

FORTTRAN TILL ABC 80

Version 1.0

COPYRIGHT DATAINDUSTRIER AB, SWEDEN

TABLE OF CONTENTS

<u>Paragraph</u>		<u>Page</u>
1.0	INTRODUCTION TO ABC 80 FORTRAN	2
1.1	Sample ABC 80 FORTRAN Program	2
1.2	ABC 80 FORTRAN Statement Format	3
1.3	How to run the ABC 80 FORTRAN compiler	4
2.0	INTEGER CONSTANTS, VARIABLES AND EXPRESSIONS	5
2.1	Integer Constants	5
2.2	Integer Variables	6
2.3	Integer Declarations	7
2.4	Integer Valued Arithmetic	8
2.5	Assignment Statements	10
2.6	Flags	11
3.0	LOGICAL OPERATORS	13
3.1	Logical Constants	13
3.2	Logical Variables	13
3.3	Logical Operators	13
3.4	Relational Operator	14
4.0	DIRECT INPUT/OUTPUT	16
5.0	CONTROL STATEMENTS	17
5.1	Unconditional GO TO Statement	17
5.2	Computed GO TO Statement	17
5.3	Logical IF Statement	18
5.4	Arithmetic IF	19
5.5	STOP, END Statement	20
6.0	THE DO AND CONTINUE STATEMENTS	21
6.1	Introduction	21
6.2	Rules Governing Use of the DO Statement	23
6.3	CONTINUE Statement	25
6.4	Nested DO Loops	26

<u>Paragraph</u>	<u>Page</u>
7.0 SUBSCRIPTED VARIABLES	28
7.1 Introduction	28
7.2 Subscripts	29
7.3 Use of Arrays	29
8.0 SUBPROGRAMS	31
8.1 Introduction	31
8.2 Subroutine subprograms	31
8.3 Function Subprograms	36
8.4 Global Variables	38
8.5 Global Subprograms	39
9.0 FLOATING POINT OPERATIONS	41
9.1 Single and double precision constants	41
9.2 Declarations	42
9.3 Floating Point Arithmetic	42
9.4 Floating Point I/O	43
9.5 Floating Point Format	44
10.0 FILE I/O	45
10.1 Introduction	45
10.2 The INCLUDEIO directive	45
10.3 The OPEN subroutine	46
10.4 The CLOSE subroutine	47
10.5 The GET and PUT subroutine	47
10.6 The WRITE Statement	49
10.7 User defined output devices	50
10.8 The READ Statement	50
10.9 User defined input devices	52
10.10 The FORMAT Statements	52
11.0 CHARACTER STRINGS	55
12.0 SPECIAL FEATURES	57
12.1 Inline Code	57
12.2 Accessing Compiler Directives	59

Appendix "A" - ABC 80 FORTRAN Keywords	61
Appendix "B" - ABC 80 FORTRAN Error Messages	62
Appendix "C" - ABC 80 FORTRAN Features not supported by FORTRAN IV	69
Appendix "D" - FORTRAN IV Features not supported by ABC 80 FORTRAN	70
Appendix "E" - Sample Program Run	71

PREFACE

This manual is not intended to teach the non-programmer FORTRAN, but rather it is intended to provide quick reference information for the FORTRAN user. The novice programmer should refer first to a standard FORTRAN text (see Bibliography) before reading this manual.

1.0 INTRODUCTION TO ABC 80 FORTRAN

1.1 Sample ABC 80 FORTRAN Program

A FORTRAN program consists of an ordered sequence of statements. These statements may specify arithmetic operations, the input of data from a console or input port, or the outputting of data to a CRT or output port. Normally, statements are executed in the order in which they are written. However, another group of statements allows the user to conditionally or unconditionally alter the 'program flow'. Finally, a group of statements exist which do not perform any action but rather provide information concerning the nature of the procedure.

A sample ABC 80 FORTRAN program is:

```
C      SAMPLE PROGRAM
      I = 0
5     I = I + 1
      J = I * I
      OUTPUT (10) = J+I
      IF (I.LT.10) GO TO 5
      STOP
      END
```

As we see above, each line may contain only one statement or may be blank. Also since blanks have no significance in most statements, we may rewrite the sample program as,

```
      I = 0
5     I = I + I
      J = I * 1
      OUTPUT (10) = J+I
      IF (1.LT.10) GO TO 5
      STOP
      END
```

Handwritten annotations:
An arrow points from the handwritten "1" in "1.LT.10" to the "1" in "I * 1".
An arrow points from the handwritten "I" in "I * 1" to the "I" in "I = I + I".
A bracket groups the "1" in "I * 1" and the "1" in "1.LT.10" with a question mark.

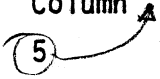
ABC 80 FORTRAN Keywords (see Appendix "A") and user defined symbolic names should not contain any imbedded spacing.

However, judicious use of blanks and blank lines can greatly improve the readability of a program while facilitating a better understanding of the algorithm employed.

1.2 ABC 80 FORTRAN Statement Format

ABC 80 FORTRAN statement record consists of up to 72 characters. These are used in the following manner.

Column 1 A 'C' in this position indicates the statement is to provide explanatory information and is only listed at compile time.

Column 5  An optional integer statement number used to reference a selected point in a program.

Column 6 A non-blank character indicates that the previous statement is continued on the present line starting at the following character.

Column Actual ABC 80 FORTRAN statement.
7 - 72

Column Ignored.
73 +

NOTE: Comments may not be interspersed between continuation statements. Thus, the following is not valid:

```
      INTEGER I ATOP
C      INITIALIZE A/D CONVERTER
      *I/OFCH/
```

1.3 How to run the ABC 80 FORTRAN compiler

The compiler is started by:

`FORTRAN, <opt> <source file>, <object file>`

<opt> is one or more of the following options

- L - list program on printer
- O - generate an object file
- I - list generated code
- D - display program on console

If no options are specified an object file and printer listing will be generated.

<source file> is the name of the file containing the source program. The default extension is '.FTN'

<object file> is the name of the generated object file. If omitted the object file will be named similar to the source file but with the extension '.ABS'.

Examples:

`FORTRAN,D TEST`

`FORTRAN,LOI SORT.TXT, BINSORT.ABS`

`FORTRAN,OD KALLE.FTN, KALLE.ABS`

2.0 INTEGER CONSTANTS, VARIABLES AND EXPRESSIONS

2.1 Integer Constants

ABC 80 FORTRAN supports two forms of integer constants, single byte and double byte. An integer and its sign must therefore be converted into either an 8 or 16 bit number. Therefore, single precision (8 bit) numbers are restricted to the range -128--+127 while double precision (16 bit) numbers may range from -32768--+32767. In either case the left (most significant) bit is relegated to storing the sign of the integer. A '0' in this bit means the number is positive, while a '1' indicates that it is negative.

In addition to decimal constants, ABC 80 FORTRAN also offers the capability of working with binary, hexadecimal, and octal constants. All hexadecimal (base 16) constants are suffixed by the letter 'H'. For example, 11H is a hexadecimal constant equal to decimal 17. However, care must be taken to ensure that a 'Hex' constant is not confused with an identifier as in the case FFH. In such cases, the hexadecimal number must be prefixed by the numeral '0' to eliminate any possible confusion, i.e., 0FFH. Octal (base 8) constants may be specified in either of two ways. The number may be suffixed by either the letter 'Q' or the letter 'O'. Thus, the following are equivalent representations of the decimal number 110: 11Q, 11O. Binary numbers are similarly suffixed by the letter 'B' and are thus written in the following form: 10100B (decimal 20). Decimal (base 10) constants do not require any suffix to identify them.

The following are valid integer constants:

Hexadecimal:	0FH	1BH	-3FH
--------------	-----	-----	------

Octal:	22Q	-152O	47Q
--------	-----	-------	-----

Q ?

Floating point constants, variables and expressions are discussed in section 9.

Binary:	1B	10101010B	11111B
Decimal:	20	+1	-16

The following are invalid integers:

1.5	(no decimal point allowed)
4,000	(no commas allowed)
FCAH	(must have leading '0')

2.2 Integer Variables

While a constant has but one unalterable value, a variable is able to take on one or many values during the course of a program. An integer variable may take on any value that is specified in the range discussed in the previous section. Thus, a single precision variable may vary from -128 to +127 while a double precision variable may vary from -32768 to +32767.

A symbolic name or identifier in ABC 80 FORTRAN may consist of up to 31 symbols with the following restrictions, namely,

- (a) the first symbol must be a letter (A-Z)
- (b) the symbols must be a letter (A-Z), digit, '_' or '?', and
- (c) the symbolic name must not be a ABC 80 FORTRAN Keyword.

Thus the following symbols are valid:

```
I
JKL
INPUT DATA
INPUT DATA FROM THE ATOD CONV
```

while these are not:

5M1	(first character not a letter)
1\$2	(invalid symbol \$)
CARRY	(ABC 80 FORTRAN Keyword)

It should be noted that standard FORTRAN accepts only up to six character variable names. The use of longer variable names which ABC 80 FORTRAN allows is recommended in that it tends to make a program self-documenting.

2.3 Integer Declarations

Integer variables must start with a letter from I through N unless an explicit declaration statement is used. The INTEGER*1 or INTEGER*2 statement can be used to explicitly state the attributes of any integer variables. For example,

```
INTEGER*2  A,B,C
```

states that the variable A,B,C are all two byte integer variables.

Since implicit integer variables default to two-byte (double precision) variables, the INTEGER statement is also useful for defining byte variables. For example,

```
INTEGER*1  I,TEXAS
```

declares I and TEXAS to be single precision (1 byte) integer variables.

An additional feature of the INTEGER declaration is its ability to initialize variables. An example of this facility is:

```
INTEGER*1  A/17H/,B
```

```
INTEGER*2  C/-1762/
```

which will give the variables the following initial values:

```
A  23
B  indeterminate
C  - 1762
```

As we will see later, the capability is even more useful for initializing array variables.

