if you wish this function to return some value other than the host processor values then you must modify it. Typically, this modification would take the form of a lookup table reference.

See also:

ICHAR

6.28 CLOG10: Logarithm Base 10

Synopsis:

```
COMPLEX FUNCTION CLOG10(COMPLEX)
```

Description:

`CLOG10(a)` is a specific function that returns the logarithm base 10 of the argument. The argument is complex and the result is complex.

See also:

ALOG10, CDLOG10, DLOG10, and LOG10

6.29 CLOG: Natural Logarithm

Synopsis:

```
COMPLEX FUNCTION CLOG(COMPLEX)
```

Description:

`CLOG(a)` is a specific function that returns the natural logarithm of the argument. The argument is complex and the result is complex.

See also:

ALOG, CDLOG, DLOG, and LOG

6.30 CMPLX: Complex Value

Synopsis:

```
COMPLEX FUNCTION CMPLX(INTEGER*2)

COMPLEX FUNCTION CMPLX(INTEGER)

COMPLEX FUNCTION CMPLX(REAL)

COMPLEX FUNCTION CMPLX(REAL*8)

COMPLEX FUNCTION CMPLX(COMPLEX)

COMPLEX FUNCTION CMPLX(COMPLEX*16)

COMPLEX FUNCTION CMPLX(REAL,REAL)

COMPLEX FUNCTION CMPLX(REAL*8,REAL*8)

COMPLEX FUNCTION CMPLX(REAL*8,REAL)

COMPLEX FUNCTION CMPLX(REAL,REAL*8)
```

Description:

`CMPLX(a)` or `CMPLX(a1,a2)` is a generic function that performs type conversion and returns a complex value. CMPLX can have one or two arguments. A single argument can be integer, real, double precision, complex, or double complex. If two arguments are used, the arguments must be of the same type and must both be integer, real, or double precision. For a single integer, real, or double precision argument, the result is complex, with the argument used as the real part and the imaginary part zero. For a single complex argument, the result is the same as the argument. For two arguments `a1` and `a2`, the result is complex, with argument `a1` used as the real part and argument `a2` used as the imaginary part. CMPLX does not have specific names.

See also:

DCMPLX

6.31 CONJG: Conjugate

Synopsis:

```
COMPLEX FUNCTION CONJG(COMPLEX)

COMPLEX*16 FUNCTION CONJG(COMPLEX*16)
```

Description:

`CONJG(a)` is a generic function that returns a conjugate of a complex or double complex argument. The result is the same type as the argument. For a complex or double complex argument `(ar,ai),` the result is `(ar,-ai)` with the imaginary part negated.

See also:

DCONGJ

6.32 COSH: Hyperbolic Cosine

Synopsis:

```
REAL FUNCTION COSH(REAL)

REAL*8 FUNCTION COSH(REAL*8)
```

Description:

`COSH(a)` is a generic function that returns a hyperbolic cosine. The result is real or double precision, depending on the argument type.

See also:

DCOSH

6.33 COS: Cosine

Synopsis:

```
REAL FUNCTION COS(REAL)

REAL*8 FUNCTION COS(REAL*8)

COMPLEX FUNCTION COS(COMPLEX)

COMPLEX*16 FUNCTION COS(COMPLEX*16)
```

Description:

`COS(a)` is a generic function that returns a cosine. The argument is assumed to be in radians. The result is real, double precision, complex or double complex, depending on the argument type.

See also:

CCOS, CDCOS, and DCOS

6.34 CSIN: Sine

Synopsis:

```
COMPLEX FUNCTION CSIN(COMPLEX)
```

Description:

`CSIN(a)` is a specific function that returns a sine. The argument is assumed to be in radians. The argument and result are both complex.

See also:

CDSIN, DSIN, and SIN

6.35 CSQRT: Square Root

Synopsis:

```
COMPLEX FUNCTION CSQRT(COMPLEX)
```

Description:

`CSQRT(a)` is a specific function that returns the principal square root of its argument. The argument and result are complex.

See also:

CDSQRT, DSQRT, and SQRT

6.36 DABS: Absolute Value

Synopsis:

```
REAL*8 FUNCTION DABS(REAL*8)
```

Description:

`DABS(a)` returns the absolute value of its double precision argument. The result is double precision and equals `a`, if `a` is greater than or equal to zero, else it is `-a`.

See Also:

ABS, IABS, CABS, CDABS and I2ABS.

6.37 DACOS: Arccosine

Synopsis:

```
REAL*8 FUNCTION DACOS(REAL*8)
```

Description:

`DACOS(a)` returns the double precision arccosine of its double precision argument. The result is expressed in radians.

See also:

ACOS

6.38 DASIN: Arcsine

Synopsis:

```
REAL*8 FUNCTION DASIN(REAL*8)
```

Description:

`DASIN(a)` is a specific function that returns an arcsine. The result is expressed in radians. The result and argument are double precision.

See also:

ASIN

6.39 DATE: Current Date

Synopsis:

```
SUBROUTINE DATE(CHARACTER*(*))
```

Description:

`DATE(a)` obtains the current date as set within the system. The date is returned as an 8-character character string of the form `mm/dd/yy`. If the argument is too short to receive eight characters, the return value is truncated. If the argument is longer that eight, then trailing positions are padded with blanks.

See also:

TIME

6.40 DATAN2: Arctangent of Quotient

Synopsis:

```
REAL*8 FUNCTION DATAN2(REAL*8,REAL*8)
```

Description:

`DATAN2(a1,a2)` is a specific function that returns the arctangent of `a2` divided by `a1`. The result is expressed in radians. The result and argument are both double precision. If both arguments are zero, the result is zero.

See also:

ATAN2

6.41 DATAN: Arctangent

Synopsis:

```
REAL*8 FUNCTION DATAN(REAL*8)
```

Description:

`DATAN(a)` is a specific function that returns an arctangent. The result is expressed in radians. The result and argument are both double precision.

See also:

ATAN

6.42 DBLE: Double Precision Value

Synopsis:

```
REAL*8 FUNCTION DBLE(INTEGER*2)

REAL*8 FUNCTION DBLE(INTEGER)

REAL*8 FUNCTION DBLE(REAL)

REAL*8 FUNCTION DBLE(REAL*8)

REAL*8 FUNCTION DBLE(COMPLEX)

REAL*8 FUNCTION DBLE(COMPLEX*16)
```

Description:

`DBLE(a)` is a generic function that performs type conversion and returns a double precision result. The argument can be integer, real, double precision, or complex. For an integer or real argument, the result has as much precision as the double precision field can contain. For a double precision argument, the result is the argument For a complex argument, the real part is used, and the result has as much precision as the double precision field can contain.

See also:

REAL

6.43 DCMPLX: Double Complex Value

Synopsis:

```
COMPLEX*16 FUNCTION DCMPLX(INTEGER*2)

COMPLEX*16 FUNCTION DCMPLX(INTEGER)

COMPLEX*16 FUNCTION DCMPLX(REAL)

COMPLEX*16 FUNCTION DCMPLX(REAL*8)

COMPLEX*16 FUNCTION DCMPLX(COMPLEX*16)

COMPLEX*16 FUNCTION DCMPLX(COMPLEX)

COMPLEX*16 FUNCTION DCMPLX(REAL*8,REAL*8)
```

Description:

`DCMPLX(a)` or `DCMPLX(a1,a2)` is a generic function that performs type conversion and returns a double complex value. DCMPLX can have one or two arguments. A single argument can be integer, real, double precision, complex , or double complex. If two arguments are used, the arguments must be of the same type and must both be integer, real, or double precision. For a single integer, real, or double precision argument, the result is double complex, with the argument used as the real part and the imaginary part zero. For a single double complex argument, the result is the same as the argument. For two arguments `a1` and `a2`, the result is double complex, with argument `a1` used as the real part and argument `a2` used as the imaginary part. `DCMPLX` does not have specific names.

See also:

CMPLX

6.44 DCONJG: Conjugate

Synopsis:

```
COMPLEX*16 FUNCTION DCONJG(COMPLEX*16)
```

Description:

`DCONJG(a)` is a specific function that returns a conjugate of a double complex argument. The result is double complex. For a double complex argument `(ar,ai),` the result is `(ar,-ai)` with the imaginary part negated.

See also:

CONGJ

6.45 DCOSH: Hyperbolic Cosine

Synopsis:

```
REAL*8 FUNCTION DCOSH(REAL*8)
```

Description:

`DCOSH(a)` is a specific function that returns a hyperbolic cosine. The result and argument are double precision.

See also:

COSH

6.46 DCOS: Cosine

Synopsis:

```
REAL*8 FUNCTION DCOS(REAL*8)
```

Description:

`DCOS(a)` is a specific function that returns a cosine. The argument is assumed to be in radians and the result is in radians. The result and argument are both double precision.

See also:

CCOS, CDCOS, and DCOS

6.47 DDIM: Positive Difference

Synopsis:

```
REAL*8 FUNCTION DDIM(REAL*8,REAL*8)
```

Description:

`DDIM(a1,a2)` is a specific function that returns a double precision positive difference between its double precision arguments. If `a1` is greater that `a2` then the value of

```
a1 - a2
```

is returned, else zero is returned.

See also:

DIM, IDIM, and I2DIM

6.48 DEXP: Exponential

Synopsis:

```
REAL*8 FUNCTION DEXP(REAL*8)
```

Description:

`DEXP(a)` is a specific function that returns a double precision result from a double precision argument. It returns the exponential of its argument.

See also:

CDEXP, CEXP, and EXP

6.49 DIM: Positive Difference

Synopsis:

```
REAL FUNCTION DIM(REAL,REAL)

REAL*8 FUNCTION DIM(REAL*8,REAL*8)

REAL*8 FUNCTION DIM(REAL,REAL*8)

REAL*8 FUNCTION DIM(REAL*8,REAL)

INTEGER*2 FUNCTION DIM(INTEGER*2,INTEGER*2)

INTEGER FUNCTION DIM(INTEGER,INTEGER)
```

Description:

`DIM(a1,a2)` is a generic function that returns a positive difference. The result is integer, real, or double precision, depending on the type of the first argument. If `a1` is greater that `a2` then the value of

```
a1 - a2
```

is returned, else zero is returned.

See also:

DDIM, IDIM, and I2DIM

6.50 DIMAG: Imaginary Part

Synopsis:

```
REAL*8 FUNCTION DIMAG(COMPLEX*16)
```

Description:

`DIMAG(a)` is a specific function that returns the imaginary part of a double complex argument. The double precision result is `ai`, where the complex argument is `(ar,ai)`.

See also:

AIMAG

6.51 DINT: Truncation

Synopsis:

```
REAL*8 FUNCTION DINT(REAL*8)
```

Description:

`DINT(a)` returns a double precision integer value after the truncation of its double precision argument. The result is 0 if the absolute value of `a` is less than 1. If the absolute value of `a` is greater than or equal to 1, the result is the largest integer with the same sign as argument `a` that does not exceed the magnitude of `a`.

See also:

AINT

6.52 DLOG10: Logarithm Base 10

Synopsis:

```
REAL*8 FUNCTION DLOG10(REAL*8)
```

Description:

`DLOG10(a)` is a specific function that returns the logarithm base 10 of the argument. The argument is double precision and the result is double precision.

See also:

ALOG10, CDLOG10, CLOG10, and LOG10

6.53 DLOG: Natural Logarithm

Synopsis:

```
REAL*8 FUNCTION DLOG(REAL*8)
```

Description:

`DLOG(a)` is a specific function that returns the natural logarithm of the argument. The argument is double precision and the result is double precision.

See also:

ALOG, CDLOG, CLOG, and LOG

6.54 DMAX1: Maximum Value

Synopsis:

```
REAL*8 FUNCTION DMAX1(REAL*8,...)
```

Description:

`DMAX1(a1,a2[,an]...)` is a specific function that returns the value of the largest argument. All arguments are double precision, and the result is double precision.

See also:

AMAX0, AMAX1, MAX, I2MAX0, MAX0, and MAX1

6.55 DMIN1: Minimum Value

Synopsis:

```
REAL*8 FUNCTION DMIN1(REAL*8,...)
```

Description:

`DMIN1(a1,a2[,an]...)` is a specific function that returns the value of the smallest argument. All arguments are double precision, and the result is double precision.

See also:

AMIN0, AMIN1, I2MIN0, MIN0, MIN1, and MIN

6.56 DMOD: Remainder

Synopsis:

```
REAL*8 FUNCTION DMOD(REAL*8,REAL*8)
```

Description:

`DMOD(a1,a2)` is a specific function that returns the remainder of `a1` divided by `a2`. The result and both arguments are double precision. The result is

```
a1-(int(a1/a2)*a2).
```

If `a2` equals zero, then the result is zero.

See also:

AMOD, I2MOD, and MOD

6.57 DNINT: Nearest Integer

Synopsis:

```
REAL*8 FUNCTION DNINT(REAL*8)
```

Description:

`DNINT(a)` is a specific function that returns a double precision result from a double precision argument. It computes the nearest integer to its double precision argument. In particular the result is

```
dint(a+0.5) if a >= 0.0 and
dint(a-0.5) if a <= 0.0
```

where "`dint`" is the FORTRAN intrinsic function `DINT`.

See also:

ANINT, IDNINT, I2NINT, and NINT

6.58 DPROD: Product

Synopsis:

```
REAL*8 FUNCTION DPROD(REAL,REAL)
```

Description:

`DPROD(a1,a2)` is a specific function that returns a double precision product. The arguments are real, and the result is double precision. The result is `a1*a2`.

See also:

None

6.59 DSIGN: Transfer of Sign

Synopsis:

```
REAL*8 FUNCTION DSIGN(REAL*8,REAL*8)
```

Description:

`DSIGN(a1,a2)` is a specific function that returns a double precision result from two double precision arguments. The result reflects the magnitude of the first value combined with the sign of the second value. In particular

`if a1 > 0.0` and `a2 < 0.0` return `-a1`

or

`if a1 < 0.0` and `a2 > 0.0` return `-a1`

else return `a1`

See also:

ISIGN, I2SIGN, and SIGN.

6.60 DSINH: Hyperbolic Sine

Synopsis:

```
REAL*8 FUNCTION DSINH(REAL*8)
```

Description:

`DSINH(a)` is a specific function that returns a hyperbolic sine. The result and argument are double precision.

See also:

SINH

6.61 DSIN: Sine

Synopsis:

```
REAL*8 FUNCTION DSIN(REAL*8)
```

Description:

`DSIN(a)` is a specific function that returns a sine. The argument is assumed to be in radians. The argument and result are both double precision.

See also:

CSIN, CDSIN, and SIN

6.62 DSQRT: Square Root

Synopsis:

```
REAL*8 FUNCTION DSQRT(REAL*8)
```

Description:

`DSQRT(a)` is a specific function that returns the principal square root of its argument. The argument and result are double precision.

See also:

CDSQRT, CSQRT, and SQRT

6.63 DTANH: Hyperbolic Tangent

Synopsis:

```
REAL*8 FUNCTION DTANH(REAL*8)
```

Description:

`DTANH(a)` is a specific function that returns a hyperbolic tangent. The result and argument are double precision.

See Also:

TANH

6.64 DTAN: Tangent

Synopsis:

```
REAL*8 FUNCTION DTAN(REAL*8)
```

Description:

`DTAN(a)` is a specific function that returns a tangent. The argument is assumed to be in radians. The result and argument are double precision.

See also:

TAN

6.65 EXIT: Stop Program Execution

Synopsis:

```
SUBROUTINE EXIT(INTEGER)
```

Description:

A call to `EXIT(a)` stops program execution. It performs the identical operation as would be performed by executing the FORTRAN statement

```
STOP a
```

See also:

STOP statement description.

6.66 EXP: Exponential

Synopsis:

```
REAL FUNCTION EXP(REAL)

REAL*8 FUNCTION EXP(REAL*8)

COMPLEX FUNCTION EXP(COMPLEX)

COMPLEX*16 FUNCTION EXP(COMPLEX*16)
```

Description:

`EXP(a)` is a generic function that returns an exponential. The result is real, double precision, complex, or double complex depending on the argument type.

See also:

CDEXP, CEXP, and DEXP

6.67 FLOAT: Real Value

Synopsis:

```
REAL FUNCTION FLOAT(INTEGER*2)

REAL FUNCTION FLOAT(INTEGER)
```

Description:

`FLOAT(a)` is a specific function that returns a real result from an integer argument. It simply converts the argument to its real, floating point equivalent.

See also:

REAL and SNGL

6.68 GETCL: Get Command Line

Synopsis:

```
SUBROUTINE GETCL(CHARACTER*(*))
```

Description:

`GETCL(a)` returns any command line string entered along with the request to execute the program. If that string is shorter than the character length of `a`, then it is truncated. If the command line string is shorter than the character length of `a`, then `a` is padded with blanks.

See also: None

6.69 IABS: Absolute Value

Synopsis:

```
INTEGER FUNCTION IABS(INTEGER)

INTEGER*2 FUNCTION IABS(INTEGER*2)
```

Description:

`IABS(a)` is a generic function that returns an absolute value of its argument. The result is integer or short integer depending on the argument type. It is `a`, if `a` is greater than or equal to zero, else it is `-a`.

See Also:

ABS, DABS, CABS, CDABS and I2ABS.

6.70 IAND: Bitwise And

Synopsis:

```
INTEGER*2 FUNCTION IAND(INTEGER*2,...)

INTEGER FUNCTION IAND(INTEGER,...)
```

Description:

`IAND(a1,a2[,an]...)` is a generic function which accepts short integers or integers. The return value is the successive bitwise AND of its arguments, and is the same type as the first parameter.

See also:

None

6.71 ICHAR: Integer Value of Character

Synopsis:

```
INTEGER*2 FUNCTION ICHAR(CHARACTER*(*))
```

Description:

`ICHAR(a)` is a specific function that returns an integer value from a character argument. The value returned depends on the collating weight of the character in the collating sequence used.

See also:

CHAR

6.72 IDIM: Positive Difference

Synopsis:

```
INTEGER FUNCTION IDIM(INTEGER,INTEGER)
```

Description:

`IDIM(a1,a2)` is a specific function that returns an integer positive difference between its integer arguments. If `a1` is greater that `a2` then the value of

```
a1 - a2
```

is returned, else zero is returned.

See also:

DDIM, DIM, and I2DIM

6.73 IDINT: Integer Value

Synopsis:

```
INTEGER FUNCTION IDINT(REAL*8)
```

Description:

`IDINT(a)` is a specific function that performs conversion to integer, and returns an integer result from a double precision argument. The conversion to integer is performed via simple truncation.

See also:

IFIX, INT, INT2, and INT4

6.74 IDNINT: Nearest Integer

Synopsis:

```
INTEGER FUNCTION IDNINT(REAL)
```

Description:

`IDNINT(a)` is a specific function that returns an integer result from a real argument. It computes the nearest integer to its real argument. In particular the result is

```
int(a+0.5) if a >= 0.0
```

and

```
int(a-0.5) if a <= 0.0
```

where "int" is the FORTRAN intrinsic function INT.

See also:

ANINT, DNINT, I2NINT, and NINT

6.75 IFIX: Integer Value

Synopsis:

```
INTEGER FUNCTION IFIX(INTEGER*2)

INTEGER FUNCTION IFIX(INTEGER)

INTEGER FUNCTION IFIX(REAL)

INTEGER FUNCTION IFIX(REAL*8)

INTEGER FUNCTION IFIX(COMPLEX)

INTEGER FUNCTION IFIX(COMPLEX*16)
```

Description:

`IFIX(a)` is a generic function that performs conversion to integer, and returns an integer result. The conversion to integer for noncomplex arguments is performed via simple truncation. For complex arguments only the real part is used and truncated.

See also:

IDINT, INT, INT2, and INT4

6.76 INDEX: Location of Substring

Synopsis:

```
INTEGER FUNCTION INDEX(CHARACTER*(*),CHARACTER*(*))
```

Description:

`INDEX(a1,a2)` is a specific function that returns the location of a substring within a string. Both arguments must be character string arguments. If string `a2` occurs as a substring within string `a1`, the result is an integer indicating the starting position of the substring `a2` within `a1`. If `a2` does not occur as a substring within `a1`, the result is 0. If `a2` occurs as a substring more than once within `a1`, only the starting position of the first occurrence is returned.

See also:

None

6.77 INT2: Integer Value

Synopsis:

```
INTEGER*2 FUNCTION INT2(INTEGER)

INTEGER*2 FUNCTION INT2(INTEGER*2)

INTEGER*2 FUNCTION INT2(REAL)

INTEGER*2 FUNCTION INT2(REAL*8)

INTEGER*2 FUNCTION INT2(COMPLEX)

INTEGER*2 FUNCTION INT2(COMPLEX*16)
```

Description:

`INT2(a)` is a generic function that performs conversion to short integer, and returns a short integer result. The conversion to integer for noncomplex arguments is performed via simple truncation. For complex arguments only the real part is used and truncated.

See also:

IDINT, IFIX, INT, and INT4

6.78 INT4: Integer Value

Synopsis:

```
INTEGER FUNCTION INT4(INTEGER*2)

INTEGER FUNCTION INT4(INTEGER)

INTEGER FUNCTION INT4(REAL)

INTEGER FUNCTION INT4(REAL*8)

INTEGER FUNCTION INT4(COMPLEX)
```

```
INTEGER FUNCTION INT4(COMPLEX*16)
```

Description:

`INT4(a)` is a generic function that performs conversion to integer, and returns an integer result. The conversion to integer for noncomplex arguments is performed via simple truncation. For complex arguments only the real part is used and truncated.

See also:

IDINT, IFIX, INT, and INT2

6.79 INT: Integer Value

Synopsis:

```
INTEGER FUNCTION INT(INTEGER*2)

INTEGER FUNCTION INT(INTEGER)

INTEGER FUNCTION INT(REAL)

INTEGER FUNCTION INT(REAL*8)

INTEGER FUNCTION INT(COMPLEX)

INTEGER FUNCTION INT(COMPLEX*16)
```

Description:

`INT(a)` is a generic function that performs conversion to integer, and returns an integer result. The conversion to integer for noncomplex arguments is performed via simple truncation. For complex arguments only the real part is used and truncated.

See also:

IDINT, IFIX, INT2, and INT4

6.80 ISIGN: Transfer of Sign

Synopsis:

```
INTEGER FUNCTION ISIGN(INTEGER,INTEGER)
```

Description:

`ISIGN(a1,a2)` is a specific function that returns an integer result from two integers. The result reflects the magnitude of the first value combined with the sign of the second value. In particular

if `a1 > 0.0` and `a2 < 0.0` return `-a1`

or

if `a1 < 0.0` and `a2 > 0.0` return `-a1`
else return `a1`

See also:

DSIGN, I2SIGN, and SIGN.

6.81 I2ABS: Absolute Value

Synopsis:

```
INTEGER*2 FUNCTION I2ABS(INTEGER*2)
```

Description:

`I2ABS(a)` returns the short integer absolute value of its short integer argument. The result is `a`, if `a` is greater than or equal to zero, else it is `-a`.

See Also:

ABS, IABS, DABS, CABS, and CDABS

6.82 I2DIM: Positive Difference

Synopsis:

```
INTEGER*2 FUNCTION I2DIM(INTEGER*2,INTEGER*2)
```

Description:

`I2DIM(a1,a2)` is a specific function that returns a short integer positive difference between its short integer arguments. If `a1` is greater that `a2` then the value of

```
a1 - a2
```

is returned, else zero is returned.

See also:

DDIM, DIM, and IDIM

6.83 I2MAX0: Maximum Value

Synopsis:

```
INTEGER*2 FUNCTION I2MAX0(INTEGER*2,...)
```

Description:

`I2MAX0(a1,a2[,an]...)` is a specific function that returns the value of the largest argument. All arguments are short integer, and the result is short integer.

See also:

AMAX0, AMAX1, MAX, DMAX1, MAX0, and MAX1

6.84 I2MIN0: Minimum Value

Synopsis:

```
INTEGER*2 FUNCTION I2MIN0(INTEGER*2,...)
```

Description:

`I2MIN0(a1,a2[,an]...)` is a specific function that returns the value of the smallest argument. All arguments are short integer, and the result is short integer.

See also:

AMIN0, AMIN1, DMIN1, MIN0, MIN1, and MIN

6.85 I2MOD: Remainder

Synopsis:

```
INTEGER*2 FUNCTION I2MOD(INTEGER*2,INTEGER*2)
```

Description:

`I2MOD(num,dem)` computes the value of the remainder of `num` divided by `dem`. If `dem` is zero, then the result is zero. Both arguments are the result are short integer.

See also:

AMOD, DMOD, and MOD

6.86 I2NINT: Nearest Integer

Synopsis:

```
INTEGER*2 FUNCTION I2NINT(REAL)
```

Description:

`I2NINT(a)` is a specific function that returns a short integer result from a real argument. It computes the nearest integer to its real argument. In particular the result is

`int(a+0.5)` if `a >= 0.0`

and

`int(a-0.5)` if `a <= 0.0`

where `int` is the FORTRAN intrinsic function INT.

See also:

ANINT, DNINT, IDNINT, and NINT

6.87 I2SIGN: Transfer of Sign

Synopsis:

```
INTEGER*2 FUNCTION I2SIGN(INTEGER*2,INTEGER*2)
```

Description:

`I2SIGN(a1,a2)` is a specific function that returns a short integer result from two short integer arguments. The result reflects the magnitude of the first value combined with the sign of the second value. In particular

if `a1 > 0.0` and `a2 < 0.0` return `-a1`

or

if `a1 < 0.0` and `a2 > 0.0` return `-a1`
else return `a1`

See also:

DSIGN, ISIGN, and SIGN.

6.88 LEN: Number of Characters

Synopsis:

```
INTEGER FUNCTION LEN(CHARACTER*(*))
```

Description:

`LEN(a)` is a specific function that returns the length of a character string. The argument is a character string, and the result is an integer indicating the length of the string.

See also:

None

6.89 LGE: Lexically Greater or Equal

Synopsis:

```
LOGICAL*2 FUNCTION LGE(CHARACTER*(*),CHARACTER*(*))
```

Description:

`LGE(a1,a2)` is a specific function that returns a result indicating lexically greater than or equal to. The arguments are character strings. The result is true only if `a1` follows `a2`, or `a1` is equal to `a2`, in the collating sequence.

See also:

CHAR, LGT, LLE, and LLT

6.90 LGT: Lexically Greater

Synopsis:

```
LOGICAL*2 FUNCTION LGT(CHARACTER*(*),CHARACTER*(*))
```

Description:

`LGT(a1,a2)` is a specific function that returns a result indicating lexically greater than. The arguments are character strings. The result is true only if `a1` follows `a2` in the collating sequence.

See also:

CHAR, LGE, LLE, and LLT

6.91 LLE: Lexically Less or Equal

Synopsis:

```
LOGICAL*2 FUNCTION LLE(CHARACTER*(*),CHARACTER*(*))
```

Description:

`LLE(a1,a2)` is a specific function that returns a result indicating lexically less than or equal to. The arguments are character strings. The result is true only if `a1` precedes `a2`, or `a1` is equal to `a2`, in the collating sequence.

See also:

CHAR, LGE, LGT, and LLT

6.92 LLT: Lexically Less

Synopsis:

```
LOGICAL*2 FUNCTION LLT(CHARACTER*(*),CHARACTER*(*))
```

Description:

`LLT(a1,a2)` is a specific function that returns a result indicating lexically less than. The arguments are character strings. The result is true only if `a1` precedes `a2` in the collating sequence.

