CFD

A FORTRAN-BASED LANGUAGE FOR ILLIAC IV

COMPUTATIONAL FLUID DYNAMICS BRANCH

AMES RESEARCH CENTER
NATIONAL AERONAUTICS AND
SPACE ADMINISTRATION

Version 2.0

A FORTRAN based language for ILLIAC IV

CFD

This manual describes the CFD language. The first chapter gives a brief discussion of the ILLIAC IV hardware and goes on to describe the CFD language in general terms. The second chapter is arranged to provide a quick definition and syntactical reference to the various elements of CFD by means of a box format. Each element is described in terms of its use and syntax and, when appropriate, examples are given.

An appendix gives additional information on the conversion of CFD to FORTRAN.

A technical note gives translator conventions and information about the needed run-time package.

Computational Fluid Dynamics Branch
Ames Research Center
National Aeronautics and Space Administration

CHAPTER 1.

CFD Code: Introduction and Overview

1.1 Objectives, limitations, and future possibilities

CFD code has been developed to satisfy the need for a simple FORTRAN based language to control processing on the Illiac IV computer. It is suitable for inherently parallel programs in which the bulk of computation involves "vectors" of length 64. Specialized routines, where direct use of hardware registers leads to a significant speed increase, should be programmed in ASK, the Illiac IV Assembler Language.

The Illiac has a very special architecture and its full potential can be realized only by careful (and sometimes clever) design of algorithms and programs. For most applications, however, the parallelism in the problem (and hence the main structure of the program) will be obvious, and the major effort involved will be that of organizing the large low-speed memory (i.e., disks) for efficient use of Illiac's high-speed random access core.

The structure and limitations of CFD code are determined by three factors: 1) the architecture and instruction set of Illiac, 2) the expected nature of the "average" program, and 3) the ease of writing the translator. No attempt has been made to hide the hardware limitations from the user since he must know them in order to write an efficient program. The programmer refers to variable names which in fact refer to memory. As in FORTRAN, direct access to registers is not allowed. On the Illiac, however, there will be many algorithms which can be greatly speeded up by manipulation among registers. These algorithms will usually be prime candidates for library-subprogram status, and should be coded in ASK. CFD on the other hand performs best when used for heavy but straightforward calculations of vectors.

The future development of CFD code depends strongly on the user reaction to the current version. The objective is to make the coding process (but <u>not</u> the program design) easier and less time consuming. The CFD branch is interested in <u>any</u> suggestions that would make CFD code more useful, or easier to use.

2.1 The ILLIAC as seen by a CFD programmer

The ILLIAC IV is basically a 64 computer array led by a small highly specialized control computer. Each member of the array is called a processing element (PE) and the control computer is the control unit (CU). The PE's operate in parallel, controlled by the instruction stream in the CU, and communicate data with the CU as well as among themselves.

2.2 The Control Unit

The primary functions of the control unit are program control and the scalar integer arithmetic involved in indexing PE variables. The CU memory contains 64 addressable memory locations. Locations 57 through 64 are reserved for system constants. Locations 49 through 56 are reserved for subprogram linkage when subprograms are used, but are available to the programmer in main programs that do not call subprograms. The remaining 48 locations, 1 through 48, are available to the programmer without restriction.

CFD has a command that allows the programmer to transfer 8-word blocks between CU and PE memory to facilitate any juggling required by the small size of available CU memory. The CU memory as used in CFD code requires the introduction of some very simple and yet "non-FORTRAN" concepts: 1) the CU memory is shared among all routines as though it were in COMMON, and 2) the programmer must lay out his part of CU memory (1 through 48, or 1 through 56) explicitly, that is, he must assign a CU memory address to each CU variable.

The arithmetic hardware of the CU has no provision for floating-point operations; all such operations must be performed in the PE's. The PE's operate on vectors, however, so that any scalar operation in the PE's is using only 1/64 of the machine potential. The CU can perform integer addition and subtraction, but most CU instructions are concerned with logical arithmetic and program control (branching). CFD control statements such as GO TO's, DO's, RETURN's, and the argument addressing when subprograms are called are processed in the CU. CFD statements, or parts of them, that do not require the PE's can be performed at the same time as the preceding PE operations. This is called "overlap." When a long string of PE instructions appears, the CU may become idle at times. This time may be used if the programmer distributes primarily CU statements uniformally through the program. A much more serious problem occurs when the PE's become idle. This may occur when a long string of primarily CU statements is encountered. The moral here is that scalar statements that are performed in the CU should be distributed as uniformly as possible among the vector statements. This will make optimum use of the instruction overlap of the CU and PE's.

2.3 The processing elements

The bulk of computation on ILLIAC is done in the PE's. Each of the 64 PE's has a memory of 2048 64-bit words. This memory, which we will call core, should be viewed as a high speed working area only, since it is not likely that the average ILLIAC program and data base will fit into such a small area. At present, main memory is the ILLIAC disk system. The crucial problem in using ILLIAC is to prevent the PE's from becoming idle while waiting for I/O between core and disk. Failure here can result in execution times that are orders of magnitude greater than necessary.

The arithmetic hardware in the PE's has a complete floating-point as well as integer instruction set. The PE's operate in lock-step (i.e., they all execute the same instruction at the same time). The only control mechanism in the PE's is the ability of a PE to be "disabled." When a PE is disabled its memory is protected and will not be altered even though its arithmetic hardware continues to function. The memory of a disabled PE may be read, but not written over by the arithmetic hardware. The status of the PE's, whether enabled or disabled, is called the mode. The mode can be thought of as a 64-bit string. Each bit controls one PE; when the bit is 0 the corresponding PE is disabled (off), and when the bit is 1 the PE is enabled (on). Control of PE processing then takes one of two forms: 1) a branch determines what CFD statement will be executed next, and 2) a mode pattern determines which PE's will execute the following CFD statements. These can be used separately or in combination. CU operations are not affected by the mode.

CFD code does not hide ILLIAC limitations from the programer. For example, in an arithmetic statement involving multiplication the variable being defined must be a vector. Any arithmetic expression that involves multiplication, division, exponentiation, or floating-point addition or subtraction may appear only where vectors are allowed because the required hardware is available only in the PE's.

3.1 Card format

CFD is written in a card form very similar to FORTRAN. The differences from FORTRAN are

1) An * must appear in column 6 except for assignment statements. These must have column 6 blank (comment cards are excepted).

2) A hyphen as the last non-blank character in columns 7 through 73 indicates that the card is to be continued. Columns 1 through 6 are ignored on continuation cards, which must number 19 or less.

3) No blanks are required after column 6, and those present are ignored.

4) Statement numbers may not be greater than 4 digits in length, must not contain embedded blanks, and must appear between columns 2 and 5, inclusive. Leading zeroes are permitted in statement numbers.

5) The cards must be punched in EBCDIC (029 punch or equivalent).

6) Column 1 is reserved for a C indicating a comment, an F indicating a FORTRAN statement, or an A indicating an ASK statement. The ASK statements must appear in columns 2 through 64 and are <u>not</u> continued in the same way as CFD statements. Embedded ASK statements are <u>not</u> included in the relocatable object file. However, <u>if</u> ASK output is requested, the translator emits the ASK unchanged except that the emitted ASK is shifted one column to the left. <u>Every</u> ASK statement must be flagged by an A in column 1.

7) Columns 74 through 80 are ignored.

8) A single quote (') may be used in any column after column 6 to denote that the remainder of the card image is a comment. The continuation hyphen, if present, must preceded the quote.

The most common programmer error with respect to card format will probably be the omission of the * in column 6. Remember, all statements except assignment statements, comment statements, and embedded ASK or FORTRAN statements require an * in column 6!

3.2 Naming conventions

Names in CFD follow FORTRAN rules: they consist of an alphabetic character which may be followed by as many as five alphameric characters. Both the ASK assembler and CFD require some reserved names. The following symbols may not appear as names in a CFD program because they are reserved by the ASK assembler.

ABS	END	MLI	SLA	TAGS
BEGIN	ENTRY	PAIR	TAGC	TAGW
BLOCK	IOWDA	RELOC	TAGI	USE
DEC	LAYOUT	RWA	TAGR	WDA
DVI				

The following symbols have a special meaning in CFD and may not be used as names

MBIT1	MBIT2	MODE	ON	OFF

3.3 Classes of named quantities and their residences

There are four classes of named quantities in CFD: 1) variables, 2) subprograms, 3) common blocks, and 4) disk areas. Variables may be divided into three subclasses: 1) scalars, 2) arrays, and 3) vector aligned arrays. The distinction between the second and third subclasses is very important. An array may reside in either PE or CU memory and may be of any length, limited only by memory size. These arrays may not be used as vectors in vector expressions, and may not have more than one subscript. Vector aligned arrays on the other hand must reside in PE memory, and may have one, two, or three subscripts. The range of the first subscript of a vector aligned array is always 64. All vector aligned arrays have their first word in the first PE, hence the nomenclature "vector aligned." A subprogram may be either a FUNCTION or a SUBROUTINE, and is referenced in the usual FORTRAN manner.

There are five types of variables: CU INTEGER, CU REAL, CU LOGICAL, PE REAL, and PE INTEGER. As in FORTRAN there is a pre-defined convention: variable names beginning with A-H and O-Z are taken as PE REAL and those beginning with I-N are taken as CU INTEGER. This convention can be overridden by IMPLICIT type statements which in turn may be overridden by explicit type statements. PE LOGICAL variables are not allowed.

Examples:

```
*IMPLICIT  CU REAL(D), CU LOGICAL (M)
*IMPLICIT  PE INTEGER (I)
*CU INTEGER INDEX
*CU LOGICAL BOOLE
*PE REAL   MACH(*,64)
```

CFD logical variables are quite different from FORTRAN logical variables. FORTRAN logical variables have one value (either .TRUE. or .FALSE.) while CFD logical variables always have 64 values, one in each bit of the 64-bit word. In this sense they are vectors, and when used to control the PE's, each PE receives one bit. CFD logical variables should then be viewed as a 64-bit string. The first bit corresponds to the first PE, the second to the second, and so on. CFD has three special logical variables: MODE, MBIT1, MBIT2. All three of these variables reside in the PE's and are reserved symbols. MODE contains the current machine mode pattern; MBIT1 and MBIT2 refer to logical variables into which the MODE may be stored more quickly than all other CFD logical variables which reside in the CU.

3.4 Indexing arrays

There are three classes of indices in CFD. The first class consists of integer scalar expressions involving not more than one nonsubscripted CU integer variable and not more than one integer constant. This is similar to the standard FORTRAN index. The second class of index is the * which may appear only in the first index of a vector aligned variable. The * indicates that the quantity is to be used as a vector. The * may be thought of as a generalized PE number. For example

```
            A(*) = B(*,J)    (assume all PE's are on)

means:      A(I) = B(I,J), I = 1, 64
```

The third class of index may also be used only with vector aligned arrays, and it must appear in the second index. This index is itself a vector aligned array and is called a "vector index." For example

```
            A(*) = B(*,J(*))      (assume all PE's are on)

means:      A(I) = B(I,J(I)), I = 1, 64
```

This is the only place in CFD where a subscript may be subscripted. A vector index may not be indexed by a vector however. The quantity

```
            VECTOR(*-I+5,-J(* + K-1, M+2, N-1)-INDEX-3, INDEX+2)
```

shows the allowed indexing for vector aligned variables. CU arrays and PE arrays that are not vector aligned may have a single class 1 index. We will discuss indexing more fully in connection with PE arithmetic.

3.5 Allocation of storage

CU Memory

The programmer has 48 words (56 if execution involves a main program only) of CU memory at his disposal. Each CU variable must be assigned a CU address by use of an EQUIVALENCE statement. For example, the sequence

```
*CU INTEGER D(5)
*CU LOGICAL BOOLE
   .
   .
   .
*EQUIVALENCE (3,D(2),BOOLE), (1,I), (2,J)
```

assigns the variable I to the first word of CU memory, J to the second, BOOLE to the third, and the array D to the second through sixth words.

PE Memory

PE memory is allocated in the usual FORTRAN manner with several exceptions: 1) all common blocks must be allocated in a **BLOCK DATA** subprogram whether or not they are initialized by DATA statements, 2) all scalars must be declared before the first executable statement, and 3) vector alignment must be stated explicitly.

The layout of PE arrays is done in the standard FORTRAN fashion. Subscripts vary most rapidly left to right. This is illustrated in the following example.

```
*DIMENSION U(*,2,2)
```

PE(1)	PE(2)	....	PE(64)	Vector Notation
U(1,1,1)	U(2,1,1)		U(64,1,1)	U(*,1,1)
U(1,2,1)	U(2,2,1)	....	U(64,2,1)	U(*,2,1)
U(1,1,2)	U(2,1,2)		U(64,1,2)	U(*,1,2)
U(1,2,2)	U(2,2,2)		U(64,2,2)	U(*,2,2)

It can be seen from this example that the first subscript ranges "across" the PE's while the second two range "down" the PE's. The second two subscripts behave in the normal FORTRAN manner, but the first index has some special properties that are unique to Illiac. These will be fully discussed in connection with PE arithmetic.

Example:

```
*PE INTEGER  LIMIT, INDEX(*), SCRTCH(64)
```

This statement identifies LIMIT as a PE INTEGER variable and INDEX as a PE INTEGER vector aligned array. The * in the first index location declares a variable to be vector aligned, and the range of the first index as 64. Thus INDEX and SCRTCH are both arrays of length 64, but only INDEX is forced to be vector aligned. SCRTCH may not be used as a vector.

The size of PE memory may demand the use of equivalence to conserve storage. The CFD EQUIVALENCE statement for PE variables is used in the same way as in FORTRAN. Thus

```
*DIMENSION VINDEX(*), VECTOR(*,10)
*EQUIVALENCE (VINDEX(1), SCALAR, VECTOR(1,2))
```

causes the second row of VECTOR to coincide with VINDEX, and the first word of this row to coincide with SCALAR. All common blocks must be allocated within a BLOCKDATA subprogram. For example

```
*BLOCK DATA
*IMPLICIT PE REAL (A-Z)
*COMMON/VELOCT/U(*,64), V(*,64)
*COMMON/THERMO/P(*,64), RHO(*,64), T(*,64)
*COMMON/CONSER/MASS(*,64), MOMENT(*,64), ENERGY(*,64)
*END
```

allocates the space for block VELOCT, THERMO, and CONSER.

No check is made to determine if the allocated block is the longest block under a given name, this is the programmer's responsibility. Common blocks appearing in the same BLOCKDATA subprogram may be overlapped through the use of EQUIVALENCE. Thus

```
*EQUIVALENCE (MASS(1,1),RHO(1,1))
```

causes common blocks THERMO and CONSER to overlap.

Common blocks are always vector aligned, that is their first word is in PE(1). Thus the statement

```
*COMMON/WORK/SCRTCH(60), VDUMY(*)
```

is in error because WORK and VDUMY cannot both be vector aligned. For the same reason, the following sequence is invalid

```
*PE REAL MATRIX
*DIMENSION MATRIX(*,64), ROW(*)
*EQUIVALENCE (ROW(6), MATRIX(3,14))
```

since the first words of ROW and MATRIX cannot both be in PE(1).

3.6 Data initialization

CU Memory

Values of CU variables may not be defined by DATA statements since the Illiac loader loads only PE memory. CU variables may be defined only by executable statements.

PE Memory

PE variables may be initialized by DATA statements in the usual FORTRAN manner.

Example:

```
*IMPLICIT  PE INTEGER (X)
*PE REAL MATRIX (*,64)
*DIMENSION XX(*), S(40)
*DATA  MATRIX, S(5)/2048*0., 2049*1./
*DATA  XX/0,1,2,3,60*0/
```

The variable list must have a one to one correspondence with the data list.

3.7 Program control

There are two kinds of program control in ILLIAC: 1) branching, and 2) enabling or disabling PE's. These may be used separately or in combination. Branching is the type of control used in serial computers and determines which statement will be executed next. In ILLIAC, however, it is also necessary to specify which PE's will participate in the execution of a vector statement.

Control of execution sequence

The following statements are implemented in CFD with their standard FORTRAN form and meaning.

GO TO (absolute, computed, and assigned)
ASSIGN
CONTINUE
RETURN
STOP
CALL
END

One of the most frequently used control statements is the DO statement, and in CFD it is slightly more general than in FORTRAN. As in FORTRAN the index must be greater than zero. The differences from FORTRAN are:

1) The increment must be a constant, but may be negative

2) The starting and limit values may be simple arithmetic expressions of the form $\pm v$, $+c$, $\pm v \pm c$ where v is a CU INTEGER variable and c is a constant. Leading plus signs are optional.

Example:

```
*DO 1234 INDEX = J+10, -K+2, -1
```

Logical IF statements are implemented in CFD, but arithmetic IF statements are not. IF statements are of two basic kinds: 1) those with a single true/false result, and 2) those with 64 true/false results, one for each PE. The first kind is similar to its FORTRAN counterpart, while the second is concerned with enabling PE's. The first kind decides what is to be done and the second decides in which PE's it will be done. There are no single result (one bit) logical variables in CFD, so the form of the first kind of IF is quite restricted. There are three basic forms: 1) that involving arithmetic tests between CU INTEGER expressions using only addition and subtraction,

2) that involving quantified logical expressions, and 3) that testing for I/O request completion. A logical expression in CFD implies 64 true/false results, and "quantifying" it reduces it to one true/false result. The logical quantifiers in CFD are .ANY., .ALL., .NOT ANY., and .NOT ALL..

Examples:

```
*IF(INDEX .GT. LIMIT) RETURN
*IF(.NOT ANY. ((A(*).GT.EPSLON))) STOP
```

In the second example, the true/false result of the expression (A(*).GT.EPSLON) is determined _in every PE_. If and only if the expression is false _in every PE_ the STOP is executed, otherwise the program continues at the next statement. In other words, the program stops if and only if A is less than or equal to EPSLON _in every PE_. In the statement

```
*IF(.ALL.((SCALAR.LE.VALUE))) A(*)=0.
```

the .ALL. is logically unnecessary, but is required because the floating-point comparison of SCALAR and VALUE is done in the PE arithmetic hardware. Remember that the CU has no floating-point hardware. The result of the comparison is the same in all PE's, and the .ALL. (.ANY. gives the same result here) is required to reduce the result to a single true/false value.

Control of the PE's

The PE's are controlled in two ways: 1) the instruction stream in the CU determines the machine instruction to be executed, and 2) the enabling mode pattern in the PE's determines which PE's will perform the instruction and which will remain idle. At the CFD level, the enabling mode controls _only one thing_, namely, those PE's whose values are being defined in a vector arithmetic assignment statement. Vector arithmetic statements do not alter variables in disabled PE's. The enabling mode pattern is the logical variable MODE, a reserved symbol, at all times except when the vector assignment statement following a vector IF is executed. In that case the enabling mode is the result of the vector IF. For example

```
*IF((A(*).LT.0.))A(*) = -A(*)
```

is one way to replace A(*) by its absolute value.

If the sequence

```
MODE = (A(*).LT.0.)
A(*) = -A(*)
```

is used A(*) is replaced by its absolute value as before, but

now the enabling mode has been set so that only the PE's in which A was negative will be active in following statements. Use of the IF is logically equivalent to

```
MBIT1 = MODE
MODE = (A(*).LT.0.)
A(*) = -A(*)
MODE = MBIT1
```

This is the procedure used if more than one statement is to be executed with the enabling mode as (A(*).LT.0.) since only one vector arithmetic statement may follow a vector IF. The logical variable MODE is not altered by a vector IF statement.

The distinction between scalar logical IF statements and vector logical IF statements is that the scalar IF determines what statement will be executed next while the vector logical IF determines which PE's will execute the vector assignment statement that the IF controls.

In the above example the logical result of the vector relation (A(*).LT.0.) is evaluated in each PE and is not controlled by MODE.

During execution the enabling mode will usually be MODE. MODE may be altered <u>only</u> by its appearance in a logical assignment statement. The statement

```
MODE = ON
```

,for example, enables all PE's. In addition to MODE, the symbols ON and OFF have special meanings in CFD. The symbols ON and OFF represent logical constants implying "all PE's enabled," and "all PE's disabled," respectively. With the exception of MODE, MBIT1, and MBIT2 all logical variables must reside in the CU.

In a logical assignment statement a logical variable is assigned the value of a logical expression. The basic building block of a logical expression is the "base mode" which may be a logical variable, logical constant, vector relation, or any of these preceded by .NOT., which implies logical negation.

A vector relation consists of two vector arithmetic expressions separated by one of the following: .GT., .LT., .GE., .LE., .EQ., or .NE.. <u>Vector relations must always be enclosed in parentheses</u>.

The logical expression may simply be a base mode, or it may contain operators having base modes as operands. There are three kinds of operators: 1) bit setting operators, 2) bit shifting and rotating operators, and 3) logical operators.

The two kinds of bit setting operations are .TURN ON., and .TURN OFF. and are used to turn on (enable) or turn off (disable) discrete bits of the variable being defined. The bits themselves are specified in a list following the operator.

Examples:

```
MASK = ON .TURN OFF. 1, 64, INT
MODE = MODE .TURN OFF. .FIRST. I+2, .LAST. J-1
LOGICL = MASK .TURN ON. K-6 .TO. K+6
```

The list may indicate individual PE's or ranges of PE's and the PE numbers may be simple arithmetic expressions containing one nonsubscripted CU INTEGER variable and/or one integer constant. PE numbers must be in the range 1, 64 for bit-settings operations.

The two kinds of bit shifting operations are "end-off" shifts (.SHL., .SHR. for left and right shifts respectively) and "end-around" shifts (.RTL., .RTR. for left and right; these are usually called "rotates" rather than shifts). In the end-off shifts, vacated bits are set to zero, for example

(1111..111) .SHR.3 → (0001....1)

In the end-around shifts, bits that are shifted off one end appear at the other, for example

(111100...00) .RTL. 3 → (100...00111)

Shift distances must be in the range 0,63 for bit-shifting operations.

The three logical operators are .NOT., .AND., and .OR.. The following tables present the logical operators. The 1's represent on, true, or enabled while the 0's represent off, false, or disabled.

A	.NOT. A
0	1
1	0

A	B	A .AND. B
0	1	0
1	1	1
0	0	0
1	0	0

A	B	A .OR. B
1	1	1
0	1	1
1	0	1
0	0	0

The identities

.NOT. (A .AND. B) = .NOT. A .OR. .NOT. B

and

.NOT. (A .OR. B) = .NOT. A .AND. .NOT. B

may be useful to the programmer.

The logical quantifiers have the following meanings:

.ANY.(A) is true if and only if at least one A_i is true

.ALL.(A) " " " " " " all A_i are true

.NOT ANY.(A) is true if and only if all A_i are false

.NOT ALL.(A) " " " " " " at least one A_i is false

The following identities follow from these definitions:

.NOT ANY.(A) = .ALL.(.NOT.A)

.NOT ALL.(A) = .ANY.(.NOT.A)

Logical quantifiers may appear only in scalar logical IF statements, and <u>their operand must be enclosed in parentheses</u>.

The operands of logical assignment statements are not altered by the statements.

There are a number of logical expressions that, although logically and syntactically meaningful, are of little or no use. Such invalid expressions are eliminated by the following rules.

1) ON and OFF may not be used with any logical operator, logical quantifier, or end-around shifting operator (.RTR. and .RTL.).

2) OFF may not be used with end-off shifts (.SHL. and .SHR.) or with .TURN OFF..

3) ON may not be used with .TURN ON..

4) logical expressions containing the shifting operators .RTL. or .RTR. may not be quantified.

5) logical expressions containing .TURN ON. may not be quantified by .ANY. or .NOT ANY..

6) logical expressions containing .TURN OFF. may not be quantified by .ALL. or .NOT ALL..

Examples of valid logical assignment statements:

```
MASK = MASK .RTR. 1

MODE = (U(*)**2 + V(*)**2 .GT. C(*)**2)

MODE = MODE .AND. MASK
```

No CFD vector logical expression may contain more than one non-arithmetic operator (i.e. logical, bit setting, or shifting operators) with the exception of .NOT..

3.8 Arithmetic

CU arithmetic

There are three kinds of scalar arithmetic statements, all of which are specific and restricted. The limited vocabulary for CU arithmetic reflects the absence of the required hardware. The first kind of statement is an arithmetic assignment statement involving only CU INTEGER variables, integer constants, and the + and - operators. No parentheses are allowed.

Examples:

```
KOUNT = KOUNT + 1

NUMBER = LAST - INITL + 1
```

The second kind of statement involves the transfer of single words of data. No arithmetic is done, and the data may be real or integer and have any residence (CU or PE).

Examples:

```
PI = 3.14159

VECTOR(6) = SCALAR

SCALAR = VECTOR(6)
```

The third kind of statement has no FORTRAN equivalent and is required in ILLIAC to facilitate any necessary juggling between CU and PE memory due to the limited size of CU memory. The TRANSFER command allows the programmer to move blocks of 8 words between CU and PE memory.