The initialized values will be loaded into memory at the same time the program is loaded. This means that if the programmer should not expect the values to be re-initialized if a program is executed and re-executed without reloading. Also any program which will be contained in PROM should ensure that tables, etc., using data initialization on are also contained in PROM.

2.4 Integer Valued Arithmetic

ABC 80 FORTRAN provides five basic arithmetic operators; namely, +, -, *, /, and unary -. These symbols represent addition, subtraction, multiplication, division and change of sign respectively. Formulas or expressions are built up using any combination of these operators with appropriate variables or constants. As in ordinary algebra, the hierarchy of these operators, unless otherwise specified by parenthesis, is:

	*	/	(equal priority)
then	+	-	(equal priority)

When no other rules apply, or in the case of equal priority operations, all expressions are evaluated from left to right.

In writing expressions, the following rules apply:

- (1) Parenthesis must be used in order to force a lower priority operation to occur ahead of one of a higher priority;

Example: $(X + Y) * (36 / (18 + X))$

When in doubt about the order in which an expression will be evaluated, parenthesis should be used.

- (2) Parentheses do not imply multiplication, simply groupings.

Example: $(A+B)(C+D)$ must be written as
 $(A+B) * (C+D)$

When the result of an expression exceeds the range of numbers available in ABC 80 FORTRAN, an overflow is said to occur. Care should be taken to avoid the possibility of overflows whenever possible. For example,

$1000 * 100 / 20$
should be written as $1000 * (100 / 20)$

so that division occurs first and an overflow is avoided. In ABC 80 FORTRAN, overflows will be ignored and execution will continue with a potentially meaningless value. No real time error message is generated.

Integer division may result in an answer which is not an integer. In such cases, the result is truncated and the fractional part is ignored. For example, $(5/2)$ would yield a result of 2. Thus, when making a calculation such as $(5/3) * 6$ a good idea would be to try rearrange the expression to read $6/3 * 5$ thereby providing the correct answer 10 rather than 6.

Finally, it should be noted that single byte arithmetic is modulo 256. This means that when counting the upper limit of the range the following sequence occurs: 126, 127, -128, -127, -126 etc. Thus we see that the addition of 125 and 10 will yield a result of -121. Similarly, double byte arithmetic is carried out modulo 65535.

For efficiency, a programmer should ensure all the operands in an expression are of similar type and precision. However, if an operation is performed between a single and double byte operand, then the single byte operand will be converted to a double byte

operand and the result will be a double byte value. The only exception to this rule is in the case of an '=' operator in which the precision of the left hand operand takes priority and the least significant byte of the right hand side is used.

Example: ~~INTEGER*2~~ X,Y
 ~~INTEGER*1~~ A1,B1
 C B1 is converted to double byte
 X = Y*B1
 C X+Y is converted to single byte
 A1 = X+Y

2.5 Assignment Statement

The general form of an assignment statement is:

(variable)=(expression)

Such a statement gives instructions to perform the required computation and assign the resulting value to the variable at the left of the equals sign.

In fact, the equals sign is not the same as that employed in conventional mathematical notation. Rather, it states that the value of the expression on the right is assigned to the variable on the left. For this reason, it is known as the "assignment ~ operator". While the statement $N=N+1$ is meaningless mathematically, it is interpreted by the compiler to mean assign to N the old value of N incremented by one. Such statements are very useful, but it must be remembered that the previous value of N is lost. Some examples of assignment statements are:

```
BOB=34
SUSAN=INPUT@DATA/34 BOB)
JACK=JACK+3
```

A unique FORTRAN extension available in ABC 80 FORTRAN is its ability to accept multiple assignment operators in an assignment statement. An example of this is:

```
A=B+C=D+E
```

In this case, C is assigned to D + E and then A is assigned to B + C. Thus, assignment operators are evaluated from right to left.

Some examples of incorrect assignment statements are:

```
Z - 2 = X+Y      (left side must be single variable)
B = 4(A - C)     (missing operator)
```

2.6 Flags

Four reserved variables are available for examination after every assignment statement. These are:

CARRY	--.FALSE.if carry flip flop is clear
	--.TRUE.if carry flip flop is set
PARITY	--.FALSE.if result has odd parity
	--.TRUE.if result has even parity
SIGN	--.FALSE. - positive number
	--.TRUE. - negative number
ZERO	--.FALSE.if result is non-zero
	--.TRUE.if result is zero.

These flags may be used as any variable might be employed. For example,

```
IF (CARRY) GO TO 5
```

or

```
A = B + C
```

```
D = E +(CARRY.AND.1)
```

or

```
A = E - D
```

```
IF (ZERO) GO TO 17
```

The complements of the four flags are also available, namely,

```
CARRYOFF
```

```
SIGNOFF
```

```
PARITYOFF
```

```
ZEROOFF
```


3.0 LOGICAL OPERATORS

3.1 Logical Constants

A logical constant is a single byte number that has two possible values.

.FALSE. representing false .
.TRUE representing true

3.2 Logical Variables

These are in fact nothing more than single byte integer variables. However, the user regards them as taking on only one of two possible values, either .FALSE. or .TRUE.

ABC 80 FORTRAN dose not support an explicit LOGICAL type variable.

3.3 Logical Operators

There are four logical operators available for use in ABC 80 FORTRAN. These are .AND., .OR., .XOR. and .NOT. The following truth table summarizes the action of these operators.

<u>X</u>	<u>Y</u>	<u>X.AND.Y</u>	<u>X.OR.Y</u>	<u>X.XOR.Y</u>	<u>.NOT.X</u>
T	T	T	T	F	F
T	F	F	T	T	F
F	T	F	T	T	T
F	F	F	F	F	T

WHERE T REPRESENTS A TRUE CONDITION
AND F REPRESENTS A FALSE CONDITION

Logical operators also serve another function. They allow the programmer to mask, set or reset a bit or group of bits in a variable. If we wish to clear the upper four bits of the byte

variable BOB then the following expression will accomplish this:

(BOB.AND.OFH)

The constant OFH is referred to as a mask.

3.4 Relational Operator

There are six relational operators in ABC 80 FORTRAN. A list of these follows.

.EQ.	equal to
.LT.	less than
.GT.	greater than
.NE.	not equal to
.LE.	less than or equal to
.GE.	greater than or equal to

These six binary (two operands) operators always result in a logical value; '00' if the relation is false and 'FF' if the relation is true. By combining logical and relational operators, complex expressions may be easily written.

Example 1:

```
X =(A.LE.B) .AND. OFCH
if A ≤ B      then X = OFCH
if A > B      then X = 00H
```

Example 2:

```
X = .NOT. (A.XOR.B)
X equals exclusive NOR of a,b
```

NOTE:

Due to the limitations of the Z80 and the use of signed arithmetic, certain restrictions must be obeyed when using a relational operator, either in an assignment statement or a logical IF statement. These rules are:

- a, The results of the two expressions (e_1 and e_2) which are being compared must be

$$\begin{aligned} -64 &\leq e_1, e_2 \leq 63 \text{ (single precision)} \\ -16384 &\leq e_1, e_2 \leq 16383 \text{ (double precision)} \end{aligned}$$

This insures that all relational operators will function correctly without any danger of overflow conditions occurring.

- b, If e_1 and/or e_2 do not fall within the limits above, then the programmer must insure that

$$\begin{aligned} e_1 - e_2 &\leq 127 \text{ (single precision)} \\ e_1 - e_2 &\leq 32767 \text{ (double precision)} \end{aligned}$$

Once again, by observing this rule, the relational operators will consistently work.

4.0 DIRECT INPUT/OUTPUT

ABC 80 FORTRAN provides the user with an efficient method of accessing the I/O-system at its most basic level. This is done by the use of

INPUT (n) and OUTPUT (n)

where $n, 0 \leq n \leq 63$, is the port number which must be a constant. INPUT is used exactly as one would use an INTEGER*1 function call statement. Thus it returns a single 8 bit value which has been sampled by port 'n' of the CPU. It may be used as part of an expression.

Example:

A =(INPUT (06) + INPUT (07)) / 2

OUTPUT (n) must always appear on the left hand side of an assignment statement. It transfers the eight bit result of the assignment statement into port 'n' of the CPU. If the assignment statement yields a double byte result, only the lower (least significant) byte is output. A sample of the OUTPUT statement is shown below:

OUTPUT (16) = (A + C) / 2 + INPUT (17)

5.0 CONTROL STATEMENTS

5.1 Unconditional GO TO Statement

This statement allows the user to unconditionally transfer control to some other statement within his program. Although this is a very straightforward statement, two rules must be followed when using it; namely,

- 1, Control must only be transferred to executable statements. Hence, control cannot be transferred to FORMAT, DECLARATION, or END statements.
- 2, The statement immediately following the GO TO statement must have a statement number or it can never be referenced.

5.2 Computed GO TO Statement

This statement is similar in function to the unconditional GO TO except that it allows multi-way branching. This capability is very useful whenever a different set of calculations is to be performed depending upon the value of a single index variable or expression.

The general form of the statement is

$$\text{GO TO } (n_1, n_2, n_3, \dots, n_m) , E$$

where $n_1, n_2, n_3, \dots, n_m$ are valid statement numbers,

and E is a non-negative ABC 80 FORTRAN expression.

It should be noted that unlike FORTRAN IV, ABC 80 FORTRAN allows expressions in place of the index variable. Thus the following computed GO TO will execute the indicated jumps depending upon the value of the expression (MUSSE+2).

```
GO TO (10,20,30,40), MUSSE+2
50 CONTINUE
```

<u>BRANCH</u>	<u>VALUE OF MUSSE+2</u>
10	1
20	2
30	3
40	4
50	5

5.3 Logical IF Statement

The general form of the logical IF statement is given by either

```
IF (logical expression) GO TO n
or IF (logical expression) RETURN
or IF (logical expression) ASSIGNMENT
or IF (logical expression) CALL subroutine
```

where 'n' is any statement number.

