DIBOL-83 Language Reference Manual

Order No. AA-U066A-TK

Includes Update Notice: AD-U066A-T1

October 1983

Supersession/Update Information: This manual incorporates Update Notice AD-U066A-T1.

Operating System and Version:

CTS-300 V8.0
RSTS/E DIBOL V5.0
VAX/VMS DIBOL V2.0
Professional Host Tool Kit DIBOL V1.6
PRO/Tool Kit DIBOL V1.6
RSX-11M V4.3
RSX-11M-PLUS V2.1

First Printing, May 1983
First Update, October 1983

CONTENTS

CONTENTS (Cont.)

FIGURES

TABLES

PREFACE

The *DIBOL-83 Language Reference Manual* contains reference information on all aspects of the Standard DIBOL-83 Language. It does not include information on any particular operating systems or their specific effect on DIBOL-83.

AUDIENCE

This manual is written for:

The programmer who is new to DIBOL but is experienced in another high-level language.

The experienced DIBOL programmer.

MANUAL ORGANIZATION

The manual is organized as follows:

This Preface orients the reader to the format used throughout the manual, and to the terms and symbols used within the text.

Chapter 1 contains information related to the language elements such as the character set, statement types, program structure, syntax, labels, array subscripting, literals, substrings, and expressions.

Chapter 2 references all Data Division statements including the COMMON, RECORD, and SUBROUTINE statements, and describes field definitions.

Chapter 3 references all Procedure Division statements and explains the Value Assigment Statements.

Chapter 4 contains information related to Compiler Directives such as .TITLE, .INCLUDE, .LIST, .PAGE, and others.

Chapter 5 references all Universal External Subroutines.

The Appendix contains the DIBOL-83 Character Set.

The Glossary defines terms and phrases as used in this manual.

MANUAL FORMAT

This manual provides the reader with fast information retrieval.

The major subject discussed on each page in chapters two through five is displayed in **bold lettering** in the upper outer corner of each page. This bold lettering allows the reader to quickly find particular information.

The majority of the pages contain five main sections:

The **FUNCTION** section briefly describes or defines the subject matter.

The **FORMAT** section describes the correct structure or make-up of a statement, subroutine, etc., and explains each portion of the structure.

The **RULES** section provides guidelines, parameters, advice, and limitations for the particular subject matter. The rules are not necessarily presented in order of importance.

The **ERROR CONDITIONS** sections list compiler errors and run-time errors. The run-time errors will also indicate their assigned error number and whether they are Trappable (T) or Non-trappable (NT). All listed errors are particular to the subject matter, statement, or subroutine being discussed.

The **EXAMPLES** section illustrates the use of the particular subject matter.

DOCUMENT SYMBOLS

The symbols defined below are used throughout this manual.

Symbol	Definition
afield	is the name of an alpha field.
aliteral	is an alpha literal.
ch	is a decimal expression that evaluates to an input/output channel number.
dexp	is a decimal expression that can be any valid combination of operands and operators. In its simplest cases, dexp can be a dfield or a dliteral.
dfield	is the name of a decimal field.
dliteral	is a decimal literal.
field	is the name of either an alpha or a decimal field.
label	is a Procedure Division statement label.
literal	is either an alpha or a decimal literal.

lowercase
: (characters) mean elements of the language which are supplied by the programmer.

non-trappable error
: is an error that causes program termination and cannot be trapped.

record
: is the name of a record.

subroutine
: is the name of a subroutine.

trappable error
: is an error that can cause program termination but may be trapped using the ONERROR statement.

UPPERCASE
: (characters) mean elements of the language which must be used exactly as shown.

[ ] represent brackets and mean optional arguments.

| | represent vertical lines and mean a single choice must be made from a list or arguments

... represent a horizontal ellipsis and mean the preceding item can be repeated as indicated.

•
• represents a vertical ellipsis and means that not all of the statements in a figure or example are shown.
•

CHAPTER 1
DIBOL-83 LANGUAGE ELEMENTS

A DIBOL-83 program is a sequence of statements that describes a method for performing a task. These statements are translated by the DIBOL-83 compiler for subsequent execution by the DIBOL-83 run-time system under the control of the operating system.

1.1 DIBOL-83 CHARACTER SET

A DIBOL-83 program consists of symbolic characters that form the elements of the language. A subset of the American Standard Code for Information Exchange (ASCII) characters comprise this set of symbolic characters. Characters used as data are also selected from this character set.

Appendix A lists the ASCII characters and their associated numeric codes.

1.2 STATEMENT TYPES

A statement is the basic unit of expression in the DIBOL-83 language.

DIBOL-83 statements fall into six functional groups:

Compiler Directives and Declarations

Data Specification Statements

Data Manipulation Statements

Control Statements

Intertask Communications Statements

Input/Output Statements

A statement has one or more elements. The first element is usually an English language verb that characterizes or symbolizes an action to be performed (such as READ, WRITE, SLEEP, OPEN, and CALL).

The other elements of a statement may be arguments, expressions, or other statements. Arguments consist of symbolic data names, references to statement labels, and expressions of data values or relationships. Arguments specify the objects of the action being performed by the statement.

1.2.1 Compiler Directives and Declarations

- Compiler Directives and Declarations are instructions that provide information about the program to the compiler.
- Compiler Directives and Declarations are not executable at run-time.
- Compiler Directives may appear anywhere in the program. They are discussed in Chapter 4.
- Declarations are limited to either the Data Division (SUBROUTINE) or Procedure Division (BEGIN-END and PROC-END). They are discussed in the chapter devoted to those respective program divisions.
- The Compiler Directives are:

.IFDEF-.ENDC	causes statements that follow to be compiled if a specified variable is defined (.ENDC marks the end of the statements controlled by the .IFDEF and .IFNDEF statements).
.IFNDEF-.ENDC	causes statements that follow to be compiled if a specified variable is not defined (.ENDC marks the end of the statements controlled by the .IFDEF and .IFNDEF statements).
.INCLUDE	causes the compiler to open a specified file and continue the compilation using that file.
.LIST	enables the compiler to list source code.
.NOLIST	inhibits the listing of compiler source code.
.PAGE	causes a top-of-page command to occur.
.TITLE	causes a top-of-page command to occur and a new title to be placed in the page header.

- The Declarations are:

BEGIN-END	indicates the start (BEGIN) or finish (END) of a sequence of blocked statements.
PROC-END	separates Data Division Statements from Procedure Division Statements (PROC) and indicates the last statement in a program (END).
SUBROUTINE	identifies a program as an external subroutine.

1.2.2 Data Specification Statements

- Data Specification Statements identify and define the characteristics (i.e., whether it is alpha or numeric decimal, its size, and its symbolic name) of the data processed by a DIBOL-83 program.
- The Data Specification Statements are:

COMMON	describes a record, whose fields can be accessed from external subroutines.
RECORD	describes a record.
field definition	describes the name, data type, and size of a field in a record.

1.2.3 Data Manipulation Statements

- Data Manipulation Statements perform conversion and value assignment.
- The Data Manipulation Statements are:

CLEAR	sets a variable to zero or spaces.
INCR	increments a variable by one.
LOCASE	converts UPPERCASE letters to lowercase.
UPCASE	converts lowercase letters to UPPERCASE.
value assignment statement	assigns the value in the source to the destination.

1.2.4 Control Statements

- Control Statements modify the order of statement execution within a program.
- The Control Statements are:

CALL	calls a subroutine within the program.
DETACH	disconnects the terminal from its associated program.
DO-UNTIL	causes repetitive execution of a statement until a condition is true.
FOR	causes repetitive execution of a statement.
GOTO	transfers control to another statement.
IF	executes a statement if a condition is true.

IF-THEN-ELSE	allows conditional execution of one of two statements.
OFFERROR	disables trapping of run-time errors.
ONERROR	enables trapping of run-time errors.
RETURN	causes control to return from a subroutine.
SLEEP	suspends program operation for a specified time interval.
STOP	terminates program execution.
USING	executes one statement out of a list of statements.
WHILE	causes a statement to be executed repetitively while a condition is true.
XCALL	calls an external subroutine.

1.2.5 Intertask Communications Statements

- Intertask Communications Statements allow communication between programs.
- The Intertask Communications Statements are:

RECV	receives a message from another program.
SEND	transmits a message to another program.

1.2.6 Input/Output Statements

- Input/Output Statements control the transmission and reception of data between memory and input/output devices.
- The Input/Output Statements are:

ACCEPT	receives a character from a device.
CLOSE	terminates use of an input/output channel and closes the associated file.
DELETE	deletes a record from an indexed file.
DISPLAY	sends a character string to a device.
FORMS	sends special printer control codes.
LPQUE	requests a file to be printed.
OPEN	initializes a file in preparation for input/output operations.
READ	reads a record from a file (direct access).
READS	reads the next record in sequence from a file.
STORE	adds a record to an indexed file.
UNLOCK	releases a record for use by another program.
WRITE	writes a record to a file (direct access).
WRITES	writes the next record in sequence to a file.

1.3 PROGRAM STRUCTURE

A DIBOL program contains two major parts: a Data Division and a Procedure Division. The Data Division contains statements that define and identify the data used by the program. The Procedure Division contains statements that execute certain tasks.

Figure 1-1 shows a schematic drawing of a DIBOL-83 program structure.

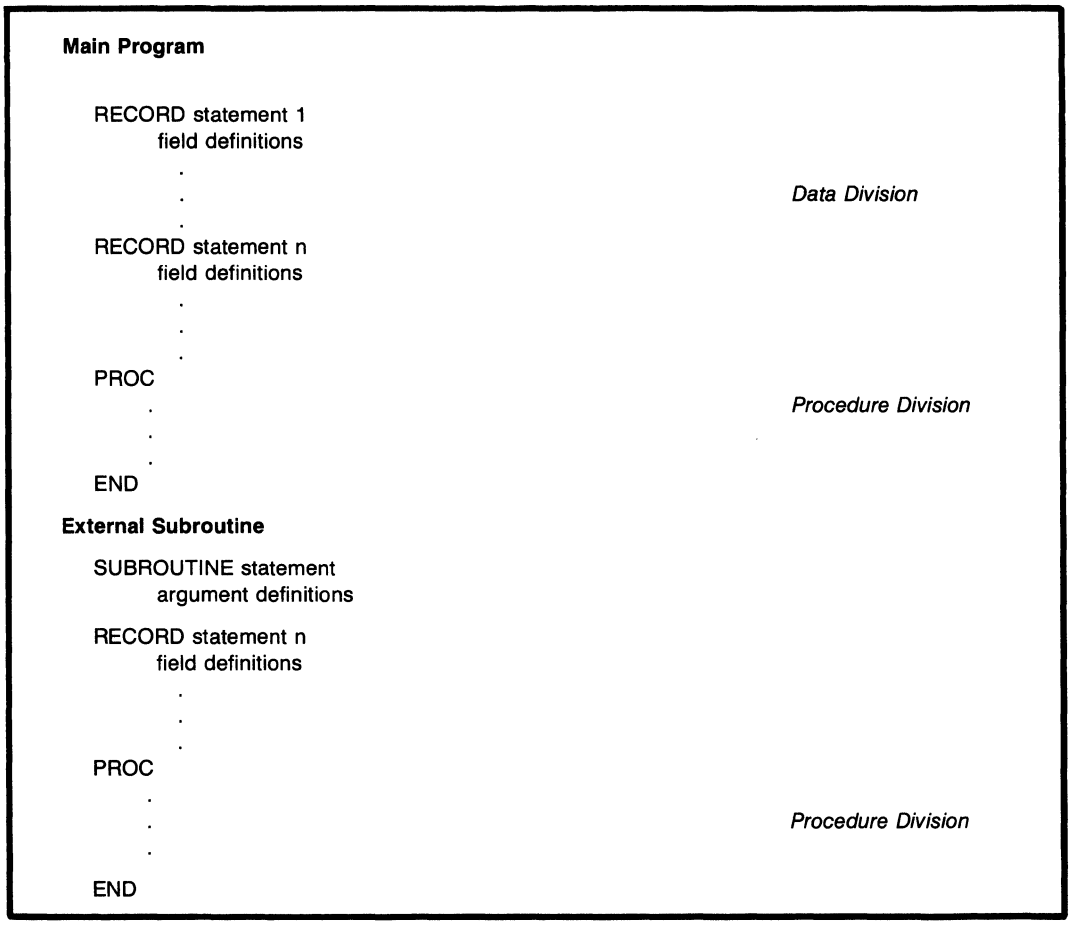


Figure 1-1 DIBOL-83 Program Structure

1.4 STATEMENT LINE SYNTAX

GENERAL RULES

- A statement line can contain 511 characters.
- A program may contain no more than one statement per line.
- A statement can begin anywhere on a line.