Examples:

```
*TRANSFER(16)        PEBLK = CUBLK

*TRANSFER (8)        BUFFER(L) = VECTOR (I,J,K)
```

The first statement indicates that two blocks of 8, starting at CUBLK are to be transferred to the two blocks of 8 starting at PEBLK. The second statement specifies transfer of one block starting at VECTOR(I,J,K) in PE memory to the block starting at BUFFER(L) in the CU. The command always contains the number of words to be transferred (an integer multiple of 8), the source and the destination. If the source is CU, the destination must be PE, and vice versa. The first word in source and destination blocks **should be the first word in a block of** 8 (but not necessarily the same block). In the CU, blocks of 8

begin at locations 1, 9, 17, ... and in the PE's at PE 1, 9, 17, ... This must be insured by the programmer through use of EQUIVALENCE.

The two legal uses of the TRANSFER statement are:

1) To save blocks of CU variables in PE memory when CU space is needed and to return such blocks to the CU. The blocks must not be used while residing in PE memory.

2) To fetch to the CU blocks of 8 PE floating-point words that will be broadcast as scalars back to the PE's in PE arithmetic statements. <u>This is restricted to floating-point data</u>.

PE arithmetic

PE arithmetic is <u>vector</u> arithmetic, even when an expression involves only scalars. Expressions must be either real or integer, and mixed type expressions are not allowed. Type changes across the "=" are also forbidden. Integer PE arithmetic is primarily used for computation of vector indices.

The standard FORTRAN operations +, -, *, /, and ** are implemented. The order of computation is the same as FORTRAN. The exponents in CFD must be integer constants in the range 2 through 10 and may not be exponentiated themselves. Thus, A**2**3 is invalid, but its equivalent (A**4)**2 is valid.

The variable being defined in a PE arithmetic assignment must be vector aligned, and its first subscript must be * alone. This convention is followed because the enabling mode (MODE) then corresponds directly to the PE's of the defined variable. The expression whose value is assigned to the defined variable may contain parentheses and FUNCTION references.

Examples:

```
F(*) =  ((X(*)*A(*) + B(*))*X(*) + C(*))*X(*) + D(*)

FTRAN(*) =  FFT(VRTCTY(*,I,J))*OMEG(*)

ZVORT(*,I,J) =  DY(U(*,I,J)) - DX(V(*,I,J))

ENERGY(*,I) =  U(*,I)**2 + V(*,I)**2
```

In these examples, FFT, DY, and DX are FUNCTION subprograms.

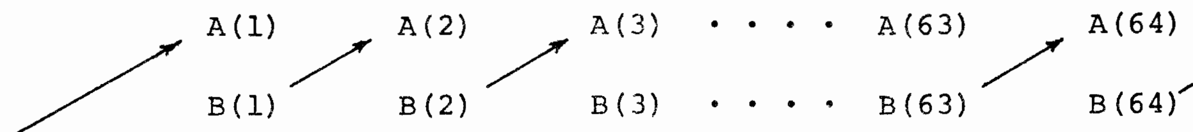
When the first subscript contains an *, the subscript possesses some non-FORTRAN qualities. Assume that all PE's are enabled, then the statement

A(*) = B(*-1)

is equivalent to the statements

A(1) = B(64), A(2) = B(1), A(3) = B(2), ..., A(64) = B(63)

illustrated by the following diagram.



The array B may be thought of as being arranged on a cylinder, with the first subscript being wrapped around the cylinder and the second and third subscripts arrayed parallel to its axis. PE(1) follows PE(64).

Suppose the central difference of the vector P(*) is needed, its value, given by

D1P(*) = P(*+1) - P(*-1)

may have no meaning in PE's 1 and 64 unless P is in fact periodic. The difference would not be computed in these PE's if the statement above were preceded by

MODE = ON.TURN OFF. 1, 64

An * in the first subscript then implies that the variable is a vector. When the first subscript contains no * the variable is used as a scalar, the same value being used in every PE. For example, the CFD statement

A(*,I,J) = B(*,I)/C(J,I)

is equivalent to

A(1,I,J)	A(2,I,J)	. . .	A(64,I,J)
↑	↑		↑
B(1,I) / C(J,I)	B(2,I) / C(J,I)	. . .	B(64,I) / C(J,I)

The single value C(J,I) is the divisor in every PE. When the * is missing, the first subscript behaves in the usual FORTRAN manner.

When the first subscript contains an *, the second subscript, if present, may contain an integer vector. This allows each PE to refer to a different position in its memory. Suppose the variable IV has been declared a PE INTEGER vector and has been assigned the value i in PE(i), that is

```
IV(1) = 1
IV(2) = 2
   .
   .
IV(64) = 64
```

Then, if AMATRX is a 64x64 matrix,

```
DIAG(*) = AMATRX(*,IV(*))
```

defines DIAG as the diagonal of the array AMATRX.

Example:

Suppose we need the row sums of AMATRX. These are computed by the statements: (all PE's on)

```
     ROWSUM(*) = 0.
    *DO 1 I = 1, 64
  1  ROWSUM(*) = ROWSUM(*-1) + AMATRX(*,IV(*-I+1))
     ROWSUM(*) = ROWSUM(*-1)
```

which leaves the sum of row 1 in ROWSUM(1), the sum of row 2 in ROWSUM(2), and so forth. This algorithm uses the "end-around" nature of the * index. Verification of this algorithm is a useful exercise toward understanding of both vector indexing and the end-around nature of the * subscript.

3.9 Subprograms

There are two kinds of subprograms in CFD, the FUNCTION and the SUBROUTINE. Both are used in the same way as their FORTRAN counterparts. There are, however, several differences in implementation between FORTRAN and CFD.

Communication between programs is achieved in four ways. The first two, subprogram arguments and common blocks, are used in the same manner as in FORTRAN. The last two communication links are through CU memory and the special logical variables (MODE, MBIT1, and MBIT2), which are shared by all programs. They may be thought of as being in common.

There may be as many as seven subprogram arguments, each of which may be of three kinds: 1) subprogram names, 2) common block names, and 3) vector aligned arrays. At the call or function reference, the arguments may be subprogram names, common block names, or vector quantities. A vector quantity may be either an array name, a subscripted array name, or a PE arithmetic expression. If an array name is subscripted with anything other than * alone as the first subscript, or is indexed by a vector in the second subscript it is considered a PE arithmetic expression, even though no arithmetic operations are involved. In FORTRAN, an arithmetic expression as an argument at call must coincide with a scalar argument in the subprogram to which the subprogram does not assign a value. In CFD the only variables passed through an argument list are vectors. Thus a PE arithmetic expression at call must coincide with a single row vector argument in the subprogram to which the subprogram does not assign values. All subprogram arguments are passed by row address in CFD except subprogram names which are passed by word address. PE arithmetic expressions are evaluated as a single vector at call, and placed in system's scratch space. The address of the scratch row used is then passed to the subprogram. Constants are arithmetic expressions. The following list gives examples of arguments at call, and how they are passed.

	Argument at call	Address passed to subprogram
variables:	A	A(*), A(*,1), A(*,1,1)
	A(*)	A(*)
	A(*,I)	A(*,I)
	A(*,I,J)	A(*,I,J)
expressions†:	A(*-1)	scratch row
	A(*,IV(*))	scratch row
	-A(*)	scratch row
	A(*)*B(*)	scratch row
	3.14159	scratch row
	SCALAR	scratch row

†The expressions are evaluated and placed in a scratch row whose row address is passed to the subprogram. The evaluation is controlled by the current mode (MODE). Values passed through a scratch row are defined only in PE's enabled by the MODE. Thus, all disabled PE's will have garbage in the scratch rows.

The ability of the programmer to pass to the subprogram the name of a common block is a non-FORTRAN feature of CFD that serves several purposes. These purposes depend on one essential fact: when a subprogram requires more than one variable, but all the variables required are stored in a fixed relationship to one another and this relationship is known when the subprogram is written, then only one address is required by the subprogram at execution time. The same end can be achieved through EQUIVALENCE to dummy variables within a subprogram. This is also allowed in CFD, but not in FORTRAN. This feature of CFD is useful when PE memory is laid out in blocks for I/O buffering.

Example 1:

Calling Program

```
*DIMENSION A(*,10,10),C(*,3),U(*),V(*)
*COMMON/COMBLK/VARIBL(*,3)
     .
     .
     .
*CALL SUBPRG(COMBLK,A,C(*,I),U(*)*V(*))
```

Called Program

```
*SUBROUTINE SUBPRG(A,B,C,D)
     .
     .
     .
*DIMENSION B(*,10,10),C(*),D(*)
*COMMON/A/X(*),Y(*),Z(*)
     .
     .
     .
 X(*) = B(*,I,J) + C(*)*D(*)/Y(*)
     .
     .
*RETURN
     .
     .
```

Example 2:

Calling Program

```
U(*) = CONT(U(*))
```

Called Program

```
*FUNCTION CONT(VEL)
     .
     .
     .
*DIMENSION VEL(*,2)
     .
     .
     .
*CALL DIFF(VEL(*,2) - VEL(*,1),CONT)
     .
     .
     .
*RETURN
     .
     .
```

Example 3:

Calling Program

```
D2V(*) = FORDIF(FORDIF(V(*+1)))
```

Called Program

```
*FUNCTION FORDIF(VECTOR)
*DIMENSION VECTOR(*)
     .
     .
     .
 FORDIF(*) = VECTOR(*) - VECTOR(*-1)
     .
     .
     .
*RETURN
```

3.10 CFD Supplied functions

In CFD, an intrinsic function (CFD supplied function) is always a vector function whose argument is a vector arithmetic expression. The 10 intrinsics

(ABS, IABS, IFIX, FLOAT, EXP, ALOG, SIN, COS, ATAN, and SQRT)

have their familiar FORTRAN meanings, and are invoked in the usual FORTRAN manner.

Example:

```
*PE INTEGER      I(*), J(*)
*PE REAL         A(*), B(*), C(*)

 A(*) = EXP(FLOAT(I(*))*ALOG(B(*)))
```

With the exception of ABS, the intrinsic symbols are not reserved symbols, and may be specified by the programmer. In other words, the pre-definition of these symbols as intrinsics may be overridden by the programmer.

Example:

```
*PE REAL FLOAT(*,2,5)
```

This statement overrides the pre-defined specification of FLOAT as an intrinsic and specifies it as a PE vector aligned array.

A CFD intrinsic must not be declared EXTERNAL.

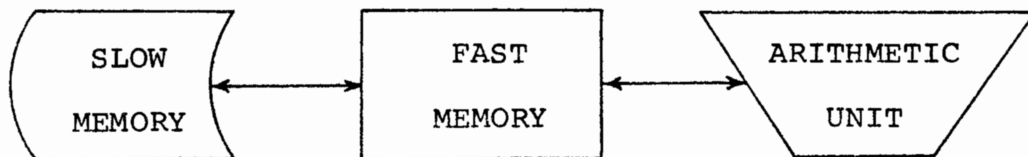
Example:

```
*EXTERNAL SIN
```

This statement specifies SIN as a <u>user supplied</u> subprogram, and overrides its pre-definition as an intrinsic.

4.1 Historical introduction to memory input/output

Since EDSAC, the world's first operational computer, programmers have had to contend with the problem of memory hierarchy and cost effectiveness in computer operation. In the case of EDSAC, a University of Manchester computer, there was the 1024-bit Williams Tube memory which had an access time on the order of microseconds, as well as a 128-word drum with a slower access time, measured in milliseconds. On the other hand, in terms of cost per bit stored the drum was cheaper than the fast access tube memory. In the case of today's conventional machines (IBM, CDC, Burroughs, etc.) the hierarchy of memory still exists, but these machines have very fast random access memory as well as magnetic rotating memory (drums, disks, tapes, etc.). The original problem remains: given a larger, cheaper, slower memory in addition to a smaller, more expensive, and faster memory what is the most cost-effective way to use the total computer?



The common way to utilize a computer in a cost-effective manner is to minimize the time during which the arithmetic unit is idle. Two ways by which to minimize the idle time of the arithmetic unit will be discussed in the following paragraphs.

In recent years sophisticated operating systems have been designed so that a computer may be used in a cost-effective manner. In this case the basic concept is the partition of the fast memory into N parts. In each of these partitions the system will place a separate program as well as part of its data base. When the job (in partition A, say) using the arithmetic unit requires data that currently does not reside in fast memory the operating system allows the arithmetic unit to begin processing a job resident in a different partition (say partition B). While the job in partition B is utilizing the arithmetic unit the data needed in partition A is transferred from the slower memory to the faster memory. Thus this method minimizes the idle time of the arithmetic unit by having several programs all using the same arithmetic unit.

Another method by which to minimize the idle time is to feed the small fast storage from the larger, slow one, transferring data in large blocks. The idle time is minimized by the fact that the <u>programmer</u> has insured that all data have

been moved to fast memory before they are needed by the arithmetic unit. This method is called asynchronous I/O since the data movement is not synchronized with the instruction stream, i.e., data movement may be taking place in one area of the fast memory while the arithmetic unit is processing data in another area.

Thus, there are basically two ways to cost-effectively use a computer. In the first case, it is the responsibility of an operating system. In the second, it is the responsibility of the programmer.

ILLIAC IV has a comparatively small, fast, random access memory (128 K words for 64 arithmetic units) and a large "slowly" rotating disk memory (15,900 K words). Furthermore, ILLIAC IV will have only a minimal operating system which implies that it will be the programmers responsibility to use ILLIAC IV in a cost-effective manner. The remainder of this section will be devoted to a discussion of how a CFD programmer might use the ILLIAC IV in a cost-effective manner given the limited amount of high speed memory.

4.2 Description of ILLIAC IV I/O hardware

The user, in order to efficiently use ILLIAC IV, must be aware of some of the hardware architecture of the ILLIAC Array and ILLIAC disk system (Figure 1). The remainder of this section will discuss each of the major parts.

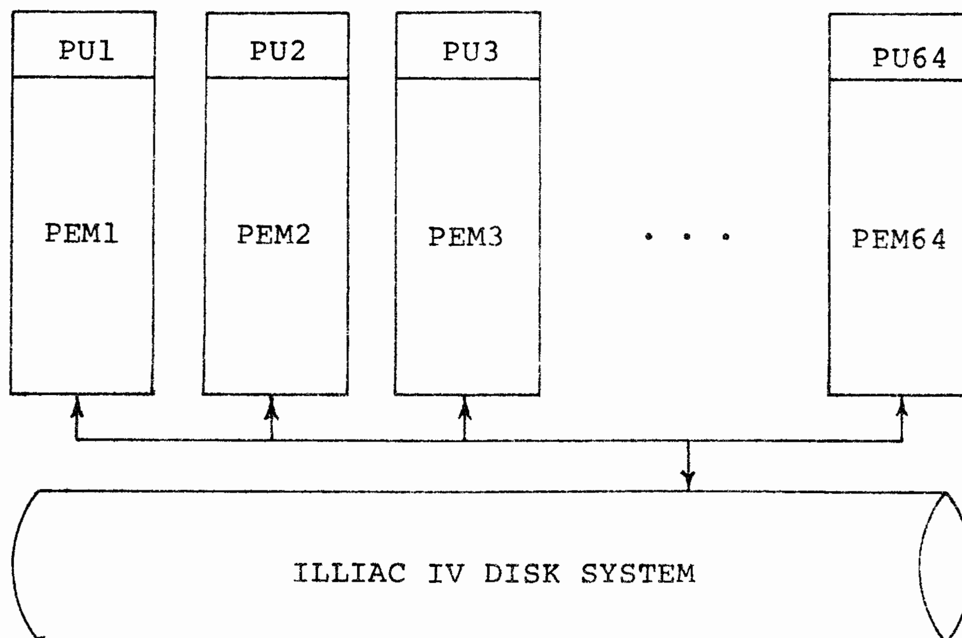


Figure 1.

4.3 Array memory

A Processing Element Memory (PEM) consists of 2048 64-bit words. Thus the total array memory may be thought of as a 64x2048 matrix of 64-bit words or 131,072 sequential locations as illustrated below.

PEM1	PEM2	PEM3		PEM62	PEM63	PEM64
0	1	2	. . .	61	62	63
64	65	66	. . .	125	126	127
128	129	130		189	190	191
⋮	⋮	⋮		⋮	⋮	⋮
130,994	130,945	130,946		131,005	131,006	131,007
131,008	131,009	131,040		131,069	131,070	131,071

4.4 ILLIAC pages

The smallest amount of data that may be moved to and from the Array memory is called an ILLIAC page. Each ILLIAC page contains 1024 64-bit words. Data may be transferred to or from the full array, either half of the array or any quarter of the array.

If the transfer refers to the full array, consecutive Array memory locations beginning with an address in PEM1 are affected. Thus, for each page transferred, 16 full rows (16 words in each PEM) are transferred.

If the transfer refers to half of the array, locations in vertically adjacent half rows will be affected. The beginning address for these half rows is in PEM1 or PEM33. Thus, for each page transferred 32 half rows (32 words in either the first 32 PEMs or the last 32 PEMs) are transferred.

Similarly, if the transfer refers to a quarter of the Array locations in vertically adjacent quarter rows will be affected. The beginning address for these quarter rows is in PEM1, PEM17, PEM33, or PEM49. Thus, for each page transferred 64 quarter rows (64 words in each of 16 adjacent PEMs) are transferred.

See Figure 2 for a graphic example of the full, half, and quarter array transfers.

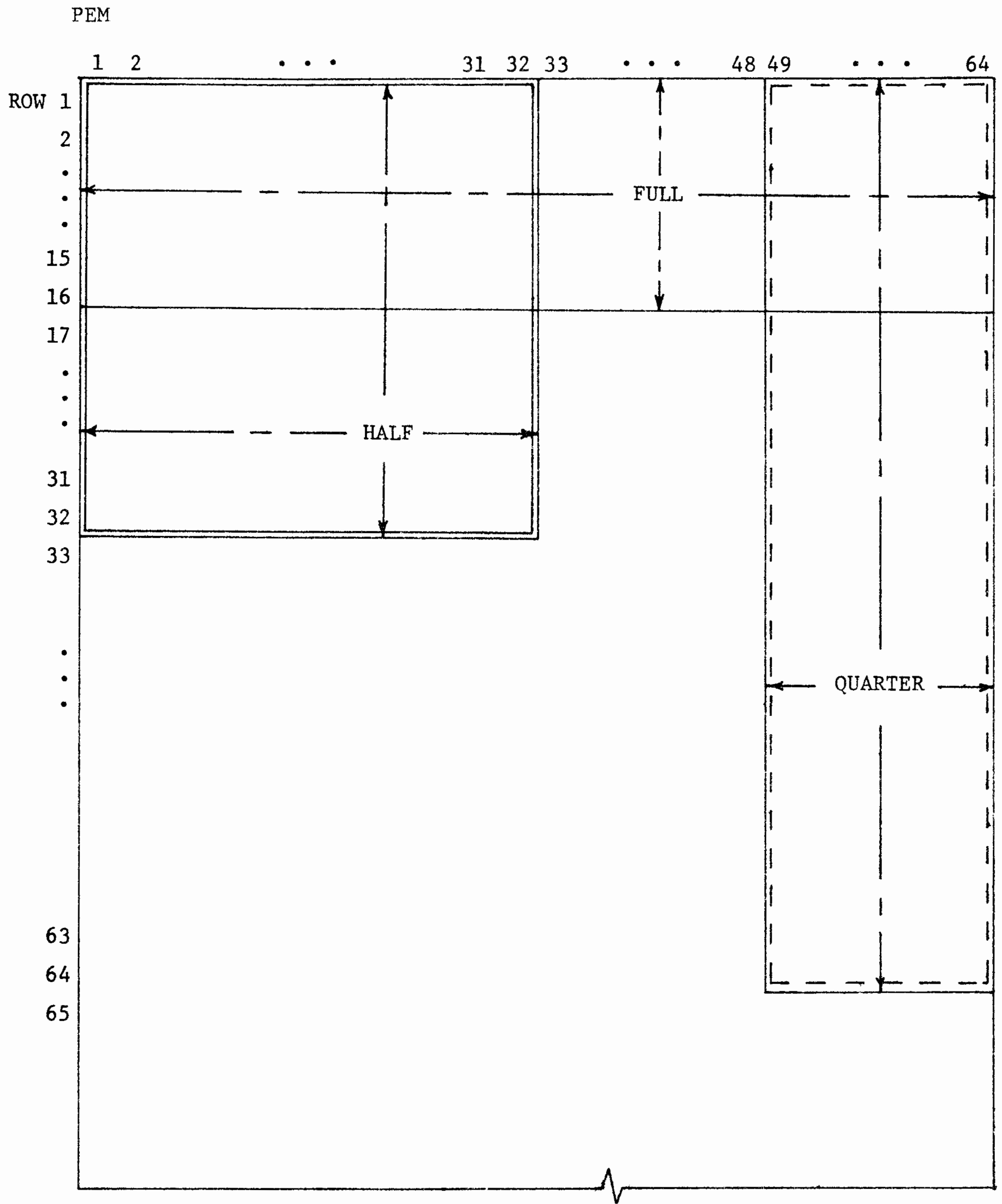


Figure 2.

4.5 ILLIAC disks

The ILLIAC disk system is made up of 13 ILLIAC disks. Each disk has four bands of three hundred ILLIAC pages each. The disks are synchronized so that all the corresponding pages of each band of each disk are under the fixed read/write heads at the same moment. As such, it consists of 52 bands with 300 pages per band giving a total of 15,600 pages or equivalently 15,974,400 64-bit words.

The rotation period of the disks is 40 milliseconds and since 300 pages can be transferred each revolution, the transfer rate is 133 microsecond per page. Put another way, the time it takes to transfer 64 words is approximately equal to the time it takes to complete 19 PE additions or 15 PE multiplications.

4.6 General concept of asynchronous I/O

When a programmer uses a high-level language (e.g., ALGOL, COBAL, FORTRAN, PL/I, etc.) all I/O, be it to disks, tapes, etc., is logically synchronous. That is, when an I/O request is made all processing within the program is stopped until the I/O request is completed. This usually is not inefficient because most operating systems will have the arithmetic unit servicing another program while the I/O request is being completed. The assembly language programmer, on the other hand, has what is called asynchronous I/O. In brief, asynchronous I/O simply means that the arithmetic unit can be processing one block of data while I/O is being done on another block of data. The obvious restriction here is that you must be sure an I/O request is completed before attempting to use any of the memory locations affected by the I/O request.

Because there is only a minimal operating system for ILLIAC IV it is the programmer's responsibility to overlap his computation and disk I/O so as to prevent the PU's from becoming idle. Unlike conventional sequential machines ILLIAC IV's disks will be synchronized. That is, by keeping track of the execution time it will be possible to know exactly which page will be under the read/write heads at all times. Thus, if both the computation time and the amount of I/O is known it is possible to place the data on the ILLIAC disk so that when a certain block of data is needed from the disk it will just be coming under the read/write heads. Similarly, any disk area which is to be written into can be allocated such that when a write request is issued, this area is just coming to the read/write heads. In general, the disk's being synchronous allows the programmer to manage his data and disk transfers in such a way as to minimize the time required in accessing a rotating memory device.

5.1 ILLIAC IV I/O in CFD - Introduction

CFD I/O, like the rest of the CFD language, reflects the hardware limitations of ILLIAC IV. CFD I/O is I/O between Array memory and the ILLIAC IV disks and this I/O is asynchronous in nature. Thus the CFD WRITE statements transfer the contents of Array memory locations to ILLIAC disk locations and CFD READ statements transfer the contents of disk locations to Array locations.

Other ILLIAC IV limitations reflected in CFD I/O are: 1) the quantum in which I/O may be performed is one ILLIAC page (1024 64-bit words), 2) that all I/O is performed between quarter Array boundaries. Additionally, there is a special CFD I/O IF statement to determine the status of a READ or WRITE request.

5.2 DISK AREA statement

Although disk areas are laid out by the MAP subsystem the CFD translator must be made aware of all disk areas to be accessed by the main program and all its subprograms. (See System Guide for the ILLIAC IV User for details about the MAP subsystem.) The CFD translator must also know the total number of pages allocated in each disk area. This information must be supplied in the DISK AREA statement which must be the statement immediately before the first executable statement.

Example:

```
*DISK AREA INPUT(20),OUTPUT(60)
```

5.3 READ/WRITE statements

There are three kinds of READ statements: READ, READH, and READQ. These statements transfer pages of data from the disk to the full, half, or quarter of the array, respectively. There are four arguments in a READ statement: a request number, an array location, a disk location, and the number of pages to be transferred.

Example 1:

```
*COMMON/ONE/RHO(*,16,6),PHI(*,16,6)
     .
     .
     .
*DISK AREA  ONESAV(48)
     .
     .
     .
*READ (5,RHO(1,1,1),ONESAV(25),6)
     .
     .
     .
```

Example 1 contains a READ statement which causes 6 full array pages to be transferred from the disk beginning at the 25th page of ONESAVE to the Array beginning at RHO(1,1,1).