The logical expression may use any of the six available relational operators or any of the four logical operators previously mentioned. The action of the statement is to execute the GO TO, RETURN, ASSIGNMENT or CALL statement following the closing parentheses if and only if the logical expression is true. Otherwise control is passed to the next sequential statement. If an expression is used, then the jump will be taken if the value of the expression is negative.

Some examples of logical IF statements are:

```
IF (A.EQ.0) GO TO 17
IF (A+B.GT.(C*D)/(2+B) ) RETURN
IF ((A.GT.B).AND.(C.GT.D)) GO TO 25
IF (A.LT.72) I=4*A+2
IF (C.LT.2.AND.DOG.GE.7)CAT = HAT = 121
```

5.4 Arithmetic IF

The arithmetic IF statement takes the following form:

IF (expression) statement number 1, statement number 2, statement number 3.

Each of the three statement numbers can be equal or different from one another. The expression must be enclosed in brackets and can be any valid ABC 80 FORTRAN expression. When this statement is executed, the expression is evaluated and a jump is then made to one of the statement numbers in the list.

If the expression is negative, then a jump is made to statement number one. If the expression is zero, then a jump is made to statement number 2. If the expression is non-zero and positive, then a jump is made to statement number 3.

For example, consider the calculation of the function:

$$A = \frac{B^2 + 5B - 3}{(B - 5)} \text{ for } B = 0, 5, 10, 15, 20, 25$$

B = 0.0

5 IF (B-5.0) 6,7,6

6 A = (B*B+5.0*B-3.0) / (B-5.0)

WRITE (1,3) B,A

3 FORMAT (5X,F6.1,'THE ANSWER', E20.7)

7 B = B+5.0

IF (B-25.0)5,5,8

8 STOP

END

In the example, the first arithmetic IF statement is used to avoid a division by zero in the following statement. The other arithmetic IF statement is used to exit from the program if B becomes greater than 25.0.

NOTE: The logical IF statement should be used instead of an arithmetic IF statement wherever it is practical to do so. The logical IF usually makes the program more readable and also produces code which is usually more efficient.

5.5 STOP, END Statements

The STOP statement may be written wherever it is necessary to stop executing statements in a program. It is known as "control" statements because it controls the execution of the program. STOP does not stop the computer but merely terminates execution of the program and returns to the operating system. There normally is a STOP at the end of every program when computation is over. However, there are other uses. In order to check input data for consistency, STOP statements may be incorporated to give the operator an indication that something is wrong.

A STOP statement may appear anywhere in a program and there may be more than one.

The END statement must be physically the last statement of every program. It is also used to terminate the declaration of both subroutine and function sub-programs. Optionally, a transfer address may be placed after the END statement as the following example indicates:

```
END 3AFCH
```


6.0 THE DO AND CONTINUE STATEMENTS

6.1 Introduction

The DO statement is one of the most powerful and widely used statements available to the FORTRAN user. It allows the programmer to execute a section of program repeatedly while automatically varying the value of an integer variable between repetitions. Thus, a program loop may be easily written without the need of an IF statement. The DO statement takes on one of two general forms, namely,

	DO n i= m ₁ , m ₂ ,
or	DO n i= m ₁ , m ₂ , m ₃
where n	is the statement number of the termination state- ment of the DO statement (also known as the 'object');
i	is an integer variable whose value is varied by the DO statement. It is commonly called the 'index'.
m ₁	is an expression giving i its initial value.
m ₂	is an expression stating i's upper bound.
m ₃	is an expression yielding the increment which will be added to 'i' at the end of each repetition of the Do loop. If it is not present as in the first general form, +1 is used.

The action of the DO statement is to repeatedly execute all statements between the DO statement itself and the object. This block of statements is referred to as the scope or range of the DO loop. The first execution will have $i=m_1$, the second with $i=m_1 + m_3$ and each successive run will have i incremented by m_3 . The repeated executions will continue until control is transferred outside the scope of the DO loop.

There are two methods by which control can be transferred outside the range of a DO loop. The normal exit occurs when the DO loop upper bound has been exceeded, i.e. $i > m_2$.

This causes control to be transferred to the next executable statement after the object. The second method utilizes a GO TO, IF or RETURN statement to carry control out of the DO loop. Regardless of the exit mechanism employed, the index variable *i* is no longer available for use by the programmer.

However, the index variable *i* is available for use during the execution of the DO loop as shown in the following example.

```
C Example 6.1 --Cubes of even integers ten with
a DO statement
INTEGER*2 CUBE (5)
DO 15 J=2, 10,2
15 CUBE (J/2) = J*J*J
CALL DISPLAY (CUBE)
STOP
END
```

By contrast, if we did not use a DO loop the following program would result

```
C Example 6.2 -- Cubes of even integers  $\leq$  ten without
a DO statement
INTEGER*2 CUBE (5)
J = 2
15 CUBE (J/2) = J*J*J
J=J+2
IF (J.LE.10) GO TO 15
CALL DISPLAY (CUBE)
STOP
END
```

The index variables need not be employed in any function other than counting the number of times a loop is executed. In this regard, it is important that the range of the DO is executed precisely the required number of time. Many errors occur because

the DO loop is executed once too many or once too few times. A little consideration can save a lot of wasted effort. Some examples follow:

```
DO 20 I=1, 9, 3
DO 30 JACK=2, 9, 5
DO 40 KAL=2, 9, 11
DO 50 MACK=11, 1000, 37
```

in which the number of executions are 3, 2, 1 and 27 respectively. Some examples of how expressions may be used in the parameters of a DO statement are shown below.

```
DO 20 MY= -5, 7, 2
DO 36 KITE= (A=B+C), (A+B+C)
DO 71 KLINGON = 7, -B, M+N
```

6.2 Rules Governing Use of the DO Statement

- (1) The DO index must be either a single or double byte integer variable while the remaining DO parameters must be integer constants, variables or arithmetic expressions. Real variables and constants may not be used. If the increment parameter is not specified, it is assumed to be +1.

Thus, the following DO statements are valid:

```
DO 18 IBOB=IDAVE, IED
DO 69 ISUE=3, HEIDI, LAST
```

while these have violated the first rule:

```
DO 700 K=.017, X, -.4
DO 900 Z=7, .4
```

- (2) The parameters of a DO statement must not be modified within the range of the DO statement. Failure to observe this rule may cause unexpected results. The following example illustrates a violation of this rule.

```
      K=7*I
      DO 100 IT=K,M,17
      X=(7*I)/14
      M=X/17
12 Y(IT)=M+X
```

- (3) The range of the DO is always performed at least once regardless of the parameter values since the testing of the index value is done following execution. Thus the following DO loop will be executed once.

```
      DO 100 J=1,-100
```

- (4) The object (or termination statement) must be an executable statement with the following exception: (a) STOP (b) Another DO (c) GO TO.
- (5) Once a DO loop is exited, the value of the index variable shall be considered to be undefined. Execution of the following program will not guarantee K being equal to 7 as one might expect:

```
      DO 10 J=1,10
10 IF (J.EQ.7) GO TO 20
20 K=J
      STOP
      END
```

- (6) As shown in the previous example, it is possible to exit from a DO loop before the DO statement is satisfied. However, it is not permitted to enter the range of a DO statement without entering by means of the DO statement itself. An example of a violation of this rule is

```
GO TO 76
.
.
.
DO 77 I=10, 20,2
76 J=I*7
77 KIT (J)=M*I+79
```

6.3 CONTINUE Statement

The CONTINUE statement has no effect on any variables in the program. Rather, it is a dummy statement to enable the user to terminate a DO loop properly. It also provides a point where jumps within the range of the DO loop may be made in order to increment the index variable and begin executing the scope of the DO loop. An example of the CONTINUE statement is shown below:

```
DO 100 J=10, 20, 2
IF (A(J).EQ.10) GO TO 100
A(J)=A(J)+7
.
.
.
100 CONTINUE
```

6.4 Nested DO Loops

Do loops may be nested, i.e. one DO loop may contain another DO loop which may contain yet another, etc. Nested DO loops must adhere to the following three rules:

- (1) Separate index variables must be used.

```
Thus  DO  100  J=1,10
      DO  200  K=1,10
      Y=J+K
      200 CONTINUE
      100 CONTINUE
```

is valid, while

```
      DO  100  K=10,100
      DO  200  K=L,10
      Z(K)=X+K+K
      200 CONTINUE
      100 CONTINUE
```

is invalid.

- (2) All the statements in the scope of an inner DO statement must be also in the scope of the outer DO statement.

Therefore,

```
      DO  10  K=1,10
      DO  20  J=1,20
      DO  30  L=1,5
      30 CONTINUE
      20 CONTINUE
      10 CONTINUE
```

is valid, while

```
DO 10 K=1,20
DO 20 J=1,5
10 CONTINUE
20 CONTINUE
```

is invalid.

- (3) Each DO loop must have a unique object or termination statement. Thus, the following program segment is invalid:

```
DO 10 J=1,10
DO 10 K=1,5
10 X=J+K
```

The programmer should have used the CONTINUE statement as the object of the outer DO loop.

```
DO 10 J=1,10
DO 20 K=1,5
20 X=J+K
10 CONTINUE
```

7.0 SUBSCRIPTED VARIABLES

7.1 Introduction

Subscripted variables are used to relate a group of associated variables. Any ABC 80 FORTRAN variable may be subscripted. However, only single dimension array variables are permitted. The following are examples of subscripted variables.

```
A(7)
CAT (I+J)
SUSANNE (JACK*K)
```

A subscripted variable may be declared by one of the following statements:

```
DIMENSION  variable  name(size)

OR INTEGER  variable  name(size)
INTEGER*1   variable  name(size)
INTEGER*2   variable  name(size)
REAL        variable  name(size)
REAL*4      variable  name(Size)
REAL*8      variable  name(size)
```

where variable name is any valid symbolic name, and size denotes the number of variables reserved and is any positive non-zero constant $\leq 32,767$. It should be noted that the DIMENSION statement automatically declares the array variable to be double precision (2 byte for integer, 8 bytes for real). Per standard FORTRAN conventions, any DIMENSIONed variables beginning with the letters I-N are integer variables while all others are real.