See also:

CHAR, LGE, LGT, and LLE

6.93 LOG10: Logarithm Base 10

Synopsis:

```
REAL FUNCTION LOG10(REAL)

REAL*8 FUNCTION LOG10(REAL*8)

COMPLEX FUNCTION LOG10(COMPLEX)

COMPLEX*16 FUNCTION LOG10(COMPLEX*16)
```

Description:

`LOG10(a)` is a specific function that returns the logarithm base 10 of the argument. The result is always the same type as the argument.

See also:

ALOG10, CDLOG10, CLOG10, and DLOG10

6.94 LOG: Natural Logarithm

Synopsis:

```
REAL FUNCTION LOG(REAL)

REAL*8 FUNCTION LOG(REAL*8)

COMPLEX FUNCTION LOG(COMPLEX)

COMPLEX*16 FUNCTION LOG(COMPLEX*16)
```

Description:

`LOG(a)` is a generic function that returns the natural logarithm of the argument. The result is always the same type as the argument.

See also:

ALOG, CDLOG, CLOG, and DLOG

6.95 MAX0: Maximum Value

Synopsis:

```
INTEGER FUNCTION MAX0(INTEGER,...)

INTEGER*2 FUNCTION MAX0(INTEGER*2,...)
```

Description:

`MAX0(a1,a2[,an]...)` is a generic function that returns the value of the largest argument. The arguments are integer, and the result is the same type as the first arguments.

See also:

AMAX0, AMAX1, MAX, DMAX1, I2MAX0, and MAX1

6.96 MAX1: Maximum Value

Synopsis:

```
INTEGER FUNCTION MAX1(REAL,...)
```

Description:

`AMAX0(a1,a2[,an]...)` is a specific function that returns the value of the largest argument. All arguments are real, and the result is integer.

See also:

AMAX0, AMAX1, MAX, DMAX1, I2MAX0, and MAX0

6.97 MAX: Maximum Value

Synopsis:

```
INTEGER*2 FUNCTION MAX(INTEGER*2,...)

INTEGER FUNCTION MAX(INTEGER,...)

REAL FUNCTION MAX(REAL,...)

REAL*8 FUNCTION MAX(REAL*8,...)
```

Description:

`MAX(a1,a2[,an]...)` is a generic function that returns the value of the largest argument. The result is the same type as its first argument.

See also:

AMAX0, AMAX1, DMAX1, I2MAX0, MAX0, and MAX1

6.98 MIN0: Minimum Value

Synopsis:

```
INTEGER FUNCTION MIN0(INTEGER,...)

INTEGER*2 FUNCTION MIN0(INTEGER*2,...)
```

Description:

`MIN0(a1,a2[,an]...)` is a generic function that returns the value of the smallest argument. All arguments are integer, and the result is the same type as its first argument.

See also:

AMIN0, AMIN1, DMIN1, I2MIN0, MIN1, and MIN

6.99 MIN1: Minimum Value

Synopsis:

```
INTEGER FUNCTION MIN1(REAL,...)
```

Description:

`MIN1(a1,a2[,an]...)` is a specific function that returns the value of the smallest argument. All arguments are real, and the result is integer.

See also:

AMIN0, AMIN1, DMIN1, I2MIN0, MIN0, and MIN

6.100 MIN: Minimum Value

Synopsis:

```
INTEGER*2 FUNCTION MIN(INTEGER*2,...)

INTEGER FUNCTION MIN(INTEGER,...)

REAL FUNCTION MIN(REAL,...)

REAL*8 FUNCTION MIN(REAL*8,...)
```

Description:

`MIN(a1,a2[,an]...)` is a generic function that returns the value of the smallest argument. The type of the result is the same as the type of the first argument.

See also:

AMIN0, AMIN1, DMIN1, I2MIN0, MIN0, and MIN1

6.101 MOD: Remainder

Synopsis:

```
INTEGER*2 FUNCTION MOD(INTEGER*2,INTEGER*2)

INTEGER FUNCTION MOD(INTEGER,INTEGER)

REAL FUNCTION MOD(REAL,REAL)

REAL*8 FUNCTION MOD(REAL*8,REAL*8)

REAL*8 FUNCTION MOD(REAL,REAL*8)

REAL*8 FUNCTION MOD(REAL*8,REAL)

INTEGER FUNCTION MOD(INTEGER,INTEGER)
```

Description:

`MOD(a1,a2)` is a generic function that returns the remainder of `a1` divided by `a2`. The result is integer, real, or double precision, depending on the argument type. The result is

```
a1-(int(a1/a2)*a2).
```

If `a2` equals zero, then the result is zero.

See also:

AMOD, DMOD, and I2MOD

6.102 NINT: Nearest Integer

Synopsis:

```
INTEGER FUNCTION NINT(INTEGER*2)

INTEGER FUNCTION NINT(INTEGER)

INTEGER FUNCTION NINT(REAL)

INTEGER FUNCTION NINT(REAL*8)
```

Description:

`NINT(a)` is a generic function that returns an integer result from an arithmetic, noncomplex argument. It computes the nearest integer to its argument. In particular the result is

`int(a+0.5)` if `a >= 0.0`

and

`int(a-0.5)` if `a <= 0.0`

where `int` is the FORTRAN intrinsic function INT.

See also:

ANINT, DNINT, IDNINT, and I2NINT

6.103 REAL: Real Value or Part

Synopsis:

```
REAL FUNCTION REAL(INTEGER*2)

REAL FUNCTION REAL(INTEGER)

REAL FUNCTION REAL(REAL)

REAL FUNCTION REAL(REAL*8)

REAL FUNCTION REAL(COMPLEX)

REAL FUNCTION REAL(COMPLEX*16)
```

Description:

`REAL(a)` is a generic function that performs type conversion and returns a real result. The argument can be integer, real, double precision, or complex. For a complex argument `(ar,ai)`, the result is `real(ar).`

See also:

FLOAT and SNGL

6.104 SIGN: Transfer of Sign

Synopsis:

```
INTEGER*2 FUNCTION SIGN(INTEGER*2,INTEGER*2)

INTEGER FUNCTION SIGN(INTEGER,INTEGER)

REAL FUNCTION SIGN(REAL,REAL)

REAL*8 FUNCTION SIGN(REAL*8,REAL*8)

REAL*8 FUNCTION SIGN(REAL,REAL*8)

REAL*8 FUNCTION SIGN(REAL*8,REAL)
```

Description:

`SIGN(a1,a2)` is a generic function whose result is the same type as its first argument. The result reflects the magnitude of the first value combined with the sign of the second value. In particular

if `a1` > 0.0 and `a2` < 0.0 return `-a1`

or

if `a1` < 0.0 and `a2` > 0.0 return `-a1`
else return `a1`

See also:

DSIGN, ISIGN, and I2SIGN.

6.105 SINH: Hyperbolic Sine

Synopsis:

```
REAL FUNCTION SINH(REAL)

REAL*8 FUNCTION SINH(REAL*8)
```

Description:

`SINH(a)` is a generic function that returns a hyperbolic sine. The result is real or double precision, depending on the argument type.

See also:

DSINH

6.106 SIN: Sine

Synopsis:

```
REAL FUNCTION SIN(REAL)

REAL*8 FUNCTION SIN(REAL*8)

COMPLEX FUNCTION SIN(COMPLEX)
```

```
COMPLEX*16 FUNCTION SIN(COMPLEX*16)
```

Description:

`SIN(a)` is a generic function that returns a sine. The argument is assumed to be in radians. The result is real, double precision, complex, or double complex depending on the argument type.

See also:

CDSIN, CSIN, and DSIN

6.107 SNGL: Real Value

Synopsis:

```
REAL FUNCTION SNGL(REAL*8)
```

Description:

`SNGL(a)` is a specific function that returns a real result from a double precision argument. The result is the double precision value converted to real.

See also:

FLOAT and REAL

6.108 SQRT: Square Root

Synopsis:

```
REAL FUNCTION SQRT(REAL)

REAL*8 FUNCTION SQRT(REAL*8)

COMPLEX FUNCTION SQRT(COMPLEX)

COMPLEX*16 FUNCTION SQRT(COMPLEX*16)
```

Description:

`SQRT(a)` is a generic function that returns the principal square root of its argument. The result is real, double precision, complex, or double complex depending on the argument type.

See also:

CDSQRT, CSQRT, and DSQRT

6.109 TANH: Hyperbolic Tangent

Synopsis:

```
REAL FUNCTION TANH(REAL)

REAL*8 FUNCTION TANH(REAL*8)
```

Description:

`TANH(a)` is a generic function that returns a hyperbolic tangent. The result is real or double precision, depending on the argument type.

See also:

DTANH

6.110 TAN: Tangent

Synopsis:

```
REAL FUNCTION TAN(REAL)

REAL*8 FUNCTION TAN(REAL*8)
```

Description:

`TAN(a)` is a generic function that returns a tangent. The argument is assumed to be in radians. The result is real or double precision, depending on the argument type.

See also:

DTAN

6.111 TIME: Current Time

Synopsis:

```
SUBROUTINE TIME(CHARACTER*(*))
```

Description:

`TIME(a)` obtains the current time as set within the system. The time is returned as an 8-character character string of the form `hh:mm:ss`. If the argument is too short to receive eight characters, the return value is truncated. If the argument is longer that eight, then trailing positions are padded with blanks.

See Also:

DATE\

7. CONTROLLING RUNTIME BEHAVIOR

Once an application has been processed by the compiler and converted into an executable form, its runtime behavior can be controlled by passing execution control switches to the runtime system via the command line for the application.

These include the following:

Switch	Description of Use
Ccode	Establishes the runtime conventions code: (Codes may be summed to achieve composite effects) Code — Meaning 0 — Standard conventions 2 — Interprets carriage control to output 4 — Checks substring lengths for overflow 8 — Execute an explicit pause for the PAUSE statement 16 — Use VAX FORTRAN runtime conventions
Inumber	Assigns the unit whose number is indicated to standard input.
Onumber	Assigns the unit whose number is indicated to standard output.
Tnumber	Assigns the unit whose number is indicated to be a terminal — i.e., reads from standard input, writes to standard output
Vname	Specifies the name of a file which is to be used as the virtual disk file.
Svalue	Specifies that the file specified by the V parameter is to be created during the execution of this program.
Zvalue	Specifies the maximum number of sheets to be allocated for virtual memory.

7.1 Interpreting Carriage Control to Output

In traditional FORTRAN environments, the first character of each character record written contains a format control:

blank means single space;

zero means double space;

one means start a new page.

By default, the runtime library simply writes these characters to coded records without checking or interpreting them.

If the carriage control convention flag is set, however, the runtime library strips the first character and replaces it with the appropriate control characters.

7.2 Checking Substring Lengths for Overflow

Normally, substrings are not checked to make certain that they do not overflow the strings from which they are taken. This convention provides for rapid execution of substring statements and conforms to the normal FORTRAN conventions which allow the user to play memory games with substrings — i.e., many real FORTRAN programs contain deliberate substring overflows.

Alternatively, if careful programming conventions are being stressed, it may be desirable to check for substring overflows at runtime. This flag provides this facility.

7.3 Executing an Explicit PAUSE

In traditional FORTRAN environments, the PAUSE statement executed by jobs running in the background sent a message to the operator console and then suspended execution until a "go" was received from the operator. By default the runtime library now treats these PAUSE statements as merely displaying informational messages to the console — execution is not physically suspended.

The explicit PAUSE flag causes the application to pause until the Enter or Return key is pressed, or until a record is read from the standard input file. Since the concept of an "operator's console" is no longer well-defined for jobs running in the background, care should be used with this flag. It may cause unexpected results.

7.4 Using VAX FORTRAN Runtime Conventions

The VAX runtime conventions, though they are technically compliant with the 1977 FORTRAN standard, are somewhat different from normal conventions. Most of these differences can be dealt with at source code processing time; however, one cannot.

Under VAX runtime conventions, file record lengths are expressed in long words rather than in bytes. The VAX FORTRAN runtime convention flag informs the runtime system that lengths are to be multiplied times the size of a long word before they are used.

7.5 Assigning a Standard Input Unit

Some FORTRAN runtime systems assume that a certain unit is to be assigned to the standard input file. The Inumber command line switch assigns the unit number specified to the standard input file.

7.6 Assigning a Standard Output Unit

Some FORTRAN runtime systems assume that a certain unit is to be assigned to the standard output file. The Onumber command line switch assigns the unit number specified to the standard output file.

7.7 Assigning a Standard Terminal Unit

Some FORTRAN runtime systems assume that a certain unit is to be assigned to standard input for reads and to standard output for writes. Such files are referred to as "terminal" files. The Tnumber command line switch assigns the unit number specified to the terminal file.

7.8 Specifying a Virtual Filename

The Vname command line switch specifies the name of a file which is to be used as the virtual disk file. This may be an existing PROMULA array datafile or it may be a file to be created during the execution of the application program.

7.9 Specifying a Virtual File Size

The Svalue command line switch specifies that the file whose name is specified by the Vvalue parameter is to be created during the execution of this program. The parameter "value" specifies the size of that file in bytes. If the S switch is used with no corresponding name specified, then the name "pfc.dba" is used. Note that the value to be used with this switch is displayed by the compiler when an application requiring virtual memory is processed.

7.10 Specifying a Virtual Sheet Count

The Zvalue command line switch specifies the maximum number of sheets to be allocated for virtual memory. This parameter is needed when both dynamic and virtual allocations are being made. It prevents the virtual manager from exhausting all memory before the dynamic manager has a chance to satisfy its memory needs. The maximum setting for this parameter is 254.

8. THE PROMULA INTERFACE

The PROMULA FORTRAN compiler converts FORTRAN source codes into executable code. The PROMULA Application Management System, henceforth PROMULA, is an integrated programming environment which is designed to use a data structure which is completely compatible with the one used by FORTRAN. Based on multi-dimensional arrays (dimensioned variables), the PROMULA data structure is a generalization of the FORTRAN data structure. The PROMULA language itself is an elegant notation for managing and manipulating such arrays. Using the two tools together -- the FORTRAN compiler and PROMULA — can add significant value to FORTRAN applications.

A typical program-user interface for 'old' batch-oriented FORTRAN programs is shown below. The user manipulates what goes in and what comes out of the program via a text editor. The program itself is a computational 'black box' whose logic transforms the program inputs into the program outputs. The instructions controlling the running of the program are usually in a batch procedure of commands to the operating system — written in JCL, the Job Control Language of the operating system.

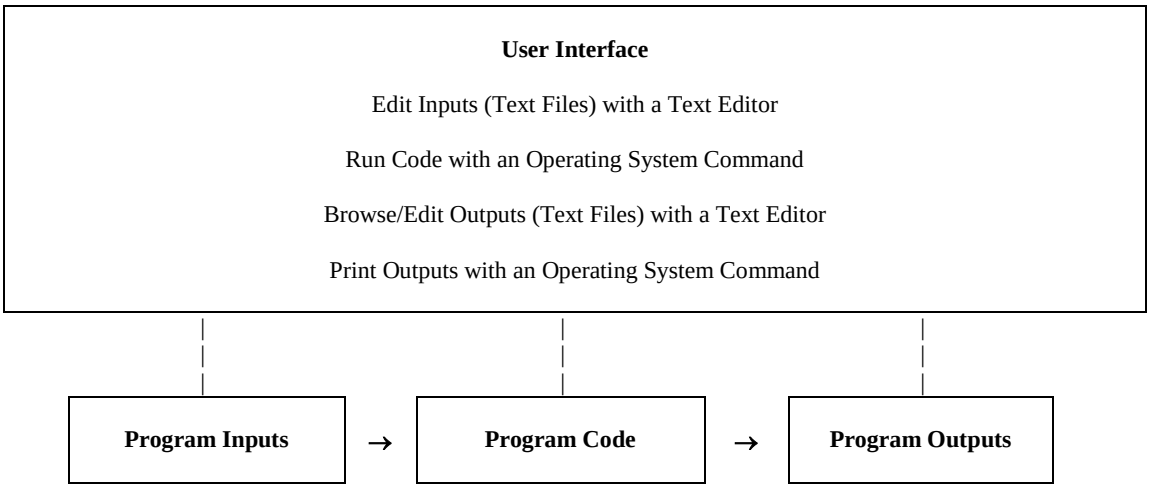

Figure 8-1. A Typical FORTRAN Program-User Interface

In this scheme, the program's logic and the program's information are "locked in" the program code. Typically, the program's inputs and outputs are separate text files that can be accessed sequentially by the program and can be manipulated by the user via a text editor. How do you make this interface more user friendly? Can you do it without diminishing the current functionality of the program? Can you add value and functionality without modifying the current code? The answer is yes. The PROMULA tools make it possible to add value to existing FORTRAN programs. The purpose of this chapter is to show you how.

A particularly difficult problem faces anyone porting an existing program to a PC or some other contemporary platform. Traditional programs, though perfectly useful, are rejected by contemporary users because they are not sufficiently user friendly. In designing the FORTRAN compiler, we felt very strongly that this particular problem could not be ignored. Simple compilation was not enough. Our solution to this problem is to automatically link the "disk image" used by the virtual memory manager directly into our PROMULA system, and to allow the compiler to use virtual memory for selected variables.

At issue is more than just the user interface. There are myriad tools available on the PC; spreadsheets, database managers, graphics packages, statistical packages, etc., which can be used to work with the data now locked up in existing FORTRAN

programs. Once the information can be gotten to disk in a structured form, all of these can be used to enhance the value of that information.

Traditional techniques allow you to work only with the explicit information either read or written by the program. Working with the information in the program simply means working with its input files and its output reports. If this is not sufficient, you must go into the program and modify it. This required modification is often time consuming and can introduce errors. The technique we are presenting here requires virtually no changes in the FORTRAN code. The FORTRAN compiler puts into the executable the logic needed to place values on the disk at the point of their creation or use. This logic introduces no errors and requires no changes in the FORTRAN source code.

Here we show through an example how the PROMULA FORTRAN Compiler can add value to an existing FORTRAN program by making its information directly accessible to the PROMULA system. If you do not presently use PROMULA itself, this will show you how the use of an "integrated system" can greatly simplify the use of, and even breathe life into, old, batch-oriented FORTRAN programs.

With PROMULA, it is possible to enhance the user interface by creating a database that contains the information content of the FORTRAN program — what goes in and what comes out as well as intermediate results.

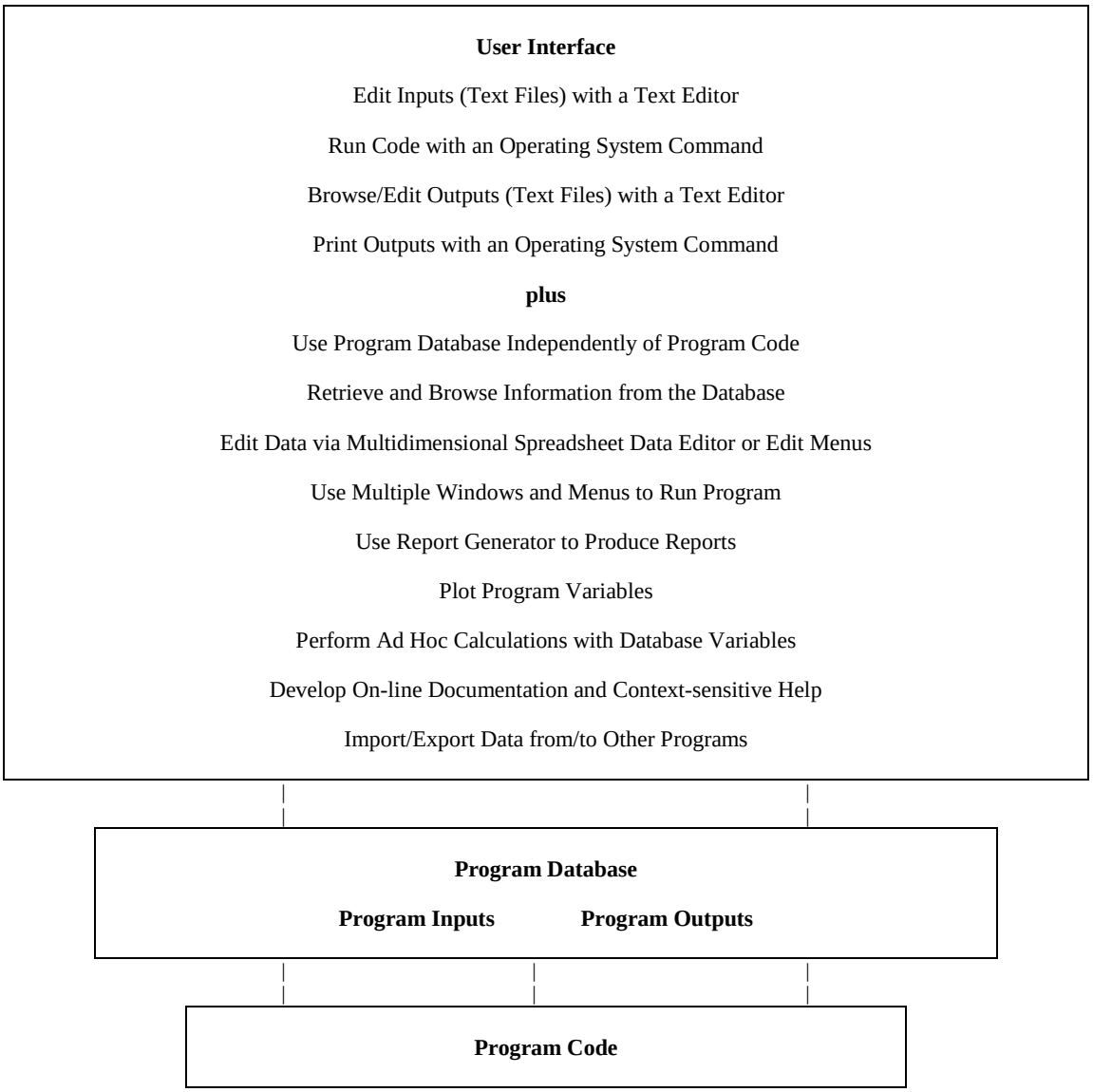

Figure 8.2. A PROMULA-Enhanced FORTRAN Program-User Interface

8.1 Transforming a FORTRAN Program with PROMULA

PROMULA and the FORTRAN compiler can be used in tandem to upgrade existing FORTRAN codes. The interface between these two systems truly adds value to existing FORTRAN programs.

Traditionally, FORTRAN programs are computational 'boxes'; they read in data and write out reports. By making the internal data structure of a FORTRAN program a PROMULA dataset, you are able to 'unlock' the information content of the FORTRAN box and make it immediately accessible to users via a database and a structured, user-friendly interface.

Consider the following scenario. You have been given the assignment to port a useful, but large, FORTRAN program from the company mainframe to the PC or some other desktop platform. The program runs on a mainframe, requires about 8 Megabytes of system memory, and has a batch user interface — its inputs are fixed-format, sequential text files and so are its outputs. Your mission is to run this program on the desktop without losing any of its functionality, reproduce its behavior and its results, and make it more user friendly.

With PROMULA, this portation project will go like this:

(1) Make a list of all the program variables that you wish to deposit in a random-access database for later use by the program and for independent use by other applications. Write a database description file (in PROMULA) and create a PROMULA database that will contain the selected variables. Each variable in the database is simply defined by an identifier, a structure (its dimensions, if any), a type, and a descriptor. The dimensionality of variables in the database must be identical to that in the program. For example, the definitions:

```
DEFINE SET
      week(52)
      day(7)
      hour(24)
      year(10)
END

DEFINE VARIABLE
      hdata(hour,day,week,year) TYPE=REAL(10,2), "Hourly Data"
END
```

describe a four-dimensional variable, `hdata`, containing 87,360 values (= 52 x 7 x 24 x 10), the hourly values for 10 years classified by hour, day, week, and year. On PC-DOS, this variable alone is bigger that 64 Kilobytes and would make it difficult for the program to compile using a typical compiler available on the PC.

(2) Compile and link the FORTRAN program. During the compilaton introduce references to a PROMULA database containing all the variables in the program that cause it to be big — larger than the typical memory resources available on, say, the PC.

The augmented compilation effectively separates the program's calculations from its database. The database can now reside on disk, can be as large as we wish, and still be accessible to the program via the PROMULA virtual memory manager. The program database is also accessible to PROMULA for independent data management, manipulation, and ad hoc analysis.

The same code that runs on the mainframe now runs on the desktop as well and reproduces results without making any code changes or sacrificing any of the program's capabilities. There is no need to 'downsize' the code to run it on the PC. As a result, there is no need to maintain two different codes, one on the mainframe and one on the PC.

(3) Write a PROMULA user-interface program to manage the program database and to perform the following functions:

- Browse/edit program inputs
- Edit the program source code
- Translate, compile and link the program source code
- Run the executable model code
- Browse or print program outputs
- Plot program inputs and outputs

PROMULA's built-in windowing, menu management, data management, and graphics features are used to produce a contemporary user interface. The programming requirements for this kind of program are very modest; and once you've done one, the code is transferrable to other applications as well.