There are also three kinds of WRITE statements: WRITE, WRITEH, and WRITEQ. These statements have the same general form and function as the three READ statements with the exception that the transfer is from the Array to the disks.

The request number in these READ/WRITE statements is required because CFD I/O is asynchronous. That is, a read/write statement is only a request that I/O be performed, not an insurance that the I/O has been performed before the execution of the next statement.

5.4 I/O IF statement

Since CFD I/O is asynchronous in nature, CFD provides a way to determine if a particular I/O request is completed. In general, the I/O IF statement functions as a scalar IF statement: if the statement in the parentheses is true the statement after the parentheses is executed, if the statement is false, execution continues at the next statement. The truth values are as follows given that *i* is the request number of a previously executed READ/WRITE statement:

	request *i* completed	request *i* not completed
.COM. *i*	TRUE	FALSE
.NOTCOM. *i*	FALSE	TRUE

Example:

```
     *DIMENSION  ABAR(*,32,6)
        :
     *DISK AREA  A(96)
        :
     *READH (9,ABAR(1,1,3),A(16),2)
        :
     *IF (.NOTCOM.9) GO TO 1234
        :
1234 *CONTINUE
        :
```

5.5 WAIT statement

The WAIT statement causes a pause in execution until a particular I/O request has been completed.

Example:

```
      .
      .
      .
    *READ(63,A(1,1,1),AREA(1),2)
      .
      .
      .
    *WAIT 63
```

Note:

Since CFD is a FORTRAN based language, the FORTRAN convention of numbering beginning with one (1) has been followed. An example being that in this manual we speak of PE1 through PE64. Thus, it should be noted that the majority of documents concerning the ILLIAC IV and its languages do not follow this FORTRAN convention. Instead, these documents begin numbering with zero (0) and thus would, for example, refer to PE0 through PE63. This convention of numbering from zero (0) is followed in the System Guide for the ILLIAC IV User, unlike this manual.

STATEMENTS

CFD statements are composed of CFD key words used in conjunction with the basic elements of the language (constants, variables, and expressions). The five categories of CFD statements are:

1. Arithmetic and Logical Assignment Statements: Replace the current value(s) of a designated variable after calculations have been performed.

2. Control Statements: Govern the flow and terminate the execution of the object program.

3. Input/Output Statements: Exchange information between a user's program and a named collection of data residing on secondary ILLIAC memory.

4. Specification Statements: Declare the properties of variables, arrays, and subprograms (e.g., type, residence, and amount of storage reserved).

5. Subprogram Statements: Define and name functions and subroutines.

All categories of CFD statements except category 1 (Arithmetic and Logical Assignment Statements) must be flagged by an asterisk in column 6 of the first card.

CODING CFD STATEMENTS

Card Input

The statements of a CFD source program can be written on a standard FORTRAN coding form, Form GX28-7327 (see Figure 3). CFD statements are written one to a line from columns 7 through 72. If a statement is too long for one line, it may be continued on as many as 19 successive lines by placing a hyphen in column 73, or as the last nonblank character (excluding comment), of each continued line. All blanks after column 6 are ignored.

Columns 2 through 5 of the first line of a statement may contain a statement number consisting of from one through four decimal digits. Leading zeros in a statement number are ignored. Statement numbers may appear anywhere in columns 2 through 5 and may be assigned in any order; the value of statement numbers does not affect the order in which the statements are executed in a CFD program.

Columns 74 through 80 are not significant to the CFD compiler and may, therefore, be used for program identification, sequencing, comment, or any other purpose.

Comments to explain the program may be written in columns 2 through 80, if the letter C is placed in column 1. Comments may appear anywhere within the source program after column 7 if preceded by a quote ('). They are not processed by the CFD compiler, but are printed on the source program listing. Blanks may be inserted where desired to improve readability.

ASK statements may be embedded within a CFD-code program by placing the letter A in column 1 of each ASK card. When the CFD translator encounters one of these cards, the A is deleted and the card is shifted left one column (column 2 becomes column 1, column 3 becomes column 2, etc.). This new card image is then inserted into the translator generated ASK. FORTRAN statements may be embedded by placing the letter F in column 1 of each FORTRAN card. (See appendix for details.)

IBM FORTRAN Coding Form

PROGRAM: MATIN
PROGRAMMER: Harvard Lomax
DATE:
PUNCHING INSTRUCTIONS

STATEMENT NUMBER	CONT.	FORTRAN STATEMENT
	*	SUBRØUTINE MATIN(A,B,DET)
	*	CU LØGICAL MASK(2)
	*	PE INTEGER IPIVT(*,20,20),IRØW(*),ICØL(
	*	DIMENSIØN A(*,20,20)B(*,20),DET(*),SAVE
	*	EQUIVALENCE (1,MASK(1)),(3,I),(4,J),(5,
		MBIT1=MØDE
	*	DØ 20 J=1,K
20		IPIVT(*,J,1)=0
		IRØW(*)=0
		DET(*)=1.0

Figure 3. FORTRAN coding form (remember that columns 6 and 73 are used differently in CFD code).

<u>CONSTANTS</u>

A constant is a fixed, unvarying quantity. The two classes of constants are those that deal with numbers (numerical constants) and mode values (logical constants).

Numerical constants may be integer or real numbers, logical constants may be ON or OFF.

INTEGER CONSTANTS

Definition
Integer Constant - a whole number written without a decimal point. Maximum Magnitude: 8388607, i.e., $(2^{23}-1)$.

An integer constant may be positive, zero, or negative; if unsigned, it is assumed to be positive. Its magnitude must not be greater than the maximum and may not contain embedded commas.

Examples: Valid Integer Constants:

```
0
91
173
-12
```

Invalid Integer Constants:

```
0.0             (contains a decimal point)
27.             (contains a decimal point)
3145903612      (exceeds the allowable range)
5,396           (contains embedded comma)
```

REAL CONSTANTS

Definition
Real Constant - a number with a decimal point optionally followed by a decimal exponent. The exponent is written as the letter E followed by a signed or unsigned, one-or two-digit integer constant. The value of the exponent must be between -75 and 75 inclusive if 64-bit is used. (If 32-bit arithmetic is used the value of the exponent must be between -19 and 19 inclusive.) The total length of a real constant; including the exponent, decimal point, and optional sign must not exceed 24 nonblank characters. Magnitude: 0 or 16^{-65} through 16^{63} (i.e., approximately 10^{75}).

A real constant may be positive, zero, or negative (if unsigned, it is assumed to be positive) and must be of the allowable magnitude. It may not contain embedded commas. The decimal exponent E permits the expression of a real constant as the product of a real constant times 10, raised to a desired power.

Examples: Valid Real Constants

+0.	
-999.9999	
0.0	
5764.1	
7.0E+0	(i.e., $7.0 \times 10^0 = 7.0$)
19761.25E+1	(i.e., $19761.25 \times 10^1 = 197612.5$)
7.E3	
7.0E3	(i.e., $7.0 \times 10^3 = 7000.0$)
7.0E03	
7.0E+03	
7.0E-03	(i.e., $7.0 \times 10^{-3} = .007$)
.7E-2	
21.98753829457168	
1.0000000	

Invalid Real Constants

0	(missing a decimal point)
3,471.1	(contains embedded comma)
1.E	(missing a one- or two-digit integer constant following the E; note that it is not interpreted as 1.0×10^0)
1.2E+113	(E is followed by a 3-digit integer constant)

LOGICAL CONSTANTS

Definition
Logical Constant -- The two logical values are: ON OFF

The logical constants ON and OFF specify that the value of the logical variable they replace, or the term of the expression they are associated with, is either all ON or all OFF (64 bits), respectively (see "Logical Expressions").

SYMBOLIC NAMES

Definition
<u>Symbolic Name</u> -- from 1 through 6 alphameric (i.e., 0 through 9, or alphabetic, A through Z) characters, the first of which must be alphabetic. A list of reserved names appears in chapter 1.

Symbolic names are used in a program unit (i.e., a main program or a subprogram) to identify elements in the following classes.

- An array and the elements of that array (see "Arrays")
- A variable (see "Variables")
- An intrinsic function (see "System Subprograms")
- A FUNCTION subprogram (see "FUNCTION Subprograms")
- A SUBROUTINE subprogram (see "SUBROUTINE Subprograms")
- A block name (see "BLOCK DATA Subprograms")
- A disk area (see "DISK AREA statement")

Symbolic names must be unique within a class in a program unit and can identify elements of only one class with the exception that a FUNCTION subprogram name must also be a variable name in the FUNCTION subprogram.

Once a symbolic name is used as a FUNCTION subprogram name, a SUBROUTINE subprogram name, a block name, or an external procedure name in any unit of an executable program, no other program unit of that executable program can use that name to identify an entity of these classes in any other way.

<u>VARIABLES</u>

A CFD variable is a symbolic representation of a quantity, or quantities (i.e., a vector), that may assume different values. The value(s) of a variable may change either for different executions of a program or at different stages within the program.

For example, in the statement

```
A(*) = B(*)+5.0
```

both A and B are vector variables. The values of B are determined by some previous statement and may change from time to time; the values of A vary whenever this computation is performed with new values for B. All variables must appear in at least one nonexecutable (specification) statement, other than a DATA statement, so that they may be put in the symbol table. In addition, all CU variables must be assigned a CU memory location in an EQUIVALENCE statement.

VARIABLE NAMES

The use of meaningful variable names can serve as an aid in documenting a program. That is, someone other than the programmer may look at the program and understand its function. For example, to compute the distances some molecules have traveled in a certain length of time at a given rate of speed for each molecule, the following statement could have been written:

```
X(*) = Y(*)*Z
```

where the * in parentheses indicate vector quantities and the * outside parentheses indicates multiplication. However, it would be more meaningful to someone reading this statement if the programmer had written

```
DIST(*) = RATE(*)*TIME
```

<u>Examples</u>: Valid Variable Names:

```
B292
RATE
SQ704
```

Invalid Variable Names:

```
B292704            (contains more than 6 characters)
4ARRAY             (first character is not alphabetic)
SI.X               (contains a special character)
```

The three ways a programmer may declare the type and residence of a variable are by use of the:

1. Predefined specification contained in the CFD language,
2. IMPLICIT specification statement,
3. Explicit specification statements.

TYPE DECLARATION BY THE PREDEFINED SPECIFICATION

The predefined specification is a convention used to specify variables as CU integer or PE real:

1. If the first character of the variable name is I, J, K, L, M, or N, the variable is integer with CU residence.

2. If the first character of the variable name is any other letter, the variable is real with PE residence.

This convention is similar to the traditional FORTRAN method of implicitly specifying the type of a variable as being either integer or real.

TYPE DECLARATION BY THE IMPLICIT SPECIFICATION STATEMENT

This statement allows a programmer to specify the type and residence of variables in much the same way as was specified by the predefined convention. That is, in both, the type and residence is determined by the first character of the variable name. However, the programmer, using the IMPLICIT statement, has the option of specifying which initial letters designate a particular variable type and residence. Further, the IMPLICIT statement is applicable to all types of variables -- PE INTEGER, PE REAL, CU INTEGER, CU REAL, and CU LOGICAL.

The IMPLICIT statement overrides the variable type and residence as determined by the predefined convention. For example, if the IMPLICIT statement specifies that variables beginning with the letters A through M are PE REAL variables, and variables beginning with the letters N through Y are CU INTEGER variables, then the variable ITEM (which would be treated as a CU INTEGER variable under the predefined convention) is now treated as a PE REAL variable. Note that variables beginning with the letter Z are (by the predefined convention) treated as PE REAL variables. The IMPLICIT statement is presented in greater detail in the section "Type Statements."

TYPE DECLARATION BY EXPLICIT SPECIFICATION STATEMENTS

Explicit specification statements (PE INTEGER, PE REAL, CU INTEGER, CU REAL, and CU LOGICAL) differ from the first two ways of specifying the type of a variable, in that an explicit specification statement declares the type and residence of a particular variable by its name, rather than as a group of variables beginning with a particular character.

For example, assume:

1. That an IMPLICIT specification statement overrode the predefined convention for variables beginning with the letter I, by declaring them to be PE REAL.

2. That a subsequent explicit specification statement declared that the variable named ITEM is CU LOGICAL.

Then, the variable ITEM is CU LOGICAL and all other variables beginning with the character I are PE REAL. Note that variables beginning with the letters J through N are specified as CU INTEGER by the predefined convention.

The explicit specification statements are discussed in greater detail in the section "Type Statements."

ARRAYS

A CFD array is a set of variables identified by a single variable name. A particular variable in the array, either scalar or vector, may be referred to by its position in the array (e.g., first variable; third variable; seventh variable; first row, third plane, etc.). Consider the array named NEXT, which consists of five variables, each currently representing these values:

273, 41, 8976, 59, and 2

NEXT(1)	represents 273
NEXT(2)	represents 41
NEXT(3)	represents 8976
NEXT(4)	represents 59
NEXT(5)	represents 2

Each variable in this array consists of the name of the array (i.e., NEXT) followed by a number enclosed in parentheses, called a subscript. The variables that constitute the array are called subscripted variables. Therefore, the value of the subscripted variable NEXT(1) is 273; the value of NEXT(2), is 41; etc.

The subscripted variable NEXT(I) refers to the "<u>Ith</u>" subscripted variable in the array, where I is an integer variable that can assume a value of 1, 2, 3, 4, or 5, in this array.

To refer to the first element of an array, the array name must be subscripted; the array name does not represent the first element. The number of subscripts must correspond to the declared dimensionality, except in the EQUIVALENCE statement, which is explained in "EQUIVALENCE Statement."

Because the ILLIAC IV is a parallel processor, the most efficient way to treat arrays for which the same calculations are to be performed on each element is to ensure vector alignment. If this is done, calculations may be performed on whole (or partial) rows of data. Each row contains 64 words with one numeric value to each word. Vector aligned variables are indicated by an asterisk in the first subscript.

Example:

```
     DO 10 I = 1,5
     DO 10 J = 1,64                 (FORTRAN)
 10  A(J,I) = B(J,I)+C(J,I)

    *DO 10 I = 1,5
 10  A(*,I) = B(*,I)+C(*,I)         (CFD-code equivalent)
```

The ILLIAC IV is not only faster than many machines in each PE, but because it can perform 64 operations at once, the ILLIAC's speed advantage is multiplied. It is to the programers advantage to treat the ILLIAC not as a serial machine that can process vectors, but rather as a parallel machine that can process scalars.

SUBSCRIPTS

A subscript is an index that specifies one of the coordinates that identify a particular element of an array. From one to three subscripts are used in accordance with the dimensionality of the array being subscripted. Multidimensional subscripts are separated by commas. The subscripts, enclosed in parentheses, follow the array name.

General Form

Subscripts -- may be one of eleven forms:

General subscript (all subscripts)

+c

±i

±i±c

Vector subscript (first subscript, vector aligned PE variables only, * indicates vector usage)

*

*±c

*±i

*±i±c

Vector index subscript (second subscript, used only when first index is a vector subscript)

±v

±v±c

±v±i

±v±i±c

where v represents the absolute value of a subscripted (first subscript must be a vector subscript and second subscript must not be a vector index subscript), vector aligned, PE variable

i represents a nonsubscripted CU INTEGER variable

c represents an integer constant

The leading plus signs are optional.

Whatever subscript form is used, its evaluated result must always be greater than zero except where the vector subscript is used. In this case, the optional expression following the asterisk is a relative PE number and may be less than, equal to, or greater than zero. For example, when reference is made to the subscripted variable V(I-2), the value of I must be greater than 2. If, however, V is a vector aligned variable, reference to V(*-2) is legal under any circumstances.

Examples:

```
ARRAY (IHOLD)
NEXT(19)
MATRIX(I-5)
A(*)
B(6,10,2)
C(*-2, N(*+1,J4+1,K)+J-16,2)
```

These are valid subscripted variables for their corresponding arrays:

Array Name	Subscripted Variable
E	E(5, 100, J)
TABLE	TABLE (1, 1, 1)
S	S(*, J, K)
INVRSE	INVRSE(I+2, JOB-3,KFRAN)

Consider the following array, named LIST, consisting of two subscript parameters, the first ranging from 1 through 64, the second from 1 through 3.

Assume LIST is a vector aligned PE integer array.

	PE 1	PE 2	PE 3	PE 4	PE 5	PE 6	PE 7	...	PE 63	PE 64
Row 1	82	44	4	24	7	10	1	...	2	1
Row 2	12	2	13	16	14	2	31	...	7	10
Row 3	91	14	1	8	31	91	12	...	13	2

The correct reference for the number in PE 2, Row 3, would be

LIST (2,3)

LIST (2,3) has the value 14; LIST (4,1) has the value 24.

Ordinary mathematical notations might use LIST i,j to represent any element of the array LIST. In CFD, this is written as LIST(I,J), where I equals 1, 2, 3, ..., 64, and J equals 1, 2, or 3.

DECLARING THE SIZE OF AN ARRAY

The size of an array is determined by the number of subscript parameters of the array and the maximum value of each subscript. This information must be given for all arrays before using them in a CFD program so that a storage area of sufficient size may be reserved. Declaration of this information is made by a DIMENSION statement, a COMMON statement, or by one of the explicit specification statements (PE INTEGER, PE REAL, CU INTEGER, CU REAL, and CU LOGICAL); each is discussed in "Specification Statements."

ARRANGEMENT OF ARRAYS IN STORAGE

Arrays are stored in ascending storage locations, with the value of the first of their subscripts increasing most rapidly, and the value of the last increasing least rapidly.

Examples: The array named A, consisting of one subscript parameter, which varies from 1 to 5, appears in storage as

A(1) A(2) A(3) A(4) A(5)

The vector aligned PE array named B consists of two subscript parameters; the first subscript varies over the range from 1 to 64, and the second varies from 1 to 3. The array appears in ascending storage locations in this order

	PE 1	PE 2	PE 3	PE 4	PE 5	PE 6	...	PE 64
Row 1	B(1,1)	B(2,1)	B(3,1)	B(4,1)	B(5,1)	B(6,1)	...	B(64,1)
Row 2	B(1,2)	B(2,2)	B(3,2)	B(4,2)	B(5,2)	B(6,2)	...	B(64,2)
Row 3	B(1,3)	B(2,3)	B(3,3)	B(4,3)	B(5,3)	B(6,3)	...	B(64,3)

Note that B(1,2) and B(1,3) follow in storage B(64,1) and B(64,2), respectively although they are on different rows.

The following list is the order of an array named C that consists of three subscript parameters; the first subscript varies from 1 to 64; the second, from 1 to 2; and the third, from 1 to 3.

	PE 1	PE 2	PE 3	PE 4		PE 64
Row 1	C(1,1,1)	C(2,1,1)	C(3,1,1)	C(4,1,1)	...	C(64,1,1)
Row 2	C(1,2,1)	C(2,2,1)	C(3,2,1)	C(4,2,1)	...	C(64,2,1)
Row 3	C(1,1,2)	C(2,1,2)	C(3,1,2)	C(4,1,2)	...	C(64,1,2)
Row 4	C(1,2,2)	C(2,2,2)	C(3,2,2)	C(4,2,2)	...	C(64,2,2)
Row 5	C(1,1,3)	C(2,1,3)	C(3,1,3)	C(4,1,3)	...	C(64,1,3)
Row 6	C(1,2,3)	C(2,2,3)	C(3,2,3)	C(4,2,3)	...	C(64,2,3)

Note that C(1,2,1), C(1,1,2), C(1,2,2), C(1,1,3), and C(1,2,3) follow C(64,1,1), C(64,2,1), C(64,1,2), C(64,2,2), and C(64,1,3) respectively in storage, although they are on different rows.

EXPRESSIONS

Expressions in their simplest form consist of a single constant or variable. They may also designate a computation or show a relationship between two or more constants and/or variables. Expressions may appear in arithmetic and logical assignment statements and in certain control statements.

CFD provides two kinds of expressions: arithmetic and logical. The value of an arithmetic expression is always a number(s) whose type is integer or real. However, the evaluation of a logical expression always yields a truth value, true or false, for scalar logical expressions, or a 64 bit binary string, for vector logical expressions.

ARITHMETIC EXPRESSIONS

The simplest arithmetic expression consists of a single constant, variable, or subscripted variable of the type integer or real. If the constant, variable, or subscripted variable is of the type integer, the expression is integer. If it is of the type real, the expression is real.

Examples:

Expression	Type of Quantity	Type of Expression
3	Integer Constant	Integer
I	Integer Variable	Integer
3.0	Real Constant	Real
A	Real Variable	Real
3.14E3	Real Constant	Real
B(I)	Real Variable	Real

In the expression B(I), the subscript I , which must always represent a CU integer, does not affect the type of the expression. That is, the type of the expression is determined solely by the type of constant, variable, or subscripted variable appearing in that expression.

More complicated arithmetic expressions containing two or more constants and/or variables may be formed by using arithmetic operators that express the computations to be performed.

Arithmetic Operators

The arithmetic operators are:

Arithmetic Operator	Definition
**	Exponentiation
*	Multiplication
/	Division
+	Addition
-	Subtraction

RULES FOR CONSTRUCTING ARITHMETIC EXPRESSIONS: These are the rules for constructing arithmetic expressions that contain arithmetic operators:

1. All desired computations must be specified explicitly. That is, if more than one constant, variable, subscripted variable, or function reference (see "SUBPROGRAMS") appears in an arithmetic expression, they must be separated from one another by an arithmetic operator. For example, the two vector aligned PE variables A and B will not be multiplied if written as

   ```
   A(*)xB(*) or A(*)B(*) or A(*)·B(*)
   ```

   If multiplication is desired, then the expression must be written as

   ```
   A(*)*B(*) or B(*)*A(*)
   ```

2. No two arithmetic operators may appear in sequence in the same expression. For example, these expressions are invalid

   ```
   A(*)*/B(*) and A(*)*-B(*)
   ```

   However, in the expression, A(*)*-B(*), if the - is meant to be a minus sign rather than the arithmetic operator designating subtraction, then the expression could be written as

   ```
   A(*)*(-B(*))
   ```

   In effect, -B(*) will be evaluated first, and then A(*) will be multiplied with it (for further uses of parentheses, see Rule 6).

3. The type of an arithmetic expression is determined by the type of the operands (where an operand is a variable, constant, function reference, or another expression) in the expression. All variables and constants used in an arithmetic or comparison operation must be of the same type

because CFD supports no automatic conversion between different types. In an arithmetic statement, the type of the defined variable and the type of the arithmetic expression to the right of the equal sign must be the same because conversion of type across the equal sign is not supported.

4. The arithmetic operator denoting exponentiation (i.e.,**) may be used to raise arithmetic expressions to integer constant powers (between 2 and 10 inclusive).

5. Order of Computation: Where parentheses are omitted, or where the entire arithmetic expression is enclosed within a single pair of parentheses, the order in which the operations are performed is as follows:

Operation	Hierarchy
Evaluation of Functions (see "Subprograms")	1st (highest)
Exponentiation (**)	2nd
Multiplication and Division (* and /)	3rd
Addition and Subtraction (+ and -)	4th

Furthermore, if two operators of the same hierarchy are used consecutively, the component operations of the expression are performed from left to right. Thus the arithmetic expression A(*)/B(*)*C(*) is evaluated as if the result of the division of A(*) by B(*) was multiplied by C(*).

For example, the expression

```
(A(*)*B(*)/C(*)**2+D(*))
```

is evaluated in this order

a.	C(*)**2	Call the results X(*)	(exponentiation)
b.	A(*)*B(*)	Call the results Y(*)	(multiplication)
c.	Y(*)/X(*)	Call the results Z(*)	(division)
d.	Z(*)+D(*)	Final operation	(addition)

In CFD code, an exponent may not be raised to a power. If two or more exponents appear, parentheses must be used to force left to right processing of the expression. Also, the

value of the exponents must be between 2 and 10 inclusive. Thus the expression

```
A(*)**16
```

must be written as

```
(A(*)**4)**4      (values of exponents may vary but
                   the basic structure of the expres-
                   sion must be the same)
```

and not as

```
A(*)**2**4
```

```
(A(*)**4)**4    is evaluated as

   a. A(*)**4   Call the results Z(*)
   b. Z(*)**4   Final operation
```

6. Use of Parentheses: Parentheses may be used in arithmetic expressions, as in algebra, to specify the order in which the arithmetic operations are to be performed. Where parentheses are used, the expression within the parentheses is evaluated before the result is used.

   For example, the expression

```
(B(*)+((A(*)+B(*))*C(*))+A(*)**2)
```

   is evaluated in this order

```
a.  (A(*)+B(*))        Call the results X(*)
b.  (X(*)*C(*))        Call the results Y(*)
c.  A(*)**2            Call the results Z(*)
d.  B(*)+Y(*)+Z(*)     Final operations
```

7. Integer Division: When division is performed using two integers, the answer is truncated and an integer answer is given. For example, if I=9 and J=2, then the expression (I/J) would yield an integer answer of 4 after truncation.