Examples of valid declarations are:

```
(1)  DIMENSION A(60),B(10)
(2)  INTEGER *1 CAT (91)
      INTEGER*2 DOG (917)
```



```
(3)  DIMENSION SAM (79),PHASE (2)
      INTEGER*1 KIRK (20)
      REAL*4     BET (4)
      REAL*8     MAX (14)
```

Examples of invalid declarations are:

```
(1)  DIMENSION A(10)
      INTEGER 2*A(10) (attempt to redefine variable already
                        declared)
(2)  INTEGER*1 A(-7) (negative sign)
```

7.2 Subscripts

Subscripts may be any valid ABC &O FORTRAN expression. The expression may evaluate to either a single or double precision result within the range 1 to n, where n is the declared size of the array. No run time checks are made on the range of the subscript, so the programmer should take care to stay within the declared range.

The following are valid subscripts:

```
(X*4+J)
(Y=3*INPUT(17))
```

7.3 Use of Arrays

The use of subscription is shown in the two programs which follow.

```
(a)  SUBROUTINE FINDSMALLEST (X,N,SMALLEST)
      INTEGER*1 X(100),N,SMALLEST
      SMALLEST = X(1)
      DO 10 J=2,N
      IF (X(J).GE.SMALLEST) GO TO 10
      SMALLEST = X(J)
10  CONTINUE
      RETURN
      END
```

(b) Vector Multiplication Subprogram

```
SUBROUTINE ARRAYMULT (X,Y,Z)
  INTEGER*1 X(100),Y(100)
  INTEGER*2 Z(100)
  DO 10 J=1,100
10 Z(J)=X(J)*Y(J)
  RETURN
END
```

8.0 SUBPROGRAMS

8.1 Introduction

If a programmer finds that some computation recurs throughout his program, he may want to set up a function to carry out the computation. For such cases, the function and subroutine subprograms have been designed.

The main feature of these is that they are compiled independently of the main program. The locally declared variable names are completely independent of the variable names in the main program and in other subprograms because subprograms may have their own data and program sections. In other words, function and subroutine subprograms can be completely independent, yet it is easy to set up communication between the main program and the subprogram(s). This allows a large program to be divided into smaller sections that can be compiled independently. It also allows the subprogram to be used with other main programs, as long as the main program adheres to the conventions required by the subprogram. Also, individual subprograms can be checked out and tested before they are put together by the main program. However, their main advantage is that they prevent duplication of effort whenever a group of statements is summoned in several portions of one program, or in many programs.

8.2 Subroutine subprograms

We will first write a main program and then convert it to a subprogram to illustrate how it may be used in conjunction with another main program. The following example inputs a string of N(50) numbers from port (FE) and outputs the maximum of these to port 00H.

```
DIMENSION A(50)
N = INPUT (0FEH)
DO 2, I=1,N
```

```
2 A(I) = INPUT (OFEH)
  MAX = A(1)
  IF (N.EQ.1) GO TO 7
  DO 6 I = 2,N
  IF (A(I).LE.MAX) GO TO 6
  MAX = A(I)
6 CONTINUE
7 OUTPUT (OOH) = MAX
  STOP
  END
```

The program can be expressed simply as follows: Input N and A, do appropriate computations, and output max.

This program is now converted into a "subroutine subprogram":

```
      SUBROUTINE FINDMAX  (N,A,MAX)
      DIMENSION A(50)
      MAX = A(1)
      IF (N.EQ.1) RETURN
      DO 6 I=2,N
      IF (A(I).LE.MAX) GOTO 6
      MAX = A(I)
6 CONTINUE
      RETURN
      END
```

Note that the input and output statements are no longer present and that a return statement has been added. The input/output function has been assumed by the subroutine statement.

SUBROUTINE FINDMAX (N,A,MAX) identifies the program as a subroutine subprogram. The name FINDMAX is followed by a list of "input-output parameters" enclosed in parentheses and separated by commas. There is no explicit designation as to which parameters are responsible for input and which for output. Since the

subprogram obviously requires values for A and N before it can function, they are designated as input parameters. On the other hand, the subprogram creates an assignment to the variable MAX, hence it is implicitly assumed as an output parameter.

A RETURN statement is employed in a subprogram whenever a STOP would occur in the main program. The subprogram will be executed and when the RETURN statement is encountered, control is transferred back to the first statement following the CALL statement in the main program. This is illustrated in the trivial main program which utilizes the previous subprogram:

```
DIMENSION A(50)
N = INPUT (OFCH)
DO 2, I = 1,N
2 A(I) = INPUT(OFCH)
CALL FINDMAX (N,A,MAX)
OUTPUT(00)=MAX
STOP
END
```

The CALL statement causes control to transfer to the first executable statement in the subprogram FINDMAX . It also defines the input-output parameters using an "argument list" in the subprogram. When the return is encountered on completion of the subroutine, control is transferred back to the main program and the output statement, OUTPUT(00H)=MAX, is the next statement executed.

The names of the entries in the CALL statement argument list need not be the same as those of the subroutine parameter list. They merely have to correspond exactly in number and type. Thus, a valid statement which would be used to call the FIND MAX subprogram could read

```
CALL FINDMAX (Q,R,LARGE)
```

The parameters in the subroutine statement are "formal parameters" which are replaced by the respective values when the CALL statement is executed. Thus in the statement above, N takes on the value of Q, A is the same array as R, and finally LARGE will take on the value produced by MAX in the subprogram. However, it is necessary that the arrays declared in the subprogram have the same dimension as those in the main program.

All variable names in the subprogram are specified only in that subprogram; they are unknown outside the subprogram. Thus in our example, MAX may be used in the main program, or another subprogram, without any confusion. The same holds true with statement numbers in subprograms. They can be identical with statement numbers in the main program or other subprograms and no conflicts result.

Note that a subroutine always has an END statement as its final statement. All the subroutines required by the main program must be placed before the main program, or any other subprogram which they are referenced. Each of the subroutines as well as the main program is known as a "program segment". Remember that each "job" has only one main program but may have as many subroutines as required.

It is worthy to note that the arguments in the list for the CALL need not be simple variables. They can be any expressions which when evaluated, yield an expression of the proper type. The type is determined according to the type of the corresponding dummy variable in the parameter list of the subroutine. Thus an acceptable CALL statement may look like:

```
FINDMAX (N-1+1,A,MAX)
```

as long as $N-1+1$ yields an integer value corresponding in type to the integer N in the subroutine. The second argument in the call must be an array name, since the corresponding entry in

the subroutine parameter list is also an array name. It would be invalid to use an expression here, since an expression cannot yield an array name as its value.

It is also important to note that a subroutine may call a second subroutine. The second subroutine can call a third subroutine etc. as long as the subroutine does not call itself, either directly or indirectly.

In conclusion, we may state the following rules governing operation of subroutines, namely:

- (1) The general form of the subroutine statement is

```
SUBROUTINE Subroutine Name (Formal Parameter List)
or
SUBROUTINE Subroutine Name
```

The name can be any valid ABC 80 FORTRAN name. No formal parameter list is required if there is no interchange of values between the main program and subprogram.

- (2) Each Subroutine subprogram begins with a Subroutine statement and finishes with an END.
- (3) The subroutine subprogram is referenced by the CALL statement which has the general form

```
CALL Subroutine Name (Argument List)
or
CALL Subroutine Name
```

- (4) When the subprogram uses one of its dummy variables, it is actually using the value of the corresponding variable or expression in the calling argument list. Thus the argument and parameter lists provide a two-way means of communication between the calling program and the called subprogram. When an expression or subscripted variable is used, the value is passed but not returned.

- (5) A RETURN transfers control back to the calling program.
- (6) All variable names and statement numbers in a subroutine are local to that subprogram unless they are declared global.
- (7) A subroutine subprogram can call other subprograms except itself. Thus no recursion is permitted.

8.3 Function Subprograms

These are very similar to subroutine subprograms. The general form of the declaration is:

Function Type FUNCTION Function Name (Parameter List)
or

Function Type FUNCTION Function Name

The function type can be INTEGER*1, INTEGER*2, REAL*4 or REAL*8. If its declaration is omitted, then it is assumed to be of type INTEGER*2. The function name may be any valid ABC 80 FORTRAN variable name. The parameter list is identical to that described in the previous section on subroutines, although a parameter list may be omitted as previously discussed.

In fact, function subprograms are quite similar to subroutine subprograms. The following example will point out the differences:

```
FUNCTION SUM(N,X)
  INTEGER*2 X(50)
  SUM = 0
  DO 6,I = 1,N
6 SUM=SUM+X(I)
  RETURN
END
```


The one major difference between this and a subroutine is that the name of the subprogram, SUM, appears as a variable within the subprogram and is assigned a value. This is always the case with function subprograms but never with subroutine subprograms.

The other major difference is the way in which the function subroutine is called. The CALL statement is never used here, but always used in subroutine subprograms. Instead, it is called simply by writing the name of the function in an expression, along with an appropriate argument list. The following is a main program which calls SUM:

```
      INTEGER*2 A(50),B(50)
      N=INPUT(1CH)
      M=INPUT(27)
      DO 2 I=1,N
2    A(I)=INPUT(29)
      DO 3 I=1,M
3    B(I)=INPUT(30)
      AVG=(SUM(N,A)+SUM(M,B))/(M+N)
      OUTPUT (00) = AVG
      STOP
      END
```

This program reads two vectors and computes their composite average, AVG. As can be seen function subprograms are useful when a single value is to be returned, since the call is included as part of an expression rather than as a separate statement.