RULES FOR LINE CONTINUATION

- The ampersand symbol (&) specifies line continuation. This allows lengthy statements to be continued onto additional physical lines.
- The ampersand symbol must be placed at the first character position in the continuation line.
- A statement can be continued for a maximum of 511 characters, including ampersand symbols, spaces, horizontal tabs, Carriage Return, and Line Feed characters.
- Comments **cannot** be continued by an ampersand; they require a semicolon.

RULES FOR DELIMITERS

- Delimiters separate the elements of the language (keywords, labels, symbols, literals).
- Delimiters are listed in Table 1-1.

Table 1-1
DIBOL-83 Delimiters

Name	Symbol	Name	Symbol
Addition	+	Percent	%
Colon	:	Period	.
Comma	,	Pound	–
Division	/	Right Parenthesis	)
Double Quotes	''	Single Quote	'
Equal	=	Space	
Left Parenthesis	(	Subtraction	—
Multiplication	*	Tabs	< TAB >

RULES FOR COMMENTS

- Comments are used to explain the source program.
- Comments are ignored by the compiler.
- Comments are preceded by a semicolon (;).
- Comments can follow a statement on a line.
- Comments can be placed on any statement line by preceding the comment with a semicolon (;).
- Comments can be placed on a line all by themselves (full line comments).
- Comments **cannot** be continued by an ampersand; they require a semicolon.

COMPILER ERROR CONDITIONS

- Line too long
- Extra characters at statement end

RUN-TIME ERROR CONDITIONS

None

EXAMPLES

Statement Line Syntax

Lengthy statement lines can be continued onto additional lines. A comment cannot be placed on the first of the two lines; a comment can only be placed at the logical end of the statement as shown.

```
TEST1,  IF (INVENT+ORDER .GT. SHIPPD
&       .AND.CASH.GT.MINIM) GOTO GETMOR ;Time to order?
```

The following examples illustrate comments. The first example shows a commented statement and the second example shows a full line comment.

```
RECORD CUST                 ; Customer record

; This program prints the Accounts Past Due Report
```

Comments can be continued onto multiple lines by using a semicolon as follows:

```
READS (1,CUST,EOF)      ; Read the sequentially next
                        ; ... customer master file
```

The basic elements of the language are separated by delimiters. In the following example, the space used as a delimiter between the keyword GOTO and the label TEST1 is missing. This statement will generate a compiler error.

```
GOTOTEST1
```

The following statement will also generate a compiler error because there is an extra space in the middle of the label TEST1:

```
GOTO        TE        ST1
```

1.5 PROCEDURE DIVISION STATEMENT LABELS

- A statement label is a unique symbolic name that identifies a statement in the Procedure Division of a DIBOL program.
- A label consists of up to 6 characters, the first of which must be alphabetic. The remaining characters can be alphabetic, numeric, $, or _(underscore).
- Only the first 6 characters of a label are significant; remaining characters are ignored.
- A label may begin anywhere on a line as long as it immediately precedes and is separated from its associated statement by a comma.
- A label cannot be used to identify more than 1 statement.
- Compiler Directives and Declarations (except for BEGIN-END) cannot have labels.

COMPILER ERROR CONDITIONS

- Duplicate label

RUN-TIME ERROR CONDITIONS

- None

EXAMPLES

- The following labels (LOOP6, X_RTN, and BAD$) are all legal:

```
LOOP6, IF I.GT.MAX GOTO DONE

X_RTN, RETURN

BAD$,  WRITES (CH,'Bad Input')
```

- The following label (DO_PAYROLL) is legal but will be truncated to six characters (i.e., DO_PAY):

```
DO_PAYROLL, OPEN (CH,U,'PAYROL.DDF')
```

- The following labels (6X, _RTN, and $BAD) are not legal because they do not begin with a letter:

```
6X,   IF I.GT.MAX GOTO DONE

_RTN, RETURN

$BAD, WRITES (CH,'Bad Input')
```

1.6 ARRAY SUBSCRIPTING

DEFINITION

Array Subscripting references a specific variable within an array of variables (see section 2.5 for information on array definitions).

FORMAT

array (*subscript*)

where:

array	is an alpha array, decimal array , or a record being referenced.
subscript	is a decimal expression that refers to a field in an array.

RULES

- Array subscripting can be used in any Procedure Division statement where a data field of the same type is allowed.
- *Subscript* indicates the specific field to be referenced within the array.
- *Subscript* should be between 1 and the number of fields in the array as specified in the Data Division.
- If *subscript* exceeds the number of fields within the array, portions of other fields may be referenced. A **Subscript error** occurs when *subscript* specifies a field which is outside the Data Division.
- A reference to an array without *subscript* accesses the first field in the array.

COMPILER ERROR CONDITIONS

- Invalid data type
- Invalid array element count
- Subscript too complex
- Too many subscripts

RUN-TIME ERROR CONDITIONS

- 7 T Subscript error

EXAMPLES

- The following examples all assume that the Data Division contains the following information:

```
RECORD
        NAME,  4A3, 'LAS', 'FIR', 'MID', 'ADD'
        CODE,  D4, 0617, 1739, 5173, 2480
PROC
```

- Using an array name without a subscript will access the first element of the array as shown in the following examples:

```
Field       Data Accessed

NAME        LAS

CODE        0617
```

- The following examples illustrate the use of subscripts with array names:

```
Field       Data Accessed

NAME(1)     LAS

NAME(3)     MID

NAME(4)     ADD

CODE(1)     0617

CODE(4)     2480
```

- Data beyond the end of the array can also be accessed as in the following examples:

```
Field       Data Accessed

NAME(5)     061

NAME(6)     717
```

- If the data to be accessed is beyond the end of the Data Division a **Subscript error** will occur. For example:

```
Field       Data Accessed

CODE(5)     Subscript error

NAME(10)    Subscript error
```

1.7 LITERALS

DEFINITION

Literals are alpha or decimal values permanently defined in a program.

RULES

- A literal cannot be altered during program execution.
- Alpha literals are specified by enclosing a character string within a pair of either single (') or double quote ('') characters.
- Double or single quotes can appear within literals following these guidelines:
  - — A single quote can appear in a literal that is enclosed in single quotes by immediately following the quote character with a second quote character ('O''Hare') within the literal.
  - — A single quote can appear in a literal that is enclosed in double quotes (''O'Hare'').
  - — A double quote can appear in a literal that is enclosed in double quotes by immediately following the double quote character with a second double quote character within the literal ('' '' ''END of FILE'' '' '').
  - — A double quote can appear in a literal that is enclosed in single quotes (' ''END OF FILE'' ').
- Literals cannot be subscripted.
- Decimal literals can be any valid DIBOL number.
- The value for a literal can be whatever is possible for that data type.

COMPILER ERROR CONDITIONS

- End quote missing
- Decimal literal too big, truncated to — ***
- Invalid numeric literal

RUN-TIME ERROR CONDITIONS

- None

EXAMPLES

The following numbers are all legal decimal literals:

```
-99234780113

+000431

10000000000

--1 (same as +1)
```

The following numbers are not legal decimal literals because they contain characters other than the plus sign (+), the minus sign (—), and the decimal digits (0 through 9).

```
$10

1,000,000

10.00
```

The following are legal alpha literals:

```
"PAYROLL NUMBER"

'Invalid customer number'

'$10'

"1,000,000"
```

The apostrophe character (') can be used in the literal by inserting two apostrophes for each one desired, or by using the quote character ('') to start and end the literal. Both of the following literals puts a single apostrophe character in O'Hare.

```
'O''Hare'

"O'Hare"
```

1.8 SUBSTRINGS

DEFINITION

Substrings reference a portion of a variable.

FORMAT

field (*start*,*end*)

where:

field	is an alpha field, decimal field, or record being referenced.
start	is a decimal expression that specifies the position of the first character of the data.
end	is a decimal expression that specifies the position of the last character of the data.

RULES

- Substrings can be specified in any Procedure Division statement where a data field of the same type is allowed.
- The starting position must be greater than or equal to 1.
- The starting position must be less than or equal to the ending position.
- The ending position should not exceed the field size as specified in the Data Division.
- If the ending position exceeds the field size, portions of other fields may be referenced. A **Subscript error** occurs when a subscript specifies data which is outside the Data Division.
- If the length of a decimal substring is greater than 18 characters a **Number too long** error is generated.

COMPILER ERROR CONDITIONS

- Invalid array element count
- Stack overflow
- Subscript too complex
- Too many subscripts

RUN-TIME ERROR CONDITIONS

- 7 T Subscript error
- 15 T Number too long

EXAMPLES

All of the following examples assume that the Data Division contains the following information:

```
RECORD  REC
        AM,     A13, 'abcdefghijklm'
        NZ,     A13, 'nopqrstuvwxyz'
        NUM,    D10, 1234567890
PROC
```

The following examples illustrate the use of substrings:

Field	Data Accessed
AM(2,3)	bc
AM(4,4)	d
AM(10,13)	jklm
REC(1,10)	abcdefghij
REC(27,28)	12
NUM(4,8)	45678
NUM(10,10)	0
NZ(12,13)	yz
AM(NUM(5,5),9)	efghi

Any data that is beyond the end of the named field can be accessed as illustrated in the following examples:

Field	Data Accessed
AM(12,15)	lmno
NZ(13,15)	z12

If the data to be accessed is beyond the end of the Data Division a **Subscript error** will occur. For example:

Field	Data Accessed
NUM(10,11)	Subscript error
NZ(30,30)	Subscript error

1.9 DECIMAL EXPRESSIONS

DEFINITION

Decimal expressions are valid combinations of operands and operators.

GENERAL RULES

- If X and Y are operands, the following are decimal expressions:
    - X binary operator Y
    - unary operator X
    - (X)
- Operators in a decimal expression represent various arithmetic, relational, or Boolean functions of the DIBOL-83 language (see Table 1-2).
- Decimal expressions are allowed on the right side of the equal sign (=) in value assignment statements and in any Procedure Division Statement (except XCALL) where a decimal literal can be used.
- Unary operators require 1 operand.
- Binary operators require 2 operands.
- Operators require operands to be the correct data type.
- Decimal expressions are evaluated according to the order of precedence shown in Table 1-2. Operators with equal precedence are evaluated from left to right in a decimal expression.
- The order of expression evaluation can be altered by using parentheses. Expressions enclosed in parentheses are evaluated before other elements of the decimal expression in which they appear. Additional levels of precedence are achieved by nesting; the innermost expressions are evaluated first.