8. Results of Arithmetic Expressions: The result of any arithmetic expression will be a vector quantity unless:

   a. The expression consists entirely of the addition and subtraction of one or more CU integer variables, constants, and expressions containing only these arithmetic operators and data items,

   b. The expression consists of any single real scalar variable or constant.

In these cases, the result may be a scalar.

Arrays may be referenced either as vectors (assuming the array is PE vector aligned), if the first subscript is a vector subscript, or as scalars, if the first subscript is a general subscript. (See "Subscripts" for definitions of vector and general subscripts.)

The following list illustrates legal and illegal assignment statements using scalar and vector variables and expressions.

Variable	= Expression	Legality	
vector	= vector	legal	computation in PE's
vector	= scalar	legal	computation in PE's
scalar	= scalar	legal	computation in CU†
scalar	= vector	illegal	

LOGICAL EXPRESSIONS

The simplest form of logical expression consists of a single logical constant, logical variable, or logical subscripted variable, the value of which is always a 64 bit binary string.

More complicated logical expressions may be formed by using logical and relational operators. These expressions may be in one of two forms:

†This type of assignment is slow and should be avoided if possible.

1. Relational operators combined with arithmetic expressions whose type is integer or real,

2. Logical, bit setting, and bit shifting operators and logical quantifiers combined with logical constants, logical variables, or vector logical expressions.

Item 1 is discussed in the following section, "Relational Operators"; item 2 is discussed in "Operators and Quantifiers."

Relational Operators

The six relational operators, each of which must be preceded and followed by a period, are:

Relational Operator	Definition
.GT.	Greater than (>)
.GE.	Greater than or equal to (≥)
.LT.	Less than (<)
.LE.	Less than or equal to (≤)
.EQ.	Equal to (=)
.NE.	Not equal to (≠)

The relational operators express an arithmetic condition which can result in either a 64 bit binary string or a truth value. Those conditions that result in binary strings are vector relational expressions and those that result in truth values (i.e. true or false) are scalar relational expressions.

All comparisons between arithmetic expressions involving only the addition and subtraction of CU integer variables and constants are performed in the CU and are scalar relational expressions. All other comparisons, including those involving single PE scalars, are done in the PE's and are vector relational expressions. If the comparison is performed in the CU, the result is a truth value. If the comparison is performed in the PE's, the result is a 64-bit binary string. This binary string consists of 64 truth values, one for each PE. If the comparison is true in PE_n, bit n of the result is set to 1. If the comparison is false, bit n is set to 0. Only arithmetic expressions whose type is integer or real may be combined by relational operators. For example, assume that the types of these variables have been specified as

Variable Names	Type
ROOT, E	Real variables: ROOT is a vector, E resides in CU
A, I, F	Integer variables: F is a vector, A and I reside in CU
L	Logical variable

Then, the following examples illustrate scalar and vector relational expressions using the relational operators.

Examples: Relational Expressions Using Relational Operators:

Scalar Relational Expressions (no parentheses allowed)

```
I.LT.2
I.GT.A+32
I+A.EQ.0
```

Vector Relational Expressions (the outermost parentheses are required)

```
(ROOT(*)*E.GT.2.714)
(E**2.LT.0.0)
(F(*).GE.F(1))
(F(*).NE.F(I))
(E.GT.97.420E-3)
(ROOT(*).LE.0.0)
(I.LT.2)
```

Only those conditions involving arithmetic expressions consisting solely of addition and subtraction of CU integer variables and constants can be scalar relational expressions. All others will be vector relational expressions. Unlike arithmetic expressions, the use of single PE scalars results in a vector relational expression.

Operators and Quantifiers

Logical Operators

The three logical operators, each of which must be preceded and followed by a period are as follows. $\underline{a}$ and $\underline{b}$ represent logical variables, or vector relational expressions; they are 64 bit binary strings. $\underline{a}_n$ and $\underline{b}_n$ refer to bit n of $\underline{a}$ and $\underline{b}$. $\underline{a}$ and $\underline{b}$ are not changed by logical operators.

Logical Operator	Definition
.NOT. $\underline{a}$	If $\underline{a}_n$ is 1, then .NOT. $\underline{a}_n$ is 0. If $\underline{a}_n$ is 0, then .NOT. $\underline{a}_n$ is 1.
$\underline{a}$ .AND. $\underline{b}$	If $\underline{a}_n$ and $\underline{b}_n$ are both 1, then $\underline{a}_n$ .AND. $\underline{b}_n$ is 1. If either $\underline{a}_n$ or $\underline{b}_n$ or both are 0, then $\underline{a}_n$ .AND. $\underline{b}_n$ is 0.

$\underline{a}$ .OR. $\underline{b}$	If $\underline{a}_n$ is 1 or $\underline{b}_n$ is 1 or $\underline{a}_n$ and $\underline{b}_n$ are both 1, then $\underline{a}_n$ .OR. $\underline{b}_n$ is 1. If $\underline{a}_n$ and $\underline{b}_n$ are both 0, then $\underline{a}_n$ .OR. $\underline{b}_n$ is 0.

The operand(s) and result of all three logical operators are 64-bit binary strings.

Two logical operators may appear in sequence only if the second one is the logical operator .NOT..

Bit-Setting Operators

The two bit-setting operators, both of which must be preceded and followed by a period, are as follows. $\underline{a}$ represents a logical constant or variable, or a vector relational expression. $\underline{a}$ is a 64-bit binary string. $\underline{c}$ is a range of PE's. The definition of a range of PE's follows the discussion of the bit setting operators. $\underline{a}$ is not changed by bit setting operators.

Operator	Definition
$\underline{a}$ .TURN ON. $\underline{c}$	Turn on (set to 1) all bits of $\underline{a}$ that are specified in range $\underline{c}$. A specified bit may already be on; this has no adverse effect on the execution of this operation.
$\underline{a}$ .TURN OFF. $\underline{c}$	Turn off (set to 0) all bits of $\underline{a}$ that are specified in range $\underline{c}$. A specified bit may already be off; this has no adverse effect on the execution of this operation.

A range of PE's is defined as a list of range elements, each of which specifies a single PE, or group of adjacent PE's to be turned on or off. The basic structure of the range element is the PE number. A PE number takes the form of a general subscript. It's value when evaluated must lie between 1 and 64 inclusive. Range elements consist of PE numbers, or range specifiers. There are three range specifiers, each of which must be preceded and followed by periods. The four kinds of range elements and their definitions are listed below. $\underline{d}$ and $\underline{e}$ represent PE numbers.

Range Elements	Definition
$\underline{d}$	Evaluate $\underline{d}$ and set $bit_{\underline{d}}$ according to the bit setting operator specified.
.FIRST. $\underline{d}$	Evaluate $\underline{d}$ and set all bits between bit_1 and $bit_{\underline{d}}$ inclusive according to the bit setting operator specified.

.LAST. *d*	Evaluate *d* and set all bits between $bit_{(64-d+1)}$ and bit_{64} inclusive according to the bit setting operator specified.
d .TO. *e*	Evaluate *d* and *e* (this range element is undefined if *d* is greater than *e*) and set all bits between bit_d and bit_e inclusive according to the bit setting operator specified.

Either bit setting operator must be followed by a list of one or more range elements. If more than one range element is provided, they must be separated by commas. The bit setting operator .TURN ON. may not be used in conjunction with the logical constant ON and .TURN OFF. may not be used with OFF. The result of a bit setting operation is a 64-bit binary string.

Bit-Shifting Operators

The four bit-shifting operators, each of which must be preceded and followed by a period, are listed below. *a* represents a logical constant or variable, or a vector relational expression. *f* represents a shift distance. A shift distance is identical in syntax to a general subscript. The value of *f* is computed modulo 64 and the result is between 0 and 63 inclusive. *a* is not changed by bit-shifting operators.

Operator	Definition
a .SHL. *f*	Shift *a* (a 64 bit binary string) left (end off) *f* bits, filling on the right with zeros.
a .SHR. *f*	Shift *a* right (end off) *f* bits, filling on the left with zeros.
a .RTL. *f*	Rotate *a* left (end around) *f* bits. Bits rotated off the left end reappear on the right end.
a .RTR. *f*	Rotate *a* right (end around) *f* bits. Bits rotated off the right end reappear on the left end.

The logical constant OFF may not be used in conjunction with bit-shifting operators; ON may be used only with .SHL. and .SHR.. The result of a bit-shifting operation is a 64-bit binary string.

Logical Quantifiers

Logical quantifiers are used to reduce a 64-bit binary string to a single truth value. Since the result of a comparison in the PE-s is a 64-bit binary string, a logical quantifier

must be used if a branch in the program is to be controlled by such a comparison. $\underline{g}$ stands for a logical variable, a vector relational expression, or a vector logical expression (logical expressions will be defined later). (All of the items listed above are in fact vector logical expressions.) $\underline{g}$, of course, is in the form of a 64-bit binary string. The four logical quantifiers are as follows: $\underline{g}_n$ refers to bit n of $\underline{g}$. $\underline{g}$ is not changed by logical quantifiers.

Quantifier	Definition
.ANY.($\underline{g}$)	If $\underline{g}_n$=1 for any n then .ANY.($\underline{g}$) is true. If $\underline{g}_n$=0 for all n then .ANY.($\underline{g}$) is false. .ANY. is equivalent to a logical OR of all 64 bits.
.ALL.($\underline{g}$)	If $\underline{g}_n$=1 for all n then .ALL.($\underline{g}$) is true. If $\underline{g}_n$=0 for any n then .ALL.($\underline{g}$) is false. .ALL. is equivalent to a logical AND of all 64 bits.
.NOT ANY.($\underline{g}$)	If $\underline{g}_n$=1 for any n then .NOT ANY.($\underline{g}$) is false. If $\underline{g}_n$=0 for all n then .NOT ANY.($\underline{g}$) is true. .NOT ANY. is equivalent to a logical AND of the logical complements of all 64 bits.
.NOT ALL.($\underline{g}$)	If $\underline{g}_n$=1 for all n then .NOT ALL.($\underline{g}$) is false. If $\underline{g}_n$=0 for any n then .NOT ALL.($\underline{g}$) is true. .NOT ALL. is equivalent to a logical OR of the logical complements of all 64 bits.

The parentheses surrounding the operand of a logical quantifier are required. See page 1.15 for restrictions in the use of quantifiers.

Logical Expressions

There are two kinds of logical expressions in CFD code. Scalar logical expressions control the flow through the program and vector logical expressions control the pattern of enabled and disabled PE's during computation. No more than one logical, bit shifting, or bit setting operator may appear in a single logical expression. Also, only one logical quantifier may appear. This restriction does not apply to the logical operator .NOT..

General Form

Vector logical expressions

a
a .AND. b
a .OR. b
a .TURN ON. c
a .TURN OFF. c
a .SHL. f
a .SHR. f
a .RTL. f
a .RTR. f

where a and b are logical variables or constants, or vector relational expressions (remember the required parentheses). They are 64-bit binary strings. a or b may be the logical complement of a or b (.NOT. a, .NOT. b).
c is a range of PE's.
f is a shift distance (in general subscript syntax).

Scalar logical expressions

h
.ANY.(g)
.ALL.(g)
.COM. n
.NOT ANY.(g)
.NOT ALL.(g)
.NOT COM. n

where h is a scalar relational expression (no parentheses allowed).
g is a vector logical expression
n is an I/O request number

ALL OF THE ABOVE GENERAL FORMS ARE SUBJECT TO THE RESTRICTIONS GIVEN IN THE DISCUSSION OF EACH OPERATOR OR QUANTIFIER.

Only those expressions which, when evaluated, have the form of a 64-bit binary string may be combined with the operators and quantifiers mentioned above to form logical expressions.

Assume that the types of the variables are specified as

Variable names	Type
ROOT, E,X,Y	Real variables: ROOT is a vector; X,Y reside in CU
A,I,F	Integer variables: F is a vector; A,I reside in CU
L,M	CU logical variables

Then the following illustrates both scalar and logical expressions. All parentheses appearing in these examples are required.

Examples: Scalar Logical Expressions

```
I.LT.2
I.GT.A+32
I+A.EQ.0
.ANY.((ROOT(*)*E.GT.2.714))
.NOT ALL.(L)
.NOT ANY.(MODE)
.ALL.((X.GT.Y))          (comparison of CU real variables is
                          done in PE's)
.ALL.((F(*).EQ.F(1)))
.NOT ALL.((ROOT(*).GT.0.0).OR.M)
.NOT ANY.((E.GT.ROOT(*)).AND.MODE)
```

Examples: Vector Logical Expressions

```
(F(*).LE.0).TURN OFF. 1, 64, A.TO.32
(ROOT(*).GT.3.14159).AND.L
MODE.OR.L
(X.GT.Y).AND..NOT.MODE
.NOT.(F(*).EQ.0)
ON.SHL. A-6
OFF.TURN ON. A .TO. I
MODE.RTL. 2
L.OR..NOT. (ROOT(*-I)+X.GT.E**2-Y)
MODE
```

Order of Computations in Logical Expressions

The heirarchy of operations in a logical expression is given below. Operations with higher hierarchy (smaller numbers) are performed first. No more than one logical, bit shifting or bit setting operator may appear in a logical expression. Also, only one logical quantifier may appear.

Operation	Hierarchy
Evaluation of functions	1st (highest)
Exponentiation (**)	2nd
Multiplication and division (* and /)	3rd
Addition and subtraction (+ and -)	4th
.LT., .LE., .EQ., .GE., .GT., .NE.	5th
.NOT.	6th
.AND., .OR., .TURN ON., .TURN OFF., .SHL., .SHR., .RTL., .RTR.	7th
.ANY., .ALL., .NOT ANY., .NOT ALL.	8th (lowest)

Use of Parentheses in Logical Expressions

Parentheses must be used to surround both vector relational expressions and the operand of a logical quantifier. If a vector relational expression is quantified, two sets of parentheses are required.

Example:

```
IF(.ANY.((RES.GT.EPSLON))) GO TO 5
```

Aside from these uses, parentheses are prohibited except for use within vector relational expressions to alter the order of arithmetic operation.

Evaluation of Vector Relational Expressions

Vector relational expressions are evaluated in all PE's unless explicitly masked by the programmer.

Example:

```
LOGICL = (A(*)/B(*).GT.C(*))
```

The operations are carried out in all PE's. If A(*), B(*), and C(*) had been defined under the mode pattern MASK, then they are undefined in PE's that were not enabled by MASK. This could lead to overflow of A(*)/B(*) and certainly leads to meaningless results in those PE's. This possibility can be avoided by use of the statement

```
LOGICL = MASK.AND.(A(*)/B(*).GT.C(*))
```

ARITHMETIC AND LOGICAL ASSIGNMENT STATEMENT

General Form	
a = b	
where	a is any subscripted or nonsubscripted variable
	b is any arithmetic or logical expression
Note:	Variable a must be of the same type as expression b. That is, if a is a real variable then b must be a real expression. If a is a logical variable then b must be a vector logical expression (logical variables do not store truth values, only 64 bit mode patterns). If a is integer then b must be integer. If a is a vector, it's first subscript must consist entirely of an asterisk. THE DEFINED VARIABLE IN AN ARITHMETIC STATEMENT MUST NOT BE A ROTATED VECTOR.

The CFD arithmetic and logical assignment statements closely resemble conventional algebraic equations; however, the equal sign of the CFD arithmetic statement specifies replacement rather than equivalence. That is, the expression to the right of the equal sign is evaluated, and the resulting value replaces the current value of the variable to the left of the equal sign.

Assume that the type of the following variables has been specified as:

Variable Names	Type
I, J, W	Integer variables: J, W are vectors, I resides in the CU
A, B, C, D	Real variables: A is a vector
G, H	CU logical variables (64 bits)

Then, the following examples illustrate valid arithmetic and logical assignment statements using constants, variables, and subscripted variables of various types.

Statements	Description
A(*) = B	The values of A are replaced by the current value of B.
I = I + 1	The value of I is replaced by the value of I + 1.
A(*) = C*D	The most significant part of the product of C and D replaces the values of A.
G = ON	The value of G is replaced by the logical constant ON.
H = .NOT.G	H is replaced by the logical complement of G (see "Logical Operators").
G = (A(*).GT.3.0)	For each PE_n where $A > 3.0$ set bit_n of G to 1. For all others, $bit_n = 0$.
H = G.AND..NOT.(A(*).GT.B)	For each PE_n where $A > B$ set bit_n of a temporary result to 1. The result of a logical AND between G and the logical complement of this temporary result is H.
MODE = ON.TURN OFF.I	All PE's except PE_I are enabled.
G = H.OR.(C.LT.D)	Bit_n is set to 1 for all PE's in which either $C < D$ or bit_n of H is 1.
MODE = MODE.RTL.2	PE_n assumes the enabled/disabled status of PE_{n+2} (end around).

Note: The use of MODE, MBIT1, or MBIT2 on the right side of a logical assignment statement when the left side is a CU logical variable inhibits overlap and should be avoided if possible. On the other hand, the use of MODE, MBIT1, or MBIT2 on the left side of a logical assignment statement is preferable to the use of a CU logical variable when the right side is a vector relational expression.

Normally, CFD statements are executed sequentially; that is, after one statement has been executed, the statement immediately following it will be executed. This section discusses the statements that may be used to alter and control the normal sequence of statement execution in the program. Also, the Vector Logical IF statement as a means of temporarily changing the mode is discussed here because of syntactic similarities with the Scalar Logical IF statement.

GO TO STATEMENTS

The GO TO statements cause control to be transferred to the statement specified by a statement number. The three GO TO statements are: unconditional GO TO, computed GO TO, and assigned GO TO. Every time the same unconditional GO TO statement is executed, a transfer to the same specified statement is made. However, the computed and assigned GO TO statements cause control to be transferred to one of several statements, depending upon the current value of a particular variable.

Unconditional GO TO Statement

General Form
GO TO X where X is an executable statement number

This GO TO statement causes control to be transferred to the statement specified by the statement number. Every subsequent execution of this GO TO statement results in a transfer to that same statement.

Example:

```
50*GO TO 25
10 A(*)=B(*)+C(*)
       :
25 C(*) = E(*)**2
       :
```

Explanation: Every time statement 50 is executed, control is transferred to statement 25. Note that the statement following an unconditional GO TO must be numbered.

Computed GO TO Statement

General Form
GO TO ($\underline{x}_1$, $\underline{x}_2$, $\underline{x}_3$, ...,$\underline{x}_n$), $\underline{i}$ where $\underline{x}_1$, $\underline{x}_2$,...,$\underline{x}_n$ are executable statement numbers, $\underline{i}$ is a nonsubscripted CU integer variable

This statement causes control to be transferred to the statement numbered $\underline{x}_1$, $\underline{x}_2$, $\underline{x}_3$,..., or $\underline{x}_n$, depending on whether the current value of $\underline{i}$ is 1, 2, 3, ..., or n, respectively. If the value of $\underline{i}$ is outside the allowable range, the next statement is executed.

Example:

```
    *GO TO (25, 10, 50, 7), ITEM
 100*CALL SUBROT
       .
       .
       .
  50 A(*) = B(*)+C(*)
       .
       .
       .
   7 C(*) = E(*)**2+A(*)
       .
       .
       .
  25 L = (C(*).GT.D(*)).AND.L
       .
       .
       .
  10 B(*) = 21.3E02
```

Explanation: If the value of the integer variable ITEM is 1, statement 25 will be executed next. If ITEM is equal to 2, statement 10 will be executed next, and so on. If ITEM is outside the range 1,4 statement 100 is executed next.

ASSIGN and Assigned GO TO Statements

General Form
ASSIGN $\underline{i}$ TO $\underline{m}$ . . . GO TO $\underline{m}$, $(\underline{x}_1, \underline{x}_2, \underline{x}_3, \ldots, \underline{x}_n)$ where $\underline{i}$ is an executable statement number, $\underline{x}_1, \underline{x}_2, \underline{x}_3, \ldots, \underline{x}_n$ are executable statement numbers, $\underline{m}$ is a nonsubscripted CU integer variable to which is assigned one of these statement numbers: $\underline{x}_1$, $\underline{x}_2, \underline{x}_3, \ldots \underline{x}_n$.

The assigned GO TO statement causes control to be transferred to the statement numbered $\underline{x}_1, \underline{x}_2, \underline{x}_3, \ldots$, or $\underline{x}_n$, depending on whether the current assignment of $\underline{m}$ is $\underline{x}_1, \underline{x}_2, \underline{x}_3, \ldots$, or $\underline{x}_n$. For example, in the statement

```
*GO TO N, (10, 25, 8)
```

if the current assignment of the integer variable N is statement 8, that statement is executed next. If the current assignment of N is statement 10, that statement is executed next. Similarly, if N is assigned statement number 25, that statement is executed next.

The current assignment of the integer variable $\underline{m}$ is determined by the last ASSIGN statement executed. Only an ASSIGN statement may be used to initialize or change the value of the integer variable $\underline{m}$. The value of the integer variable $\underline{m}$ is not the integer statement number; ASSIGN 10 TO I is not the same as I = 10.

Example 1:

```
       .
       .
       .
    *ASSIGN 50 TO NUMBER
  10*GO TO NUMBER, (35, 50, 25, 12, 18)
  35 A(*) = 1.0
       .
       .
       .
  50 A(*) = B(*)+C(*)
       .
       .
       .
```

Computed GO TO Statement

General Form
GO TO (x_1, x_2, x_3, ...,x_n), i where x_1, x_2,...,x_n are executable statement numbers, i is a nonsubscripted CU integer variable

This statement causes control to be transferred to the statement numbered x_1, x_2, x_3,..., or x_n, depending on whether the current value of i is 1, 2, 3, ..., or n, respectively. If the value of i is outside the allowable range, the next statement is executed.

Example:

```
    *GO TO (25, 10, 50, 7), ITEM
 100*CALL SUBROT
       .
       .
       .
  50 A(*) = B(*)+C(*)
       .
       .
       .
   7 C(*) = E(*)**2+A(*)
       .
       .
       .
  25 L = (C(*).GT.D(*)).AND.L
       .
       .
       .
  10 B(*) = 21.3E02
```

Explanation: If the value of the integer variable ITEM is 1, statement 25 will be executed next. If ITEM is equal to 2, statement 10 will be executed next, and so on. If ITEM is outside the range 1,4 statement 100 is executed next.

ASSIGN and Assigned GO TO Statements

General Form
ASSIGN $\underline{i}$ TO $\underline{m}$ . . . GO TO $\underline{m}$, $(\underline{x}_1,\underline{x}_2,\underline{x}_3,\ldots,\underline{x}_n)$ where $\underline{i}$ is an executable statement number, $\underline{x}_1,\underline{x}_2,\underline{x}_3,\ldots,\underline{x}_n$ are executable statement numbers, $\underline{m}$ is a nonsubscripted CU integer variable to which is assigned one of these statement numbers: $\underline{x}_1$, $\underline{x}_2,\underline{x}_3,\ldots\underline{x}_n$.

The assigned GO TO statement causes control to be transferred to the statement numbered $\underline{x}_1,\underline{x}_2,\underline{x}_3,\ldots,$or $\underline{x}_n$, depending on whether the current assignment of $\underline{m}$ is $\underline{x}_1,\underline{x}_2,\underline{x}_3,\ldots,$or $\underline{x}_n$. For example, in the statement

```
*GO TO N, (10, 25, 8)
```

if the current assignment of the integer variable N is statement 8, that statement is executed next. If the current assignment of N is statement 10, that statement is executed next. Similarly, if N is assigned statement number 25, that statement is executed next.

The current assignment of the integer variable $\underline{m}$ is determined by the last ASSIGN statement executed. Only an ASSIGN statement may be used to initialize or change the value of the integer variable $\underline{m}$. The value of the integer variable $\underline{m}$ is not the integer statement number; ASSIGN 10 TO I is not the same as I = 10.

Example 1:

```
       .
       .
       .
   *ASSIGN 50 TO NUMBER
 10*GO TO NUMBER, (35, 50, 25, 12, 18)
 35 A(*) = 1.0
       .
       .
       .
 50 A(*) = B(*)+C(*)
       .
       .
       .
```

Explanation: Statement 50 is executed immediately after statement 10. Note that the statement following an assigned GO TO must be numbered.

Example 2:

```
        .
        .
    *ASSIGN 10 TO ITEM
        .
        .
  13*GO TO ITEM, (8, 12, 25, 50, 10)
  12 A(*) = 0.0
        .
        .
   8 A(*) = B(*)+C(*)
        .
        .
  10 B(*) = C(*)+D(*)
    *ASSIGN 25 TO ITEM
    *GO TO 13
        .
        .
  25 C(*) = E(*)**2
        .
        .
```

Explanation: The first time statement 13 is executed, control is transferred to statement 10. On the second execution of statement 13, control is transferred to statement 25.