8.2 EXPO, an Exposure Analysis Model

The program listing on the following two pages shows a relatively simple FORTRAN program which calculates the cumulative effects of exposure to various pollutant types. It is basically a FORTRAN 66 program, though it has OPEN statements and uses "T" FORMAT specifications. Both of these are extensions to the 66 dialect. Characters are used, but are stored in numeric variables; therefore, this program would not be usable with a pure FORTRAN 77 compiler. Except for this presentation, this program has no other purpose.

```
      PROGRAM EXPO
C
C THIS PROGRAM COMPUTES EXPOSURE LEVELS GIVEN INPUT TIME SERIES
C
      INTEGER I,IDAY(50),IMNTH(50),IMOD,IOPT,IYR(50),LUN1,LUN2,
     +       MASS(2),NUM,TIME(2),TITLE(20),XLN(2)
      REAL ABS,AVGE(50),AVGI(50),CONC(50),CUME(50),CUMEV,CUMI(50),
     +       CUMIV,DUR,EXPOS(50),FREQ,PERD(50),SEVER(50),TAKE(50),VBL,
     +       XMULT
      LUN1=5
      LUN2=6
      OPEN(LUN1,FILE='EXPO.DAT',STATUS='OLD')
      OPEN(LUN2,FILE='EXPO.RPT',STATUS='NEW')
C
C INITIALIZE THE CUMULATIVE VALUES
C
      CUMEV = 0.0
      CUMIV = 0.0
C
C READ INPUT CONTROLS AND WRITE OUTPUT HEADERS
C
      READ(LUN1,1000) (TITLE(I),I=1,20)
      READ(LUN1,1005) (MASS(I),I=1,2),(XLN(I),I=1,2),(TIME(I),I=1,2)
      READ(LUN1,1010) IMOD,IOPT
      READ(LUN1,1020) ABS,XMULT
      WRITE(LUN2,2007)
      WRITE(LUN2,2000) (TITLE(I),I=1,20),(MASS(I),I=1,2),
     1                 (XLN(I),I=1,2),(TIME(I),I=1,2)
C
C DETERMINE WHETHER FREQUENCY AND DURATION WILL BE READ IN TIMESERIES
C
      IF(IOPT.EQ.0) READ(LUN1,1020) VBL,FREQ,DUR
      IF(IOPT.EQ.1) READ(LUN1,1020) VBL
C
C WRITE OUTPUT HEADER FOR SELECTED EXPOSURE ROUTE
```

```
C
       WRITE(LUN2,2001)
       WRITE(LUN2,2010)
       WRITE(LUN2,2011)
       WRITE(LUN2,2020)
C
C READ NUMBER OF ENTRIES IN TIMESERIES
C
       READ(LUN1,1010) NUM
       DO 30 I=1,NUM
          IF(IOPT.EQ.0) READ(LUN1,1030) IDAY(I),IMNTH(I),IYR(I),
     +           CONC(I)
          IF(IOPT.EQ.1) READ(LUN1,1030) IDAY(I),IMNTH(I),IYR(I),
     +           CONC(I),FREQ,DUR
          IF(IOPT.EQ.2) READ(LUN1,1030) IDAY(I),IMNTH(I),IYR(I),
     +           CONC(I),VBL,FREQ,DUR
```

(Continued on next page)

```
C
C CONVERT UNITS OF INPUT CONCENTRATION
C
            CONC(I) = CONC(I) * XMULT
C
C COMPUTE SEVERITY, PERIODICITY, EXPOSURE AND INTAKE
C
            SEVER(I) = CONC(I) * VBL
            PERD(I) = FREQ * DUR
            EXPOS(I) = SEVER(I) * PERD(I)
            TAKE(I) = EXPOS(I) * ABS
C
C COMPUTE CUMULATIVE AND AVERAGE VALUES
C
            CUMEV = CUMEV + EXPOS(I)
            CUME(I) = CUMEV
            AVGE(I) = CUMEV / I
            CUMIV = CUMIV + TAKE(I)
            CUMI(I) = CUMIV
            AVGI(I) = CUMIV / I
            WRITE(LUN2,2030) IDAY(I),IMNTH(I),IYR(I),CONC(I),
      +   EXPOS(I),CUME(I),AVGE(I),TAKE(I),CUMI(I),AVGI(I)
    30 CONTINUE
  1000 FORMAT(20A4)
  1005 FORMAT(3(2A4,2X))
  1010 FORMAT(BN,3I5)
  1020 FORMAT(8F10.0)
  1030 FORMAT(3I2,4X,F6.0)
  2000 FORMAT(T2,20A4,//,
     1    T2,'MASS (M) UNITS = ',2A4,/,
     2    T2,'LENGTH (L) UNITS = ',2A4,/,
     3    T2,'TIME (T) UNITS = ',2A4,//)
       2001 FORMAT(T45,'INHALATION EXPOSURE ROUTE',/,
     1    T45,'-------------------------',/)
  2007 FORMAT(T2,'************************************',/
     1    T2,'*      EXPOSURE ANALYSIS MODEL       *',/
     2    T2,'*            VERSION 1               *',/
     3    T2,'************************************',///)
  2010 FORMAT(T26,'AMBIENT',T54,'CUMULATIVE',T70,'AVERAGE',
     1    T99,'CUMULATIVE',T116,'AVERAGE',/,T2,'DAY',
     2    T8,'MONTH',T16,'YEAR',T23,'CONCENTRATION',
     3    T40,'EXPOSURE',T55,'EXPOSURE',T70,'EXPOSURE',
     4    T86,'INTAKE',T101,'INTAKE',T116,'INTAKE',/)
  2011 FORMAT(T25,'(M/L**3)',T41,'(M/T)',T57,'(M)',T71,'(M/T)',
```

```
     1    T86,'(M/T)',T102,'(M)',T117,'(M/T)')
  2020 FORMAT(T2,'-----------------',T23,'-------------',
     1    T40,'---------------------------------------',
     2    T86,'---------------------------------------',/)
  2030 FORMAT(2X,I2,T9,I2,T17,I2,T25,G9.3,T40,G9.3,T55,G9.3,T70,G9.3,
     1    T85,G9.3,T100,G9.3,T115,G9.3)
       STOP
       END
```

8.3 The Initial Compilation

The filename for this program is "EXPO.FOR". We compile it by simply entering the following at the operating system command level:

```
pf expo.for -o expo
```

This produces an executable file. When we run this program, it reads the input data from an input text file, does some calculations, and writes the output report on an output text file:

EXPO.DAT → expo → EXPO.RPT

The input and output datafiles for this program, EXPO.DAT and EXPO.RPT, are shown in Figures 8-3 and 8-4, respectively.

```
RESIDENTIAL AREA, INHALATION, ADULT/CHILE, 14 DAY, 50 PPM, 100M DOWNWIND
UG        METER     MONTH
1    0
1.0       1.0
576.7     1.0       1.0
17
010879    4.6E-1
010979    5.2E-1
011079    3.8E-2
011179    1.7E-2
011279    1.1E-2
010180    6.8E-3
010280    4.8E-3
010380    1.0E-2
010480    2.4E-2
010580    5.0E-2
010680    6.5E-2
010780    5.4E-2
010880    1.2E-4
010980    6.7E-5
011080    2.5E-5
011180    1.4E-5
011280    6.1E-6
```

Figure 8-3. Input Data for EXPO

```
************************************
*     EXPOSURE ANALYSIS MODEL      *
*     VERSION 1                    *
************************************

RESIDENTIAL AREA, INHALATION, ADULT/CHILD, 14 DAY, 50 PPM, 100M DOWNWIND

MASS (M) UNITS = UG
LENGTH (L) UNITS = METER
TIME (T) UNITS = MONTH

                                                    INHALATION EXPOSURE ROUTE
```

DAY	MONTH	YEAR	AMBIENT CONCENTRATION (M/L**3)	EXPOSURE (M/T)	CUMULATIVE EXPOSURE (M)	AVERAGE EXPOSURE (M/T)	INTAKE (M/T)	CUMULATIVE INTAKE (M)	AVERAGE INTAKE (M/T)
1	8	79	.460	265.	265.	265.	265.	265.	265.
1	9	79	.520	300.	565.	283.	300.	565.	283.
1	10	79	.380E-01	21.9	587.	196.	21.9	587.	196.
1	11	79	.170E-01	9.80	597.	149.	9.80	597.	149.
1	12	79	.110E-01	6.34	603.	121.	6.34	603.	121.
1	1	80	.680E-02	3.92	607.	101.	3.92	607.	101.
1	2	80	.480E-02	2.77	610.	87.1	2.77	610.	87.1
1	3	80	.100E-01	5.77	616.	77.0	5.77	616.	77.0
1	4	80	.240E-01	13.8	630.	69.9	13.8	630.	69.9
1	5	80	.500E-01	28.8	658.	65.8	28.8	658.	65.8
1	6	80	.650E-01	37.5	696.	63.3	37.5	696.	63.3
1	7	80	.540E-01	31.1	727.	60.6	31.1	727.	60.6
1	8	80	.120E-03	.692E-01	727.	55.9	.692E-01	727.	55.9
1	9	80	.670E-04	.386E-01	727.	51.9	.386E-01	727.	51.9
1	10	80	.250E-04	.144E-01	727.	48.5	.144E-01	727.	48.5
1	11	80	.140E-04	.807E-02	727.	45.4	.807E-02	727.	45.4
1	12	80	.610E-05	.352E-02	727.	42.8	.352E-02	727.	42.8

Figure 8-4. Output Report from EXPO

At this point we have a normal FORTRAN program, however, we have added no value to that program. It has the same capabilities, strengths, and weaknesses as before.

8.4 A PROMULA Datafile Description

Examining the FORTRAN program, we notice that all of the variables in the output report along with two intermediate calculations are stored in arrays. It is these values which we would like to examine more closely.

We now wish to access the information in these variables by means other than this FORTRAN program. To do this, we need a description of the data on the file so that the individual values can be accessed and manipulated. The datafile needs to be described. The simple PROMULA program below contains such a description. For the sake of readers not familiar with PROMULA notation, we have sequenced the lines and will give a brief discussion of the elements of this description.

```
001    DEFINE PROGRAM "Exposure Analysis Model"
002    OPEN SEGMENT "expo.xeq", STATUS=NEW

003    DEFINE FILE
004    dbase, TYPE=ARRAY
005    END FILE

006    OPEN dbase "expo.dba", STATUS=NEW

007    DEFINE SET
008    tp(50), "Time Points"
009    END

010    DEFINE VARIABLE dbase
011    iday(tp)  TYPE=INTEGER(5)  "Day"
012    imnth(tp) TYPE=INTEGER(5)  "Month"
013    iyr(tp)   TYPE=INTEGER(5)  "Year"
014    avge(tp)  TYPE=REAL(10,1)  "Average Exposure (M/T)"
015    avgi(tp)  TYPE=REAL(10,1)  "Average Intake (M/T)"
016    conc(tp)  TYPE=REAL(10,5)  "Ambient Concentration (M/L**3)"
017    cume(tp)  TYPE=REAL(10,1)  "Cumulative Exposure (M)"
018    cumi(tp)  TYPE=REAL(10,1)  "Cumulative Intake (M)"
019    expos(tp) TYPE=REAL(10,1)  "Exposure (M/T)"
020    perd(tp)  TYPE=REAL(10,1)  "Period"
021    sever(tp) TYPE=REAL(10,1)  "Severity"
022    take(tp)  TYPE=REAL(10,1)  "Intake (M/T)"
023    END

024    DEFINE VARIABLE
025    date(tp)  TYPE=DATE(10)    "Date of exposure"
```

```
026    END

027    DEFINE RELATION
028    KEY(tp,date)
029    END

030    DEFINE PROCEDURE main
031    OPEN dbase "expo.dba", STATUS=OLD
032    SELECT tp IF iday(tp) GT 0
033    date(tp) = iday(tp) + imnth(tp)*100 + iyr(tp)*10000
034    END main

035    END PROGRAM, DO(main)
036    STOP
```

Lines 1 and 2 define the program and the file to be used to contain the actual database description. Lines 3-5 specify that an array file is to be defined, and Line 6 actually creates that file with the name "EXPO.DBA". Lines 7-9 define the "time-points" classification scheme which classifies the values of the variables to be manipulated. Lines 10-23 define the actual variables to be contained on the datafile. Notice that along with the variable identifiers and structures, the binary types and descriptions are also included. This information documents the datafile and is also later used by PROMULA to work with these variables. Lines 24-29 define a variable "date" which is not on the database, but which will be used as a "KEY" to the individual entries in the "time-points" classification scheme. The significance of this key relation will become clearer later when we actually work with the values. Lines 30-34 contain the minimal amount of startup logic that we will need to work with the actual values. Line 31 opens the datafile. Line 32 looks at the day values and restricts the "time-points" classification scheme to those entries where a day is defined. Line 33 computes actual date values from the raw inputs from the FORTRAN program. This is all the logic we need to work with the values. The actual "working with the values" will be done in PROMULA command mode to simplify this discussion. The final two lines end the processing of the program and end the session with PROMULA.

The final step is to process this description using PROMULA itself. This is done by compiling the above source file, EXPO.PRM, to produce the following two files:

EXPO.DBA, which is the actual datafile

EXPO.XEQ, which contains the additional logic and structure discussed above.

We are now ready to add value to the FORTRAN program.

8.5 The Virtual Compilation

Having defined the EXPO.DBA datafile, we can now re-compile the EXPO.FOR program by making explicit reference to this datafile. This is done as follows:

```
pf expo.for expo.gbl -o expo
```

The file `expo.gbl` is used during the compilation. It contains the cross-reference map linking the variables in the FORTRAN program, EXPO.FOR, with the variables in the PROMULA database, EXPO.DBA, and is listed below:

```
expo.dba
expo    iday      iday    virtual
expo    imnth     imnth   virtual
expo    iyr       iyr     virtual
expo    avge      avge    virtual
expo    avgi      avgi    virtual
expo    conc      conc    virtual
expo    cume      cume    virtual
expo    cumi      cumi    virtual
```

```
expo    expos     expos    virtual
expo    perd      perd     virtual
expo    sever     sever    virtual
expo    take      take     virtual
```

The new C code produced by this translation, EXPO.C, is different than the one we produced before without making explicit reference to the EXPO.DBA datafile. The new translation makes virtual reference to the EXPO.DBA variables. Again we compile and link this code to produce a new executable, EXPO.EXE. The size of this file is smaller than the one we produced earlier by about 2400 bytes (the number of bytes needed to store the 12 vector variables of the program — each storing 50 values or 200 bytes — which are now stored in the EXPO.DBA file instead — see Lines 11-22 of the EXPO.PRM datafile description).

8.6 Syntax of the Globals File

In most other FORTRAN host systems the instructions to access information from a database must be placed directly in the FORTRAN source code. To add value to the FORTRAN application, it must be rewritten and typically be re-engineered. The philosophy behind the design of the PROMULA host system is that the original FORTRAN source code not be changed; rather the compiler itself is instructed to make certain variables accessible to the database manager during the compilation step. It is the PROMULA FORTRAN compiler that adds value to the application, not the programmer.

Any variable in a FORTRAN program can be uniquely defined in terms of its identifier and the global symbol dominating it. Here global symbols are either common area names or subprogram names. Given this observation, the syntax of the globals file, such as the one shown above is straight-forward.

The first line of the globals file simply contains the name of the PROMULA database to be used to contain the program values. Remember that the compiler does not actually create a database; rather it converts references to variables in the program to references to variables on the database. It must have access to the database so that it can locate the values.

The subsequent lines simply specify the program variables, their corresponding database variables, and the access method to be used. The fields themselves are free-format. The first field defines a global symbol — common block name or subprogram name — which contains a desired variable. The second field contains the local identifier of the variable itself. The third field contains the name of the corresponding variable on the database. Note that in the example above, the local names and the database names are always the same. In general, this is not required.

The final field defines the actual access method to be used. At this point the only two methods are "virtual" and "dynamic". With virtual access variable values are accessed using a virtual memory scheme. Using purely virtual access there is no limit on the amount of information that can be referenced; however, there may be a performance penalty. With dynamic access the memory needed to contain the variable is allocated at runtime, and the entire set of values is read in at that time. When the use of the variable is completed, the values are written back out and the memory released. Depending upon the situation, dynamic access can be more efficient.

8.7 Using EXPO with PROMULA

As far as the user is concerned, we can now run the program and get the same results. As before, the program reads the input data from an input text file, does some calculations, and writes the output report on an output text file:

EXPO.DAT —> expo —> EXPO.RPT

The run time may have gone up a little because the program now reads from and writes to the PROMULA database EXPO.DBA.

To use the EXPO program with PROMULA, we execute it in precisely the same manner as before, except we must tell the virtual memory system that it is to use the file "EXPO.DBA" as the actual disk file. To do this we enter the following at the operating system level:

```
expo Vexpo.dba
```

The V option tells the program to use the EXPO.DBA datafile. After the run is complete, a report file EXPO.RPT is created and the file "EXPO.DBA" now contains the variable values as they were computed within the program. To verify this, we will use PROMULA again. We will take you through the individual steps.

First enter "PROMULA" at the operating system level to get the PROMULA Main Menu as shown in Figure 8-5 below.

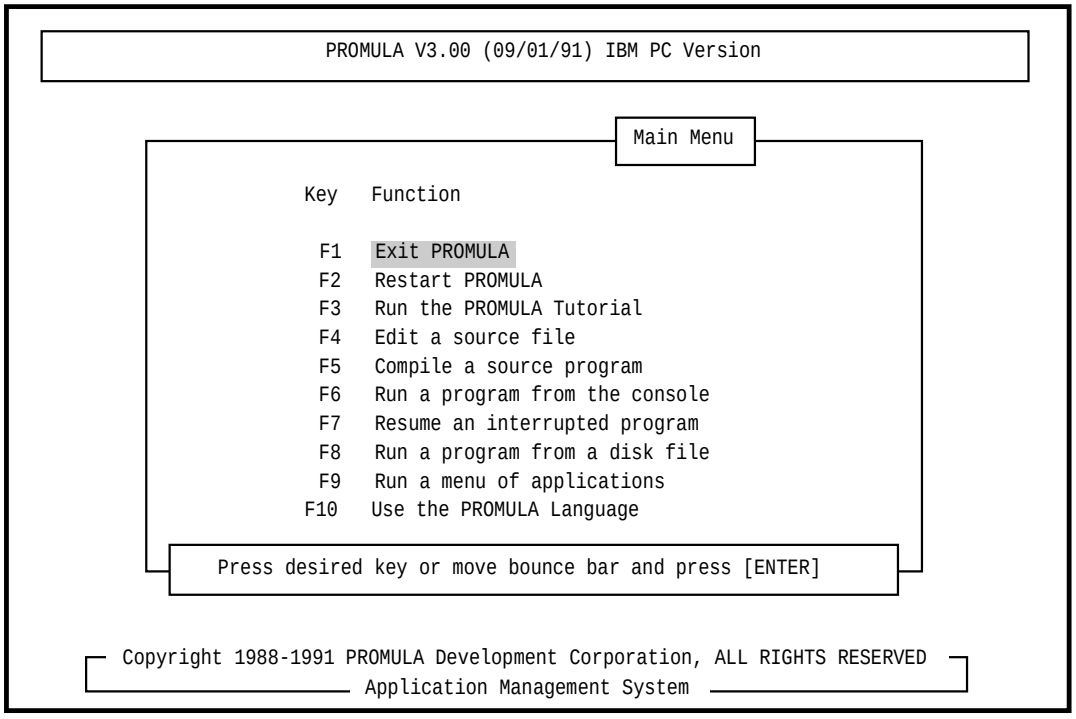

Figure 8-5. PROMULA Main Menu

We wish to work with an existing program, so press the F6 key which results in the dialog shown below.

```
Enter the filename of the program to be executed
                                                                    ?
expo.xeq
```

Upon completion of this request, we are returned to the Main Menu, again as shown in Figure 8-5. Here, we press the F10 key to work with PROMULA directly. We will not do extensive work here, only enough to demonstrate that the values from the program are indeed now fully exposed in a highly usable form.

Initially we can use the BROWSE VARIABLE statement, shown below, to obtain a browsable listing of the variables available. This listing is shown in Figure 8-6. Note that the variable descriptions are all obtained from our initial definitions.

```
PROMULA? BROWSE VARIABLE
```

```
AVGE  Average Exposure (M/T)
AVGI  Average Intake (M/T)
CONC  Ambient        Concentration
(M/L**3)
CUME  Cumulative Exposure (M)
CUMI  Cumulative Intake (M)
EXPOS Exposure (M/T)
PERD  Period

            Press any key to continue
```

Figure 8-6. BROWSE VARIABLE Listing

Next we can look at a particular variable in the list — say, "expos". To do this we enter the command shown below. The result is shown in Figure 8-7. Note that the values match those in the output report shown in Figure 8-4. The date keys are used to label the rows in the report, as specified in the KEY RELATION defined in the program (Line 28 of the program listing). The number of decimal places shown is 1 as corresponds to the type specification for the variable. Finally, the range of values has been restricted to the 17 actual ones, as specified in the SELECT statement (Line 32 of the program listing).

```
PROMULA? BROWSE expos
```

```
                    Exposure (M/T)

                                (1)
              08/01/79         265.3
              09/01/79         299.9
              10/01/79          21.9
              11/01/79           9.8
              12/01/88           6.3
              01/01/88           3.9
              02/01/88           2.8
              03/01/88           5.8
              04/01/88          13.8
              05/01/88          28.8
              06/01/88          37.5
              07/01/88          31.1
              08/01/88     6.9284E-2
              09/01/88     3.8639E-2
              10/01/88     1.4418E-2
              11/01/88     8.8738E-3
              12/01/88     3.5179E-3


  End: Exit  Fn Shift-Fn PgUp PgDn Home Arrows: Browse
```

Figure 8-7. Values of "expos"

Suppose next that we wish to see these values shown to 4 decimal places, so that we can see all of the significant digits. The statement shown below does this. The result is shown in Figure 8-8. This display emphasizes that it is the actual values within the program that are stored on the datafile, and not the reported values as shown in the report.

```
PROMULA? BROWSE exposl:10:4
```

```
                    Exposure (M/T)

                                (1)
               08/01/79        265.2820
               09/01/79        299.8840
               10/01/79         21.9146
               11/01/79          9.0039
               12/01/79          6.3437
               01/01/80          3.9216
               02/01/80          2.7682
               03/01/80          5.7678
               04/01/80         13.8488
               05/01/80         28.8358
               06/01/80         37.4855
               07/01/80         31.1418
               08/01/80          0.0692
               09/01/80          0.0386
               10/01/80          0.0144
               11/01/80          0.0081
               12/01/80          0.0035

      End: Exit  Fn Shift-Fn PgUp PgDn Home Arrows:  Browse
```

Figure 8-8. Actual Values Stored on Datafile

Finally, suppose that we wish to plot values — say, average exposure versus concentration . The statement below is a simple plot request to do this. The result is shown in Figure 8-9.

```
PROMULA? PLOT LINE(conc,avge),
                                                                    ?
TITLE"Average Exposure as a Function of Concentration",
                                                                    ?
LEGEND"A Character-Resolution Plot", XLABEL"Concentration"
```

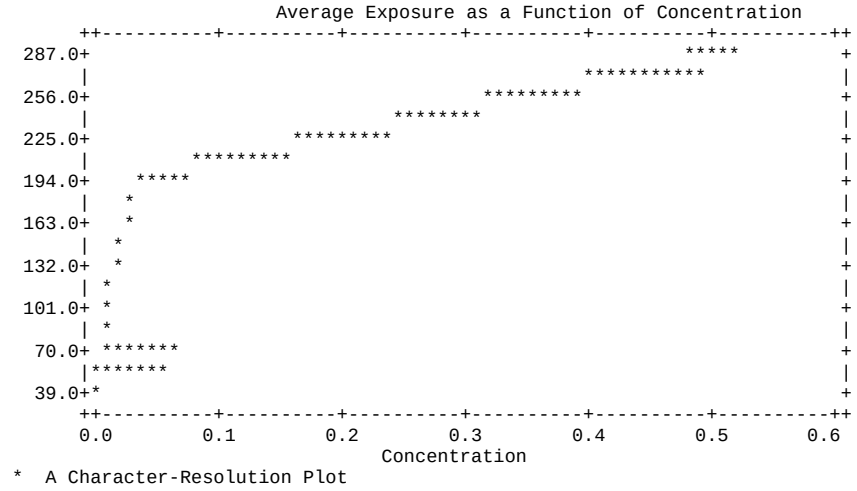

Figure 8-9. Average Exposure Plot Versus Concentration

As you can see from this example, we have "extracted" information out of a batch FORTRAN program by using PROMULA. The example has only scratched the surface. Using the PROMULA system as part of your translation strategy can make all of your programs usable in entirely new ways. FORTRAN programs can share their information with users and other programs.

9. ERROR MESSAGES

The error messages for the FORTRAN compiler are divided into three groups:

(1) Control program errors

(2) FORTRAN preprocessor errors

(3) Runtime errors

9.1 Control Program Errors

The control program interprets the command line as entered by the user and then directs the operations of the FORTRAN preprocessor, the C compiler, and the linker as requested. In particular the control program works as follows.

First, it checks to make certain that all of its support files are present in their expected locations. If it is unable to locate some file then it issues the following error message and exits:

```
PROMULA FORTRAN Compiler installation error
```

If you get this error, either something was done incorrectly during the installation of the compiler, or the file system on your computer has been damaged. Please refer to the platform specific installation instructions which are separate from this manual for further instructions.

Second, the control program processes all entries on its command line to determine the type of tasks to be performed and the resources required to perform them. If there is any entry on the command line which it cannot interpret, it exits with the following message:

```
Bad argument:
```

The "bad argument" message is followed by the particular command line argument which the control program was unable to interpret. Refer to the chapter on using the compiler for a discussion of the valid command line options.

Having checked the command line, the control program processes each FORTRAN source file specified along with any prototype and global files to form intermediate C source files. These C source files are then converted to compiled form via the C compiler. During this conversion you may get some messages from the host C compiler. These depend upon the particular platform. The only messages which the control program might issue are:

```
insufficient memory
```

or

```
File formation error
```

The "insufficient memory" message means that the control program itself is unable to satisfy its modest memory requirements. The "file formation error" message means that the temporary files formed during processing cannot be formed into the final object or executable files needed. Either of these messages indicates a serious problem beyond the domain of this compiler.

Finally, if an executable is to be formed, all compiled files formed and any additional compiled and library files included on the command line are passed to the linker along with the specified linker options.

9.2 FORTRAN Preprocessor Errors

By far the bulk of the messages issued by the PROMULA FORTRAN compiler are issued by the preprocessor, which converts the FORTRAN source to C source. The messages are divided into three general groups:

(1) Syntax errors

(2) Warnings, comments, and notes

(3) Fatal preprocessor errors

Each error message is actually a template from which the actual message is built.

Identifier	Description of variable part
[object]	An object type – such as variable, parameter, statement function, etc.
[ident]	A user defined identifier of a symbol – note that these are always shown in upper case to make them stand out.
[number]	A number or count derived from the user supplied information.
[token]	An actual token or tokens within the source statement – note that these are always shown in upper case to make them stand out.
[type]	A binary type – such as integer, real, integer*2, etc.
[label]	A statement or format label.
[option]	A statement option description.
[statement]	A statement identifier.

9.2.1 Syntax Errors, Warnings, Comments, and Notes

When a syntax error; is encountered, a message is issued and the remainder of the statement is skipped. In addition, the formation of the intermediate C output is blocked. With warnings, comments, and notes processing continues. These are for your information only. See the chapter on using the compiler for details on how to control these optional messages.

Each message consists of four parts separated by a colon as is shown in the sample message below:

```
3: err101.for: E101: The statement function identifier IALPHA may not be assigned a value.
```

The first part of the message is the record number within the source file of the start of the statement causing the message. The second part is the name of the source file itself. The third part is the message identifier, which consists of a letter followed by a number. This identifier letter indicates the type of the message:

Letter	Type of message
E	Syntax error
W	Warning
C	Comment
N	Note

The identifier number can be used to look up the message in the following listing.

Under the theory that a picture is worth a thousand words, rather than trying to describe the circumstances surrounding each error, a typical code fragment generating each error is presented. It is the intent of the message system that each message be as clear, self-contained, and specific as possible.

Error: 101

Message:

The [object] identifier [ident] may not be assigned a value.

Example of message 101:

```
 If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR101
       2          IALPHA(X) = IFIX(X) / 2
       3          DO 10 IALPHA = 1,5
3: err101.for: E101: The statement function identifier IALPHA may not be
                     assigned a value.
       4          WRITE(*,*) I
       5       10 CONTINUE
       6          END
```

Error: 102

Message:

The array [ident] is being subscripted with more than [number] expressions.

Example of message 102:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR102
       2          DIMENSION A(10,15)
       3          A(I,J) = A(I,J,K) + B
3: err102.for: E102: The array A is being subscripted with more than 2
                     expressions.
       4          STOP
       5          END
```

Error: 103

Message:

The statement function either calls itself or there is a missing dimension for [ident].

Example of message 103:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR103(I,B)
       2          C(I)=C(I+1)/B+I/C(I-1)
2: err103.for: E103: The statement function either calls itself or there is
                     a missing dimension for C.
       3          A = C(3)
       4          WRITE(*,*) A,I
       5          END
```

Error: 104

Message:

The substring expression for [ident] is terminated by the symbol [token] rather than a right parenthesis.

Example of message 104:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR104
       2          CHARACTER*10 ERR_MES
       3          ERR_MES(1:3,5) = "Bad Version"
3: err104.for: E104: The substring expression for ERR_MES is terminated by
                     the symbol , rather than a right parenthesis.
       4          STOP
       5          END
```

Error: 105

Message:

The [object] identifier [ident] may not be used to represent a function.

Example of message 105:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR105
       2          STRUCTURE /SHIP/
       3  1       CHARACTER*10 OWNER
       4  1       CHARACTER*10 DESTINATION
       5  1       END
       6          RECORD/SHIP/ NINA, PINTA, SANTA_MARIA
       7          CHARACTER*10 WHERE
       8          WHERE = NINA(3)
8: err105.for: E105: The record identifier NINA may not be used to represent
                     a function.
       9          STOP
      10          END
```

Error: 106

Message:

The [object] identifier [ident] cannot contain a value.