RULES FOR +, —, *, AND /

- Decimal expressions deal with integers only. So output data can be correctly formatted for printing (see section 3.2.5), the position of an implied decimal point in a decimal value must be determined by the program.
- Decimal expressions that produce intermediate results exceeding 18 digits generate the error **Number too long.**
- The unary plus (+) operator has no effect on a value since unsigned values are assumed to be positive. This operator is useful only to facilitate reading a program listing.

- The unary minus (—) operator is used to negate its operand. Successive minuses are combined algebraically.
- The addition (+), subtraction (—), multiplication (*) and division (/) operators perform standard signed integer arithmetic.
- Division by zero is illegal.
- Any fraction resulting from division is truncated.

RULES FOR #

- The rounding number operator (#) specifies numeric rounding.
- The first operand specifies the numeric value to be rounded.
- The second operand is a decimal expression that evaluates to a number between 0 and 15 which specifies the number of rightmost digits to truncate after rounding takes place.
- The least significant digit of the truncated value is rounded upward by 1 if the digit to its right is greater than or equal to 5.

RULES FOR RELATIONAL OPERATORS

- Relational expressions produce decimal results (either true (non-zero) or false (zero)). These expressions can be used as operands with Boolean operators.
- In comparisons using relational operators, only like data types are allowed as operands, i.e., decimal/decimal or alpha/alpha.
- In an alpha relational comparison, the operand values are compared on a character by character basis from left to right. The comparison is limited to the size of the shortest operand.

COMPILER ERROR CONDITIONS

- None

RUN-TIME ERROR CONDITIONS

- 15 T Number too long
- 30 T Divide by zero

Table 1-2
Table of Operator Precedence

(from highest to lowest)

Operator	Description
( )	parentheses
+ and —	unary plus and unary minus
#	rounding
* and /	multiplication and division
+ and —	addition and subtraction
.EQ. .NE. .GT. .LT. .GE. .LE.	relational comparison
.NOT.	unary logical operator which changes true to false and false to true
.AND.	Boolean AND
.OR. and .XOR.	Boolean OR and exclusive OR

Table 1-3
Unary Operator Table

The following table indicates the legal data type(s) which can be used as an operand for a particular unary operator. The data type result is also shown.

UNARY OPERATORS

	+	—	NOT
Operand Data Type	D	D	D
Result Data Type	D	D	D

Table 1-4
Binary Operator Table

The following table indicates the legal data type(s) that can be used as an operand for a particular unary operator. The data type result is also shown.

BINARY OPERATORS

Data Types of:	+	—	#	*	/	EQ	NE	GT	LT	GE	LE	OR	XOR	AND
Operands	D	D	D	D	D	A/D	A/D	A/D	A/D	A/D	A/D	D	D	D
Result	D	D	D	D	D	D	D	D	D	D	D	D	D	D

Table 1-5
Truth Table

Boolean AND			Boolean OR			Boolean XOR			Boolean NOT	
exp .AND. exp		Result	exp .OR. exp		Result	exp .XOR. exp		Result	.NOT. exp	Result
true	true	true	true	true	true	true	true	false	true	false
true	false	false	true	false	true	true	false	true	false	true
false	true	false	false	true	true	false	true	true		
false	false	false	false	false	false	false	false	false		

EXAMPLES

The following examples all assume that the Data Division contains the following information:

```
RECORD
          MONEY,   D6, 127654
          Y,       D3, -326
          A,       D1, 4
          B,       D2, 10
          C,       D2, 20
          D,       D1, 5
PROC
```

The following examples illustrate the use of arithmetic operators:

Expression	Result
A+B-C	-6
A*D	20
C/D	4
B/A	2 (The remainder is discarded)

The order of evaluation of the subexpressions can be modified by using parentheses, as in the following examples:

Expression	Result
B+C/D*A	26
B+C/(D*A)	11
(B+C)/(D*A)	1 (The remainder is discarded)
((B+C)/D)*A	24

The following examples illustrate the use of the rounding operator (*#*):

```
Expression          Result

MONEY#A             13

Y#2                 -3

Y#A                  0

(MONEY+Y)#1         12733

Y#1                 -33
```

The Relational and Boolean operators produce true (non-zero) or false (zero) results. These operators are most commonly used in the IF, IF-THEN-ELSE, DO-UNTIL, and WHILE statements. They can be used anywhere that a decimal expression is allowed. The following examples illustrate the use of these operators:

```
Expression          Result

A.EQ.4              1 (true)

A.NE.4              0 (false)

'ABC'.EQ.'DEF'      0 (false)

A.EQ.4.AND.B.EQ.10  1 (true)

A.AND.B             1 (true)

A.AND.0             0 (false)
```

CHAPTER 2
DATA DIVISION

2.1 INTRODUCTION

This chapter contains information on Data Division Statements.

The Data Division is the first division of a DIBOL-83 program. It contains RECORD and COMMON statements and associated field definitions that define all program variables. Variables used in the Procedure Division of a program must be defined in the Data Division. The Data Division also contains a SUBROUTINE statement if the program is an external subroutine. These statements are separated from the Procedure Division by the PROC statement.

RECORD STATEMENT

2.2 RECORD STATEMENT

FUNCTION

RECORD defines the areas of memory where variable data is stored.

FORMAT

RECORD [*name*][,*X*]

where:

name is the record name.

X is the redefinition indicator.

GENERAL RULES

- Storage is allocated contiguously in memory in the order the RECORD statements appear in the program.
- RECORD must be followed by at least 1 field definition.
- The total size of the fields within a named record cannot exceed 16,383 characters.

RULES FOR RECORD NAME

- A RECORD *name* consists of up to 6 characters, the first of which must be alphabetic. Remaining characters can be alphabetic, numeric, $, or _ (underscore).
- Only the first 6 characters of a RECORD *name* are significant; remaining characters are ignored.
- A *name* cannot be used to identify more than 1 RECORD area, COMMON area, or field.
- If a RECORD *name* is not specified, only named fields within that record can be referenced.

RULES FOR REDEFINITION INDICATOR

- The redefinition indicator permits redefinition of fields within the record being redefined.
- An redefining RECORD references the same memory area as the record being redefined.
- The new field definitions are specified following the redefining RECORD statement.
- The size of the redefining RECORD (the sum of the sizes of all its fields) must not be greater than the size of the record being redefined.

- In a main program, RECORD can redefine RECORD or COMMON.
- In an external subroutine RECORD cannot redefine COMMON and vice versa. RECORD can redefine RECORD.
- Fields in a redefining RECORD cannot be assigned initial values.

COMPILER ERROR CONDITIONS

- Overlay error

RUN-TIME ERROR CONDITIONS

- None

EXAMPLES

The following record *names* (6X and _PAY) are not legal because they do not begin with an alphabetic character:

```
RECORD    6X

RECORD    _PAY
```

The following example shows a record (OUTPUT) used to format printed output data. The values for MN, DAY, and YR are obtained from Procedure Division Statements. The unnamed fields contain initial values used for formatting the output record.

```
RECORD OUTPUT
       ,       A8, 'Date is '
       MN,     D2              ; Month goes here
       ,       A1, '/'
       DAY,    D2              ; Day goes here
       ,       A1, '/'
       YR,     D2              ; Year goes here
```

In the following example, the record (OUTPUT) has been redefined so that the date (in the format mm/dd/yy) can be more easily accessed. A statement that accesses the DATE field will receive the contents of the MN, DAY, and YR fields separated by the slash character (/).

```
RECORD OUTPUT
       ,       A8, 'Date is '
       MN,     D2              ; Month goes here
       ,       A1, '/'
       DAY,    D2              ; Day goes here
       ,       A1, '/'
       YR,     D2              ; Year goes here

RECORD ,X
       ,       A8              ; Redefines 'Date is '
       DATE,   A8              ; Redefines MN / DAY / YR
```

COMMON STATEMENT

2.3 COMMON STATEMENT

FUNCTION

COMMON defines the areas in memory where variable data is stored. This data is to be shared between the main program and external subroutines.

FORMAT

COMMON [*name*][,*X*]

where:

name is the COMMON name.

X is the redefinition indicator.

GENERAL RULES

- COMMON must be followed by at least 1 field definition.
- The total size of the fields within a named COMMON area cannot exceed 16,383 characters.
- COMMON is similar to RECORD except that fields defined within a COMMON area are available for use by the main program or by any external subroutine.
- If COMMON appears in a main program, space is allocated in memory just as it is done for a RECORD statement.
- If COMMON appears in an external subroutine, memory is not allocated. All fields that appear in the subroutine's COMMON area must reference the main program's COMMON area.
- Data cannot be shared between two external subroutines via the COMMON statement unless the data is defined in the main program.
- COMMON and RECORD areas may be intermixed in the Data Division.
- When the main program is linked with its external subroutines, a correlation is made between the field names defined in the COMMON areas of the subroutine and those of the main program.
- If a field is named in a COMMON area of an external subroutine but there is no corresponding field name in the main program, an error message is generated when the program is linked.
- It is not necessary for the COMMON area of an external subroutine to contain all the COMMON fields defined in the main program unless all are needed. For those that are needed it is necessary that fields of the same types, names, and sizes be defined in the Data Division of the main program and external subroutine. It is important that the sizes and types correspond. Otherwise the operation will be incorrect and unpredictable problems may occur.

- Fields in COMMON areas in subroutines cannot be assigned an initial value.
- The fields in the COMMON area of the subroutine do not need to be defined in the same order as they are in the main program. The data is stored according to the order of the main program's field definitions.

RULES FOR COMMON NAMES

- A COMMON *name* consists of up to 5 characters, the first of which must be alphabetic. Remaining characters can be alphabetic, numeric, $, or _(underscore).
- Only the first 5 characters of a COMMON *name* are significant; remaining characters are ignored.
- A *name* cannot be used to identify more than 1 RECORD area, COMMON area, or field.
- If a COMMON *name* is not specified, only named fields within that COMMON area can be referenced.
- For each COMMON *name* and field name within COMMON, the compiler appends a dollar sign ($). This sign ($) ensures that COMMON *names* are unique and do not conflict with other global names.

RULES FOR REDEFINITION INDICATOR

- The redefinition indicator permits redefinition of fields within the record being redefined.
- A redefining COMMON references the same memory area as the record being redefined.
- The new field definitions are specified following the redefining COMMON statement.
- The size of the redefining COMMON (the sum of the sizes of all its fields) must not be greater than the size of the record being redefined.
- In a main program, COMMON can redefine RECORD or COMMON and vice versa.
- In an external subroutine COMMON cannot redefine RECORD and vice versa.
- Fields in a redefining COMMON cannot be assigned initial values.

COMPILER ERROR CONDITIONS

- None

RUN-TIME ERROR CONDITIONS

- None

EXAMPLES

The following COMMON *names* (REC6, A_REC, and BAD$) are all legal:

```
COMMON REC6

COMMON A_REC

COMMON BAD$
```

The following COMMON *name* (PAYROLL_RECORD) is legal but it will be truncated to 5 characters (i.e.,PAYRO):

```
COMMON PAYROLL_RECORD
```

The following example contains a main program which has two COMMON areas and two external subroutines. One subroutine (XSUB2) uses both COMMON areas, while the other subroutine (XSUB1) uses only one. Neither of the two subroutines allocates memory storage area for the COMMON areas; instead, the subroutines' COMMON areas point to the main program's memory storage area.

```
                    Main Program

COMMON    EMP                       ; Employee record
          NAME,    A20              ; Employee name
          SAL,     D5               ; Salary
COMMON
          DATE,    D5               ; Current date

                  Subroutine XSUB1

COMMON
          DATE,    D5               ; Current date

                  Subroutine XSUB2

COMMON
          DATE,    D5               ; Current date
COMMON    EMP                       ; Employee record
          NAME,    A20              ; Employee name
          SAL,     D5               ; Salary
```

2.4 FIELD DEFINITIONS

FUNCTION

Field definitions define variables within a RECORD or COMMON area.