In order to define a single precision function, the function declaration must include the appropriate type specification, otherwise a double precision function is assumed. Thus, single precision functions can be defined as:

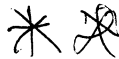
```
      REAL*4      FUNCTION DSQRT(X)
      INTEGER*1    FUNCTION SUM(N,X)
```

The transfer of control from a function subroutine occurs after the RETURN statement which returns control to the point in the expression which contains the function name and argument list. The name assumes the value assigned to it in the subprogram, and computation of the expression is resumed.

Function subroutines normally return only one value to the calling program. However, they may return several by assigning values to the dummy variables, just as in the case in subroutine subprograms.

The function subprogram must be declared prior to its reference in either the main program or other subprograms. Within the body of the function subprogram declaration, the function name can be used as a variable. However, after the declaration has been completed, the function name cannot be used on the left hand side of an assignment statement.

8.4 Global Variables



Global variables refer to variables which are common to more than one segment of a program, i.e. mainline, subroutines, and functions. A variable may be declared global by placing its declaration ahead of all other program segments in which it is to be referenced. The following example illustrates how the array A may be declared as a 50 byte global array for the entire program:

```
C   PROGRAM START
C   DECLARE A(50)
C
C   INTEGER*1 A(50)
C
C   SUBROUTINE PROGRAM SEGMENTS GO HERE
C
```

.
.
.

```
C    PUT MAINLINE PROGRAM HERE
      .
      .
      .
      STOP
      END
```

8.5 Global Subprograms

Just as a hierarchy of global variables exists, subprograms also have a hierarchy. The following rule must be followed, namely, a subprogram segment must appear ahead of any reference to that same subprogram. Thus, if a SQRT function is used in subroutine MAGNITUDE, which in turn is used by subprogram THETA, then these subprograms must be arranged in the sequence shown below in order to be compiled correctly.

```
C    PROGRAM START
      INTEGER 2  FUNCTION SQRT (X)
      .
      .
      .
      END

C
      SUBROUTINE MAGNITUDE (X,Y)
      .
      .
      .
      Z = SQRT ( X+Y Y)
      RETURN
      END

C
C
      SUBROUTINE THETA (X,Y)
      .
      .
      .
```

CALL MAGNITUDE (X,Y)

.

.

.

RETURN

END

C

C MAINLINE PROGRAM

.

.

.

STOP

END

9.0 FLOATING POINT OPERATIONS

9.1 Single and Double Precision Constants

Two types of floating point constants are available in ABC 80 FORTRAN, namely, single and double precision. These numbers are stored according to IBM 370 standard; namely, a signed exponent followed by a 3 or a 7 byte mantissa. The dynamic range of both is approximately

$$.5397665 \times 10^{-78} \text{ to } .7237005 \times 10^{76}$$

Single precision constants provide 7 significant digits while double precision provides 16 digits. Thus, the programmer should select the type which provides the required precision while bearing in mind the 2 to 1 difference in storage requirements.

An 'E' is used to indicate a floating point constant will be single precision. A 'D' is used for all double precision constants. A single precision default is made if the exponent indicator is not on the number.

Some valid floating point constants are:

1.269	- single precision
-.001431E02	- single precision
10.2103E-62	- single precision
10.	- single precision
12.45123198156011D0	- double precision
10.D11	- double precision

The following are invalid:

1	- no decimal point
3110E82	- exponent too large
-1.23E-82	- exponent too small

9.2 Declarations

Floating point variables may be declared by means of two declarative (non-executable) statements.

```
REAL*4    variable list
REAL*8    variable list
```

The first statement is used for single precision (4 byte) variables, while the second may be used to declare double precision variables. It should be noted that these variables may be simple variables or single dimensional arrays.

ABC 80 FORTRAN follows the FORTRAN IV convention in that it assumes symbolic names that do not begin with the letter I through N are REAL*4 variables. Thus, the following would be REAL*4 variables unless another declaration is present:

```
TOPOFSTACK
BOB
SUE
```

9.3 Floating Point Arithmetic

ABC 80 FORTRAN provides the same real operators as those it provides for integer operations, namely, +, -, *, / and unary-. These symbols represent addition, subtraction, multiplication, division and change of sign respectively. Formulae or expressions are built up using any combination of these operators with appropriate variables or constants. Real arithmetic follows the same priority rules as those of integer arithmetic. (see Section 2.4).

Mixed mode arithmetic is not allowed. This means that real and integer quantities cannot be mixed in an expression. Conversions between types is done using the assignment operator.

Example:

```
INTEGER*1  I
INTEGER*2  J
REAL*4     A
REAL*8     B
```

```
I = B+A*2.0      - converts real result to integer
B = I/J*2        - converts integer result to real
```

It should be noted that calculations using both REAL*4 and REAL*8 types will be performed in REAL*8.

The user is cautioned not of used mixed mode expressions, i.e.

```
X = A + M
```

or

```
X = A + 1
```

9.4 Floating Point I/O

Real numbers may be input using the READ statement just as in the case of integer numbers. Once again, commas are used as delimiters between numbers.

The outputting of real numbers may be accomplished by means of the formatted WRITE statement. Once again its operation is the same as in the case of integer I/O. However, E or F format must be used in the FORMAT specification statement. Also, real constants may be included in the parameter list of the WRITE statement.

An example of floating point I/O is shown below:

```
READ (1) X, Y, Z
Z = X*Y/Z
WRITE (1, 10)X, Y, Z, -2.5639E-03
```

9.5 Floating Point Format

Two format codes are available for use with the floating point option.

These are:

- Fw.d - total field width of 'w' with 'd' decimal positions to the right of the decimal point. Note: No exponent is printed.
- Ew.d - total field width of 'w' with 'd' significant digits. Note: Since exponent is printed, $w \geq d + 7$.

Thus, when the magnitude of real numbers is small and well defined, the F format should be used, while in all other cases the E format should be employed.

The following program will generate the indicated line of output:

```
C      DEMONSTRATE E and F formats
      X = 1725.683
      Y = -12.7316
      Z = -9293.69143184 E-08
      WRITE (1, 10) X, Y, Z
10     FORMAT ('', F8.2, 2X, E10.3, 2X, E14.6)
      STOP
      END
```

Outputted line

1725.68-0.127E02 - 0.929369E-08

It should be noted that if the field specification is too small for the number involved (e.g., 256.12 with F5.2 format) then asterisks will be printed.

10.0 FILE I/O

10.1 Introduction

Apart from the direct input/output access method using INPUT and OUTPUT statements two other access modes are available to the ABC 80 FORTRAN user - record I/O and formatted I/O.

The record I/O method allows access to a file in a random fashion, i.e. the data can be accessed in any order.

The formatted I/O method is useful for sequential input or output of data such as integers, real numbers and strings.

10.2 The INCLUDEIO directive

In order to establish communication between a program and peripheral devices the ABC 80 FORTRAN includes a set of system subroutines to perform various input/output functions. These subroutines are made available to the programmer via the ABC 80 FORTRAN keyword INCLUDEIO. If the program is to perform I/O this directive should be included in the program before any user defined subroutines or function but after any global variables as shown in this example

```
C      FIRST DEFINE PROGRAM LOCATION
      COMPILER(1)=08000H
      COMPILER(3)=0BFFFH
C      GLOBAL VARIABLES
      INTEGER*1 FLAG1,FLAG2,GLOBE
C      LOAD THE SYSTEM SUBROUTINES
      INCLUDEIO
C      USER WRITTEN SUBROUTINES
      .
      .
      .
      MAIN PROGRAM
      .
      .
      .
```

The I/O subsystem consists of four user callable subroutines - OPEN, CLOSE, GET and PUT.

10.3 The OPEN subroutine

This subroutine is called to enable access either to an existing file or to create a new file. The format is:

```
CALL OPEN (LUN,FNAME,MODE,STATUS)
```

the parameters should be:

LUN	Logical unit number in the range 1-8 by which all further references to the file will be made.
FNAME	An INTEGER*1 array containing the name of the file or device in ASCII terminated by either a blank or a null byte.
MODE	Either 0 to open an already existing file or 1 to create a new file similar to BASIC's 'prepare'.
STATUS	An INTEGER*1 variable which will be set to a return status by the I/O subsystem. This will be 0 if the call was successful, or one of ABC 80's internal error codes in case of an error.

Once a file has been OPEN:ed the program may perform formatted (READ, WRITE) or record I/O (GET,PUT) accesses to it.

To facilitate communication through the ABC 80 console the system initially opens this device as logical unit 4, thus removing the need of explicitly calling the OPEN subroutine.

The OPEN subroutine requires a filename specification. The following example shows a convenient way of defining the filename and how to open files.

C OPEN CONSOLE AND A DISC FILE

```
INTEGER*1  CON(G)/'CON:  '/
INTEGER*1  DATF(12)/'SAMPLE.DAT  '/
INTEGER*1  STATUS
      .
      .
      .
CALL OPEN(1,CON,0,STATUS)
CALL OPEN(2,DATF,1,STATUS)
```

10.4 The CLOSE subroutine

This subroutine is used to close a file in a proper way when no more accesses should be done. The format is

```
CALL CLOSE(LUN)
```

The parameter is

LUN Logical unit number of the file to be closed.

10.5 The GET and PUT subroutine

These subroutines may be used to perform unformatted random access to ABC 80 files. The amount of data transferred upon calls to these routines is equal to the record length of the corresponding device. The format is

```
CALL GET(LUN,RNUM,BUFFER,STATUS)
CALL PUT(LUN,RNUM,BUFFER,STATUS)
```

GET reads data from the file and PUT outputs data to the file. The parameter should be

LUN Logical unit number of the file

- RNUM** An INTEGER*2 expression evaluating to the logical number of the record to be accessed. A value of -1 may be used to access the file in a sequential manner. If the file is not a random access file (like the console, 'CON:') this parameter is ignored and the access will always be sequential.
- BUFFER** An INTEGER*1 array to be used as data area for the record. The array must be large enough to hold the entire record. For disc files the record size is 253 bytes. For the console, 'CON:', the record size is 1 byte.
- STATUS** An INTEGER*1 variable which will be set to a nonzero return value if any errors occurred during the access. The value will then be one of ABC 80's internal I/O error codes.