THE IF STATEMENTS

There are two kinds of IF statements: the Scalar Logical IF statement and the Vector Logical IF statement.

The Scalar Logical IF statement enables the user to evaluate a scalar logical expression and execute or skip a specified statement depending on whether or not the scalar logical expression is true or false. The mode is not changed by a Scalar Logical IF statement.

The Vector Logical IF statement enables the user to set the mode temporarily to the 64-bit result of a vector logical expression. After the mode is set, the Vector Arithmetic Assignment statement specified is executed (unlike a Scalar Logical IF, the specified statement is always executed) and the old mode is then restored.

Scalar Logical IF Statement

General Form
IF(a)s where a is a scalar logical expression s is any statement except a specification statement, DO statement, or another logical IF statement

The Scalar Logical IF statement is used to evaluate the logical expression a and to execute or skip statement s, depending on whether the expression is true or false, respectively.

Example 1:

```
       .
       .
       .
  5*IF(J.LE.0) GO TO 25
 10 C(*) = D(*)+E(*)
 15*IF(.ANY.((A(*).GT.B(*))))C(*)=2.0*A(*)/C(*)
 20 MODE = G.AND.H
       .
       .
       .
 25 W(*) = X(*)**2
       .
       .
       .
```

Explanation: In statement 5, if the expression is true (i.e., J is less than or equal to 0), the statement GO TO 25 is executed next, and control is passed to statement 25. If the expression is false (i.e., J is greater than 0), the statement GO TO 25 is ignored, and control is passed to statement 10.

In statement 15, if the value of the expression is true (i.e., A is greater than B in any PE), the values of C(*), in PE's enabled by MODE, are replaced by the values of the expression 2.0*A(*)/C(*), and statement 20 is executed. If the value of the expression is false, the values of C(*) remain unchanged, and statement 20 is executed next. Note that the vector relational expression is surrounded by parentheses and that the operand of an .ANY. or .ALL. operation must be surrounded by parentheses. When a vector relational expression is used as the operand of an .ANY. or .ALL. both pairs of parentheses are required.

Example 2: Assume that P and Q are logical variables.

```
        .
        .
        .
  5*IF(.ANY.(P.OR..NOT.Q))A(*) = B(*)
 10 C(*) = B(*)**2
        .
        .
        .
```

Explanation: In statement 5, if the expression is true, the statement A(*)=B(*) is executed next, for PE's enabled by MODE, and control continues to statement 10. If the expression is false, the statement A(*)=B(*) is skipped and statement 10 is executed.

Example 3: Assume that A and B are vector aligned PE arrays

```
        .
        .
        .
 10*IF(.ALL.((A(*).GT.B(*))))A(*) = A(*)-B(*)
 20 B(*)=B(*)*2.0
```

Explanation: In statement 10, if the expression is true, the statement A(*)=A(*)-B(*) is executed, for PE's enabled by MODE, and control passes to statement 20. If the expression is false, control passes directly to statement 20 and the statement A(*) = A(*)-B(*) is skipped.

Vector Logical IF Statement

General Form	
IF(a)s	
where	a is a vector logical expression
	s is a vector arithmetic assignment statement

The Vector Logical IF statement is used to evaluate the logical expression a and to execute statement s using the result of a (a 64-bit binary string) as the temporary mode.

Example 1:

```
        .
        .
    *IF((A(*).LT.0.0))A(*) = 0.0
        .
        .
        .
```

Explanation: The value of A is set to 0.0 in all PE's in which A was originally negative. Note that the parentheses

surrounding the vector logical expression and the parentheses in the vector relational expression are both required.

Example 2: Assume L and M are CU logical

```
        .
        .
        .
    *IF (.NOT.L.OR.M)A(*) = B(*)
        .
        .
        .
```

Explanation: The complement of L is logically OR'ed with M (both are 64-bit binary strings) and the statement A(*)=B(*) is executed using the result of the OR as the mode (A is set equal to B in every PE_n, where bit n of M is on or bit n of L is off).

Example 3:

```
        .
        .
        .
    *IF(.NOT.MODE)A(*) = 0.0
        .
        .
        .
```

Explanation: A is set to zero in all PE's disabled by MODE.

DO Statement

General Form

	End of range	DO variable		Initial value	Test value	Increment
DO	$\underline{x}$	$\underline{i}$	=	$\underline{m}_1$,	$\underline{m}_2$,	$\underline{m}_3$

where $\underline{x}$ is the statement number of an executable statement that follows the DO statement

$\underline{i}$ is a nonsubscripted CU integer variable ($\underline{i}$ must never be less than or equal to zero)

$\underline{m}_1$ and $\underline{m}_2$ are signed or unsigned CU integer variables or integer constants, or the sum or difference of a CU integer and a constant ($\pm\underline{v}$, $+\underline{c}$, $\pm\underline{v}\pm\underline{c}$ where $\underline{v}$ is a CU integer and $\underline{c}$ is an integer constant). When $\underline{m}_1$ $\underline{m}_2$ are evaluated, both results must be greater than zero. $\underline{m}_3$ must be an integer scalar constant whose sign is that of the expression ($\underline{m}_2$-$\underline{m}_1$). ($\underline{m}_3$, is optional; if it is omitted, its value is assumed to be +1. In this case, the preceding comma must also be omitted.)

The DO statement is a command to execute repeatedly the statements that follow, up to and including the statement numbered $\underline{x}$. The range of a DO is that set of statements that will be executed repeatedly; i.e., it is the sequence of consecutive statements immediately following the DO statement. The first time the statements in the range of the DO are executed, $\underline{i}$ is initialized to the value $\underline{m}_1$; each succeeding time $\underline{i}$ is incremented by the value $\underline{m}_3$. The statements within the range of the DO loop will be executed for all values of $\underline{i}$ that are between m_1 and m_2 inclusive. The test is performed at the end of the loop; therefore, all DO loops are executed at least once. Deliberately specifying a DO loop in which the sign of the increment is wrong is an error. For example:

```
10*DO 40 I = L,10,-1        (where L=2)
20*DO 40 J = 2,16,-1
30*DO 40 K = 16,2
40*CONTINUE
```

In the above example, all three DO statements are in error. Statement 10 will compile, however, as the value of L cannot be known at compile time. If statement 10 was executed, the statements within the range of the DO loop would be executed once. Statements 20 and 30, however, are recognized as errors.

Programming Considerations in Using a DO Loop

1. The index of a DO statement ($\underline{i}$) may not be changed by a statement within the range of the DO loop, or by any subprograms that are called within the range of a DO loop.

2. A DO statement may contain other DO statements within its range. All statements in the range of an inner DO must be in the range of the outer DO. A set of DO statements satisfying this rule is called a nest of DOs.

   Example 1:

```
      *DO 50 I = 1, 4                                    }
       A(*,I) = B(*,I)**2                                } Range of
      *DO 50 J = 1, 5        } Range of                  }   outer DO
    50 C(*,J+1) = A(*,I)     }   inner DO                }
```

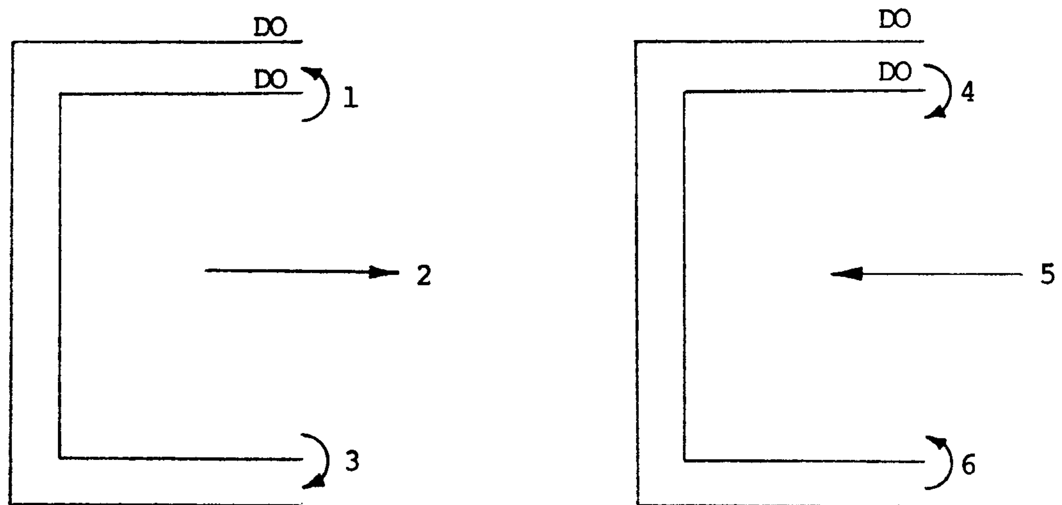
Example 2:

```
  *DO 10 INDEX = L, M
   N = INDEX + K                                                    )
  *DO 15 J = 1, 100, 2                    ) Range of               ) Range of
15 TABLE(*,J,N) = SUM(*,J,N)-1.           )   inner DO              )  outer DO
10 B(*,N) = A(*,N)
```

3. A transfer out of the range of any DO loop is permissible at any time.

4. If, and only if, a transfer is made from the range of an innermost DO loop, transfer back into that loop is allowed, provided the index (i) is not changed outside the range. A transfer back into the range of any other DO within a nest of DOs is not permitted.

Example:



Explanation: The transfers specified in the example by the numbers 1, 2, and 3 are permissible; those specified by 4, 5, and 6 are not.

5. The index (i) may be changed by statements outside the range of the DO statement only if no transfer is made back into the range of the DO statement.

6. The last statement in the range of a DO loop (statement x) may not be a GO TO, STOP, RETURN, or another DO statement. Also, the last statement may not be a logical IF statement containing any of those statements.

CONTINUE Statement

General Form
CONTINUE

CONTINUE is a dummy statement which may be placed anywhere in the source program without affecting the sequence of execution. It may be used as the last statement in the range of a DO statement to avoid ending the DO loop with any of the statements that are not permitted as the last statement in the range of a DO.

Example 1:

```
            .
            .
            .
      *DO 30 I = 1, 20
       A(*,I) = B(*+I)
      *IF(.ANY.((A(*,I).LT.0.0))) RETURN
   30 *CONTINUE
       C(*) = A(*,1)*B(*)
            .
            .
            .
```

Explanation: The CONTINUE statement is used as the last statement in the range of the DO statement, to avoid ending the DO loop with an IF statement containing a RETURN.

Example 2:

```
            .
            .
            .
      *DO 30 I=1,20
      *IF(.ANY.((A(*,I).GT.B(*,I))))GO TO 40
       A(*,I) = B(*,I)
      *GO TO 30
    40 A(*,I) = 0.0
    30*CONTINUE
            .
            .
            .
```

Explanation: The CONTINUE statement provides a branch point that enables the programmer to bypass the execution of statement 40.

STOP Statement

General Form
STOP

This statement terminates the execution of the object program.

END Statement

General Form
END

The END statement is a nonexecutable statement that defines the end of a source program or source subprogram for the compiler. Physically, it must be the last statement of each program or subprogram. Execution of an END statement is equivalent to execution of a STOP statement.

The specification statements provide the compiler with information about the nature of the data used in the source program. In addition, they supply the information required to allocate storage locations for this data. Specification statements describing data must appear at the beginning of the source program, and must precede any statements which refer to that data. The specification statements are the IMPLICIT, Type, DIMENSION, COMMON, SCRATCH, EQUIVALENCE, and DATA statements. All variables used in the program must appear in at least one of these statements, other than the DATA statement.

THE TYPE STATEMENTS

There are two kinds of type statements: the IMPLICIT specification statement and the explicit specification statements.

The IMPLICIT specification statement enables the user to specify the type and residence (CU or PE) of a group of variables or arrays according to the initial character of their names.

The explicit specification statements enable the user to

1. Specify the type and residence (CU or PE) of a variable or array according to its particular name.

2. Specify the dimensions of an array.

IMPLICIT Statement

General Form
IMPLICIT <u>type</u> ($\underline{a}_1$,$\underline{a}_2$,...),...,<u>type</u> ($\underline{a}_1$,$\underline{a}_2$,...) where <u>type</u> represents one of the following: CU INTEGER PE INTEGER CU REAL PE REAL CU LOGICAL $\underline{a}_1$,$\underline{a}_2$,...represent single alphabetic characters each separated by commas, or a range of characters (in alphabetic sequence) deonted by the first and last characters of the range separated by a hyphen (e.g., (A-D))

IMPLICIT statements, if specified, should be the first statements in a program, except for the FUNCTION, SUBROUTINE, or BLOCK DATA statements in a subprogram.

The IMPLICIT type statement enables the user to declare the type and residence of the variables appearing in his program by specifying that variables beginning with certain designated letters are of a certain type.

Example 1:

```
*IMPLICIT PE REAL (A-H, O-Z), CU INTEGER (I-N)
```

Explanation: All variables beginning with the characters I through N are declared as CU INTEGER.

All other variables (those beginning with the characters A through H and O through Z) are declared as PE REAL.

Note that the statement in Example 1 performs the same function of typing variables as the predefined convention (see "Type Declaration by the Predefined Specification").

Example 2:

```
*IMPLICIT CU INTEGER (A-H), CU REAL (I-K), CU LOGICAL (L,M,N)
```

Explanation: All variables beginning with the characters A through H are declared as CU INTEGER. All variables beginning with the characters I through K are declared as CU REAL. All variables beginning with the characters L, M, and N are declared as CU LOGICAL.

Since the remaining letters of the alphabet (O through Z) were left undefined by the IMPLICIT statement, the predefined convention will take effect. Thus, all variables beginning with the characters O through Z are declared as PE REAL.

Explicit Specification Statements

General Form

type $\underline{a}(\underline{k}_1),\underline{b}(\underline{k}_2),\ldots,\underline{z}(\underline{k}_n)$

where *type* is

CU INTEGER	PE INTEGER
CU REAL	PE REAL
CU LOGICAL	

$\underline{a},\underline{b},\ldots,\underline{z}$ represent variable, array, or function names (see "SUBPROGRAMS")

$(\underline{k}_1)$, $(\underline{k}_2)$,...,$(\underline{k}_n)$ are optional; each $\underline{k}$ is composed of one of the following array allocation specifications:

i_1

$*$ } PE variables only

$*,i_2$ } PE variables only

$*,i_2,i_3$ } PE variables only

in which i_1,i_2, and i_3 are unsigned integer constants. If the first index is an asterisk, the variable must reside in the PE's because an asterisk denotes vector alignment. In vector aligned arrays, the range of the first index is taken to be 64. Only PE arrays may be vector aligned, or have more than one subscript.

The explicit specification statements declare the type (INTEGER, REAL, or LOGICAL) and residence (CU or PE) of a variable or array by its name, rather than by its initial character. This differs from the other ways of specifying the type and residence of a variable or array (i.e., the predefined convention and the IMPLICIT statement). Also, the information necessary to allocate storage for arrays (dimension information) may be included within the statement. However, if this information does not appear in an explicit specification statement, it must appear in a DIMENSION or COMMON statement (see "DIMENSION Statement" or "COMMON Statement").

In the same manner in which the IMPLICIT statement overrides the predefined convention, the explicit specification statements override the IMPLICIT and predefined convention.

Example 1:

```
*CU INTEGER ITEM,VALUE(8)
```

Explanation: This statement declares the variables ITEM and VALUE to be CU INTEGERS. VALUE is an array, 8 words long. Both ITEM and VALUE must appear in an EQUIVALENCE statement, so that they may be assigned a CU address.

Example 2:

```
*PE REAL ALPHA(*),BETA,U(*,64),V(*,64),Z(*,64)
```

Explanation: This statement declares ALPHA,BETA,U,V, and Z to be PE REAL variables. ALPHA,U,V, and Z are also dimensioned. Note that BETA is PE REAL by predefined convention. One reason BETA might be present is because:

> Any variable used in a program must appear in at least one specification statement, other than a DATA statement, so that it may be put in the symbol table.

An Explicit type statement is an excellent method of including PE scalars that do not appear elsewhere.

ADDITIONAL SPECIFICATION STATEMENTS

DIMENSION Statement

General Form

DIMENSION $\underline{a}_1(\underline{k}_1),\underline{a}_2(\underline{k}_2),\underline{a}_3(\underline{k}_3),\ldots,\underline{a}_n(\underline{k}_n)$

where $\underline{a}_1, \underline{a}_2, \underline{a}_3,\ldots, \underline{a}_n$ are array names

$\underline{k}_1, \underline{k}_2, \underline{k}_3,\ldots,\underline{k}_n$ are each composed of one of the following array allocation specifications:

i_1
*
$*,i_2$ } PE variables only
$*,i_2,i_3$

in which i_1,i_2, and i_3 are unsigned integer constants. If the first index is an asterisk, the variable must reside in the PE's because an asterisk denotes vector alignment. In vector aligned arrays, the range of the first index is taken to be 64. Only PE arrays may be vector aligned, or have more than one subscript.

The information necessary to allocate storage for arrays used in the source program may be provided by the DIMENSION statement. The following examples illustrate how this information may be declared.

Examples:

```
*DIMENSION A(10), ARRAY (*,5,5), LIST(*,100)
*DIMENSION B(*,50),TABLE(*,8,2)
```

Adjustable Dimensions

Adjustable dimensions are not allowed.

COMMON Statement

General Form	
COMMON	$/\underline{r}/\underline{a}\ (\underline{k}_1),\underline{b}(\underline{k}_2),\ldots/\underline{r}/\underline{c}(\underline{k}_3),\underline{d}(\underline{k}_4),\ldots$
where	$\underline{a},\underline{b},\ldots,\underline{c},\underline{d}\ldots$ are PE variable or array names
	$\underline{k}_1,\underline{k}_2,\ldots\underline{k}_3,\ldots,\underline{k}_n$ are optional and are each composed of one of the following array allocation specifications:
	i_1
	*
	$*,i_2$
	$*,i_2,i_3$
	in which i_1,i_2, and i_3 are unsigned integer constants. In vector aligned arrays, the range of the first index is taken to be 64.
	/r/... represent common block names consisting of one through six alphameric characters, the first of which is alphabetic. These names must always be embedded in slashes.

If common is to be used within a group of programs, a BLOCK DATA subprogram must be included. The BLOCK DATA subprogram must contain the longest version of every common block used (see "Arguments Passed to Subprograms" for the single exception to this rule).

Variables or arrays that appear in a calling program or a subprogram may be made to share the same storage locations with variables or arrays in other subprograms by use of the COMMON statement. For example, if one program contains the statement

```
*COMMON/COMMN/TABLE
```

and a second program contains the statement

```
*COMMON/COMMN/LIST
```

the variable names TABLE and LIST refer to the same storage locations.

If the main program contains the statements:

```
*PE REAL A,B,C
*COMMON/COMMN/A,B,C
```

and a subprogram contains the statements:

```
*PE REAL X,Y,Z
*COMMON/COMMN/X,Y,Z
```

A shares the same storage location as X; B shares the same storage location as Y; and C shares the same storage location as Z.

Consider the following examples:

<u>Example 1</u>:

Main Program	Subprogram
`*COMMON/ALPHA/R(*),A,B,C,D(10)`	`*COMMON/ALPHA/S(*),W,X,Y,Z(10)`

<u>Explanation</u>: The main program references the common block with the variables R,A,B,C, and D. The subprogram refers to the same storage as the variables S,W,X,Y, and Z. For proper alignment vector aligned arrays appear first in the common blocks. The importance of this will be explained in Example 2.

Example 2:

Main Program	Subprogram
*COMMON/MATRX/R(1024),A(*),B	*COMMON/MATRX/S(*,16),Z(*),R

Explanation: In the main program, R begins on a row boundary, because all common blocks begin on a row boundary. R may not be used as a vector variable, however, because it is not specified as one. The size of R is quite important. If R was not a multiple of 64 in length A would not begin on a row boundary. In the subprogram, S may be used as a vector variable even though it refers to the same storage as R (the R in the main program). Also Z and A refer to the same row. Note that the R in the subprogram refers to B in the main program, not the array R. The two R's are different variables referring to different storage locations. The correspondence of variables in different subprograms that appear in COMMON depends on the position of the variables in the COMMON statements, not on the variable names.

Example 3:

```
*COMMON /W/ A(*), B(*), C(15), D(48), E
*COMMON /X/ A(*), C(15), D(48), E, B(*)
*COMMON /Y/ A(*), C(15), B(*), D(48), E
*COMMON /Z/ C(15), DMY1(49), A(*), D(48), DMY2(16), B(*), E
```

Explanation: These four sample common statements show various arrangements of five variables in common. Block W is the recommended method of arranging variables. X is also correct; however a mistake in adding the dimension of C, D, and E, or a change in their dimensions could throw B off the row boundary it must be on. Y is in error because B cannot follow C and still begin on a row boundary. Z is also correct; however the insertion of dummy variables into the common block can be extremely wasteful of storage.

Arrays need not be dimensioned in the COMMON statement in which they appear. If the arrays are dimensioned in a DIMENSION or Explicit type statement, only the name need appear in the COMMON statement. Vector aligned arrays must fall on row boundaries and should appear first, regardless of where they are dimensioned.

Labeled Common

In each of the preceding two examples, the common storage area (common block) established is called a named common area. The variables that appeared in the COMMON statements were assigned locations relative to the beginning of this named common area. Variables

and arrays may be placed in separate common areas. Each of these separate areas (or blocks) is given a name consisting of one through six alphameric characters (the first of which is alphabetic); those blocks that have the same name occupy the same storage space. <u>The use of blank (unnamed) COMMON is prohibited</u>.

Variables that are to be placed in labeled (or named) common are preceded by a common block name enclosed in slashes. For example, the variables A,B, and C will be placed in the labeled common area HOLD by this statement

```
*COMMON/HOLD/A,B,C
```

Common entries appearing in COMMON statements are cumulative throughout the program. For example, consider

```
*COMMON /R/ D, E /S/ F
*COMMON /S/ I, J /R/ P
```

These two statements have the same effect as the single statement

```
*COMMON /R/ D, E, P /S/ F, I, J
```

The length of a common area may be increased by using an EQUIVALENCE statement (see "EQUIVALENCE Statements") only within a BLOCK DATA subprogram.

<u>Programming Considerations</u>

Variables in a common block may be in any order. However, all vector aligned variables must fall on row boundaries.

Proper alignment is achieved either by arranging the variables so that vector aligned variables come first, or by constructing the block so that dummy variables force proper alignment.

SCRATCH Statement

General Form

SCRATCH $a_1(k_1), a_2(k_2), a_3(k_3), \ldots, a_n(k_n)$

where $a_1, a_2, a_3, \ldots, a_n$ are PE variable or array names

$k_1, k_2, k_3, \ldots, k_n$ are optional and are each composed of one of the following array allocation specifications:

i_1

*

*, i_2

*, i_2, i_3

in which i_1, i_2, and i_3 are unsigned integer constants. In vector aligned arrays, the range of the first subscript is taken to be 64.

The variables listed in a SCRATCH statement will be assigned to a block of memory. Moreover, the order in the list strictly determines the position in the block. Variables listed in a SCRATCH statement may be in any order. However, all vector aligned variables must fall on row boundaries. Thus variables in SCRATCH are allocated in a manner similar to variables in COMMON. Multiple use of SCRATCH may be made via EQUIVALENCE, as with COMMON.

Information (data) declared SCRATCH may be communicated down a CALL chain only through the argument list of the called subprograms, and can not be communicated up the CALL chain. (Note: SCRATCH variables, therefore, may not be equivalenced to COMMON variables or incoming subprogram dummy arguments.)

Example:

```
*  SUBROUTINE   A(D)
*  EXTERNAL     SUBRTN
*  DIMENSION    D(*),E(*)
*  SCRATCH      B(*),C(*)
*  EQUIVALENCE  (B (1),E (1))
       .
       .
       .
```

```
          B(*)=D(*)**2
               .
               .
               .
        * CALL  SUBRTN (B,C)
          D(*)=B(*) / C(*)
        * RETURN
```

Explanation: Subroutine A allocates 2 user rows requested of PE memory for temporary storage. The allocation is active from entry into A until RETURN from A. Variables B and E occupy the same row and since B is in SCRATCH E must also be considered to be in SCRATCH. Since the memory addresses of B,C, and E are known only at execution (and may differ for each entry to A!) the information in B,C, and E must be passed through the argument lists of called routine. Otherwise, the information is completely local to the current execution of subroutine A. Upon RETURN from A, the information in B,C,E is lost.

EQUIVALENCE Statement

General Form
EQUIVALENCE $(\underline{p}_1, p_2, p_3, \ldots)$, $(a, \underline{c}_1, \underline{c}_2, \underline{c}_3)$ where $\underline{p}_1, \underline{p}_2, \underline{p}_3, \ldots$ are PE variables (may be subscripted) a is a CU address (1-48 or 1-56 if no subprograms are used) $\underline{c}_1, \underline{c}_2, \underline{c}_3, \ldots$ are CU variables (may be subscripted)

The EQUIVALENCE statement must be used to assign a CU memory location to each CU variable used. There are 48 CU memory locations available for programer use (56 if the program unit contains no subprograms). It is the programer's responsibility to assign addresses in such a way as to avoid conflict. It is important to remember that all CU variables are effectively in common all the time. If a CU variable is changed, all other CU variables assigned to that same location are changed, no matter what routine they appear in. Each CU variable must appear once and only once in an EQUIVALENCE statement.

