Extended Disk Fortran

User’s Manual
IMPORTANT NOTICE

This manual, and the programs it describes, are copyrighted by Processor Technology Corporation. All rights are reserved. Processor Technology software packages are distributed through authorized dealers solely for sale to individual retail customers. Wholesaling of these packages is not permitted under the agreement between Processor Technology and its dealers. No license to copy or duplicate is granted with distribution or subsequent sale.
<table>
<thead>
<tr>
<th>SECTION</th>
<th>CONTENTS</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>INTRODUCTION</td>
<td>1-1</td>
</tr>
<tr>
<td>1.0</td>
<td>INTRODUCTION</td>
<td>1-1</td>
</tr>
<tr>
<td>1.1</td>
<td>HOW TO USE THIS BOOK</td>
<td>1-1</td>
</tr>
<tr>
<td>1.2</td>
<td>SYMBOLS AND CONVENTIONS</td>
<td>1-2</td>
</tr>
<tr>
<td>1.3</td>
<td>SYSTEM REQUIREMENTS</td>
<td>1-3</td>
</tr>
<tr>
<td>1.4</td>
<td>SOFTWARE CONFIGURATION</td>
<td>1-3</td>
</tr>
<tr>
<td>2</td>
<td>THE PTDOS FORTRAN LANGUAGE</td>
<td>2-1</td>
</tr>
<tr>
<td>2.0</td>
<td>INTRODUCTION</td>
<td>2-1</td>
</tr>
<tr>
<td>2.1</td>
<td>CHARACTERISTICS OF PTDOS FORTRAN STATEMENTS</td>
<td>2-1</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Statements and Statement Lines</td>
<td>2-1</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Continuations</td>
<td>2-1</td>
</tr>
<tr>
<td>2.1.3</td>
<td>Comments</td>
<td>2-2</td>
</tr>
<tr>
<td>2.1.4</td>
<td>Labels</td>
<td>2-2</td>
</tr>
<tr>
<td>2.1.5</td>
<td>Characters in FORTRAN Programs</td>
<td>2-2</td>
</tr>
<tr>
<td>2.2</td>
<td>ELEMENTS OF THE LANGUAGE</td>
<td>2-3</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Constants</td>
<td>2-3</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Variables</td>
<td>2-4</td>
</tr>
<tr>
<td>2.2.3</td>
<td>Expressions</td>
<td>2-5</td>
</tr>
<tr>
<td>2.2.4</td>
<td>Order of Evaluation in Expressions</td>
<td>2-5</td>
</tr>
<tr>
<td>2.2.5</td>
<td>Integer Expressions</td>
<td>2-6</td>
</tr>
<tr>
<td>2.2.6</td>
<td>Real Expressions</td>
<td>2-7</td>
</tr>
<tr>
<td>2.2.7</td>
<td>Logical Expressions</td>
<td>2-7</td>
</tr>
<tr>
<td>2.2.8</td>
<td>Internal Formats and Ranges of Values</td>
<td>2-8</td>
</tr>
<tr>
<td>3</td>
<td>PREPARING THE SOURCE PROGRAM FILE</td>
<td>3-1</td>
</tr>
<tr>
<td>3.0</td>
<td>INTRODUCTION</td>
<td>3-1</td>
</tr>
<tr>
<td>3.1</td>
<td>CREATING A PROGRAM</td>
<td>3-1</td>
</tr>
<tr>
<td>3.2</td>
<td>FORMAT OF A PROGRAM FILE</td>
<td>3-2</td>
</tr>
<tr>
<td>4</td>
<td>HOW TO COMPILE, ASSEMBLE, AND EXECUTE A FORTRAN PROGRAM</td>
<td>4-1</td>
</tr>
<tr>
<td>4.0</td>
<td>INTRODUCTION</td>
<td>4-1</td>
</tr>
<tr>
<td>4.1</td>
<td>COMPILATION AND ASSEMBLY</td>
<td>4-1</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Parameters in the FORTRAN Command Line</td>
<td>4-2</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Compilation Errors</td>
<td>4-3</td>
</tr>
<tr>
<td>4.2</td>
<td>EXECUTION</td>
<td>4-3</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Runtime Errors</td>
<td>4-4</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Large Programs</td>
<td>4-4</td>
</tr>
<tr>
<td>5</td>
<td>PTDOS FORTRAN STATEMENTS</td>
<td>5-1</td>
</tr>
<tr>
<td>5.0</td>
<td>INTRODUCTION</td>
<td>5-1</td>
</tr>
<tr>
<td>5.1</td>
<td>ASSIGNMENT STATEMENTS</td>
<td>5-1</td>
</tr>
</tbody>
</table>
6.5 OTHER UTILITY SUBROUTINES ........................................ 6-14
    MOVE, CBTOF, BIT

7 SYSTEM FUNCTIONS ...................................................... 7-1
7.0 INTRODUCTION ....................................................... 7-1
7.1 GENERAL MATHEMATICAL FUNCTIONS ................................. 7-1
7.2 TRIGONOMETRIC FUNCTIONS .......................................... 7-3
    SIN, COS, TAN, ATAN, ATAN2
7.3 COMPARING CHARACTER STRINGS ...................................... 7-3
    COMP
7.4 EXECUTING ASSEMBLY-LANGUAGE PROGRAMS ........................ 7-4
    CALL

8 ASSEMBLY-LANGUAGE INTERFACE ........................................ 8-1

APPENDICES

1 FORTRAN STATEMENT SUMMARY .......................................... A1-1
2 SUMMARY OF SYSTEM SUBROUTINES .................................... A2-1
3 TABLE OF FUNCTIONS .................................................. A3-1
4 DEVICE INTERFACE ..................................................... A4-1
5 COMPILATION ERROR MESSAGES ........................................ A5-1
6 EXECUTION ERRORS ..................................................... A6-1
7 COMPARISON OF PTDOS AND ANSI STANDARD FORTRAN ............. A7-1
8 BIBLIOGRAPHY .......................................................... A8-1
SECTION 1

INTRODUCTION

1.0. FORTRAN for PTDOS

Extended Disk FORTRAN is an 8080 version of FORTRAN that uses the Helios II disk memory system. The combination of the powerful FORTRAN language and the fast random access memory of Helios II offers users a unique problem-solving system.

Extended Disk FORTRAN is used as a subsystem of the Processor Technology Disk Operating System, PTDOS, and will be called PTDOS FORTRAN throughout this manual. Many other subsystems of PTDOS are available to the FORTRAN user. Users of PTDOS FORTRAN can create programs quickly and easily in PTDOS editors, or work with existing FORTRAN source files in either normal or ALS-8 format.

PTDOS FORTRAN gives you access to all the disk operating system capabilities from your FORTRAN program, including:

- File management, including creating, killing, and changing attributes.
- Random access to data files.
- Input from and output to device files.

The special capabilities of the PTDOS FORTRAN language itself include:

- Free format input and output.
- Character string data type and a string comparison function.
- A COPY statement to copy files of source statements into a FORTRAN source program.
- Assembly language interface. Assembly language statements can be included in the source file and assembly routines can be called from the FORTRAN program.
- Direct control over the video display.
- Access to absolute memory locations, including individual bits.
- Program-controlled time delay.
- A pseudo-random number generator function.
- Program control of runtime error trapping.
- Ability to chain a sequence of programs.

PTDOS FORTRAN is both a subset and a superset of ANSI standard FORTRAN. To adapt FORTRAN to work efficiently on the Sol Terminal Computer, some ANSI standard features have been left out. On the other hand, there are added features that are particularly suited to the computer terminal - disk memory - video display system. A detailed comparison of PTDOS FORTRAN and version X3.9 - 1966 of ANSI standard FORTRAN appears in Appendix 7.

1.1. HOW TO USE THIS BOOK

This book presents PTDOS FORTRAN, describing how to use it within the PTDOS system. The basic definitions are given, as well as detailed descriptions of the statements, functions, and subroutines that make up PTDOS FORTRAN.
This book is not intended as an introduction to FORTRAN. Several introductory FORTRAN books are listed in Appendix 8 as recommended reading for new FORTRAN users.

Read this book from cover to cover first, as a text. The material is presented in order of use after the basic information and definitions in Section 2. After you are familiar with PTDOS FORTRAN, you can use this book as a reference. In addition, statement, subroutine, and function summaries are presented in Appendices 1, 2, and 3, respectively.

Section 2 describes the elements of PTDOS FORTRAN and gives the fundamental definitions.

Section 3 tells how to create a PTDOS FORTRAN source program. It describes the overall format of a source file and presents compilation options.

Section 4 describes alternative ways to compile, assemble, and execute PTDOS FORTRAN programs.

Section 5 presents each PTDOS FORTRAN statement. The statements are presented in order of increasing difficulty. This section is likely to be used often for reference.

Sections 6 and 7 describe system-supplied subroutines and functions, respectively. PTDOS FORTRAN subroutines in particular provide the capability for fast, efficient input and output, and control over disk files.

Section 8 tells how you can include assembly-language statements in a FORTRAN program and call assembly-language routines from a FORTRAN program.

1.2. SYMBOLS AND CONVENTIONS

The symbol <CR> is used in examples throughout this document to indicate that the user presses the carriage return key. For example:

User: FORTRAN FSOURCE,FLIST,,FOBJ <CR> The user types the line shown followed by a carriage return.

Command and statement forms use upper- and lowercase characters to differentiate between characters to be typed literally and terms indicating the type of information to be inserted. For example, the following statement form indicates that the word ENDFILE should be typed followed by a unit number selected by the user:

ENDFILE unit

Punctuation in command and statement forms should be interpreted literally. For example, the statement form below indicates that the word INTEGER should be followed by one or more variable names separated by commas:

INTEGER var1, var2, ...

The ellipses indicate an indefinite number of arguments.

Optional elements of command and statement forms are enclosed in braces. For example:

STOP {character string} The character string following the word STOP may be included or omitted, depending on the desired result.

The word "list" is sometimes used in statement forms to indicated one or more elements separated by commas. For example:

ACCEPT input list The word ACCEPT is followed by one or more input items separated by commas.
In formatted values, the letter b represents a single blank. For example:

```
b12bb
```
The number 12 is preceded by one blank and followed by two blanks.

1.3. SYSTEM REQUIREMENTS
PTDOS FORTRAN must be used as part of the PTDOS operating system. PTDOS and FORTRAN together require 32K of memory.

The hardware recommended is a Helios II disk memory system and its computer with two or more disks.

In addition to hardware, you should have several data diskettes and a Helios II Disk Memory System Manual.

The full version of FORTRAN is provided on a Processor Technology Corporation diskette. The FORTRAN software is contained on six files. The amount of memory used at any time depends on the files that are active, but at least 32K of memory should be provided.

1.4. SOFTWARE CONFIGURATION
All PTDOS system files and the following PTDOS FORTRAN files are required to use PTDOS FORTRAN:

- FORTRAN — FORTRAN compiler
- FORTEROR — compiler error message file
- FORTGO — runtime package for the quick compile option
- FORTDEFS — definitions needed for the quick compile option
- FORTRUN — source code for the long compile option
- RUNTIME — COPY statements for the runtime package of the long compile option

Assuming you start with a PTDOS System Diskette and a PTDOS FORTRAN Diskette, you should use GET or COPY commands to rearrange the files until you have the following configuration:

<table>
<thead>
<tr>
<th>Unit:</th>
<th>Default</th>
<th>0</th>
<th>1</th>
<th>Any</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTDOS system files</td>
<td>FORTGO</td>
<td>FORTRUN</td>
<td>FORTRAN</td>
<td></td>
</tr>
<tr>
<td>FORTEROR</td>
<td>FORTGO</td>
<td>FORTRUN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FORTDEFS</td>
<td>FORTGO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RUNTIME</td>
<td>FORTGO</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The default unit is usually 0, but you can reset it to any available unit. One example of a configuration that can be used for any FORTRAN operation is:

- **Unit 0 (Default)**
  - PTDOS system files
  - FORTEROR
  - FORTDEFS
  - RUNTIME
  - FORTGO
  - FORTRAN

- **Unit 1**
  - FORTRUN

FORTRAN is usually copied to a separate unit because it is a very large file. However, you can arrange to have all PTDOS and FORTRAN files on unit 0 by changing the COPY FORTRAN/1 statement in RUNTIME to COPY FORTRAN.

Before using PTDOS FORTRAN it is a good idea to set the lowest address of the buffer to about 8400 to allow more memory for FORTRAN operations, provided enough memory is available.

The steps listed below show one of many possible ways to prepare your system for PTDOS FORTRAN operations:
1. Connect the Sol Terminal Computer to a Helios II Disk System.

2. Supply power to the computer, disk system, and video display. The prompt > should appear on the screen indicating that the SOLOS or CUTER monitor program is active.

3. Insert the PTDOS System Diskette in Unit 0 of the HELIOS, then type:

   \texttt{BOOTSTRAP <CR>}

   The system loads PTDOS and displays messages followed by the prompt *.

4. Insert the PTDOS FORTRAN diskette in Unit 1 of HELIOS and copy all but FORTRUN to Unit 0:

   \texttt{GET /0,1=FORTRAN,RUNTIME,FORTGO,FORTDEFS,FORTEROR <CR>}

5. If possible, allow more memory space by decreasing the lowest buffer address:

   \texttt{SET BU = 8400 <CR> (or use the \texttt{CONFIGR} command)}
SECTION 2

THE PTDOS FORTRAN LANGUAGE

2.0. INTRODUCTION

The FORTRAN language consists of statements, which are individual instructions to the computer that can be arranged to describe a process. Such an arrangement of statements is called a program. For example, the following FORTRAN program contains 10 statements that are arranged to add a list of numbers:

C THIS PROGRAM ADDS NUMBERS
C
    TYPE 'ENTER AS MANY AS 100 NUMBERS'
    TYPE 'TO TERMINATE ENTRIES, TYPE 0'
    DO 10 I = 1,100
    ACCEPT '?', A
    IF (A .EQ. 0) GO TO 20
10    SUM = SUM + A
20    TYPE 'THE SUM IS ', SUM
    END

The above program consists of a “main routine” only. A FORTRAN program must include a main routine. It may also include subroutines and functions, which are sometimes called “subprograms” collectively. The term “routine” refers to any of the logical parts: main routine, subroutine, or function.

This section describes the characteristics of FORTRAN statements and the entities that these statements act on (constants, variables, and expressions).

2.1. CHARACTERISTICS OF PTDOS FORTRAN STATEMENTS

The example above demonstrates some of the properties of FORTRAN statements. The statements adhere to a fixed format and certain character positions have special meanings. The rules for FORTRAN statements are described in detail in the units that follow.

2.1.1. Statements and Statement Lines

The body of a statement line starts in column 7 and ends with a carriage return. It can be any length but only the first 72 characters are retained. Only one statement is allowed on a statement line.

A statement is a complete instruction to the computer and can be comprised in one or more statement lines. The entire statement can contain up to 530 characters including blanks.

2.1.2. Continuations

To continue a statement on an additional statement line, place any character except 0 or blank in column 6 of the new line. For example:

    DO 100 I =
    C1,20
    TYPE 'THIS STATEMENT IS CONTINUED
* BELOW'

Notice the blank before BELOW. Statement lines are not padded with blanks between the final carriage return and column 72, so a blank is necessary to produce CONTINUED BELOW instead of CONTINUEDBELOW.

2-1 FORTRAN
2.1.3. Comments
To include a comment in your program, place a C in column 1. This causes the rest of the line to be completely ignored by the compiler. It is a good practice to insert many comments in a long program to serve as documentation. For example:

    C Environmental data will now be read from "DATA3"
    CALL OPEN(5,'DATA3')
    READ (5,100)E1,E2,E3
    
2.1.4. Labels
Statement labels allow statements to be referred to by other statements. For example:

    GO TO 70
    
    70 ACCEPT 'ANSWER = ', ANSWER
    
    The labels need not be in any particular sequence. Statements should not be labelled unless they are referenced. A labelled statement that is not referenced produces a warning message during compilation.

A statement label must be an integer between 0 and 99999 placed anywhere in columns 1 through 5. The placement in columns 1 through 5 does not affect the value of the label. For example, the following statements are identical:

    10 CONTINUE
    10 CONTINUE

In any program or subprogram (subroutine or function), a given label can only be used once. Labels are unique to each logical block, so the same label that appears in the main program can also be used in a subroutine or function.

2.1.5. Characters in FORTRAN Programs
Any characters may appear in comment lines.

Characters that appear between single quotation marks in a FORTRAN statement are treated literally and are called character strings. There is one exception: when a pair of backslashes appear in a character string, the characters between them are interpreted as a hexadecimal constant. In character strings, lowercase characters are not converted to uppercase and blanks are retained.

In all other cases, blanks in FORTRAN statements are ignored. For example, the two statements below are identical:

    WRITE ( 1 , 200 ) A , B , C
    WRITE (1,200)A,B,C

Lower case letters that are not within a character string are converted to upper case during compilation. For example, the variable names below all represent the same variable:
Special Characters

The following characters have special meanings in PTDOS FORTRAN:

' Quantities enclosed in single quotes are treated as string constants.
* Statements preceded with an asterisk are interpreted as assembly language statements.
b Blanks are ignored, except between single quotes.
\ A constant enclosed in backslashes in a character string is assumed to be the hexadecimal code for an ASCII character.
& If a FORMAT statement contains an ampersand, the character following the ampersand is interpreted as a control character (unless it is also $).
$ Preceding a constant with a dollar sign indicates that it is a hexadecimal constant.
# Preceding a constant with a number sign indicates that it is a hexadecimal constant to be stored internally in binary format.

2.2. ELEMENTS OF THE LANGUAGE

FORTRAN statements perform operations on constants, variables, and expressions. Each represents a value stored in the computer. The units below discuss constants, variables, and expressions and describe how values are stored internally.

2.2.1. Constants

A constant is a quantity that has a fixed value. PTDOS FORTRAN has numerical, string, and logical constants. A numerical constant is an integer or real number, a string constant is a sequence of characters, and a logical constant represents a value of true or false.

Numerical Constants

A decimal numerical constant can be expressed in any of the following forms:

Examples

<table>
<thead>
<tr>
<th>Type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>1, -4000, 7179986</td>
</tr>
<tr>
<td>Floating point</td>
<td>1.73, -7811.81, .00016</td>
</tr>
<tr>
<td>Exponential</td>
<td>8.4E10, 987E-2, -2.4002E-5</td>
</tr>
</tbody>
</table>

You can specify a hexadecimal constant by preceding the number with a dollar sign. Examples are: $E060 and -$CC00. A hexadecimal constant represented this way is stored internally the same as any other integer constant. The maximum absolute value for a hexadecimal constant is $FFFF.

Another way to specify a hexadecimal constant is to precede the number with a number sign (#). Examples are: #E060 and -#CC00. This representation causes the value to be stored in binary form in the first two bytes of a variable. The number is stored high byte followed by low byte. This representation is useful for specifying ASCII codes. For example, the hexadecimal ASCII code for a carriage return is #OD00.

String Constants

A string constant, or character string, is specified by enclosing a string of characters in single quotation marks. For example:

'John Smith'
'CITIES AND STATES'

To include single quotation marks in a character string, use two together. For example:

'Mike"s Place'
By specifying the hexadecimal ASCII code for a character, you can include in a character string any character that can be generated. Characters enclosed in backslashes are interpreted as hexadecimal constants. For example:

\'Hurry\21\' is interpreted as 'Hurry!'

TYPE \'\7F\' displays a rubout when executed.

\'This is a backslash: \5C\'

The last example demonstrates the only way a backslash can be included in a string.

Note: Never use \0 as part of a character string in an input or output list (see the ACCEPT, TYPE, READ, and WRITE statements). It will cause an error.

Logical Constants

The logical constants are .TRUE. and .FALSE.. They can be assigned to any variable. Numerically, .TRUE. has the value 1 and .FALSE. has the value 0. In logical tests, any nonzero number is interpreted as .TRUE..

2.2.2. Variables

A variable is an quantity that can have different values at different times. Values may be assigned to variables with the assignment operator (=). For example:

ALPHA = 17.5
ALPHA = ALPHA + 3
ICHR$ = 'x y'
IVAL = 10

In PTDOS FORTRAN variable names may have one to six alphanumeric characters and the first character must be alphabetic. There are two types of variable names: integer and real. By default, variable names starting with I, J, K, L, M, or N are integer names and all others are real. The type of any variable name can be changed using one of the type statements described later.

If a real value is assigned to an integer variable name, FORTRAN truncates the fractional part of the value. If an integer value is assigned to a real variable name, FORTRAN converts the value to a real value. For example:

IX = 17.9 assigns 17 to IX
AX = 10 assigns 10. to AX

You can assign character strings to real or integer variable names using an assignment, READ, or DATA statement. Any variable can contain up to six characters, but you can only assign up to four characters to an integer variable using the assignment statement. For example:

NAME = 'Wats'

Only four characters can be assigned to an integer variable using an assignment statement.

ANAME = 'Watson'

Six characters can be assigned to a real variable.

DATA NAME /'Watson'/

Six characters can be assigned to an integer variable using the DATA statement.

When a string with fewer than six characters is assigned to a variable, the characters are stored left-justified and the remainder of the word is filled with NULs (binary zeros).

Words that identify FORTRAN functions should be avoided as variable names. In addition, the reserved word COPY may never be used as a variable name.
2.2.3. Expressions

An expression is any valid combination of constants, variables, functions, and operators. An expression is evaluated by performing operations on quantities preceding and/or following an operator. These quantities are called operands. Examples of some expressions with their operators and operands are:

<table>
<thead>
<tr>
<th>Operand</th>
<th>Operator</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>IANS</td>
<td>*</td>
<td>SUMXY</td>
</tr>
<tr>
<td>A</td>
<td>.GT.</td>
<td>ALPHA</td>
</tr>
<tr>
<td></td>
<td>.NOT.</td>
<td>ANSWER</td>
</tr>
<tr>
<td>INCOME</td>
<td>-</td>
<td>EXPENS</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>SAL</td>
</tr>
</tbody>
</table>

The .NOT. and unary minus (−) operators precede an operand. All other operators join two operands.

There are three types of operators: arithmetic, relational, and logical. A PTDOS FORTRAN expression may include any of the following operators:

**Arithmetic Operators**

- ** or ^ Exponentiation
- * Multiplication
- / Division
- + Addition or Unary Plus
- − Subtraction or Unary Minus

**Relational Operators**

- .LT. Less than
- .LE. Less than or equal to
- .EQ. Equal to
- .NE. Not equal to
- .GE. Greater than or equal to
- .GT. Greater than

**Logical Operators**

- .NOT. Logical negation
- .AND. Logical conjunction
- .OR. Logical disjunction
- .XOR. Logical exclusive disjunction

(Logical operators are described in unit 2.2.7, below.)

2.2.4. Order of Evaluation in Expressions

When FORTRAN evaluates an expression, it scans from left to right. It performs higher-order operations first, and the results become operands for lower-order operations. For example:

```
x .GT. VAL1 - VAL2
```

The value of VAL1 − VAL2 becomes an operand for the .GT. operator.
Thus, operators act on expressions.

The hierarchy of operator evaluations is as follows:

```
Highest  ** or ^ (up arrow)
        * and /
        + and - (including unary + and -)
        .LT., .LE., .EQ., .NE., .GE., .GT.
        .NOT.
        .AND.
        .OR. and .XOR.

Lowest
```

Note: System and user functions are evaluated before any of the operators.

The examples that follow demonstrate the order of evaluation in FORTRAN expressions:

```
ALPHA * BETA / (-GAMMA)  First
                         Second  (When operators have the same order,
                         Third they are evaluated left to right.)
EXP(ARG)**2              First  (Functions are evaluated first.)
                         Second
```

ANSI Standard 3.9 does not prescribe the evaluation order for multiple successive exponentiation. PTDOS FORTRAN performs such exponentiation from left to right. Some FORTRANs evaluate it differently. For example:

```
BASE ** PWR1 ** PWR2     First
                         Second
-3**J                    First
                         Second
-A ** B ** C             First
                         Second
                         Third
```

You can use parentheses to change the order of evaluation. Expressions in parentheses are evaluated before any other part of an expressions. For example:

```
ANSWER .EQ. A .AND. .NOT. B
               First
               Second
               Third

ANSWER .EQ. (A .AND. .NOT. B)
               First
               Second
               Third
```

2.2.5. Integer Expressions

An integer expression has an integer value. It results only when both operands acted on by an operator are integers or when a real expression is assigned to an integer variable:
integer operator integer
integer variable = real

If an operation in an integer expression results in a noninteger value, the result is truncated to an integer. For example:

\[ K = 3 / 2 \] assigns 1, not 1.5 to K.
\[ A = 3 / 2 \] assigns 1 to A. Even though A is real, 3/2 is an integer expression and its value is 1.

**CAUTION:**

Constants without decimals are integer constants and are not converted to real values in integer expressions. Be sure to include decimal points in numerical constants when you want real results, even if there are no places to the right of the decimal point. For example, \( \text{SIN}(1/2) = 0 \) but \( \text{SIN}(1./2.) = 0.479 \ldots \)

The value of an integer expression can have up to eight digits. If an integer operation results in more than eight digits, a runtime error will occur.