This example shows the use of GET and PUT to copy a disc file record by record:

```
10      INUM=0
        CALL GET(F1,INUM,BUFFER,STATUS)
        IF (STATUS.EQ.38) GOTO 20
        CALL PUT(F2,INUM,BUFFER,STATUS)
        INUM=INUM+1
        GOTO 10
20      CALL CLOSE(F1)
        CALL CLOSE(F2)
```

10.6 The WRITE Statement

This statement is used to output or 'write-out' a sequence of variable expressions and/or strings. Its general form is

```
WRITE (a,b) list
```

where 'a' is the output device number, 'b' is the number of the FORMAT statement to be used in executing the WRITE statement, and 'list' is a series of parameters (expressions or implied DO loops) delimited by commas.

NOTE: Only implied DO loops may be enclosed by parentheses, although parentheses may be used within an expression.

Each item in the list is output according to the specified FORMAT statement. It is worth noting the ability to place expressions (including constants) in the list. This can further reduce the number of statements in a program. Some examples of valid WRITE statements are:

```
WRITE (2,2) APPLE,SUE, BOB  
WRITE (4,701) CAT = DOG +7, I = I+1, 18
```

Arrays may be output by using an implied DO loop in a WRITE statement. These have the form:

```
WRITE (a,b) (Array(Index),Index=Init,Final,Incr)
```

where Array is a one dimensional array variable;
 Index is any valid symbolic name;
 Init is the initial value of the index variable;
 Final is the upper bound the Index may assume;
 Incr is the amount by which the index is incremented
 on each pass;

NOTE: Enclosing parentheses must be present.

Some examples follow:

```
WRITE (1,17) A, (B(J),J=1,17),(C(J),J=1,14)  
WRITE (2,10) (LIST DATA(I),I=7,49,7)
```

10.7 User defined output devices

In order to operate, the WRITE statement requires a character oriented output routine. If the user wishes to use an output routine that is not referenced by the unit numbers, an absolute address or expression may be used in place of the unit number, e.g.,

I=3

WRITE(0E81EH+I,10)

would cause the output program to call the character oriented device subroutine at address 0E821H. The output program places the character in the C register and assumes the H,L,D,E,B, registers are unchanged when control is returned.

10.8 The READ Statement

This statement allows the user to input a series of values from the console device. It has the general form:

READ (b,ERR = n) list
or READ (b) list

where b is the input device number;
n is the statement number of the error handler, and
list is the series of variables to be input.

The execution of this statement causes an input subroutine to be called which fills a buffer in memory. The user may type in the values to be assigned to each of the variables in the list. Commas are used as delimiters. The READ statement allows editing capability on the input record by deleting and echoing back rubbed out characters.

An example of a READ statement and the input data which may be entered is shown below.

READ (4,ERR=10) A, PROCESS 2,TOP
(data typed on console) 17,14179,0 (carriage/cursor return)

As indicated above, a CR denotes the end of the input data sequence and causes the input buffer values to be assigned to the corresponding variables.

An array of values may also be input using a READ statement and implied DO loop.

```
                READ (4,ERR=10)(A(I),I=1,4)
(data typed on console) 15,21,-37,-126(CR)
```

Character arrays may be input without leading blanks by using the supplied function STRING in conjunction with a READ statement.

Its two parameters specify the array name and the length of the string to be input. This function automatically ignores leading blanks and pads with trailing blanks. An example follows:

```
Ø = blank                READ (4,ERR=20)STRING(A,5)
(data typed on console) ØØEDØ(CR)
```

will result in the array A containing

```
EDØØØ
```

If the string is longer than the specified size, then it is truncated. Also, since the input buffer is only 80 characters, the string function second operand should be a single byte expression. If it is not, it is truncated.

Commas are normally delimiters on input; thus inputting a comma as an element of a string as a special case. This is handled by enclosing the string in quotes. In this case all leading and trailing blanks, plus embedded commas enclosed in quotes are included as part of the string.

Consider the previous example with "ØE,DØ"(CR) typed on the console. The array will contain:

```
ØE,DØ
```

10.9 User defined input devices

(a) Console device

The user can access a device by providing the address of the service routine. The read statement will allow line editing on an "interactive" device. In order to support this feature, the routine must be able to echo characters. The following conditions must be serviced by the character oriented routine.

- (1) When the routine is called and the CARRY flag is set, then the routine should output the character in the C register to the device.
 - (2) When the routine is called and the CARRY flag is reset, then the routine should input a character from the device and place it in the A register.
 - (3) All registers should be unchanged except for the PSW.
- NOTE:
- a rubout (7FH) will cause the previous character to be echoed and removed from the input buffer.
 - a carriage return (ODH) will indicate an end of input can cause a carriage return to be echoed back.
 - a line feed will be echoed at the beginning of the READ.

10.10 The FORMAT Statements

The general form of this statement is:

$$n \text{ FORMAT}(S_1, S_2, \dots, S_m)$$

where n is the statement number (must be present), and S_1, S_2, \dots, S_m are format codes.

The following format codes are available in ABC 80 FORTRAN:

- wHxx. . . - string field of width W
 - 'string' - hollerith
 - nX - "n" spaces
 - / - skip to next line
 - Iw - integer field of width w
 - Aw - alphanumeric field of width w
 - Zw - hexadecimal field of width w
 - Fw.d - floating point field of width 'w' with
'd' decimal positions to the right of the
exponent is printed.
 - Ew.d - floating point field of width 'w' with 'd'
significant digits.
- Note: Since exponent is printed, $W \geq d + 7$.

Any combination of these may be used in a FORMAT statement.
These are some examples:

```
10 FORMAT (10X,'A=',12,5X,'B=',16)
225 FORMAT (10X,A3,12,5X,A3,16)
16 FORMAT (1X,Z4,/,1X,Z4)
```

A format code may be repeated using a field count, e.g.

```
10 FORMAT (312,2A3)
```

If any values remain after all format codes have been scanned, then it will repeat the FORMAT statement until all the data has been output.

If the first element of an output line is a hollerith string, then the first character will be interpreted as a carriage control, otherwise single spacing is assumed.

The following characters may be used:

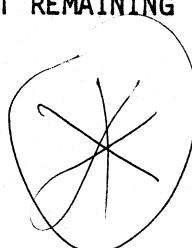
blank	-	single spacing
0	-	double spacing
-	-	triple spacing
1	-	skip to top of page (clear screen)
+	-	carriage return, but do not advance

Any other character will be ignored and result in single spacing.

NOTE: The first character of each formatted output line is a print control character. For line buffered devices, a carriage return will be needed to print the last line of output. This can be accomplished by using slash format before the program exists. For character oriented devices, the characters are printed as they are received so slash format will not be required in this case.

Example:

```
C      OUTPUT TO A LINE BUFFERED PRINTER
.
.
.
C      END OF PROGRAM - PRINT REMAINING CHARACTERS IN
      BUFFER
      WRITE(2,3)
3      FORMAT('-', '+')
      STOP
      END
```



11.0 CHARACTER STRINGS

So far, we have dealt strictly with numerical and logical computation. However, often alphabetic characters are used in calculations. For example, to order the names BOB and BILL alphabetically, the two alphabetic "strings" must be compared.

ABC 80 FORTRAN has some features which make such character manipulation possible. Each byte in the computer memory is capable of storing one ASCII character. Hence, one character can occupy the space used by a single precision integer variable. In fact, a common way to store character strings is a values assigned to integer variables, particularly arrays. Then the variable name becomes the name of the character string. In the following declaration

```
INTEGER*1 A/'B'//,M/18/
```

A and M are both declared to be integers. A is initialized to represent the character B and M has the value 18. A character string must be enclosed by quotes. If two successive quotes are found, then a quote is inserted in the string at that point.

e.g. 'IT'S TRUE'

If the string required is longer than 1 character an array is the easiest way to store it. For example, we can store the phrase "the rain in Spain" in a subscripted variable of dimension 17:

```
INTEGER*1 A(17)/'THE RAIN IN SPAIN'/
```

The result of such a declaration is to assign the following characters to the variable:

```
A(1) ='T'  
A(2) ='H'  
A(3) ='E'  
ETC.
```

If the variable A is then printed out using a 17A1 format. then the correct string will be printed.

Declaration statements may be used to initialize character variables prior to execution time. The following examples show how this may be done:

```
INTEGER*1 A/'S'/  
INTEGER*2 A/'ST'/
```

However, it must be remembered that initializations done by declaration statements (such as INTEGER) are done by the compiler at compile time. These values should be loaded before execution or else stored in PROM with the program.

Character strings are valuable in printing titles or explanatory notes when the results of a program are printed out, and they are also useful in 'comparing' character strings, that is, arranging them alphabetically. The following program illustrates how N, two letter words, may be stored, assuming the words have already been read in as elements of the double-precision array WORD(J).

```
DO 5 I=1, N-1  
DO 10 J=I,N  
IF(WORD(J).LE.WORD(J+1)) GO TO 10  
K=WORD (J)  
WORD(J) = WORD(J + 1)  
WORD(J + 1) = K  
10 CONTINUE  
5 CONTINUE
```

12.0 SPECIAL FEATURES

12.1 Inline Code:

ABC 80 FORTRAN allows user to easily link machine code into their FORTRAN source program. Although ABC 80 FORTRAN produces reasonably efficient code, situations arise where machine language coding is the only way to obtain the required level of throughput.

A special statement is available to facilitate use of the feature. Its general form is

```
INLINE /00H byte string of machine code statements/
```

All statements begins with 00H. 