Example 1: CU variables

```
*DIMENSION I(8), J(4), L(6)
*EQUIVALENCE (1,I1,J(1)), (5,I(1),L(3))
```

Explanation: Variables I1 and J(1) are assigned to the first word of CU memory. Arrays are arranged in storage such that J(2) is the second word of CU memory and J(3) is the third, etc. L(1) is assigned to the third word also, and L(3) and I(1) are assigned the fifth word, etc. EQUIVALENCE statements must not be constructed in such a way as to extend arrays beyond either end of CU memory.

The EQUIVALENCE statement also provides the option for controlling the allocation of PE storage within a single program or subprogram. It is analogous to the option of using the COMMON statement to control the allocation of data storage among several programs. When the logic of the program permits, the number of storage locations used can be reduced by causing locations to be shared by two or more variables of the same or differing types. The EQUIVALENCE statement cannot be used to obtain mathematical equality of two variables.

Example 2: PE variables

```
*DIMENSION B(5), C(*,10), D(*,8,5)
*EQUIVALENCE (A,B(1),C(5,3)), (D(36,2,1),E)
```

Explanation: This EQUIVALENCE statement indicates that the variables A,B(1), and C(5,3) are assigned to the same storage locations; also that D(36,2,1) and E are assigned to the same storage locations. In this case, the subscripted variables refer to the position in an array. Note: Variables or arrays that are not mentioned in an EQUIVALENCE statement are assigned to unique storage locations. The EQUIVALENCE statement must not contradict itself or any previously established equivalences. For example, the further equivalence specification of B(2) with any element of the array C, other than C(6,3), is invalid.

Example 3:

```
*DIMENSION B(5), C(*,10), D(*,8,5)
*EQUIVALENCE (A,B(1),C(133)), (D(100),E)
```

Explanation: This EQUIVALENCE statement indicates that the variable A, the first variable in the array B, namely B(1), and the 133rd variable in the array C, namely C(5,3), are to be assigned the same storage locations. Also, it also specifies that D(100), i.e., D(36,2,1), and E are to share the same storage locations. Note: The effects of the EQUIVALENCE statements in examples 1 and 2 are the same. Also, EQUIVALENCE statements must not be constructed in such a way as to prevent vector alignment of vector variables (including common blocks, see "BLOCK DATA Subprograms"). For example:

```
*DIMENSION A(*), B(*,2)
*EQUIVALENCE (A(3), B(1,1))
```

A and B cannot both begin on a row boundary if A(3) and B(1,1) are to occupy the same location in PE memory. Therefore, one or both of these two statements are in error.

Variables that are brought into COMMON through EQUIVALENCE statements may not increase the size of the block, except in a BLOCK DATA subprogram, as indicated by these statements:

```
*BLOCK DATA
*DIMENSION D(*,3)
*COMMON /X/ A(*),B(*),C(*)
*EQUIVALENCE (B(1),D(1,1))
```

This would cause a common area (i.e., X) to be established containing the vector aligned row variables A, B, and C. The EQUIVALENCE statement would then cause the variable D(1,1) to share the storage location with B(1), D(1,2) would share with C(1), and D(1,3) would extend the size of the common area, in this manner

```
A(1)                (lowest location of the common area i.e., X)
  .
  .
  .
B(1), D(1,1)
  .
  .
  .
C(1), D(1,2)
  .
  .
  .
      D(64,3)       (highest location of the common area)
```

Since arrays must be stored in consecutive forward locations, a variable may not be made equivalent to another variable of an array in such a way as to cause the array to extend before the beginning of the common area. For example, this EQUIVALENCE statement is invalid

```
*BLOCK DATA
*DIMENSION  D(*,3)
*COMMON /X/ A(*),B(*),C(*)
*EQUIVALENCE (B(1),D(1,3))
```

because it would force D(1,1) to precede A(1), as follows:

```
        D(1,1)
   :
A(1), D(1,2)     (lowest location of the common area X)
   :
B(1), D(1,3)
   :
C(64)            (highest location of the common area)
```

Programming Considerations

Two variables in one common block may not be made equivalent. Two or more variables from separate common blocks may be equivalenced only within a BLOCK DATA subprogram. The equivalencing of variables from two or more common blocks causes the common blocks to overlap. This equivalencing must be done in such a way as to preserve the vector alignment of all common blocks, and of all vector aligned variables contained within them.

DATA Initialization Statement

General Form

DATA $\underline{v}_1,\ldots,\underline{v}_n/\underline{i}_1*\underline{d}_1,\ldots,\underline{i}_n*\underline{d}_n/,\underline{v}_{n+1},\ldots,\underline{v}\ /\underline{i}_{n+1}*\underline{d}_{n+1},\ldots,\underline{i}\ *\underline{d}\ /,\ldots$

where $\underline{v}_1,\ldots,\underline{v}$ are PE variables, subscripted PE variables (in which case, the subscripts must be integer constants), or PE array names

$\underline{d}_1,\ldots,\underline{d}$ are values representing integer or real data constants

$\underline{i}_1,\ldots,\underline{i}$ represent unsigned integer constants indicating the number of consecutive variables that are to be assigned the value of $\underline{d}_1,\ldots,\underline{d}$

A data initialization statement is used to define initial values of PE variables and arrays. There must be a one-for-one correspondence between these variables (i.e., $\underline{v}_1,\ldots,\underline{v}$) and the data constants (i.e., $\underline{d}_1,\ldots,\underline{d}$).

<u>Example 1</u>:

```
*DIMENSION D(*,10)
*DATA A, B, C/5.0,6.1,7.3/,D/320*1.0,320*2.0/
```

<u>Explanation</u>: The DATA statement indicates that the variables A, B, and C are to be initialized to the values 5.0, 6.1, and 7.3, respectively. Also, the statement specifies that the first 5 rows (320 words) in the array D are to be initialized to the value 1.0, and the second 5 rows to the value 2.0.

<u>Example 2</u>:

```
*DIMENSION A(5), B(*,3)
*DATA A/5*1.0/,B/192*2.0/
```

<u>Explanation</u>: The DATA statement specifies that all the variables in the arrays A and B are to be initialized to the values 1.0 and 2.0, respectively.

A variable or array residing in a COMMON block may be given initial values, by use of the DATA statement, only in the BLOCK DATA subprogram in which the block is declared.

TRANSFER Statement

General Form
TRANSFER(n) a = b where n is the number of words to be transferred (a multiple of 8) a is a CU location and b is a PE location, or a is a PE location and b is a CU location.

The TRANSFER statement causes the block of 8 words containing a to be assigned the values of the block of 8 words containing b. In the CU, blocks of 8 words begin at locations 1, 9, 17, ..., and in the PE's blocks of 8 words begin in PE(1), PE(9), PE(17), ... Use of the TRANSFER statement requires explicit layout of PE memory by the programmer. This can be achieved by specification with *, or through EQUIVALENCE. For example

```
*DIMENSION      SAVE(*), LOGICL(16)
*EQUIVALENCE    (SAVE(57),BLK8),  (1,MEMORY), (9,LOGICL(1))
      .
      .
*TRANSFER(8)    MEMORY = SAVE(9)
*TRANSFER(8)    BLK8   = LOGICL(9)
*TRANSFER(8)    SAVE(6)= LOGICL(12)
```

The first TRANSFER statement causes CU locations 1 through 8 to be assigned the values SAVE(9) through SAVE(16), respectively. The second TRANSFER statement causes SAVE(57) through SAVE(64) to be assigned the values of LOGICL(9) through LOGICL(16), respectively. In both of these cases the locations indicated are the first word in a block of 8 words.

The third TRANSFER statement causes SAVE(1) through SAVE(8) to be assigned the values of LOGICL(9) through LOGICL(16), respectively. In this statement the locations specified are not the first words in their respective blocks of 8 words.

Programming Considerations

The TRANSFER statement transfers blocks of 8 64-bit words regardless of type; however, there are only two legal uses for a TRANSFER statement:

1) To save blocks of CU memory by TRANSFER to PE memory, and to return such blocks to CU memory. Use of these blocks while resident in PE memory is illegal and may lead to erroneous results.

2) To TRANSFER blocks of PE floating-point data to the CU when the data are to be used as scalars in PE arithmetic expressions. The use of TRANSFER with integer data for this use is illegal and may lead to erroneous results.

CHECK Statement

General Form
CHECK (<u>x</u>,<u>a</u>) $\underline{v}_1,\underline{v}_2,\ldots,\underline{v}_n$
where <u>x</u> is the statement number of an executable statement
<u>a</u> is a CU LOGICAL variable. Either <u>x</u> or <u>a</u> or both <u>x</u> and <u>a</u> must be present. If only one is present the comma must be omitted.
$\underline{v}_1,\underline{v}_2,\ldots\underline{v}_n$ are variable names (may be subscripted) or COMMON block names

The CHECK statement may only be used when all CFD arithmetic is to be done in 32-bit mode. The CHECK statement causes the two halves of the listed variables to be compared in all enabled PE's. If a statement number appears in the CHECK statement and any pair of halves have been found to be not equal control will be transferred to the indicated statement. If a logical variable appears after the CHECK statement the bit(s) corresponding to the PE('s) where the compare fails will be set to 1 (.TRUE.), the remaining bits will be set to 0 (.FALSE.).

The CHECK statement may not terminate a DO loop, or appear in an IF statement.

Example:

```
* SUBROUTINE   FUN   (DUM,A)
* CU LOGICAL  ERROR
* CU REAL RSCLR
* PE INTEGER  I(*),ISCLR
* DIMENSION  A(*),B(*),E(6)
* COMMON  /BB/  C(*,20)
* COMMON  /DUM/  D(*)
* EQUIVALENCE  (1,RSCLR),(2,INDEX)
       .
       .
       .
* CHECK (99)  BB,A(*),D(6)
       .
       .
       .
* CHECK (ERROR) E(2),B(INDEX),C(*-3,INDEX),DUM
       .
       .
       .
* CHECK (99,ERROR) RSCLR, C(*,I(*+2)),ISCLR
```

Explanation: In the first CHECK statement the enabled PE's will compare the halves of the following: the entire COMMON block BB (or the amount of BB known to this subprogram), the row A(*), and the word D(6). The program will branch to statement 99 if one of the compares fails (i.e., if the two halves of any of the compared words are unequal.)

The second CHECK statement compares all the listed variables in enabled PE's and places the failure information (.TRUE. for PE's which failed) in the CU LOGICAL variable ERROR. The third CHECK statement compares the indicated data and places the failure pattern in ERROR. It will also branch to statement 99 if a failure was detected.

It is sometimes desirable to write a program that, at various points, requires the same computation to be performed with different data for each calculation. Writing that program would be simplified if the statements required to perform the computation could be written only once and then could be referred to freely, with each subsequent reference having the same effect as though these instructions were written at the point in the program where the reference was made.

For example, to take the cube root of a number, a program must be written. If a general program were written to take the cube root of any number, it would be desirable to be able to combine that program (or subprogram) with other programs where cube root calculations are required.

The CFD language provides for the above situation through the use of subprograms. There are two classes of subprograms: FUNCTION subprograms and SUBROUTINE subprograms.

The first class of subprograms are called functions, which differs from SUBROUTINE subprograms, in that functions return at least one value to the calling program; SUBROUTINE subprograms need not return any (and are more efficient if they don't).

MODE AND SUBPROGRAMS

The MODE is, in effect, in common when calling a SUBROUTINE subprogram; i.e., if the MODE is altered in the SUBROUTINE subprogram, upon return to the calling program this altered MODE will be in use. On the other hand, if the MODE is altered in a FUNCTION subprogram it must be remembered that the altered MODE is not communicated to the calling program; i.e., the old MODE is restored prior to returning to the calling program. Deliberately changing the mode within a subprogram may be a useful and even valuable technique. However, the differences between the way MODE is treated in a FUNCTION subprogram and a SUBROUTINE subprogram must be kept in mind.

MBIT1 and MBIT2 are, in effect, in common.

NAMING SUBPROGRAMS

A subprogram name consists of from one through six alphameric characters, the first of which must be alphabetic. Certain subprogram names must be typed (as variables are) according to the following rules.

1. Type declaration of FUNCTION subprograms - Such declaration must be made in one of three ways: by the predefined convention, by the IMPLICIT statement, or by an explicit specification (see "Type Specification of the FUNCTION Subprogram")

2. Type declaration of a SUBROUTINE subprogram - The type of a SUBROUTINE subprogram cannot be defined, because the results that are returned to the calling program are dependent only on the type of the variable names appearing in the argument list of the calling program and/or the arguments in COMMON.

ARGUMENTS PASSED TO SUBPROGRAMS

The means by which arguments are passed to subprograms differ somewhat from FORTRAN. First, CFD-code allows certain arguments not allowed in FORTRAN (i.e., common block names). The common block name in the calling program corresponds to a dummy name in a subprogram. For example:

Block Data	Calling Program	Subprogram
:	:	
*COMMON/X/A(*),B(*)	*COMMON/X/A(*),B(*)	*SUBROUTINE SUB(Y)
:	:	:
*END	*CALL SUB(X)	*COMMON/Y/C(*),D(*)
	:	:
	*END	*END

In the above example, common block X is passed to SUB. There, it is referenced as common block Y. Note that common block Y is a dummy argument and it need not appear in a BLOCK DATA subprogram. This is the single exception to the rule. All common blocks whose names are used as dummy arguments need not appear in the BLOCK DATA subprogram, all others must appear. Within subroutine SUB, C and D may be assigned values. These values will be in A and B respectively when control returns to the calling program.

All arguments are passed to subprograms as PE memory row addresses. Addresses of scalar variables and constants are not passed. Rather, these variables and constants are evaluated as arithmetic expressions. The results of the evaluations (controlled by MODE) are stored in a scratch row of PE memory. The address of this row is then passed to the subprogram. Thus subprograms are limited, as far as returning results through the argument list goes, to vector aligned PE arrays and elements of common blocks,

the names of which have been passed to them. This also means that the dummy arguments corresponding to constants, scalars, and expressions in the calling program must be vector aligned PE arrays. Scalars may only be returned through CU memory.

Calling Program	Called Program
common block name	common block name
subprogram name	subprogram name
everything else	PE vector aligned array name

System subprograms are not limited to these types of arguments. Before using a system supplied subprogram, the user should check the write-up to be sure his arguments conform to the type required (see "CFD Supplied Mathematical Function Subprograms").

FUNCTIONS

A function is a statement of the relationship between a number of variables. To use a function in CFD, it is necessary to

1. Define the function (i.e., specify what calculations are to be performed),

2. Refer to the function by name, where required in the program.

Function Definition

The three steps in the definition of a function are

1. The function must be assigned a unique name by which it may be called (see "Naming Subprograms").

2. The arguments of the function must be stated.

3. The procedure for evaluating the function must be stated.

Function Reference

The name of a function, appearing in any CFD arithmetic expression, refers to the function. Thus, the appearance of a function, with its arguments in parentheses, causes the computations to be performed as indicated by the function definition. The resulting quantity replaces the function reference in the expression and assumes the type and residence of the function. The type and residence of the name used for the reference must agree with the type and residence of the name used in the definition.

FUNCTION SUBPROGRAMS

The FUNCTION subprogram is a CFD subprogram consisting of any number of statements. It is an independently written program that is executed wherever its name appears in another program.

General Form
FUNCTION <u>name</u> ($\underline{a}_1, \underline{a}_2, \underline{a}_3, \ldots, \underline{a}_n$) ⋮ RETURN ⋮ END where <u>name</u> is subprogram name (see "Naming Subprograms") $a_1, a_2, a_3, \ldots, \underline{a}_n$ ($1 \leq n \leq 7$) are vector aligned array names, common block names, or dummy names of SUBROUTINE or other FUNCTION subprograms.

Since the FUNCTION is a separate subprogram, the PE variables not in common and statement numbers within it do not relate to any other program.

The FUNCTION subprogram may contain any CFD statement except a SUBROUTINE statement, another FUNCTION statement, or BLOCK DATA statement.

The arguments of the FUNCTION subprogram (i.e., $\underline{a}_1, \underline{a}_2, \underline{a}_3, \ldots, \underline{a}_n$) may be considered as dummy variable names. These are replaced at the time of execution by arguments supplied in the function reference in the calling program (see "Arguments Passed to Subprograms"). The actual arguments may be: A common block name, any real or integer constant, any real or integer subscripted or nonsubscripted variable, a PE array name, an arithmetic expression, or the name of another subprogram. The actual arguments must correspond in number, order, and type (not residence) to the dummy arguments. If the actual argument corresponds to a dummy argument that is defined or redefined in the subprogram, the argument must be a PE vector or a vector aligned PE array name. All arguments in a subprogram refer to the storage area assigned to the arguments by the calling program or to temporary storage areas (see "Arguments Passed to Subprograms").

The relationship between variable names used as arguments in the calling program and the dummy variables used as arguments in the FUNCTION subprogram is illustrated by

Example 1:

Calling Program

```
*DIMENSION B(*),C(*)
      :
 A(*) = SOMEF(B,C)
      :
```

FUNCTION Subprogram

```
*FUNCTION SOMEF(X,Y)
*DIMENSION X(*),Y(*)
 SOMEF(*) = X(*)/Y(*)
*RETURN
*END
```

Explanation: The values of the variable B of the calling program are used in the subprogram as the values of the dummy variable X; the values of C are used in place of the dummy variable Y. Thus if B(1) = 10.0 and C(1) = 5.0, then A(1) = B(1)/C(1), which is equal to 2.0.

The name of the function must be assigned a value at least once in the subprogram as the argument of a CALL statement or as the variable name on the left side of an arithmetic statement.

Example 2:

Calling Program

```
*PE INTEGER I
*DIMENSION X(*),Y(*,64),I(*)
      :
 ANS(*) = ROOT1*CALC(X,Y,I)
      :
```

FUNCTION Subprogram

```
  *FUNCTION CALC (A,B,J)
  *PE INTEGER I,J
  *CU INTEGER K
  *DIMENSION A(*),B(*,64),I(*),J(*)
  *EQUIVALENCE(40,K)
   I(*) = J(*)*2
  *GO TO (10,20),K
10 CALC(*) = A(*)*B(*,I(*))
  *RETURN
20 CALC(*) = 0.0
  *RETURN
  *END
```

Explanation: The values of X, Y, and I are used in the FUNCTION subprogram as the values of A, B, and J, respectively. The values of CALC are computed, and these values are returned to the calling program, where the values of ANS are computed. The variable I in the argument list of CALC in the calling program is not the same as the variable I appearing in the subprogram.

When a dummy argument is an array or common block name, an appropriate DIMENSION, COMMON, or explicit specification statement must appear in the FUNCTION subprogram. None of the dummy arguments may be given initial data values nor may they appear as variables in a common block.

All user supplied functions must be indexed by a single asterisk when they are assigned a value. This need not be present if the function name appears in a CALL statement.

Type Specification of the FUNCTION Subprogram

Every FUNCTION name must be assigned a type and residence, both in the FUNCTION subprogram and in all routines that call it. The type and residence must be the same in all routines. The specification may be by predefined convention, IMPLICIT statement, or explicit statement. All FUNCTIONS must be declared PE resident. The FUNCTION name may not be typed in the FUNCTION statement.

RETURN and END Statements in a FUNCTION Subprogram

All FUNCTION subprograms must contain both an END and at least one RETURN statement. The END statement specifies, for the compiler, the end of the subprogram; the RETURN statement signifies a logical conclusion of the computation and returns any computed value and control to the calling program. More than one RETURN statement may be used in a CFD subprogram.

Example:

```
  *FUNCTION DAV (D,E,F)
  *DIMENSION A(*),B(*)
  *DIMENSION D(*),E(*),F(*)
        .
        .
        .
  *IF (.ANY.((D(*).GT.E(*)))) GO TO 20
10 A(*) = D(*) + 2.0*E(*)
        .
        .
        .
 5 A(*) = F(*)+2.0*E(*)
        .
        .
        .
  *IF (.ANY.((A(*).LT.0.0))) GO TO 30
20 DAV(*) = A(*)+B(*)**2
  *RETURN
30 B(*) = A(*-1)-B(*+1)
        .
        .
        .
   DAV(*) = B(*)**2
  *RETURN
  *END
```

SUBROUTINE SUBPROGRAMS

The SUBROUTINE subprogram is similar to the FUNCTION subprogram in many respects: the rules for naming FUNCTION and SUBROUTINE subprograms are the same, they both require an END statement, and they both contain the same sort of dummy arguments. Like the FUNCTION subprogram, the SUBROUTINE subprogram is a set of commonly used computations, but it does not need to return any results to the calling program, as does the FUNCTION subprogram.

The CALL statement (discussed later in this section) is used in a main program or another subprogram to invoke a SUBROUTINE subprogram.

Since the SUBROUTINE is a separate subprogram, the PE variables not in common and statement numbers within it do not relate to any other program.

The SUBROUTINE statement must be the first statement in the subprogram. The SUBROUTINE subprogram may contain any CFD statement except a FUNCTION statement, another SUBROUTINE statement, or a BLOCK DATA statement. If IMPLICIT statements are used in a SUBROUTINE subprogram, they must immediately follow the SUBROUTINE statement.

General Form
SUBROUTINE <u>name</u> ($\underline{a}_1,\underline{a}_2,\underline{a}_3,\ldots,\underline{a}_n$) . . . RETURN END where <u>name</u> is the subprogram name (see "Naming Subprograms") $\underline{a}_1,\underline{a}_2,\underline{a}_3,\ldots,\underline{a}_n$ $(0 \le \underline{n} \le 7)$ are arguments. (There need not be any.) Each argument used must be a vector aligned array name, a common block name, or the dummy name of another SUBROUTINE or FUNCTION.

The SUBROUTINE subprogram may use one or more of its arguments to return values to the calling program. Any arguments so used must appear on the left side of an arithmetic statement within the subprogram, as arguments of a CALL statement, or as

arguments in a function reference. The SUBROUTINE name must not appear in any other statement in the SUBROUTINE subprogram.

The arguments ($\underline{a}_1, \underline{a}_2, \underline{a}_3, \ldots, \underline{a}_n$) may be considered as dummy variable names that are replaced at the time of execution by arguments supplied in the CALL statement. The actual arguments must correspond in number, order, and type (not residence) to the dummy arguments. Dummy arguments may not be given initial data values.

Example: The relationship between variable names used as arguments in the calling program and the dummy variables used as arguments in the SUBROUTINE subprogram is illustrated in this example. The object of the subprogram is to copy one array directly into another.

Calling Program

```
*DIMENSION X(*,64),Y(*,64)
    :
*CALL COPY (X,Y,K)
    :
```

SUBROUTINE Subprogram

```
   *SUBROUTINE COPY (A,B,N)
   *PE INTEGER N
   *DIMENSION A(*,64),B(*,64),N(*)
   *EQUIVALENCE(18,J)
    J = N(1)
   *DO 10 I = 1, J
10 B(*,I) = A(*,I)
   *RETURN
   *END
```

As you can see, an integer scalar has been passed through the argument list. The method is very crude however. If the CU location of J is known in the main program, the above effect could have been produced by assigning K to J and leaving K and N out of the argument lists altogether. The above example shows a method of passing scalars through the argument list. It is much easier to pass scalars through the CU because CU memory is shared among all routines. The above method should not be used unless absolutely necessary.

CALL Statement

The CALL statement is used only to call a SUBROUTINE subprogram.

General Form
CALL name ($\underline{a}_1, \underline{a}_2, \ldots, \underline{a}_n$) where name is the subroutine's subprogram name. $\underline{a}_1, \underline{a}_2, \underline{a}_3, \ldots, \underline{a}_n$ ($0 \le n \le 7$) are the actual arguments (if any) that are being supplied to the SUBROUTINE subprogram

Entry into a SUBROUTINE subprogram is made by a CALL statement that refers to that subroutine's subprogram name. Entry is made at the first executable statement following the SUBROUTINE statement.

Examples:

```
CALL OUT
CALL MATMPY (X,5,40,Y,7,2)
CALL QDRTIC (X,Y,Z,ROOT1,ROOT2)
CALL SUB1 (X(*)+Y(*)*5,SINE)
```

The CALL statement transfers control to the SUBROUTINE subprogram and replaces the dummy variables with the value of the actual arguments that appear in the CALL statement. The arguments in a CALL statement may be: a common block name, any constant, any real or integer subscripted or nonsubscripted variable, a PE array name, an arithmetic expression, or the name of a subprogram.