FORMAT

[*name*], | A / *D* | *n* [,*literal*]

where:

name is the field name.

A declares the field to be alpha.

D declares the field to be zoned decimal.

n is the size of the field.

literal is an alpha literal or a decimal literal (initial value).

GENERAL RULES

- The field size in an alpha field cannot exceed 16,383.
- The field size in a zoned decimal field cannot exceed 18.

RULES FOR FIELD NAME

- A field *name* in a RECORD consists of up to 6 characters, the first of which must be alphabetic. Remaining characters can be alphabetic, numeric, $, or _(underscore).
- Only the first 6 characters of a field *name* in a RECORD are significant; remaining characters are ignored.
- A field *name* in a COMMON area consists of up to 5 characters, the first of which must be alphabetic. Remaining characters can be alphabetic, numeric, $, or _(underscore).
- Only the first 5 characters of a field *name* in a COMMON area are significant; remaining characters are ignored.
- A *name* cannot be used to identify more than 1 RECORD area, COMMON area, or field.
- If no *name* is used, the field can be accessed either as part of the entire record by using the record *name*, or by subscripting down from a previous record or field.

RULES FOR SETTING INITIAL VALUES

- The initial value of a field is set by inserting a *literal* after the type and size specification.
- A comma must be used to separate the *literal* from the preceding type and size specification.
- The *literal* must be the same data type and should contain the same number of characters or digits as specified for the field.
- If the *literal* is longer than the field size, a warning is generated during program compilation.
- If the *literal* is shorter than the field size the initial value will be left-justified (for alpha literals), or right-justified (for decimal literals).
- Leading signs (+ and —) in decimal literals, as well as delimiting apostrophes in alpha literals, are not counted when calculating the size of a *literal*.
- If no initial value is specified, the field is initialized to all spaces if it is an alpha field, or to all zeros, if it is a decimal field.

COMPILER ERROR CONDITIONS

- Overlay error

RUN-TIME ERROR CONDITIONS

- None

EXAMPLES

The following field *names* (DATE, ER_1, and CTR$) are all legal:

```
RECORD
        DATE,    A11          ; Date (dd-mmm-yyyy)
        ER_1,    D1           ; Error indicator
        CTR$,    D2           ; Counter
```

The following field *name* (EMPLOYEE_NAME is legal but will be truncated to 6 characters (i.e., EMPLOY). The same field *name* will be truncated to 5 characters when used in a COMMON area (i.e., EMPLO).

```
RECORD
        EMPLOYEE_NAME,    A20        ; Employee name
```

The following record contains both named and unnamed fields. The 3 unnamed fields all have initial values (named fields can also have initial values). The third field is a 2 character alpha field; however, the initial value for the field contains only a single right parenthesis character ()). The initial value will be left justified in the A2 field and the rightmost character will be cleared to a space.

```
RECORD
        ,       A1, '('
AREA,   D3                      ; Area code
        ,       A2, ')'
EXCH,   D3                      ; Telephone exchange
        ,       A1, '-'
NMBR,   D4                      ; Telephone number
```

The following example shows 2 decimal fields which have initial values. The first field (LINE) is a 2 digit decimal field; however, the initial value is only a single digit. The initial value will be right justified in LINE and the leftmost digit in LINE will be cleared to a zero.

```
RECORD
        LINE,   D2, 1           ; Line number
        COLUMN, D2, 80          ; Column number
```

ARRAY DEFINITIONS

2.5 ARRAY DEFINITIONS

FUNCTION

An array is a group of fields which share the same data type, field size, and symbolic name (array name).

FORMAT

[*name*], *m* | *A* / *D* | *n* [,*literal*]

where:

name is the field name.

m is the array field count.

A declares the field to be alpha.

D declares the field to be zoned decimal.

n is the size of each field in the array.

literal is an alpha literal or a decimal literal (initial value).

GENERAL RULES

- The field size in an alpha field cannot exceed 16,383.
- The field size in a zoned decimal field cannot exceed 18.

RULES FOR FIELD NAME

- An array *name* in a RECORD consists of up to 6 characters, the first of which must be alphabetic. Remaining characters can be alphabetic, numeric, $, or _(underscore).
- Only the first 6 characters of an array *name* in a RECORD are significant; remaining characters are ignored.
- An array *name* in a COMMON area consists of up to 5 characters, the first of which must be alphabetic. Remaining characters can be alphabetic, numeric, $, or _(underscore).
- Only the first 5 characters of an array *name* in a COMMON area are significant; remaining characters are ignored.
- A *name* cannot be used to identify more than 1 RECORD area, COMMON area, or field.
- If no *name* is used, the fields within the array can be accessed either as part of the entire record by using the record *name*, or by subscripting down from a previous record or field.

RULES FOR ARRAY FIELD COUNT

- The array field count (*m*) may be any positive decimal value up to 8191.
- If no array field count is specified, it is assumed to be 1.
- Array data is referenced by using the array variable name with a subscript.

RULES FOR SETTING INITIAL VALUES

- The initial value of a field is set by inserting a *literal* after the type and size specification.
- A comma must be used to separate the *literal* from the preceding type and size specification.
- The *literal* must be the same data type and should contain the same number of characters or digits as specified for the field.
- If the *literal* is longer than the field size, a warning is generated during program compilation.
- If the *literal* is shorter than the field size the initial value will be left-justified (for alpha literals), or right-justified (for decimal literals).
- Leading signs (+ and —) in decimal literals, as well as delimiting apostrophes in alpha literals, are not counted when calculating the size of a *literal*.
- If no initial value is specified, an alpha field is initialized to all spaces and a decimal field is initialized to all zeros.
- Fields within an array may be initialized by specifying a series of initial values separated from each other by commas.
- It is not necessary to initialize all fields of an array, but array fields that are to be initialized must reside at the beginning of the array and must be contiguous.

COMPILER ERROR CONDITIONS

- Invalid array field count

RUN-TIME ERROR CONDITIONS

- None

EXAMPLES

The arrays (DAYS and MONTHS) in the following example have initial values for all of their fields:

```
RECORD
      DAYS,   12D2, 31,28,31,30,31,30,31,31,30,31,30,31
      MONTHS, 12A3, 'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun'
&                 , 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'
```

2.6 SUBROUTINE STATEMENT

FUNCTION

SUBROUTINE identifies a program as an external subroutine.

FORMAT

SUBROUTINE *name*

where:

name is the subroutine name.

RULES

- SUBROUTINE must be the first statement (excluding Compiler Directives and/or comments) in the Data Division of an external subroutine.
- SUBROUTINE is used to establish a logical connection between the subroutine and the calling program.
- SUBROUTINE may be followed by 1 or more argument definitions.

RULES FOR SUBROUTINE NAME

- A subroutine *name* consists of up to 6 characters, the first of which must be alphabetic. Remaining characters can be alphabetic, numeric, $, or _ (underscore).
- Only the first 6 characters of a subroutine *name* are significant; remaining characters are ignored.

COMPILER ERROR CONDITIONS

- None

RUN-TIME ERROR CONDITIONS

- None

2.6.1 Subroutine Argument Definition

FUNCTION

Subroutine argument definitions specify the data linkages between an external subroutine and the program that called the external subroutine.

FORMAT

[*name*], | A |
          | D |

where:

name is the subroutine's internal name for the subroutine argument.

A declares the field to be alpha.

D declares the field to be zoned decimal.

RULES

- If a record is passed as an argument, references cannot be made to its fields; the entire record can only be referred to in a subroutine as a single alpha field.
- The size of the argument is the size of the data as specified in the calling program.
- Argument definitions should correspond in number and data type with the arguments specified in the XCALL statement in the calling program.
- The first argument definition specified refers to the data item referenced in the first argument in the XCALL statement. The second argument definition refers to the second XCALL argument, etc.

RULES FOR SUBROUTINE ARGUMENT NAME

- An argument name consists of up to 6 characters, the first of which must be alphabetic. Remaining characters can be alphabetic, numeric, $, or _(underscore).
- Only the first 6 characters of a subroutine argument name are significant; remaining characters are ignored.
- A *name* cannot be used to identify more than 1 RECORD area, COMMON area, or field.

COMPILER ERROR CONDITIONS

- Subroutine dummy argument not zero

RUN-TIME ERROR CONDITIONS

- None

EXAMPLES

In the following example, the main program calls the external subroutine (CNVRT) to change the format of the date. It passes the arguments DATE and XDATE. These arguments are represented in the subroutine as OLD and NEW.

Main Program

```
RECORD
        DATE,   D6, 010750
        XDATE, A11
PROC
        XCALL CNVRT (DATE,XDATE)   ; Convert the date
        OPEN (1,O,'TT:')           ; Open the terminal
        WRITES (1,XDATE)           ; Display the date
        CLOSE 1                    ; Close the terminal
        STOP
```

External Subroutine

```
SUBROUTINE CNVRT            ; Convert the date format
        OLD,  D             ; Date (mmddyy)
        NEW,  A             ; Date (dd-mmm-yy)

RECORD  ODATE               ; Old date format
        MM,   D2            ; Month
        DD,   D2            ; Day
        YY,   D2            ; Year

RECORD  NDATE               ; New date format
        DAY,  A2            ; Day
        ,     A1,'-'
        MONTH,A3            ; Month
        ,     A1,'-'
        YEAR, D2            ; Year

RECORD
        MNAME,12A3,'Jan','Feb','Mar','Apr','May','Jun'
&                 ,'Jul','Aug','Sep','Oct','Nov','Dec'

PROC
        ODATE=OLD
        DAY=DD              ; Move day to new format
        YEAR=YY             ; Move year to new format
        MONTH=MNAME(MM)     ; Move month to new format
        NEW=NDATE           ; Return new date
        RETURN
```

CHAPTER 3
THE DIBOL-83 PROCEDURE DIVISION STATEMENTS

3.1 INTRODUCTION

The DIBOL-83 Procedure Division statements process data and control program execution. These statements are verbs containing arguments and expressions.

This chapter contains information on value assignment statements, data conversion, and data formatting. The Procedure Division statements are arranged alphabetically for easy reference.

VALUE ASSIGNMENT STATEMENTS

3.2 VALUE ASSIGNMENT STATEMENTS

FUNCTION

Value Assignment statements:

- Move data.
- Store the results of arithmetic expressions.
- Convert and format data.
- Clear variables.

FORMAT

destination = *source*

where:

destination
is a record or field which contains the data to be stored.

source is a record, field, literal, or expression which contains the data to be stored.

RULES

- The contents of the *source* are moved to the *destination.*
- The source data is not altered unless the *destination* location is one of the source elements (for example, A=A+1).
- The *destination* is the field or record defined in a Data Division statement and can be either alpha or decimal.
- The source data is always converted to the data type defined for the *destination.*

3.2.1 Moving Alpha Data

FUNCTION

Value assignment statements move alpha data.