The machine code statements have each of their bytes separated by a comma. For example, the following INLINE statement would decimally adjust the contents of memory location 0FC1H:

```
INLINE /00H,21H,0C1H,0FH,7EH,27H,77H/
```

and is equivalent to the assembly sequence

```
LX1    H,0FC1H
MOV    A,M
DAA
MOV    M,A
```

Very often, it is desirable to access variables, statement numbers or subroutines when using the INLINE code feature. This is accomplished by the supplied function ADDRESS (parameter). The parameter of the function may be a symbolic name, subroutine name or statement number. ADDRESS returns a two byte value which corresponds to the actual memory location of the function argument. For example, suppose instead of location 0FC1H in the previous example, we wish decimally to adjust the single precision integer variable SUE. The code for this routine might be written as follows:

```
      INLINE  /00H,21H,ADDRESS(SUE),7EH,27H,77H/
```

which is equivalent to the assembler program

```
      LXI     H,SUE
      MOV     A,M
      DAA
      MOV     M,A
```

The following example outputs 0H1H to port 7FH and jumps immediately to statement number 120 in the same program.

```
      INLINE  /00H,3EH,0E1H,0D3H,7FH,0C3H,ADDRESS(120)/
```

The scope of the ADDRESS function is related to the program segment in which it occurs. Thus, an ADDRESS call in a subroutine which has a statement number as its argument will return the address of that statement number within the subroutine body.

Particular attention should be paid when using INLINE code in FUNCTION subprograms in that use of the function name as the ADDRESS argument will yield the function location, not the function subprogram location.

12.2 Accessing Compiler Directives

In order to allow the user to locate his program and variables anywhere in the Z80 space memory, a special set of variables, COMPILER (1) is provided. The general form of assigning such a variable is

COMPILER (n) = CONSTANT

where (n) specifies which set of compiler is affected and the CONSTANT is a double byte integer. The three possible choices for 'n' have the following results:

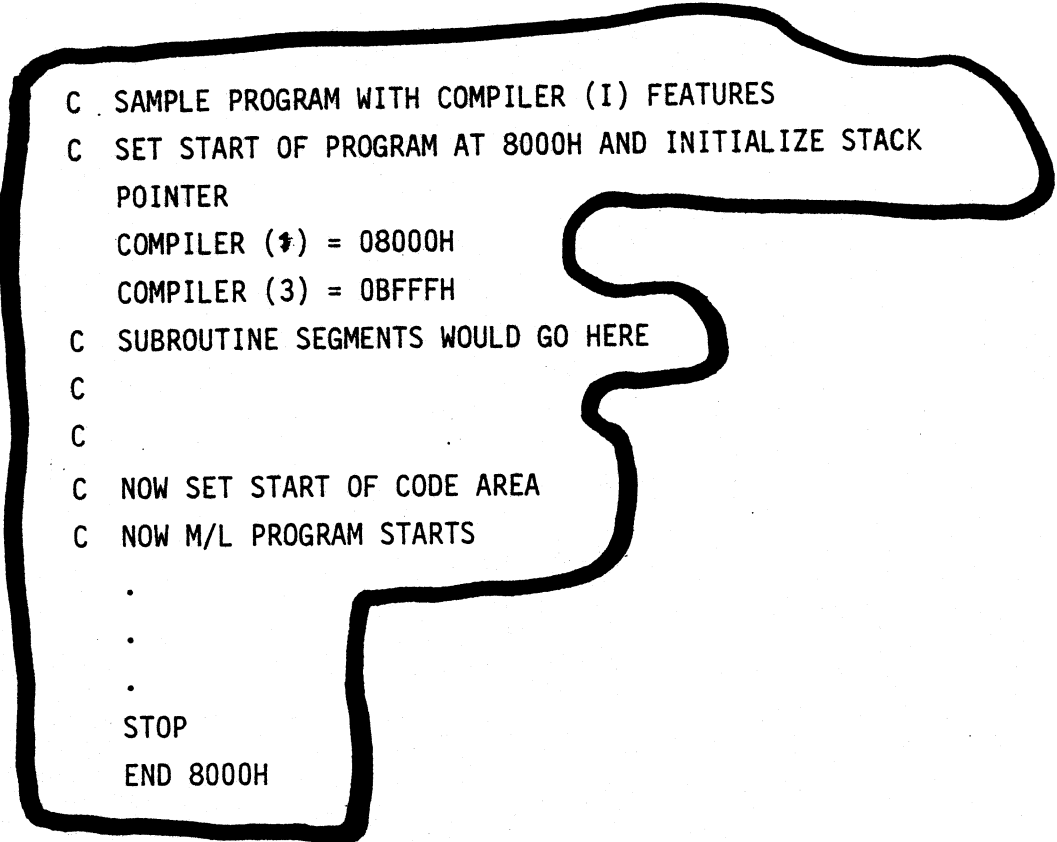
n=1 allows the user to set the start address of the code specified by constant. It initializes the stack pointer and a jump to the start of the mainline program;

n=2 allows the user to set the start address where generated code is to be located. It is useful to skip over non-existent memory blocks or reserved areas of memory.

n=3 allows the user to set the top address of the RAM memory space.

In order to return to the command interpreter upon encountering a stop statement the program should be located so it does not modify locations 0C000H through 0C6FFH.

The following program illustrates how the COMPILER feature is used:



```
C . SAMPLE PROGRAM WITH COMPILER (I) FEATURES
C SET START OF PROGRAM AT 8000H AND INITIALIZE STACK
  POINTER
  COMPILER (1) = 08000H
  COMPILER (3) = 0BFFFH
C SUBROUTINE SEGMENTS WOULD GO HERE
C
C
C NOW SET START OF CODE AREA
C NOW M/L PROGRAM STARTS
  .
  .
  .
  STOP
  END 8000H
```


APPENDIX "A"

ABC80 FORTRAN

The following list of reserved words may not be used as symbolic names in a DATABOARD FORTRAN program.

ADDRESS	FORMAT	
CALL	FUNCTION	READ
CARRY	GO	REAL
CARRYOFF	GO TO	RETURN
COMPILER	IF	SIGN
	INCLUDEIO	
CONTINUE	INLINE	SIGNOFF
DIMENSION	INPUT	STOP
	INTEGER	STRING
DO		SUBROUTINE
	OUTPUT	TO
END	PARITY	WRITE
ERR	PARITYOFF	ZERO
		ZEROOFF

APPENDIX "B"ERROR MESSAGESSyntax

- 00 - undecodable statement
- 01 - expecting constant; found none
- 02 - expecting constant; found none
- 03 - expecting identifier or constant; found none
- 04 - expecting statement number; found none
- 05 - expecting equal symbol; found none
- 06 - expecting comma; found none
- 07 - expecting right bracket; found none
- 08 - expecting end of statement; found another symbol
- 09 - expecting left bracket; found none
- 0A - invalid use of keyword
- 0B - expecting slash; found none
- 0C - invalid use of equal operator

Card Format and Contents

- 10 - columns 1 - 5 of a continuation card are not blank or invalid characters found in statement number field.
- 11 - illegal character used in statement
- 12 - the first symbol of a statement is not an identifier
- 13 - invalid characters in columns 1 - 5; probable cause --statement is left of column 7
- 14 - statement record must be greater than 7 characters.
- 15 - length of input record must not exceed 72 characters; use continuation card instead.

- 16 - input buffer overflow; use shorter symbols and avoid using unnecessary blanks if this error occurs.

Sub-program

- C0 - missing subprogram (i.e. subroutine has not been declared).
- C1 - expecting array name in parameter list.
- C2 - too many arguments in subprogram reference.
- C3 - subprogram previously defined first reference used.
- C4 - illegal or blank subprogram name.
e.g. SUBROUTINE (A,B)
- C5 - subroutine used as function
- C6 - missing parameter list.
- C7 - illegal use of array name in parameter list.
- C8 - precision of arrays do not match.

Input/Output

- E0 - invalid expression for unit number in READ/WRITE
- E1 - statement number in I/O statement is not a FORMAT statement number.
- E2 - missing or invalid FORMAT statement number I/O statement.
- E3 - invalid I/O list.
- E4 - expecting array name in implied DO.

- E5 - invalid implied DO loop.
- E6 - no input list for READ statement.
- E7 - expecting array or variable name in string input function.
- E8 - invalid string input function.
- E9 - OUTPUT function must be followed by an equals operator.
- EA - INPUT function cannot be followed by an equals operator.
- EB - INPUT or OUTPUT operation must have a single fixed integer constant for port number.
- EC - expressions or constants are not allowed in READ input list.

Constants:

- 20 - integer constant has character out of range of base.
i.e. 1010201B.
- 22 - illegal use of decimal point.
- 23 - statement number greater than 99999 or otherwise invalid.
- 24 - null string found.
- 25 - no closing quote or next line not a continuation in string constant.
- 26 - bad end of record found (i.e. expecting CR,LF).
- 27 - expecting exponent value.
- 28 - mixed mode arithmetic.
- 29 - logical operand is a Float variable.
- 2A - illegal use of Float variable.

Compiler Errors

- 30 - phase error; statement number has changed between passes.
- 31 - symbol table overflow (all variable memory used).
- 32 - internal stack overflow; either DO loop nesting too deep or expression too complex to analyze.

Declaration Statement

- 40 - length specification in declaration statement out of range
e.g. INTEGER*4.
- 41 - expression not allowed in array size declaration
e.g. DIMENSION A(5+6)
- 42 - no dimension specified for variable in DIMENSION statement
e.g. DIMENSION A,B(5),C(5).
- 43 - expecting integer constant for array size declaration
e.g. INTEGER*1 A,B,C
DIMENSION Z(A)
- 44 - attempt to redefine variable previously declared.
e.g. INTEGER*1 A,B,C,A

DO Loops

- 50 - object of DO loop has already appeared
e.g. 10 A=B+C
DO 10 I=1,5
- 51 - object of DO loop must be unique
e.g. DO 10 I=1,5
DO 10 J=1,4
10 A=B+C

improperly nested DO loop

```
      DO 10 I=1,5
        DO 5 J=1,4
          10 A = B + C
          5 C = D + E
```

- 52 - invalid DO loop parameter list
e.g. DO 10 I=1
missing upper bound of loop.
- 53 - initial value of loop defined improperly.
e.g. DO 10 I= (I+X)), 10
- 54 - upper value of loop defined improperly.