Example of message 106:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR106
       2          DIMENSION A(10,20)
       3          A(2,3) = 0.0
       4          ENTRY BVALUE
       5          A(2,3) = A(2,3) + BVALUE
5: err106.for: E106: The entry point identifier BVALUE cannot contain a
                     value.
```

```
       6          STOP
       7          END
```

Error: 107

Message:

The second quantity [token] in the complex constant is not numeric.

Example of message 107:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR107
       2          COMPLEX A
       3          A = (0,0) + (6,I)
3: err107.for: E107: The second quantity I in the complex constant is not
                     numeric.
       4          END
```

Error: 108

Message:

A complex constant is terminated by the symbol [token] rather than a right parenthesis.

Example of message 108:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR108
       2          COMPLEX A
       3          A = (0,0) + (6,7I)
3: err108.for: E108: A complex constant is terminated by the symbol I rather
                     than a right parenthesis.
       4          END
```

Error: 109

Message:

There are too few right parentheses in the expression terminated by the symbol [token].

Example of message 109:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR109
       2          COMPLEX A
       3          A = ((0,0) + (6,7) * (5,5)
3: err109.for: E109: There are too few right parentheses in the expression
                     terminated by the symbol END_OF_STATEMENT.
       4          END
```

Error: 110

Message:

An improper symbol [token] is being used to introduce a factor.

Example of message 110:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR110
       2          LOGICAL FINISH, STARTED
       3          FINISH = .NAT. STARTED
3: err110.for: E110: An improper symbol . is being used to introduce a
                     factor.
       4          END
```

Error: 111

Message:

The binary type [type] cannot be combined with the binary type [type] in this context.

Example of message 111:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR111

       2          LOGICAL*4 TEST
       3          REAL*4 ALPHA, BETA
       4          TEST = .TRUE.
       5          ALPHA = BETA + TEST
5: err111.for: E111: The binary type real*4 cannot be combined with the
                     binary type logical*4 in this context.
       6          END
```

Error: 112

Message:

The variable [ident] has already been allocated to the common block [ident].

Example of message 112:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR112
       2          COMMON/A/JOE,FRED,FRANK(10)
       3          COMMON/BBB/FRED
3: err112.for: E112: The variable FRED has already been allocated to the
                     common block A.
       4          COMMON /C/FRANK
4: err112.for: E112: The variable FRANK has already been allocated to the
                     common block A.
       5          JOE = 5
       6          WRITE(*,*) JOE
       7          END
```

Error: 113

Message:

The entry name [token] is not unique.

Example of message 113:

```
If Line# Nl Source
-- ----- -- ------
       1          FUNCTION ERR113(A)
       2          ERR113=A
       3          ENTRY ERR113(B)
3: err113.for: E113: The entry name ERR113 is not unique.
       4          ERR113=B
       5          END
```

Error: 114

Message:

An illegal binary combination of [type] with [type] is being made.

Example of message 114:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR114
       2          LOGICAL TEST
       3          INTEGER*4 ALPHA
       4          ALPHA = TEST // ALPHA
4: err114.for: E114: An illegal binary combination of integer*4 with
                     integer*4 is being made.
       5          END
```

Error: 115

Message:

An expression of type [type] is being used in the [type] context.

Example of message 115:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR115
       2          INTEGER*1 I1
       3          COMPLEX BBB
       4          I1 = BBB
4: err115.for: E115: An expression of type complex*8 is being used in the
                     integer*1 context.
       5          BBB = I1
5: err115.for: E115: An expression of type integer*1 is being used in the
                     complex*8 context.
       6          END
```

Error: 116

Message:

The code location line number [label] is being referenced as a format number.

Example of message 116:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR116
       2      100 WRITE(*,*) 'Hello World'
       3          WRITE(*,100) 'Goodbye World'
3: err116.for: E116: The code location line number 100 is being referenced
                     as a format number.
       4          STOP
       5          END
```

Error: 117

Message:

The code location line number [label] is being defined twice.

Example of message 117:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR117
       2      100 WRITE(*,*) 'Hello World'
       3      100 WRITE(*,*) 'Goodbye World'
3: err117.for: E117: The code location line number 100 is being defined
                     twice.
       4          STOP
       5          END
```

Error: 118

Message:

The format line number [label] is being referenced as a code location.

Example of message 118:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR118
       2      100 FORMAT(1X,A20)
       3          WRITE(*,100) 'Hello World'
       4          GOTO 100
4: err118.for: E118: The format line number 100 is being referenced as a
                     code location.
       5          END
```

Error: 119

Message:

The format line number [label] is being defined twice.

Example of message 119:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR119
       2      100 FORMAT(1X,A20)
       3          WRITE(*,100) 'Hello World'
       4          WRITE(*,100) 'Goodbye World'
       5      100 FORMAT(1X,A30)
5: err119.for: E119: The format line number 100 is being defined twice.
       6          STOP
       7          END
```

Error: 120

Message:

The logical unit specification [token] is of the wrong type.

Example of message 120:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR120
       2          LOGICAL TFILE
       3          TFILE = .TRUE.
       4          WRITE(TFILE,*) 'Hello World'
4: err120.for: E120: The logical unit specification TFILE, is of the wrong
                    type.
       5          STOP
       6          END
```

Error: 121

Message:

The internal file specification begins with the symbol [token] rather than an identifier.

Example of message 121:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR121
       2          ENCODE(8,100,10.5) 'Hello World'
2: err121.for: E121: The internal file specification begins with the symbol
                    10.5 rather than an identifier.
       3     100  FORMAT(A8)
       4          STOP
       5          END
```

Error: 122

Message:

The read element is the symbol [token] rather than an identifier.

Example of message 122:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR122
       2       10 FORMAT(a10)
       3          READ(IFILE,10) I,J,K
       4          READ(IFILE,*) 'Hello World'
4: err122.for: E122: The read element is the symbol 'Hello World' rather
                     than an identifier.
       5          STOP
       6          END
```

Error: 123

Message:

The statement function argument name [ident] may not be repeated.

Example of message 123:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR123
       2          E005A(A,A)=A*A
2: err123.for: E123: The statement function argument name A may not be
                     repeated.

       3          PRINT *,'ERR123'
       4          END
```

Error: 124

Message:

The implied do-loop counter [ident] is not an integer variable.

Example of message 124:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR124
       2          DIMENSION A(10)
       3       10 FORMAT(10A10)
       4          READ(IFILE,10) (A(I),R=1,10)
4: err124.for: E124: The implied do-loop counter R is not an integer
                     variable.
       5          STOP
       6          END
```

Error: 125

Message:

The implied do-loop minimum is followed by the symbol [token] rather than a comma.

Example of message 125:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR125
       2          DIMENSION A(10)
       3       10 FORMAT(10A10)
       4          READ(IFILE,10) (A(I),I=1=10)
4: err125.for: E125: The implied do-loop minimum is followed by the symbol
                     = rather than a comma.
       5          STOP
       6          END
```

Error: 126

Message:

The implied do-loop specification is followed by the symbol [token] rather than a right parenthesis.

Example of message 126:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR126
       2          DIMENSION A(10)
       3       10 FORMAT(10A10)
       4          READ(IFILE,10) (A(I),I=1,10=)
4: err126.for: E126: The implied do-loop specification is followed by the
                     symbol = rather than a right parenthesis.
       5          STOP
       6          END
```

Error: 127

Message:

The [option] parameter was not supplied.

Example of message 127:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR127
       2          OPEN(FILE='ERR127.OUT',STATUS='NEW')
2: err127.for: E127: The logical unit number parameter was not supplied.
       3          STOP
       4          END
```

Error: 128

Message:

Dimension [number] for variable [ident] has an invalid extent of [number].

Example of message 128:

```
If Line# Nl Source
-- ----- -- ------
```

```
        1          SUBROUTINE ERR128

        2          PARAMETER (ITEM=0,JJ=-5)
        3          REAL B(3:2),C(0)
        4          COMMON /A/A(-4),E(ITEM)
        5          DIMENSION F(JJ:0),G(JJ)
        6          INTEGER II(-4,JJ:-6)
        7          COMMON H(ITEM:13)
        8          DOUBLE PRECISION DD(ITEM:JJ),DF(JJ:ITEM)
        9          R=4.523
       10          WRITE(1) B,C,A,E,F,G,II,H,DD,DF,R
       11          END
11: err128.for: E128: Dimension 1 for variable B has an invalid extent of 0.
11: err128.for: E128: Dimension 1 for variable C has an invalid extent of 0.
11: err128.for: E128: Dimension 1 for variable A has an invalid extent of -4.
11: err128.for: E128: Dimension 1 for variable E has an invalid extent of 0.
11: err128.for: E128: Dimension 1 for variable G has an invalid extent of -5.
11: err128.for: E128: Dimension 1 for variable II has an invalid extent of -4.
11: err128.for: E128: Dimension 2 for variable II has an invalid extent of 0.
11: err128.for: E128: Dimension 1 for variable DD has an invalid extent of -4.
```

Error: 129

Message:

The information for the [option] was supplied redundantly.

Example of message 129:

```
If Line# Nl Source
-- ----- -- ------

       1          PROGRAM ERR129
       2          OPEN(6,FILE='ERR127.OUT',UNIT=7,STATUS='NEW')
2: err129.for: E129: The information for the logical unit number was
                     supplied redundantly.
       3          STOP
       4          END
```

Error: 130

Message:

The statement option [option] is followed by the symbol [token] rather than by a line number.

Example of message 130:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR130
       2          OPEN(6,FILE='ERR127.OUT',ERR='YES',STATUS='NEW')
2: err130.for: E130: The statement option ERR= is followed by the symbol
                     'YES' rather than by a line number.
       3      77  STOP
       4          END
```

Error: 131

Message:

The end-of-statement was reached before the parameter [option] was supplied.

Example of message 131:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR131
       2        1 FORMAT('Hello World ')
       3          ASSIGN 1
3: Err131.for: E131: The end-of-statement was reached before the parameter
                     "to" was supplied.
       4          WRITE(*,LABEL)
       5          STOP
       6          END
```

Error: 132

Message:

An undefined keyword parameter [token] has been entered when [option] was expected.

Example of message 132:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR132
       2        1 FORMAT('Hello World ')
       3          ASSIGN 1 FOR LABEL
3: err132.for: E132: An undefined keyword parameter FORLABEL has been
                     entered when "to" was expected.
       4          WRITE(*,LABEL)
       5          STOP
       6          END
```

Error: 133

Message:

The variable [ident] in the [statement] statement is not an undimensioned integer.

Example of message 133:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR133
       2          INTEGER IJ(10)
       3          ASSIGN 10 TO IJ
3: err133.for: E133: The variable IJ in the assign statement is not an
                     undimensioned integer.
       4          ASSIGN 20 TO R
4: err133.for: E133: The variable R in the assign statement is not an
                     undimensioned integer.
       5          R=10.0
       6          GOTO IJ,(10,20,30)
6: err133.for: E133: The variable IJ in the goto statement is not an
                     undimensioned integer.
       7       10 WRITE(*,*) R
       8          GOTO R
8: err133.for: E133: The variable R in the goto statement is not an
                     undimensioned integer.
       9       20 WRITE(*,*) IJ
      10          GOTO IJ
```

```
10: err133.for: E133: The variable IJ in the goto statement is not an
                      undimensioned integer.
      11       30 GOTO R,(10,20)
11: err133.for: E133: The variable R in the goto statement is not an
                      undimensioned integer.
      12          END
```

Error: 134

Message:

The symbol [token] within the data value list is not a constant.

Example of message 134:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR134
       2          DIMENSION A(5)
       3          DATA A/1.0,2.0,BETA,4.0,5.0/
3: err134.for: E134: The symbol BETA within the data value list is not a
                     constant.
       4          WRITE(*,*) A
       5          STOP
       6          END
```

Error: 135

Message:

The sign character has been applied to a non-numeric constant value [token].

Example of message 135:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR135
       2          INTEGER A(3)
       3          DATA A/'77'x,'ff'x,-'1'x/
3: err135.for: E135: The sign character has been applied to a non-numeric
                     constant value '1'x.
       4          WRITE(*,*) A
       5          STOP
       6          END
```

Error: 136

Message:

The first member of the complex value [token] is not a numeric constant.

Example of message 136:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR136
       2          COMPLEX A(3)
       3          DATA A/(0.0,1.0),(2.0,4.0),("0",0.0)/
3: err136.for: E136: The first member of the complex value "0" is not a
                     numeric constant.
```

```
       4          WRITE(*,*) A
       5          STOP
       6          END
```

Error: 137

Message:

The symbol [token] was used for the option [option] rather than a variable identifier.

Example of message 137:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR137
       2          OPEN(6,FILE='ERR127.OUT',IOSTAT=32,STATUS='NEW')
2: err137.for: E137: The symbol 32 was used for the option IOSTAT= rather
                     than a variable identifier.
       3          STOP
       4          END
```

Error: 138

Message:

The format does not have a line number.

Example of message 138:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR138
       2          PRINT 1
       3          FORMAT("Hello World")
3: err138.for: E138: The format does not have a line number.
       4          STOP

       5          END
```

Error: 139

Message:

The format specification is preceded by the symbol [token] rather than a left parenthesis.

Example of message 139:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR139
       2          PRINT 1
       3        1 FORMAT "Hello World")
3: err139.for: E139: The format specification is preceded by the symbol
                     "Hello World" rather than a left parenthesis.
       4          STOP
       5          END
```

Error: 140

Message:

The format specification ends in the symbol [token] rather than a right parenthesis.

Example of message 140:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR140
       2          PRINT 1
       3        1 FORMAT("Hello World"
3: err140.for: E140: The format specification ends in the symbol
                     END_OF_STATEMENT rather than a right parenthesis.
       4          STOP
       5          END
```

Error: 141

Message:

A Hollerith string extends beyond the end of the specification.

Example of message 141:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR141
       2          DATA IDAT ...                          /4HB/
2: err141.for: E141: A Hollerith string extends beyond the end of the
                     specification.
       3          WRITE(*,'(A4)') IDAT
       4          STOP
       5          END
```

Error: 142

Message:

A delimited string has no closing delimeter.

Example of message 142:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR142
       2          DATA IDAT/"B/
2: err142.for: E142: A delimited string has no closing delimeter.
       3          WRITE(*,'(A4)') IDAT
       4          STOP
       5          END
```

Error: 143

Message:

The if conditional expression is preceded by the symbol [token] rather than a left parenthesis.

Example of message 143:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR143
       2          IF I .EQ. 33) PRINT *,'Hello World'
2: err143.for: E143: The if conditional expression is preceded by the symbol
                    I rather than a left parenthesis.
       3          STOP
       4          END
```

Error: 144

Message:

The if conditional expression is followed by the symbol [token] rather than a right parenthesis.

Example of message 144:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR144
       2          IF(I .EQ. 33, PRINT *,'Hello World'
2: err144.for: E144: The if conditional expression is followed by the symbol
                    , rather than a right parenthesis.
       3          STOP
       4          END
```

Error: 145

Message:

The [statement] statement contains the unrecognized symbol [token].

Example of message 145:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR145
       2          IF(I .EQ. 45) THEN PRINT *,'Hello World'
2: err145.for: E145: The if statement contains the unrecognized symbol
                    PRINT.
       3  1       STOP
       4  1       END
```

Error: 146

Message:

The [object] [ident] may not be allocated to common.

Example of message 146:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR146
       2          PARAMETER (ICON=3,CON=3.)
       3          REAL A,B
       4          INTEGER Z,CON
```

```
       5          COMMON E,CON
5: err146.for: E146: The parameter CON may not be allocated to common.
       6          WRITE(*,*) A,Z
       7          END
```

Error: 147

Message:

The members of a complex constant are separated by the symbol [token] rather than a comma.

Example of message 147:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR147
       2          COMPLEX A(3)
       3          DATA A/(0.0,1.0),(2.0,4.0),(0.0/0.0)/
3: err147.for: E147: The members of a complex constant are separated by the
                    symbol / rather than a comma.
       4          WRITE(*,*) A
       5          STOP
       6          END
```

Error: 148

Message:

The second member of a complex constant is the symbol [token] rather than a numeric constant.

Example of message 148:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR148
       2          COMPLEX A(3)
       3          DATA A/(0.0,1.0),(2.0,4.0),(0.0,"0.0")/
3: err148.for: E148: The second member of a complex constant is the symbol
                    "0.0" rather than a numeric constant.
       4          WRITE(*,*) A
       5          STOP
       6          END
```

Error: 149

Message:

The default else is not the last part of the blocked if.

Example of message 149:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR149
       2          IF(I .EQ. 99) THEN
       3  1           PRINT *,"Goodbye"
       4  1       ELSE
       5  1           PRINT *,"What now"
       6  1       ELSE IF(I .EQ. 98)
6: err149.for: E149: The default else is not the last part of the blocked
```

```
                     if.
       7  1           PRINT *,"Hello"
       8  1       ENDIF
8: err149.for: E166: The statement introduced by the keyword ENDIF is not a
                     valid statement type.
       9  1       END
```

Error: 150

Message:

Negative and zero if branches are separated by the symbol [token] rather than a comma.

Example of message 150:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR150
       2          IF(I - 6) 10/15,20
2: err150.for: E150: Negative and zero if branches are separated by the
                     symbol / rather than a comma.
       3       10 PRINT *,'Hello'
       4       15 PRINT *,'Goodbye'
       5       20 STOP
       6          END
```

Error: 151

Message:

The zero if branch is the symbol [token] rather than an integer constant.

Example of message 151:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR151
       2          IF(I - 6) 10,IJK,20
2: err151.for: E151: The zero if branch is the symbol IJK rather than an
                     integer constant.
       3       10 PRINT *,'Hello'
       4       15 PRINT *,'Goodbye'
       5       20 STOP
       6          END
```

Error: 152

Message:

The positive if branch is [token] rather than an integer constant.

Example of message 152:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR152
       2          IF(I - 6) 10,15,IJK
2: err152.for: E152: The positive if branch is IJK rather than an integer
                     constant.
```

```
       3       10 PRINT *,'Hello'
       4       15 PRINT *,'Goodbye'
       5       20 STOP
       6          END
```

Error: 153

Message:

The assign statement keyword is followed by the symbol [token] rather than an integer statement number.

Example of message 153:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR153(WHICH)
       2          INTEGER*2 LABEL,WHICH
       3        1 FORMAT('Hello World ',I5)
       4        2 FORMAT('Hello America ',I5)
       5          IF(WHICH .GT. 0) THEN
       6  1           ASSIGN 90001 TO LABEL
       7  1       ELSE
       8  1           ASSIGN 90002 TO LABEL
       9  1       ENDIF
      10          GOTO LABEL
      11    90001 ASSIGN WHICH TO LABEL
11: err153.for: E153: The assign statement keyword is followed by the symbol
                      WHICHTOLABEL rather than an integer statement number.
      12          GOTO 90003
      13    90002 ASSIGN 2 TO LABEL
      14    90003 WRITE(*,LABEL) WHICH
      15          RETURN
      16          END
```

Error: 154

Message:

A complex constant is terminated by the symbol [token] rather than a right parenthesis.

Example of message 154:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR154
       2          COMPLEX A(3)
       3          DATA A/(0.0,1.0),(2.0,4.0),(0.0,0.0/
3: err154.for: E154: A complex constant is terminated by the symbol / rather
                     than a right parenthesis.
       4          WRITE(*,*) A
       5          STOP
       6          END
```

Error: 155

Message:

The constant [token] of type [type] is being read into variable [ident] of type [type].

Example of message 155:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR155
       2          DIMENSION A(3)
       3          DATA A/(0.0,1.0),(2.0,4.0),(0.0,0.0/
3: err155.for: E155: The constant (0.0,1.0) of type complex*8 is being read
                     into variable A of type real*4.
       4          WRITE(*,*) A
       5          STOP
       6          END
```

Error: 156

Message:

The do statement dummy variable identifier is the symbol [token] rather than an identifier.

Example of message 156:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR156
       2          DO 10 1,5
2: err156.for: E156: The do statement dummy variable identifier is the
                     symbol 5 rather than an identifier.
       3          PRINT *, I
       4       10 CONTINUE
       5          STOP
       6          END
```

Error: 157

Message:

The do statement dummy [ident] is a variable of type [type].

Example of message 157:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR157
       2          CHARACTER*4 I
       3          DO 10 I = 1,5
3: err157.for: E157: The do statement dummy I is a variable of type
                     character*4.
       4          PRINT *, I
       5       10 CONTINUE
       6          STOP
       7          END
```

Error: 158

Message:

The do statement dummy is followed by the symbol [token] rather than an equals sign.

Example of message 158:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR158
       2          INTEGER*4 I
       3          DO 10 I / 1,5
3: err158.for: E158: The do statement dummy is followed by the symbol /
                     rather than an equals sign.
       4          PRINT *, I
       5       10 CONTINUE
       6          STOP
       7          END
```

Error: 159

Message:

The [object] [ident] being used to control the data range is not an integer constant.

Example of message 159:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR159
       2          PARAMETER (IVAL = N * N)
       3          DIMENSION A(100)
       4          DATA (A(I),I=1,IVAL)/100*1.2/
4: err159.for: E159: The parameter IVAL being used to control the data range
                     is not an integer constant.
       5          STOP
       6          END
```

Error: 160

Message:

The do statement minimum value is followed by the symbol [token] rather than a comma.

Example of message 160:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR160
       2          INTEGER*4 I
       3          DO 10, I = 1 TO 5
3: err160.for: E160: The do statement minimum value is followed by the
                     symbol TO5 rather than a comma.
       4          PRINT *, I
       5       10 CONTINUE
       6          STOP
       7          END
```

Error: 161

Message:

The do statement maximum value is followed by the symbol [token] rather than a comma.

Example of message 161:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR161
       2          INTEGER*4 I
       3          DO 10 I = 1, 5 BY 1
3: err161.for: E161: The do statement maximum value is followed by the
                     symbol BY1 rather than a comma.
       4          PRINT *, I
       5       10 CONTINUE
       6          STOP
       7          END
```

Error: 162

Message:

The [object] [ident] may not be dimensioned.

Example of message 162:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR162
       2          PARAMETER (ICON=3,CON=3.)
       3          DIMENSION ICON(3)
3: err162.for: E162: The parameter ICON may not be dimensioned.
       4          WRITE(*,*) A,Z
       5          END
```

Error: 163

Message:

The computed goto is opened by the symbol [token] rather than a left parenthesis.

Example of message 163:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR163
       2          READ *,I
       3          GO TO = 10,20) I
3: err163.for: E163: The computed goto is opened by the symbol = rather than
                     a left parenthesis.
       4       10 PRINT *, 'Hello'
       5       20 STOP
       6          END
```

Error: 164

Message:

An element in the computed goto statement list [token] is not a statement label.

Example of message 164:

```
If Line# Nl Source
```

```
-- ----- -- ------
       1          PROGRAM ERR164
       2          READ *,I
       3          GO TO (10,Z0) I
3: err164.for: E164: An element in the computed goto statement list Z0 is
                     not a statement label.
       4       10 PRINT *, 'Hello'
       5       20 STOP
       6          END
```

Error: 165

Message:

The computed goto is closed by the symbol [token] rather than a right parenthesis.

Example of message 165:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR165
       2          READ *,I
       3          GO TO (10,20  I
3: err165.for: E165: The computed goto is closed by the symbol I rather than
                     a right parenthesis.
       4       10 PRINT *, 'Hello'
       5       20 STOP
       6          END
```

Error: 166

Message:

The statement introduced by the keyword [token] is not a valid statement type.

Example of message 166:

```
If Line# Nl Source
-- ----- -- ------
       1          PROMGRA ERR166
1: err166.for: E166: The statement introduced by the keyword PROMGRAERR166
                     is not a valid statement type.
       2          DIMENSION AMAT(10),IMAT(5)
       3          EQUIVALENCE (IMAT(2),AMAT(4))
       4          STOP
       5          END
```

Error: 167

Message:

The call statement subroutine name is [token] and not an identifier.

Example of message 167:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR167
       2          CALL 999(A,B,C)
```

```
2: err167.for: E167: The call statement subroutine name is 999 and not an
                     identifier.
       3          STOP
       4          END
```

Error: 168

Message:

The variable [ident] is used as an assigned format but is never assigned a format label.

Example of message 168:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR168
       2          ASSIGN 10 TO IA
       3          GOTO IA,(10,20)
       4       10 A = 1
       5          WRITE(*,IA) A
       6        1 FORMAT(1X,F10.0)
       7       20 RETURN
       8          END
8: err168.for: E168: The variable IA is used as an assigned format but is
                     never assigned a format label.
```

Error: 169

Message:

An element in an implicit list is the symbol [token] rather than a single letter.

Example of message 169:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR169
       2          IMPLICIT INTEGER(A,BC)
2: err169.for: E169: An element in an implicit list is the symbol BC rather
                     than a single letter.
       3          PRINT *,A,B,C
       4          STOP
       5          END
```

Error: 170

Message:

An upper bound in an implicit list range is the symbol [token] rather than a single letter.

Example of message 170:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR170
       2          IMPLICIT INTEGER(A-DE)
2: err170.for: E170: An upper bound in an implicit list range is the symbol
                     DE rather than a single letter.
       3          PRINT *,A,B,C,D,E
       4          STOP
```

```
       5          END
```

Error: 171

Message:

The implicit statement is closed by the symbol [token] rather than a right parenthesis.

Example of message 171:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR171
       2          IMPLICIT INTEGER (A,B,C
2: err171.for: E171: The implicit statement is closed by the symbol
                    END_OF_STATEMENT rather than a right parenthesis.
       3          PRINT*,A,B,C
       4          STOP
       5          END
```

Error: 172

Message:

An element in the statement list is [token] rather than an identifier.

Example of message 172:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR172
       2          SAVE A,B,99
2: err172.for: E172: An element in the statement list is 99 rather than an
                    identifier.
       3          PRINT *,A,B
       4          STOP
       5          END
```

Error: 173

Message:

The function keyword [token] is not well-formed in the function declaration.

Example of message 173:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR173
       2          INTEGER TEST
       3          PRINT *, TEST(1.0,2.0)
       4          STOP
       5          END
       6         INTEGER 999 FUNCTION TEST(A,B)
6: err173.for: E173: The function keyword 999 is not well-formed in the
                    function declaration.
       7          INTEGER A,B
       8          TEST = A + B
       9          RETURN
```

```
      10          END
```

Error: 174

Message:

The function keyword has been replaced by [token] in the function declaration.

Example of message 174:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR174
       2          INTEGER TEST
       3          PRINT *, TEST(1.0,2.0)
       4          STOP
       5          END
       6          INTEGER FONCTION TEST(A,B)
6: err174.for: E174: The function keyword has been replaced by FONCTIONTEST
                     in the function declaration.
       7          INTEGER A,B
       8          TEST = A + B
       9          RETURN
      10          END
```

Error: 175

Message:

A subroutine or program may not have a type specification.

Example of message 175:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR175
       2          INTEGER TEST
       3          PRINT *, TEST(1.0,2.0)

       4          STOP
       5          END
       6          INTEGER SUBROUTINE TEST(A,B)
6: err175.for: E175: A subroutine or program may not have a type
                     specification.
       7          INTEGER A,B
       8          TEST = A + B
       9          RETURN
      10          END
```

Error: 176

Message:

The program or subprogram identifier [token] is not well formed.

Example of message 176:

```
If Line# Nl Source
```

```
-- ----- -- ------
       1          PROGRAM 176
1: err176.for: E176: The program or subprogram identifier 176 is not well
                     formed.
       2          PRINT *,'Hello World'
       3          STOP
       4          END
```

Error: 177

Message:

The program statement closing parenthesis is missing.

Example of message 177:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR177(INPUT,OUTPUT
1: err177.for: E177: The program statement closing parenthesis is missing.
       2          PRINT *, 'Hello World'
       3          STOP
       4          END
```

Error: 178

Message:

Entry points are allowed in subprograms only.