FORMAT

afield = | *afield* / *aliteral* |

where:

afield is an alpha field or record which is the destination.

afield is an alpha field, alpha literal, or record which is the source.
aliteral

RULES

- The source is moved to the destination and is left-justified.
- If the source is smaller than the destination, the unused rightmost character positions in the destination are cleared to spaces.
- If the source data is larger than the destination, the rightmost characters that cause overflow are truncated.

COMPILER ERROR CONDITIONS

- None

RUN-TIME ERROR CONDITIONS

- 8 NT Writing into a literal

EXAMPLES

In the following example, the value of NAME2 (which contains 'Johnson') is moved to NAME1. Since NAME1 is only 4 characters long, only the first 4 characters of 'Johnson' are moved. NAME1 will contain 'John' and the entire record will contain 'JohnJohnson'.

```
RECORD
        NAME1,  A4, 'Fred'
        NAME2,  A7, 'Johnson'
PROC
        NAME1=NAME2
```

In the following example, the value of B (which contains 'FGH') is moved to A. 'FGH' will be left-justified in A and the rightmost characters in A will be cleared to spaces. A will contain 'FGH ' and the entire record will contain 'FGH FGH'.

```
RECORD
        A,      A5, 'ABCDE'
        B,      A3, 'FGH'
PROC
        A=B
```

3.2.2 Moving Decimal Data

FUNCTION

Value assignment statements move decimal data.

FORMAT

dfield = dexp

where:

dfield is a decimal field which is the destination.

dexp is a decimal expression which is the source.

RULES

- The sign of the source data is preserved in the destination field.
- The source is moved to the destination and is right-justified.
- If the source is smaller than the destination, the unused leftmost digit positions in the destination are cleared to zeros.
- If the source is larger than the destination, the leftmost digits that cause overflow are truncated.

COMPILER ERROR CONDITIONS

- None

RUN-TIME ERROR CONDITIONS

- 8 NT Writing into a literal

EXAMPLES

In the following example, the value of A (which contains 1234) is moved to B. Since B is shorter than A, 1234 is right-justified in B and the digits that cause overflow (12) are truncated. B will contain 34.

```
RECORD
        A,      D4, 1234
        B,      D2
PROC
        B=A
```

In the following example, the value of A (which contains 1234) is moved to C. Since C is longer than A, 1234 is right-justified in C and the leftmost digits are cleared to zero. C will contain 0000001234.

```
RECORD
        A,      D4, 1234
        B,      D2,
PROC
        C=A
```

In the following example the result of A*B (1234*34 = 41956) is moved to C. Since C is only 4 digits long, 4196 is right-justified in C and the leftmost digit is truncated. C will contain 1956.

```
RECORD
        A,      D4, 1234
        B,      D2, 34
        C,      D4
PROC
        C=AB
```

3.2.3 Alpha-to-Decimal Conversion

FUNCTION

Value assignment statements convert alpha data to its decimal equivalent.

FORMAT

$$dfield = \left| \begin{array}{l} afield \\ aliteral \end{array} \right|$$

where:

dfield is a decimal field which is the destination.

$\left| \begin{array}{l} afield \\ aliteral \end{array} \right|$ is an alpha field, alpha literal, or record which is the source.

RULES

- The source may contain up to 18 digits with any number of plus (+) or minus (—) characters. Plus and minus characters are treated as unary operators and are combined algebraically.
- Spaces in the source are ignored.
- The source is moved to the destination and is right-justified.
- If the source is smaller than the destination, the unused leftmost digit positions in the destination are cleared to zeros.
- If more than 18 digits are moved, or, if the source is larger than the destination, the leftmost digits that cause overflow are truncated.

COMPILER ERROR CONDITIONS

- None

RUN-TIME ERROR CONDITIONS

- 20 T Bad digit
- 8 NT Writing into a literal

EXAMPLES

In the following example, the value of A (which contains '910111213141') is moved to B.

Since B is shorter than A,'910111213141' is right-justified in B and the digits that cause overflow (91) are truncated. B will contain 011213141.

```
RECORD
        A,      A12, '910111213141'
        B,      D10
PROC
        B=A
```

In the following example, the value of A (which contains '65444321') is moved to C. Since C is longer than A, '65444321' is right-justified in C and the leftmost digits are cleared to zero. C will contain 0065444321.

```
RECORD
        A,      A8, '65444321'
        C,      D10
PROC
        C=A
```

In the following example, the value of A (which contains '-0065432178') is moved to C. C will contain 006543217x. The 'x' is the internal representation for -8 (see Appendix A).

```
RECORD
        A,      A11, '-0065432178'
        C,      D10
PROC
        C=A
```

3.2.4 Decimal-to-Alpha Conversion

FUNCTION

Value assignment statements convert decimal data to its alpha equivalent.

FORMAT

afield = *dexp*

where:

afield is an alpha field or record which is the destination.

dexp is a decimal expression which is the source.

RULES

- The source is moved to the destination and is right-justified.
- If the source is negative, an additional character should be allocated in the destination for the minus sign. A leading minus sign is inserted to the left of the leftmost nonspace character in the destination.
- If the source is smaller than the destination, the unused leftmost character positions in the destination are cleared to spaces.
- If the source is larger than the destination, the leftmost characters that cause overflow are truncated.
- Leading zeros are cleared to spaces.
- If the source is zero, a single right-justified zero is moved to the destination; remaining character positions to the left are cleared to spaces.

COMPILER ERROR CONDITIONS

- None

RUN-TIME ERROR CONDITIONS

- 8 NT Writing into a literal

EXAMPLES

In the following example, the value of A (which contains 87654321) is moved to B. Since B is shorter than A, 87654321 is right-justified in B and the digits that cause overflow (8765) are truncated. B will contain '4321'.

```
RECORD
        A,      D8, 87654321
        B,      A4
PROC
        B=A
```

In the following example, the value of A (which contains 1234) is moved to C. Since C is longer than A, 1234 is right-justified in C and the leftmost characters are cleared to spaces. C will contain '1234'.

```
RECORD
        A,      D4, 1234
        C,      A6
PROC
        C=A
```

In the following example, the value of A (which contains 0x, the internal representation for -08 (see Appendix A)) is moved to C. C will contain '-8'.

```
RECORD
        A,      D2, -08
        C,      A3
PROC
        C=A
```

In the following example, the value of A (which contains 000) is moved to C. C will contain '0'.

```
RECORD
        A,      D3, 000
        C,      A3
PROC
        C=A
```

In the following example, the value of A (which contains 123t, the internal representation for -1234 (see Appendix A)) is moved to C. C will contain '234'.

```
RECORD
        A,      D4, -1234
        C,      A3
PROC
        C=A
```

If a decimal field can have a negative value, space must be made for the sign in the alpha field. In the following example, the value of A (which contains -1234) is moved to C. C will contain '1234' with no minus sign.

```
RECORD
        A,      D4, -1234
        C,      A4
PROC
        C=A
```

3.2.5 Formatting Data

FUNCTION

Value assignment statements permit decimal data to be converted to its alpha equivalent and formatted.

FORMAT

afield = *dexp*, *formatstring*

where

afield	is an alpha field or record which is the destination.
dexp	is a decimal expression which is the source.
formatstring	is an alpha field, alpha literal, or record which contains format control characters.

RULES

- The source is formatted according to the format string, moved to the destination, and right-justified.
- If the formatted data is smaller than the destination, the unused leftmost character positions in the destination are cleared to spaces.
- If the formatted data is larger than the destination, the leftmost characters that cause overflow are truncated.
- The format string forms a picture or specification of what the converted data is to look like. It is composed of one or more format control characters (see Table 3-1).
- The format string may also contain other DIBOL-83 characters (except for the format control characters themselves) that are to be inserted in the formatted data.
- The format string should be large enough to represent the entire source, since only those digits that are specified by the format string are moved.

Table 3-1
FORMAT CONTROL CHARACTERS

Character	Description
X	Each X represents a digit position. An X causes a digit in the source to be placed in the corresponding position in the destination. If there are more Xs than source digits, a leading zero is inserted for each additional X.
Z	Each Z represents a digit position. A Z suppresses a leading zero in this character position if Z is to the left of the decimal point (see below). When placed to the right of the decimal point, zeros are suppressed only if all digits are zeros.
*	Each asterisk (*) represents a digit position. It replaces a leading zero with an * symbol in this position.
money sign	Each money sign (for example, $) represents a digit position. It replaces leading zeros beginning at this character position with leading spaces and a single money sign. Format characters to the left of the money sign are ignored. Any character can be used for the money sign by calling the MONEY external subroutine, although it is initially set to $. Any character with an established format meaning should not be used, for example, ,Z,X, —.
,	
—	When used as the first or last character in a format string, the minus sign (—) causes the sign of the number being formatted to be placed in that position. If the number is negative, a minus appears, otherwise a space is inserted. When used elsewhere in a format string, this will cause a minus to be placed in that position in the formatted data. A minus cannot be used to the left of a money sign format character.
	NOTE The following descriptions on the decimal point (.) and comma (,) are reversed when international data formatting is selected via the FLAGS external subroutine.
.	A decimal point (.) causes a decimal point to be inserted in the corresponding position in the formatted data and causes zeros to the right of it to become significant.
,	The comma (,) causes a comma to be inserted in the corresponding position in the formatted data if there are significant digits to the left.

COMPILER ERROR CONDITIONS

- None

RUN-TIME ERROR CONDITIONS

- 8 NT Writing into a literal

EXAMPLES

The following examples assume that the Data Division contains the following fields:

```
RECORD
        F,      A8
```

The following examples illustrate data formatting:

```
Statement                   Result

F= 123, 'xxxxxxxx'          '00000123'

F= 123, 'ZZZZZZZZ'          '     123'

F= 123, '********'          '*****123'

F= 123, '$$$$$$$$'          '    $123'

F= -1123, '-XXX,XXX'        '-001,123'

F= 123, '$$$$$.XX'          '   $1.23'

F= -123, '$***.**-'         '$**1.23-'

F= 12345678,'X,XXX.XX'      '3,456.78'
```

3.2.6 Clearing Variables

FUNCTION

Value assignment statements clear variables.

FORMAT

field =

where:

field is an alpha field, decimal field, or record which is to be cleared.

RULES

- If the destination is an alpha field, it is cleared to spaces.
- If the destination is a decimal field, it is cleared to zeros.
- If the destination is a record containing decimal fields, the entire record, including the decimal fields, is cleared to spaces.
- If the equal sign (=) is followed by anything on the same line (other than a comment) it is treated as an assignment statement.

NOTE

Whenever possible use the CLEAR statement to clear fields.

COMPILER ERROR CONDITIONS

- None

RUN-TIME ERROR CONDITIONS

- 8 NT Writing into a literal

EXAMPLES

When clearing a field, the equal sign (=) cannot be followed by anything on the same line (other than a comment). If anything follows the equal sign, the statement is interpreted as a value assignment statement. In the following example, the statement is not legal. It is interpreted as A=ELSE.

```
IF A.EQ.B THEN A= ELSE STOP
```

See CLEAR for examples on clearing fields. Whenever possible use the CLEAR statement to clear fields.

ACCEPT

3.3 ACCEPT

FUNCTION

ACCEPT inputs a character from a terminal.

FORMAT

ACCEPT (*ch, field*[,*label*])

where:

ch is a decimal expression that evaluates to a channel number as specified in a previous OPEN statement.

field is an alpha field, decimal field, or record which will contain the character input from the terminal.

label is a statement label where control is to be transferred when a (CTRL/Z) is detected.