General Errors

- 60 - operand for the ADDRESS keyword must be either a number or a user defined symbol e.g. INLINE/OC3H, ADDRESS (A+5)/.
- 63 - constant out of range for compiler directive.
e.g. COMPILER (4) = 1000H
- 64 - expecting constant address for compiler directive.
e.g. COMPILER (3) = A+1
- 65 - missing end statment for subroutine or function
e.g. SUBROUTINE AB C(X,Y,Z)
 X+Y+Z
 RETURN
 FUNCTION TEST (A)
 TEST = A 10
 RETURN
 END

- 66 - return not valid in main program.
- 67 - multiply defined statement number.
- 68 - invalid data initialization statement.
- 69 - operand of address function must be previously declared if not a statement number.

FORMAT statement

- 70 - FORMAT statement syntax error.
- 71 - invalid field count or field specification.
e.g. 10 FORMAT (1X,I5,5X,I)
- 72 - no closing quote in string format.
- 73 - '03' value not allowed in quoted string of format.
- 74 - no statement number on format.
- 75 - field count or field specification greater than 127.
- 76 - missing decimal point in F or E format.
- 77 - field width and decimal specification are incompatible.

IF Statement

- A0 - statement invalid after logical IF;
e.g. IF(A.GT.B) IF(C) GO TO 10
- A1 - invalid expression in logical IF;
- A2 - cannot use RETURN in logical IF in an interrupt subroutine.

- A3 - invalid statement list in arithmetic IF.
- A4 - arithmetic IF statement cannot be last statement in DO loop.
- A5 - statement number not found on the statement following arithmetic IF.

APPENDIX "C"ABC 80 FORTRAN FEATURES NOT SUPPORTED BY
FORTRAN IV

1.0 Additional Statements/Functions

COMPILER (1)

COMPILER (2)

COMPILER (3)

INPUT (PORT)

OUTPUT (PORT)

ADDRESS

INLINE

2.0 Symbolic Names - up to 31 characters long.

3.0 Other Features - expressions may contain equals operator.

APPENDIX "D"FORTRAN IV FEATURES NOT SUPPORTED
BY ABC 80 FORTRAN

1.0 Statements and Functions

EQUIVALENCE

COMMON

DATA

NEW

ENCODE

DECODE

ENDFILE

BACKSPACE

REWIND

PRINT

PUNCH

PAUSE

STATEMENT FUNCTION

2.0 Arrays - multiple dimensional arrays not supported.

3.0 Data Types - maximum 2 byte for integer data type.

4.0 FORMAT - repeated format not allowed.
e.g. FORMAT (1X,5(15,F6.1))

APPENDIX "E"
SAMPLE PROGRAM

The following program shows an example of the capabilities of ABC 80 FORTRAN and demonstrates the use of subprograms and direct I/O.

```

0000 C      PROGRAM LOCATION DEFINITIONS
0000 C
0000      COMPILER(1)=08000H
8006      COMPILER(3)=0BFFFH
8006 C
8006 C      PRINT ONE LINE ON UART
8006 C      *****
8006 C
8006 C      SUBROUTINE OUTLINE(LINE)
8006 C
8006 C      INTEGER*1 LINE(2),INDEX,SIZE
8006 C
8006 C      SIZE=LINE(1)
8011      OUTPUT(1)=75Q
8015      DO 20 INDEX=2,SIZE+1
8020 10     IF ((INPUT(1).AND.2).EQ.2) GOTO 10
8029      OUTPUT(0)=LINE(INDEX)
8036 20     CONTINUE
8041      RETURN
8042 C
8042      END

```

PROGRAM STORAGE= 0060

VARIABLE STORAGE= 0005

SYMBOL TABLE

```

8020 10
8036 20
BFFC SIZE
BFFD INDEX
BFFE LINE

```

```

8042 C
8042 C      SUBROUTINE TO READ A LINE
8042 C      *****
8042 C
8042      SUBROUTINE INPLINE(LINE)
8042 C
8042      INTEGER*1 LINE(2),INDEX,SIZE,CHAR
8042 C
8042      OUTPUT(1)=75Q
8046      OUTPUT(2)=0
804A      SIZE=LINE(1)
8055      DO 20 INDEX=2,SIZE+1
8061 10    IF ((INPUT(1).AND.200Q).EQ.200Q) GOTO 10
806A      CHAR=INPUT(0).AND.177Q
8071      OUTPUT(0)=CHAR
8076      OUTPUT(2)=0
807A      LINE(INDEX)=CHAR
8088      IF (CHAR.EQ.15Q) GOTO 30
8090      IF (CHAR.EQ.12Q) GOTO 30
8098 20    CONTINUE
80A3 30    LINE(1)=INDEX-1
80B0      RETURN
80B1 C
80B1      END

```

PROGRAM STORAGE= 0111

VARIABLE STORAGE= 0006

SYMBOL TABLE

```

80A3 30
8061 10
8098 20
BFF6 CHAR
BFF7 SIZE

```

1 CHAR
2 CON
3D PUT
J28 GET
01F CLOSE
300A OPEN
8006 IOLINK

BFF8 INDEX
BFF9 LINE

```

80B1 C
80B1 C      MAIN PROGRAM
80B1 C      *****
80B1 C
80B1      INTEGER*1 HEAD(14)/13,15Q,12Q,'UART-TEST',15Q,12Q/
80B1      INTEGER*1 ECHO(14)/13,15Q,12Q,'THE ECHO:',15Q,12Q/
80B1      INTEGER*1 VECTOR(81)
80B1 C
80B1 10     CALL OUTLINE(HEAD)
80BA      VECTOR(1)=80
80BF      CALL INPLINE(VECTOR)
80C6      IF (VECTOR(2).EQ.'*') GOTO 900
80CF      CALL OUTLINE(ECHO)
80D7      CALL OUTLINE(VECTOR)
80E0      GOTO 10
80E3 C
80E3 900    STOP
80E6 C
80E6      END 08000H
    
```

PROGRAM STORAGE= 0053

VARIABLE STORAGE= 0109

TOTAL PROGRAM STORAGE= 0230

TOTAL VARIABLE STORAGE= 0120

SYMBOL TABLE

```

80E3 900
80B1 10
BF88 VECTOR
    
```


** ABC-80 FORTRAN COMPILER V1.1 **

BFD9 ECHO
BFE7 HEAD
8042 INPLINE
8006 OUTLINE

```

0000 C
0000 C      PROGRAM FOR SHOWING
0000 C      CONTENTS OF A TEXT FILE
0000 C      USING BLOCKED ACCESSES.
0000 C
0000 C      DEFINE PROGRAM LOCATION
0000 C
0000      COMPILER(1)=8000H
8006 C
8006 C      DEFINE RAM TOP
8006 C
8006      COMPILER(3)=9000H
8006      INCLUDEIO
8052      INTEGER*1 CON(5) // CON: '/'
8052      INTEGER*1 CHAR(1), STATUS
8052      INTEGER*1 FNAM(20)
8052      INTEGER*1 BUFFER(256)
8052      INTEGER*2 RECNUM
8052 C
8052 C      OPEN CONSOLE
8052 C
8052      CALL OPEN(4, CON, 0, STATUS)
8071 C
8071 20      WRITE (4, 442)
8079 442      FORMAT(/, '+ENTER FILE NAME >')
8079      READ(4, ERR=25) STRING(FNAM, 19)
808C      GOTO 10
808F C
808F C      BAD FILE NAME
808F C
808F 25      WRITE (4, 445)
8097 445      FORMAT(/, '+FILE NOT FOUND !')
8097      GOTO 20
809A 10      FNAM(20)=' '
809F C
809F C      OPEN INPUT FILE
809F C
809F      CALL OPEN(1, FNAM, 0, STATUS)
80BD      IF (STATUS.NE.0) GOTO 25

```

```

80C5      RECNUM=0
80CB C
80CB C      READ A RECORD
80CB C
80CB 6D      CALL GET(1,RECNUM,BUFFER,STATUS)
80F1      IF (STATUS.NE.0) GOTO 900
80F9 C
80F9      DO 70 I=1,253
80FF      CHAR(1)=BUFFER(I)
810D      IF (CHAR(1).NE.9) GOTO 90
8116      I=I+1
8120 C
8120 C      PROCESS COMPRESSED SPACES
8120 C
8120 65      IF (BUFFER(I).EQ.0) GOTO 70
812D      CHAR(1)=' '
8132      CALL PUT(4,0,CHAR,STATUS)
8150      BUFFER(I)=BUFFER(I)-1
8164      GOTO 65
8167 C
8167 C      IGNORE CHAR IF <= CHR(8)
8167 C
8167 9D      IF (CHAR(1).LE.8) GOTO 80
8170      CALL PUT(4,0,CHAR,STATUS)
818E      IF (CHAR(1).NE.13) GOTO 70
8197 C
8197 C      FOLLOW RETURN BY LINEFEED
8197 C
8197      CHAR(1)=10
8199      GOTO 90
819C 7D      CONTINUE
81AC C
81AC 8D      RECNUM=RECNUM+1
81B6      GOTO 60
81B9 C
81B9 C
81B9 C      END WITH FINAL I/O STATUS
81B9 C
81B9 900     WRITE(4,446) STATUS

```

```
81C7 446   FORMAT(//,'+** END OF FILE ',1I2,' **')
81C7       STOP
81CA C
81CA       END 8000H
```

PROGRAM STORAGE= 0458

VARIABLE STORAGE= 0287

TOTAL PROGRAM STORAGE= 2878

TOTAL VARIABLE STORAGE= 0621

SYMBOL TABLE

```
81AC 80
8120 65
8167 90
8EB8 I
819C 70
81B9 900
80CB 60
809A 10
808F 25
8071 20
8EBA RECNUM
8EBC BUFFER
8FBC FNAM
8FDD STATUS
```