Example of message 178:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR178
       2          ENTRY SAM
2: err178.for: E178: Entry points are allowed in subprograms only.
       3          WRITE(*,*) 'HI'
       4          END
```

Error: 179

Message:

A statement may not goto itself.

Example of message 179:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR179
       2          I=1
       3       10 GOTO(20,30,10),I
3: err179.for: E179: A statement may not goto itself.
       4       20 I = I + 1
       5          GOTO 10
       6       30 STOP
       7          END
```

Error: 180

Message:

The pointer specification is preceded by the symbol [token] rather than a left parenthesis.

Example of message 180:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR180
       2          PARAMETER (IP=4, JP = 10)
       3          POINTER P, V(0:IP,0:JP))
3: err180.for: E180: The pointer specification is preceded by the symbol P
                    rather than a left parenthesis.
       4          REAL A(55)
       5          DATA A/3*123.456,2*3.45,5*7.89,45*6.78/
       6          P = LOC(a)
       7          PRINT *," computed= ",V(0,0)," should be =",A(1)
       8          STOP
       9          END
```

Error: 181

Message:

The pointer identifier [token] is not a valid identifier symbol.

Example of message 181:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR181
       2          PARAMETER (IP=4, JP = 10)
       3          POINTER (,P, V(0:IP,0:JP))
3: err181.for: E181: The pointer identifier , is not a valid identifier
                    symbol.
       4          REAL A(55)
       5          DATA A/3*123.456,2*3.45,5*7.89,45*6.78/
       6          P = LOC(a)
       7          PRINT *," computed= ",V(0,0)," should be =",A(1)
       8          STOP
       9          END
```

Error: 182

Message:

The pointer is separated from its referent by the symbol [token] rather than by a comma.

Example of message 182:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR182
       2          PARAMETER (IP=4, JP = 10)
       3          POINTER (P = V(0:IP,0:JP))
3: err182.for: E182: The pointer is separated from its referent by the
                    symbol = rather than by a comma.
```

```
       4          REAL A(55)
       5          DATA A/3*123.456,2*3.45,5*7.89,45*6.78/
       6          P = LOC(a)
       7          PRINT *," computed= ",V(0,0)," should be =",A(1)
       8          STOP
       9          END
```

Error: 183

Message:

The referent of the pointer [token] is not a valid identifier symbol.

Example of message 183:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR183
       2          PARAMETER (IP=4, JP = 10)
       3          POINTER (P ,, V(0:IP,0:JP))
3: err183.for: E183: The referent of the pointer , is not a valid identifier
                     symbol.
       4          REAL A(55)
       5          DATA A/3*123.456,2*3.45,5*7.89,45*6.78/
       6          P = LOC(a)
       7          PRINT *," computed= ",V(0,0)," should be =",A(1)
       8          STOP
       9          END
```

Error: 184

Message:

The data statement subscript expression is too complex to process.

Example of message 184:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR184
       2          DIMENSION IVAL(100)
       3          DATA I,J,IVAL(J/I)/50,100,32/
3: err184.for: E184: The data statement subscript expression is too complex
                     to process.
       4          STOP
       5          END
```

Error: 185

Message:

The pointer specification ends in the symbol [token] rather than a right parenthesis.

Example of message 185:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR185
       2          PARAMETER (IP=4, JP = 10)
       3          POINTER (P , V(0:IP,0:JP)
```

```
3: err185.for: E185: The pointer specification ends in the symbol
                     END_OF_STATEMENT rather than a right parenthesis.
       4          REAL A(55)
       5          DATA A/3*123.456,2*3.45,5*7.89,45*6.78/
       6          P = LOC(a)
       7          PRINT *," computed= ",V(0,0)," should be =",A(1)
       8          STOP
       9          END
```

Error: 186

Message:

The value [number] for subscript [number] of array [ident] with minimum of [number] and maximum of [number] is out of range.

Example of message 186:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR186
       2          DIMENSION AVAL(2:15,3:12)
       3          DATA I,J,AVAL(2,2)/6,4,34.5/
3: err186.for: E186: The value 2 for subscript 2 of array AVAL with minimum
                     of 3 and maximum of 12 is out of range.
       4          STOP
       5          END
```

Error: 187

Message:

Data is being assigned the [object] [ident] which is not a variable.

Example of message 187:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR187
       2          PARAMETER (PI = 3.14159)
       3          DATA PI/3.14159/
3: err187.for: E187: Data is being assigned the parameter PI which is not a
                     variable.
       4          STOP
       5          END
```

Error: 188

Message:

Data is being assigned to the subprogram argument [ident].

Example of message 188:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR188(IARGU)
       2          DATA IARGU/56/
2: err188.for: E188: Data is being assigned to the subprogram argument IARGU.
```

```
       3          RETURN
       4          END
```

Error: 189

Message:

The data statement variable list symbol [token] is not valid.

Example of message 189:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR189
       2          DIMENSION A(100)
       3          DATA (A(I),I=1,100),100*1.2/
3: err189.for: E189: The data statement variable list symbol 100 is not
                     valid.
       4          STOP
       5          END
```

Error: 190

Message:

The data implied do-loop variable is followed by the symbol [token] rather than an equals sign.

Example of message 190:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR190
       2          DIMENSION A(100)
       3          DATA (A(I),I*1,100)/100*1.2/
3: err190.for: E190: The data implied do-loop variable is followed by the
                     symbol * rather than an equals sign.
       4          STOP
       5          END
```

Error: 191

Message:

The variable [ident] is used in an assigned goto but is never assigned a statement label.

Example of message 191:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR191
       2          GOTO IA,(10,20)
       3       10 A = 1
       4          WRITE(*,*) A
       5       20 RETURN
       6          END
6: err191.for: E191: The variable IA is used in an assigned goto but is
                     never assigned a statement label.
```

Error: 192

Message:

The data implied do-loop minimum is followed by the symbol [token] rather than a comma.

Example of message 192:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR192
       2          DIMENSION A(100)
       3          DATA (A(I),I=1=100)/100*1.2/
3: err192.for: E192: The data implied do-loop minimum is followed by the
                     symbol = rather than a comma.
       4          STOP
       5          END
```

Error: 193

Message:

The data implied do-loop range is followed by the symbol [token] rather than a right parenthesis.

Example of message 193:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR193
       2          DIMENSION A(100)
       3          DATA (A(I),I=1,100/100*1.2/
3: err193.for: E193: The data implied do-loop range is followed by the
                     symbol / rather than a right parenthesis.
       4          STOP
       5          END
```

Error: 194

Message:

The data array [ident] is being subscripted with more than [number] expression(s).

Example of message 194:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR194
       2          DIMENSION A(100)
       3          DATA (A(I,10),I=1,100/100*1.2/
 3: err194.for: E194: The data array A is being subscripted with more than
                      1 expression(s).
       4          STOP
       5          END
```

Error: 195

Message:

The data substring range for [ident] contains the symbol [token] rather than a colon.

Example of message 195:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR195
       2          CHARACTER*100 MESSAGE
       3          DATA MESSAGE(56,63)/'Hello World'/
3: err195.for: E195: The data substring range for MESSAGE contains the
                     symbol , rather than a colon.
       4          STOP
       5          END
```

Error: 196

Message:

The data substring range for [ident] is terminated by the symbol [token] rather than a right parenthesis.

Example of message 196:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR196
       2          CHARACTER*100 MESSAGE
       3          DATA MESSAGE(56:63/'Hello World'/
3: err196.for: E196: The data substring range for MESSAGE is terminated by
                     the symbol / rather than a right parenthesis.
       4          STOP
       5          END
```

Error: 197

Message:

The data statement specification is closed by the symbol [token] rather than a slash.

Example of message 197:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR197
       2          CHARACTER*100 MESSAGE
       3          DATA MESSAGE(56:63)*'Hello World'/
3: err197.for: E197: The data statement specification is closed by the
                     symbol * rather than a slash.
       4          STOP
       5          END
```

Error: 198

Message:

The data statement value list contains too many values.

Example of message 198:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR198
       2          CHARACTER*100 MESSAGE
       3          DATA MESSAGE(56:63)/'Hello World','Goodbye'/
3: err198.for: E198: The data statement value list contains too many values.
       4          STOP
       5          END
```

Error: 199

Message:

The indexing constant [token] is of type [type] rather than of type integer.

Example of message 199:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR199
       2          PARAMETER(IPI = 100, PI = 3.14159)
       3          DIMENSION A(IPI,PI)
3: err199.for: E199: The indexing constant PI is of type real*4 rather than
                     of type integer.
       4          STOP
       5          END
```

Error: 200

Message:

An unnamed field is being declared with the symbol [token] rather than with %fill.

Example of message 200:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR200
       2          STRUCTURE /STRA/
       3  1           CHARACTER*1 CHR
       4  1           CHARACTER*3 %FULL
4: err200.for: E200: An unnamed field is being declared with the symbol
                     %FULL rather than with %fill.
       5  1           INTEGER*4 ICHR
       6  1       END STRUCTURE
       7          STOP
       8          END
```

Error: 201

Message:

The argument list for [object] [ident] ends with the symbol [token] rather than a closing right parenthesis.

Example of message 201:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR201
       2          SFUNC(X,Y) = SIN(X)*COS(Y)
       3          VAL = SFUNC(2.3,5.4,3.8)
3: err201.for: E201: The argument list for statement function SFUNC ends
                     with the symbol , rather than a closing right
                     parenthesis.
       4          STOP
       5          END
```

Error: 202

Message:

The dimension specification is terminated by the symbol [token] rather than a right parenthesis.

Example of message 202:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR202
       2          DIMENSION A(10(/10*1.0/
2: err202.for: E202: The dimension specification is terminated by the symbol
                     ( rather than a right parenthesis.
       3          STOP
       4          END
```

Error: 203

Message:

The value [number] is not within the valid statement or format number range.

Example of message 203:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR203
       2       10 FORMAT(1X,I10)
       3          READ(5,0) IA
3: err203.for: E203: The value 0 is not within the valid statement or format
                     number range.
       4          IF(IA .EQ. 45) GOTO 12345678
4: err203.for: E203: The value 12345678 is not within the valid statement or
                     format number range.
       5          WRITE(6,0) IA
5: err203.for: E203: The value 0 is not within the valid statement or format
                     number range.
       6    12345 STOP
       7          END
```

Error: 204

Message:

The [object] [ident] is of base type [type] and cannot be assigned a variable length string specification.

Example of message 204:

```
If Line# Nl Source
-- ----- -- ------

       1          PROGRAM ERR204
       2          INTEGER I*4,J*2,K*(*)
2: err204.for: E204: The variable K is of base type integer*4 and cannot be
                     assigned a variable length string specification.
       3          STOP
       4          END
```

Error: 205

Message:

The element length specification for [ident] is closed by the symbol [token] rather than a parenthesis.

Example of message 205:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR205
       2          INTEGER I*4,J*(2,K*(4)
2: err205.for: E205: The element length specification for J is closed by the
                     symbol , rather than a parenthesis.
       3          STOP
       4          END
```

Error: 206

Message:

The character size specification [token] for [ident] is not a constant.

Example of message 206:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR206
       2          INTEGER I*4,J*(2),K*(N)
2: err206.for: E206: The character size specification N for K is not a
                     constant.
       3          STOP
       4          END
```

Error: 207

Message:

The declaration statement value list contains too many values.

Example of message 207:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR207
       2          CHARACTER*100 MESSAGE/'Hello World','Goodbye'/
```

```
2: err207.for: E207: The declaration statement value list contains too many
                     values.
       3          STOP
       4          END
```

Error: 208

Message:

The [statement] statement contains the unrecognized symbol [token].

Example of message 208:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR208
       2          DIMENSION A(10),B(10))
2: err208.for: E208: The dimension statement contains the unrecognized
                     symbol ).
       3          STOP
       4          END
```

Error: 209

Message:

The common block identifier [token] is not a valid identifier symbol.

Example of message 209:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR209
       2          COMMON/123/A(10),B(113)
2: err209.for: E209: The common block identifier 123 is not a valid
                     identifier symbol.
       3          STOP
       4          END
```

Error: 210

Message:

The common block identifier is followed by the symbol [token] rather than a slash.

Example of message 210:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR210
       2          COMMON/CB123,A(10),B(113)
2: err210.for: E210: The common block identifier is followed by the symbol ,
                     rather than a slash.
       3          STOP
       4          END
```

Error: 211

Message:

The equivalence specification is preceded by the symbol [token] rather than a left parenthesis.

Example of message 211:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR211
       2          DIMENSION AMAT(10),IMAT(5)
       3          EQUIVALENCE IMAT(2),AMAT(4))
3: err211.for: E211: The equivalence specification is preceded by the symbol
                     IMAT rather than a left parenthesis.
       4          STOP
       5          END
```

Error: 212

Message:

The equivalence element identifier [token] is not a valid identifier symbol.

Example of message 212:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR212
       2          DIMENSION AMAT(10),IMAT(5)
       3          EQUIVALENCE (10,IMAT(2),AMAT(4))
3: err212.for: E212: The equivalence element identifier 10 is not a valid
                     identifier symbol.
       4          STOP
       5          END
```

Error: 213

Message:

An equivalence relation is being assigned the [object] [ident] which is not a variable.

Example of message 213:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR213
       2          PARAMETER(IVAL = 99)
       3          DIMENSION AMAT(10),IMAT(5)
       4          EQUIVALENCE (IMAT(2),IVAL)
4: err213.for: E213: An equivalence relation is being assigned the parameter
                     IVAL which is not a variable.
       5          STOP
       6          END
```

Error: 214

Message:

An equivalence relation is being assigned to the subprogram argument [ident].

Example of message 214:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR214(AMAT)
       2          DIMENSION AMAT(10),IMAT(5)
       3          EQUIVALENCE (IMAT(2),AMAT(4))
3: err214.for: E214: An equivalence relation is being assigned to the
                     subprogram argument AMAT.
       4          STOP
       5          END
```

Error: 215

Message:

The size of dimension [number]([ident]) of the equivalence element [ident] has not yet been resolved.

Example of message 215:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR215
       2          DIMENSION AMAT(N),IMAT(5)
       3          EQUIVALENCE (IMAT(2),AMAT(4))
3: err215.for: E215: The size of dimension 1(N) of the equivalence element
                     AMAT has not yet been resolved.
       4          STOP
       5          END
```

Error: 216

Message:

The equivalence element [ident] is a [object] rather than a variable.

Example of message 216:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR216
       2          EQUIVALENCE (E(1),F)
       3          COMMON /D/D
       4          EQUIVALENCE (A,B)
       5          DIMENSION B(20)
       6          Z=D(1)+B(1)+A(2)
       7          WRITE(*,*) B,D,E

       8          END
8: err216.for: E216: The equivalence element A is a function rather than a
                     variable.
```

Error: 217

Message:

The equivalence element subscript specification [token] for [ident] is not a constant.

Example of message 217:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR217
       2          DIMENSION AMAT(10),IMAT(5)
       3          EQUIVALENCE (IMAT(2),AMAT(N))
3: err217.for: E217: The equivalence element subscript specification N for
                     AMAT is not a constant.
       4          STOP
       5          END
```

Error: 218

Message:

The symbol [token] is not a valid format specifier.

Example of message 218:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR218
       2       10 FORMAT(1X,I10)
       3          READ(5,FMT=10.0) IA
3: err218.for: E218: The symbol 10.0 is not a valid format specifier.
       4          STOP
       5          END
```

Error: 219

Message:

The equivalence element [ident] is closed by the symbol [token] rather than a right parenthesis.

Example of message 219:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR219
       2          DIMENSION AMAT(10),IMAT(5)
       3          EQUIVALENCE (IMAT(2(,AMAT(4))
3: err219.for: E219: The equivalence element IMAT is closed by the symbol
                     ( rather than a right parenthesis.
       4          STOP
       5          END
```

Error: 220

Message:

The first element [ident] of the equivalence is followed by the symbol [token] rather than a comma introducing the next element.

Example of message 220:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR220
       2          DIMENSION AMAT(10),IMAT(5)
       3          EQUIVALENCE (IMAT(2) AMAT(4))
3: err220.for: E220: The first element IMAT of the equivalence is followed
                     by the symbol AMAT rather than a comma introducing the
                     next element.
       4          STOP
       5          END
```

Error: 221

Message:

The formal argument [ident] has already been defined earlier in the argument list.

Example of message 221:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR221(ALPHA,BETA,ALPHA)
1: err221.for: E221: The formal argument ALPHA has already been defined
                     earlier in the argument list.
       2          RETURN
       3          END
```

Error: 222

Message:

The [object] [ident] is not an undimensioned variable as is required in this context.

Example of message 222:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR222
       2          INTEGER IVEC(10)
       3          DEFINE FILE IVEC(100,512,U,IREC)
3: err222.for: E222: The array variable IVEC is not an undimensioned
                     variable as is required in this context.
       4          STOP
       5          END
```

Error: 223

Message:

The equivalence group with the last element [ident] is closed by the symbol [token] rather than a right parenthesis.

Example of message 223:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR 223
```

```
       2          DIMENSION AMAT(10),IMAT(5)
       3          EQUIVALENCE (IMAT(2),AMAT(4)
3: err223.for: E223: The equivalence group with the last element AMAT is
                     closed by the symbol END_OF_STATEMENT rather than a
                     right parenthesis.
       4          STOP
       5          END
```

Error: 224

Message:

An unexpected end-of-file was encountered while reading the source file.

Example of message 224:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR22
       2          DIMENSION AMAT(10)
       3          READ(5) AMAT
3: err224.for: E224: An unexpected end-of-file was encountered while reading
                     the source file.
```

Error: 225

Message:

The variable [ident] substring subscript [number] is not in the valid range of [number] to [number].

Example of message 225:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR225
       2          CHARACTER*10 NAME,HOME,SON
       3          DATA NAME(0:9)/'Freddy'/
3: err225.for: E225: The variable NAME substring subscript 0 is not in the
                     valid range of 1 to 10.
       4          DATA HOME(1:13)/'Columbus Ohio'/
4: err225.for: E225: The variable HOME substring subscript 13 is not in the
                     valid range of 1 to 10.
       5          DATA SON(5:1)/'Andy'/
5: err225.for: E225: The variable SON substring subscript 1 is not in the
                     valid range of 5 to 10.
       6          WRITE(*,*) NAME
       7          STOP
       8          END
```

Error: 226

Message:

The parameter element [ident] is followed by the symbol [token] rather than an equals sign.

Example of message 226:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR226
       2          PARAMETER ( PI, 3.14159)
2: err226.for: E226: The parameter element PI is followed by the symbol ,
                     rather than an equals sign.
       3          PRINT *, PI
       4          STOP
       5          END
```

Error: 227

Message:

The parameter element [ident] definition is followed by the symbol [token] rather than a constant.

Example of message 227:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR227
       2          PARAMETER ( PI = 3.14159
2: err227.for: E227: The parameter element PI definition is followed by the
                     symbol END_OF_STATEMENT rather than a constant.
       3          PRINT *, PI
       4          STOP
       5          END
```

Error: 228

Message:

The namelist statement begins with the symbol [token] rather than a slash.

Example of message 228:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR228
       2          NAMELIST SHIP/ A,B,C,I1,I2
2: err228.for: E228: The namelist statement begins with the symbol SHIP
                     rather than a slash.
       3          C = 0.0
       4          READ(*,SHIP,END=10)
4: err228.for: E129: The information for the an option was supplied
                     redundantly.
       5          IF(C .NE. 0.0) STOP
       6       10 PRINT *,'NO DATA FOUND'
       7          STOP
       8          END
```

Error: 229

Message:

The namelist identifier [token] is not a valid identifier symbol.

Example of message 229:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR 229
       2          namelist// SHIP/ A,B,C,I1,I2
2: err229.for: E229: The namelist identifier / is not a valid identifier
                     symbol.
       3          C = 0.0
       4          READ(*,SHIP,END=10)
4: err229.for: E129: The information for the an option was supplied
                     redudantly.
       5          IF(C .NE. 0.0) STOP
       6       10 PRINT *,'NO DATA FOUND'
       7          STOP
       8          END
```

Error: 230

Message:

The namelist identifier [ident] has already been defined as a [object].

Example of message 230:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR230
       2          INTEGER SHIP
       3          NAMELIST/ SHIP/ A,B,C,I1,I2
3: err230.for: E230: The namelist identifier SHIP has already been defined
                     as a variable.
       4          C = 0.0
       5          READ(*,SHIP,END=10)
       6          IF(C .NE. 0.0) STOP
       7       10 PRINT *, 'NO DATA FOUND'
       8          STOP
       9          END
```

Error: 231

Message:

The namelist identifier [ident] is followed by the symbol [token] rather than a slash.

Example of message 231:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR231
       2          NAMELIST/ SHIP, A,B,C,I1,I2
2: err231.for: E231: The namelist identifier SHIP is followed by the symbol
                     , rather than a slash.
       3          C = 0.0
       4          READ(*,SHIP,END=10)
       5          IF(C .NE. 0.0) STOP
       6       10 PRINT *,'NO DATA FOUND'
       7          STOP
       8          END
```

Error: 232

Message:

The namelist [ident] element identifier [token] is not a valid identifier symbol.

Example of message 232:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR232
       2          NAMELIST/ SHIP/,A,B,C,I1,I2
2: err232.for: E232: The namelist SHIP element identifier , is not a valid
                    identifier symbol.
       3          C = 0.0
       4          READ(*,SHIP,END=10)
       5          IF(C .NE. 0.0) STOP
       6       10 PRINT *,'NO DATA FOUND'
       7          STOP
       8          END
```

Error: 233

Message:

The namelist [ident] element [ident] is a subprogram argument.

Example of message 233:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR233(J1,K1,I1)
       2          NAMELIST/ SHIP/ A,B,C,I1,I2
2: err233.for: E233: The namelist SHIP element I1 is a subprogram argument.
       3          C = 0.0
       4          READ(*,SHIP,END=10)
       5          IF(C .NE. 0.0) RETURN
       6       10 PRINT *,'NO DATA FOUND'
       7          STOP
       8          END
```

Error: 234

Message:

The external element identifier [token] is not a valid identifier symbol.

Example of message 234:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR234
       2          EXTERNAL AFUNC,,BFUNC,CFUNC
2: err234.for: E234: The external element identifier , is not a valid
                    identifier symbol.
       3          CALL PROCESS(AFUNC,IVAL)
       4          CALL PROCESS(BFUNC,IVAL)
4: err234.for: E248: The variable argument BFUNC has been entered where a
                    subprogram argument is required.
       5          CALL PROCESS(CFUNC,IVAL)
```

```
5: err234.for: E248: The variable argument CFUNC has been entered where a
                     subprogram argument is required.
       6          STOP
       7          END
```

Error: 235

Message:

The entry identifier [token] is not a valid identifier symbol.

Example of message 235:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR235(A)
       2          A = A * 10
       3          ENTRY (ERR235A(A))
3: err235.for: E235: The entry identifier ( is not a valid identifier
                     symbol.
       4          PRINT *,A
       5          RETURN
       6          END
```

Error: 236

Message:

The alternate return indicated by the symbol [token] may be used only in subroutines.

Example of message 236:

```
If Line# Nl Source
-- ----- -- ------
       1          FUNCTION ERR236(*)
1: err236.for: E236: The alternate return indicated by the symbol * may be
                     used only in subroutines.
       2          ERR236=5.0
       3          RETURN 1
       4          END
```

Error: 237

Message:

The filename [token] within the parenthetical form of the include statement is not a valid identifier symbol.

Example of message 237:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR237
       2          INCLUDE("DDL237")
2: err237.for: E237: The filename "DDL237" within the parenthetical form of
                     the include statement is not a valid identifier symbol.

       3          STOP
       4          END
```

Error: 238

Message:

The include statement filename [token] is not a valid string constant or identifier symbol.

Example of message 238:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR238
       2          INCLUDE 238DDL
2: err238.for: E238: The include statement filename 238 is not a valid
                     string constant or identifier symbol.
       3          STOP
       4          END
```

Error: 239

Message:

The include statement file [token] cannot be opened.

Example of message 239:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR239
       2          INCLUDE "DDL239.INC"
2: err239.for: E239: The include statement file DDL239.INC cannot be opened.
       3          STOP
       4          END
```

Error: 240

Message:

The length specification [number] associated with [ident] is not available for the [type] type.

Example of message 240:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR240
       2          LOGICAL*4 ALPHA,BETA*3
2: err240.for: E240: The length specification 3 associated with BETA is not
                     available for the logical type.
       3          STOP
       4          END
```

Error: 241

Message:

The type statement length specification [token] is not available for the [type] type.

Example of message 241:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR241
       2          LOGICAL*3 ALPHA,BETA*2
2: err241.for: E241: The type statement length specification 3 is not
                     available for the logical*4 type.
       3          STOP
       4          END
```

Error: 242

Message:

The character statement variable length string [token] is not a constant.

Example of message 242:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR242
       2          CHARACTER*(N) ALPHA,BETA
2: err242.for: E242: The character statement variable length string N is not
                     a constant.
       3          STOP
       4          END
```

Error: 243

Message:

The character statement variable length string is closed by the symbol [token] rather than a right parenthesis.

Example of message 243:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR243

       2          CHARACTER*(10, ALPHA,BETA
2: err243.for: E243: The character statement variable length string is
                     closed by the symbol , rather than a right parenthesis.
       3          STOP
       4          END
```

Error: 244

Message:

The character statement variable length string [token] is not a constant.

Example of message 244:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR244
       2          CHARACTER*N,ALPHA,BETA
2: err244.for: E244: The character statement variable length string N is not
```

```
                     a constant.
       3          STOP
       4          END
```

Error: 245

Message:

The argument [token] of type [type] has been entered where a call-by-value argument of type [type] is required.

Example of message 245:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR245
       2          DOUBLE PRECISION DVAL
       3          DVAL = DFLOAT(3.45)
3: err245.for: E245: The argument 3.45 of type real*4 has been entered where
                     a call-by-value argument of type integer*4 is required.
       4          STOP
       5          END
```

Error: 246

Message:

The argument [ident] has the local variable [ident] as an adjustable dimension.

Example of message 246:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR246(I)
       2          INTEGER I(J)
       3          COMMON JK
       4          CALL ETEST(I,J)
       5          END
5: err246.for: E246: The argument I has the local variable J as an
                     adjustable dimension.
```

Error: 247

Message:

The argument [token] of type [type] has been entered where a call-by-reference argument of type [type] is required.

Example of message 247:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR247
       2          COMPLEX IC,LC
       3          CALL SUB247(AVAL)
       4          CALL SUB247(IC)
       5          CALL SUB247(IC*LC)
5: err247.for: E247: The argument IC*LC of type complex*8 has been entered
                     where a call-by-reference argument of type real*4 is
                     required.
       6          STOP
```

```
       7          END
```

Error: 248

Message:

The [object] argument [ident] has been entered where a subprogram argument is required.

Example of message 248:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR248
       2          DIMENSION AVAL(10)
       3          EXTERNAL FUNC,IFUNC
       4          CALL SUB248(FUNC)
       5          CALL SUB248(IFUNC)
       6          CALL SUB248(AVAL)
6: err248.for: E248: The array variable argument AVAL has been entered where
                     a subprogram argument is required.
       7          STOP
       8          END
```

Error: 249

Message:

The argument [token] of type [type] has been entered and cannot be converted into a call-by-value argument of type [type].

Example of message 249:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR249
       2          COMPLEX ALPHA
       3          LOGICAL*1 IV
       4          ALPHA = CMPLX(IV,BETA)
4: err249.for: E249: The argument IV of type logical*1 has been entered and
                     cannot be converted into a call-by-value argument of
                     type real*8.
       5          STOP
       6          END
```

Error: 250

Message:

The common variable [ident] has the adjustable dimension [ident].