GENERAL RULES

- ACCEPT is used in I or O mode with a terminal.
- If the RETURN key on a terminal is used, a carriage return character and line feed character are generated.

RULES FOR ACCEPTING INTO AN ALPHA FIELD OR RECORD

- The character is moved to the leftmost character position of the *field* according to the rules for moving alpha data (see section 3.2.1).
- If a CTRL/Z is detected, it is interpreted as a logical end-of-file and no character is input.

RULES FOR ACCEPTING INTO A DECIMAL FIELD

- *Field* should be a 3 digit field.
- The decimal character code is moved to *field* according to the rules for moving decimal data (see section 3.2.2).
- All characters are input. CTRL/Z is input like other characters and does not terminate input.

COMPILER ERROR CONDITIONS

- None

RUN-TIME ERROR CONDITIONS

- 1 T End of file
- 8 NT Writing into a literal
- 10 NT Illegal channel number
- 11 NT Channel not open
- 21 T Bad OPEN

EXAMPLES

The following examples assume that the Data Division contains the following fields:

```
RECORD
        ACHR,       A1
        DCHR,       D3
```

In the following example ACCEPT reads a character into ACHR. When a CTRL/Z is detected, control is transferred to the statement labeled END. If 'A' is typed at the terminal, ACHR will contain 'A'.

```
ACCEPT (3,DCHR,END)
```

In the next example ACCEPT puts the decimal character code for the next character into DCGHR. When accepting into a decimal field, CTRL/Z is treated the same as all other characters. If 'A' is typed at the terminal, DCHR will contain 065 which is the decimal character code for 'A'.

```
ACCEPT (3,DCHR)
```

BEGIN—END

3.4 BEGIN-END BLOCK

FUNCTION

The BEGIN-END block is a sequence of statements preceded by BEGIN and followed by END.

FORMAT

```
BEGIN
  statement
  .
  .
  .
  END
```

where:

statement is a DIBOL Procedure Division statement.

RULES

- The BEGIN-END block may be used wherever a single executable statement is valid.
- Control can be transferred from inside a BEGIN-END block to outside the BEGIN-END block.
- BEGIN may begin on a new line.
- END may begin on a new line.
- BEGIN and END cannot be followed on the same line by any statement.
- The label on BEGIN, if present, is outside the block.
- The label on END, if present, is inside the block.

COMPILER ERROR CONDITIONS

- No END for BEGIN
- Stack overflow

RUN-TIME ERROR CONDITIONS

- None

EXAMPLES

The BEGIN-END block is particularly useful with the IF, IF-THEN-ELSE, DO-UNTIL, FOR, USING, and WHILE statements. In the following example all of the statements within the BEGIN-END block will be executed if LNECTR is greater than MAXCTR.

```
IF LNECTR.GT.MAXCTR      ; Time for a new page?
   BEGIN                 ; Yes--
   FORMS (6,0)           ; Output a form feed
   INCR PAGE             ; Increment the page number
   WRITES (6,TITLE)      ; Output title
   CLEAR LNECTR          ; Reset line counter
   END
```

In the following example the statements within the BEGIN-END block (the READS and the IF) will be repetitively executed until CUSNAM equals SPACES. The IF statement also contains a BEGIN-END block. The statements within this inner BEGIN-END block will be executed if the BALANC is greater than 100.

```
DO
    BEGIN
    READS (1,CUST,EOF)    ; Read a customer record
    IF BALANC.GT.100      ; Owe more than $100?
       BEGIN              ; Yes--
       NAME=CUSNAM        ; Save customer name
       AMT=BALANC         ; Save the balance
       WRITES (6,PLINE)   ; Print name and balance
       END
    END
UNTIL CUSNAM.EQ.SPACES
```

3.5 CALL

FUNCTION

CALL transfers program control to an internal subroutine.

FORMAT

CALL *label*

where:

label is the statement label of the first statement in the subroutine.

RULES

- Each CALL statement must be matched by a RETURN statement.
- The matching RETURN statement causes control to return to the statement logically following the CALL.

COMPILER ERROR CONDITIONS

- Label out of context block: <label name>

RUN-TIME ERROR CONDITIONS

- 66 NT R6 Stack overflow

EXAMPLES

This example shows how program control branches from one subroutine to the next and returns. The solid lines show the control path upon execution of RETURN statements.

```
    CALL PROFIT
    WRITES (6,PROFIT)                   ; Output the profit
    CLOSE 6                             ; Close the file
    STOP

           ; Subroutine to calculate profit

    PROFIT, PBT=PRICE-COST              ; Compute pre-tax profit
            CALL TAX                    ; Get the tax
            PAT=PBT-TAX                   ; Compute post-tax
 profit
            RETURN

           ; Subroutine to calculate tax

            TAX,    TAX=PBT8      ; Compute the tax
            IF TAX.GT.MAX TAX=MAX
            RETURN
```

3.6 CLEAR

FUNCTION

CLEAR sets variables to zeros or spaces.

FORMAT

CLEAR *field*[,...]

where:

field is an alpha field, decimal field, or record.

RULES

- If *field* is an alpha field it is cleared to spaces.
- If *field* is a decimal field, it is cleared to zeros.
- If *field* is a record containing decimal fields, the entire record, including the decimal fields, is cleared to spaces.

COMPILER ERROR CONDITIONS

- None

RUN-TIME ERROR CONDITIONS

- 8 NT Writing into a literal

EXAMPLES

The following examples assume that the Data Division contains the following fields:

```
RECORD  REC
        AFLD, A10
        DFLD, D5
```

The following statement will clear AFLD to all spaces:

```
CLEAR AFLD
```

The following statement will clear DFLD to all zeros:

```
CLEAR DFLD
```

The following statement will clear AFLD to all spaces and will clear DFLD to all zeros:

```
CLEAR AFLD,DFLD
```

When a record is cleared, all fields including decimal fields within the record, are cleared to spaces. The following statement will clear AFLD and DFLD to spaces.

```
CLEAR REC
```

3.7 CLOSE

FUNCTION

CLOSE terminates the use of a channel by closing the associated file and releasing both the I/O channel and the file buffer.

FORMAT

CLOSE *ch*

where:

ch is a decimal expression that evaluates to a channel number as specified in a previous OPEN statement.

RULES

- CLOSE is necessary for channels opened in O and U modes to assure that records remaining in the I/O buffer are output to the file.
- If the channel is open in O mode, CLOSE writes records remaining in the I/O buffer into the file. The end-of-file mark is placed after the last record in the file.
- If the channel is open in U mode, CLOSE writes records remaining in the I/O buffer into the file. The records are automatically unlocked.
- No error is generated if the channel is not opened.

COMPILER ERROR CONDITIONS

- None

RUN-TIME ERROR CONDITIONS

- 10 NT Illegal channel number
- 22 T I-O error
- 25 T Output file full
- 40 T Record locked

EXAMPLES

There are three parts to the following example. First, a new file is created and a single record is written into it. Second, the newly created file is opened for input and the record is read. Finally, the record that was read is displayed on the screen. All I/O operations use the same channel. The channel number can be reused following the CLOSE statement.

```
RECORD
        DAT,     A80
PROC
;
; Create a new file (TEST.DDF)
;
         OPEN (3,O,'TEST.DDF')        ; Create file
         WRITES (3,'This is a test') ; Output a record
         CLOSE 3                      ; Close TEST.DDF
;
; Read the record written into newly created file
;
         OPEN (3,I,'TEST.DDF')  ; Open TEST.DDF for input
         READS (3,DAT)          ; Read a record
         CLOSE 3                ; Close the input file
;
; Display the record that was read
;
         OPEN (3,O,'TT:')       ; Open the terminal
         WRITES (3,DAT)         ; Display the data
         CLOSE 3                ; Close the terminal
         STOP
```

DELETE

3.8 DELETE

FUNCTION

DELETE eliminates a record from an indexed file.

FORMAT

DELETE (*ch*,*keyfld*)

where:

ch is a decimal expression that evaluates to a channel number as specified in a previous OPEN statement.

keyfld is an alpha field or record which identifies the record to be deleted.

RULES

- DELETE is used in U:I mode.
- The record to be deleted is the record most recently read on the specified channel and must still be locked.
- DELETE serves as a signal to the file system that the record is no longer valid. The action taken is system dependent.
- *Keyfld* must match the key field of the last record read.

COMPILER ERROR CONDITIONS

- None

RUN-TIME ERROR CONDITIONS

- 10 NT Illegal channel number
- 11 NT Channel not open
- 21 T Bad OPEN
- 22 T I-O error
- 40 T Record locked
- 52 T Illegal key
- 53 T Key not same
- 61 T No current record

EXAMPLES

In the following example all of the customer records in the indexed file are read. When a customer with a balance of less than $20 is found, that customer's record is deleted.

```
RECORD REC
       NAME,     A10                 ; Customer name
       BAL,      D6                  ; Customer balance
PROC
       OPEN (1,U:I,'CUSBAL.ISM')     ; Open the indexed file
LOOP,  READS (1,REC,OUT)             ; Read the next record
       IF BAL.LT.20                  ; Balance less than $20?
           DELETE (1,NAME)           ; YES--Delete the record
       GOTO LOOP
OUT,   CLOSE 1                       ; Close the file
       STOP
```

3.9 DETACH

FUNCTION

DETACH disconnects the program from its associated terminal.

FORMAT

DETACH

RULES

- When DETACH is executed, the message DETACHING is displayed at the terminal and the program continues its execution.
- Attempting to perform I/O to the terminal suspends the program's execution until a terminal is reassigned to the detached program.
- DETACH has no effect on a program executing in a non-multi-tasking or detached environment.
- The terminal number associated with a detached program is -1, regardless of the number of the terminal from which the program detaches.

COMPILER ERROR CONDITIONS

- None

RUN-TIME ERROR CONDITIONS

- None

EXAMPLES

The following program allows the operator to enter the name of a file to print. Once the file name is entered, the terminal is no longer required by the program. Therefore, the DETACH statement is used so that another program may be run at the terminal.

```
RECORD
        FILE,   A20                     ; File name to print
        LINE,   A132                    ; Line to print
PROC
        OPEN (1,I,'TT:')                ; Open the terminal
        WRITES (1,'Enter file name')    ; Display prompt
        READS (1,FILE)                  ; Accept the file name
        CLOSE 1                         ; Close the terminal
        DETACH                          ; Release the terminal
;
; The remainder of the program runs detached
;
        OPEN (1,I,FILE)                 ; Open the print file
        OPEN (6,O:P,'LP:')              ; Open the printer
LOOP,   READS (1,LINE,EOF)              ; Read the next line
        WRITES (6,LINE)                 ; Print the line
        GOTO LOOP
EOF,    CLOSE 1                         ; Close the print line
        CLOSE 6                         ; Close the printer
        STOP
```

3.10 DISPLAY

FUNCTION

DISPLAY outputs characters to a device or file.

FORMAT

DISPLAY (*ch*, | *afield* / *aliteral* / *dexp* | [,...])

where:

ch is a decimal expression that evaluates to a channel number as specified in a previous OPEN statement.

afield
aliteral
dexp is an alpha field, alpha literal, record, or decimal expression which contains the characters to output.

RULES

- DISPLAY is used in O:P mode with a sequential file; in I and O modes with a terminal; and in O mode with a printer.
- DISPLAY uses the ASCII decimal character code (see Appendix A).
- If the data is alpha, the characters are output to the device as presented.
- If the data is decimal, the number is treated as a single character code.
- A number that exceeds the character code range (0 through 255) is converted by dividing the number by 256 and taking the remainder as a character code (e.g., 257 is interpreted as 001).
- A negative number produces unpredictable results.