The arguments in a CALL statement must agree in number, order, and type (not residence) with the corresponding arguments in the SUBROUTINE subprogram. If an actual argument corresponds to a dummy argument that is defined or redefined in the referenced subprogram, the actual argument must be a PE vector aligned array name or the name of a common block, some part of which is changed. All arguments in a subprogram refer to the storage area assigned to the arguments by the calling program or to temporary storage areas (see "Arguments Passed to Subprograms").

RETURN Statement in a SUBROUTINE Subprogram

General Form
RETURN

The sequence of execution following the RETURN statement of a SUBROUTINE subprogram is to the next statement following the CALL in the calling program. In a main program, a RETURN statement performs the same function as a STOP statement.

Example:

```
       Calling Program                      Subprogram

              .                            *SUBROUTINE SUB (X,Y,Z)
              .                                  .
              .                                  .
    10*CALL SUB (A,B,C)                          .
    20 Y(*) = A(*)+B(*)                    *RETURN
              .                            *END
              .
              .
      *END
```

Explanation: After SUB is called, execution of the main program resumes at statement 20.

The EXTERNAL Statement

General Form
EXTERNAL a,b,c,...
where a,b,c,... are names of all subprograms in the user's library that are referenced by the program, whether or not they are used as arguments.

BLOCK DATA SUBPROGRAM

To allocate storage for and enter data into COMMON blocks, a separate subprogram must be written. This separate subprogram contains only the DATA, COMMON, DIMENSION, EQUIVALENCE, and type statements associated with the common blocks being defined.

General Form
BLOCK DATA . . . END

1. The BLOCK DATA subprogram may not contain any executable statements.

2. The first statement of this subprogram must be the BLOCK DATA statement.

3. All elements of all COMMON blocks must be listed in the COMMON statement, even though they do not all appear in the DATA statement.

4. Data may be entered into more than one COMMON block in a single BLOCK DATA subprogram.

CFD SUPPLIED MATHEMATICAL FUNCTION SUBPROGRAMS

The intrinsic functions, listed in the following table, may appear anywhere in a vector arithmetic expression, subject to the following rules.

1.) the function must be of the same type as the remainder of the expression,

2.) the function argument must be a vector arithmetic expression of the required type,

3.) the function mnemonic has not been specified by the user (e.g. declared EXTERNAL).

Function	Name	Type (Vector)	Arguments		Definition	Code
			No.	Type (Vector Expression)		
Exponential	EXP	Real	1	Real	e^{arg}	out-of-line
Natural Logarithm	ALOG	Real	1	Real	ln(arg)	out-of-line
Arctangent	ATAN	Real	1	Real	arctan(arg)	out-of-line
Trig. Sine	SIN	Real	1	Real	sin(arg)	out-of-line
Trig. Cosine	COS	Real	1	Real	cos(arg)	out-of-line
Square Root	SQRT	Real	1	Real	$\sqrt{arg}$	out-of-line
Absolute Value	ABS IABS	Real Integer	1 1	Real Integer	$\|arg\|$	in-line
Float	FLOAT	Real	1	Integer	Convert from Integer to Real	in-line
Fix	IFIX	Integer	1	Real	Convert from Real to Integer	in-line

Table 1.- Intrinsic Mathematical Functions

Function	Name	Type (Vector)	No	Type (Vector Expression)	Definition	Code
Maximum variable	ROWMAX	Real	1	Real	max (arg (1),...., arg(64))	out-of-line
Minimum variable	ROWMIN	Real	1	Real	min (arg (1),...., arg(64))	out-of-line
Sum of the values in a row	ROWSUM	Real	1	Real	$\sum_{j=1}^{64} arg(j)$	out-of-line

CFD SUPPLIED RUN-TIME CLOCK

The ILLIAC IV has an external progrmable clock. This clock has a 100 microsecond "tick". The following two CFD statements allow the programmer to set and read this clock.

General Form
CLOCKSET (<u>i</u>) CLOCKREAD (<u>j</u>) where <u>i</u> is an integer constant or variable.   <u>j</u> is a PE INTEGER scalar.

The clock always has a 48-bit value, even if 32-bit processing is requested. Reading the clock is synchronous, but setting the clock is not.

<u>Example</u>:

```
*   PE  INTEGER TIME
*   EXTERNAL  SUB
        .
        .
        .
*   CLOCKSET  (0)
*   CALL SUB
*   CLOCKREAD  (TIME)
```

<u>Explanation</u>: TIME*.0001 is the number of seconds needed to execute SUBROUTINE SUB.

DISK AREA Statement

General Form
DISK AREA $\underline{x}_1(\underline{n}_1)$, $\underline{x}_2(\underline{n}_2)$, . . . , $\underline{x}_n(\underline{n}_n)$
where $\underline{x}_1, \underline{x}_2$ ... , $\underline{x}_n$ are disk area names that are accessed by the program. These disk areas must also be known to the proper ILLIAC subsystem at the time of execution.
$\underline{n}_1$, $\underline{n}_2$, ... , $\underline{n}_n$ are the number of ILLIAC pages in their respective disk area.

Note: 1) All areas to be accessed by the main program or any of its subprograms must appear in a DISK AREA statement in the main program. 2) All subprograms must declare all disk areas to be referenced by the subprogram by placing them in a DISK AREA statement. 3) The disk area size must be the same in the main program and all subprograms. 4) There may be a maximum of 64 different disk areas accessed by a program.

Example:

main program

```
*EXTERNAL  RW, OUT, RELAX
      :
*DISK AREA INPUT(20), OUTPUT(60), PHI(120)
      :
*CALL   RW
```

subprogram

```
*SUBROUTINE  RW
      :
*DISK AREA   PHI(120)
```

Note that the main program allocates the disk areas INPUT, OUTPUT, and PHI of size 20, 60, and 120 ILLIAC pages respectively. Furthermore, PHI has been allocated 120 pages in both the main program and subprogram.

READ/WRITE Statements

General Form

```
┌        ┐
│READ    │
│READH   │
│READQ   │
│WRITE   │      (i, v, d (j), k)
│WRITEH  │
│WRITEQ  │
└        ┘
```

The READ statement transfers from the disk to the full array.
The READH statement transfers from the disk to half of the array.
The READQ statement transfers from the disk to a quarter of the array.
The WRITE statement transfers from the full array to the disk.
The WRITEH statement transfers from half of the array to the disk.
The WRITEQ statement transfers from a quarter of the array to the disk.

where

i is a general subscript (a CU integer ± an integer constant) whose value is (the request number) between 1 and 64.

v (the array location) is a vector aligned array name whose first subscript is a general subscript. Furthermore, the value of the first subscript must be
- 1 if the transfer is to/from the full array.
- 1 or 33 if the transfer is to/from half of the array.
- 1, 17, 33 or 49 if the transfer is to/from a quarter of the array.

d is a disk area name which has been specified in a DISK AREA statement.

j is a general subscript which indicates the starting page of the transfer within the disk area.

k is a general subscript which indicates the total number of pages to be transferred.

Example 1:

```
    *COMMON/ALPHA/  P(*,16,16),PTILDA(*,16,4)
    *EQUIVALENCE  (1,I),(2,J),(3,K)
    *DISK AREA  PHI(120),OUTPUT(240)
          .
          .
          .
 1  *READ(3,P(1,1,3),PHI(K+3),2)
          .
          .
          .
 2  *WRITEQ(27,PTILDA(17,1,1),OUTPUT(J),1)
```

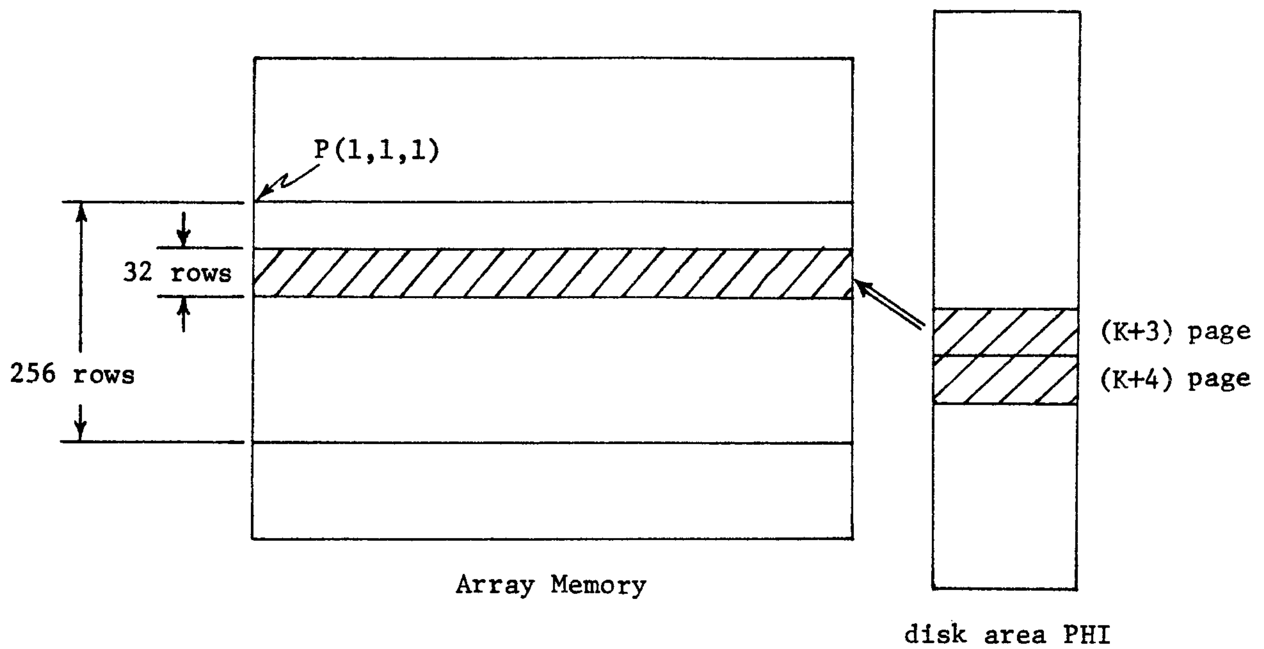


Figure 4.

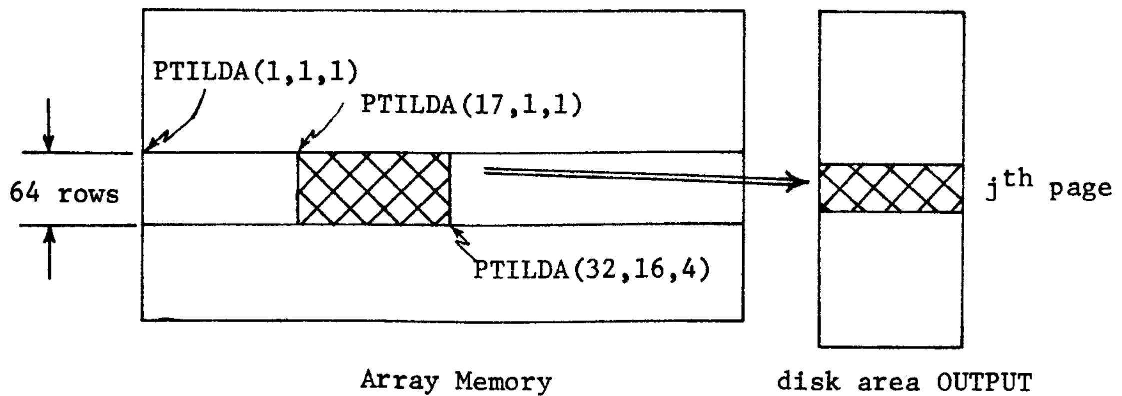


Figure 5.

Explanation: In statement 1, pages K+3 and (K+3)+1 of disk area PHI are transferred to core locations P(1,1,3) to P(64,16,4), (see Fig. 4). Note that 32 full rows of array memory were filled from the disk.

In statement 2, 64 quarter rows of PTILDA are transferred to the jth page of disk area OUTPUT (see Fig. 5). That is all of PTILDA in PE17 through PE32 are transferred to the jth page of OUTPUT.

Example 2:

```
*DIMENSION  X(*,16,16)
     :
*EQUIVALENCE (7,IREQNO)
     :
*DISK AREA  XSAVE(90)
     :
*WRITEH(IREQNO,X(33,1,1),XSAVE(1),2)
```

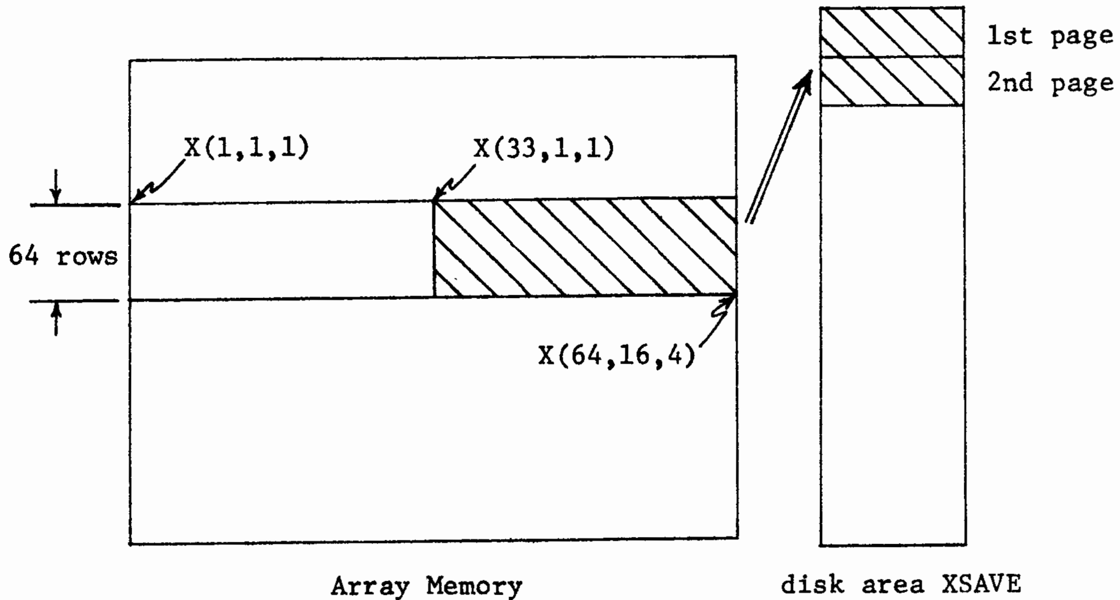


Figure 6.

Explanation: In this example, half rows are transferred from array memory to the disk. The transfer will be from X(33,1,1) through X(64,1,1) and the next 63 half rows stored below this row (see Fig. 6). This transfer is made to the first two pages of disk area XSAVE.

I/O IF Statement

Since CFD I/O is asynchronous in nature it provides a way to determine if a particular I/O request is completed.

General Form
IF (a i) b where a is either .COM. or .NOT COM. i is a general subscript whose value is the request number of a previously executed READ or WRITE statement. b is any executable CFD statement except a DO statement or another IF statement.

In general, the I/O IF statement functions as a scalar IF statement: if the statement in the parenthesis is true, the statement after the parenthesis is executed; if the statement is false, execution continues at the next statement. The truth values are as follows given that i is the request number of a previously executed READ or WRITE statement:

	request i completed	request i not completed
.COM. i	TRUE	FALSE
.NOTCOM. i	FALSE	TRUE

Example 1:

```
40 *WRITEQ(6,P(1,1,1),AREAP(1),2)
      :
20 *IF(.COM.6) GO TO 10
30 *CALL SUBA
   *GO TO 20
      :
10 *CONTINUE
```

Explanation: If at statement 20 the last I/O request having 6 as a request number has been completed the program will branch to statement number 10. On the other hand, if the I/O request 6 has not been completed execution continues a statement number 30.

Example 2:

```
     *WRITE(9,P(1,J,1),AREAP(K),2)
        .
        .
        .
  10 *CONTINUE
        .
        .
        .
  20 *IF(.NOTCOM.9)GO TO 10
  30 *READ(10,P(1,J,1),AREAP(K+2),2)
```

Explanation: When the program reaches statement 20 if the I/O request with request number 9 is completed execution continues at statement number 30. However, if the request has not been completed execution will continue at statement number 10.

WAIT Statement

General Form
WAIT n where n is a general subscript which represents an outstanding I/O request number

Example:

```
     *WRITE(23,T(1,1,1),AREA2(3),1)
     *WAIT 23
     *READ(24,T(1,1,1),AREA1(4),1)
```

Explanation: The READ statement will not be executed until the WRITE statement with request number 23 has been completed.

Note:

```
     *WAIT 23
   is equivalent to
  10 *CONTINUE
     *IF(.NOTCOM.23) GO TO 10
```

APPENDIX: CFD TO FORTRAN TRANSLATION

The CFDX translator will translate programs written in slightly restricted CFD into serial FORTRAN programs. The purpose of this document is to set forth the small subset of CFD that cannot be translated into FORTRAN and where possible to give a method by which to overcome these restrictions to the full generality of CFD.

The features of CFD that cannot be translated by CFDX are some of the non-FORTRAN features of CFD. These features will be listed in tabular form and where possible an alternative will be given.

Inserted ASK Statements

Clearly, FORTRAN compilers will not understand an ASK statement. However, the CFDX translator makes provision for this by allowing the CFDX user to place an F in column one of a card, in which case the CFDX translator will remove the F and insert the remainder of the card into the CFDX generated FORTRAN program. (Note: The card is _not_ shifted one place to the left as is the case with an imbedded ASK statement.) Therefore, the alternative is to insert equivalent FORTRAN statements for non-translatable ASK statements. CFDX will ignore any card with an A in column one and similarly CFD will ignore any card with an F in column one. Thus both types of non-CFD statements may appear in a program with no ill effect on either translator.

An example of how this alternative may be used can be seen in the following lines from a CFD program.

```
        MASK=MODE
        MODE=ON
C       LOGSUM OF RHO(*) IN ASK
A       LDA        RHO;
A       RTL        $A,-1;
A       ADRN       $R;
A       RTL        $A,-2
A       ADRN       $R;
A       RTL        $A,-4
A       ADRN       $R;
A       RTL        $A,-8;
A       ADRN       $R;
A       RTL        $A,-16;
A       ADRN       $R;
A       RTL        $A,-32;
A       ADRN       $R;
A       STA        SUM;
C       LOGSUM OF RHO(*) IN FORTRAN
```

```
F          SUMT=0.
F          DO 1234    K=1,64
F   1234   SUMT=SUMT+RHO(K)
F          DO 5678    K=1,64
F   5678   SUM(K)=SUMT
           MODE=MASK
                  .
                  .
                  .
```

COMMON BLOCK Names as SUBROUTINE Argument

The reason CFD allows COMMON BLOCK names to be used as SUBROUTINE arguments is that the number of arguments is restricted to seven. FORTRAN does not allow COMMON BLOCK names to be used as arguments; however, FORTRAN allows unlimited arguments in a subroutine call. Thus, the alternative to passing COMMON BLOCK names as arguments is to increase the argument list to contain all the members of the COMMON BLOCK (this is permissible because the seven argument restriction does not apply to CFDX).

The following is an example of a CFD program that cannot be translated by CFDX:

```
main program CFD                 subroutine CFD

COMMON/COM1/A(*),B(*),C(*)       SUBROUTINE SUB(COM2,P,Q,R,S,T)
    .                                .
    .                                .
    .                                .
CALL SUB(COM1,D,E,X,Y,Z)         COMMON/COM2/U(*),V(*),W(*)
    .                                .
    .                                .
    .                                .
```

The following is an equivalent program that can be translated by CFDX:

```
main program CFDX                subroutine CFDX

COMMON/COM1/A(*),B(*),C(*)       SUBROUTINE SUB(U,V,W,P,Q,R,S,T)
    .                                .
    .                                .
    .                                .
CALL SUB(A,B,C,D,E,X,Y,Z)        DIMENSION U(*),V(*),W(*)
    .                                .
    .                                .
    .                                .
```

Overlapping COMMON BLOCKS

FORTRAN does not permit COMMON blocks to be overlapped by EQUIVALENCE statements. Therefore CFDX will not allow such overlapping. There is no simple alternative to this problem. However, such overlapping in CFD is done to minimize storage requirements in ILLIAC IV; the machine upon which CFDX generated FORTRAN will be executed should have sufficient storage to allow the rearrangement of COMMON blocks so that there is no overlap. This is achieved by removal of the EQUIVALENCE statements that cause the overlap.

TRANSFER statement

Although CFD allows the transfer of CU logical words to PE memory for temporary storage the CFDX translator does not allow such a transfer due to word size conflicts. Thus the TRANSFER statement may be used to transfer real and/or integer words, but may not transfer logical words.

The CFDX translator also requires that the PE location given in the TRANSFER statement must reside in PE(1), PE(9), PE(17), ... , or PE(57). Moreover the PE location must be specified by an array which is of sufficient length to accommodate the number of words being transferred. The necessary PE alignment can be achieved by specification with *, or through EQUIVALENCE. For example

```
*DIMENSION      SAVE(*), LOGICL(16)
*EQUIVALENCE    (SAVE(57),BLK8), (1,MEMORY), (9,LOGICL(1))
     .
     .
     .
*TRANSFER(8)    MEMORY=SAVE(9)
*TRANSFER(8)    SAVE(21)=LOGICL(12)
*TRANSFER(8)    BLK8=LOGICL(12)
     .
     .
     .
```

The first TRANSFER statement causes CU locations 1 through 8 to be assigned the values SAVE(9) through SAVE(16), respectively. The second TRANSFER statement is not allowed by the CFDX translator although it is permissible in CFD and would cause SAVE(17) through SAVE(24) to be assigned the values of LOGICL(9) through LOGICL(16), respectively. The CFDX translator does not allow this transfer statement because SAVE(21) is not in a PE with a number equal to one modulo eight. An equivalent statement which would be acceptable to the CFDX translator would be

```
*TRANSFER(8)    SAVE(17)=LOGICL(12)
```

This TRANSFER statement is acceptable to CFDX because SAV(17) resides in PE(17) and 17=1 (mod 8). [Note in this example SAVE(21) is not assigned the value of LOGICL(12). The value of LOGICL(12) is assigned to SAVE(20).]

The third TRANSFER statement is also <u>not</u> allowed by CFDX although it is permissible in CFD and would cause SAVE(57) through SAVE(64) to be assigned the values of LOGICL(9) through LOGICL(16), respectively. In this case as in the first the PE location indicated is the first word of a block of eight words. However, unlike the first case the PE location is specified by a scalar and not an array of sufficient length to accommodate a transfer of eight words. Two equivalent groups of statements which are acceptable to the CFDX translator would be

```
    *DIMENSION      SAVE(*), LOGICL(16)
    *EQUIVALENCE    (SAVE(57),BLK8), (1,MEMORY), (9,LOGICL(1))
        .
        .
        .
    *TRANSFER(8)    SAVE(57)=LOGICL(12)
```

and

```
    *DIMENSION      SAVE(*), LOGICL(16), BLK8(8)
    *EQUIVALENCE    (SAVE(57), BLK8(1)), (1,MEMORY), (9,LOGICL(1))
        .
        .
        .
    *TRANSFER(8)    BLK8(1)=LOGICL(12)
```

Note in both cases the PE location is specified by an array with at least seven more words beyond the location specified. And that the location specified resides in a PE whose number is equal to one modulo eight.

<u>FUNCTION Subprograms</u>

The CFDX translator will not accept user written FUNCTION subprograms because FORTRAN FUNCTION subprograms return only one value and CFD FUNCTION subprograms return 64 values. The alternative is to replace FUNCTION subprograms and their uses with SUBROUTINE subprograms and the appropriate CALL and assignment statements. This restriction does not apply to intrinsic FUNCTION subprograms.

An example of a CFD FUNCTION subprogram and its use is illustrated by

Overlapping COMMON BLOCKS

FORTRAN does not permit COMMON blocks to be overlapped by EQUIVALENCE statements. Therefore CFDX will not allow such overlapping. There is no simple alternative to this problem. However, such overlapping in CFD is done to minimize storage requirements in ILLIAC IV; the machine upon which CFDX generated FORTRAN will be executed should have sufficient storage to allow the rearrangement of COMMON blocks so that there is no overlap. This is achieved by removal of the EQUIVALENCE statements that cause the overlap.

TRANSFER statement

Although CFD allows the transfer of CU logical words to PE memory for temporary storage the CFDX translator does not allow such a transfer due to word size conflicts. Thus the TRANSFER statement may be used to transfer real and/or integer words, but may not transfer logical words.