Example of message 250:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR250(IDIM)
       2          COMMON/ALPHA/ A(IDIM)
       3          WRITE(*,*) 'HI'
       4          END
4: err250.for: E250: The common variable A has the adjustable dimension
                     IDIM.
```

Error: 251

Message:

The value-argument [token] has been entered where a subprogram argument is required.

Example of message 251:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR251
       2          EXTERNAL FUNC,IFUNC
       3          CALL SUB251(FUNC)
       4          CALL SUB251(IFUNC)
       5          CALL SUB251(AVAL*BVAL)
5: err251.for: E251: The value-argument AVAL*BVAL has been entered where a
                     subprogram argument is required.
       6          STOP
       7          END
```

Error: 252

Message:

The symbol [token] is not valid in the formal argument list.

Example of message 252:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR252(ALPHA,BETA,*,32)
1: err252.for: E252: The symbol 32 is not valid in the formal argument list.
       2          RETURN
       3          END
```

Error: 253

Message:

The local variable [ident] has an adjustable dimension [ident].

Example of message 253:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR253(IDIM)
       2          REAL A(IDIM)
       3          WRITE(*,*) 'HI'
       4          END
4: err253.for: E253: The local variable A has an adjustable dimension IDIM.
```

Error: 254

Message:

The formal argument list contains the symbol [token] rather than a comma or right parenthesis.

Example of message 254:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR254(A,B*)
1: err254.for: E254: The formal argument list contains the symbol * rather
                     than a comma or right parenthesis.
       2          STOP
       3          END
```

Error: 255

Message:

The alternate return label [token] is not an integer constant.

Example of message 255:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR255
       2          CALL SUB255(A,B,*STOP)
2: err255.for: E255: The alternate return label STOP is not an integer
                     constant.
       3          PRINT *, 'Hello'
       4      100 STOP
       5          END
```

Error: 256

Message:

The actual argument list contains the symbol [token] rather than a comma or right parenthesis.

Example of message 256:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR256
       2          CALL SUB256(A,B&100)
2: err256.for: E256: The actual argument list contains the symbol & rather
                     than a comma or right parenthesis.
       3          PRINT *, 'Hello'
       4      100 STOP
       5          END
```

Error: 257

Message:

The actual argument list contains more than the [number] argument(s) required by the [object] [ident].

Example of message 257:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR257
       2          AVAL = ALOG10(BVAL,100.0)
```

```
2: err257.for: E257: The actual argument list contains more than the 1
                     argument(s) required by the system function ALOG10.
       3          STOP
       4          END
```

Error: 258

Message:

The adjustable dimension [ident] for [ident] is not a variable or parameter.

Example of message 258:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR258(FUNC,N)
       2          DIMENSION FUNC(N)
       3          CALL N(FUNC)
       4          RETURN
       5          END
5: err258.for: E258: The adjustable dimension N for FUNC is not a variable
                     or parameter.
```

Error: 259

Message:

The do while conditional is followed by the symbol [token] rather than the end-of-statement.

Example of message 259:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR259
       2          DO WHILE (A .LT. B) BEGIN
2: err259.for: E259: The do while conditional is followed by the symbol
                     BEGIN rather than the end-of-statement.
       3            A = A +B
       4          END DO
4: err259.for: E166: The statement introduced by the keyword ENDDO is not a
                     valid statement type.
       5          STOP
       6          END
```

Error: 260

Message:

The reference to the [object] [ident] is followed by the symbol [token] rather than a left parenthesis.

Example of message 260:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR260
       2          SFUNC(A,B) = SIN(A) * COS(B)
       3          ALPHA = BETA + SFUNC,3.5,4.2)
3: err260.for: E260: The reference to the statement function SFUNC is
```

```
                   followed by the symbol , rather than a left parenthesis.
       4           STOP
       5           END
```

Error: 261

Message:

The argument list for [object] [ident] contains the symbol [token] rather a comma.

Example of message 261:

```
If Line# Nl Source
-- ----- -- ------
       1           PROGRAM ERR261
       2           SFUNC(A,B) = SIN(A) * COS(B)
       3           ALPHA = BETA + SFUNC(3.5)
3: err261.for: E261: The argument list for statement function SFUNC contains
                   the symbol ) rather a comma.
       4           STOP
       5           END
```

Error: 262

Message:

The structure definition identifier [token] has already been defined as a structure identifier.

Example of message 262:

```
If Line# Nl Source
-- ----- -- ------
       1           PROGRAM ERR262
       2           STRUCTURE /SHIP/
       3  1        CHARACTER*10 OWNER
       4  1        CHARACTER*10 DESTINATION
       5  1        END
       6           STRUCTURE /SHIP/
6: err262.for: E262: The structure definition identifier SHIP has already
                     been defined as a structure identifier.
       7           CHARACTER*10 OWNER2
       8           CHARACTER*10 DESTINATION2
       9           END
      10           RECORD /SHIP/ NINA, PINTA, SANTA_MARIA
      11           CHARACTER*10 WHERE
      12           WHERE = NINA(3)
      13           STOP
      14           END
```

Error: 263

Message:

The actual argument list contains less than the [number] argument(s) required by the [object] [ident].

Example of message 263:

```
If Line# Nl Source
-- ----- -- ------
       1           PROGRAM ERR263
```

```
       2          ALPHA = AMOD(BETA)
2: err263.for: E263: The actual argument list contains less than the
                     argument(s) required by the system function AMOD.
       3          END
```

Error: 264

Message:

The structure or record definition identifier is introduced by the symbol [token] rather than a slash.

Example of message 264:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR264
       2          STRUCTURE /SHIP/
       3  1       CHARACTER*10 OWNER
       4  1       CHARACTER*10 DESTINATION
       5  1       END

       6          RECORD SHIP/ NINA, PINTA, SANTA_MARIA
6: err264.for: E264: The structure or record definition identifier is
                     introduced by the symbol SHIP rather than a slash.
       7          CHARACTER*10 WHERE
       8          WHERE = NINA.OWNER
8: err264.for: E145: The assignment statement contains the unrecognized
                     symbol ..
       9          STOP
      10          END
```

Error: 265

Message:

The structure or record identifier [token] is not a valid identifier symbol.

Example of message 265:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR265
       2          STRUCTURE /SHIP/
       3  1       CHARACTER*10 OWNER
       4  1       CHARACTER*10 DESTINATION
       5  1       END
       6          RECORD //SHIP/ NINA, PINTA, SANTA_MARIA
6: err265.for: E265: The structure or record identifier / is not a valid
                     identifier symbol.
       7          CHARACTER*10 WHERE
       8          WHERE = NINA.OWNER
8: err265.for: E145: The assignment statement contains the unrecognized
                     symbol ..
       9          STOP
      10          END
```

Error: 266

Message:

The structure identifier [token] classifying the record has not been defined.

Example of message 266:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR266
       2          STRUCTURE /SHIP/
       3  1       CHARACTER*10 OWNER
       4  1       CHARACTER*10 DESTINATION
       5  1       END
       6          RECORD /SHUP/ NINA, PINTA, SANTA_MARIA
6: err266.for: E266: The structure identifier SHUP classifying the record
                     has not been defined.
       7          CHARACTER*10 WHERE
       8          WHERE = NINA.OWNER
8: err266.for: E145: The assignment statement contains the unrecognized
                     symbol ..
       9          STOP
      10          END
```

Error: 267

Message:

The structure or record definition identifier is followed by the symbol [token] rather than a slash.

Example of message 267:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR267
       2          STRUCTURE /SHIP/
       3  1       CHARACTER*10 OWNER
       4  1       CHARACTER*10 DESTINATION
       5  1       END
       6          RECORD /SHIP, NINA, PINTA, SANTA_MARIA
6: err267.for: E267: The structure or record definition identifier is
                     followed by the symbol , rather than a slash.
       7          CHARACTER*10 WHERE
       8          WHERE = NINA.OWNER
8: err267.for: E145: The assignment statement contains the unrecognized
                     symbol ..
       9          STOP
      10          END
```

Error: 268

Message:

The member identifier [token] associated with the [object] [ident] is not a valid identifier symbol.

Example of message 268:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR268
       2          STRUCTURE /SHIP/
       3  1       CHARACTER*10 OWNER
       4  1       CHARACTER*10 DESTINATION
       5  1       END
       6          RECORD /SHIP/ NINA, PINTA, SANTA_MARIA
```

```
       7          CHARACTER*10 WHERE
       8          WHERE = NINA.3
8: err268.for: E268: The member identifier 3 associated with the structure
                    SHIP is not a valid identifier symbol.
       9          STOP
      10          END
```

Error: 269

Message:

The member identifier [token] is not a member of the [object] [ident].

Example of message 269:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR269
       2          STRUCTURE /SHIP/
       3  1       CHARACTER*10 OWNER
       4  1       CHARACTER*10 DESTINATION
       5  1       END
       6          RECORD /SHIP/ NINA, PINTA, SANTA_MARIA
       7          CHARACTER*10 WHERE
       8          WHERE = NINA.OWNR
8: err269.for: E269: The member identifier OWNR is not a member of the
                    structure SHIP.

       9          STOP
      10          END
```

Error: 270

Message:

The * dimension specification for [ident] must be the last dimension.

Example of message 270:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR270(A,B,C)
       2          DIMENSION A(*,3)
2: err270.for: E270: The * dimension specification for A must be the last
       3          DIMENSION B(2,2,5,*)
       4          DIMENSION C(2,*,5)
4: err270.for: E270: The * dimension specification for C must be the last
       5          WRITE(*,*) A(1,1),B(2,2,2,2),C(2,2,2)
       6          END
```

Error: 271

Message:

The equivalence element substring specification [token] for [ident] is not a constant.

Example of message 271:

```
If Line# Nl Source
```

```
-- ----- -- ------
       1          PROGRAM ERR271
       2          CHARACTER AMAT*(10),IMAT*(5)
       3          EQUIVALENCE (IMAT(2:5),AMAT(N:N+3))
3: err271.for: E271: The equivalence element substring specification N for
                     AMAT is not a constant.
       4          STOP
       5          END
```

Error: 272

Message:

The symbol [token] is not a valid subscript expression separator, a comma was expected.

Example of message 272:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR272
       2          CHARACTER*10 ERRMES(10,20)
       3          ERRMES(1:3,5) = "Bad Version"
3: err272.for: E272: The symbol : is not a valid subscript expression
                     separator, a comma was expected.
       4          STOP
       5          END
```

Error: 273

Message:

The symbol [token] is not a valid substring range separator, a colon was expected.

Example of message 273:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR273
       2          PARAMETER(N = 5)
       3          CHARACTER AMAT*(10),IMAT*(5)
       4          EQUIVALENCE (IMAT(2:5),AMAT(N,N+3))
4: err273.for: E273: The symbol , is not a valid substring range separator,
                     a colon was expected.
       5          STOP
       6          END
```

Error: 274

Message:

The equivalence element substring specification is closed by the symbol [token] rather than a right parenthesis.

Example of message 274:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR274
       2          PARAMETER(N = 5)
```

```
       3          CHARACTER AMAT*(10),IMAT*(5)
       4          EQUIVALENCE (IMAT(2:5),AMAT(N:N+3,))
4: err274.for: E274: The equivalence element substring specification is
                     closed by the symbol , rather than a right parenthesis.
       5          STOP
       6          END
```

Error: 275

Message:

The kind parameter is the symbol [token] rather than an integer constant or the identifier of a constant integer parameter.

Example of message 275:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR275
       2          INTEGER SHORT, DOUBLE, QUAD
       3          PARAMETER (SHORT = 2, DOUBLE = 8, QUAD = 16)
       4          PRINT *, 23, 0, 1234567, 42_1, 42_SHORT
       5          PRINT *, 13.5, 0.1234567, 123.45678, 00.30_DOUBLE
       6          PRINT *, 3.0, 3., 12345., 0., .1234567_QUAD
       7          PRINT *, (1, -1), (3.14, 1.0), (3.14_*, -7),
       8         +      (-1.0, 1E-27_QUAD)
7: err275.for: E275: The kind parameter is the symbol * rather than an
                     integer constant or the identifier of a constant
                     integer parameter.
       9          END
```

Error: 276

Message:

The kind identifier [token] has not yet been defined as a constant integer parameter.

Example of message 276:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR276
       2          INTEGER SHORT, DOUBLE, QUAD
       3          PARAMETER (SHORT = 2, DOUBLE = 8, QUAD = 16)
       4          PRINT *, 23, 0, 1234567, 42_1, 42_SHORT
       5          PRINT *, 13.5, 0.1234567, 123.45678, 00.30_DOUBLE
       6          PRINT *, 3.0, 3., 12345., 0., .1234567_QUAD
       7          PRINT *, (1, -1), (3.14, 1.0), (3.14_DUBLE, -7),
       8         +      (-1.0, 1E-27_QUAD)
7: err276.for: E276: The kind identifier DUBLE has not yet been defined as a
                     constant integer parameter.
       9          END
```

Error: 277

Message:

The kind identifier [ident] is a [object] rather than a constant integer parameter.

Example of message 277:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR277
       2          INTEGER SHORT,DUBLE, QUAD
       3          PARAMETER (SHORT = 2, DOUBLE = 8, QUAD = N)
       4          PRINT *, 23, 0, 1234567, 42_1, 42_SHORT
       5          PRINT *, 13.5, 0.1234567, 123.45678, 00.30_DOUBLE
5: err277.for: E277: The kind identifier DOUBLE is a parameter rather
                     than a constant integer parameter.
       6          PRINT *, 3.0, 3., 12345., 0., .1234567_QUAD
6: err277.for: E277: The kind identifier QUAD is a parameter rather than a
                     constant integer parameter.
       7          PRINT *, (1, -1), (3.14, 1.0), (3.14_DOUBLE, -7),
       8         +      (-1.0, 1E-27_QUAD)
7: err277.for: E277: The kind identifier DOUBLE is a parameter rather than
                     a constant integer parameter.
       9          END
```

Error: 278

Message:

The kind value [number] associated with [ident] is not available for the [type] type.

Example of message 278:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR278
       2          INTEGER SHORT, DOUBLE, QUAD
       3          PARAMETER (SHORT = 2, DOUBLE = 8, QUAD = 20)
       4          PRINT *, 23, 0, 1234567, 42_1, 42_SHORT
       5          PRINT *, 13.5, 0.1234567, 123.45678, 00.30_DOUBLE
       6          PRINT *, 3.0, 3., 12345., 0., .1234567_QUAD
6: err278.for: E278: The kind value 20 associated with ERR278 is not
                     available for the real type.
       7          PRINT *, (1, -1), (3.14, 1.0), (3.14_DOUBLE, -7),
       8         +      (-1.0, 1E-27_QUAD)
7: err278.for: E278: The kind value 20 associated with ERR278 is not
                     available for the real type.
       9          END
```

Error: 279

Message:

A statement function argument identifier [token] is not a valid identifier symbol.

Example of message 279:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR279
       2          SFUNC(X,6) = SIN(X) + COS(Y)
2: err279.for E279:       A statement function argument identifier 6 is not a
                        valid identifier symbol.
       3          STOP
       4          END
```

Error: 280

Message:

A statement function argument list is terminated by the symbol [token] rather than a right parenthesis.

Example of message 280:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR280
       2          SFUNC(X,Y*) = SIN(X) + COS(Y)
2: err280.for: E280: A statement function argument list is terminated by the
                     symbol * rather than a right parenthesis.
       3          STOP
       4          END
```

Error: 281

Message:

A statement function parameter list is followed by the symbol [token] rather than an equals sign.

Example of message 281:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR281
       2          SFUNC(X,Y) * = SIN(X) + COS(Y)
2: err281.for: E281: A statement function parameter list is followed by the
                     symbol * rather than an equals sign.
       3          STOP
       4          END
```

Error: 282

Message:

The left-hand-side of the assignment statement begins with the symbol [token] which is not a valid identifier symbol.

Example of message 282:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR282
       2          DIMENSION AVAL(10,10)
       3          +VAL(I,J) = 99.5
3: err282.for: E282: The left-hand-side of the assignment statement begins
                     with the symbol + which is not a valid identifier symbol.
       4          STOP
       5          END
```

Error: 283

Message:

The right-hand-side of the assignment is preceded by the symbol [token] rather than an equals sign.

Example of message 283:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR283
       2          A *= (B + C)
2: err283.for: E283: The right-hand-side of the assignment is preceded by
                     the symbol * rather than an equals sign.
       3          STOP
       4          END
```

Error: 284

Message:

The equivalenced variable [ident] may not be redefined as being external.

Example of message 284:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR284(K)
       2          EQUIVALENCE(ISINE,SINE)
       3          EXTERNAL SINE
3: err284.for: E284: The equivalenced variable SINE may not be redefined as
                     being external.
       4          WRITE(*,*) SINE(K)
       5          END
```

Error: 285

Message:

The attribute specifier list contains the symbol [token] where a comma or :: is expected.

Example of message 285:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR285
       2          REAL :: METERS
       3          REAL, PARAMETER * :: INCHES_PER_METER = 39.37
3: err285.for: E285: The attribute specifier list contains the symbol *
                     where a comma or :: is expected.
       4          READ *, METERS
       5          PRINT *, METERS, "meters =", METERS * INCHES_PER_METER, "inche
       6          END PROGRAM ERR285
```

Error: 286

Message:

The attribute specifier [token] is not defined.

Example of message 286:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR286
```

```
       2          REAL :: METERS
       3          REAL, PARIMETER :: INCHES_PER_METER = 39.37
3: err286.for: E286: The attribute specifier PARIMETER is not defined.
       4          READ *, METERS
       5          PRINT *, METERS, "meters =", METERS * INCHES_PER_METER, "inch
       6          END PROGRAM ERR286
```

Error: 287

Message:

The attribute specifier [token] is defined as part of the standard but is not implemented in this dialect.

Example of message 287:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR287
       2          REAL :: METERS
       3          REAL, PARAMETER, PRIVATE :: INCHES_PER_METER = 39.37
3: err287.for: E287: The attribute specifier PRIVATE is defined as part of
                     the standard but is not implemented in this dialect.
       4          READ *, METERS
       5          PRINT *, METERS, "meters =", METERS * INCHES_PER_METER, "inche
       6          END PROGRAM ERR287
```

Error: 288

Message:

The dimensioned variable [ident] may not be redefined as being external.

Example of message 288:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR288(K)
       2          DIMENSION SINE(100)
       3          EXTERNAL SINE
3: err288.for: E288: The dimensioned variable SINE may not be redefined as
                     being external.
       4          WRITE(*,*) SINE(K)
       5          END
```

Error: 289

Message:

The type of [object] [ident] is undefined.

Example of message 289:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR289
       2          IMPLICIT NONE
       3          INTEGER IVAL,JVAL
       4          IVAL = 100
       5          JVAL = 200
```

```
       6          KVAL = IVAL + JVAL
6: err289.for: E289: The type of variable KVAL is undefined.
       7          STOP
       8          END
```

Error: 290

Message:

The subprogram argument [ident] may not be allocated to a common block.

Example of message 290:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR290(IARGU,JARGU)
       2          COMMON/ALPHA/A,B,JARGU
2: err290.for: E290: The subprogram argument JARGU may not be allocated to a
                     common block.
       3          RETURN
       4          END
```

Error: 291

Message:

The statement function definitions must precede any executable statements.

Example of message 291:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR291
       2          PRINT *,'Hello'
       3          SFUNC(X,Y) = COS(X) + SIN(Y)
3: err291.for: E291: The statement function definitions must precede any
                     executable statements.
       4          PRINT *, SFUNC(3.1,5.2)
       5          STOP
       6          END
```

Error: 292

Message:

The [object] identifier [ident] may not be assigned a value.

Example of message 292:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR292
       2          PARAMETER (PI = 3.14159)
       3          PI = 3.14159
3: err292.for: E292: The parameter identifier PI may not be assigned a
                     value.
       4          STOP
       5          END
```

Error: 293

Message:

The variable [ident] has type [type] and not type integer as required in this context.

Example of message 293:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR293
       2          DEFINE FILE VEC(100,512,U,IREC)
2: err293.for: E293: The variable VEC has type real*4 and not type integer
                     as required in this context.
       3          STOP
       4          END
```

Error: 294

Message:

The format contains the unrecognized specification character [token].

Example of message 294:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR294
       2          WRITE(*,1) "Hello World"
       3        1 FORMAT(1X,A10,Y5)
3: err294.for: E294: The format contains the unrecognized specification
                     character Y.
       4          STOP
       5          END
```

Error: 295

Message:

The format specification has a bad t format specification.

Example of message 295:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR295
       2          WRITE(*,1) "Hello World"
       3        1 FORMAT(T,A10)
3: err295.for: E295: The format specification has a bad t format
                     specification.
       4          STOP
       5          END
```

Error: 296

Message:

The format specification has a bad business format string.

Example of message 296:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR296
       2        1 FORMAT(1X,B'Z99999999999999999999999999999999999999999999999')
2: err296.for: E296: The format specification has a bad business format string.
       3          WRITE (6,1) 12345
       4          STOP
       5          END
```

Error: 297

Message:

A b format specification is not followed by an n or a z.

Example of message 297:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR297
       2        1 FORMAT(BC,1X,I5)
2: err297.for: E297: A b format specification is not followed by an n or a
                     z.
       3          WRITE (6,1) 12345
       4          STOP
       5          END
```

Error: 298

Message:

A variable < format specification has no closing >.

Example of message 298:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR298
       2          DIMENSION A(5)
       3          DATA A/1.,2.,3.,4.,5./
       4          DO 10 I = 1,10
       5  1       WRITE(6,100) I
       6  1 10    CONTINUE
       7          DO 20 I=1,5
       8  1       WRITE (6,101) (A(I),J=1,I)
       9  1 20    CONTINUE
      10    100   FORMAT(I<MAX(I,5)>)
      11    101   FORMAT(<I>F10.<I-1)
11: err298.for: E298: A variable < format specification has no closing >.
      12          END
```

Error: 299

Message:

The format specification has a bad Hollerith string.

Example of message 299:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR299
       2          WRITE(*,1)
       3        1 FORMAT(1X,30HHello World)
3: err299.for: E299: The format specification has a bad Hollerith string.
       4          STOP
       5          END
```

Error: 300

Message:

The executable statement number [label] was referenced but not defined.

Example of message 300:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR300(IOP)
       2          WRITE(*,*) "hello world"
       3          IF(IOP .EQ. 0) GOTO 20
       4          STOP
       5          END
5: err300.for: E300: The executable statement number 20 was referenced but
                     not defined.
```

Error: 301

Message:

The format statement number [label] was referenced but not defined.

Example of message 301:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM ERR301
       2          WRITE(*,1) "hello world"
       3          STOP
       4          END
4: err301.for: E301: The format statement number 1 was referenced but not
                     defined.
```

Error: 302

Message:

The namelist identifier [ident] may be used only as an i/o target.

Example of message 302:

```
If Line# Nl Source
```

```
-- ----- -- ------
       1          SUBROUTINE ERR302
       2          NAMELIST/NML/A,B
       3          PARAMETER(NML=3)
3: err302.for: E302: The namelist identifier NML may be used only as an i/o
                     target.
       4          REAL NML
4: err302.for: E302: The namelist identifier NML may be used only as an i/o
                     target.
       5          EXTERNAL NML
5: err302.for: E302: The namelist identifier NML may be used only as an i/o
                     target.
       6          NML(I)=I**2
6: err302.for: E302: The namelist identifier NML may be used only as an i/o
                     target.
       7          ASSIGN 10 TO NML
7: err302.for: E302: The namelist identifier NML may be used only as an i/o
                     target.
       8          GOTO NML
8: err302.for: E302: The namelist identifier NML may be used only as an i/o
                     target.
       9       10 READ(5,NML)
      10          END
```

Error: 303

Message:

The specifier [token] is not valid for the [statement] statement.

Example of message 303:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR303
       2          CLOSE(UNIT=1,END=10)
2: err303.for: E303: The specifier END= is not valid for the close statement.
       3          INQUIRE(UNIT=1,FMT=5)
3: err303.for: E303: The specifier FMT= is not valid for the inquire statement.
       4          END
```

Error: 304

Message:

Both unit number and unit name may not be specified on the [statement] statement.

Example of message 304:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR304
       2          INQUIRE(UNIT=5,FILE='DUMMY',IOSTAT=IOS)

2: err304.for: E304: Both unit number and unit name may not be specified on
                     the inquire statement.
       3          WRITE(*,*) IOS
       4          END
```

Error: 305

Message:

The entry parameter identifier [ident] is a [object] and not a variable.

Example of message 305:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR305
       2          A(I) = 3.7+I
       3          ENTRY SUM(A,I)
3: err305.for: E305: The entry parameter identifier A is a statement
                     function and not a variable.
       4          ALPHA = A(I)
       5          END
```

Error: 306

Message:

The value [token] is not a valid repeat count.

Example of message 306:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR306
       2          REAL A(10),B(10)
       3          DATA A/0*0.0,10*0.0/
3: err306.for: E306: The value 0 is not a valid repeat count.
       4          DATA (B(I),I=1,10)/0*1.0/
4: err306.for: E306: The value 0 is not a valid repeat count.
       5          WRITE(*,*) A,B
       6          END
```

Error: 307

Message:

The illegal character [token] is in the binary constant [token].

Example of message 307:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR307
       2          IT=B'0101'
       3          IP=B'0121'
3: err307.for: E307: The illegal character 2 is in the binary constant
                     B'0121'.
       4          WRITE(*,*) IT,IP
       5          END
```

Error: 308

Message:

The illegal character [token] is in the octal constant [token].

Example of message 308:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR308
       2          IT=O'0101'
       3          IP=O'0191'
3: err308.for: E308: The illegal character 9 is in the octal constant
                     O'0191'.
       4          WRITE(*,*) IT,IP
       5          END
```

Error: 309

Message:

The illegal character [token] is in the hexidecimal constant [token].

Example of message 309:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR309
       2          IT=H'0101'
       3          IP=H'01G1'
3: err309.for: E309: The illegal character G is in the hexidecimal constant
                     H'01G1'.
       4          WRITE(*,*) IT,IP
       5          END
```

Error: 310

Message:

The [object] [ident] may not be redefined as a [statement].

Example of message 310:

```
If Line# Nl Source
-- ----- -- ------
       1          FUNCTION E310(K)
       2          EXTERNAL SINE
       3          PARAMETER(SINE=3.3)
3: err310.for: E310: The function SINE may not be redefined as a parameter.
       4          E310 = SINE
       5          END
```

Error: 311

Message:

The common variable [ident] may not be redefined as a [statement].

Example of message 311:

```
If Line# Nl Source
-- ----- -- ------
       1          FUNCTION ERR311(K)
       2          COMMON J
```

```
       3          PARAMETER(J=2)
3: err311.for: E311: The common variable J may not be redefined as a
                    parameter.
       4          E595 = K + J
       5          END
```

Error: 312

Message:

The subprogram argument [ident] may not be redefined as a [statement].

Example of message 312:

```
If Line# Nl Source
-- ----- -- ------
       1          FUNCTION ERR312(K)
       2          PARAMETER(K=3)
2: err312.for: E312: The subprogram argument K may not be redefined as a
                    parameter.
       3          E595 = K
       4          END
```

Error: 313

Message:

The dimensioned variable [ident] may not be redefined as a [statement].

Example of message 313:

```
If Line# Nl Source
-- ----- -- ------
       1          FUNCTION ERR313(K)
       2          REAL A(10)
       3          PARAMETER (A=3.45)
3: err313.for: E313: The dimensioned variable A may not be redefined as a
                    parameter.
       4          E595 = K + A
       5          END
```

Error: 314

Message:

The equivalenced variable [ident] may not be redefined as a [statement].

Example of message 314:

```
If Line# Nl Source
-- ----- -- ------
       1          FUNCTION ERR314(K)
       2          COMMON J
       3          EQUIVALENCE(J,N)
       4          PARAMETER (N=-1)
4: err314.for: E314: The equivalenced variable N may not be redefined as a
                    parameter.
       5          ERR314 = K + J + N
       6          END
```

Error: 315

Message:

The [object] [ident] may not be redefined as being external.

Example of message 315:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR315(K)
       2          PARAMETER(SINE=3.3)
       3          EXTERNAL SINE
3: err315.for: E315: The parameter SINE may not be redefined as being
                    external.
       4          WRITE(*,*) SINE(K)
4: err315.for: E105: The parameter identifier SINE may not be used to
                    represent a function.
       5          END
```

Error: 316

Message:

The common variable [ident] may not be redefined as being external.