COMPILER ERROR CONDITIONS

- None

RUN-TIME ERROR CONDITIONS

- 10 NT Illegal channel number
- 11 NT Channel not open
- 21 T Bad OPEN
- 104 NT Out of range

EXAMPLES

The following example outputs the message HELLO followed by a Carriage Return (decimal character code 13) and a Line Feed (decimal character code 10):

```
DISPLAY (1,'HELLO',13,10)
```

DISPLAY is especially useful for outputting terminal control sequences. The terminal user guide lists control code sequences for cursor positioning, clearing the screen, and many other operations. Assuming that channel 1 is associated with a VT100 terminal, the following example will position the cursor to line 3, column 5:

```
DISPLAY (1,27,'[3;5H')
```

DO—UNTIL

3.11 DO-UNTIL

FUNCTION

DO-UNTIL repetitively executes a statement until a condition is true.

FORMAT

DO *statement* UNTIL *condition*

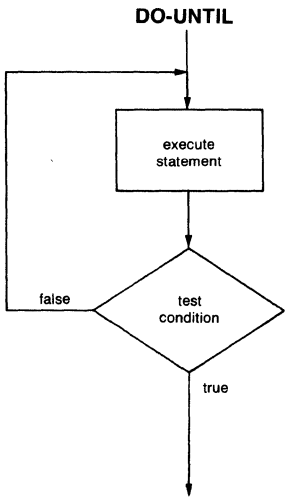
where:

statement is a DIBOL Procedure Division statement.

condition is a decimal expression.

RULES

- *Statement* is always executed at least once.
- The *condition* is evaluated following each execution of the statement.
- The *condition* is either true (non-zero) or false (zero).
- If the *condition* is false, the *statement* is executed again.
- UNTIL may be on a separate line.
- *Statement* may be on a separate line.



COMPILER ERROR CONDITIONS

- No UNTIL in DO-UNTIL statement
- Stack overflow

RUN-TIME ERROR CONDITIONS

- None

EXAMPLES

In the following example customer records (CUST) will be read until one is found with a balance (BAL) of less than $20:

```
DO
    READS (1,CUST,EOF)
UNTIL BAL.LT.20
```

The following program segment reads customer records (CUST) and creates a list of those customers with a balance over $100:

```
DO
    BEGIN
    READS (1,CUST,EOF)          ; Read a customer record
    IF BALANC.GT.100            ; Owe more than $100?
        BEGIN                   ; Yes--
        NAME=CUSNAM             ; Save customer name
        AMT=BALANC              ; Save the balance
        WRITES (6,PLINE)        ; Print name and balance
        END
    END
UNTIL CUSNAM.EQ.SPACES
```

3.12 FOR

FUNCTION

FOR repetitively executes a statement.

FORMAT

FOR *dfield* FROM *initial* THRU *final* [BY *step*] *statement*

where:

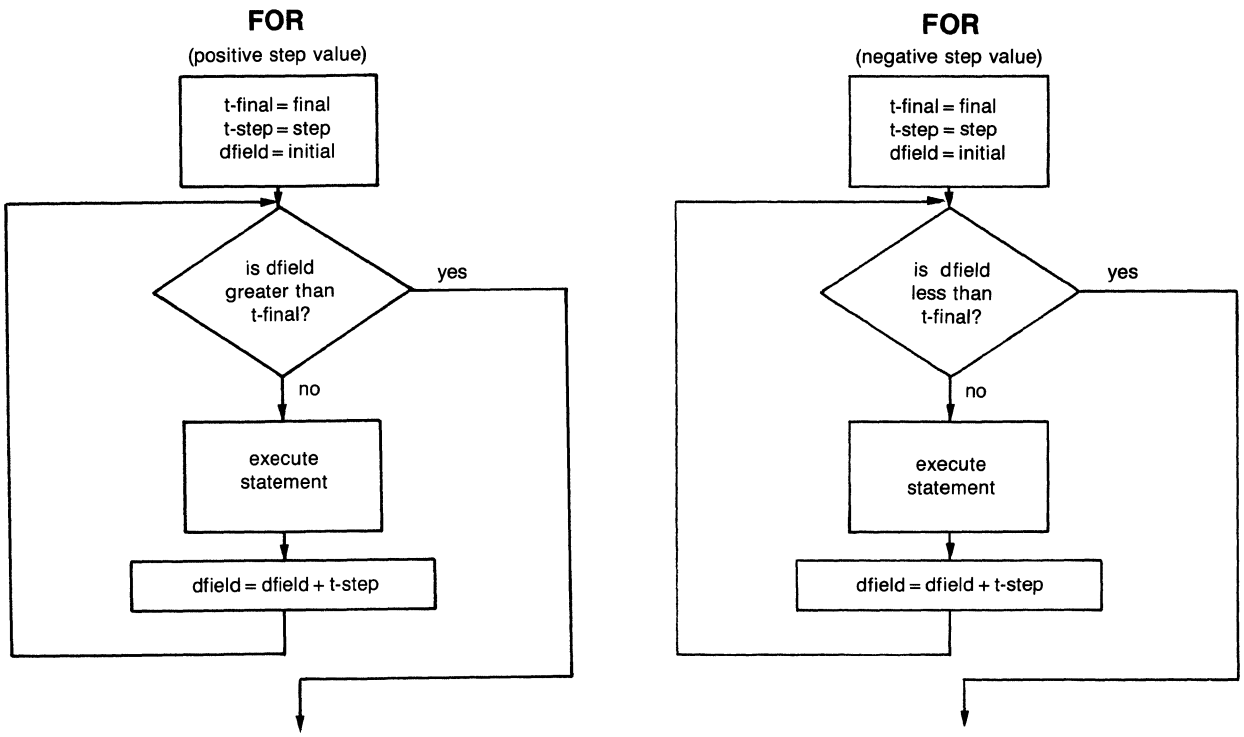
dfield is a decimal field to be incremented.

initial is a decimal expression which specifies the initial value to be assigned to *dfield*.

final is a decimal expression which specifies the final value for *dfield*.

step is a decimal expression which specifies the value to add to *dfield* each time through the loop.

statement is a DIBOL Procedure Division statement.



RULES

- Dfield cannot be a subscripted decimal field.
- FOR generates internal temporary fields to hold *step* (t—step) and *final* (t—final).
- T—final is a temporary field set to the final value and t—step is a temporary field set to the *step* value, prior to executing the loop.
- If no *step* value is specified, it is assumed to be 1.
- Prior to entering the loop, the sign of t—step is checked to insure that the *step* direction is correct. For the *step* direction to be correct, *dfield* must be less than or equal to t—final if t—step is positive, and *dfield* must be greater than or equal to t—final if t—step is negative. If the *step* direction is incorrect, the loop is not entered.
- Prior to each execution of *statement*, *dfield* is tested to determine if it has reached its limit. If *dfield* has not reached is limit, statement is executed.
- If *dfield* is not large enough to hold *final* plus the *step* value without truncation, an infinite loop may occur.
- T—step is added to *dfield* following each *statement* execution.
- If the loop is not executed, *dfield* is equal to the *initial* value.
- If the loop is exited normally, *dfield* will equal the previous value of *dfield* plus *step*.
- Modifying the *initial* value, *final* value, or *step* value in the FOR loop has no effect on the execution of the FOR loop.
- The *statement* may be on a separate line.

COMPILER ERROR CONDITIONS

- Invalid data type
- No FROM in FOR statement
- No THRU in FOR statement
- Stack overflow

RUN-TIME ERROR CONDITIONS

- 15 T Number too long (only on *initial*, *final*, or *step*)
- 87 T Argument missing

EXAMPLES

In the following example customer records 100 through 200 (inclusive) will be read and displayed:

```
FOR RECNO FROM 100 THRU 200
    BEGIN
    READ (1,CUST,RECNO)          ; Read customer record
    WRITES (8,CUST)              ; Display the record
    END
```

The FOR in the following program segment trims trailing spaces from a print line:

```
NEXT,   READS (1,LINE)                  ; Read line to print
        FOR I FROM 132 THRU 1 BY -1
            IF LINE(I,I).NE.SPACE       ; Is this a space?
                GOTO FOUND              ; No--found last character
        FORMS (6,1)                     ; Completely blank line
        GOTO NEXT
FOUND,  WRITES (6,LINE(1,I))            ; Output the line
        GOTO NEXT
```

In the following example the index field (I) is not large enough to hold the limit value plus the step (limit (99) + step (1) = 100). When the index reaches 99 it will be incremented to 100, but since the index field is only a 2 digit field, 00 will be stored in I. Therefore, the FOR statement will loop continuously.

```
RECORD  WORK
        I,      D2                  ; Loop index
PROC
        OPEN (1,O,'TT:')            ; Open terminal
        FOR I FROM 1 THRU 99
            WRITES (1,WORK)         ; Display index
        STOP
```

3.13 FORMS

FUNCTION

FORMS outputs device-dependent codes to effect forms control. These codes are normally used by printers.

FORMAT

FORMS (*ch*,*dexp*)

where:

ch is a decimal expression that evaluates to a channel number as specified in a previous OPEN statement.

dexp is a decimal expression that results in a printer control code.

RULES

- FORMS is used in O mode with a sequential file, in I and O modes with a terminal, and in O mode with a printer.
- Acceptable control code values are:

0	Transmits a Form Feed character (ASCII code 12).
1-255	Sends this many Line Feed characters (ASCII code 10) preceded by a Carriage Return character (ASCII code 13).
-1	Transmits a Vertical Tab character (ASCII code 11).
-3	Transmits a Carriage Return (ASCII code 13).

COMPILER ERROR CONDITIONS

- None

RUN-TIME ERROR CONDITIONS

- 10 NT Illegal channel number
- 11 NT Channel not open
- 22 T I-O error
- 104 NT Value out of range

EXAMPLES

The following FORMS statement will skip 3 lines:

```
FORMS (6,3)
```

The following FORMS statement will cause the printer to start a new page:

```
FORMS (6,0)
```

3.14 GOTO

FUNCTION

An unconditional GOTO transfers program control.

FORMAT

GOTO *label*

where:

label is the statement label where control is to be transferred.

RULES

- The statement may be written as GOTO or GO TO.

COMPILER ERROR CONDITIONS

- Label out of context block: <label name>

RUN-TIME ERROR CONDITIONS

- None

EXAMPLES

In the following example the GOTO will transfer control to the label NEXT:

```
NEXT,   READS (1,CUST,EOF)    ; Read a customer record
        NAME=CUSNAM           ; Save customer name
        AMT=BALANC            ; Save the balanc
        WRITES (6,PLINE)      ; Print name and balance
        GOTO NEXT
```

3.15 GOTO (COMPUTED)

FUNCTION

A computed GOTO transfers program control based on the evaluation of an expression.

FORMAT

GOTO (*label*[,...]),*dexp*

where:

label is one or more statement labels where control is to be transferred.

dexp is a decimal expression which determines to which statement label control is transferred.

RULES

- The statement may be written as GOTO or GO TO.
- Control is transferred to the statement identified by the first *label* if *dexp* is 1, to the statement identified by the second label if *dexp* is 2, etc.
- If *dexp* is negative, zero, or greater than the number of *labels*, control is transferred to the next logical statement in sequence.