The CFDX translator also requires that the PE location given in the TRANSFER statement must reside in PE(1), PE(9), PE(17), ... , or PE(57). Moreover the PE location must be specified by an array which is of sufficient length to accommodate the number of words being transferred. The necessary PE alignment can be achieved by specification with *, or through EQUIVALENCE. For example

```
*DIMENSION     SAVE(*), LOGICL(16)
*EQUIVALENCE   (SAVE(57),BLK8), (1,MEMORY), (9,LOGICL(1))
      .
      .
      .
*TRANSFER(8)   MEMORY=SAVE(9)
*TRANSFER(8)   SAVE(21)=LOGICL(12)
*TRANSFER(8)   BLK8=LOGICL(12)
      .
      .
      .
```

The first TRANSFER statement causes CU locations 1 through 8 to be assigned the values SAVE(9) through SAVE(16), respectively. The second TRANSFER statement is not allowed by the CFDX translator although it is permissible in CFD and would cause SAVE(17) through SAVE(24) to be assigned the values of LOGICL(9) through LOGICL(16), respectively. The CFDX translator does not allow this transfer statement because SAVE(21) is not in a PE with a number equal to one modulo eight. An equivalent statement which would be acceptable to the CFDX translator would be

```
*TRANSFER(8)   SAVE(17)=LOGICL(12)
```

This TRANSFER statement is acceptable to CFDX because SAV(17) resides in PE(17) and 17=1 (mod 8). [Note in this example SAVE(21) is not assigned the value of LOGICL(12). The value of LOGICL(12) is assigned to SAVE(20).]

The third TRANSFER statement is also not allowed by CFDX although it is permissible in CFD and would cause SAVE(57) through SAVE(64) to be assigned the values of LOGICL(9) through LOGICL(16), respectively. In this case as in the first the PE location indicated is the first word of a block of eight words. However, unlike the first case the PE location is specified by a scalar and not an array of sufficient length to accommodate a transfer of eight words. Two equivalent groups of statements which are acceptable to the CFDX translator would be

```
*DIMENSION      SAVE(*), LOGICL(16)
*EQUIVALENCE    (SAVE(57),BLK8), (1,MEMORY), (9,LOGICL(1))
     .
     .
     .
*TRANSFER(8)    SAVE(57)=LOGICL(12)
```

and

```
*DIMENSION      SAVE(*), LOGICL(16), BLK8(8)
*EQUIVALENCE    (SAVE(57), BLK8(1)), (1,MEMORY), (9,LOGICL(1))
     .
     .
     .
*TRANSFER(8)    BLK8(1)=LOGICL(12)
```

Note in both cases the PE location is specified by an array with at least seven more words beyond the location specified. And that the location specified resides in a PE whose number is equal to one modulo eight.

FUNCTION Subprograms

The CFDX translator will not accept user written FUNCTION subprograms because FORTRAN FUNCTION subprograms return only one value and CFD FUNCTION subprograms return 64 values. The alternative is to replace FUNCTION subprograms and their uses with SUBROUTINE subprograms and the appropriate CALL and assignment statements. This restriction does not apply to intrinsic FUNCTION subprograms.

An example of a CFD FUNCTION subprogram and its use is illustrated by

calling program

```
*DIMENSION B(*),C(*)
     .
     .
     .
 A(*) = SOMEF(B,C)
     .
     .
     .
```

FUNCTION subprogram

```
*FUNCTION SOMEF(X,Y)
*DIMENSION X(*),Y(*)
 SOMEF(*)=X(*)/Y(*)
*RETURN
*END
```

The following is an equivalent usage of a SUBROUTINE subprogram which is acceptable to the CFDX translator:

calling program

```
*DIMENSION B(*),C(*),SOMEF(*)
     .
     .
     .
*CALL SOMEFS(B,C,SOMEF)
 A(*)=SOMEF(*)
     .
     .
     .
```

SUBROUTINE subprogram

```
*SUBROUTINE SOMEFS(X,Y,Z)
*DIMENSION X(*),Y(*),Z(*)
 Z(*)=X(*)/Y(*)
*RETURN
*END
```

Equivalence to dummy arguments

CFDX will not allow the user to equivalence to dummy arguments of a subprogram. Thus,

```
*SUBROUTINE   SUB(X,Y,Z)
*DIMENSION X(*),Y(*),Z(*),A(*,16)
      .
      .
      .
*EQUIVALENCE(X(1),A(1))
```

is not allowed by the CFDX translator.

Input-Output

The CFD statement

DISK AREA $\underline{x}_1(\underline{n}_1), \underline{x}_2(\underline{n}_2), \ldots \underline{x}_n(\underline{n}_n)$

will be translated into the following FORTRAN statement:

COMMON $/\underline{x}_1/\underline{x}_1(1024,\underline{n}_1), /\underline{x}_2/\underline{x}_2(1024,\underline{n}_2), \ldots, /\underline{x}_n/\underline{x}_m(1\text{-}24,\underline{n}_n)$

The CFD statements

$$\begin{bmatrix} \text{READ} \\ \text{READH} \\ \text{READQ} \end{bmatrix} \quad (\underline{r},\underline{v},\underline{d}(\underline{n}),\underline{k})$$

will be translated into the following FORTRAN CALL statement:

CALL DISKIN($\underline{i},\underline{v},\underline{d},\underline{n},\underline{k}$)

where

$\underline{i} = 1$ if it was a READ statement.

$\underline{i} = 2$ if it was a READH statement.

$\underline{i} = 3$ if it was a READQ statement.

Similarly,

$$\begin{bmatrix} \text{WRITE} \\ \text{WRITEH} \\ \text{WRITEQ} \end{bmatrix} \quad (\underline{r},\underline{v},\underline{d}(\underline{n}),\underline{k})$$

would be translated as follows:

CALL DISKOT($\underline{i},\underline{v},\underline{d},\underline{n},\underline{k}$)

where

$\underline{i} = 1$ if it was a WRITE statement.

$\underline{i} = 2$ if it was a WRITEH statement.

$\underline{i} = 3$ if it was a WRITEQ statement.

It is up to the user to supply to appropriate subroutines, DISKIN and DISKOT. The following are examples of appropriate subroutines for an IBM 360/67 (virtual memory).

```
      SUBROUTINE DISKIN(IBNDRY,IARRAY,IDISK,IBEGIN,NOPAGE)
      DIMENSION IARRAY(1),IDISK(1)
      IF((IBEGIN.LE.0).OR.(NOPAGE.LE.0)) GO TO 9
      ISTART=(IBEGIN-1)*1024+1
      IFIN=ISTART+(NOPAGE*1024)-1
      IF(IBNDRY.NE.1) GO TO 1
      INDXA=0
      DO 2 INDXD=ISTART,IFIN
      INDXA=INDXA+1
    2 IARRAY(INDXA)=IDISK(INDXD)
      RETURN
    1 IF(IBNDRY.NE.2) GO TO 3
      INDXD=ISTART
      IRDWA=0
    5 DO 4 INDXA=1,32
      INDXRA=INDXA+IROWA
      IARRAY(INDXRA)=IDISK(INDXD)
    4 INDXD=INDXD+1
      IF(INDXD.GE.IFIN) RETURN
      IROWA=IROWA+64
      GO TO 5
    3 IF(IBNDRY.NE.4) GO TO 6
      INDXD=ISTART
      IROWA=0
    8 DO 7 INDXA=1,16
      INDXRA=INDXA+IROWA
      IARRAY(INDXRA)=IDISK(INDXD)
    7 INDXD=INDXD+1
      IF(INDXD.GE.IFIN) RETURN
      IROWA=IROWA+64
      GO TO 8
    6 WRITE(6,100)
  100 FORMAT('!!!!SYSTEMS ERROR, INVALID I/O BOUNDARY')
      STOP
    9 WRITE(6,101)
  101 FORMAT('**** I/O FAULT, STARTING PAGE OR NUMBER OF PAGES
     *IN I/O REQUEST WAS NONPOSITIVE')
      STOP
      END


      SUBROUTINE DISKOT(IBNDRY,IARRAY,IDISK,IBEGIN,NOPAGE)
      DIMENSION IARRAY(1),IDISK(1)
      IF((IBEGIN.LE.0).OR.(NOPAGE.LE.0)) GO TO 9
      ISTART=(IBEGIN-1)*1024+1
      IFIN=ISTART+(NOPAGE*1024)-1
      IF(IBNDRY.NE.1) GO TO 1
      INDXA=0
      DO 2 INDXD=ISTART,IFIN
      INDXA=INDXA+1
    2 IDISK(INDXD)=IARRAY(INDXA)
      RETURN
```

```
    1 IF(IBNDRY.NE.2) GO TO 3
      INDXD=ISTART
      IROWA=0
    5 DO 4 INDXA=1,32
      INDXRA=INDXA+IROWA
      IDISK(INDXD)=IARRAY(INDXRA)
    4 INDXD=INDXD+1
      IF(INDXD.GE.IFIN) RETURN
      IROWA=IROWA+64
      GO TO 5
    3 IF(IBNDRY.NE.4) GO TO 6
      INDXD=ISTART
      IROWA=0
    8 DO 7 INDXA=1,16
      INDXRA=INDXA+IROWA
      IDISK(INDXD)=IARRAY(INDXRA)
    7 INDXD-INDXD+1
      IF(INDXD.GE.IFIN) RETURN
      IROWA=IROWA+64
      GO TO 8
    6 WRITE(6,100)
  100 FORMAT('!!!! SYSTEMS ERROR, INVALID I/O BOUNDARY')
      STOP
    9 WRITE(6,101)
  101 FORMAT(' **** I/O FAULT, STARTING PAGE OR NUMBER OF PAGES
     *IN I/O REQUEST WAS NONPOSITIVE')
      STOP
      END
```

If the FORTRAN generated by the CFDX translator is to run on a machine without virtual memory it may be necessary for the user to remove the COMMON statements which were a result of the DISK AREA statements. Furthermore, these I/O subroutines may have to be written in such a way as to make use of some external storage device, e.g., tapes, disk, etc.

In the case of the I/O IF statement the CFDX translator assumes that all I/O requests have been completed. Thus, .NOT COM. n is always assumed to be false and .COM. n is always assumed to be true.

In the same vein the WAIT statement has no real meaning in CFDX and is thus translated into a FORTRAN CONTINUE statement. It should be noted, however, that if the machine upon which the CFDX generated FORTRAN is to be executed has the facility to use asynchronous I/O the necessary FORTRAN statement should be inserted in the CFD to make use of this facility.

Finally, with regard to CFD I/O, it is the user's responsibility to see that all the FORTRAN COMMON's (or their logical equivalent) which are associated with the CFD DISK AREA's are

properly initiated. This may be achieved by supplying the proper FORTRAN and job control commands all of which is dependent upon the machine which will execute the FORTRAN. Similarly FORTRAN or job control type I/O must be used when the task has been completed.

RWA - A Required FUNCTION Subprogram

The FORTRAN generated by the CFDX translator makes frequent calls to a FUNCTION subprogram called RWA. RWA has one argument and is defined in the following equation

$$RWA(N) \equiv (N-1)(\text{mod } 64) + 1$$

where $0 \leq M(\text{mod } 64) \leq 63$, so $1 \leq RWA(N) \leq 64$

The following is RWA written in 360 assembly language for an IBM 360/67.

```
RWA@C   CSECT
        ENTRY   RWA
        USING   RWA, 15
RWA     L       1,0(1)             ADDRESS OF ARG. IN R1
        L       1,0(1)             ARG. IN R1
        BCTR    1,0                DECREMENT R1
        N       1,=X'0000003F'     'AND' WITH MASK X'0000003F'
        LA      1,1(,1)            ADD 1 TO R1
        LR      0,1                ANSWER IN R0
        BR      14                 RETURN TO CALLING PROGRAM
        END
```

This particular program is in the systems library of the NASA-AMES 360/67. An equivalent program should be made available on any machine upon which CFDX generated FORTRAN is to be run. (It is frequently used and should be coded efficiently.)

RØWMAX, RØWMIN, and RØWSUM

CFDX does not support the intrinsic functions ROWMAX, ROWMIN, and ROWSUM. Therefore, the alternative is to insert equivalent FORTRAN statements for these unsupported functions.

Run-time Clock

CFDX does not support CLOCKSET and CLOCKREAD because there is no universal FORTRAN equivalent. The alternative is to insert the equivalent FORTRAN for your system.

TECHNICAL NOTE: TRANSLATOR CONVENSIONS

CFD has fixed conventions which allow communication of information between separately compiled modules. The four classes of modules are SUBROUTINE, subprograms, FUNCTION subprograms, BLOCK DATA subprograms (COMMON blocks), and the run-time CFD package.

In general, the user has control over the first 48 ADB locations ($D0 through $D47) and the E,E1,G and H bits of the $D register (E and EI=MODE, G=MBIT1, H=MBIT2). Therefore all CFD modules modify these registers and bits only at program direction.

SUBROUTINE Subprogram Convensions

When a SUBROUTINE is called, the calling program does the following: addresses of arguments (which must number less than or equal to seven) are stored in $D49 through $D55 in the same order as they appear in the argument list. These addresses are 24-bit addresses, therefore, the high-order 40-bits of the ADB locations may contain garbage. If the argument is a subprogram name the address is a word address. Otherwise, the address is a row address (as required for ACAR indexing of FINST instructions). The return address is passed in $C3 (by an EXCHL (3) $ICR instruction at the branch).

In the called subprogram $D48 may be used to store the return address. If this called subprogram calls another subprogram, it must save its arguments and return address. The save area of 8 PE words is aligned on an 8 word boundary to allow a BIN reload to the ADB on return from the called routine.

The last 8 ADB locations are used by CFD and should be restored if they are changed. $D56 is a scratch register, $D57 is a pointer used by SCRATCH statement, $D58 is an I/O completion mask, and $D59 through $D63 contain masks used in CFD generated code. None of the other hardware registers, e.g. $C0-$C3, $A, $B, $R, $S, $X and the not aformentioned parts of $D, need be restored by a SUBROUTINE subprogram. In addition, $D48-$D55 are free for SUBROUTINE (assuming addresses the originals contained are no longer needed) use and need not be restored.

FUNCTION Subprogram Convensions

With the following three exceptions, the linkage of a FUNCTION is the same as to a SUBROUTINE. One, on return from a FUNCTION, the FUNCTION values are passed back to the calling program in $A. Two, in a SUBROUTINE the E, E1 bits (mode) may be changed. On the other hand, a FUNCTION subprogram insures the E, E1 bits of $D are returned unchanged to the calling program. These bits may be changed within the FUNCTION as required

but must be reset to the values they were at entry prior to returning to the calling program. Three, FUNCTION's must, at call, be referenced with at least one argument.

Dynamic Allocation Stack Pointer

The CFD SCRATCH statement as well as CFD's need for working storage, requires that CFD maintain a run-time stack pointer to a scratch area. This area is located at the bottom of PE memory. (SCRATCH is equivalenced to D and is used as the base address in CFD object listings.) The row address of the first available row of scratch resides in $D57. The calling program sets the pointer ($D57) to cover active rows when moving down the call chain (by calling a subprogram). The pointer must have the same value at RETURN from the called subprogram as when it was entered. Moreover, it is the responsibility of the called subprogram to check (at entry) if sufficient scratch storage is available. If insufficient storage is available the CFD subprogram will HALT.

COMMON Blocks

All CFD COMMON blocks must appear in a BLOCK DATA subprogram. This is because the BLOCK DATA subprogram declares the COMMON block name an (ENTRY) and allocates the storage. This also necessitates that the longest version of all COMMON blocks be in the BLOCK DATA. In non-BLOCK DATA programs COMMON block names are declared EXTERNAL and the members of the COMMON block are equivalenced to the COMMON block name plus the appropriate offset. (To make use of the symbol table of the BLOCK DATA program it is suggested that the BLOCK DATA be the second program INCLUDED in the LINKED. This will insure that the symbol table address of COMMON variables is the same as their actual address.)

Required CFD Run-time modules

CFD programs must have three relocatable files INCLUDED at the LINKED stage along with all the user supplied relocatable files. (AMES) OBJ.FIRST must be the first file in any CFD ISV file. That is, (AMES)OBJ.FIRST must be the first file INCLUDED in a CFD LINKED. (AMES)OBJ.LAST must be the last file INCLUDED in any CFD LINKED. If 64-bit processing is to be used (AMES) OBJ.INTR (which contains the intrinsic functions) must be INCLUDED with the other CFD reloactable files. If 32-bit processing is to be used (AMES)OBJ.INTR32 must be INCLUDED with the other CFD relocatable files. The latest version of these files will always be in the (AMES) directory on I4TENEX.

Where more than one reference is given, the major reference is first.

ABS 2.64, 1.23
absolute value 2.64, 1.23
addition 2.14, 1.3
adjustable dimensions 2.43
.ALL. 2.22, 1.12, 1.15, 2.23
ALOG 2.64, 1.23
.AND. 1.14, 2.19, 2.24
.ANY. 2.22, 1.12, 1.15, 2.23
arc tangent 2.64, 1.23
arguments, dummy 2.54, 1.20, 2.57, 2.59, 2.60
arithmetic assignment statement 2.26, 2.1, 1.17, 2.17
arithmetic IF statement 1.11
arithmetic expression 2.14, 2.13
arithmetic operator 2.14
array memory 1.26
arrays 2.8, 1.6
array storage 2.12
array subscript see "subscript"
ASK statement 2.2, 1.4
assignment statement 2.26, 2.17, 2.1, 1.12, 1.15
ASSIGN statement 2.30, 1.11
assigned GO TO statement 2.30
asynchronous I/O 1.28, 1.25, 2.69
ATAN 2.64, 1.23

base mode 1.13
bit setting operator 2.20, 1.13
bit shifting operator 2.21, 1.13
blank COMMON 2.46
BLOCK DATA subprogram 2.62, 2.44, 1.8, 2.54
branching 1.11

CALL statement 2.60, 1.11, 2.59
card format 2.1, 1.4
card input 2.1
CFD supplied functions 2.64, 1.23
CHECK statement 2.52
CLOCKREAD statement 2.64.2, A.10
CLOCKSET statement 2.64.2, A.10
coding CFD statements 2.1
.COM. 1.30, 2.69
comment card 2.2, 1.4
common block 1.6, 1.9, 2.43, 2.11, 1.20, 2.56, 2.59
computed GO TO 2.29
COMMON statement 2.43, 2.11

constant 2.3, 2.4
continuation line (continuation statement) 1.4, 2.1
CONTINUE statement 2.37, 1.11
control statement 2.28, 2.1
control unit (CU) 1.2
COS 2.64, 1.23
CU address 1.2, 1.8, 2.6, 2.47
CU arithmetic 1.16
CU INTEGER 2.7, 1.6, 1.16, 2.39, 2.11, 2.41
CU LOGICAL 2.7, 1.6, 2.39, 2.11, 2.41
CU memory 2.47, 1.8, 1.10, 1.20, 2.6, 1.2
CU REAL 2.7, 1.6, 2.39, 2.11, 2.41

data initialization 2.49, 1.10
DATA statement 2.49, 1.10, 2.62
DIMENSION statement 2.42, 2.11
disabled PE 1.3, 1.11, 2.22
disk area 1.29
DISK AREA statement 2.65, 1.29
division 2.14, 1.3, 2.16
DO statment 2.34, 1.11
dummy arguments 2.54, 1.20, 2.57, 2.59, 2.60

enabled PE 1.3, 1.11, 2.22
END statement 2.38, 2.58, 1.11, 2.59
EQUIVALENCE statement 2.46, 1.8, 1.9, 2.6
EXP 2.64, 1.23
explicit specification statement 2.41, 2.7, 2.11, 2.6, 1.6
exponent 1.17, 2.15, 2.16
exponentiation (**) 2.14, 1.17, 1.3, 2.15
expression 2.13
EXTERNAL statement 2.62

.FIRST. 2.20
first subscript 2.10, 1.17, 1.18, 1.7
fix 2.64, 1.23
FLOAT 2.64, 1.23
FUNCTION subprogram 2.56, 2.54, 2.5, 1.20, 1.6, 1.17, 2.53

general subscript 2.10, 1.7, 2.66
GO TO statement 1.11
    unconditional 2.28
    computed 2.29
    assigned 2.30

hierarchy of operations 1.17, 2.15, 2.24

IABS 2.64, 1.23
ILLIAC IV 1.2, 2.9
ILLIAC disk 1.28
ILLIAC page 1.26
IF statement 2.31, 1.13, 1.11

arithmetic IF statement 1.11
I/O IF statement 2.69, 1.30
logical IF statement 2.31, 1.13, 1.11
scalar logical IF 2.32, 2.31, 1.13
vector IF statement 1.13, 1.12
vector logical IF 2.23, 2.31, 1.13
IFIX 2.64, 1.23
IMPLICIT statement 2.39, 2.7, 2.6
indices 2.8, 1.7
integer division 2.16, 1.3, 2.14
intrinsic function 2.64, 1.23
Input/Output hardware 1.25
Input/Output IF statement 2.68, 1.30
Input/Output statement 2.66, 2.1, 1.29
integer constant 2.3

labeled COMMON 2.45
.LAST. 2.21
limitations of CFD 1.1
lock-step 1.3
logarithm 2.64, 1.23
logical assignment statement 2.26, 1.13, 1.15, 2.1
logical constant 2.4
logical expression 2.22, 2.17, 1.13, 2.13
logical IF statement 2.31, 1.13, 1.11
logical operator 2.19, 1.13
logical quantifier 2.21, 1.12, 1.15
logical variable 1.6, 1.13

magnitude of a number 2.3
MBIT1 1.6, 2.53, 2.27, 1.13, 1.20
MBIT2 1.6, 2.53, 2.27, 1.13, 1.20
mixed type expression 1.17, 2.14
mode 2.53, 1.3
MODE, logical variable 1.12, 1.17, 2.53, 1.20
multiplication 1.3, 2.14

naming convention 1.5
natural logarithm 2.64, 1.23
.NOT. 2.19, 1.13, 1.15, 2.24
.NOT ALL. 1.15, 2.22, 1.12, 2.23
.NOT ANY. 1.15, 2.22, 1.12, 2.23
.NOT COM. 1.30, 2.69

OFF, the logical constant 2.4, 2.3, 1.15, 1.13
ON, the logical constant 2.4, 2.3, 1.15, 1.13
.OR. 2.20, 1.14, 2.24
order of computation 2.15, 2.24, 1.17
overlap 1.2

parentheses in an arithmetic expression 2.16
in a logical expression 2.25
PE arithmetic 1.17, 2.1, 2.26
PE INTEGER 2.39, 1.9, 1.6, 2.11, 2.41, 2.7
PE memory 1.8, 1.10

PE REAL 2.39, 1.6, 2.7, 2.11, 2.41
predefined specification 2.7, 2.6, 1.6
processing element (PE) 1.3, 1.2
program control 2.28, 2.1, 1.11

range specifier 2.20
read statement 2.66, 1.30
READ statement 2.66, 1.30
READH statement 2.66, 1.30
READQ statement 2.66, 1.30
real constant 2.3
relational operator 2.18
request number 1.29, 1.30, 2.66
reserved names 1.5
RETURN statement 2.61, 2.58, 1.11
rotating operator 2.21, 1.13, 1.14, 1.15
ROWMAX 2.64.1, A.10
ROWMIN 2.64.1, A.10
ROWSUM 2.64.1, A.10
.RTL. 2.21, 1.14, 1.15, 2.23
.RTR. 2.21, 1.14, 1.15, 2.23

scalar 1.6, 1.8, 2.17
scalar logical expression 2.23
scalar logical IF 2.32, 2.31, 1.13
SCRATCH statement 2.46
shifting operator 2.21, 1.14, 1.15
.SHL. 2.21, 1.14, 1.15, 2.23
.SHR. 2.21, 1.14, 1.15, 2.23
SIN 2.64, 1.23
specification statement 2.39, 2.1, 2.6
SQRT 2.64, 1.23
square root 2.64, 1.23
statement number 2.1, 1.4
STOP statement 2.38, 1.11
subprogram 2.53, 1.20, 1.6, 2.5
subprogram argument 2.54, 1.20
subprogram statement 2.54, 2.1
SUBROUTINE subprogram 2.59, 2.54, 1.20, 1.6, 2.53, 2.5
subscript 2.10, 2.8, 2.9, 2.12, 1.18, 1.7
subtraction 1.3, 2.14
symbolic names 2.5

.TO. 2.21
TRANSFER statement 2.51, 1.16
trigonometric cosine 2.64, 1.23
trigonometric sine 2.64, 1.23
.TURN OFF. 2.20, 1.15, 1.14, 2.23
.TURN ON. 2.20, 1.15, 1.14, 2.23

variable 2.5, 1.6
variable name 2.6
vector aligned 1.9, 2.43, 2.9

vector aligned array 1.6, 1.9, 2.43, 1.20, 2.17, 2.41, 2.42, 2.55, 2.59
vector arithmetic 1.17, 2.10, 2.16, 1.3
vector index subscript 2.10, 1.12, 1.19, 1.7
vector IF 1.13, 1.12
vector logical expression 2.23
vector logical IF 2.23, 2.31, 1.13
vector relation 1.13
vector subscript 2.10, 2.12, 1.19

WAIT statement 2.70, 1.31
write statement 2.66, 1.30
WRITE statement 2.66, 1.30
WRITEH statement 2.66, 1.30
WRITEQ statement 2.66, 1.30

32-bit processing 2.52, 2.3