Example of message 316:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE ERR316(K)
       2          COMMON SINE
       3          EXTERNAL SINE
3: err316.for: E316: The common variable SINE may not be redefined as being
                    external.
       4          WRITE(*,*) SINE(K)
       5          END
```

Note: 801

Message:

The [statement] statement is nonstandard.

Example of message 801:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM W801
       2        1 FORMAT(I5)
       3          ACCEPT 1,IVAL
3: warn801.for: N801: The accept statement is nonstandard.
       4          PRINT *,IVAL
4: warn801.for: N801: The print statement is nonstandard.
       5          STOP
       6          END
```

Note: 802

Message:

Omitting the period from the [token] format specification is nonstandard.

Example of message 802:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W802
       2        1 FORMAT(1X,F6,I5,E10)
2: warn802.for: N802: Omitting the period from the F format specification is
                      nonstandard.
2: warn802.for: N802: Omitting the period from the E format specification is
                      nonstandard.
       3          WRITE(1,1) A,I,B
       4          RETURN
       5          END
```

Note: 803

Message:

The exponent field appended to the D format specification is nonstandard.

Example of message 803:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W803(A,B)
       2          REAL*4 A
       3          REAL*8 B
       4        1 FORMAT(1X,E26.8E5,D30.20E5)
4: warn803.for: N803: The exponent field appended to the D format
                      specification is nonstandard.
       5          WRITE(1,1) A, B
       6          RETURN
       7          END
```

Note: 804

Message:

The [token] format specification is nonstandard.

Example of message 804:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W804
       2          DIMENSION IVAL(6)
       3        1 FORMAT(1X, 6O10)
3: warn804.for: N804: The O format specification is nonstandard.
       4          WRITE(6,1) IVAL
       5          RETURN
       6          END
```

Note: 805

Message:

The b'..' business format specification is nonstandard.

Example of message 805:

```
 If Line# Nl Source
 -- ----- -- ------
        1           PROGRAM W805
        2           WRITE (6,*), 'ZZ999, 12345'
        3           WRITE (6,'(1X,B''ZZ999'')'), 12345
3: warn805.for: N805: The b'..' business format specification is nonstandard
        4           STOP
        5           END
```

Note: 806

Message:

Omitting the comma after the [token] format specification is nonstandard.

Example of message 806:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W806
       2        1 FORMAT(1H1'Hello World')
2: warn806.for: N806: Omitting the comma after the H format specification is
                      nonstandard.
       3        2 FORMAT(1X 'Good Bye')
3: warn806.for: N806: Omitting the comma after the X format specification is
                      nonstandard.
       4          WRITE(*,1)
       5          WRITE(*,2)
       6          STOP
       7          END
```

Comment: 807

Message:

The input format contains a superfluous S specification.

Example of message 807:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM W807
       2        1 FORMAT(SP, I6)
       3          READ(5,'(SP,I5)') IV
3: warn807.for: C807: The input format contains a superfluous S
                      specification.
       4          WRITE(6,1) IV
       5          READ(5,1) IV
5: warn807.for: C807: The input format contains a superfluous S
                      specification.
      6          WRITE(6,1) IV
```

```
       7          READ(5,2) IV
       8        2 FORMAT(SS,I8)
8: warn807.for: C807: The input format contains a superfluous S
                      specification.
       9          STOP
      10          END
```

Note: 808

Message:

A comma in a format following a left parenthesis is nonstandard.

Example of message 808:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM W808
       2        1 FORMAT(,1H1,'Hello World')
2: warn808.for: N808: A comma in a format following a left parenthesis is
                      nonstandard.
       3          WRITE(6,1)
       4          WRITE(6,'(,9H Good Bye)')
4: warn808.for: N808: A comma in a format following a left parenthesis is
                      nonstandard.
       5          STOP
       6          END
```

Note: 809

Message:

A comma in a format preceding a right parenthesis is nonstandard.

Example of message 809:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM W809
       2        1 FORMAT(1X,'Hello World',)
2: warn809.for: N809: A comma in a format preceding a right parenthesis
                      is nonstandard.
       3          WRITE(6,1)
       4          WRITE(6,'(1X,8HGood Bye,)')
4: warn809.for: N809: A comma in a format preceding a right parenthesis is
                      nonstandard.
       5          STOP
       6          END
```

Note: 810

Message:

A sequence of more than one comma in a format is nonstandard.

Example of message 810:

```
If Line# Nl Source
-- ----- -- ------
```

```
       1          PROGRAM W810
       2        1 FORMAT(1X,,'Hello World')
2: warn810.for: N810: A sequence of more than one comma in a format is
                      nonstandard.
       3          WRITE(6,1)
       4          WRITE(6,'(1X,,8HGood Bye)')
4: warn810.for: N810: A sequence of more than one comma in a format is
                      nonstandard.
       5          STOP
       6          END
```

Note: 811

Message:

The use of variable expressions within a format is nonstandard.

Example of message 811:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM W811
       2          DIMENSION (5)
       3          DATA A/1.,2.,3.,4.,5./
       4          DO 10 I = 1,10
       5  1       WRITE(6,100) I
       6  1 10    CONTINUE
       7          DO 20 I=1,5
       8  1       WRITE(6,101) (A(I),J=1,I)
       9  1 20    CONTINUE
      10    100   FORMAT(I<MAX(I,5)>)
10: warn811.for: N811: The use of variable expressions within a format is
                       nonstandard.
      11    101   FORMAT(<I>F10.<I-1>)
11: warn811.for: N811: The use of variable expressions within a format is
                       nonstandard.
11: warn811.for: N811: The use of variable expressions within a format is
                       nonstandard.
      12          END
```

Comment: 812

Message:

A floating point format field width may be too small in this format.

Example of message 812:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM W812
       2        1 FORMAT(1X,E20.17)
2: warn812.for: C812: A floating point format field width may be too small
                      in this format.
       3        2 FORMAT(1X,E20.12)
       4          WRITE(6,1) VAL
       5          WRITE(6,2) VAL
       6          STOP
       7          END
```

Note: 813

Message:

The [option] option on the [statement] statement is nonstandard.

Example of message 813:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM W813
       2          OPEN(1,FILE='TEST1.DAT')
       3          OPEN(2,NAME='TEST2.DAT')
3: warn813.for: N813: The NAME= option on the open statement is nonstandard.
       4          STOP
       5          END
```

Warning: 814

Message:

The argument [token] of type [type] has been entered where an argument of type [type] has been used.

Example of message 814:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM W814
       2          CALL SUB1(A,B,C,1)
       3          CALL SUB1(I,X,Y,2)
3: warn814.for: W814: The argument I of type integer*4 has been entered
                      where an argument of type real*4 has been used.
       4          STOP
       5          END
```

Comment: 815

Message:

A data value of type [type] is being assigned to the variable [ident] of type [type].

Example of message 815:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM W815
       2          INTEGER IVAL
       3          DATA IVAL/4HFRED/
3: warn815.for: C815: A data value of type character is being assigned to
                      the variable IVAL of type integer*4.
       4          WRITE(*,'(1X,A4)') IVAL
       5          STOP
       6          END
```

Comment: 816

Message:

The binary type [type] is being used where type [type] is expected.

Example of message 816:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM W816
       2          INTEGER IVAL
       3          IVAL = .TRUE.
3: warn816.for: C816: The binary type logical*2 is being used where type
                      integer*4 is expected.
       4          WRITE(*,'(1X,I5)') IVAL
       5          STOP
       6          END
```

Note: 817

Message:

If the option unit= is omitted from the unit specifier, then it must be first.

Example of message 817:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W817
       2          READ (FMT=6,3)A
2: warn817.for: N817: If the option unit= is omitted from the unit
                      specifier, then it must be first.
       3       6  FORMAT(1X,F10.2)
       4      30  FORMAT(1X,G20.10)
       5          B=5.4
       6          WRITE (ERR=20,6,30)B
6: warn817.for: N817: If the option unit= is omitted from the unit
                      specifier, then it must be first.
       7       20 CONTINUE
       8          END
```

Warning: 818

Message:

The argument [token] is being defined with type [type] when it has been passed an argument of type [type].

Example of message 818:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM W818
       2          CALL S818(34.5,56,(8.7,6.5),.FALSE.,'F')
       3          STOP
       4          END
       5          SUBROUTINE S818(I,R,C,L,S)
       6          REAL R
6: warn818.for: W818: The argument R is being defined with type real*4 when
                      it has been passed an argument of type integer*4.
       7          COMPLEX C
       8          CHARACTER S
       9          LOGICAL L
```

```
      10          C=R+I
10: warn818.for: W818: The argument I is being defined with type integer*4
                       when it has been passed an argument of type real*4.
      11          L=.TRUE.
      12          WRITE(*,*) S
      13          END
```

Warning: 819

Message:

The TL field specification will shift position before column 1.

Example of message 819:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W819
       2          WRITE (6,20)
       3       20 FORMAT(1X,'BIRD',TL6,2HXX)
3: warn819.for: W819: The TL field specification will shift position before
                      column 1.
       4          END
```

Warning: 820

Message:

The [type] function [ident] was previously defined as having type [type].

Example of message 820:

```
If Line# Nl Source
-- ----- -- ------
       1          FUNCTION W820(J)
       2          W820 = FLOAT(I)
       3          END
       4          SUBROUTINE W820A(I)
       5          INTEGER W820
       6          I = W820(I+I)
6: warn820.for: W820: The integer*4 function W820 was previously defined as
                      having type real*4.
       7          END
```

Warning: 821

Message:

This [type] function has been previously referenced as having type [type].

Example of message 821:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM W821
       2          REAL W821A
       3          E = W821A(2)
       4          END
       5          INTEGER FUNCTION W821A(I)
```

```
       6          W821A=I
6: warn821.for: W821: This integer*4 function has been previously referenced
                      as having type real*4.
       7          END
```

Note: 822

Message:

The use of initial or redundant commas in symbol lists is nonstandard.

Example of message 822:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM W822
       2          PARAMETER ( , PI = 3.14159)
2: warn822.for: N822: The use of initial or redundant commas in symbol lists
                      is nonstandard.
       3          WRITE(*,*) PI
       4          STOP
       5          END
```

Note: 823

Message:

The common block [ident] has a [type] variable [ident] and a [type] variable [ident].

Example of message 823:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM W823
       2          COMMON/ALPHA/IVAL,RVAL,SVAL
       3          INTEGER IVAL
       4          REAL RVAL
       5          CHARACTER*10 SVAL
       6          READ(*,*) IVAL,RVAL,SVAL
       7          CALL HELPER
       8          STOP
       9          END
9: warn823.for: N823: The common block ALPHA has a real*4 variable RVAL and
                      a character*10 variable SVAL.
```

Warning: 824

Message:

The variable [ident] substring subscript [number] is not in the range of
[number] to [number].

Example of message 824:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM W824
       2          CHARACTER*10 NAME,HOME,SON
       3          NAME(0:9) = 'Freddy'
3: warn824.for: W824: The variable NAME substring subscript 0 is not in the
```

```
                        range of 1 to 10.
       4          HOME(1:13) = 'Columbus Ohio'
4: warn824.for: W824: The variable HOME substring subscript 13 is not in the
                        range of 1 to 10.

       5          SON(5:1) = 'Andy'
5: warn824.for: W824: The variable SON substring subscript 1 is not in the
                        range of 5 to 10.
       6          STOP
       7          END
```

Warning: 825

Message:

The subscript value [number] for subscript [number] of array [ident] is not in the range of [number] to [number].

Example of message 825:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W825
       2          PARAMETER(IA = 3)
       3          REAL A(0:3,4)
       4          A(-1,4)=0.
4: warn825.for: W825: The subscript value -1 for subscript 1 of array A is
                        not in the range of 0 to 3.
       5          A(3,IA+2)=0.
5: warn825.for: W825: The subscript value 5 for subscript 2 of array A is
                        not in the range of 1 to 4.
       6          END
```

Comment: 826

Message:

The variable [ident] substring subscript is of type [type] rather than integer.

Example of message 826:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W826
       2          CHARACTER*10 ABCD
       3          ABCD(1:2) = 'AB'
       4          I=5
       5          ABCD (3.0:4) = ' CD'
5: warn826.for: C826: The variable I substring subscript is of type real*4
                        rather than integer.
       6          R=5
       7          ABCD(R:I+1) = 'EF'
7: warn826.for: C826: The variable R substring subscript is of type real*4
                        rather than integer.
       8          WRITE(*,*) ABCD
       9          END
```

Comment: 827

Message:

The function [ident] has been previously referenced or defined as a subroutine.

Example of message 827:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W827
       2          CALL W827A(J,1)
       3          END
       4          FUNCTION W827A(J,I)
4: warn827.for: C827: The function W827A has been previously referenced or
                      defined as a subroutine.
       5          CALL W827C(J,I)
       6          I = W827C(J,1)
6: warn827.for: C827: The function W827C has been previously referenced or
                      defined as a subroutine.
       7          W827A=I+J
       8          END
```

Comment: 828

Message:

The subroutine [ident] has been previously referenced or defined as a function.

Example of message 828:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W828
       2          R =  W828A(J,1)
       3          END
       4          SUBROUTINE W828A(J,I)
4: warn828.for: C828: The subroutine W828A has been previously referenced or
                      defined as a function.
       5          I = W828C(J,1)
       6          CALL W828C(J,I)
6: warn828.for: C828: The subroutine W828C has been previously referenced or
                      defined as a function.
       7          END
```

Note: 829

Message:

The array [ident] has more than 7 dimensions.

Example of message 829:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W829
       2          DIMENSION A(2,2,2,2,2,2,2)
2: warn829.for: N829: The array A has more than 7 dimensions.
       3          A(1,2,1,2,1,2,1,2) = 3.3
       4          WRITE(*,*) A(1,2,1,2,1,2,1,2)
       5          END
```

Note: 830

Message:

An entry point is being defined within an if or do range.

Example of message 830:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W830
       2          DO 10 I=1,3
       3  1       ENTRY GEORGE
3: warn830.for: N830: An entry point is being defined within an if or do
                      range.
       4  1    10 CONTINUE
       5          END
```

Warning: 831

Message:

The variable [ident] subscript is of type [type] rather than integer.

Example of message 831:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W831
       2          DIMENSION B(0:1)
       3          D = B(0.0)
3: warn831.for: W831: The variable D subscript is of type real*4 rather than
                      integer.
       4          END
```

Note: 832

Message:

The use of hexidecimal, octal, or binary constants is nonstandard.

Example of message 832:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W832
       2          IT=O'77'
2: warn832.for: N832: The use of hexidecimal, octal, or binary constants is
                      nonstandard.
       3          IP=H'FF'
3: warn832.for: N832: The use of hexidecimal, octal, or binary constants is
                      nonstandard.
       4          WRITE(*,*) IT,IP
       5          END
```

Note: 833

Message:

The nonparenthetical form of the parameter statement is nonstandard.

Example of message 833:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W833
       2          PARAMETER I=3,J=4
2: warn833.for: N833: The nonparenthetical form of the parameter statement
                      is nonstandard.
       3          WRITE(*,*) I,J
       4          END
```

Note: 834

Message:

Using a repeat count with the [token] format specification is nonstandard.

Example of message 834:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W834
       2          READ (5,10) A
       3       10 FORMAT (3/,F10.0)

3: warn834.for: N834: Using a repeat count with the / format specification
                      is nonstandard.
       4          A=A + 1.0
       5          END
```

Note: 835

Message:

Omitting the field after the period in a format specification is nonstandard.

Example of message 835:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W835
       2          WRITE(5,10) 1
       3       10 FORMAT(1X,I3.)
3: warn835.for: N835: Omitting the field after the period in a format
                      specification is nonstandard.
       4          WRITE(5,20),2.
       5       20 FORMAT(1X,F6.)
5: warn835.for: N835: Omitting the field after the period in a format
                      specification is nonstandard.
       6          WRITE(5,30),3.D0
       7       30 FORMAT(1X,1P,E20.)
7: warn835.for: N835: Omitting the field after the period in a format
                      specification is nonstandard.
       8          END
```

Note: 836

Message:

Omitting the repeat count before the [token] format specification is nonstandard.

Example of message 836:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W836
       2          WRITE(*,'(X,11HHello World)')
2: warn836.for: N836: Omitting the repeat count before the X format
                      specification is nonstandard.
       3          END
```

Comment: 837

Message:

The variable [ident] has already been dimensioned.

Example of message 837:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W837(Z)
       2          REAL A(10)
       3          COMMON/AB/B(10),A(6)
3: warn837.for: C837: The variable A has already been dimensioned.
       4          REAL B(10)
4: warn837.for: C837: The variable B has already been dimensioned.
       5          Z=1.
       6          END
```

Warning: 838

Message:

The data initialization for [ident] will be ignored.

Example of message 838:

```
If Line# Nl Source
-- ----- -- ------
       1          BLOCK DATA W838
       2          PARAMETER (N=3)
       3          COMMON/E607A/A(3)
       4          REAL B(2)
       5          INTEGER IOTA
       6          DATA(A(I),I=1,N)/1.,2.,3./
       7          DATA K/2/
       8          END
8: warn838.for: W838: The data initialization for K will be ignored.
```

Note: 839

Message:

The variable [ident] is assigned a statement label but is never used in an

assigned goto.

Example of message 839:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W839
       2          ASSIGN 10 TO IA
       3          GOTO 20
       4       10 A = 1
       5          WRITE(*,*) A
       6       20 RETURN
       7          END
7: warn839.for: N839: The variable IA is assigned a statement label but is
                      never used in an assigned goto.
```

Comment: 840

Message:

In this character assignment the variable [ident] is self-referential.

Example of message 840:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W840
       2          CHARACTER*40 A,B
       3          A='ABCDEFGHIJKLMNOPQRSTUVWXYZ01234567894*()-+'
       4          A(12:40) = A
4: warn840.for: C840: In this character assignment the variable A is self-
                      referential.
       5          B='NOT USED'
       6          A='0123456789'// A
6: warn840.for: C840: In this character assignment the variable A is self-
                      referential.
       7          WRITE(*,*) A
       8          END
```

Comment: 841

Message:

The switch value for this computed goto is a constant.

Example of message 841:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W841(I)
       2          PARAMETER (J = 2)
       3          GOTO (10,1,2),J
3: warn841.for: C841: The switch value for this computed goto is a constant.
       4          GOTO (10,1,2),
       5          GOTO (10,1,2),3
5: warn841.for: C841: The switch value for this computed goto is a constant.
       6        1 CONTINUE
       7        2 CONTINUE
       8       10 RETURN
       9          END
```

Note: 842

Message:

The variable [ident] is assigned a format label but is never used as an assigned format.

Example of message 842:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W842
       2          ASSIGN 10 TO IA
       3          GOTO IA,(10,20)
       4       10 A = 1
       5          ASSIGN 1 TO IA
       6          WRITE(*,1) A
       7        1 FORMAT(1X,F10.0)
       8          RETURN
       9          END
9: warn842.for: N842: The variable IA is assigned a format label but is
                      never used as an assigned format.
```

Note: 843

Message:

It is inefficient to have a statement label on an unconditional goto.

Example of message 843:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W843
       2          I=1
       3          IF(I.EQ.0) GO TO 20
       4       20 IF(I.NE.0) GO TO 25
       5       25 GO TO 30
5: warn843.for: N843: It is inefficient to have a statement label on an
                 unconditional goto.
       6       30 RETURN
       7          END
```

Note: 844

Message:

The executable statement number [label] was defined but not referenced.

Example of message 844:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W844
       2       10 A = 1.0
       3       20 FORMAT(1X,3F10.5)
       4          WRITE(*,20) A
       5          END
5: warn844.for: N844: The executable statement number 10 was defined but not
                      referenced.
```

Note: 845

Message:

The format number [label] was defined but not referenced.

Example of message 845:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W845
       2          A = 1.0
       3       20 FORMAT(1X,3F10.5)
       4          WRITE(*,*) A
       5          END
5: warn845.for: N845: The format number 20 was defined but not referenced.
```

Comment: 846

Message:

The actual argument list contains more than the [number] argument(s) used when the [object] [ident] was previously referenced or defined.

Example of message 846:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W846
       2          CALL W846A(A)
       3          CALL W846A(A,B)
3: warn846.for: C846: The actual argument list contains more than the 1
                      argument(s) used when the subroutine W846A was
                      previously referenced or defined.
       4          END
```

Note: 847

Message:

More than 19 continuation lines in one statement is nonstandard.

Example of message 847:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W847(A,B)
       2          DIMENSION B(25)
       3          A = B(1)
       4         $  + B(2)
       5         $  + B(3)
       6         $  + B(4)
       7         $  + B(5)
       8         $  + B(6)
       9         $  + B(7)
      10         $  + B(8)
      11         $  + B(9)
      12         $  + B(10)
      13         $  + B(11)
```

```
      14         $  + B(12)
      15         $  + B(13)
      16         $  + B(14)
      17         $  + B(15)
      18         $  + B(16)
      19         $  + B(17)
      20         $  + B(18)
      21         $  + B(19)
      22         $  + B(20)
      23         $  + B(21)
      24         $  + B(22)
      25         $  + B(23)
      26         $  + B(24)
      27         $  + B(25)
3: warn847.for: N847: More than 19 continuation lines in one statement is
                      nonstandard.
      28          WRITE(*,*) A
      29          END
```

Comment: 848

Message:

The variable [ident] in the assign is either an argument or is in common or is equivalenced.

Example of message 848:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W848(IOTA)
       2          COMMON/JOT/JOT
       3          EQUIVALENCE(JOT,NAM)
       4          ASSIGN 10 TO IOTA
4: warn848.for: C848: The variable IOTA in the assign is either an argument

                      or is in common or is equivalenced.
       5          GOTO IOTA
       6       10 ASSIGN 20 TO JOT
6: warn848.for: C848: The variable JOT in the assign is either an argument
                      or is in common or is equivalenced.
       7          GOTO JOT
       8       20 ASSIGN 30 TO NAM
8: warn848.for: C848: The variable NAM in the assign is either an argument
                      or is in common or is equivalenced.
       9          GOTO NAM
      10       30 CONTINUE
      11          END
```

Note: 849

Message:

The use of debugging comments is nonstandard.

Example of message 849:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM W849
       2          I = 100 * 2
2: warn849.for: N849: The use of debugging comments is nonstandard.
```

```
       3    D     WRITE(*,'(1X,I5)') I
       4          STOP
       5          END
```

Note: 850

Message:

The use of comment characters other than * or C is nonstandard.

Example of message 850:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W850
       2          A = 10.0
       3    C     THIS IS A COMMENT
       4          B = 15.0
4: warn850.for: N850: The use of comment characters other than * or C is
                      nonstandard.
       5    $     THIS IS A COMMENT
       6          C = 20.0
       7    *     THIS IS A COMMENT
       8          RETURN
       9          END
```

Note: 851

Message:

The use of inline comments is nonstandard.

Example of message 851:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM W851
       2          DIMENSION IA(10)  ! This is the key vector
2: warn851.for: N851: The use of inline comments is nonstandard.
       3          WRITE(1) IA
       4          STOP
       5          END
```

Note: 852

Message:

The use of multiple statements on a single line is nonstandard.

Example of message 852:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM W852
       2          IA = 10; IB = 15
2: warn852.for: N852: The use of multiple statements on a single line is
                      nonstandard.
       3          WRITE(*,'(1X,2I5)') IA,IB
       4          STOP
       5          END
```

Note: 853

Message:

The standard delimeter for a character constant is the single quote.

Example of message 853:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W853
       2          CHARACTER*9 A
       3          A="ABCDEF"
3: warn853.for: N853: The standard delimeter for a character constant is the
                      single quote.
       4          END
```

Note: 854

Message:

The use of r or l character constants is nonstandard.

Example of message 854:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM W854
       2          CHARACTER*10 IA
       3          DATA IA/L"ABC"/
3: warn854.for: N854: The use of r or l character constants is nonstandard.
       4          WRITE(1,'(1X,A3)') IA
       5          STOP
       6          END
```

Note: 855

Message:

The use of radix-50 constants is nonstandard.

Example of message 855:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM W855
       2          DATA IA/3RABC/
2: warn855.for: N855: The use of radix-50 constants is nonstandard.
       3          WRITE(1,'(1X,I10)') IA
       4          STOP
       5          END
```

Note: 856

Message:

A logical or arithmetic if with less than 3 branches is nonstandard.

Example of message 856:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM W856
       2          I=0
       3          IF(I.EQ.0) 5,6
3: warn856.for: N856: A logical or arithmetic if with less than 3
                      branches is nonstandard.
       4        5 WRITE(*,*) I
       5        6 CONTINUE
       6          END
```

Note: 857

Message:

The parenthetical form of the program statement is nonstandard.

Example of message 857:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM W857(OUTPUT,TAPE99,INPUT)
1: warn857.for: N857: The parenthetical form of the program statement is
                      nonstandard.
       2          A=1
       3          WRITE(*,*) A
       4          END
```

Note: 858

Message:

The identifier [token] with more than 6 characters is nonstandard.

Example of message 858:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W858
       2          I=1
       3          JABCDEF=I
3: warn858.for: N858: The identifier JABCDEF with more than 6 characters is
                      nonstandard.
       4          WRITE(*,*) I,JABCDEF
       5          END
```

Note: 859

Message:

The continuation of the end statement is nonstandard.

Example of message 859:

```
 If Line# Nl Source
 -- ----- -- ------
        1          SUBROUTINE W859
        2          DATA A/2.0/
        3          WRITE(*,*) A
        4          E
        5         1ND
4: warn859.for: N859: The continuation of the end statement is nonstandard.
```

Note: 860

Message:

The use of return in the main program is nonstandard.

Example of message 860:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM W860
       2          I=1
       3          WRITE(*,*) I
       4          RETURN
4: warn860.for: N860: The use of return in the main program is nonstandard.
       5          END
```

Comment: 861

Message:

Data is being allocated to common storage via the variable [ident].