**2.2.6. Real Expressions**

A *real expression* has a real value. When any operand in an expression is real, the expression has a real value. Also, any expression assigned to a real variable is converted to a real expression. The following operations result in real expressions:

<table>
<thead>
<tr>
<th>real</th>
<th>operator</th>
<th>real</th>
</tr>
</thead>
<tbody>
<tr>
<td>real</td>
<td>operator</td>
<td>integer</td>
</tr>
<tr>
<td>integer</td>
<td>operator</td>
<td>real</td>
</tr>
<tr>
<td>real variable</td>
<td>=</td>
<td>integer</td>
</tr>
<tr>
<td>real variable</td>
<td>=</td>
<td>real</td>
</tr>
</tbody>
</table>

When one of the operands is an integer, it is converted to a real value before the operation occurs. For example:

\[ \frac{7}{5} \times 3.7 \]

The integer value of \( \frac{7}{5} \), 1, is converted to a real value, 1.0.

The result is \( 3.7 \times (1.0 \times 3.7) \).

Real values range between \(-99999999E+127\) and \(+999999999+127\). The smallest absolute value is 0.1E-127. If the value of a real expression is outside this range, a runtime error results.

**2.2.7. Logical Expressions**

A *logical expression* has a logical value of .TRUE. or .FALSE. corresponding to a numerical value of 1 or 0. Conversely, a logical operation interprets any zero value as .FALSE. and any nonzero value as .TRUE.

For example:

\[ 10 \ .GT. \ 100 \] has a value of .FALSE. (0).
\[ .NOT. \ 35.02 \] has a value of .FALSE. (0) because 35.02 is a nonzero value and is therefore .TRUE..
\[ X \ .NE. \ 0 \] has a value of .TRUE. unless X = 0.
\[ X \ .EQ. \ .TRUE. \] has a value of .TRUE. unless X = 0.

Notice that the last two expressions are equivalent.
Logical expressions can include any operators. Relational operators joining two operands result in numerical values of 1 or 0 and logical values of .TRUE. or .FALSE.. For example:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Numerical Value</th>
<th>Logical Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 .GT. 2</td>
<td>0</td>
<td>.FALSE.</td>
</tr>
<tr>
<td>2*3 .NE. 5</td>
<td>1</td>
<td>.TRUE.</td>
</tr>
</tbody>
</table>

The logical operator .NOT. preceding an expression results in the logical reverse of that expression's value. For example:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Numerical Value</th>
<th>Logical Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>.NOT. 0.0001</td>
<td>0</td>
<td>.FALSE.</td>
</tr>
<tr>
<td>.NOT. 0</td>
<td>1</td>
<td>.TRUE.</td>
</tr>
<tr>
<td>.NOT. 1.7/99</td>
<td>0</td>
<td>.FALSE.</td>
</tr>
<tr>
<td>.NOT. .TRUE.</td>
<td>0</td>
<td>.FALSE.</td>
</tr>
</tbody>
</table>

The logical operators .AND., .OR., and .XOR. join two expressions with the following results:

- `expression1 .AND. expression2`: True only if both are true.
- `expression1 .OR. expression2`: False only if both are false (true if one or both are true).
- `expression1 .XOR. expression2`: True only if one is true and one is false.

Examples are:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Numerical Value</th>
<th>Logical Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 .AND. 1</td>
<td>0</td>
<td>.FALSE.</td>
</tr>
<tr>
<td>15 .AND. 22</td>
<td>1</td>
<td>.TRUE.</td>
</tr>
<tr>
<td>0 .OR. .0003</td>
<td>1</td>
<td>.TRUE.</td>
</tr>
<tr>
<td>0 .XOR. .0003</td>
<td>1</td>
<td>.TRUE.</td>
</tr>
<tr>
<td>1 .OR. 3</td>
<td>1</td>
<td>.TRUE.</td>
</tr>
<tr>
<td>1 .XOR. 3</td>
<td>0</td>
<td>.FALSE.</td>
</tr>
</tbody>
</table>

Just as numerical expressions can be included in logical expressions, the numerical results of logical expressions can be used as part of numerical expressions. For example:

\[
\text{ANSWER} = 30.0 - 30 \times (AX .AND. BX)
\]  
- The value will be 30. or 0.

\[
\text{SUM} = \text{SUM} + (\text{VAL1 .XOR. VAL2})
\]  
- SUM is incremented by 1 or 0.

### 2.2.8. Internal Formats and Ranges of Values

Values are stored internally as six-byte BCD words. Integer and real numbers are stored with an eight-digit or four-byte mantissa, a one-byte exponent, and a sign byte. The format is shown below:

```
<table>
<thead>
<tr>
<th>byte 0</th>
<th>byte 1</th>
<th>byte 2</th>
<th>byte 3</th>
<th>byte 4</th>
<th>byte 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>nn</td>
<td>nn</td>
<td>nn</td>
<td>nn</td>
<td>0s</td>
<td>ee</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BCD number</td>
<td>sign</td>
<td>exponent</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

A value of one for the sign byte indicates a negative number. The number 0 is stored as an exponent of zero and the rest of the word is ignored.

The exponent for real numbers ranges from -127 to +127. For integers, the exponent is between 0 and 8, and all fractional digits are zeros. That is, integer values are maintained as integers internally.
The ranges of numbers are:

**Real:** \(-.99999999E+127\) to \(+.99999999E+127\)
The smallest absolute value is \(0.1E-127\).

**Integer:** \(-9999999\) to \(+9999999\)
The smallest absolute value is 0.

Strings can be stored in real or integer words. A string value stored in a variable can have as many as six characters. Integer variables can contain up to six characters but only four can be assigned using the assignment operator (\(=\)).
SECTION  3

PREPARING THE SOURCE PROGRAM FILE

3.0. INTRODUCTION

You can create the source code for a PTDOS FORTRAN program in either of the PTDOS editors: EDIT or EDT3. These editors produce normal text files having lines terminated by carriage returns, or ALS-8 files.

If you have FORTRAN source files (either normal format or ALS-8) written on another system, you can read the files from disk or tape, edit them with EDIT or EDT3, and store them on diskettes.

The PTDOS User’s Guide contains complete documentation for EDIT and EDT3. This section gives a quick introduction for using EDIT to create a program file and does not describe EDT3.

3.1. CREATING A PROGRAM

There are two steps to creating a source program: creating the file and entering text into the file. You can create the file separately or let EDIT do it for you. The advantage of creating the file yourself is that you can assign to the file a type and block size other than the default values. For example:

```
*CREATE MYPROG, F, 200
   |           | block size
   |           | type
file name
```

If you edit the file without first creating it, the file will be created with a type of "." and a block size of 4C0.

To enter text into your file, call the editor as follows: *EDIT file name <CR>

In EDIT, enter the statements of your program, following each statement line except the last with a carriage return. EDIT has moving-cursor editing. Up-arrow, down-arrow, left-arrow, and right-arrow are all available for moving the cursor to any position in the text.

Some of the EDIT commands are listed below. EDIT has many more capabilities than indicated here. It is very worthwhile to study EDIT documentation in section 7 of the PTDOS User’s Guide.

- Control T   Insert characters
- Control H   Delete characters
- Control B   Insert line before the current line
- Control P   Delete the current line
- Control R   Scroll up 16 lines
- Control C   Scroll down 16 lines
- Control F   Exit and update file
- ESCAPE      Abort. No file update

In the example that follows, a user creates a file using the CREATE command and enters statements into the file using EDIT:
User:  *CREATE FACTRL, F, 100 <CR>

*EDIT FACTRL <CR>

EDIT:  Editor

******
Last load addr: 0FF0
Load count: 0000
End of file at: 0FF1
Bytes free: 740E
C/R to continue

User:  <CR>

EDIT:  (Displays a screen full of nulls.)

User:  C This program computes <CR>
C factorials. <CR>
    IFACT = 1 <CR>
    ACCEPT 'ENTER A NUMBER ', N <CR>
    DO 10 I=1,N <CR>
    10 IFACT = IFACT * I <CR>
    TYPE N, ' FACTORIAL = ', IFACT <CR>
    END <CR>

EDIT:  *** Editor exit
        type a carriage
        0FF1 File start address
        return after the
        10A8 File end address
        last line.)
        00B8 File count
        OK to write to "FACTRL"?

User:  Y

The program file FACTRL is now on disk and can be compiled, assembled, and executed as described in section 4.

3.2. FORMAT OF A PROGRAM FILE

In PTDOS FORTRAN the source code of all nonsystem subprograms called directly or indirectly by the main program must be included on the program file with the main program. System subroutines and functions are automatically available at execution time.

Each program, subroutine, and function on the program file ends with an END statement. No blank lines are allowed after the END statement. Other than that, blank lines are ignored. The format of a program file is demonstrated below:
You may precede any program or subprogram with an OPTIONS declaration to specify limits for parameters, ask for more information from the compiler, etc.

The form of the OPTIONS declaration is

\texttt{\$OPTIONS option list}

where items in the option list are separated by commas. No blanks are allowed in the option list.

The table below lists and describes all options that can appear in the option list. They may appear in any order.
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
<th>Storage Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>Tells the compiler to list error numbers for compilation errors instead of explicit error messages.</td>
<td>explicit messages</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>Tells FORTRAN to print the line numbers of statements causing runtime errors.</td>
<td>no line numbers</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>Tell the compiler to compile for errors only. No assembly code is generated.</td>
<td>generates assembly code</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Tells the compiler to copy each source statement to the assembly file as a comment preceding the code it generated.</td>
<td>no source statements</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>Tells the compiler to list as comments a reference table equating user symbols, constants, and labels to internally-generated ones.</td>
<td>no reference table</td>
<td></td>
</tr>
<tr>
<td>Q</td>
<td>Allows error trapping. This option is required in routines that have ERRSET statements.</td>
<td>no error trapping allowed</td>
<td></td>
</tr>
<tr>
<td>S=n</td>
<td>Specifies the number of symbols and constants allowed.</td>
<td>50</td>
<td>n x 12 bytes</td>
</tr>
<tr>
<td>L=n</td>
<td>Indicates the number of labels allowed.</td>
<td>50</td>
<td>n x 6 bytes</td>
</tr>
<tr>
<td>T=n</td>
<td>Indicates the maximum number of temporary variables available during expression evaluation. The value of n cannot exceed 255.</td>
<td>15</td>
<td>n bytes</td>
</tr>
<tr>
<td>D=n</td>
<td>Specifies the maximum level of nesting for DO loops. The value of n cannot exceed 255.</td>
<td>5</td>
<td>n x 4 bytes</td>
</tr>
<tr>
<td>A=n</td>
<td>Indicates the maximum number of arrays that can be defined. The value of n cannot exceed 255.</td>
<td>15</td>
<td>n x 16 bytes</td>
</tr>
<tr>
<td>O=n</td>
<td>Specifies the maximum number of operators stacked during a prefix translation of an input expression. Subscripting of functions or arrays requires a double entry. The value of n cannot exceed 255.</td>
<td>40</td>
<td>n x 2 bytes</td>
</tr>
<tr>
<td>P=n</td>
<td>Specifies the maximum number of variables and constants stacked during expression evaluation. The value of n cannot exceed 255.</td>
<td>40</td>
<td>n x 2 bytes</td>
</tr>
</tbody>
</table>

For example:

```fortran
$OPTIONS X,T=20
.
.
CALL (VAL)
.
.
END
$OPTIONS S=60
SUBROUTINE MYSUB(X)
.
.
END
```

Line numbers will be listed for runtime errors and up to 20 temporary variables will be available.

Sixty symbols and constants will be allowed for the subroutine.
4.0. INTRODUCTION

A FORTRAN source program describes a process in an algebraic-like language that people can understand. Before the computer can interpret the program, the language must be converted twice: first to PTDOS assembly language (a lower-level language than FORTRAN) and then to the binary form that the computer can understand. This is represented graphically below:

```
source program

    FORTRAN compiler

          assembly-language program

    PTDOS assembler

binary object code
```

The files corresponding to the three forms of a program are:

- source program file
- assembly-language file
- binary object file

Once the binary object file has been produced, the program can be executed by simply typing the object file name.

The steps involved in compiling, assembling, and executing a FORTRAN program are described in detail below.

4.1. COMPILATION AND ASSEMBLY

The PTDOS FORTRAN compiler translates FORTRAN source code into PTDOS assembly-language code and reports errors in syntax and semantics. The PTDOS assembler translates the assembly code into a form that can be understood by the computer and inserts an instruction to start execution when the object file name is typed.

The general form of the FORTRAN command line for compiling and assembling a source program is:
FORTRAN (option list), source<A>, list, assembly, object

Any or all of FORTRAN ALS-8 Assembly-Object
the following: source file language file name
S=L, S=N, B=n, C=n file name file name

Destination for listing
and error messages

For example:

*FORTRAN FACTRL,,FOBJECT<CR> Compiles and assembles
FACTRL and writes the
executable program (object
code) on FOBJECT.

You can interrupt a compilation by pressing the MODE SELECT key.

The parameters must appear in the order shown in the general form. Items in the option list can be in any order. The parameters are explained in more detail below:

4.1.1. Parameters in the FORTRAN Command Line

S=L
The S=L parameter specifies a long compilation. The long-compile option selects only those routines needed by the program being compiled, minimizing the use of memory space.

If you do not specify S=L in the FORTRAN command line, you choose the quick-compile option. Using the quick-compile option sacrifices memory space for speed. The quick-compile option greatly reduces assembly time by loading the complete runtime package (all system subroutines and functions).

S=N
The S=N parameter specifies a search for compilation errors only. No assembly code is created. The S=N parameter in the FORTRAN command line is identical to the N option in an OPTIONS declaration.

B=n
The B=n parameter sets the number of characters allowed in a statement to n (hexadecimal). The default maximum statement size is 530 decimal or 212 hexadecimal. You can use the B=n option to override the default when an INPUT BUFFER OVERFLOW compilation message is generated. n is a hexadecimal number unless it ends with :D. For example, the following commands both increase the number of characters allowed in a statement by 10:

*FORTRAN B = 540:D, FCODE,,ACODE <CR>
*FORTRAN B = 21C, FCODE,,ACODE <CR>

C=n
The C=n option sets the maximum number of COMMON blocks allowed to n (hexadecimal). The default number of COMMON blocks allowed is 15.

source<A>
Only one parameter is required to compile a FORTRAN program: the name of the source program file. For example:

*FORTRAN PROGY <CR> Compiles the source code on file
PROGY.
The compiler uses default values for parameters that are not specified. If you do not specify a list file, the listing and error messages will be displayed on the terminal. If you do not specify an assembly file, the compiler writes the assembly code on a file named $FORTASM. If you do not specify an object file no object code is produced.

The characters <A> appended to a source file name indicate that the file is in ALS-8 format. For example:

*FORTRAN S=L,OLDFL<A>,,,OBJ <CR> Compiles the ALS-8 source file OLDFL using the long-compilation option.

list

The list file name is optional. It specifies the destination of the FORTRAN listing and compilation error messages. If it is omitted, the listing and errors are displayed at the terminal.

assembly

This parameter specifies the destination of the assembly language code. If no file name is specified, the compiler writes the assembly code on a file named $FORTASM.

Note:

To speed compilation time the compiler does not generate an end of file for the assembly file. This could cause some large assembly files to be left on disk. You should purge them periodically.

object

The last parameter specifies the name of the binary object file. This is the executable image file of the program. If no file name is specified, no object file is produced.

4.1.2. Compilation Errors

During compilation the source program is scanned for errors in syntax or semantics. All errors are reported on the list file as error messages or, if the G option was specified in the OPTIONS declaration, as error numbers.

Errors that prevent compilation are marked as fatal errors. A nonfatal error usually indicates an error in logic and generates unpredictable assembly code. A program that has an error should not be executed. The UNREFERENCED STATEMENT LABEL message is only a warning and does not indicate that the program should not be executed.

Appendix 5 lists all compilation error messages and their error numbers.

4.2. EXECUTION

You can execute the binary object file generated by the FORTRAN command by simply typing the file name. PTDOS loads the program, beginning at location 100 hexadecimal (100H), and begins execution. For example, after compilation and assembly to produce the object file FOBJECT, the program whose source code is shown in unit 3.1 can be executed as follows:

User: *FOBJECT <CR>      Typed by the user
Program: ENTER A NUMBER 3 <CR>
         3 FACTORIAL = 6
         STOP END IN - MAIN
         *

A program that has been interrupted or just finished execution can be restarted at location 100H (first, the system closes all open files). For example:
Typed by the user

User:     *FOBJECT <CR>  
Program:  ENTER A NUMBER 2 <CR>  
          2 FACTORIAL = 2 
          STOP END IN - MAIN 
User:     *EXEC 100H <CR> 
Program:  ENTER A NUMBER ...

If a program is interrupted by a runtime error caused by a PTDOS operation, it cannot be restarted at location 100H. The runtime errors caused by PTDOS operations are FILE OF, I/O ERR, and OPEN ERR.

Note:

Re-executing a program from location 100H does not reinitialize any variables initialized with the DATA statement.

Because of the nature of console input (file #0), when a program is executed, any subsequent PTDOS commands that follow the program name being executed are ignored. --> BEWARE OF THIS <--

4.2.1. Runtime Errors

There may be errors in your program other than the syntax or semantics errors that are detected during compilation. Errors detected during execution of a program are called runtime errors. Appendix 6 lists and explains all runtime error messages. Runtime error messages are displayed during execution of a program. If the X option was specified in an OPTIONS declaration, the number of the statement producing the error is also displayed.

4.2.2. Large Programs

You might write a program that compiles and assembles correctly, but generates memory protect errors when you attempt to execute it. If this happens, it is possible that the binary object file of your program is so large that there is not enough room for it below PTDOS.

PTDOS is 12K long and usually resides at 9000 to BFFF. The video display module takes up 1K of storage beyond that. (See page 3-4 of the PTDOS User’s Guide for a storage diagram.) For a system with 64K of storage, there is extra space from D001 to FFFE.

You can use the extra space above PTDOS by using a second ORG instruction to split your large program into parts that will fit below and above PTDOS.

One way to insert the ORG is to estimate what point in the FORTRAN source corresponds to the code that exceeds the lower memory available. At that point insert:

    GO TO 10
    CONTINUE
    * ORG 0D001H
    10    CONTINUE

If the program still generates a memory protect error, place the ORG earlier in your program and try again.

Alternatively, you can view the assembly-language version of the program in an editor, find out where the addresses approach 9000 (the beginning of PTDOS), and place a second ORG instruction there.
SECTION 5

PTDOS FORTRAN STATEMENTS

5.0. INTRODUCTION
A computer program consists of statements that ultimately tell the computer what actions to perform. This section describes in detail all of the PTDOS FORTRAN statements, categorizing them according to the functions they perform. The types of PTDOS FORTRAN statements are:

- Replacement
- Program termination
- Control
- Error trapping
- Input and output
- Declaration
- Subroutine and function
- Copying source files

5.1. ASSIGNMENT STATEMENTS
An assignment statement assigns the value of an expression to a variable. The form most often used is:

\[
\text{variable} = \text{expression}
\]

where \text{variable} is a simple or subscripted variable.

For example, the following are valid replacement statements:

\[
\text{PROD} = \text{A} \ast \text{B} \\
\text{VECTOR}(2,3) = 17.3
\]

The equals sign is an assignment operator and does not denote equivalence. For example:

\[
\text{SUM} = \text{SUM} + 1 \quad \text{Assigns to} \ \text{SUM} \ \text{its previous value plus one.}
\]

The expression in an assignment statement may be numerical, string, or logical.
Assignment Statement

General form:

\[
\text{variable} = \text{expression} \quad \text{Assigns the value of the expression to the variable.}
\]

\[
\text{simple or subscripted variable}
\]

Examples:

\[
X(I) = Y**2 - \text{GAMMA} \\
IV = 'MOD' \\
\text{ANSWER} = \text{OLD .EQ. NEW}
\]

The assignment statement evaluates an expression and assigns its value to a variable. The variable can be real, integer, or logical, and the value of the expression can be numerical, logical, or string.

Program Example:

\begin{verbatim}
X = 20.3    -Assigns a real value to a real variable.
Y = 'CHARS' -Assigns a string value to a real variable.
Z = X .EQ. Y -Assigns 0 (logical value of .FALSE.) to a real variable.
TYPE X,Z
END
\end{verbatim}

Logical values assigned to real or integer variables result in numerical values of 0 or 1.

A character string with as many as six characters can be assigned to a real variable, but only four characters can be assigned to an integer variable. (You can store up to six characters in integer variables using input or DATA statements, but not using the assignment statement.)

5.2. PROGRAM TERMINATION

The END, STOP, and PAUSE statements interrupt or terminate program execution. The END statement ends execution of a routine and must be the last statement of every main routine, subroutine, function, or BLOCK DATA subprogram. The STOP statement stops execution wherever it is placed in the program and optionally displays a message. The PAUSE statement can be used to delay execution of a program until the user presses a key on the terminal.

END Statement

General form:

\[
\text{END} \quad \text{Terminates execution. Required as the last statement of every routine.}
\]

Examples:

\begin{verbatim}
END
\end{verbatim}

The END statement is required at the end of every main routine, subroutine, and function. When the END statement is encountered, execution terminates and the following message is displayed:
END IN - name

Name of the routine in which the END statement was encountered.

Program Example:

$OPTIONS X
CALL MYSUB
END ← Note that program execution will end here.
$OPTIONS X
SUBROUTINE MYSUB
TYPE 'THIS IS MYSUB'
RETURN
END ← This statement is never encountered because of the
RETURN preceding it. It is required, however.

STOP Statement

General forms:

STOP [character string] Terminates program execution
STOP [n] and displays the character
     | string or integer, if
     1 to 5 digit integer present.

Examples:

STOP
STOP 'An error occurred after DO loop.'
STOP 100

The STOP statement causes termination of execution wherever it is encountered in the program. If a character string or an integer is included in the STOP statement, it is displayed when the STOP statement is executed.

Program Example:

ACCEPT 'ENTER A NUMBER BETWEEN 1 AND 10 ',N
IF (N .LT. 1 .OR. N .GT. 10) GO TO 50
.
.
50 STOP 'YOUR NUMBER WAS NOT BETWEEN 1 AND 10'
END
PAUSE Statement

General forms:

PAUSE {character string}          Interrupts execution and
PAUSE {n}                         displays the word PAUSE and
       1 to 5 digit integer          the integer or character
                                       string, if present.

Examples:

    PAUSE
    PAUSE 'DATA OUT OF SEQUENCE'
    PAUSE 250

The PAUSE statement interrupts execution and displays the word PAUSE and an optional message or integer. To continue execution, the user must press a key on the keyboard.

Program Example:

    PAUSE 'PRESS ANY KEY TO CONTINUE'
    TYPE 'SEE? EXECUTION CONTINUED.'
    END

5.3. CONTROL STATEMENTS

The statements described in this unit let you control the order in which statements are executed. With the GO TO, computed GO TO, and assigned GO TO statements, you can branch to a different part of the program. The IF statements provide logical decision making. Looping is available through DO and CONTINUE statements, which let you repeatedly execute a set of statements.

GO TO Statement

General form:

    GO TO n                      Unconditionally transfers control
       1                         to statement n.
       statement
       label

Example:

    GO TO 150

The GO TO statement causes the indicated statement to be the next statement executed. You can use the GO TO statement to branch to statements above or below it in the program. The transfer can be to any labelled statement, including a FORMAT statement, which acts like a CONTINUE statement.
**Program Example:**

```fortran
$OPTIONS X
10 ACCEPT 'WHAT IS THE TOTAL SALE? ', SALE
   IF (SALE .EQ. 0) STOP
   TAX = SALE*.065
   TYPE 'THE TAX IS ', TAX
   GO TO 10
END
```

**Computed GO TO Statement**

General form:

```
GO TO (n1,n2,...), index
    statement
    labels
```

Executes statement n1 next if index = 1, executes n2 next if index = 2, etc.

Examples:

```
GO TO (100, 70, 120), IVAL
GO TO (10,10,10,40,100), N
```

The computed GO TO statement lets you branch to one of several statements depending on the value of the index. The value of the index must be at least one and must not be greater than the number of statement labels in parentheses or a runtime error will occur.

**Program Example:**

```fortran
$OPTIONS X
ACCEPT 'ENTER MILEAGE: ', MILES
ACCEPT 'BUSINESS = 1, OTHER = 2 -- ?', ITYPE
GO TO (10,20), ITYPE
10  DEDUCT = .17 * MILES
    GO TO 30
20  DEDUCT = .07 * MILES
30   TYPE 'DEDUCTION = ', DEDUCT
    END
```

**Assigned GO TO Statement**

General form:

```
GO TO v, (n1,n2,...)
    integer
    statement
    variable
    labels
```

Executes statement v next, where v is equal to one of the labels in parentheses. The value of v is assigned by an ASSIGN statement.

Examples:

```
GO TO LABL, (10,20,30)
GO TO K, (100, 110)
```
The *assigned* GO TO statement lets you transfer control to a variable statement label. The value assigned to the variable must be one of the values listed in parentheses. The *assigned* GO TO statement is used with the ASSIGN statement, described below.

**ASSIGN Statement**

General form:

```
ASSIGN n TO v  Assigns a statement label to the
   /  
label integer statement.
|    |
|variable
```

Example:

```
ASSIGN 10 TO LABL
```

The ASSIGN statement assigns a statement label to be used in an assigned GO TO statement. The variable in the ASSIGN statement is the same as the variable used in the *assigned* GO TO statement. For example:

```
ASSIGN 20 TO LABL
.
.
IF (ANS .EQ. 0) ASSIGN 10 TO LABL
.
.
GO TO LABL,(10,20)
.
.
```

**Arithmetic IF Statement**

General form:

```
IF (exp) n1, n2, n3  Executes statement n1 next if
   |   |   | the value of exp is negative,
integer, statement executes n2 if the value of exp
real, or labels is 0, executes n3 if the value
logical expression of exp is positive.
```

Examples:

```
IF (VALUE) 100, 10, 250
IF (INT/AEXP) 20, 30, 40
IF (ACT .EQ. EST) 10, 10, 320
```

The arithmetic IF statement evaluates an expression that may be any combination of integer, real, and logical expressions. It then transfers control to one of three statements based on the expression's value compared to zero.
Program Example:

```
ADDNL = 0
ACCEPT 'ENTER MILEAGE: ', MILES
IF (MILES - 15000) 10, 10, 20
20 ADDNL = (MILES - 15000)*10
      MILES = 15000
10 FRST15 = MILES *.17
      DEDUCT = FRST15 + ADDNL
      TYPE 'MILEAGE DEDUCTION = ', DEDUCT
      END
```

Logical IF Statement

<table>
<thead>
<tr>
<th>General form:</th>
<th>Executes the specified statement if the value of exp is nonzero.</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF (exp) statement</td>
<td>Any expression FORTRAN statement except DO, END, or another</td>
</tr>
<tr>
<td>any expression</td>
<td>logical IF</td>
</tr>
</tbody>
</table>

Examples:

```
IF (IVAL) GO TO 20
IF (ANS - 300) STOP
IF (X .GE. Y) RETURN
```

The logical IF statement evaluates an expression and, based on its value, does or does not execute a specified statement. If the value is .TRUE. (nonzero) the statement within the IF statement is executed and execution continues with the next statement in sequence. If the value of exp is .FALSE. (zero) the statement within the IF statement is not executed and execution continues with the next statement in sequence.

Program Example:

```
ACCEPT 'WHAT IS THE MONTH? ', MONTH
IF (MONTH .EQ. 1 .OR. MONTH .EQ. 10 .OR. MONTH .EQ. 11)
   TYPE '21 WORK DAYS'
IF (MONTH .EQ. 2) TYPE '19 WORK DAYS'
IF (MONTH .EQ. 3 .OR. MONTH .EQ. 8) TYPE '23 WORK DAYS'
IF (MONTH .EQ. 4 .OR. MONTH .EQ. 7 .OR. MONTH .EQ. 9
   .OR. MONTH .EQ. 12) TYPE '20 WORK DAYS'
IF (MONTH .EQ. 5 .OR. MONTH .EQ. 6) TYPE '22 WORK DAYS'
END
```
DO Statement

General form:

DO n index = v1, v2 [,increment]  
statement nonsubscripted constants or variables  
label subscripted variables

The statements between the DO statement and statement n are executed repeatedly as the value of index increases or decreases from v1 to v2 in steps of 1 (or increment, if present).

Examples:

DO 100 I = 1, 10
DO 50 IND = A,Z ,.01
DO 120 A = START, END, -1.5

The DO statement lets you execute a set of statements an indicated number of times. The index increases or decreases at each repetition of the loop. Its first value is v1, subsequent values are determined by adding one or the optional increment, and the final value is v2.

The loop is executed at least once regardless of the values of index, v1, v2, and increment. The values of all arguments in the DO statement can be positive or negative, integer or real. If the increment is negative, the value of index decreases from v1 to v2.

You can change the value of the index within the loop, thus changing the number of times the loop is executed. For example, the loop below is executed once:

DO 10 A = 1, 3.5, .5
   A = 4
10 CONTINUE

Notice that fractional step sizes are allowed.

After the index reaches or exceeds its final value, whether by increments or by assignment within the loop, the next statement in sequence is executed.

Program Example:

DO 10 A = 0.1, 1, .1
   Y = SIN(A)
10 TYPE 'SIN ',A,' = ',Y
END

Nested DO Loops

You can include DO loops within other DO loops provided you do not overlap parts of one loop with another. This practice is called nesting DO loops. For example:
DIMENSION ARAY(3,4)

DO 10 I = 1,3
DO 20 J = I,4
ACCEPT '?', ARAY(I, J)
20 CONTINUE
10 CONTINUE
END

DO 10 A = 1,3
DO 20 B = 1,3,.5
PROD = A * B
10 TYPE PROD
20 CONTINUE
END

Nested DO loops can end with the same statement, as shown below:

DO 10 A = 1,3
DO 10 B = 1,3,.5
PROD = A*B
10 TYPE PROD
END

When a DO loop is nested inside an outer DO loop, all iterations of the inner loop are performed for each iteration of the outer loop. For example, in the program shown above, for each value of A, B takes the values 1, 1.5, 2, 2.5, and 3. The depth (or number) of DO loops that can be nested is set by the D= option of the OPTIONS declaration.

**CONTINUE Statement**

General form:

CONTINUE       Takes no action. Used as a reference point for control statements.

Example:

CONTINUE

The CONTINUE statement is a nonexecutable statement that can be placed anywhere in the program. It is often used as the last statement of a DO loop.

**5.4. ERROR TRAPPING**

Normally, an error occurring during execution of a FORTRAN program causes a runtime error message to be displayed. Using the routines below, you can change this default condition and control what happens when an error occurs.
ERRSET Statement

General form:

\[
\text{ERRSET n, v} \quad \text{Transfers control to statement n if a runtime error occurs. Variable statement variable v contains the error code.}
\]

Example:

\[
\text{ERRSET 150, KODE}
\]

The ERRSET statement causes control to be transferred to statement n if a runtime error occurs. The ERRSET statement can be used only if the Q option was specified in the OPTIONS declaration for this routine. Runtime error messages are not printed for errors trapped with the ERRSET statement.

If a runtime error does occur, an error code will be stored in the v argument, indicating the nature of the error. The error codes are:

1. Integer overflow
2. Conversion error
3. Parameter count error
4. Computed GO TO index out of range
5. Overflow
6. Division by zero
7. Square root of a negative number
8. Logarithm of a negative number
9. Call stack PUSH error (too many recursive subroutine calls)
10. Call stack POP error
11. File operation error
12. Illegal logical unit number
13. Unit already open
14. Open error
15. Unit not open
16. Set unit (drive) error
17. Line length too long
18. Format error
19. Input/output error during read or write
20. Invalid character during input
21. Invalid input/output list
22. Assigned GO TO error

If more than one ERRSET statement appears in a routine, the latest one executed is in effect.

If a runtime error occurs, the effect of the ERRSET statement is cleared after control transfers to the specified label. You must then execute another ERRSET statement if you want to continue trapping errors. For example:
$OPTIONS Q
ERRSET 110, IERR
.

110 TYPE 'ERROR', IERR, ' OCCURED'
ERRSET 110, IERR
.

END

ERRCLR Statement

General form:

ERRCLR

Clears the effect of the ERRSET statement.

Example:

ERRCLR

The ERRCLR statement clears the effect of the ERRSET statement in effect.

DUMP Statement

General form:

DUMP /ident/ output list

Displays ident followed by items in the output list

10-character variable names, when a runtime error that
identifier character is not trapped occurs.
strings, array
names or elements,
and/or implied DO loops

Example:

DUMP /AFTER LOOP/ 'INDEX = ',I

The DUMP statement is used to display information when a runtime error that is not trapped by the ERRSET statement occurs. If more than one DUMP statement is present, the latest one executed is in effect.

5.5. INPUT AND OUTPUT

Input and output statements transfer data to and from a program. A program can receive input from the terminal or from a disk file, and can send output to the terminal or to a disk file.

Input and output can be symbolic (data is represented as ASCII characters) or binary.

The types of input and output are represented graphically below:
5.5.1. Free-Format Terminal Input and Output

*Free-format* input means that data values can be entered at any character position in a line, and one value is distinguished from the next by a separator. Blanks between values are ignored and blank lines are ignored. In free-format input, the comma and carriage return both act as separators. During free-format input, integer values must be entered as one-to eight-digit integers and real values can be entered in any of the following forms:

**Examples:**

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>1, 2000, 40</td>
</tr>
<tr>
<td>Floating point</td>
<td>2.4, 5791.76</td>
</tr>
<tr>
<td>Exponential</td>
<td>.2 E15, -7.512E-3</td>
</tr>
</tbody>
</table>

For example:

**Program:** ENTER THREE NUMBERS: 13.2 <CR>  
**User:** .26 E3, 300 <CR>

The user enters one of the three values on one line and two on the next.

During free-format output, numerical data is displayed according to its type. Ten character positions are allowed for each integer and 18 character positions are allowed for each real number. Real numbers are displayed in exponential format. The field specifications for free-format output for integers and real numbers are I10 and E18.8, respectively.

Free-format character strings are displayed exactly as they appear in the output statement.

During free-format output, values that extend past column 63 are placed on the next line.

The ACCEPT and TYPE statements perform free-format terminal input and output exclusively. The READ and WRITE statements perform free-format terminal input and output when they are used with the appropriate arguments.

In the READ and WRITE statements, a second argument of asterisk (*) specifies free format. The first argument is the unit number. Unit number 0 is reserved for terminal input and unit number 1 is reserved for terminal output. Thus:

READ(0,*) input list  is identical to  ACCEPT input list
WRITE(1,*) output list is identical to  TYPE output list
ACCEPT Statement

General form:

\[
\text{ACCEPT input list} \quad \text{Reads values from the terminal and assigns them to items on the input list.}
\]

May include variable names, array names, array elements, implied DO loops, character strings.

Examples:

\[
\begin{align*}
\text{ACCEPT ALPHA, BETA, GAMMA} \\
\text{ACCEPT 'X = ',X} \\
\text{ACCEPT ARAY(4,1), VAL} \\
\text{ACCEPT (VALS(I), I = 1,3)}
\end{align*}
\]

The ACCEPT statement reads one or more numerical values from the terminal and assigns them to items in the input list. The values entered at the terminal must be separated by commas or carriage returns.

When an ACCEPT statement is executed, there is a pause and the program waits for the user to enter values at the terminal. It does not continue with the rest of the program until a value has been entered for each item on the input list. When the ACCEPT statement waits for input, the user must enter as many values as there are items read by a single execution of the statement. The values can be entered on one or more lines. For example:

\[
\begin{align*}
\text{ACCEPT '?', X,Y,Z} & \quad \text{When the question mark appears, the user can enter values for X, Y, and Z one per line, all on one line, or mixed.} \\
\text{DO 10 I = 1,3} \\
10 \quad \text{ACCEPT VAL(I)} & \quad \text{This statement will only read one value from each line typed at the terminal.}
\end{align*}
\]

If the input list of an ACCEPT statement includes a character string, the string is displayed when the statement is executed. This feature is useful for prompting for input. For example:

\[
\text{ACCEPT 'ENTER THE COST: ', COST}
\]

If an array name is included in the input list, the ACCEPT statement reads values for every element of the array. The input list can also include array elements, such as AR(3,5), to read a value for one element only. To read a specified part of an array, you can include an implied DO loop, which works very much like the DO statement. An example is:

\[
\text{ACCEPT (A(I),I=1,10)} \quad \text{Reads 10 values and assigns them to the first ten elements of array A.}
\]

These features for reading array values are described more fully under the READ and WRITE statements in section 5.4.
Program Example:

$OPTIONS X
ACCEPT 'ENTER VALUES FOR X, Y, & Z: ',X,Y,Z
SUM = X+Y+Z
TYPE 'X + Y + Z = ', SUM
END

When this program is executed, the following interaction occurs (assuming the object file for this program is named TOBJ):

User: TOBJ <CR>
Program: ENTER VALUES FOR X, Y, & Z: 70.2 <CR>
User: 20.2, 11.4 <CR>
Program: X + Y + Z = 0.10190000E 003
STOP END IN - MAIN

Notice that the user entered only one value on the first line and the program waited until all three values were entered.

TYPE Statement

General form:

TYPE output list Displays items in the output list
       on the terminal.

May include variable
names, character strings,
array names, array elements,
and implied DO loops

Examples:

TYPE SALARY, TAX
TYPE 'THE TOTAL IS ', TOTAL
TYPE TABL
TYPE TABL(2,3),TABL(2,4)
TYPE (VECTOR(J1),J1=1,5)

The TYPE statement displays values on the terminal. The items in the output list are displayed sequentially.

Note:

You cannot include operators or function names in the TYPE statement.

Program Example:

DIMENSION ARAY(3)
TYPE 'THIS IS AN EXAMPLE OF OUTPUT BY THE TYPE & STATEMENT.'
ARAY(1) = 17.2
ARAY(2) = 1
ARAY(3) = 3.22
TYPE (ARAY(I),I=1,3)
END
5.5.2. Formatted Input and Output

You can specify the exact format of each input or output value using the FORMAT statement. The FORMAT statement describes the format of data to be read or written by the READ or WRITE statement. For example:

```
WRITE (1,130) INC
130 FORMAT (I5)
```

These statements write the value of INC as a five-digit integer.

During formatted input, FORTRAN reads a record (up to a carriage return) and then pads the record with blanks on the right until the record agrees with the total number of characters being read. This padding takes place for all field specifications and may generate a record as large as 250 characters. For example:

```
READ (0,3) I,J
3 FORMAT (I6,I3)
```

These statements read nine characters from each input record as shown below:

<table>
<thead>
<tr>
<th>Input record</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>987654321061 &lt;CR&gt;</td>
<td>I = 987654 J = 321</td>
</tr>
<tr>
<td>bb0b13 &lt;CR&gt;</td>
<td>I = 10 J = 0</td>
</tr>
<tr>
<td>bbb9 &lt;CR&gt;</td>
<td>I = 90 J = 0</td>
</tr>
<tr>
<td>&lt;CR&gt;</td>
<td>I = 0 J = 0</td>
</tr>
</tbody>
</table>

(where b represents a blank)

During formatted input or output the maximum number of characters in a record is 250.

**FORMAT Statement**

<table>
<thead>
<tr>
<th>General form:</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORMAT(field spec1, field spec2, ...) Describes the sizes, types, and positions of data values to be read or written.</td>
</tr>
<tr>
<td>field specifications</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Examples:</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 FORMAT(A6, I4)</td>
</tr>
<tr>
<td>100 FORMAT(3F10.1)</td>
</tr>
<tr>
<td>50 FORMAT(10X, E9.2, 2X, A10)</td>
</tr>
</tbody>
</table>

The FORMAT statement is a nonexecutable statement that defines how data values are to be read or written. A FORMAT statement must have a statement label so that it can be referred to by a READ or WRITE statement.

The FORMAT statement lets you define the size and type of data to be read or written. In addition, you can select the fields to be read or the columns on which to write values. The exact descriptions of data fields is accomplished with field specifications, documented below.

**Field Specifications**

PTDOS FORTRAN allows the following field specifications in FORMAT statements:

(In the field specifications, w represents field width and d represents the number of digits following the decimal point.)
<table>
<thead>
<tr>
<th>Field Spec.</th>
<th>Description</th>
<th>Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>Character string</td>
<td>Output only</td>
</tr>
<tr>
<td>Iw</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>Fw.d</td>
<td>Floating point</td>
<td></td>
</tr>
<tr>
<td>Ew.d</td>
<td>Exponential</td>
<td></td>
</tr>
<tr>
<td>Aw</td>
<td>Alphanumeric</td>
<td></td>
</tr>
<tr>
<td>Lw</td>
<td>Logical</td>
<td></td>
</tr>
<tr>
<td>wX</td>
<td>Skip spaces</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>Skip record</td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td>No carriage return</td>
<td>Output only</td>
</tr>
</tbody>
</table>

The I, F, E, A, and L field specifications describe the sizes and type of data to be read or written.

If a number cannot be written in the specified field width, the entire field is filled with asterisks to indicate the error condition.

Blanks read with the I, F, E, or L specifications are treated as if they were zeros.

**Field Specifications: string**

A character string in a FORMAT statement causes those characters to be output when the WRITE statement referring to the format is executed. This is a convenient replacement for the Hollerith specification of ANSI FORTRAN.

You can specify the hexadecimal code for any character in a character string by enclosing the hexadecimal code in backslashes. For example:

```plaintext
    WRITE(1,100)
100  FORMAT ('This is important\21\')
```

The hexadecimal code for an exclamation mark is \21, so the above statements display:

```
This is important!
```

Placing an ampersand in front of a character in a string causes the character to be treated as a control character. For example:

```plaintext
    WRITE(1,10)
10  FORMAT('THIS IS CONTROL P: &P')
```

These statements output the characters **THIS IS CONTROL P**: followed by control P.

To output an ampersand, use two ampersands together. For example:

```plaintext
100  FORMAT('THIS & & THAT')
```

**Field Specifications: lw**

This specification reads or writes integer digits. Only integer digits or a sign can be read with an I field. For example:

```plaintext
    READ(0,100)I,J,K
100  FORMAT(I5,I3,I1)
```

If these statements are used to read the record 0010001001, 100 is assigned to I, 10 to J, and 0 to K.

When reading data with the I specification, the values must be right justified in the field with leading blanks or zeros.
A = 30.2  
B = 1000  
C = 9  
WRITE(1,10) A,B,C  
10 FORMAT(I3,I5,I1)

These statements write: b30b10009 (where b represents a blank)

Field Specifications: Fw.d

The F field is used to read or write a floating-point number. The field width \( w \) includes the number of digits preceding and following the decimal point, one position for a negative sign if the number is negative, and one position for the decimal point. For example:

\[
F10.3 \quad \text{Indicates the numerical format} \\
\text{s0000.000} \\
\text{sign}
\]

During input the F field reads \( w \) characters. If there is no decimal point in the characters read, a decimal is inserted \( d \) digits from the right. A decimal in the input value overrides the field specification. For example:

\[
\text{READ}(0,150) X, Y \]
\[
150 \quad \text{FORMAT}(F7.2,F5.1)
\]

If the input line is \(-123456789.012\), these statements read \(-123456\) and assign \(-1234.56\) to \( X \), then read \(789.0\) and assign \(789.0\) to \( Y \).

During output the F field converts a value to the form indicated. For example:

\[
X = 23.7 \\
Y = 100  \\
\text{WRITE}(1,10) X,Y  \\
10 \quad \text{FORMAT}(F10.2,F5.1)
\]

These statements display \( bbbbbb23.70100.0 \), where \( b \) represents a blank.

\[\text{Note}\]

Zero is always printed as 0.0. If it appears in a different form during output (such as 0.00000), the value was not exactly zero and digits had been truncated.

Field Specifications: Ew.d

The E field reads or writes a real value in exponential form:

\[
\text{mantissa E exponent}
\]

The value is the mantissa multiplied by ten to the power exponent. For example, the value of \( 10 \ E \ 2 \) is 100.

The field width \( w \) includes the number of digits preceding and following the decimal point, one position for a negative sign if the number is negative, and five positions for \( E \) and its exponent. For a positive number \( w \) must be at least 7 greater than \( d \) and for a negative number \( w \) must be at least 8 greater than \( d \).

During input, the E field is equivalent to an F field; that is, it reads \( w \) characters and inserts a decimal point \( d \) digits from the right if there is no decimal in the characters read. A decimal point in the input value overrides the E format. Thus, it does not cause an error to read data in the F format using the E specification.
Example:

```
READ(0,100) A,B
100 FORMAT(E10.2,E8.1)
```

If the input line is 123456789012345678, these statements assign 0.12345678E008 to A and 0.12345678E007 to B. If the input line is 1.2345 002.3000, these statements assign the values 0.123456E001 to A and 0.23E001 to B.

During output the E field converts values to exponential format. For example:

```
X = 300
Y = -528.2
WRITE(1,10)X,Y
10 FORMAT(E10.2,E9.1)
```

These statements print the line

```
0.30E 003-0.5E 003
```

Notice that E9.1 was the smallest E field possible for printing a negative value with one significant digit.

**Field Specifications: A w**

The A field reads or writes w ASCII characters. Up to six characters may be read or written for each variable name. On input if w is less than 6, then w characters will be placed in each variable, left justified and padded with blanks on the right. On output, w characters are written starting with the leftmost character in the variable. For example:

```
DIMENSION A(8)
READ(1,10)A(I),I=1,8
10 FORMAT(8A1)
```

If the input line is ALPHABET, these statements read one character for each element of the array A. A(1) = A, A(2) = L, A(3) = P, etc. (Arrays are discussed in section 5.6.2)

The A field can be used to output character-string values stored in numerical variables, but not numerical values.

Any byte can be input or output using an A field, even if it is not a character.

**Field Specifications: L w**

The L field treats values read or written as logical values.

During input the L field scans w characters until a T or F is found. The T or F can be located anywhere in the field and all characters following the T or F are ignored. If the first nonblank character is not a T or F, an error occurs. A completely blank field results in a false value. For example:

```
READ(0,10) LOG1, LOG2
10 FORMAT(L5,L3)
```

If the input line is bbbTOTAL, two true values are returned and LOG1 and LOG2 are both set to I.

During output the L field prints the letter T if the value being output is nonzero (logically true) and prints the letter F if the value is zero (logically false). For example:

```
X = 20
Y = 0
WRITE(1,100)X,Y
100 FORMAT(L3,L1)
```

These statements print the line bbbTF, where b represents a blank.
**Field Specifications: wX**

The X field spaces over w columns with a maximum of 250 character positions allowed.

During input w characters of the input record are skipped. For example:

```
READ (0,120) I,J
120 FORMAT (2X,I2,I3)
```

If the input line is 1234567, these statements assign 34 to I and 567 to J.

During output the X field prints w blanks. For example:

```
M = 273
WRITE (1,10) M
10 FORMAT (10X,'THE GAIN = ',I3)
```

These statements print the line:

```
THE GAIN = 273
```

**Field Specifications: /**

The slash causes a READ or WRITE statement to skip to the next line before proceeding.

During input / causes reading to continue at the beginning of the next input record. For example:

```
READ (0,10) A,B,C
10 FORMAT (2F8.1/F9.2)
```

Suppose the input records to be read are:

```
023.5  1.30  2.100000
50.600  18.000000
```

The statements above read two values from the first record and one value from the second. They assign 23.5 to A, 1.3 to B, and 50.60 to C.

During output / generates a carriage return and output continues on a new line. For example:

```
I = 20
J = 30
K = 40
WRITE (1,100) I,J,K
100 FORMAT (3I13)
```

These statements print:

```
20 30 40
```

**Field Specifications: Z**

The Z field is used only during output. Its presence indicates that a carriage return is not to be written at the end of the record.
Example:

\[
\begin{align*}
A &= 23.782 \\
B &= 5543.3 \\
\text{WRITE}(1,120)A \\
\text{WRITE}(1,100)B \\
120 & \text{ FORMAT(E10.2,Z)} \\
100 & \text{ FORMAT(F10.1)}
\end{align*}
\]

These statements print the following line:

\[
0.23E\ 002 \quad 5543.3
\]

Repeating Field Specifications

You can repeat a field specifications in a FORMAT statement by preceding it with the number of repetitions. For example, \( 3I5 \) means read three values as five-digit integers. The specification \( 3(I5) \) also means read three values as five-digit integers, but this form sometimes has a different effect than \( 3I5 \). The following FORMAT statements are equivalent:

\[
\begin{align*}
100 & \text{ FORMAT(}2I5, \ 3F10.2) \\
100 & \text{ FORMAT(I5,I5,F10.2,F10.2,F10.2)}
\end{align*}
\]

When the field specification to be repeated is enclosed in parentheses, the count preceding the parentheses is called a group count. The following are examples of group counts:

\[
\begin{align*}
10 & \text{ FORMAT (}3(I5)) \\
100 & \text{ FORMAT (}2(F10.2,I3)) \\
150 & \text{ FORMAT (I3,(}I2,I5)) \quad \text{When the group count is omitted, it is assumed to be 1.}
\end{align*}
\]

The program below displays values using a group count of 2 in the FORMAT statement:

\[
\begin{align*}
N &= 33 \\
\text{DO 100 I = 1,2} \\
X(I) &= I \\
Y(I) &= 10 + I \\
100 & \text{ CONTINUE} \\
\text{WRITE (1,10) N, X(1),Y(1),X(2),Y(2)} \\
10 & \text{ FORMAT (I5, }2(I3,F5.1)) \\
\text{END}
\end{align*}
\]

This program displays:

```
bbbbbbbb1b11.0bb2b12.0
```

In this example, the format:

\[
10 \text{ FORMAT(I5, 2(I3,F5.1))}
\]

has the same effect as:

\[
10 \text{ FORMAT(I5, I3,F5.1,I3,F5.1)}
\]

The difference between these two FORMAT statements occurs when the input list of the READ or WRITE statement has more items than there are field specifications in the FORMAT statement.

In a FORMAT statement without group counts, control goes to the beginning of the FORMAT statement for reading (writing) of additional values. In a FORMAT statement with group counts, additional values are read according to to last complete group.
For example:

```
READ(2,10) N,(A(I),I=1,100)
10 FORMAT(I5/(4E12.2))
```

The FORMAT statement reads a value for N from the first five columns of the current record. The / indicates the end of the record and reading continues with the next record. Four values are read from that record and the end of the FORMAT statement is reached. The statements above then cause the next record to be read. In the FORMAT statement, control returns to the group count (1) preceding the group (4E12.2). The rest of the file is read four values from each line.

Thus, you can use group counts to repeat a group of field specifications for the rest of a read operation after the initial pass through the FORMAT is finished.

Group counts can be nested to a maximum depth of two. For example:

```
FORMAT(2(E14.2,3(I2,2I5)))  is legal
but
FORMAT(2(E14.2,3(I2,2(I5))))  is not legal
```

In the case of nested group counts, the last outer group is repeated for reading additional input. For example:

```
FORMAT(I5,2(I2,3I4))
  ↑
FORMAT(E12.1,3(I5,2(I1)))
  ↑
FORMAT(E12.1,I5,I1,I1,I5,I1,I1,I5,I1,I1)
```

The arrows show where repetition begins after the first pass through the FORMAT statement.

5.5.3. File Input and Output

The READ and WRITE statements described in this unit let you read data from and write data on files. Before reading or writing a file, you must open the file and associate a unit number with the file name. This is a number for file reference and should not be confused with disk unit numbers.

File Unit Numbers

The file unit numbers that may be used to refer to FORTRAN data files depend on your PTDOS configuration. If your PTDOS configuration allows fewer than 18 files, 16 FORTRAN unit numbers are available. They are 0, 1, ..., 15. You can then use any unit number between 0 and 15, where units 0 and 1 refer to the terminal. This does not necessarily mean that there are enough PTDOS files available so that you can open all 16 files, however.

If the system configuration allows for more than 17 files, the number of FORTRAN units available is one less than the number the system is configured for. For example, if your PTDOS configuration allows 35 files, there are 34 FORTRAN units available (0, 1, ..., 33).

The maximum number of FORTRAN units that can be available is 128. Unit numbers 0, 1, ..., 127 are available when the system is configured for 129 or more files.

Opening Files

The OPEN subroutine is a system routine for opening files. This routine is discussed in more detail in the next section, but it must be introduced here since it is a necessary part of file input and output. The form of the subroutine call is:

```
CALL OPEN (unit,file name{,buffer})
```

The OPEN subroutine associates a logical unit number (other than 0 or 1) with a PTDOS disk file name. In the last argument, you can specify a buffer address or request dynamic buffering (see section 6.1.1 for more information).
An example of opening and reading from a disk file follows:

```
&
CALL OPEN(2,'INVEN')
DO 10 I = 1,1000
READ(2,*,30) IORD --Reads from disk file INVEN
IF (IORD .EQ. 79421) GO TO 20
10 CONTINUE
GO TO 30
20 TYPE 'ORDER NO. 79421 IS ON RECORD NO. ',I
STOP
30 TYPE 'ORDER NO. 79421 NOT FOUND'
END
```

**READ Statement**

General form:

```
READ(unit, format (,)end-of-file, error)) input list
     |     |     |     | May include
unit | label of a statement labels | FORMAT statement, *, or null | variables, array names, array elements, implied loops, or strings
     |     |     |     | May include
     |     |     |     | variables, array names, array elements, implied loops, or strings
```

Reads values for each item on the input list from the specified unit using the specified format.

Example:

```
READ(0,*) 'What are the values? ', VAL1, VAL2
READ(3,100)EL(2,3),A
READ(2,10,150,320) ARAY
READ(5,150,,110)(TABL(I),I=2,5)
```

The READ statement reads values for each item in the input list. It reads the values from the specified unit according to the FORMAT statement whose label is the second argument. If the second argument is *, the values are read in free format. If the second argument is null, the values are read as binary.

If an end of file is encountered while data is being read, control transfers to the statement whose label is given as the third argument. The fourth argument indicates where control is to transfer if a read error (other than end of file) occurs.

Items in the input list can be simple variable names, array names, array elements, or implied DO loops. If you are reading from the terminal (unit = 0 or any file opened using $CONIN), you can include character strings in the input list. This is useful when prompting for input.

During input one value is read and assigned to each variable name or array element. If an array name appears in the list, the program reads every element of the array.

An implied loop is an abbreviated DO loop specifying which elements of an array are to be input. Its form for a one-dimensional array is:
(array name(var), var=n1,n2,incr)

variable | step size
first value | last value

where n1, n2, and incr can be constants or nonsubscripted variables.

For example:

READ(0,10)(X(IND),IND=1,3) Reads three values and assigns them to X(1), X(2), and X(3).

Implied DO loops can be nested to any depth. That is, one can be included within another. For example:

READ(2,100)((ARAY(I,J),I=1,4),J=1,30,2)

In this example the inner loop (I=1,4) is performed for each iteration of the outer loop (J=1,30,2).

Program Example:

$OPTIONS X
   CALL OPEN(5,'FILM')
   READ (5,10,100)A,B,C
   .
   10 FORMAT(3F10.2)
   100 STOP 'ERROR IN READING FILE ' 'FILM'
   .
   END

WRITE Statement

General form:

WRITE(unit, format [,end-of-file, error]) output list

unit number FORMAT statement labels May include vari-
statement label, *, or null able names, array
or null names, array ele-
Writes the values of items ments, implied DO
in the input list on the loops, and
specified unit using the character strings

Examples:

WRITE(1,*) 'The answer is ', ANSWER
WRITE(2,10,150) 'AMOUNT ', A(1,4)
WRITE (2,10) TABLE
WRITE(1,100) ((A(M,N),M=1,3),N=1,4)

The WRITE statement writes values for each line in the output list on the specified unit according to the specified format. The third argument tells where control is to be transferred if an end of file is encoun-
tered. If an error (other than end of file) occurs during output, control transfers to the statement label given in the fourth argument.

Items in the output list can include simple variable names, array elements or names, and implied loops. See the READ statement for a discussion of these items. Character strings can be included in a WRITE statement regardless of the output unit.

Note:
You cannot include operators or function names in the output list of a WRITE statement.

Program Example:

```fortran
$OPTIONS X,G
CALL OPEN(3,'SQUARE')
DO 100 I=1,100,.5
A = I**2
100 WRITE (3,10) A
10 FORMAT(F10.2)
END
```

REWIND Statement

General form:

```
REWIND unit
```

| unit number | Moves the pointer for the next read or write to the beginning of the file. |

Example:

```
REWIND 3
```

The REWIND statement causes the next input or output operation to occur at the beginning of the file.

BACKSPACE Statement

General form:

```
BACKSPACE unit
```

| unit number | Positions the file at the beginning of the previous record. |

Example:

```
BACKSPACE 3
```

The BACKSPACE statement causes the next read or write to occur at the beginning of the preceding record. For example, when part of the second record below has just been read, executing a BACKSPACE statement means the value 12 will be read next.

```
12, 13, 19
20, 50, 72
```
ENDFILE Statement

General form:

ENDFILE unit

| unit number

Writes an end of file on the specified unit.

Example:

ENDFILE 3

The ENDFILE statement writes an end of file on the specified unit at the current read or write position. Any data beyond that position is lost.

Program Example:

CALL OPEN(2,'DAT1')
DO 10 I = 1, 1000
READ (2,*)S
IF (S.GE. 1000) GO TO 20
10 CONTINUE
STOP
20 ENDFILE 2
END

5.5.4. Dynamic Formatting

You can include statements in your program that allow a user to enter a format for input or output at execution time. The format entered by a user should be read in your program using A6 format. Using A6 format causes the input values to completely fill the computer words. The characters making up the format should be stored in an array of real or integer variables.

In the program that performs dynamic formatting, substitute the array name for the format number in the READ or WRITE statement. An example of this follows:

DIMENSION FORM(10)
READ (0,10) 'ENTER THE DATA FORMAT :', FORM
10 FORMAT (10A6)
   ! This field specification can read up to 60 characters.
CALL OPEN(4,'TEST')
READ(4,FORM) A,B,C
WRITE(1,FORM) A,B,C

When executed, this program reads the first three values from file TEST in a format entered at the terminal. It then writes the values on the terminal in the specified format. An example of execution follows (the object file is named XOBJ):

User:  *XOBJ <CR>
Program: ENTER THE DATA FORMAT: (I8/I8/I8) <CR>
                564123
                666
                1112772
                STOP  END IN - MAIN

Typed by the user

Notice that the user enters the entire argument list for a FORMAT statement, including the parentheses.
5.5.5. Binary Input and Output

Binary input and output saves storage space, though it is sometimes less convenient than symbolic input and output. All that is necessary to write binary data on a file is to leave the second argument of the WRITE statement blank. For example:

```
WRITE(5) (BINARY(I), I=1,100)  \(\text{Writes binary values on unit 5.}\)
```

The values are written in binary format, six bytes for each value. This form of output saves time because the values are written exactly as they are stored in memory.

To read a file of binary data, leave the second argument of the READ statement null. For example:

```
READ(2,,100,520)A
```

The input items are read as six-byte values and stored directly in memory (no conversion is necessary).

5.6. DECLARATION STATEMENTS

Most statements in a FORTRAN program are execution statements; that is, they perform an operation. Declaration statements do not perform an operation, but contain information essential to the program's operation. For example, END is a declaration statement that defines the end of a routine. Another declaration statement that has been discussed is the FORMAT statement.

In addition to these declaration statements, there are declaration statements that specify the types of variables, assign values to variables, and set aside storage space for arrays.

5.6.1. Type Declarations

PTDOS FORTRAN automatically assigns a type of integer to variable names beginning with I through N and real to names beginning with A through H and O through Z. You can override this implicit type setting by using the type declaration statements. The types that can be assigned to variables are INTEGER, REAL, and LOGICAL.

**INTEGER Statement**

```
General form:

    INTEGER var1, var2, ... \(\text{Declares that var1, var2, etc. are integer variables.}\)
    \text{variables or array dimensions}

Examples:

    INTEGER X,Y,Z
    INTEGER A,B,ARRAY(3,4)
    INTEGER CNT(20)
```

The INTEGER statement causes the variables listed to be integer variables regardless of their names. You can also declare dimensions for integer arrays in the INTEGER statement. Arrays are discussed in the next unit.

**Program Example:**

```
INTEGER ANSWER
ANSWER = 3.3 * 2.0
TYPE ANSWER
END
```

When executed, this program displays 6, not \(0.66000000E\, 001\), because ANSWER is an integer variable.

5-26 FORTRAN
REAL Statement

General form:

    REAL var1, var2, ...  Declares that var1, var2, etc. are real variables.
    |    | variables or array dimensions

Examples:

    REAL IVAL, LINK
    REAL IDS(15), NUM

The REAL statement causes the variables listed to be real variables regardless of their names. You can also declare dimensions for real arrays in the REAL statement. Arrays are discussed in the next unit.

Program Example:

    REAL IANS
    IANS = 7 * 2.3
    TYPE IANS
    END

This program displays 0.16100000E 002, not 16, because IANS is real.

LOGICAL Statement

General form:

    LOGICAL var1, var2, ...  Declares that var1, var2, etc. are logical variables.
    |    | variables or array dimensions

Examples:

    LOGICAL IANS, VAL
    LOGICAL COMP(3), X

The LOGICAL statement is equivalent to the INTEGER statement. You can assign the logical values .TRUE. and .FALSE. to variables declared as logical, but PTDOS FORTRAN also allows numerical variables to have logical values. Thus, variables declared as LOGICAL are no different from variables declared as INTEGER. The LOGICAL statement allows compatibility with other FORTRANS.

You can also dimension arrays in the LOGICAL statement. Arrays are discussed in the next unit.
IMPLICIT Statement

General form:

```
IMPLICIT type  (letter list)  Declares a default
     /   type for the variables
     INTEGER, REAL  One or more beginning with the
     or LOGICAL    alphabetic indicated letters.
     characters or  spans of characters
     indicated by char1-char2
```

Examples:

```
IMPLICIT INTEGER (O-Z),REAL (A-C,L)
IMPLICIT REAL (I,L-N),INTEGER (X)
```

The IMPLICIT statement changes the default type of variables beginning with the indicated letters. Individual letters and letter spans are separated by commas. Letter spans must be in ascending order. For example, (A-L) is valid but (L-A) is not.

If used, the IMPLICIT statement MUST be the first statement in the routine except for comments. That is, it must be the first statement of a main routine or immediately follow the SUBROUTINE, FUNCTION or BLOCK DATA statement if used in a subprogram.

Program Example:

```
IMPLICIT REAL (A-Z)  All variables are real.
DO 10 I=0,1,.1
    K = SIN(I)
  10 TYPE K
END
```

5.6.2. Arrays

In PTDOS FORTRAN an array is a collection of values that are referred to by the same name. Each value is an element of the array and is specified by subscripts. For example, if VEC is an array with three elements, you can refer to the individual elements of VEC as follows:

```
VEC(1) refers to the first element.
VEC(2) refers to the second element.
VEC(3) refers to the third element.
```

Subscripts can be real or integer expressions. Real values are truncated before use.

An array can have more than one dimension. An array with two dimensions can be pictured as an arrangement of rows and columns. For example, ARAY is a 2 by 3 array with the following elements:

```
 32  60
 15  50
 10  22
```

Elements of ARAY are referred to with two subscripts, the first changing more rapidly than the second. Thus:
ARAY(1,1) = 32
ARAY(2,1) = 60
ARAY(1,2) = 15
ARAY(2,2) = 50
ARAY(1,3) = 10
ARAY(2,3) = 22

The elements of a multi-dimensioned array are stored so that the first subscript varies most rapidly and the last subscript most slowly. For example, the sequential arrangement in storage for a $2 \times 2 \times 2$ array named A is:

$$A(1,1,1), A(2,1,1), A(1,2,1), A(2,2,1),$$
$$A(1,1,2), A(2,1,2), A(1,2,2), A(2,2,2)$$

You can use the DIMENSION statement to assign extra space to a variable name so that it can contain an array of values. Every array or subscripted variable must be declared in a DIMENSION statement or a type declaration before the first executable statement of the routine.

**Note:**

*Subscripted variables cannot be used as subscripts. For example, A(X(I)) is not valid.*

**DIMENSION Statement**

General form:

```
DIMENSION var1(n1, n2, ...), var2(n1, n2, ...), ...
```

Sets aside $n1 \times n2 \times \ldots$ words of storage for the specified variables.

Examples:

```
DIMENSION EXPNS(10,10)
DIMENSION A(2,3,7,2), BATCH(30)
```

The DIMENSION statement defines one or more arrays having one or more dimensions. The size of each array is $(n1 \times n2 \times \ldots)$ elements that require $(n1 \times n2 \times \ldots) \times 6$ bytes of storage at execution time. The number of dimensions cannot exceed 7.

**Program Example:**

```
DIMENSION ACTS(10,20)
CALL OPEN(2,'ACTR')
READ(2,*)((ACTS(I,J),I=1,10),J=1,20)
.
.
END
```

In subroutines, which are discussed in the next section, you may use integer variables for dimensions of an array passed to the subroutine. In the main routine, the dimensions must be integer constants.

An array must be dimensioned everywhere it is used. If you pass an array to a subroutine in the argument list, you must dimension the array in the subroutine even if you use variable dimensions instead of constants. If the values of an array's dimensions differ in the main routine and a subroutine, then only those sections of
the array specified in the subroutine can be used in the subroutine.

Program Example:

```
$OPTIONS X
  DIMENSION AR(100)
  CALL READR(AR)
  TYPE 'THE TENTH ELEMENT = ',AR(10)
END

$OPTIONS X
  SUBROUTINE READR(X)
  DIMENSION X(IDUMMY)
  TYPE 'ENTER 10 VALUES'
  ACCEPT (X(I),I=1,10)
  RETURN
END
```

If you specify a multi-dimensional array with variable subscripts in a subroutine, the actual values of the variables are used for subscript calculation at runtime. For example:

```
REAL ITEMS(A,B,C)
```

The size of array ITEMS is A*B*C.

5.6.3. Initializing Variables

### DATA Statement

**General form:**

```
DATA var1/constant list1/,,var2/constant list2/,...
```

- `variable,` - one or more constants
- `array element,` - or
- `array name` - Assigns initial values to the
  variables or array elements.

**Example:**

```
DATA ALP/35.2/,,IND/10/
DATA VECTOR/3,2,1/,,ARAY(7,1)/11.2/
DATA ISTR/'MON'/,,VEC/3*0/
```

The DATA statement initializes variable values before program execution. The constant(s) following a variable are assigned to the variable. For example:

```
DATA SINGLE/15.9/                      Initializes variable SINGLE to 15.9.
DIMENSION SEVRAL(9)
DATA SEVRAL/2,2,2/                    Initializes the first three elements
                                      of array SEVRAL to 2.
```

You can use an asterisk to indicate repetition of a value in a DATA statement. For example, the following DATA statements are equivalent:
DATA SEVERAL/2,2,2/
DATA SEVERAL/3*2/

Character strings in DATA statements are assigned to variables left-justified and filled with nulls (binary zeros) on the right. For example:

DATA CHR/'A'/ 
Assigns A followed by five nulls to variable CHR.

You can also initialize arrays with string values. For example:

DIMENSION CHARS(3)
DATA CHARS/3*'FIRST'/

An error occurs if a variable initialized in a DATA statement is not used within the program. DATA statements are processed after the END statement, so errors in DATA statements appear after the END statement in the listing file. DATA statement errors include the statement number and the name of the variable being initialized when the error occurred.

5.6.4. The COMMON Declaration

The COMMON declaration sets aside a block of memory locations that can be shared by different routines of a program. Variables and arrays named on one COMMON declaration share storage with variables and arrays named in another COMMON declaration. For example:

DIMENSION ARAY(100)
COMMON X,Y,ARAY CALL SUB1
.
.
END
SUBOUTINE SUB1
DIMENSION XRAY(2,5)
COMMON A,B,XRAY
.
.
RETURN
END

COMMON allows you to transmit data to and from subprograms without passing it in arguments.

You can include array declarations in the COMMON declaration. For example:

COMMON X,Y,ARAY(100)

has the same effect as

DIMENSION ARAY(100)
COMMON X,Y,ARAY

Note that the DIMENSION statement must precede the COMMON statement. The COMMON declarations discussed so far are blank COMMON declarations; that is, they are not labelled. Blank COMMON is indicated by no label or by two slashes. For example:

COMMON X,Y,Z
COMMON //X,Y,Z

Blank COMMON is shared among the routines of a program. In a given routine, elements may be added to blank COMMON by a series of COMMON declarations. If blank and labelled COMMON (described below) are used, blank COMMON is treated like labelled COMMON with a label of blank.
You might want to write a program that has several subprograms using variables in COMMON storage. In this case, every subprogram probably will not use every variable in COMMON storage. You can avoid including the entire COMMON declaration in all subprograms by labelling COMMON blocks. Labelled COMMON permits a main routine to share one part of COMMON storage with one subprogram and another part with another subprogram. For example:

```
COMMON /ADDR/ A1, A2 /ZIP/ Z1, Z2
:
CALL ADDRESS
CALL ZONES
END
SUBROUTINE ADDRESS
COMMON /ADDR/ A(2)
:
RETURN
END
SUBROUTINE ZONES
COMMON /ZIP/ X, Y
:
RETURN
END
```

The main routine shares COMMON block ADDR with subroutine ADDRESS and shares COMMON block ZIP with subroutine ZONES.

In this example, ADDR and ZIP are COMMON block labels. COMMON block labels may have six characters but only the first five are retained. A label cannot be the same as a subroutine or function name.

### COMMON Statement

**General forms:**

```
COMMON list
COMMON /label1/ list1 /label2/ list2 ...
```

- Up to 6 characters. Only the first 5 are retained.
- One or more variables, array names, or array declarations.
- Specifies memory locations to be shared by other routines in the program.

**Examples:**

```
COMMON NAME, ADDR, ZIP
COMMON /COUNTS/C1,C2,C3/FREE/A(20),X,ANS
COMMON / LABL1/A,B,C/LABL2/X,Y,Z//BLANK(20)
```

The COMMON statement specifies variables and arrays that are to share storage with variables and arrays named in other COMMON statements. If the first form is used, all variables and arrays in the list are shared. If labels are used, only the variables following the specified labels are shared.
Program Example:

```
COMMON RATE,Y
ACCEPT 'ENTER THE INTEREST RATE ', RATE
CALL RULE72
TYPE 'MONEY DOUBLES IN ', Y, ' YEARS'
END
SUBROUTINE RULE72
COMMON R,TIME
TIME = 72/R
RETURN
END
```

Blank and labelled COMMON can be specified in the same COMMON statement. For example:

```
COMMON A,B,C/TALLY/X,Y,Z/NEW/I,J,K//ARRAY(10)
```

In this statement, A,B,C, and ARRAY(10) are in blank COMMON; X, Y, and Z are in COMMON block TALLY; and I, J, and K are in COMMON block NEW.

5.7. SUBROUTINES AND FUNCTIONS

You need not copy the same code over and over for a computation or other process that must be repeated several times in a program. You can separate the code from the rest of the program and call it as a subroutine or function.

Subroutines and functions consist of statements stored together outside the main routine. Each subprogram must end with an END statement.

In PTDOS FORTRAN subroutines and functions cannot be compiled separately from the main program. They must all be included in a single source program file and be compiled together.

A subroutine is used via a CALL statement. During execution of a program, control transfers to a subroutine when a CALL statement is encountered and returns to the calling program when a RETURN statement is encountered.

A function is used by including its name in an expression. Control transfers to the function, which assigns a value to the function name before returning to the calling program. That value is then used where the function name appears in the expression.

Both subroutine and function calls can include arguments. Items in the argument list of a subprogram call pass values to and from corresponding items in the parameter list of a subprogram declaration. For example:

```
A = 12
B = 56.7
CALL PASS (A, B, C)
TYPE C
END
SUBROUTINE PASS(X, Y, Z)
Z = X + Y
RETURN
END
```

In this example, the subroutine uses the values of A and B assigned in the main program for X and Y. The subroutine computes a value for Z and the main program has access to that value through argument C.

The information that is actually passed from the argument list of a subprogram call to the parameter list of a subprogram declaration is the address at which each value is stored.

Whenever a variable in a subprogram appears in the parameter list of the subprogram's declaration, the compiler interprets the information passed as an address, not as a value. In other words, the variable reference is indirect. Furthermore, when a parameter receives a new value in a subprogram, that value is assigned to the
address passed in the subprogram call.

Indirect reference can lead to unexpected results in certain situations. The arguments in a subprogram call can be simple or subscripted variables, expressions, or constants. A subprogram should not assign new values to parameters if the corresponding arguments are constants or expressions.

When an argument in a subprogram call is an expression, the calling program evaluates the expression, stores the value in a temporary address, and passes that address to the subprogram. If the corresponding parameter of the subprogram receives a new value, that value is assigned to the temporary address, which is not known in the calling program. For example:

```
B = .2
C = .3
CALL EXP(B**2,C)          --The value of the first argument is not changed by
TYPE B,C
END
SUBROUTINE EXP(X,Y)
X = SIN(X)
Y = SIN(Y)
RETURN
END
```

When an argument in a subprogram call is a constant and the corresponding parameter receives a new value in the subprogram, the following occur:

(a) The new value is assigned to the address of the constant.
(b) The constant now has a new value in the calling program.

For example:

```
CALL SUBX(2,B)           --The value of 2 is 10 upon return from SUBX because I was
TYPE '2 = ', 2
TYPE 'B = ', B
END
SUBROUTINE SUBX(I,J)
I=10
J=10
RETURN
END
```

When executed, this routine displays:

```
2 = 10
B = 0.10000000E 002
```

**Note:**

*The following cannot be used for subroutine, function or common names: A, B, C, D, E, H, I, M, SP, PSW, or any PTDOS reserved name contained in PTDEFS.*

### 5.7.1. Subroutines

A *subroutine* is an independent group of statements that are activated by a `CALL` statement. Each time the `CALL` statement is executed, the statements of the subroutine are executed.

The essential parts of a subroutine are:
SUBROUTINE subroutine name ((parameter list))
.
RETURN
.
END

**CALL Statement**

**General form:**

```
CALL subroutine name ((argument list))
```

| Constants, simple or subscripted variables, or expressions | Executes the specified subroutine. |

**Examples:**

```
CALL COMP(X)
CALL EMP(100,N)
CALL ERRMSG
```

The CALL statement executes the specified subroutine, passing values to it in the argument list. When the subroutine has completed execution, it may pass the new values back to the calling routine. Then execution continues with the statement following the CALL statement.

One way to pass values to and from subroutines is to include them as arguments in a CALL statement. Variable values cannot be passed by using the same variable name in the calling program and the subroutine. A variable name is local to the routine in which it appears.

**Program Example:**

```
ACCEPT 'ENTER A NUMBER ',N
CALL FACT(N,ANS)
TYPE N,' FACTORIAL = ',ANS
END
SUBROUTINE FACT(NUMBER,ANSWER)
    ANSWER = 1
    DO 10 I = 1,NUMBER
        10   ANSWER = ANSWER * I
    RETURN
END
```
SUBROUTINE Statement

General form:

```
SUBROUTINE subroutine name ([parameter list])
    1 to 6 characters variables
    Only the first 5 have significance.
    Declares that subsequent statements up to an END constitute a subroutine.
```

Examples:

```
SUBROUTINE CHECK(BAL)
SUBROUTINE ACT(EXP,INC,DEB)
SUBROUTINE ERRMSG
```

The SUBROUTINE statement is the first statement of a subroutine. The subroutine name can have up to six characters but only the first five are seen by the assembler. For example, SUBR1 and SUBR12 represent the same subroutine to the assembler.

The parameters in the list are used in the subroutine only. Each parameter receives a value from or passes a value to an argument in the corresponding position of a CALL statement.

Program Example:

```
$OPTIONS X
REAL INT
INTEGER YEARS
ACCEPT 'ENTER PRINCIPAL, INTEREST, & YEARS: ', PRIN, &INT, Y
CALL GROW(PRIN,INT,Y,TOTPRN)
TYPE 'AFTER ',Y,' YEARS THE PRINCIPAL IS ',TOTPRN
END
$OPTIONS G,X
SUBROUTINE GROW (PRIN, RATE, ITIME, FINVAL)
FINVAL = PRIN
DO 10 I = 1, ITIME
    FINVAL = FINVAL + FINVAL*RATE
10     CONTINUE
RETURN
END
```

In this example, the CALL statement passes the values of PRIN, INT, and Y to subroutine GROW. The values are assigned to variables PRIN, RATE, and ITIME in the subroutine. Subroutine GROW passes the value of FINVAL to TOTPRN in the main program. In fact, all values in the CALL argument list are passed to the corresponding names in the subroutine's parameter list and the reverse happens after execution of the subroutine.
## RETURN Statement

<table>
<thead>
<tr>
<th>General form:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RETURN</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Exits a subroutine or function</strong></td>
</tr>
<tr>
<td><strong>and returns control to the statement following the call.</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Example:</td>
</tr>
<tr>
<td><strong>RETURN</strong></td>
</tr>
</tbody>
</table>

Use the **RETURN** statement anywhere in a subroutine or function to return control to the routine that called the subprogram. Control is transferred to the statement following the **CALL** statement.

If a **RETURN** statement is not executed in a subprogram, control will not be returned to the calling routine, rather execution will terminate at the end of the subprogram.

The **RETURN** statement is not valid in the main program.

### 5.7.2. Functions

A **function** is an independent group of statements that are activated by referring to the function name in an expression. The function computes a value and assigns that value to its name. Thus, the function name returns a value that can be used as part of an expression.

In the absence of type declarations, the first letter of a function’s name determines whether the value returned is integer or real. Functions whose names begin with **I** through **N** return integer values. All others return real values.

The type of a function can be changed by declaring the type in the function declaration. For example, the function **FUN** declared below, returns an integer value:

```plaintext
INTEGER FUNCTION FUN(ARG)
```

Another way to set the type of a function is to use a type statement within the function definition. For example:

```plaintext
FUNCTION FX(Y)
INTEGER FX       ——FX is an integer function.
FX = SIN(Y)
RETURN
END
```

A function must have at least one argument even if it is a dummy argument.

The essential parts of a function are:

```plaintext
FUNCTION function name (parameter list)
  .
  .
  RETURN
  .
END
```
FUNCTION Statement

General form:

\[
\text{(type) FUNCTION function name (parameter list) } \\
\text{ | | | | | } \\
\text{REAL, \text{ 1 to 6 characters } one or more } \\
\text{INTEGER, \text{ Only the first 5 } variable names } \\
\text{or \text{ have significance}} \\
\text{LOGICAL} \text{ Declares that the subsequent statements up to an } \\
\text{END constitute a function.}
\]

Examples:

FUNCTION XCHNG(X,Y)
FUNCTION PWR(VAL)

The FUNCTION statement is the first statement of a function. The function name can have up to six characters but only the first five are seen by the assembler. There must be at least one parameter in the parameter list even if it is a dummy parameter. You must assign a value to the function name within the function.

The optional type declaration specifies the type (real, integer, or logical) of value returned in the function. For example:

FUNCTION ANS(X)

returns a real value because the name begins with A, but

INTEGER FUNCTION ANS(X)

returns an integer value.

A function is called by including its name and arguments in an expression. The following statement is an example of a function called by specifying its name in an expression:

\[\text{ANS = PWR(17) + X}\]

The value returned for PWR depends upon the value of the argument (17 in this case).

Warning:

If you pass a constant as a parameter in a function call, the value of the constant can be changed in the main program. For example:

\[\text{ANS = PWR(17) + X}\]

The value of 17 is zero after the function call.

\[\text{. . . END FUNCTION PWR(A)}\]
\[\text{PWR = A**A}\]
\[\text{A = 0}\]
\[\text{RETURN}\]
\[\text{END}\]
Program Example:

C Main Program
DIMENSION A(100)
ANS = SUMSQ(A) / 100
TYPE 'ANSWER = ', ANS
END
C Function Definition
FUNCTION SUMSQ(X)
DIMENSION X(IDUM)
DO 10 I = 1, 100
SUMSQ = SUMSQ + X(I)**2
10 CONTINUE
RETURN
END

5.7.3. BLOCK DATA Subprogram

The BLOCK DATA subprogram initializes variables that appear in COMMON declarations. BLOCK DATA contains no executable statements, but contains declaration statements for establishing dimensions and values for variables stored in COMMON. The statements that may appear in a BLOCK DATA subprogram are:

```
BLOCK DATA
DIMENSION
COMMON
DATA
REAL
INTEGER
LOGICAL
IMPLICIT
END
```

The first statement of the routine.
Defines arrays.
Names variables and arrays in COMMON. All elements must be listed whether they are initialized or not.
Initializes variables.

Declare variable types.
The last statement of the routine.

For example:

```
BLOCK DATA
COMMON ALPHA, BETA /LABL/X,Y,A(10)
DATA /ALPHA/100, /X/58.3
END
```

5.8. COPYING SOURCE FILES

Commonly used source files can be incorporated into any routine using the COPY statement. The statements from the copied file become part of the new routine during compilation.

COPY Statement

General form:

```
COPY  file name
      | PTDOS  file name
```

Copies the statements contained in the specified into the current program.

Example:

```
COPY REPORT
```
The COPY statement inserts source statements from a specified file into the current program. The statements are inserted where the COPY statement appears in the program.

Program Example:

```
IMPLICIT REAL (A-Z)
ACCEPT 'ENTER PRINCIPAL, INTEREST, & YEARS: ', &PRIN, INT, Y
COPY INTRST
TYPE 'AFTER ', Y, ' YEARS THE PRINCIPAL IS ', TOTPRIN
END

INTRST is a text file containing the statements below. The statements are automatically inserted after line 3 of the above program during compilation:

FINVAL = PRIN
DO 10 I=1,Y
   FINVAL = FINVAL + FINVAL * INT
10 CONTINUE
```
SECTION 6

SYSTEM SUBROUTINES

6.0. INTRODUCTION

PTDOS FORTRAN supplies a number of subroutines that enable you to use the more advanced features of the system. Using the system subroutines you can access many PTDOS file handling commands, have random access to data on files, abort or delay execution, directly address memory, plot data on the screen, etc.

In the descriptions of the individual routines that follow, you can substitute the name of a variable or array containing a character string anywhere a character string is specified. For example, the second argument of the OPEN subroutine is a character string. A variable name can be used as follows:

\[ A = 'MYFILE' \]
\[ \text{CALL OPEN}(2, A) \]

**Note:**

A file name includes all characters up to the first blank or null.

6.1. FILE HANDLING

The system file-handling subroutines let you open and close files, execute PTDOS file commands from your program, and gain random access to files. These routines make the power of the Helios II Disk System available to your FORTRAN programs.

6.1.1. Opening and Closing Files

The OPEN and CLOSE subroutines control access to PTDOS files. You must open a file before reading from or writing to it. All open files are closed automatically when the program terminates or chains to another program. If you want to protect a file from future access in a program or reuse the unit number, close the file with the CLOSE subroutine.

**OPEN Subroutine**

**General form:**

\[ \text{CALL OPEN} \ (\text{unit, 'file name' } [], \text{ buffer}) \]

- **logical unit**
- **PTDOS file**
- **buffer address**
- **name**

Opens the specified file and associates the name with a logical unit.

**Examples:**

\[ \text{CALL OPEN}(2, 'MYFILE') \]
\[ \text{CALL OPEN}(3, 'SML', 65535) \]
\[ \text{CALL OPEN}(5, 'ARCHV/1') \]
The OPEN subroutine provides access to a disk file. The first argument can be any unit number except 0 or 1 (0 is reserved for terminal input and 1 is reserved for terminal output).

For the second argument, you can enter the file name as a character string or use a variable or array name and assign a file name to it or enter the name at execution time.

You can specify a buffer address in the last argument. The address must be external to the system and to user-protected memory. Specifying your own buffer address reduces management overhead and doesn't take up system buffer space.

The system will do dynamic buffering if you specify 65535 as the buffer address. This causes PTDOS to allocate the buffer each time a request is made for the file and to deallocate the buffer when the request is fulfilled. Thus, if many files are open at the same time, a small amount of buffer area can service all the files.

If you open a PTDOS file that does not exist, the program will create the file with a type of . , a block size of 4C0 hexadecimal, and no attributes.

You can use the OPEN statement for terminal input and output using the following special file names:

\[
\begin{align*}
\&$CONIN \quad \text{terminal input} \\
\&$CONOUT \quad \text{terminal output}
\end{align*}
\]

The OPEN statement does not open the terminal, but associates it with a file unit. This feature can be useful when testing a program with terminal input and output before using the program for file input and output. For example:

```fortran
IF (TEST .EQ. 1) FN = '$CONIN'
CALL OPEN(2, FN)
READ (2, *) (A(I), I=1, N)
```

READ and WRITE statements associated with $CONIN and $CONOUT through the OPEN statement work exactly the same as reading unit 0 or writing unit 1. For example:

```fortran
CALL OPEN(11, '$CONIN')
READ(11, *) 'INPUT QUANTITY ', QUANT
and
CALL OPEN(10, '$CONOUT')
WRITE (10, 100) (A(I), I=1, 100)
```

**CLOSE Subroutine**

**General form:**

```fortran
CALL CLOSE (unit)
```

<table>
<thead>
<tr>
<th>logical unit number</th>
</tr>
</thead>
</table>

Closes a file that was previously opened with the OPEN routine.

**Example:**

```fortran
CALL CLOSE (3)
```

The CLOSE subroutine closes the specified file. The unit number is then available for reuse.
The CLOSE subroutine does not write an end of file at the current read or write position. If you want to write an end of file and delete the remainder of the file, use the ENDFILE statement before calling CLOSE.

**Note:**

*Attempting to close unit 0 or 1 causes a runtime error.*

**Program Example:**

```fortran
DIMENSION FN(2), DATA(20)
READ(0,10) 'INPUT FILE = ', (FN(I),I=1,2)
CALL OPEN(2,FN)
READ(2,*) (DATA(I),I=1,20)
CALL CLOSE(2)
READ(0,10) 'OUTPUT FILE = ', (FN(I),I=1,2)
CALL OPEN(2,FN)
WRITE(2,*) (DATA(I),I=1,20)
10 FORMAT(2A6)
END
```

This program copies 20 values from the input file to the output file. In the sample execution that follows, the user enters $CONIN$ for the input file and $CONOUT$ for the output file, so both input and output occur at the terminal.

**User:**

*COPY <CR> (Assuming the object file name is COPY)*

**Program:**

INPUT FILE = $CONIN$ <CR>

User:

1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0 <CR>

**Program:**

OUTPUT FILE = $CONOUT$ <CR>

0.10000000E 001 0.20000000E 001 0.30000000E 001

.
.

STOP END IN - MAIN

### 6.1.2. Random Access to Files

Normal access to data files is sequential. Each time a READ or WRITE statement is called, it reads or writes the next data item in sequence. Thus, reading a data item near the end of a file requires reading all data items up to that point.

With random access, you can set the next read or write position to any point in the file. Thus, you can read information from any location on a file without reading the information that precedes it, and you can overwrite particular values in a file without having to rewrite the entire file.

The position in a file is usually expressed as the number of bytes from the beginning of the file. For example, position 10 is eleven bytes from the beginning of the file (the byte at the beginning of the file is byte 0).

You may also express position as the number of blocks from the beginning of a file, as described under the SEEK subroutine.

The current position in a file is the place the next input or output operation will occur. After a read or write operation, the current position is just after the last byte read or written. The current position may be reset using any of the following:

- REWIND statement
- BACKSPACE statement
- SEEK subroutine
- SPACE subroutine

The subroutines described in this unit provide random access to any data file. The RANDOM subroutine (or the PTDOS RANDOM command) must be called first to make a file accessible as a random file. The SEEK
subroutine positions a random file to a specific byte or block. The SPACE subroutine spaces forward or backward relative to the current file position, or spaces to the end of the file. The CONTRL subroutine makes random access more efficient.

**RANDOM Subroutine**

General form:

```
CALL RANDOM (unit) Make the specified file
         | ready for random access.
logical unit number
```

Example:

```
CALL RANDOM(4)
```

The RANDOM subroutine makes an existing file available for random access. It has the same function as the PTDOS RANDOM command. The read or write position after the RANDOM routine has been executed is after the end of file. The RANDOM routine (or the PTDOS RANDOM command) must be called before you can position the file using the SEEK subroutine. RANDOM need only be called once for a particular file.

**SEEK Subroutine**

General form:

```
CALL SEEK (unit, position [,block])
logical unit number byte or block number position is a block no.
```

Examples:

```
CALL SEEK(2,132)
CALL SEEK(5,20,1)
```

The SEEK subroutine positions a file to the beginning of a specified byte or block. If the third argument is present, the location specifies a block number, otherwise it specifies a byte number.

The file indicated must have been set up for random access with the RANDOM subroutine or PTDOS RANDOM command. The position cannot be greater that 65,535 bytes or 128 blocks.
Program Example:

    CALL OPEN(2,'FOUT')
    CALL RANDOM(2)
    DO 10 I=1,10
      POS = 10*I
      CALL SEEK(2,POS)
      READ(2,5)C
      WRITE(1,5)C
    10 CONTINUE
    5 FORMAT(A6)
    END

If FOUT contains:

01234567890
9876543210987654321
11223344556677889900112233
123123123123123123123123123123123123123123

The program above reads and displays:

0          (byte 10)
109876     (byte 20)
1          (byte 30)
556677     (etc.)
001122
231231
312312
123123
231231
3123

Notice that the carriage return at the end of each record counts as one byte.

SPACE Subroutine

General form:

    CALL SPACE (unit, displacement, 'direction')

        logical unit  constant    +, -, or E
        number        or variable  number of bytes
                      to space over

Changes the file's next read or write position by the number of bytes specified in displacement.

Examples:

    CALL SPACE(3,50,'+')
    CALL SPACE(2,DIS,'-')
    CALL SPACE(5,1,'E')

The SPACE subroutine lets you space forward or backward a specified number of bytes from the current file position. The third argument indicates the direction of spacing or an advance to the end of the file: + means space forward, - means space backward, and E means advance to the end of the file. Any other value for
direction is treated as +. For example, the statements below position unit 3 to six bytes before the end of file:

```
CALL SPACE(3,1,'E')
CALL SPACE(3,6,'-')
```

A file can be spaced forward or backward up to 65,535 bytes. If the beginning or end of file is encountered, a runtime error is generated.

**Program Example:**

```
CALL OPEN(2,'DATA')
CALL RANDOM(2)
REWIND 2
READ(2,10)X,Y
CALL SPACE(2,6,'-')
READ(2,10)Z
CALL SPACE(2,60,'+')
READ(2,10)A
WRITE(1,10)X,Y,Z,A
10 FORMAT(4A6)
END
```

If DATA contains:

```
01234567890ABCDEFGHIJKLMNOPQRSTUVWXYZ
9876543210987654321
123123123123123123123123123123123123
```

the program above reads and displays:

```
01234567890AVWXYZ 231231
```

The first six characters of the first record are read and assigned to X, the next six characters are assigned to Y. The current position after the first READ statement is at the beginning of the second record. The SPACE subroutine backs up six bytes (the carriage return counts as a byte) and reads the last five characters of the first record for Z. Again, the current position becomes the first byte of the second record. The SPACE subroutine advances 30 bytes and reads the value for A (231231).

### 6.1.3. File Management

You can create files, kill files, and perform many other PTDOS file functions within a PTDOS FORTRAN program. The subroutines described in this unit work like the corresponding PTDOS commands described in section 1 of the PTDOS User’s Guide.
CREATE Subroutine

General form:

CALL CREATE('file name','type',block size)

PTDOS file  PTDOS size of disk blocks
name       file   type

Create a file with the
specified type and block
size.

Examples:

CALL CREATE('MYFILE','I', 200)
CALL CREATE('ZONES','F',51CO)

The CREATE subroutine creates a file with the specified name, type, and block size. Only the first letter is used for the file type. See section 3.5 of the PTDOS User's Guide for more about file types. No attributes are assigned to the file.

Note:

Attempting to create a file that already exists does not cause a runtime error.

KILL Subroutine

General form:

CALL KILL ('file name') Deletes the specified file

from the PTDOS system.

Example:

CALL KILL('DAT3')

The KILL subroutine deletes the specified file from disk. No error is generated if the file does not exist.

CHNAME Subroutine

General form:

CALL CHNAME('old name', 'new name') Changes the file

name from old

name to new name.

Example:

CALL CHNAME('ADAT','FINAL')

The CHNAME subroutine changes the name of a PTDOS file.
CHTYPE Subroutine

General form:

CALL CTYPE('file name','type')

PTDOS file name PTDOS type

Changes the type of the specified file to that indicated.

Example:

CALL CTYPE('XFIL','T')

The CTYPE subroutine changes the type of the specified file to the PTDOS file type given in the second argument.

CHATTR Subroutine

General form:

CALL CHATTR('file name', attributes)

PTDOS file name PTDOS file attributes

Changes the attributes of the specified file to those listed.

Examples:

CALL CHATTR('MYFILE',1)
CALL CHATTR('NUMS',32)

The CHATTR subroutine resets the attributes of a PTDOS file to those specified. Attributes are specified as bits in the first byte. Single attribute values are:

1  KILL protected
2  WRITE protected
4  READ protected
8  INFORMATION protected
16  ATTRIBUTE change protected
32  NAME and TYPE change protected
64  Disk allocation prohibited
128 User attribute

For more information about attribute values, see the PTDOS User's Guide.

FORTRAN does not check the new attributes.

Caution:

Remember that a file with both attribute protection and kill protection cannot be removed from disk.
INFO Subroutine

General form:

CALL FINFO('file name', array name)  Retrieves status information about the specified file and stores it in the array.

Example:

CALL FINFO('D1FIL', D1INFO)

The FINFO subroutine retrieves status information about a file and places it in the specified array. The form described on page 2-24 of the PTDOS User's Guide, under FINFO. The array receiving the information must have at least four elements.

See documentation of the CBTOF routine in unit 6.4 for a program example of FINFO.

SETUNT Subroutine

General form:

CALL SETUNT(drive)  Changes the system default drive to the one specified.

PTDOS disk unit

Example:

CALL SETUNT(1)

The SETUNT subroutine changes the system default drive to the one specified. The default drive is the disk unit searched when you specify a file name without appending /unit. For example:

CALL SETUNT(1)
CALL OPEN(5,'MULT')  This statement attempts to open MULT on disk unit 1.

Program Example:

CALL CREATE('SDATE', 'F', $100)
CALL OPEN(2,'SDATE')
TYPE 'ENTER SALES
DO 10 I=1,100
ACCEPT '?', SLSNO, SLSAMT
IF (SLSNO .EQ. 0) GO TO 20
WRITE(2,30) SLSNO, SLSAMT
10 CONTINUE
20 CALL CHATTR('SDATE', 3)
30 FORMAT(I8,F12.2)
END

This program reads values from the terminal and stores them on a disk file. It then sets the attributes of the file to KILL and WRITE protected.
**CONTRL Subroutine**

General form:

```
CALL CONTRL(unit,op,DEin,HLin,Aout,DEout,HLout)
```

<table>
<thead>
<tr>
<th>Logical unit number</th>
<th>CTLOP operator</th>
<th>Values returned in DE and HL registers</th>
<th>Values input to DE, HL, and A registers</th>
</tr>
</thead>
</table>

Allows program control over devices or returns information about devices.

Examples:

```
CALL CONTRL(0,2,0,'?',0,0,0)
CALL CONTRL(FILE,4,0,$6500,0,0,0)
```

The CONTRL subroutine implements the CTLOP system call, described in section 2 of the PTDOS User's Guide. Using CONTRL, you can perform any of the following operations from your program:

<table>
<thead>
<tr>
<th>CTLOP operator</th>
<th>Operation performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Return driver status</td>
</tr>
<tr>
<td>1</td>
<td>Forms control</td>
</tr>
<tr>
<td>2</td>
<td>Set PTDOS prompt</td>
</tr>
<tr>
<td>3</td>
<td>Reset device</td>
</tr>
<tr>
<td>4</td>
<td>Load random index</td>
</tr>
<tr>
<td>5</td>
<td>Echo on</td>
</tr>
<tr>
<td>6</td>
<td>Echo off</td>
</tr>
<tr>
<td>7</td>
<td>Special status read</td>
</tr>
</tbody>
</table>

The values passed to and from the routine in the DE, HL, and A registers are described in the PTDOS User's Guide under the description of the CTLOP operation.

Examples:

```
CALL CONTRL(0,2,0,'?',0,0,0)
Sets the PTDOS prompt to '?' instead of no prompt.

CALL CONTRL(2,4,0,$6500,0,0,0)
Moves the index block of the random file on unit 2 from disk into memory location 6500. Since each SEEK operation on a random file refers to the index block, making the index block available in memory decreases the access time.
```

**Program Example:**

```
IMPLICIT INTEGER (A-Z)
CALL CONTRL(7,0,0,0,A,DE,HL)
TYPE 'PROTECTION = ',A
TYPE 'CHRS = ',DE
END
```

This program displays the attributes and device characteristics of file number 7.
6.2. SPECIAL TERMINAL INPUT AND OUTPUT

Besides the terminal input and output available through FORTRAN statements, the following features are available through system subroutines:

- Character input
- Status test
- Plotting

**CTEST Subroutine**

**General form:**

```fortran
CALL CIN (var{,parity})
        variable | any value
        Calls CIN to read a single character from the terminal and stores it in var. If parity is present, the first bit is set to zero.
```

**Examples:**

```fortran
CALL CIN (CHAR)
CALL CIN (X,1)
```

The CIN subroutine accepts a single character from the terminal. The character is stored in the leftmost byte of `var` in 8-bit binary format. If the second argument is present, the leftmost bit of `var` is set to zero. Otherwise, the leftmost bit remains as read from the terminal.

*For example:*

```fortran
80 CALL CIN (CHAR,1)
    IF (COMP(CHAR,#0000,1) .NE. 0) GO TO 80
```

| Hexadecimal code for carriage return |

These statements wait for a carriage return from the terminal before continuing. The COMP function compares character strings as described in section 7.

**CTEST Subroutine**

**General form:**

```fortran
CALL CTEST (status)
        variable
Ubuntu
```

Tests the input status of the terminal.

**Example:**

```fortran
CALL CTEST (ISTAT)
```

The CTEST subroutine determines whether there is a character ready for input from the terminal. The *status* is zero if there is no character and one if there is a character. For example:
10 CALL DELAY(10)  Waits 0.1 second
CALL CTEST(STATUS)  Tests for a character
IF (STATUS .EQ. 0) GO TO 10  Repeats loop if there is no character
CALL CIN (CHAR)  Reads the character

PLOT Subroutine

General form:

CALL PLOT (x, y, 'text', n [,m])  Displays n characters of text m times, starting at coordinates x,y.
variables  variables

Examples:

CALL PLOT(1,1,'\^',1)
CALL PLOT(XCO,YCO,'123',CX,CY)

You can use the PLOT subroutine to draw graphs or figures or display characters at a specific position on the screen.

In a Processor Technology Sol System or a system using the VDM-1 Video Display Module, the screen is represented in the memory map of 8080 computer memory as a 1K block beginning at CO00 hexadecimal. A character placed at address CO00 will appear on the screen in the first character position of the first line.

The x and y coordinates of the PLOT routine address a position on the screen defined by:

\[(x-1) * 64 + y-1 + 0CC00H\]

where x must be between 1 and 16 and y must be between 1 and 64 (the screen has 16 lines and 64 character positions per line).

The PLOT routine does not prevent you from addressing locations outside the screen’s memory. By using values of x and y outside the specified ranges, you can use PLOT to place characters in fixed memory locations. For example, you could use the PLOT subroutine to perform memory map input and output to peripheral devices.

The fourth argument must be present to tell the routine how many characters of text to display. The fifth argument, which tells how many sets of the specified characters to display, is optional.

The first four arguments are required with one exception: you can call PLOT with no arguments to clear the screen.

The Processor Technology VDM-1 module output port is at C8 and the Sol output port is at FE. The VDM-1 output port must be at either 0C8 or 0FE hexadecimal.

Program Example:

CALL PLOT
DO 10 A = 1,39
    B = 16 - A**2/100
    CALL PLOT (B,A,'\*',1)
10 CONTINUE
END

This program plots the curve \[Y = X**2.\]
6.3. PROGRAM TERMINATION
The subroutines described in this unit let you terminate execution without a message, abort execution, or delay execution.

EXIT Subroutine

General form:

```
CALL EXIT           Terminates execution.
```

Example:

```
CALL EXIT
```

The EXIT subroutine terminates execution in the same way as the STOP statement except that CALL EXIT does not display STOP at the terminal.

ABORT Subroutine

General form:

```
CALL ABORT (error)   Aborts execution and displays the specified number error code.
```

Example:

```
CALL ABORT (89)
```

The ABORT subroutine terminates execution using the ABTOP system call and displays an error code on the terminal. All open files are closed and control returns to PTDOS.

DELAY Subroutine

General form:

```
CALL DELAY (time)    Waits time/100 seconds before returning from the variable or constant subroutine.
```

Examples:

```
CALL DELAY (10)
CALL DELAY (TFAC)
```

The DELAY subroutine allows a time delay of 0.01 to 635.36 seconds. The routine waits time/100 seconds before returning from the call. The argument must be between 0 and 63535, causing the following time delays:
time = 1     delay = 0.01 seconds
time = 2     delay = 0.02 seconds
...
time = 63535 delay = 635.35 seconds
time = 0     delay = 635.36 seconds

The timing is hardware dependent so the wait may not be exactly the same on all processors. However, the wait will be in units of exactly .01 seconds on processors with 500 nanosecond CPU clocks.

6.4. PROGRAM LINKING
You can link separately-compiled FORTRAN object files using the CHAIN subroutine. No variable values are preserved between links. You can link any number of executable programs by using the CHAIN subroutine at the end of each program to call the next program.

CHAIN Subroutine

General form:

CALL CHAIN ('program name')  Loads the specified program
| object overwriting the existing
| file name one in memory.

Example:

CALL CHAIN ('PART2')

The CHAIN subroutine loads another program and may overwrite the existing one. The new program must be in object form and have a file type beginning with I (image).

If the specified program does not exist or an error occurs during loading, a FILE OP error occurs.

Program Example:
P1 is the name of the object file of the following program:

TYPE 'THIS IS PART 1'
CALL CHAIN ('P2')
END

P2 is the name of the object file of the following program:

TYPE 'THIS IS PART 2'
END

Execution:

User:       P1 <CR>
First program:  THIS IS PART 1
Second program: THIS IS PART 2
                     STOP END IN - MAIN

If the program specified in the CHAIN subroutine does not have a starting address, the subroutine loads the program and returns control to the statement that follows the call to CHAIN. This allows assembly-language routines to be loaded into memory from a FORTRAN program (see section 8).

6.5. OTHER UTILITY SUBROUTINES
The subroutines discussed in this unit let you address memory directly, convert binary values to decimal values, and set a specific bit in a variable.
MOVE Subroutine

General form:

CALL MOVE (n,loc1,disp1,loc2,disp2)  Moves n bytes from loc1 to loc2. If
disp is positive  loc is a character
expression  expression  string. If disp
character string  character string  is negative loc's
or memory address  or memory address  value is a
memory address.

Examples:

CALL MOVE (6,'ABCDEF',0,$CC00,-1)
CALL MOVE (2,A,-1,$CC00,-1)
CALL MOVE (1024, $CC00, -1, A, -1)

The MOVE subroutine allows direct access to memory for both reading and writing. It moves n bytes from loc1 to loc2. The arguments loc1 and loc2 specify either a memory address to be used or a character string to be moved.

The interpretations of loc1 and loc2 depends on the values of disp1 and disp2, respectively. If disp is negative, loc contains a memory address. If disp is positive, loc is treated as a character string. In the first case, the displacement (disp) is added to loc.

There is no reason to move a value to a character string, so the value of disp2 should be negative.

Explanation of examples:

CALL MOVE (6,'ABCDEF',0,$CC00,-1)  Moves ABCDEF to address CC00 hexadecimal.
CALL MOVE (2,A,-1,$CC00,-1)  Moves 2 bytes from the address stored in A to address CC00 hexadecimal.
CALL MOVE (1024,$CC00,-1,A,-1)  Moves 1024 bytes starting with address $CC00 to the address specified by A.
CBTOF Subroutine

General form:

CALL CBTOF (loc1, displacement, loc2 [, flag])

variable number of variable any nonzero bytes value

Converts the 8-bit or 16-bit binary number located at loc1 + displacement to its decimal equivalent and stores is in loc2. The binary number is a 16-bit number unless the fourth parameter is present.

Examples:

CALL CBTOF (BVAR, 0, DVAR)
CALL CBTOF (ARAY(1,1),6, VAL(2))
CALL CBTOF (X, 1, Y, 1)

The CBTOF subroutine converts a 16-bit or 8-bit unsigned binary number to a decimal floating point value. The number to be converted is located at loc1 + displacement if displacement is positive. If displacement is negative, then loc1 contains (rather than is) the address to be used, and the number to be converted is at the indicated address plus the absolute value of displacement.

If flag is not present, the binary number is treated as a 16-bit value stored in standard 8080 format (two's complement binary). If flag is present, the binary number is treated as an 8-bit value. The converted number is stored in the third argument.

Program Example:

DIMENSION A(4)
CALL FINFO('SUTIL', A)
CALL CBTOF(A(1), 0, ID)
CALL CBTOF(A(1), 6, NBLKS)
CALL CBTOF(A(1), 12, BLKSZ)
WRITE(1, 10) ID, NBLKS, BLKSZ
10 FORMAT('ID = ', I5, ' # BLOCKS = ', I5,
& ' BLOCKSIZE = ', I5)
END

This program retrieves in binary form status information about disk file SUTIL. It converts the information to decimal form and displays it at the terminal.
BIT Subroutine

General form:

CALL BIT (variable, displacement, 'code')

  variable name  constant  S, R, or F
  or variable
  bit displacement

Sets bit 0 + displacement of
variable to 1, 0, or flip.

Examples:

CALL BIT (NEWVAR, 0, 'S')
CALL BIT (COUNT, B1)

The BIT subroutine lets you set the individual bits of a variable. The displacement a bit placement.
A bit can be set to 1 ("S"), reset to 0 ("R"), or flipped to its opposite value ("F").
Recall that the form of a stored value is:

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
\mid \text{sign} & \text{nn} & \text{nn} & \text{nn} & \text{nn} & \text{0s} & \text{ee} & \text{exponent}
\end{array}
\]

where each set of two digits represents a byte. Each byte consists of eight bits. While bytes are numbered from
left to right, the bits within a byte are numbered from right to left. Thus, to set the last bit of a value (the
rightmost bit of byte 5), use a displacement of 40 and to set the first bit of a value (the leftmost bit of
byte 0), use a displacement of 7.

In general, the byte affected by the BIT subroutine is:

\[
\text{variable} + (\text{displacement} / 8)
\]

and the bit within the byte is:

\[
\text{MOD (displacement, 8)}
\]

where bit 0 is the rightmost bit and bit 7 is the leftmost bit of the byte.
SECTION 7
SYSTEM FUNCTIONS

7.0. INTRODUCTION
PTDOS FORTRAN provides general mathematical functions, trigonometric functions, and a string comparison function.

7.1. GENERAL MATHEMATICAL FUNCTIONS
In the table of functions below, \( \text{exp} \), \( \text{exp1} \), and \( \text{exp2} \) represent numeric expressions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{ABS(exp)} )</td>
<td>The real absolute value of ( \text{exp} ).</td>
</tr>
<tr>
<td>( \text{AINT(exp)} )</td>
<td>The truncated value of ( \text{exp} ).</td>
</tr>
<tr>
<td>( \text{ALOG(exp)} )</td>
<td>The natural logarithm of ( \text{exp} ).</td>
</tr>
<tr>
<td>( \text{ALOG10(exp)} )</td>
<td>The logarithm base 10 of ( \text{exp} ).</td>
</tr>
<tr>
<td>( \text{AMAX0(exp1,exp2,...)} )</td>
<td>The largest of the integer values represented (up to 254 values).</td>
</tr>
<tr>
<td>( \text{AMAX1(exp1,exp2,...)} )</td>
<td>The largest of the real values represented (up to 254 values).</td>
</tr>
<tr>
<td>( \text{AMIN0(exp1,exp2,...)} )</td>
<td>The smallest of the integer values represented (up to 254 values).</td>
</tr>
<tr>
<td>( \text{AMIN1(exp1,exp2,...)} )</td>
<td>The smallest of the real values represented (up to 254 values).</td>
</tr>
<tr>
<td>( \text{AMOD(exp1,exp2)} )</td>
<td>The real remainder when ( \text{exp1} ) is divided by ( \text{exp2} ).</td>
</tr>
<tr>
<td>( \text{DIM(exp1,exp2)} )</td>
<td>The positive difference between ( \text{exp1} ) and ( \text{exp2} ).</td>
</tr>
<tr>
<td>( \text{EXP(exp)} )</td>
<td>The constant ( e ) raised to the power ( \text{exp} ).</td>
</tr>
<tr>
<td>( \text{FLOAT(exp)} )</td>
<td>( \text{exp} ) converted to a real number.</td>
</tr>
<tr>
<td>( \text{IABS(exp)} )</td>
<td>The integer absolute value of ( \text{exp} ).</td>
</tr>
<tr>
<td>( \text{IDIM(exp1,exp2)} )</td>
<td>The positive difference between ( \text{exp1} ) and ( \text{exp2} ).</td>
</tr>
<tr>
<td>( \text{IFIX(exp)} )</td>
<td>The truncated value of ( \text{exp} ).</td>
</tr>
<tr>
<td>( \text{INT(EXP)} )</td>
<td>The truncated value of ( \text{exp} ).</td>
</tr>
<tr>
<td>( \text{ISIGN(exp)} )</td>
<td>The sign of ( \text{exp} ); if positive, (-1) if negative, (0) if zero.</td>
</tr>
</tbody>
</table>
MAX0(exp1,exp2,...) The largest of the integer values represented (up to 254 values).

MAX1(exp1,exp2,...) The largest of the real values represented (up to 254 values).

MIN0(exp1,exp2,...) The smallest of the integer values represented (up to 254 values).

MIN1(exp1,exp2,...) The smallest of the real values represented (up to 254 values).

MOD(exp1,exp2) The integer remainder when exp1 is divided by exp2.

RAND(exp) Entry exp in a table of random numbers.

SIGN(exp) The sign of exp; +1 if positive, -1 if negative, 0 if zero.

SQRT(exp) The square root of exp.

Examples:

\[
\text{ANS} = \text{ALOG}(X) \div \text{MOD}(\text{SNUM}, D) \\
\text{IF} (\text{EXP}(\text{PWR}) \cdot \text{GT. RS}) \text{GO TO 100}
\]

All the PTDOS FORTRAN mathematical functions are ANSI standard functions except for RAND. This function behaves as if it were returning an entry from a table of random numbers. RAND's argument determines which entry in the table is returned:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Value returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The next entry in the table</td>
</tr>
<tr>
<td>-1</td>
<td>The first entry in the table. The table pointer is reset to the first entry.</td>
</tr>
<tr>
<td>n</td>
<td>The entry following n.</td>
</tr>
</tbody>
</table>

Although the random numbers generated are between 0 and 1, numbers in any range may be obtained with an appropriate expression. The following statement gives a random number between 0 and 99:

\[
I = \text{RAND}(0) \times 100
\]
7.2. TRIGONOMETRIC FUNCTIONS

General forms:

\[
\begin{align*}
\text{SIN}(\text{exp}) & \quad \text{The sine of exp radians.} \\
\text{COS}(\text{exp}) & \quad \text{The cosine of exp radians.} \\
\text{TAN}(\text{exp}) & \quad \text{The tangent of exp radians.} \\
\text{ATAN}(\text{exp}) & \quad \text{The arctangent of exp; the answer is in radians.} \\
\text{ATAN2}(\text{exp1},\text{exp2}) & \quad \text{The arctangent of exp1/exp2; the answer is in radians.}
\end{align*}
\]

\text{exp} is a numerical expression

Examples:

\[
\begin{align*}
\text{VIN} & = \text{TAN(SA)} \times 2 \\
\text{IF} & (\text{SIN(A)} \cdot \text{EQ.} \cdot \text{TAN(A)}) \quad \text{STOP}
\end{align*}
\]

7.3. COMPARING CHARACTER STRINGS

General form:

\[
\begin{align*}
\text{COMP}('\text{string1}', '\text{string2}', n) & \quad \text{Compares n characters of string1 with string2 and returns +1 if string1 is greater than string2, -1 if string1 is less than string2, and zero if they are equal.}
\end{align*}
\]

integer

Examples:

\[
\begin{align*}
\text{ANS} & = \text{COMP}('\text{WORD}', '\text{AAA}', 3) \\
\text{IF} & (\text{COMP}('\text{CHAR}', '0A00', 1)) \text{ GO TO 10}
\end{align*}
\]

The COMP function compares characters strings character by character, from left to right. The characters are compared according to their ASCII character codes.

The first two arguments of the function can be string constants or the names of variables or arrays containing strings. For example:

\[
\begin{align*}
10 & \text{ CALL CIN(ONECHR)} \\
& \text{ IF} (\text{COMP(ONECHR, '/'}, 1) \cdot \text{NE.} \cdot 0) \text{ GO TO 10} \\
& \text{CALL CIN(DRIVE)}
\end{align*}
\]

You can also specify the hexadecimal code in binary form for a character. This is useful when testing for special characters such as line feed ('0A00) or carriage return ('0D00). For example:

\[
\begin{align*}
10 & \text{ CALL CIN(X)} \\
& \text{ IF} (\text{COMP(X, '\D\'}), 1) \cdot \text{NE.} \cdot 0) \text{ GO TO 10}
\end{align*}
\]

These statements read characters until a carriage return is encountered.
7.4. EXECUTING ASSEMBLY-LANGUAGE PROGRAMS

General form:

\[
\text{CALL(address,argument)} \quad \text{Executes the assembly routine starting at address, passing begin-execution variable the value of argument to it. address of or constant assembly routine}
\]

Examples:

\[
\text{WORD = CALL($7000, STR)} \\
\text{IF (CALL($6500, A(2))) STOP}
\]

The CALL function begins execution of an assembly-language program that is loaded in memory. You can use the CHAIN subroutine to load an assembly routine. For more information see section 8.

The second argument is evaluated, converted to a 16-bit binary number, and then passed to the called routine in both the BC and DE register pairs. The assembly routine places the value to be returned in register pair HL before executing an 8080 RET instruction. Since H and L consist of 16 bits, the value returned is limited to a positive integer between 0 and 65535.

The stack must be maintained because the return address is placed on it when the assembly routine is called.
SECTION 8

ASSEMBLY-LANGUAGE INTERFACE

PTDOS Assembly Language statements can be directly inserted into a FORTRAN program if they are preceded by an asterisk in column 1. The line that contains the asterisk will be directly output to the assembly file without further processing (except that the asterisk is deleted).

When using this feature, always put a FORTRAN CONTINUE statement immediately preceding the first assembly language statement in each group. The CONTINUE statement will ensure that the assembly language statements are inserted at the expected place.

Program Example:

```
J=0
DO 1,I=1,100
  1   J=J
C
C The following is equivalent to the DO loop above,
C but it is partially coded in assembly language.
C
  K=0
  CONTINUE

**
  * LXI H,100
  * SHLD COUNT
  * JMP SKIP
  * COUNT DW 0
  * SKIP EQU $
  **
      K=K
  CONTINUE
C
C The following is the code for the end of the DO loop.
C
  **
  * LHLD COUNT
  * DCX H
  * SHLD COUNT
  * MOV A,H
  * ORA L
  * JNZ SKIP
  **
      TYPE J,K
      END
```

When executed, this program displays the final value for both indices.

You can call assembly-language programs from a FORTRAN program using the CHAIN subroutine to load the assembly routine and the CALL function to execute it. The assembly routine to be executed must be an assembled program on an image file (the file type begins with I).
Program Example:

```
ACCEPT 'TYPE A NUMBER', VAL
CALL CHAIN('DUBL')
Y = CALL($6666,VAL)
TYPE 'THE NUMBER DOUBLED = ', Y
END
```

The value of VAL is passed to the routine at 6666 in both BC and DE register pairs.

where 6666 is the hexadecimal starting location of assembly routine DUBL, shown below:

```
ORG 6666H
DUBL MOV H,D
      MOV L,E       COPY TO HL
      DAD H       DOUBLE IT
      RET

*  
* DOUBLED VALUE RETURNED IN HL
*  
```
APPENDIX 1

FORTRAN STATEMENT SUMMARY

variable = expression
Assigns the value of expression to variable.

ACCEPT input list
Reads values from the terminal and assigns them to names in the input list.

ASSIGN n TO v
Assigns a statement label to a variable in an assigned GO TO statement.

BACKSPACE unit
Positions the file at the beginning of the previous record.

BLOCK DATA
Begins a BLOCK DATA subprogram for initializing COMMON variables.

CALL subroutine name (argument list)
Executes the named subroutine passing values to it through the argument list.

COMMON /label1/list1/label2/list2...
Declares variables and arrays to be shared among routines.

CONTINUE
Causes no action. Usually a dummy statement for transfer of control.

COPY file name
Copies the source file into the current program.

DATA var/const/,, array/list of constants/,, etc.
Initializes variables, arrays or array elements.

DIMENSION var1(n1,n2,...), var2(n1,n2,...),...
Sets aside space for the arrays indicated.

DO n index = v1,v2{,increment}
Executes subsequent statements to statement n as index increases or decreases from v1 to v2.

END
Required as last statement of every routine.

DUMP /ident/ output list
Displays ident followed by items in the output list when a runtime error that is not trapped occurs.

ENDFILE
Writes an end of file at the current read/write position.

ERRSET n, v
Transfers control to statement n if a runtime error occurs. Variable v contains the error code.

ERRCLR
Clears the effect of the ERRSET statement.

FORMAT(field specifications list)
Describes the format of input or output for READ or WRITE statements.

FUNCTION function name(parameter list)
Beginning a function definition.

GO TO n
Transfers control to statement n.

GO TO (n1,n2,...), index
Transfers control to statement n1 if index = 1, transfers to n2 if index = 2, etc.
GO TO v, (n1,n2,...) Transfers control to v, where v must equal one of n1,n2...

IF(exp) n1,n2,n3 Transfers control to n1, n2, or n3 for a minus, zero, or plus value of exp.

IF(exp) statement Executes the statement only if the value of expression exp is true (nonzero).

IMPLICIT type (letter list) Changes the default type of variables beginning with the indicated letters.

INTEGER var1, var2,... Declares that var1, var2, etc. are integer variables.

LOGICAL var1, var2,... Declares that var1, var2, etc. are logical variables.

PAUSE (character string) Interrupts execution until any key is typed, displaying the word PAUSE and the string.

PAUSE n Interrupts execution until any key is typed, displaying the word PAUSE and the integer n.

READ(unit, format{,end-of-file, error}) input list Reads values from a file and assigns them to names in the input list.

REAL var1, var2,... Declares that var1, var2, etc. are real variables.

RETURN Returns control from a function or subroutine.

REWIND Positions the file to byte 0.

STOP (character string) Terminates execution, displaying the string.

STOP n Terminates execution, displaying the integer n.

SUBROUTINE subroutine name ((parameter list)) Begins a subroutine definition.

TYPE output list Displays values from the output list at the terminal.

WRITE(unit, format {, end-of-file, error}) output list Writes values from the output list on the specified file.
APPENDIX  2

SUMMARY OF SYSTEM SUBROUTINES

CALL ABORT(error code)  Terminates execution and displays the error code.
CALL BIT(variable,disp,code)  Sets, resets, or flips bit 0 + disp of variable.
CALL CBTOF(loc1,displacement,loc2{,flag})  Converts a binary number to its decimal equivalent.
CALL CHAIN('program name')  Links to another program.
CALL CHATTR('file name',attribute codes)  Changes the attributes of the specified file.
CALL CHNAME('old name', 'new name')  Changes the file name from old name to new name.
CALL CHTYPE('file name','type')  Changes the type of the specified file.
CALL CIN(var{,parity})  Reads a single character from the terminal.
CALL CLOSE(unit)  Closes the specified file.
CALL CONTRL(unit,op,DEin,HLin,Aout,DEout,HLout)  Allows program control over devices or gets information.
CALL CREATE('file name','type', block size)  Creates a PTDOS file.
CALL CTEST(status)  Determines the input/output status of the terminal.
CALL DELAY(time)  Waits time/100 seconds before returning from the call.
CALL EXIT  Terminates execution.
CALL FINFO('file name', array name)  Retrieves information about the specified file.
CALL KILL('file name')  Deletes the specified file.
CALL MOVE(n, loc1, disp1, loc2, disp2)  Moves n bytes from loc1 to loc2.
CALL OPEN(unit, 'file name'{, buffer})  Opens the specified file for input or output.
CALL PLOT(x, y, 'text', n{,m})  Plots characters on the VDM screen.
CALL RANDOM(unit)  Allows random access to the specified file.
CALL SEEK(unit,location{,blk})  Positions a random file to a specified byte or block.
CALL SETUNT(drive)  Changes the system default drive.
CALL SPACE(unit, displacement,'direction')  Space forward, backward, or to the end of the file.
APPENDIX 3

TABLE OF FUNCTIONS

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Definition</th>
<th>Number of Arguments</th>
<th>Type of Arguments</th>
<th>Type of Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>(</td>
<td>a</td>
<td>)</td>
<td>1</td>
</tr>
<tr>
<td>AINT</td>
<td>( \text{sign}(a) \times</td>
<td>x</td>
<td>\times a ) where ( x ) is the largest integer ( \leq a )</td>
<td>1</td>
</tr>
<tr>
<td>ALOG</td>
<td>Natural log</td>
<td>1</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>ALOG10</td>
<td>Log base 10</td>
<td>1</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>AMAX0</td>
<td>Maximum</td>
<td>Up to 254</td>
<td>Integer</td>
<td>Real</td>
</tr>
<tr>
<td>AMAX1</td>
<td>Maximum</td>
<td>Up to 254</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>AMIN0</td>
<td>Minimum</td>
<td>Up to 254</td>
<td>Integer</td>
<td>Real</td>
</tr>
<tr>
<td>AMIN1</td>
<td>Minimum</td>
<td>Up to 254</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>AMOD</td>
<td>( a1 \times \text{sign}(x) \times</td>
<td>x</td>
<td>\times a ) where ( x = a1/a2 )</td>
<td>2</td>
</tr>
<tr>
<td>ATAN</td>
<td>( x = \text{arctan}(a) \ - \pi/2 \leq x \leq \pi/2 )</td>
<td>1</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>ATAN2</td>
<td>( x = \text{arctan}(a1/a2) \ - \pi/2 \leq x \leq \pi/2 )</td>
<td>2</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>CALL</td>
<td>Execute assembly routine</td>
<td>2</td>
<td>Either</td>
<td>Real</td>
</tr>
<tr>
<td>COMP</td>
<td>Compare strings</td>
<td>3</td>
<td>Either</td>
<td>Real</td>
</tr>
<tr>
<td>COS</td>
<td>( \cos(a) ) a in radians</td>
<td>1</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>DIM</td>
<td>( a1 \times \text{min}(a1, a2) )</td>
<td>2</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>EXP</td>
<td>e to the power exp</td>
<td>1</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>FLOAT</td>
<td>Make real</td>
<td>1</td>
<td>Integer</td>
<td>Real</td>
</tr>
<tr>
<td>IBABS</td>
<td>Absolute value</td>
<td>1</td>
<td>Integer</td>
<td>Integer</td>
</tr>
<tr>
<td>IDIM</td>
<td>( a1 \times \text{min}(a1, a2) )</td>
<td>2</td>
<td>Integer</td>
<td>Integer</td>
</tr>
<tr>
<td>IFIX</td>
<td>Truncate</td>
<td>1</td>
<td>Real</td>
<td>Integer</td>
</tr>
<tr>
<td>INT</td>
<td>( \text{sign}(a) \times</td>
<td>x</td>
<td>) where ( x ) = largest integer ( \leq a )</td>
<td>1</td>
</tr>
<tr>
<td>ISIGN</td>
<td>Sign</td>
<td>1</td>
<td>Integer</td>
<td>(-1, 0, +1)</td>
</tr>
<tr>
<td>MAX0</td>
<td>Maximum</td>
<td>Up to 254</td>
<td>Integer</td>
<td>Integer</td>
</tr>
<tr>
<td>MAX1</td>
<td>Maximum</td>
<td>Up to 254</td>
<td>Real</td>
<td>Integer</td>
</tr>
<tr>
<td>MIN0</td>
<td>Minimum</td>
<td>Up to 254</td>
<td>Integer</td>
<td>Integer</td>
</tr>
<tr>
<td>MIN1</td>
<td>Minimum</td>
<td>Up to 254</td>
<td>Integer</td>
<td>Integer</td>
</tr>
<tr>
<td>MOD</td>
<td>( a1 \times \text{sign}(x) \times</td>
<td>x</td>
<td>\times a ) where ( x = a1/a2 )</td>
<td>2</td>
</tr>
<tr>
<td>RAND</td>
<td>Random number</td>
<td>1</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>SIGN</td>
<td>Sign</td>
<td>1</td>
<td>Real</td>
<td>(-1., 0., +1.)</td>
</tr>
<tr>
<td>SIN</td>
<td>( \sin(a) ) a in radians</td>
<td>1</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>SQRTE</td>
<td>Square Root</td>
<td>1</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>TAN</td>
<td>( \tan(a) ) a in radians</td>
<td>1</td>
<td>Real</td>
<td>Real</td>
</tr>
</tbody>
</table>

where:
- \( a \) = argument
- \( a1 \) = first argument
- \( a2 \) = second argument
APPENDIX 4

DEVICE INTERFACE

If you want to interface with data input or output devices such as tape units or line printers, you can use the PTDOS Device File feature described in section 3 of the PTDOS User’s Guide.

The device file must have the format described and perform all the necessary port assignments, bit assignments, etc. You can then access the device from a FORTRAN program by opening the device file and performing input or output. For example, the following statements read data from tape using device file CTAPE2, which is listed in Appendix C of the PTDOS User’s Guide as an example of a device file:

```
DIMENSION A(100)
CALL OPEN(2,'CTAPE2')
READ(2,*)(A(I),I=1,100)
```
APPENDIX 5

COMPILATION ERROR MESSAGES

Errors that occur during compilation are indicated in the list file by either an error code or an error message. Error codes are listed if you specify G as an option in the OPTIONS declaration; otherwise error messages are listed. The following are PTOS FORTRAN compilation error codes and messages:

00  *FATAL* compiler error
01  Syntax error, 2 operators in a row
02  unexpected continuation
03  input buffer overflow
04  invalid character for FORTRAN statement
05  unmatched parenthesis
06  statement label > 99999
07  invalid character in label field
08  invalid HEX digit in constant
09  expected constant or variable not found
0A  8 bit overflow in constant
0B  unidentifiable statement
0C  statement not implemented
0D  missing quote
0E  SUBROUTINE/FUNCTION/BLOCK DATA not first in routine
0F  columns 1-5 of continuation not blank
10  invalid label
11  RETURN not valid in main program
12  syntax error on unit specification
13  missing comma after ) in COMPUTED GO TO
14  missing variable in COMPUTED GO TO
15  invalid variable in assigned GO TO
16  invalid LITERAL, missing quote
17  number of subscripts declared exceeds max of 7
18  invalid SUBROUTINE or FUNCTION name
19  subscript not POSITIVE INTEGER CONSTANT
1A  FUNCTION requires at least one argument
1B  syntax error
1C  invalid argument in SUBROUTINE or FUNCTION call
1D  first character of variable not alphabetic
1E  ASSIGNED/COMPUTED GO TO variable not type integer
1F  label already defined
20  specification of array must be integer
21  invalid variable name in type specification
22  invalid DIMENSION specification
23  dimension specification not integer or is negative
24  variable already appeared in type statement
25  invalid subroutine name in CALL
26  SUBPROGRAM arg. can’t be initialized
27  improperly nested DO loops
28  unit not integer constant or variable
29  Array size exceed 5461 elements
2A  invalid use of unary operator
2B  variable DIMENSION not valid in main program
variable dimensioned array must be argument
DO, END or LOGICAL IF cannot follow LOGICAL IF
undefined label
WARNING: unreferenced label
FUNCTION or ARRAY missing left parenthesis
invalid argument of FUNCTION or ARRAY
DIMENSION spec. must appear before executable stmts
unexpected character in expression
unrecognized logical opcode
param count error on built-in FUNCTION or ARRAY
*COMPILER ERROR * popped off bottom of operand stack
expecting end of statement, not found
statement too complex, increase P= and/or O= tables
invalid delimiter in ARITHMETIC IF
invalid statement number in IF
HEX constant > FFFF (HEX)
replacement not allowed within IF
multiple assignment statement not implemented
subscripted-subscripts not allowed
subscript stack overflow, increase P= or O=
missing left (in READ/WRITE
invalid unit specified
invalid FORMAT number
invalid element in I/O list
built-in function invalid in I/O list
cannot subscript a constant
variable not dimensioned
invalid subscript
missing comma
index in IMPLIED DO must be a variable
invalid starting value for IMPLIED DO
ending value of IMPLIED DO invalid
increment of IMPLIED DO invalid
illegal use of built-in function
variable cannot be dimensioned in this context
invalid EOF or ERROR exit label
invalid constant
exponent overflow in constant
invalid exponent
character after . invalid
integer overflow
integer underflow (too small)
missing = in DO
string constant not allowed
invalid variable in DATA list
DATA symbol not used in program, line
invalid constant in DATA list
error in DATA list specification
built-in function in DATA list
no filename specified on COPY
runtime format not array name
dump label invalid or more than 10 characters
more than 1 IMPLICIT is invalid
IMPLICIT not first in main, 2nd in subprogram
data type not REAL, INTEGER, or LOGICAL
illegal IMPLICIT specification
improper character sequence in IMPLICIT
variable already dimensioned
Q option must be specified for ERRSET/ERRCLR
HEX constant of zero (0) invalid in I/O stmt
argument cannot be in COMMON
illegal COMMON block name
variable already in COMMON
array specification must preceed COMMON
executable statement is invalid in BLOCK DATA
HEX constant of 27H (') invalid in FORMAT
Invalid number following STOP or PAUSE
NOT USED
invalid label in assigned GO TO
invalid variable in assigned GO TO
THROUGH 80 ARE NOT USED
*FATAL* missing $OPTIONS statement
*FATAL* missing = in $OPTIONS statement
*FATAL* invalid digit in number in $OPTIONS
*FATAL* value exceeds 255 in $OPTIONS
*FATAL* COMMON table overflow, increase C=
*FATAL* unknown option (letter before =)
*FATAL* missing END statement
*FATAL* LABEL TABLE overflow, increase L=
*FATAL* SYMBOL TABLE overflow, increase S=
*FATAL* ARRAY STACK overflow, increase A=
*FATAL* DO LOOP STACK overflow, increase D=
*FATAL* stack overflow
*FATAL* stack overflow
*FATAL* internal stacks exceed available memory
*FATAL* MEMORY ERROR (address in HL of ABORT)
*FATAL* OPEN error on COPY file
*FATAL* too many routines to compile (> 62)
*FATAL* no more room to store DATA statements
APPENDIX 6

EXECUTION ERRORS

Execution, or runtime, errors may occur during execution of a program that compiled correctly, causing termination of that program. An execution error causes an error message to be displayed. Execution error messages have the form:

RUNTIME ERROR error message, CALLED FROM LOC. location
PGM WAS EXECUTION LINE line number IN ROUTINE routine name

where:
error message is one of the messages shown below.
location is the memory location of the error-producing call to the runtime package.
line number specifies the line in which the error occurred if the X option was selected in the OPTIONS declaration; otherwise the line number is ?? . The line numbers are shown on the source listing from the compiler (if present, the OPTIONS declaration is line 1).
routine name is the name of the routine in which the error occurred.

If several line numbers are listed in the error message, the error actually occurred in the first line specified.
The error messages that may occur during execution are:

<table>
<thead>
<tr>
<th>Message</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| /0      | DIVIDE BY ZERO  
Attempted to divide by zero. |
| ASN GOTO| ASSIGNED GO TO ERROR  
Address assigned not in list behind it. |
| CALL POP| CALL STACK POP ERROR  
A RETURN that does not have a corresponding CALL or FUNCTION reference has been executed. This error is usually caused by user assembly language programs. |
| CALL PSH| CALL STACK PUSH ERROR  
Caused by more than 62 recursive subprogram calls. Avoid having a subprogram call itself or call a routine that called it. |
| CHAN OPN| UNIT ALREADY OPEN  
Attempted to open a file on a FORTRAN unit that is already open. |
| COM GOTO| COMPUTED GO TO INDEX OUT OF RANGE  
The variable specified in a computed GO TO is either less than one or greater than the number of statement labels specified. |
| CONVERT | 16 BIT CONVERSION ERROR  
An overflow occurred while converting a number to internal 16-bit binary form. This error can occur when converting the unit number in input/output statements, evaluating subscripts, or converting floating-point numbers to 16-bit binary form. |
| FILE OP | FILE OPERATION ERROR  
An error from any of the system subroutines that deal with PTDOS |
files generates this message. The error could result from such things as an invalid file name, spacing beyond the beginning or end of a file, etc. The PTDOS UTIL function supplies a detailed explanation of the error and lists the PTDOS operation that was in error. The UTIL routine destroys part of the FORTRAN runtime package (that part located in CXBUF). If you want to rerun your program after executing UTIL, you must first reload the program from disk.

FRMT ERR
FORMAT ERROR
A formatted READ or WRITE referred to an invalid FORMAT specification.

I/O ERR
I/O ERROR
An error occurred during a READ or WRITE operation. This message is generated if there is no error label specified in the statement. In addition, a READ statement with no end-of-file label generates this message if an end of file is encountered. The PTDOS UTIL function supplies a detailed explanation of the error.

I/O LIST
INVALID I/O LIST
A formatted READ or WRITE statement has an error in the input or output list. This error only occurs when a user assembly program does not construct the input or output list correctly. It never occurs from FORTRAN-generated code.

ILL CHAN
ILLEGAL UNIT NUMBER
A unit number that is less than 2 or greater than 15 has been passed to one of the input/output routines.

INP ERR
INPUT ERROR
An invalid character has been encountered while reading a number. Possible causes are two decimal points in a number, an E in an F-type field, a decimal point in an I-type field, etc.

INT-OVER
INTEGER OVERFLOW
An integer value has more than eight digits.

LNE LENG
LINE LENGTH ERROR
Attempted to read or write a record more than 250 characters long. This count includes a carriage return at the end of a line and (for ALS8 files) the byte count.

LOG(-#)
LOG OF NEGATIVE NUMBER
 ALOG or ALOG10 was called with a negative argument.

NOT OPEN
UNIT NOT OPEN
Attempted to read, write, rewind, or perform some operation on a FORTRAN unit number that is not open.

OPEN ERR
OPEN ERROR
An error occurred during execution of an OPEN statement. This message encompasses all open errors except for a nonexistent file. For information about which open error occurred, use the PTDOS UTIL function.

OVERFLOW
FLOATING POINT OVERFLOW
The result of a floating-point operation resulted in a number too large to be stored.

PARM CNT
PARAMETER COUNT ERROR
A subprogram call had too many or too few arguments.
<table>
<thead>
<tr>
<th>SET UNIT</th>
<th>SET UNIT ERROR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>An error occurred while changing the default unit (drive). Use the UTIL function for more information.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SQRT(-#)</th>
<th>SQRT OF NEGATIVE NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The argument of the square root function is negative.</td>
</tr>
</tbody>
</table>
APPENDIX 7

COMPARISON OF PTDOS AND ANSI STANDARD FORTRAN

The PTDOS FORTRAN language includes the following extensions to version X3.9-1966 of ANSI Standard FORTRAN:

1. Free-format input and output.
2. An IMPPLICIT statement for the implicit typing of variables and functions.
3. Character string data type and a string comparison function.
4. Optional end-of-file and error branches in READ and WRITE statements.
5. A COPY statement to copy files of source statements into a FORTRAN source program.
6. Assembly-language interface. Assembly-language statements can be included in the source file and assembly routines can be called from the FORTRAN program.
7. File management from the FORTRAN program, including creating, killing, and changing attributes.
8. Random access to data files.
9. Input from and output to device files.
10. Direct control over the video display.
11. Access to absolute memory locations, including individual bits.
12. Program-controlled time delay.
13. A pseudo-random number generator function.
14. Program control of runtime error trapping.
15. Ability to chain a sequence of programs.

PTDOS FORTRAN does not have the following features of ANSI standard FORTRAN:

1. Double precision, including double-precision functions, statements and format specifications.
2. Complex numbers, including complex statements and functions.
3. EQUIVALENCE
4. Extended DATA statement. ANSI FORTRAN allows DATA statements such as:

   DATA X, Y, Z/10, 20, 30/

In PTDOS FORTRAN this statement would have to be changed to:

   DATA X/10/, Y/20/, Z/30/

5. Hollerith field specifications are not available in FORMAT or DATA statements. They must be replaced with character strings.
6. The following format specifications are not available: carriage control characters, D, G, H, and P.
7. Statement functions are not available.
8. Only the first five characters of function or subroutine names or COMMON labels are retained. For example, PTDOS FORTRAN does not differentiate between MYSUB1 and MYSUB2.
9. The following cannot be used for subroutine, function, or COMMON names: A, B, C, D, E, H, L, M, SP, PSW, or any PTDOS reserved name contained in the file PTDEFS.
APPENDIX 8

BIBLIOGRAPHY

1. *Computer Science: Fortran Language Programming*  
   Forsythe, Keenan, Organick, and Stenberg  
   John Wiley & Sons, Inc. 1970

2. *FORTRAN IV, Second Edition*  
   Elliott I. Organick and Loren P. Meissner  
   Addison-Wesley Publishing Co. 1966

3. *FORTRAN IV with WATFOR and WATFIV*  
   Cress, Dirksen, and Graham  
   Prentice-Hall, Inc. 1970

   Donald E. Knuth  
   Addison-Wesley Publishing Co. 1973

5. *A Guide to FORTRAN Programming*  
   Daniel D. McCracken  
   John Wiley & Sons, Inc. 1961

6. *History and Fundamentals of Programming Languages*  
   Jean E. Sammet

7. *Programming Proverbs for FORTRAN Programmers*  
   Henry F. Ledgard  
   Hayden 1975

8. *Standard FORTRAN: A Problem-Solving Approach*  
   Laura Cooper and Marilyn Smith  
   Houghton Mifflin Company 1973

9. *The Elements of Programming Style*  
   Brian W. Kernighan and P. J. Plauger  
   McGraw-Hill Book Co. 1974

10. *Software Tools*  
    Brian W. Kernighan and P. J. Plauger  
    Addison-Wesley Publishing Co. 1976
Extended Disk FORTRAN Update

Subject: Additional Programs on FORTRAN Diskette

The FORTRAN diskette which this update accompanies includes six files not described in the Extended Disk FORTRAN manual: ASSM, CLOCK.F, DIGIT, STAT.F, CMPLX.F, and SOLGO which is information-protected.

The file ASSM is an updated version of the disk assembler that must be used with the FORTRAN compiler. Since it is appreciably faster than the previous version, you will probably want to use it for any future assembly language programming as well. The ASSM command syntax has not changed, and the ASSM Subsystem Manual applies without update.

With a PTDOS system diskette in unit 0 (make sure you have a backup) and the FORTRAN data diskette in unit 1, the old version may be replaced with the PTDOS command line:

*REATR ASSM; GET /0, I=-1, ASSM, S=-1; REATR ASSM,KWINE <CR>

The asterisk * is the prompt provided by PTDOS. <CR> represents the typing of the CARRIAGE RETURN key.

The file CLOCK.F is the FORTRAN source code for a digital clock demonstration program that uses the PLOT subroutine to directly place a clock display on the video screen. The display is updated every five seconds. Before creating an executable image file, which will be called CLOCK, you must move the data file DIGIT to unit 0 where CLOCK expects to read it. With the FORTRAN diskette in unit 1 and a PTDOS system diskette in unit 0, use the following PTDOS command:

*COPY DIGIT/1, DIGIT <CR>

Now CLOCK.F may be compiled and run with the PTDOS command line:

*FORTRAN CLOCK.F/1,,CLOCK; CLOCK <CR>

The above command line creates an image file CLOCK on the default unit (normally unit 0) from the FORTRAN source file CLOCK.F on unit 1. CLOCK may be rerun later by merely typing its name:

*CLOCK <CR>
When CLOCK begins executing, it will ask whether 12 hour or 24 hour format is desired and what the starting time is to be. A starting time 30 to 60 seconds in the future should be specified to the nearest ten seconds; the seconds unit's position must be 0. When an accurate time reference reaches the specified starting time, type any key. CLOCK will begin running at the specified time. Pressing the MODE key or the CONTROL and 0 keys together will abort CLOCK and return control to PTDOS.

The value of IFACT, initialized near the beginning of the program, is used to control the clock's timing. It may have to adjusted slightly to regulate the speed of the clock. Increase IFACT if the clock runs fast; decrease it if the clock runs slow.

The file STAT.F is the FORTRAN source code for a demonstration program that performs a simple statistical analysis of data contained in a user-specified file. It may be compiled and run by typing the PTDOS command line:

*FORTRAN STAT.F/1,,;STAT; STAT <CR>

and rerun later by typing the command:

*STAT <CR>

When STAT begins execution, it will ask for the name of a data file to analyze and whether or not the user desires an ordered listing of the data on that file. It then produces a self-explanatory statistical summary on a file chosen by the user. The data file read by STAT is simply a collection of at least 10 and no more than 3000 numeric values separated by carriage returns. These values may be integer, floating point, or exponential values since they are read by a free-format READ statement. See Section 5.5.1. of the manual before creating a data file.

The file CMPLX is the FORTRAN source code for a set of subroutines that perform complex addition, subtraction, multiplication, division, absolute value, square root, and can also be used for conversion between rectangular and polar coordinates. They may be included in a user-written program by means of a FORTRAN COPY statement.

The last file, SOLGO, is an information-protected bonus program. Its purpose will be revealed in the future. Do not KILL it, but do not ask about it yet.
FORTRAN UPDATE

SUBJECTS:

Distinction between "terminal" and "console" in Extended Disk FORTRAN
Errata and Addenda to Extended Disk FORTRAN User's Manual, Revised
Descriptions of Subroutines, Appendices 6 and 7.

MANUALS AFFECTED:


CURRENT PUBLICATIONS:

731040

With the information contained in this update, the Extended Disk
(Without the update, the manual describes Release 1.0.)

The modifications described herein are of several sorts. Some are
errata to the existing documentation; others, addenda reflecting
improvements that have generated Release 1.1 of FORTRAN from
Release 1.0. Still others are elaborations whose intent is to
clarify certain aspects of FORTRAN not treated at length, or not given

Short revisions have been keyed to the pages in the manual where they
should appear. The errata and addenda in the second part of the
update are of this type. Longer revisions have not been keyed to
pages; for the most part, they replace recognizable counterparts in
the existing manual (e.g., the new description of a subroutine
replaces the description of that subroutine in the manual). The
treatment of "terminal" and "console," below, is a supplement to the
manual, rather than a revision of existing material.
DISTINCTION BETWEEN "TERMINAL" AND "CONSOLE" IN EXTENDED DISK FORTRAN

The word "terminal," as used in the FORTRAN User's Manual, does not have exactly the same meaning as the word "console." The terminal comprises logical units 0 and 1, which may or may not correspond to the console input and output devices, respectively.

FORTRAN logical units 0 and 1 are permanently associated with the PTDOS Command Interpreter (CI) input and output files. (See the PTDOS User's Manual.) When the system is bootstrapped, the CI input file is file #0, the console input device, and the CI output file is file #1, the console output device. If nothing is done to change those settings, the FORTRAN ACCEPT and READ(0,...) statements will refer to the console input file, and the TYPE and WRITE(1,...) statements will refer to the console output file. In this case, the "terminal" will actually correspond to the "console."

In other cases, the CI input and output files are not associated with PTDOS files #0 and #1, i.e., the "terminal" does not actually correspond to the "console." The PTDOS SETIN and SETOUT commands explicitly change the assignment of CI input and output files; the CI input file assignment may be changed implicitly and temporarily by the DO command macro preprocessor.

If the CI input file assignment has been changed, the FORTRAN ACCEPT and READ(0,...) statements no longer refer to the console input device. Likewise, if the CI output file assignment has been changed, the FORTRAN TYPE and WRITE(1,...) statements no longer refer to the console output device. For example, if a FORTRAN program is executed as part of a command file (which automatically becomes the CI input file), an ACCEPT statement in that program will expect its input not from the console, but from the command file. Similarly, if CI output has been directed to a disk file, TYPE will send its output to that disk file, rather than to the console.

Unlike the TYPE and ACCEPT statements, READ and WRITE may use the console input and output files, respectively, even if logical units 0 and 1 are not currently associated with those files. The special file names $CONIN and $CONOUT, recognized by FORTRAN, always refer to the console, PTDOS files #0 and #1. If you wish to write a FORTRAN program that will take its input from the console, even if the CI input file has been changed, use the OPEN statement to associate a logical unit (other than 0 or 1) with $CONIN, and READ from that unit. If you want to guarantee that the output from a program will go to the console, even if CI output has been redirected, use the OPEN statement to associate a logical unit (other than 0 or 1) with $CONOUT, and WRITE to that unit. (The reason not to use 0 or 1 is that those unit numbers will already be associated with the current CI input and output files.)
ERRATA AND ADDENDA TO EXTENDED DISK FORTRAN USER'S MANUAL

The following changes should be made on the specified pages of the manual.

p 1-3
Add the following to the list of files required to use PTDOS FORTRAN:

RFORTGO - FORTGO ORGed above D000H
FORTDEFR - definitions for use with RFORTGO

Add FORTDEFR to the list of files residing on the default unit. Add RFORTGO to the list of files residing on unit 0.

p 2-1
Replace the text of Section 2.1.2 with the following:

If a statement is too long to fit on a single line, it may be continued on one or more additional lines. Each continuation line must have a character other than blank or 0 in column 6 and blanks in columns 1-5.

If a character string is continued, its interpretation depends on whether or not the P=n parameter was specified on the FORTRAN command line. If this parameter is omitted, the statement lines are not padded with blanks between the final carriage return and column 72. For example, the statement

    TYPE 'THIS STATEMENT IS CONTINUED<CR>
   * BELOW'

outputs the line

    THIS STATEMENT IS CONTINUED BELOW

to the terminal.

If P=72 appeared in the FORTRAN command line, the line output by the above example would have 24 blanks separating the last two words. (If P=64 appeared, there would be 16 blanks.)

p 2-3
Add the following at the bottom of the page:

It is also possible to specify a string constant in Hollerith format, i.e., by preceding the string with a decimal number and an H. The number specifies the length of the string that follows the H. No terminal delimiter is used. For example:

    28HTHIS IS A HOLLERITH CONSTANT
In the fourth paragraph of Section 2.2.2, delete the phrase "but you can only assign up to four characters to an integer variable using an assignment statement."

Replace the last paragraph with the following:

A variable may not have the same name as one of the system functions listed in Appendix 3. In addition, COPY may not be used as a variable name.

Add the following to the first paragraph of Section 3.2:

The source program may be composed of no more than 62 routines, including the main program. System subroutines and functions referenced by the program are not included in this count.

Add the following to the second paragraph:

The $ preceding an OPTIONS declaration is optional.

Replace the description of the X option with the following:

Causes line numbers to be listed during execution tracing or when a runtime error occurs.

Add S=R and P=n to the list of options allowed in the FORTRAN command line.

Add the following to Section 4.1.1:

S=R

The S=R parameter specifies that the runtime package used by the quick-compile option will be loaded above D000H.

P=n

The P=n parameter specifies that input lines will be blank-padded out to column number n. (A carriage return will be converted to a blank also.) The only values allowed for n are 64 and 72.
Add SET UNIT to the list of runtime errors that may be caused by PTDOS operations.

Replace the second paragraph of Section 4.2.2 with:

PTDOS extends downward in memory from BFFFH to 9000H or below, depending on the size of the system-managed buffer area. The SOLOS ROM and scratchpad memory occupy the space from C000H to CBFFH, and the video display memory (Sol or VDM) extends from CC00H to CFFFH. (See the PTDOS User's Manual for further information on memory management.) A compiled FORTRAN program normally resides in user memory from 100H to the bottom of the PTDOS buffer area (e.g., 9000H). However, in a system with 64K of memory, the 12K extending from D001H to FFFFH is also available as user memory.

Replace the last paragraph of the description of assignment statements with the following:

A character string with as many as six characters may be assigned to a variable or array element.

Replace the list of codes and messages with the following:

The error codes and corresponding runtime error messages are:

1 INTEGER OUT OF RANGE
2 16 BIT CONVERSION ERROR
3 ARGUMENT COUNT ERROR
4 COMPUTED GOTO INDEX OUT OF RANGE
5 FLOATING POINT OVERFLOW
6 DIVIDE BY ZERO
7 SQRT OF NEGATIVE NUMBER
8 LOG OF NEGATIVE NUMBER
9 CALL STACK PUSH ERROR
10 CALL STACK POP ERROR
11 FILE OPERATION ERROR
12 ILLEGAL LOGICAL UNIT NUMBER
13 LOGICAL UNIT ALREADY OPEN
14 OPEN ERROR
15 LOGICAL UNIT NOT OPEN
16 SET UNIT ERROR
17 LINE LENGTH ERROR
18 INVALID FORMAT
19 I/O ERROR
20 INPUT ERROR
21 INVALID I/O LIST
22 ASSIGNED GOTO LABEL NOT IN LIST

The runtime error messages are explained in Appendix 6.
Replace the second sentence of the section entitled "Field Specifications: string" with the following:

Strings appearing in FORMAT statements may be delimited by single quotes or expressed in Hollerith format, i.e., with a preceding length count and H.

Add the following:

Field Specifications: Tw

The T tabs within the current record: i.e., the pointer specifying the next character to be input or output is positioned at character w of the input or output record. (The first character is numbered 1.)

For example:

    READ (0,10) I,J,K,I2
    10 FORMAT (3I1,T1,I1)

In this example, the first, second, and third digits are read into variables I, J, and K, respectively. Then the pointer is moved back to character position 1 in the input line, so that the first digit is read again, this time into variable I2.

The second sentence in the first paragraph of Section 5.5.3 should end:

"...associate a logical unit number with the file name."

The following should be added to the end of that paragraph:

Logical unit numbers may range from 0 to 63. Logical units 0 and 1 refer to the PTDOS Command Interpreter input and output files, respectively.

The section entitled "File Unit Numbers" should be deleted.

Add the following to Section 5.5.5:

Unlike the formatted READ and WRITE statements, binary READ and WRITE statements cannot have empty input and output lists. (Such statements would have no effect.)
Add the following section:

5.5.6 DECODE and ENCODE Statements

The FORTRAN WRITE statement, discussed above, is a way of converting variables of different types and lengths, as well as literals, into a character string that appears either on an output device or in an output file. Conversely, the READ statement interprets a character string existing in an input file (or device), so that the components of that string may be manipulated separately within the FORTRAN program. PTDOS FORTRAN offers two statements whose operation is similar to that of WRITE and READ, except that the character string is situated in memory, rather than on an external device or file.

DECODE is like READ; it translates a character string into a series of values and assigns those values to items in an input list. It differs from READ in that the record being "decoded" is a string in memory, rather than a string typed on the console or read from another file.

ENCODE is like WRITE; it builds a character string from a series of values whose names are given in an output list. It differs from WRITE in that the record being "encoded" is put into a string in memory, rather than written on the display device or another file.

**DECODE Statement**

General form:

```
DECODE(string,length,format) input list
```

Under control of the specified format, interprets the character string in string, and assigns values to variables in the input list.

- **string** is a variable name or array name (not an array element) specifying the beginning address of the character string;
- **length** is a number or variable specifying the length of the string in bytes, or the length of each record, if the format prescribes multiple records;
- **format** is a format number or the name of a variable or array containing the format to be used, or an * to indicate free format; and
- **input list** is a list of the variables into which the values derived from the character string will be placed.

Format and input list are subject to the same restrictions as in the READ statement.

DECODE is similar to READ. The character string beginning at location string and continuing for length bytes is read into the variables in the input list, according to the prescribed format. For example, the sequence of statements
C DECODE DATE INTO MONTH, DAY AND YEAR SO THAT JULIAN DATE MAY BE
C CALCULATED. EVERY TWO MEMBERS OF DATE CONTAIN A STRING OF THE FORM
C MM/DD/YY.

DIMENSION DATE(2)
DECODE(DATE,8,2000) MONTH, IDAY, IYR
2000 FORMAT(I2,1X,I2,1X,I2)

will result in the assignment of the appropriate two-digit integers
to the items in the input list. For example, if the string beginning
at DATE(1) consists of the characters 12/08/75, MONTH will receive a
value of 12, IDAY will receive a value of 8, and IYR will receive a
value of 75. (The slashes will not be assigned, because the 1X's in
the format cause them to be ignored.)

It is possible to DECODE a character string consisting of multiple
records, if the format contains slashes (/). In that case, a slash
indicates that the next item in the list should be read from the next
record of the string, i.e., from the next group of length bytes in
the string. For example, the string

'ABCDEFGHIJ'

might be decoded as a single record of length 10, or as multiple
records, e.g., two records of length 5.

If a record in string is not as long as the format and input list
would suggest - that is, if length is less than the number of
characters required to satisfy the list - the rest of the input list
will be filled as though there were additional blanks in the input
record: string variables will be blank-filled, and numeric variables
will be zero-filled. If a record is longer than the format and
input list would suggest - that is, if length is greater than the
number of characters required to satisfy the list, the list will be
satisfied, and the rest of the record will be ignored.

ENCODE Statement

General form:

ENCODE(string,length,format) output list

Under control of the specified format,
constructs a string consisting of the
values named in the output list, and
places that string in string.
where string is a variable name or array name (not an array element) specifying the beginning address of the character string;
length is a number or variable specifying the length of the string in bytes, or the length of each record, if the format prescribes multiple records;
format is a format number or the name of a variable or array containing the format to be used, or an * to indicate free format and
output list is a list of the variables whose values will be used to construct the character string.

Format and output list are subject to the same restrictions as in the WRITE statement.

The ENCODE statement is similar to WRITE. The values corresponding to the items in the output list are written, in order and according to the prescribed format, to memory locations beginning at string. For example, after the sequence of statements

```
C ENCODE A SUBTITLE IN ARRAY SBTITL
C
DIMENSION SBTITL(3)
DATA IAREA /5/, IREG /'WEST '/
ENCODE(SBTITL,14,1000) IAREA, IREG
1000 FORMAT('AREA ',I1,' - ',A5)
```

SBTITL will contain the character string AREA 5 - WEST. Notice that literals indicated in the format, as well as values named in the output list, are represented as characters in SBTITL.

The value of the length argument determines the length of the output record. If the string defined by the output list and format is longer than the specified number of bytes, only length bytes will be written to the output record, and the rest of the output list will not be encoded. If the string defined by the output list and format is shorter than the specified number of bytes, the remainder of the output record will be padded with blanks.

It is possible to ENCODE a character string consisting of multiple records, if the format contains slashes (/). In that case, the records are stored sequentially, and each record is length characters long. Records will not end with carriage returns; unless a carriage return is included explicitly in the format, i.e., as an ASCII value between backslashes in a literal string, no carriage returns will be put into string.

p 5-31

The first sentence should end:

" ... filled with blanks on the right."

In the subsequent example, change "nulls" to blanks."
p 5-32

Add the following to the description of the COMMON statement:

If an array name appears in a COMMON statement without dimension information, it must be dimensioned in a preceding DIMENSION, INTEGER, REAL, or LOGICAL statement.

p 5-40

Add the following:

5.9 EXECUTION TRACING

PTDOS FORTRAN provides an execution tracing facility for use in debugging programs. If tracing is enabled, FORTRAN will list on the console the name of each subprogram as it begins execution. In addition, for any routine that contains an OPTIONS declaration including the X option, the line number of each statement executed will be listed on the console. The line number displayed upon entry to a subprogram is always ???.

The form of the messages displayed on the console is:

    Pgm is executing line number in routine name

where number is the line number (or ?? upon entry to a subprogram), and name is the name of the routine being executed.

Execution tracing is enabled and disabled by means of the following statements:

General form:

    TRACE ON     Enables execution tracing.
    TRACE OFF    Disables execution tracing.

p 6-1

In the third sentence of Section 6.1.1, delete the phrase "or chains to another program." After that sentence, add the following:

Open files will be closed automatically by the CHAIN subroutine unless its second argument is negative (see Section 6.4).

p 6-2

Delete the paragraph beginning "READ and WRITE statements ...", including the example.
Section 6.1.2 implies incorrectly that the SPACE subroutine may be used to position only random files. In fact, it is not necessary to call RANDOM (or use the PTDOS RANDOM command) before calling SPACE.

Change the type to 'T' in the first example of use of the CREATE subroutine.

Replace the second sentence describing the CREATE subroutine with the following:

Only the first character of the string specifying the file type will be used; it is not possible to create an image file with the CREATE subroutine.

Add the following to the description of the CTYPE subroutine:

Only the first character of the string specifying the file type will be used; it is not possible to specify an image type with CTYPE.

Replace the second and third sentences describing the CHATTR subroutine with the following:

Each attribute is represented by one of the bits in a single byte value; a specific attribute will be set if the corresponding bit is a one. The value that will set a desired combination of attributes may be constructed by adding the appropriate values from the following list.

The program example at the bottom of the page is actually an example of the use of the CHATTR subroutine.

Change the title of the first box to "CIN Subroutine."

Replace the last two paragraphs describing the PLOT subroutine with the following:

If PLOT is called with no arguments, the screen is cleared and a zero is output to the video display scroll control register. This action forces memory location CC00H to be displayed at the upper left-hand corner of the screen. If the video display is provided by a VDM-1, its scroll control port address must be set to either C8H or FEH.
The description of the ISIGN function should read:
The sign of exp; +1 if positive, -1 if negative, 0 if zero.

Delete the "f" preceding AMOD in the first example.

Add the following entries to the statement summary:

DECODE (string,length,format) input list
Reads values from the character string in string and assigns them to variables in the input list.

ENCODE (string,length,format) output list
Writes values from the output list to string.

TRACE ON
Enables execution tracing.

TRACE OFF
Disables execution tracing.

The following entries replace those in the list:

CALL BIT(var,disp,'op',result)
Sets, resets, flips, or tests a single bit.

CALL CBTOF(loc,disp,var,flag)
Converts a binary number to its floating point equivalent.

CALL CHAIN('program name',action)
Chains to another program.

Add the following entry to the list:

CALL OUT (port,value)
Outputs value to a port.
Make the following changes in the list of compilation error codes:

6F    Illegal trace statement
71    Comma missing in ENCODE or DECODE statement
74-7F NOT USED
80    *FATAL* no program to compile

Reference 6 should be:

Programming Languages: History and Fundamentals
Jean E. Sammet
Prentice-Hall, Inc. 1969

REVISED DESCRIPTIONS OF SUBROUTINES, APPENDICES 6 AND 7

The following descriptions---as well as the revisions of Appendices 6 and 7---replace their counterparts in the manual. Where such a counterpart does not exist, the subroutine or function was added to FORTRAN after the release of the manual.

BIT Subroutine

General form:

    CALL BIT (var, disp, 'action' [,result])

variable name \ S, R, F, or T \ expression \ variable name

Sets, resets, flips, or tests a single bit of the variable var.

Examples:

    CALL BIT (NEWVAR, 0, 'S')
    CALL BIT (COUNT, 1, 'T', Bl)

The BIT subroutine sets, resets, flips, or tests a single bit of the variable var. The third argument specifies which action should be taken.

'S' means set the bit to 1;
'R' means reset the bit to 0;
'F' means flip the bit, i.e., change 0 to 1 and vice versa
'T' means test the bit and return its value in the fourth argument.

The fourth argument must appear if the third argument is 'T', and may not appear otherwise.
The value of the expression disp determines which bit of var will be affected. To determine the proper value for disp, consider the internal format of a stored value:

```
  byte byte byte byte byte byte byte
  0  1  2  3  4  5
value= nn nn nn nn nn 0s ee
  / / sign exponent
```

Where each n is a digit, and each byte consists of 8 bits numbered 0 to 7, from right to left. The value to assign to disp is given by the equation:

```
disp = (byte * 8) + bit
```

Where "byte" is the number of the desired byte (0 to 5), and "bit" is the number of the desired bit (0 to 7) within that byte. For example, to set the last bit of a value (i.e., bit 0 of byte 5), use

```
disp = (5 * 8) + 0 = 40
```

To set the first bit of a value (i.e., bit 7 of byte 0), use

```
disp = (0 * 8) + 7 = 7
```

CBTOF Subroutine

General form:

```
CALL CBTOF (loc, disp, var [ , flag ])
```

```
variable number of variable any
name or bytes value
memory address
```

Converts an 8- or 16-bit unsigned binary number whose location is determined by loc and disp into a floating point number, and stores that number in the variable var.

Examples:

```
CALL CBTOF (BVAR,0,DVAR)
CALL CBTOF (ARAY(1,1),6,VAL(2))
CALL CBTOF (X,1,Y,1)
```

The CBTOF subroutine converts an 8- or 16-bit unsigned binary number into its decimal floating point equivalent and stores the result in var.

If disp is positive or zero, loc is assumed to be the variable or array element that contains the binary form of the number. Disp specifies the displacement in bytes from the first byte of loc to the first byte of the binary number. For example, if the binary number occupies the second and third bytes of loc, the value of disp should
be 1; if the binary number occupies the first byte of `loc`, the value of `disp` should be 0.

If `disp` is negative, `loc` is assumed to contain or be an absolute memory address. If `loc` is a variable name or array element, that variable is assumed to contain the address of the first byte of the binary number. To specify an absolute address as the first argument, use the form `$имв`, where `имв` is a hexadecimal address.

If `flag` is omitted, the binary number is assumed to be a 16-bit quantity stored low-order byte first and ranging from 0 through 65535. If `flag` is present, the binary number is assumed to be an 8-bit quantity ranging from 0 through 255.

Program example:

```
DIMENSION A(4)
CALL FINFO('SUTIL',A)
CALL CBTOF(A(1),0,ID)
CALL CBTOF(A(1),6,NBLKS)
CALL CBTOF(A(1),12,BLKSZ)
WRITE(1,10)ID,NBLKS,BLKSZ
10 FORMAT('ID = ',I5,' # BLOCKS = ',I5,
  & ' BLOCKSIZE = ',I5)
END
```

This program retrieves status information for disk file `SUTIL` in binary form. It converts the information to decimal form and displays it at the terminal.

**CHAIN Subroutine**

General form:

```
CALL CHAIN ('file'{,action})
```

Loads the specified file into memory, and executes it if it has a starting address.

Examples:

```
CALL CHAIN ('PART2')
CALL CHAIN ('OVERLAY1',-1)
```

The CHAIN subroutine loads an image format file into memory at its normal load address, and executes it if it has a starting address. The specified file may be any existing image format file; because CHAIN is used most frequently to chain FORTRAN programs, all of which are loaded at 100H, the loaded program usually overwrites all or part of the calling program.

If the specified file does not exist or if an error occurs during loading, a FILE OP error occurs.

If the loaded program contains a starting address, it will begin execution immediately at that address; if not, the CHAIN subroutine
will return control to the statement that follows the call to CHAIN. If it is necessary to pass an argument to an assembly language program that is loaded by CHAIN, that program should contain no starting address and should be executed by the CALL function. (See Section 7.4.)

If the loaded program does not overwrite the caller, an 8080 return instruction will return control to the FORTRAN statement immediately following the call to CHAIN. If it is necessary to return a value to the calling program, an assembly language program loaded by CHAIN should be executed by the CALL function.

The optional second argument of CHAIN may be used to specify whether the runtime package is to be reloaded and whether or not files left open by the calling program are to be closed by CHAIN. If the value of the action argument is negative, the runtime package will not be reloaded and open files will be left open. If the value of action is greater than zero, the runtime package will not be reloaded and open files will be closed. If action is equal to zero or is omitted, the runtime package will be reloaded and open files will be closed.

Note that if the runtime package is not reloaded, a loaded FORTRAN program must expect the runtime package to reside at the same location as did the program that called CHAIN, i.e., neither or both of them were compiled with the $=R$ option, and neither or both of them were compiled with the long compile option.

Program Example:

Pl is the name of the image file containing the object code for the following program:

```
    TYPE 'THIS IS PART 1'
    CALL CHAIN ('P2')
    END
```

P2 is the name of the image file containing the object code for the following program:

```
    TYPE 'THIS IS PART 2'
    END
```

Execution:

User: P1 <CR>

First program: THIS IS PART 1

Second program: THIS IS PART 2

STOP END IN - MAIN
CONTRL Subroutine

General form:

CALL CONTRL (unit,op,DEin,HLin,Aout,DEout,HLout)
   /   /     /     \
   logical unit operation \ values returned in
   number code \ A, HL, and DE registers
      \ values supplied
      in DE and HL
      registers

Allows program control over devices or returns information about devices.

Examples:

    CALL CONTRL (0,2,0,'?',X,Y,Z)
    CALL CONTRL (FILE,4,0,$6500,X,Y,Z)

The CONTRL subroutine provides a mechanism by means of which a
FORTRAN program can make a Control/Status (CTLOP) system call. The
CTLOP system call allows a user program to control a peripheral
device or obtain information about its status.

A detailed description of the various operations that may be
performed and the significance of the data supplied or returned in
the A, DE, and HL registers may be found in Sections 7.2 and 9.2.2 of

Examples:

    CALL CONTRL (0,2,0,'?',X,Y,Z)
        Sets the console input prompt to ?.

    CALL CONTRL (2,4,0,$6500,X,Y,Z)
        Moves the index block of the random file
        associated with logical unit 2 from disk to
        memory location 6500H.

Program Example:

    IMPLICIT INTEGER (A-Z)
    CALL CONTRL (7,0,0,0,AR,DE,HLL)
    TYPE 'PROTECTION = ',AR
    TYPE 'CHRS = ',DE
    END

This program displays the protection attributes and device
characteristics of logical unit 7.
MOVE Subroutine

General form:

CALL MOVE (n,loc1,displ,loc2,disp2)   
/ / / / / \ 
expression /expression /expression
/ / / / / 
character string character string
or memory address or memory address

Moves n bytes from loc1 to loc2. If either disp is positive or zero, the corresponding loc is the symbolic name of the starting location. If disp is negative, loc contains or is an absolute memory address.

Examples:

CALL MOVE (6,'ABCDEF',0,$CC00,-1)  
CALL MOVE (2,A,-1,$CC00,-1)    
CALL MOVE (1024,$CC00,-1,A,-1)  
CALL MOVE (10,COUNT,2,ADD1,-1)

The MOVE subroutine moves n bytes from loc1 to loc2. The interpretations of loc1 and loc2 depend on the values of displ and disp2, respectively.

If either disp is positive or zero, the corresponding loc is assumed to be the symbolic name of the location containing the first byte of the string to be moved (if loc1), or the first byte of the destination for the moved string (if loc2). Thus, loc1 or loc2 may be a variable name, an array name, or an array element. Loc1 may also be a literal string enclosed in single quotes; in that case, the literal string contains the first and subsequent bytes of the string to be moved. In any of these cases, disp specifies the displacement in bytes from the first byte of loc to the actual starting byte.

If either disp is negative, the corresponding loc is assumed to contain or be an absolute memory address. If loc is a variable name or array element, that variable is assumed to contain the address of the first byte to be moved, or the first destination location. To specify an absolute memory address, use the form $addr, where addr is a hexadecimal address.
Explanation of examples:

CALL MOVE (6,'ABCDEFG',0,$CC00,-1)
   Moves ABCDEF to address CC00H

CALL MOVE (2,A,-1,$CC00,-1)
   Moves 2 bytes from the address stored in A to address
   CC00H

CALL MOVE (1024,$CC00,-1,A,-1)
   Moves 1024 bytes starting with address CC00H to the
   address specified by A.

CALL MOVE (10,COUNT,2,ADD1,-1)
   Moves 10 bytes, starting at the third byte of COUNT, to
   locations starting at the address stored in ADD1.

OUT Subroutine

General form:

CALL OUT (port, value) Outputs value to the specified port.

Example:

CALL OUT (254,0)

The OUT subroutine enables a FORTRAN program to output an 8-bit value
at a specified hardware port. If value or port lies outside the
range 0-255, its value modulo 256 is used.

Explanation of example:

CALL OUT (254,0)

This example outputs a zero to port FEH, the Sol video display scroll
control register. As a result, all 16 lines of the video memory will
appear on the screen, with line 0 at the top.
APPENDIX 6

EXECUTION ERRORS

A program may compile correctly and still generate "execution errors" at runtime. Unless an execution error is trapped by means of an ERRSET statement, it causes a message to be displayed and execution of the program to be terminated.

Execution error messages have the form:

Runtime error: message, called from location

Pgm was executing line number in routine name

where:

message is one of the messages shown below.

location is the memory location of the error-producing call to the runtime package.

The locations assigned to individual statements are NOT indicated on any listing generated by FORTRAN. To obtain a listing with locations shown, first compile the program, specifying the B option (to get FORTRAN statements interspersed with assembly language code); then use the PTDOS assembler to assemble $FORTASM (or the "assembly" file named in the FORTRAN command line) and specify the listing option.

number specifies the line in which the error occurred. That number will be intelligible only if the X option was declared in the named routine. (See the description of the OPTIONS statement in Section 3.2.) Otherwise, the line number will appear as ???? If several line numbers are listed, the error actually occurred in the first line specified.

name is the name of the routine in which the error occurred.

Unless they are trapped, four of the execution errors - namely, FILE OP, I/O ERR, OPEN ERR, and SET UNIT - result in calls to the PTDOS Explain Error Utility (UXOP). UXOP supplies a detailed explanation of the error and returns control to PTDOS. After one of these errors has occurred, it is no longer possible to rerun the FORTRAN program in memory, because UXOP uses memory occupied by the runtime package. In order to run the program again, you must reload it from disk.
The error code for any runtime error may be supplied as an argument in the ERRSET statement, which has the form:

\[
\text{ERRSET n, v}
\]

and signifies "If error v occurs, transfer control to the statement labeled n. (See page 5-10 for a description of ERRSET.) It is advisable to assume that any variable will be undefined after an error that involves it.

The possible execution errors and their ERRSET codes are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Message</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| 3    | ARG CNT | ARGUMENT COUNT ERROR  
Too many or too few arguments were passed to a subprogram. |
| 22   | ASN GOTO| ASSIGNED GOTO LABEL NOT IN LIST  
The variable in an assigned GOTO statement did not have the value of any label in the list following it (n1, n2, etc.). |
| 10   | CALL POP| CALL STACK POP ERROR  
A RETURN without a destination was executed. This error can be caused only by a user's assembly language program. |
| 9    | CALL PSH| CALL STACK PUSH ERROR  
More than 62 nested subprogram calls were made. |
| 4    | COM GOTO| COMPUTED GOTO INDEX OUT OF RANGE  
The variable specified in a computed GOTO statement has a value either less than one or greater than the number of statement labels specified. |
| 2    | CONVERT | 16 BIT CONVERSION ERROR  
An overflow has occurred during the conversion of a number to internal 16-bit binary form. This error can occur during 1) the conversion of a unit number in an input/output statement, 2) the evaluation of a subscript, or 3) the conversion of a floating-point number to 16-bit binary form. |
| 6    | DIV ZERO| DIVIDE BY ZERO  
An attempt was made to divide by zero |
| 11   | FILE OP | FILE OPERATION ERROR  
An error occurred during an operation involving a PT DOS file. This error results in a call to UXOP; see the note above. |
18 FORMAT INVALID FORMAT
A formatted READ or WRITE referred to an invalid FORMAT specification.

12 ILL UNIT ILLEGAL LOGICAL UNIT NUMBER
A logical unit number less than 2 or greater than 63 was passed to one of the input/output rougtines.

20 INPT ERR INPUT ERROR
The representation of a number in the input file contained an invalid character. Examples of invalid characters are a second decimal point in a number, an E in an F-type field, a decimal point in an I-type field, etc.

1 INT RANG INTEGER OUT OF RANGE
An integer operation resulted in a number too large to be stored.

19 I/O ERR I/O ERROR
An error occurred during a READ or WRITE operation, and there was no error label (for an error other than an end-of-file) or no end-of-file label (for an end-of-file error). This error results in a call to UXOP; see the note above.

21 I/O LIST INVALID I/O LIST
A formatted READ or WRITE statement contained an error in the input or output list. This error can occur only if the I/O list was constructed by a user's assembly language program.

17 LINE LEN LINE LENGTH ERROR
An attempt was made to read or write a record more than 250 characters long.

8 LOG NEG LOG OF NEGATIVE NUMBER
ALOG or ALOG10 was called with a negative argument.

14 OPEN ERR OPEN ERROR
An error occurred during the execution of an OPEN statement. This message encompasses all OPEN errors, except the case of a nonexistent file. (If the file named in the OPEN statement does not exist, it is created.) This error results in a call to UXOP; see the note above.

5 OVERFLOW FLOATING POINT OVERFLOW
A floating-point operation resulted in a number too large to be stored.
16 SET UNIT ERROR
An error occurred during reassignment of the default unit (drive) number. This error results in a call to UXOP; see the note above.

7 SQRT NEG
SQRT OF NEGATIVE NUMBER
The SQRT function was called with a negative argument.

15 UNIT CLO
LOGICAL UNIT NOT OPEN
There was an attempt to read, write, rewind, or perform some other operation on a logical unit that was not associated with a file.

13 UNIT OPN
LOGICAL UNIT ALREADY OPEN
A file has been assigned a logical unit number already assigned to another file.
Extended Disk FORTRAN includes these extensions to ANSI standard FORTRAN (X3.9-1966)

* File management, including creating, killing, and changing attributes.
* Random access to data files
* Input from and output to device files, including character-at-a-time terminal input
* Direct control over the video display
* Free format input and output
* Optional end-of-file and error branches in READ and WRITE statements
* Arrays with up to seven dimensions
* Hexadecimal constants
* Character string data type and string move and compare routines
* ENCODE and DECODE statements for formatting data in memory
* IMPLICIT statement for implicit typing of variables and functions
* COPY statement to copy files of source statements into a FORTRAN source program
* Assembly language interface - embedded assembly language statements and calls to assembly language routines
* Access to absolute memory locations, including individual bits
* Program-controlled time delay
* Pseudo-random number generator function
* Program control of runtime error trapping
* Execution tracing
* Ability to chain a sequence of programs
PTDOS FORTRAN does not conform to ANSI X3.9-1966 in the following respects:

* Double precision variables, constants, functions, and format specifications are not provided.

* Complex variables, constants, and functions are not provided.

* There is no EQUIVALENCE statement.

* ANSI FORTRAN allows DATA statements such as:

  DATA X,Y,Z/10,20,30/

In PTDOS FORTRAN this statement would have to be changed to:

  DATA X/10/,Y/20,Z/30/

* Statement functions are not allowed.

* The following format specifications are not available: carriage control characters, D, G, and P.

* If an array name appears without dimension information in a COMMON statement, it must be dimensioned in a preceding DIMENSION, INTEGER, REAL, or LOGICAL statement.

* A variable may not have the same name as one of the system functions listed in Appendix 3. COPY may not be used as a variable name.

* Only the first five characters of function or subroutine names or COMMON labels are retained. For example, PTDOS FORTRAN does not differentiate between MYSUBL and MYSUB2.

* The following cannot be used for subroutine, function or COMMON names: A, B, C, D, E, H, L, M, SP, PSW, or any PTDOS reserved name contained in the files PTDEFS and NPTDEFS.

* The SIGN and ISIGN functions provided by PTDOS FORTRAN have a single argument. In ANSI FORTRAN, these functions have two arguments and transfer the sign of the second argument to the first.

* The hyperbolic tangent function, TANH, is not provided.