Example of message 861:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W861
       2          COMMON A,D
       3          EQUIVALENCE (C,A)
       4          DATA D,B,C/1.0,2.0,3.0/
       5          WRITE(*,*) A,B,D
       6          END
6: warn861.for: C861: Data is being allocated to common storage via the
                      variable D.
6: warn861.for: C861: Data is being allocated to common storage via the
                      variable C.
```

Note: 862

Message:

Initializing the function value via a data statement is nonstandard.

Example of message 862:

```
If Line# Nl Source
-- ----- -- ------
       1          FUNCTION W862()
       2          DATA W862/4./
2: warn862.for: N862: Initializing the function value via a data statement
```

```
                    is nonstandard.
       3          WRITE(*,*) W862
       4          END
```

Comment: 863

Message:

The data variable list is longer than the value list, extra ignored.

Example of message 863:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W863
       2          INTEGER I(6),J(5),K(4)
       3          DATA Y,Z/1./
3: warn863.for: C863: The data variable list is longer than the value list,
                    extra ignored.
       4          DATA I/5*0/
4: warn863.for: C863: The data variable list is longer than the value list,
                    extra ignored.
       5          DATA J/2,3/
5: warn863.for: C863: The data variable list is longer than the value list,
                    extra ignored.
       6          DATA K/1,2,3,4/
       7          WRITE(*,*) I,J,Y,Z
       8          END
```

Note: 864

Message:

Declarative statements following executable statements is nonstandard.

Example of message 864:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W864
       2          X=1.3
       3          REAL I(10)
3: warn864.for: N864: Declarative statements following executable statements
                    is nonstandard.
       4          WRITE(*,*) X,I
       5          END
```

Note: 865

Message:

Specification statements following data statements is nonstandard.

Example of message 865:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W865
       2          DATA A/5.6/
       3          REAL B
```

```
3: warn865.for: N865: Specification statements following data statements is
                      nonstandard.
       4          WRITE(*,*) A
       5          END
```

Comment: 866

Message:

The [type] type has previously been assigned to [ident].

Example of message 866:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W866(AB)
       2          INTEGER A,Z(2)
       3          CHARACTER*5 A
3: warn866.for: C866: The integer*4 type has previously been assigned to A.
       4          COMPLEX Z
4: warn866.for: C866: The integer*4 type has previously been assigned to Z.
       5          AB=1.
       6          END
```

Note: 867

Message:

The argument [ident] has an expression as an adjustable dimension.

Example of message 867:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W867(A,B,C,D,I)
       2          COMMON J,K
       3          REAL A(I),B(J+1),C(K)
       4          WRITE(*,*) A,B,C,D

       5          END
5: warn867.for: N867: The argument B has an expression as an adjustable
                      dimension.
```

Note: 868

Message:

Implicit statements following specification statements is nonstandard.

Example of message 868:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W868(SAM,A)
       2          INTEGER SAM
       3          REAL A(SAM)
       4          IMPLICIT COMPLEX(S)
4: warn868.for: N868: Implicit statements following specification statements
```

```
                    is nonstandard.
       5          WRITE(*,*) A
       6          END
```

Warning: 869

Message:

The do index variable [ident] is already the index for an outer loop.

Example of message 869:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W869
       2          REAL A(10)
       3          DATA A/10*4.5/
       4          DO 10 J=1,10
       5  1    10 WRITE(6,*)(A(J),J=1,3)
       6          DO 20 I=1,10
       7  1       DO 20 I=1,10
7: warn869.for: W869: The do index variable I is already the index for an
                      outer loop.
       8  2    20 A(I)=I
       9          END
```

Comment: 870

Message:

The array [ident] is being subscripted with less than [number] expressions.

Example of message 870:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W870
       2          REAL Y(10,10),Z(6,3)
       3          DATA Z(3)/2./
3: warn870.for: C870: The array Z is being subscripted with less than 2
                      expressions.
       4          DO 10 J=1,100
       5  1    10 Y(J)=0.
5: warn870.for: C870: The array Y is being subscripted with less than 2
                      expressions.
       6          WRITE(*,*) Y,Z
       7          END
```

Note: 871

Message:

The statement function argument [ident] is never used.

Example of message 871:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W871(A,B,C)
       2          ASF(G)=B+A+C
```

```
2: warn871.for: N871: The statement function argument G is never used.
       3          A=SIN(B)+ASF(1.0)
       4          RETURN
       5          END
```

Note: 872

Message:

Equivalencing [ident] of type [type] with [ident] of type [type] is nonstandard.

Example of message 872:

```
If Line# Nl Source
-- ----- -- ------

       1          SUBROUTINE W872
       2          DOUBLE PRECISION D
       3          COMPLEX C
       4          CHARACTER*10 S1,S2,S3,S4,S5
       5          LOGICAL L
       6          INTEGER I
       7          REAL R
       8          EQUIVALENCE(I,S1)
       9          EQUIVALENCE(R,S2)
      10          EQUIVALENCE(D,S3)
      11          EQUIVALENCE(C,S4)
      12          EQUIVALENCE(L,S5)
      13          WRITE(*,*) 'HI'
      14          END
14: warn872.for: N872: Equivalencing S1 of type character*10 with I of type
                       integer*4 is nonstandard.
14: warn872.for: N872: Equivalencing S2 of type character*10 with R of type
                       real*4 is nonstandard.
14: warn872.for: N872: Equivalencing S3 of type character*10 with D of type
                       real*8 is nonstandard.
14: warn872.for: N872: Equivalencing S4 of type character*10 with C of type
                       complex*8 is nonstandard.
14: warn872.for: N872: Equivalencing S5 of type character*10 with L of type
                       logical*4 is nonstandard.
```

Comment: 873

Message:

The equivalence with [ident] is expanding the size of variable [ident] by [number] bytes.

Example of message 873:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W873
       2          REAL X(10),Y(13)
       3          EQUIVALENCE (X(5),Y(1))
       4          WRITE(*,*) X,Y,'HI'
       5          END
5: warn873.for: C873: The equivalence with Y is expanding the size of
                      variable X by 28 bytes.
```

Comment: 874

Message:

The equivalence with [ident] is expanding the size of common [ident] by [number] bytes.

Example of message 874:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W874
       2          REAL X(10)
       3          COMMON/XX/Y
       4          EQUIVALENCE (X(9),Y)
       5          WRITE(*,*) 'HI'
       6          END
6: warn874.for: C874: The equivalence with X is expanding the size of common
                      XX by 4 bytes.
```

Note: 875

Message:

Omitting commas from the [statement] statement is nonstandard.

Example of message 875:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W875

       2          IMPLICIT INTEGER(A)REAL(I)
2: warn875.for: N875: Omitting commas from the implicit statement is
                      nonstandard.
       3          AA=2
       4          II=4
       5          WRITE(*,*) AA,II
       6          END
```

Comment: 876

Message:

Common block [ident] is [number] bytes longer than when previously defined.

Example of message 876:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W876
       2          COMMON/E156X/A(99)
       3          CHARACTER*21 B,C,E*3,F*3
       4          COMMON/E156Y/B
       5          COMMON/E156Z/E
       6          E = 'EEE'
       7          F = 'FFF'
       8          WRITE(*,*) E,F
       9          END
      10          SUBROUTINE W876A
      11          COMMON/E156X/A(100
```

```
      12          CHARACTER*21B,C,E*3,F*3
      13          COMMON/E156Y/B,C
      14          COMMON /E156Z/E,F
      15          E = 'EEE'
      16          F = 'FFF'
      17          WRITE(*,*) E,F
      18          END
18: warn876.for: C876: Common block E156X is 4 bytes longer than when
                       previously defined.
18: warn876.for: C876: Common block E156Y is 21 bytes longer than when
                       previously defined.
18: warn876.for: C876: Common block E156Z is 3 bytes longer than when
                       previously defined.
```

Comment: 877

Message:

Common block [ident] is [number] bytes shorter than when previously defined.

Example of message 877:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W877
       2          COMMON/W877X/A(100)
       3          CHARACTER*21 B,C,E*3,F*3
       4          COMMON/W877Y/B,C
       5          COMMON /W877Z/E,F
       6          E = 'EEE'
       7          F = 'FFF'
       8          WRITE(*,*) E,F
       9          END
      10          SUBROUTINE W877A
      11          COMMON/W877X/A(99)
      12          CHARACTER*21 B,C,E*3,F*3
      13          COMMON/W877Y/B
      14          COMMON/W877Z/E
      15          E = 'EEE'
      16          F = 'FFF'
      17          WRITE(*,*) E,F
      18          END
18: warn877.for: C877: Common block W877X is 4 bytes shorter than when
                       previously defined.
18: warn877.for: C877: Common block W877Y is 21 bytes shorter than when
                       previously defined.
18: warn877.for: C877: Common block W877Z is 3 bytes shorter than when
                       previously defined.
```

Comment: 878

Message:

The [statement] statement is ignored in block data subprograms.

Example of message 878:

```
If Line# Nl Source
-- ----- -- ------
       1          BLOCK DATA W878
       2          COMMON/PI/PI
```

```
       3          DATA PI/3.141592/
       4          PI=3.14
4: warn878.for: C878: The assignment statement is ignored in block data
                      subprograms.
       5          CALL E159A
5: warn878.for: C878: The call statement is ignored in block data
                      subprograms.
       6          END
```

Note: 879

Message:

The conditional expression was of type [type] rather than logical.

Example of message 879:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W879(A,B)
       2          IF(A+B)GO TO 10
2: warn879.for: N879: The conditional expression was of type real*4 rather
                      than logical.
       3          B = 4.2
       4      10  IF(A.LE.10.)B=0.
       5          END
```

Note: 880

Message:

The use of redundant commas to indicate missing arguments is nonstandard.

Example of message 880:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W880(A,B)
       2          A=W880B(A,)
2: warn880.for: N880: The use of redundant commas to indicate missing
                      arguments is nonstandard.
       3          WRITE(*,*) A,B
       4          END
```

Note: 881

Message:

The [type] control value for the computed goto has been truncated to integer.

Example of message 881:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W881(I,J)
       2          REAL I
       3          DOUBLE PRECISION J
       4          GOTO(10,20,30),I
4: warn881.for: N881: The real*4 control value for the computed goto has
                      been truncated to integer.
```

```
       5       10 I=7.8
       6          RETURN
       7       20 GO TO(10,30),J
7: warn881.for: N881: The real*8 control value for the computed goto has
                      been truncated to integer.
       8          RETURN
       9       30 I=07.8
      10          RETURN
      11          END
```

Comment: 882

Message:

A value of one will be assumed for the zero format count field.

Example of message 882:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W882
       2          WRITE(*,6)
       3        6 FORMAT(3X,0F10.6)
3: warn882.for: C882: A value of one will be assumed for the zero format
                      count field.
       4          WRITE(*,8)
       5        8 FORMAT(3X,0('ABC',3X))
5: warn882.for: C882: A value of one will be assumed for the zero format
                      count field.
       6          END
```

Comment: 883

Message:

There is a T format specification within a repeated parenthetical group.

Example of message 883:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W883
       2          INTEGER K(6)
       3          DATA J,K/0,1,2,3,4,5,6/
       4          WRITE(*,10) J,(K(N),N=1,6)
       5          WRITE(*,20) J,(K(N),N=1,2)
       6       10 FORMAT(1X,I10,3(T5,I3,I5))
6: warn883.for: C883: There is a T format specification within a repeated
                      parenthetical group.
       7       20 FORMAT(1X,I10,/,(T5,I3,I5))
       8          END
```

Note: 884

Message:

The use of zero length character constants is nonstandard.

Example of message 884:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W884
       2       10 FORMAT(' ','')
2: warn884.for: N884: The use of zero length character constants is
                      nonstandard.
       3       20 FORMAT(' ',"")
3: warn884.for: N884: The use of zero length character constants is
                      nonstandard.
       4       30 FORMAT(' ',**)
4: warn884.for: N884: The use of zero length character constants is
                      nonstandard.
       5          CHARACTER*2 A,B
       6          A=''
6: warn884.for: N884: The use of zero length character constants is
                      nonstandard.
       7          WRITE(*,10) A
       8          WRITE(*,20) A
       9          WRITE(*,30) B
      10          END
```

Note: 885

Message:

An implicit high-low range specification is nonstandard.

Example of message 885:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE E885(A)
       2          IMPLICIT INTEGER(D-A)
2: warn885.for: N885: An implicit high-low range specification is
                      nonstandard.
       3          A=1
       4          END
```

Note: 886

Message:

Including a repeat count before the [token] format specification is nonstandard.

Example of message 886:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W886
       2          WRITE(*,10)
       3       10 FORMAT(1X,1S,I1,6,2SP,I2,3SS,I3)

3: warn886.for: N886: Including a repeat count before the S format
                      specification is nonstandard.
3: warn886.for: N886: Including a repeat count before the , format
                      specification is nonstandard.
3: warn886.for: N886: Including a repeat count before the S format
                      specification is nonstandard.
3: warn886.for: N886: Including a repeat count before the S format
                      specification is nonstandard.
       4          WRITE(*,20)
```

```
       5       20 FORMAT(10T10,6'A',2TL2,5"B",6TR6,'C',7)
5: warn886.for: N886: Including a repeat count before the T format
                      specification is nonstandard.
5: warn886.for: N886: Including a repeat count before the ' format
                      specification is nonstandard.
5: warn886.for: N886: Including a repeat count before the T format
                      specification is nonstandard.
5: warn886.for: N886: Including a repeat count before the " format
                      specification is nonstandard.
5: warn886.for: N886: Including a repeat count before the T format
                      specification is nonstandard.
5: warn886.for: N886: Including a repeat count before the ) format
                      specification is nonstandard.
       6          WRITE(*,30)
       7       30 FORMAT(1X,E10.2,G20.10)
       8          READ(*,40) I
       9       40 FORMAT(2BZ,I2,2*A*,3BN)
9: warn886.for: N886: Including a repeat count before the * format
                      specification is nonstandard.
      10          END
```

Comment: 887

Message:

Output format contains a superfluous bn or bz specification.

Example of message 887:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W887
       2          IA=1
       3          WRITE(6,10)IA
       4       10 FORMAT (BZ,1X,I10)
4: warn887.for: C887: Output format contains a superfluous bn or bz
                      specification.
       5       20 FORMAT (1X,BZ,I10)
       6          WRITE (6,20) IA
6: warn887.for: C887: Output format contains a superfluous bn or bz
                      specification.
       7          END
```

Note: 888

Message:

The block data subprogram contained no data statements.

Example of message 888:

```
If Line# Nl Source
-- ----- -- ------
       1          BLOCK DATA W888
       2          COMMON/PI/PI
       3          COMMON/D/D(10)
       4          END
4: warn888.for: N888: The block data subprogram contained no data statements.
```

Note: 889

Message:

The use of zero-width or free-form format value editing specifications is nonstandard.

Example of message 889:

```
    If Line# Nl Source
    -- ----- -- ------
           1          SUBROUTINE W889
           2       10 FORMAT(F0.5)
    2: warn889.for: N889: The use of zero-width or free-form format value
                          editing specifications is nonstandard.
           3       30 FORMAT(I0)
    3: warn889.for: N889: The use of zero-width or free-form format value editing
                          specifications is nonstandard.
           4       40 FORMAT(L0)
    4: warn889.for: N889: The use of zero-width or free-form format value editing
                          specifications is nonstandard.

          5       60 FORMAT(D0.0)

   5: warn889.for: N889: The use of zero-width or free-form format value editing

                         specifications is nonstandard.
           6       70 FORMAT(E0.0)
     6: warn889.for: N889: The use of zero-width or free-form format value editing
                          specifications is nonstandard.
           7       80 FORMAT(G0.0)
     7: warn889.for: N889: The use of zero-width or free-form format value editing
                          specifications is nonstandard.
           8          WRITE(*,10)
           9          WRITE(*,30)
          10          WRITE(*,40)
          11          WRITE(*,60)
          12          WRITE(*,70)
          13          WRITE(*,80)
          14          END
```

Comment: 890

Message:

The zero statement label is ignored.

Example of message 890:

```
    If Line# Nl Source
    -- ----- -- ------
           1          SUBROUTINE W890
       2: warn890.for: C890: The zero statement label ignored.
           2     0000 REAL A
       3: warn890.for: C890: The zero statement label ignored.
           3     0000 A=1.
           4          WRITE(*,*) A
           5          END
```

Comment: 891

Message:

The statement function [ident] has been previously declared as being external.

Example of message 891:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W891(I)
       2          EXTERNAL JJ
       3          JJ(K)=K**3/(K-I)**KK(K)
3: warn891.for: C891: The statement function JJ has been previously declared
                      as being external.
       4          LL(I)=KK(I)
       5          I=LL(I)-JJ(I)
       6          END
```

Note: 892

Message:

List-directed I/O for internal files is nonstandard.

Example of message 892:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W892
       2          CHARACTER*128 LINE
       3          A=3
       4          WRITE(LINE,*)'A=',A
4: warn892.for: N892: List-directed I/O for internal files is nonstandard.
      5          WRITE(*,*) LINE
      6          END
```

Warning: 893

Message:

There is an equivalence contradiction for [ident].

Example of message 893:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W893
       2          REAL A(10), B(20)
       3          EQUIVALENCE (A(1),B(10)),(A(10),B(5))
       4          WRITE(*,*) A,B
       5          END
5: warn893: for: W893: There is an equivalence contradiction for B.
```

Comment: 894

Message:

The do loop ends on a transfer statement.

Example of message 894:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W894
       2          INTEGER A(6)
       3          DATA J/1/
       4          DO 10 I=1,6
       5  1       A(I)=I
       6  1    10 IF(J-2)15,15,30
6: warn894.for: C894: The do loop ends on a transfer statement.
       7       15 DO 20 I=1,6
       8  1       A(I)=I
       9  1    20 ENDDO
      10       30 DO 60 I=1,6
      11  1       A(I)=I
      12  1    60 RETURN
12: warn894.for: C894: The do loop ends on a transfer statement.
      13       65 DO 70 I=1,6
      14  1       A(I)=I
      15  1    70 STOP 70
15: warn894.for: C894: The do loop ends on a transfer statement.
      16       75 DO 80 I=1,6
      17  1       A(I)=I
      18  1    80 GOTO(65,75,85,105)I

18: warn894.for: C894: The do loop ends on a transfer statement.
      19       85 DO 90 I=1,6
      20  1       ASSIGN 105 TO IOT
      21  1    90 GOTO IOT ,(105,65)
21: warn894.for: C894: The do loop ends on a transfer statement.
      22      105 DO 110 I=1,6
      23  1       A(I)=I
      24  1   110 DO 115 J = 1,5
      25  2       WRITE(*,*) J
      26  2   115 CONTINUE
26: warn894.for: C894: The do loop ends on a transfer statement.
      27          END
```

Comment: 895

Message:

The character [token] has already been assigned type [type].

Example of message 895:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W895(A)
       2          IMPLICIT INTEGER(A-Z),REAL(I-J)
2: warn895.for: C895: The character I has already been assigned type
                      integer*4.
2: warn895.for: C895: The character J has already been assigned type
                      integer*4.
       3          A=1
       4          END
```

Note: 896

Message:

References to nonconstants like [object] [ident] in parameter statements is nonstandard.

Example of message 896:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W896
       2          PARAMETER (I=3,K=J)
2: warn896.for: N896: References to nonconstants like variable J in
                      parameter statements is nonstandard.
       3          PARAMETER(L=J+3)
3: warn896.for: N896: References to nonconstants like variable J in
                      parameter statements is nonstandard.
       4          PARAMETER(A=1/3.,B=1./A)
       5          J = 99
       6          WRITE(*,*) I,K,L,A,B,Z,W,Y
       7          END
```

Comment: 897

Message:

The actual argument list contains less than the [number] argument(s) used when the [object] [ident] was previously referenced or defined.

Example of message 897:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W897
       2          CALL DEMO(A,B)
       3          CALL DEMO(A)
3: warn897.for: C897: The actual argument list contains less than the 2
                      argument(s) used when the subroutine DEMO was
                      previously referenced or defined.
       4          END
```

Comment: 898

Message:

The statement labels on elseif, else, or endif may not produce the desired effect.

Example of message 898:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W898
       2          DATA A/2.0/
       3          IF(A.GE.2.)THEN
       4  1       I=3
       5  1       GOTO (10,30,40),I
       6  1    10 ELSEIF(A.LT.2.)THEN
6: warn898.for: C898: The statement labels on elseif, else, or endif may not
                      produce the desired effect.
       7  1       A=2.
       8  1    30 ELSE
8: warn898.for: C898: The statement labels on elseif, else, or endif may not
                      produce the desired effect.
       9  1       A=4.
      10  1    40 ENDIF
10: warn898.for: C898: The statement labels on elseif, else, or endif may not
                       produce the desired effect.
      11          END
```

Comment: 899

Message:

The subprogram [ident] is not an intrinsic function.

Example of message 899:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W899
       2          INTRINSIC INT,IFIX,IDINT,XREAL,FLOAT,SNGL,DBLE,CMPLX,ECHAR
2: warn899.for: C899: The subprogram XREAL is not an intrinsic function.
2: warn899.for: C899: The subprogram ECHAR is not an intrinsic function.
       3          A=3.1415
       4          WRITE(*,*) A
       5          END
```

Warning: 900

Message:

An implied do must be used for the assumed size array [ident].

Example of message 900:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W900(C,D)
       2          REAL C(*),D(10,*)
       3          CALL E569A(C,D)
       4          WRITE(2) (C(I),I=1,10), D
4: warn900.for: W900: An implied do must be used for the assumed size array
                      D.
4: warn900.for: C870: The array D is being subscripted with less than 2
                      expressions.
       5          END
```

Warning: 901

Message:

Only the real part of a complex value is used in an arithmetic if.

Example of message 901:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W901(C)
       2          COMPLEX C
       3          IF(C**2)10,20,20
3: warn901.for: W901: Only the real part of a complex value is used in an
                  arithmetic if.
       4       20 RETURN
       5       10 END
```

Warning: 902

Message:

The global symbol [ident] is being used to represent both a subprogram and a common area.

Example of message 902:

```
If Line# Nl Source
-- ----- -- ------
       1          SUBROUTINE W902
       2          COMMON/W902/A,B,C
       3          A = B + C
       4          RETURN
       5          END
5: warn902.for: W902: The global symbol W902 is being used to represent
                      both a subprogram and a common area.
```

Warning: 903

Message:

The equivalence element subscript value [number] for subscript [number] of array [ident] with minimum of [number] and maximum of [number] is out of range.

Example of message 903:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM W903
       2          PARAMETER(NA = 10, IA = 11)
       3          DIMENSION AA(NA,6), BB(4)
       4          EQUIVALENCE (BB,AA(IA,2))
4: warn903.for: W903: The equivalence element subscript value 11 for
                      subscript 1 of array AA with minimum of 1 and maximum
                      of 10 is out of range.
       5          BB(3) = 99
       6          STOP
       7          END
```

Warning: 904

Message:

The formal argument list contains more than the [number] arguments used when this subprogram was previously referenced or defined.

Example of message 904:

```
If Line# Nl Source
-- ----- -- ------
       1          PROGRAM W904
       2          CALL SUB904(A,B)
       3          STOP
       4          END
       5          SUBROUTINE SUB904(A,B,C,D)
5: warn904.for: W904: The formal argument list contains more than the 2
                      arguments used when this subprogram was previously
                      referenced or defined.
       6          RETURN
```

```
      7         END
```

9.2.2 Fatal Preprocessor Errors

Fatal preprocessor errors; occur when requested or needed files cannot be opened or when available memory has been exhausted. When a fatal error occurs, the preprocessor exits to the control program. No intermediate C files are produced.

File problems cannot be avoided, but are usually simple to correct. The preprocessor always gives the name of the file it is trying to open when such a problem occurs.

The preprocessor is as conservative as possible in its memory utilization. Extremely large source files can be processed. Typically, other FORTRAN processors have much tighter memory constraints. There are several command line options which allow control of memory allocation for systems with limited memory. Refer to the chapter on using the compiler for a discussion of these options should the need arise.

The fatal errors have the same general form as the syntax messages, except that the identifying letter is an "F". The messages themselves are self-explanatory and are listed below:

Number	Message
501	Insufficient procedure storage for promotion operator.
502	Insufficient definition storage for main control table.
503	Insufficient symbol table storage.
504	Insufficient value storage for scalar data value.
505	Insufficient value storage for array values.
506	Insufficient definition storage for dummy table.
507	Insufficient definition storage for dimension control table.
508	Insufficient definition storage for common table.
509	Insufficient definition storage for prototype table.
510	Insufficient procedure storage for next opcode.
511	Insufficient procedure storage for opcode support word.
512	Insufficient procedure storage for opcode support byte.
513	Insufficient procedure storage for opcode parameters.
514	Insufficient value storage for integer constant.
515	Insufficient storage for parameter type list.
516	Insufficient user space for common size table.
517	Insufficient value storage for runtime data string.
518	Insufficient value storage for include following record.
519	Insufficient unresolved external symbol storage space.
520	Insufficient procedure storage for required quantity storage.
521	Include statement files nested more than 5 deep.
522	Statement contains more than \1 characters – use the QIn flag to increase this value
523	Constant string length exceeds 255 characters.
524	Length of scratch storage exceeded during namelist processing.
525	The DEMO license does not allow use of the include statement.
526	The PILOT license can only include files less than 1000 bytes long.
527	Insufficient value storage for character variable length specification.
528	Insufficient system memory to continue processing. The source code being processed must be shortened.
529	Insufficient value storage for alternate symbol display name.
530	Insufficient value storage for symbol reference information control table.
531	The include file storage of \1 bytes is insufficient, use the QHn flag to increase it.

9.3 Runtime Error Messages

Runtime error messages occur when an error occurs during the execution of programs formed using the PROMULA FORTRAN compiler. These are contained within the runtime library and are fatal. Each message has a code and a short description as follows:

Code	Text of message	Code	Text of message
ELUN_EOF	end of file encountered	EWRS_IFS	bad format specification
EOPN_EOF	end of file encountered	EWRT_IFS	bad format specification
ERBV_EOF	end of file encountered	EWVL_IFS	bad format specification
ERTX_EOF	end of file"	ENXF_EFS	bad of format control character
EWTX_EOF	write beyond end of file	ENXF_BTF	bad T format
ELUN_RDO	write to readonly file	ENXF_BUS	bad B business format string
EINT_NAF	no active file structure	ENXF_BBF	bad BN,Z format
ELUN_NAF	no active file structure	EOPN_LNR	unit number out of range
EINT_TMF	too many files open	E XF_DEL	mising terminating delimeter
ELUN_TMF	too many files open	ENXF_HOL	bad Hollerith string
ENAM_TMF	too many files open	ERCK_BUF	internal buffer exceeded
ESIO_TMF	too many files open	ERDX_MLP	missing left parenthesis
ECLO_PCF	physical close failure	ERDX_COM	missing comma
EOPN_POF	physical open failed	ERDX_MRP	missing right parenthesis
ECLO_POF	physical open failure	ERDZ_MLP	missing left parenthesis
EBCK_FRT	at front of file	ERDZ_COM	missing comma
EBCK_DIR	direct access file	ERDZ_MRP	missing right parenthesis
ELUN_PWF	physical write failed	ERNL_MNI	missing namelist identifier
ERWV_PWF	physical write failure	ERNL_MVI	missing variable identifier
EWBV_PWF	physical write error	ERNL_UVI	undefined variable identifier
EWEF_PWF	physical write failure	ERNL_SSV	subscripted scalar variable
ERDB_IFS	invalid format specification	ERNL_NNS	non-numeric subscripts
ERDD_IFS	invalid format specification	ERNL_TMS	too many subscripts
ERDF_IFS	invalid format specification	ERNL_EQL	missing equals sign
ERDI_IFS	invalid format specification	ERNL_BSI	bad string input
ERDL_IFS	invalid format specification	ERNL_MLP	complex missing left pren
ERDS_IFS	invalid format specification	ERNL_COM	complex missing comma
ERDT_IFS	invalid format specification	ERNL_MRP	complex missing right pren
EWRB_IFS	bad format specification	ESTD_NNC	non-numeric character in field
		EOPN_BFZ	setting buffer size failed