COMPILER ERROR CONDITIONS

- Label out of context block: <label name>

RUN-TIME ERROR CONDITIONS

- None

EXAMPLES

In the following statement control will be transferred to the label LOOP if the value of KEY is 1; to the label LIST if the value of KEY is 2; and to the label TOTAL if the value of KEY is 3. If the value of KEY is less than 1 or greater than 3, control will be transferred to the statement following the GOTO.

```
GOTO (LOOP,LIST,TOTAL), KEY
```

3.16 IF

FUNCTION

IF executes a statement if a condition is true.

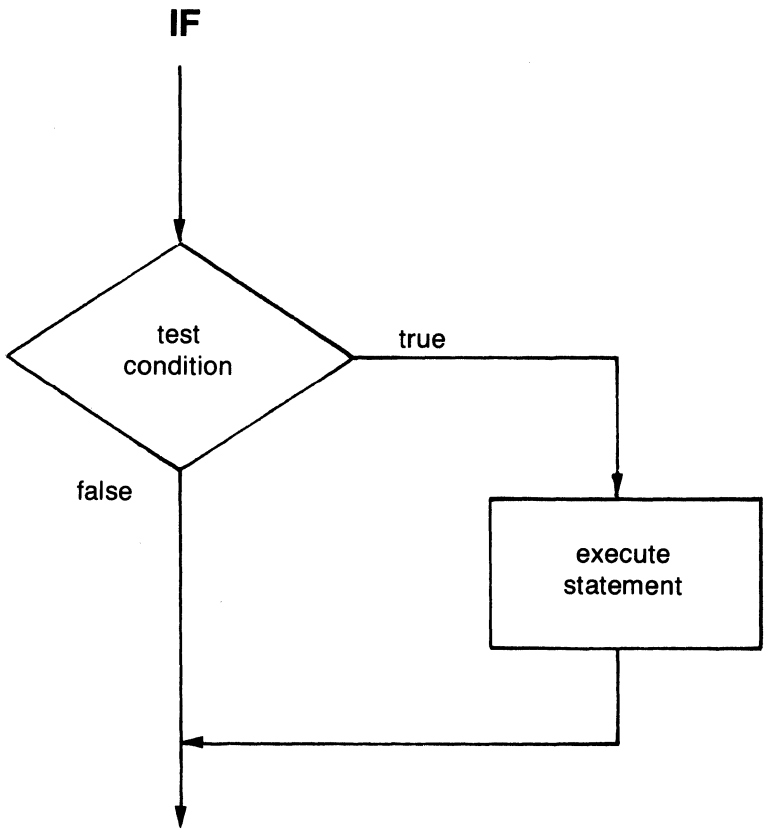
FORMAT

IF *condition statement*

where:

condition is a decimal expression which determines whether or not the statement is executed.

statement is a DIBOL Procedure Division statement.



RULES

- The *condition* is either true (non-zero) or false (zero).
- If the *condition* is true, *statement* is executed.
- If the *condition* is false, *statement* is not executed.
- *Statement* may be on a separate line.

COMPILER ERROR CONDITIONS

- None

RUN-TIME ERROR CONDITIONS

- None

EXAMPLES

In an alpha relational comparison, the operands are compared on a character basis from left to right according to their value in the collating sequence specified by their character codes (see Appendix A). The comparison is limited to the size of the shorter alpha field. For example, the following statement compares a 3 character alpha field to a 5 character alpha field. Since only the first 3 characters are compared, the result of the following statement is true:

```
IF 'ABC'.EQ.'ABCDE' STOP
```

The following IF statements are all valid:

```
The following IF statements are all valid:

    IF A.EQ.B GOTO LABEL3

    IF (SLOT.NE.202) READS (CH,RECNAM,EOF)

    IF SALES.LT.PROFIT+TAX-RENT
        STOP

    IF DONE STOP

    IF LNECTR.GE.MAXCTR
        BEGIN
        FORMS (6,0)
        WRITES (6,TITLE)
        CLEAR LNECTR
        END
```

IF—THEN—ELSE

3.17 IF-THEN-ELSE

FUNCTION

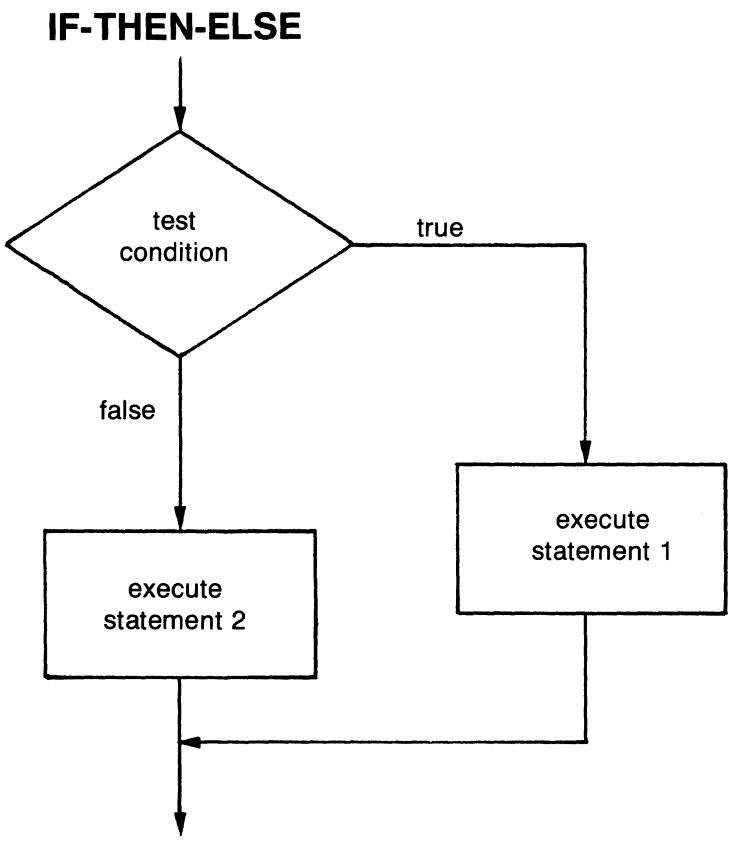
IF-THEN-ELSE executes 1 of 2 statements based on a condition.

FORMAT

IF *condition* THEN *statement1* ELSE *statement2*

where:

condition	is a decimal expression that determines which statement is executed.
statement1	is a DIBOL Procedure Division statement.
statement2	is a DIBOL Procedure Division statement.



RULES

- The *condition* is either true (non-zero) or false (zero).
- If the *condition* is true, *statement1* is executed.
- If the *condition* is false, *statement2* is executed.
- THEN may be on a separate line.
- ELSE may be on a separate line.
- *Statement1* may be on a separate line.
- *Statement2* may be on a separate line.

COMPILER ERROR CONDITIONS

- IF statement error - THEN without ELSE

RUN-TIME ERROR CONDITIONS

- None

EXAMPLES

In the following statement, the cost of an item is calculated differently, depending upon whether it is discountable:

```
IF DISCNT.EQ.'Y'          ; Is item discountable?
  THEN                    ; Yes--
    COST=PRICE-DIS+TAX    ; Get cost w/ discount
  ELSE
    COST=PRICE+TAX        ; Get cost w/o discount
```

The following example performs the same type of operation except the TAX and DIS calculations are performed within the IF statement:

```
IF DISCNT.EQ.'Y'            ; Is item discountable?
  THEN                      ; Yes--
    BEGIN
    DIS=PRICE/10            ; Calculate the discount
    TAX=(PRICE-DIS)*5/100   ; Calculate the tax
    COST=PRICE-DIS+TAX      ; Get cost w/ discount
    END
  ELSE
    BEGIN
    TAX=PRICE*5/100         ; Calculate the tax
    COST=PRICE+TAX          ; Get cost w/o the discount
    END
```

INCR

3.18 INCR

FUNCTION

INCR increases a decimal field by 1.

FORMAT

INCR *dfield*

where:

dfield is a decimal field to be incremented.

RULES

- The field to be incremented (*dfield*) can contain positive numbers, negative numbers, and spaces.
- Spaces are treated as zeros.
- If the size of the resulting value is larger than dfield, the leftmost digits that cause overflow are truncated.

COMPILER ERROR CONDITIONS

- None

RUN-TIME ERROR CONDITIONS

- 8 NT Writing into a literal

EXAMPLES

The following INCR statements are all valid (assuming that the fields being incremented are all decimal).

```
INCR CNTR

INCR A(3)

INCR C(H,6)

IF LNECTR.LT.MAXCTR INCR LNECTR
```

3.19 LOCASE

FUNCTION

LOCASE converts uppercase characters to corresponding lowercase characters.

FORMAT

LOCASE *afield*

where:

afield is an alpha field or record that contains the characters to be converted.

RULES

- The following non-alphabetic symbols are converted:

Uppercase		Lowercase	
@	(064)	`	(140)
[	(133)	{	(173)
\	(134)	|	(174)
]	(135)	}	(175)
^	(136)	~	(176)

- Other non-alphabetic characters are unaffected.

COMPILER ERROR CONDITIONS

- None

RUN-TIME ERROR CONDITIONS

- 8 NT Writing into a literal

EXAMPLES

In the following example the first LOCASE statement changes the characters 'THIS IS A TEST' to lowercase. After the first LOCASE statement is executed, the contents of REC are 'This is a test [OF LOCASE]' to lowercase. After the second LOCASE statement is executed, the contents of REC are 'this is a test { of locase } '.

```
RECORD  REC
        A,      A14, 'THIS IS A TEST'
        B,      A12, ' [OF LOCASE]'
PROC
        LSCASE A(2,14)
        LOCASE REC
        STOP
```

3.20 LPQUE

FUNCTION

LPQUE queues a file to be printed by the printer spooler.

FORMAT

LPQUE (*filespec*[,LPNUM:*dexp*][,COPIES:*dexp*]
[,FORM: | *afield* / *aliteral* |][,DELETE])

where:

filespec is an alpha field, alpha literal, or record which contains the file specification of the file to be printed.

LPNUM:*dexp*
is a decimal expression that specifies the printer.

COPIES:*dexp*
is a decimal expression which specifies the number of copies to print.

FORM:*afield*
aliteral
is an alpha field, alpha literal, or record which specifies the type or name of the form to be inserted into the printer before the file is printed.

DELETE deletes the file after all copies have been printed.

RULES

- Optional qualifiers prefaced by a keyword can occur in any order.
- LPQUE sends a request to the printer spooler to print the file.
- Multiple LPQUE statements cause the print requests to be queued.
- If no printer identification is specified, the system's default printer(s) are used.
- If no copy count is specified, or if it is less than 1, it is assumed to be 1.
- If a form is specified, a system specific forms request is issued.

COMPILER ERROR CONDITIONS

- Invalid LPQUE keyword

RUN-TIME ERROR CONDITIONS

- 10 NT Illegal channel number
- 11 NT Channel not open
- 18 T File not found
- 120 T Queue not available or invalid queue name

EXAMPLES

In the following example the LPQUE statement requests the printing of one copy (NBR = 1) of the file CHECK.LIS. Before printing begins, the form CHECKS should be placed in the printer.

```
RECORD
        NBR,      D2, 01
        FILE,     A9, 'CHECK.LIS'
PROC
        LPQUE (FILE,COPIES:NBR,FORM:'CHECKS')
        STOP
```

3.21 OFFERROR

FUNCTION

OFFERROR disables trapping of run-time errors.

FORMAT

OFFERROR

RULES

- This statement may be written as OFFERROR or OFF ERROR.
- When OFFERROR is executed, run-time errors normally detected by the ONERROR statement are treated as non-trappable.

COMPILER ERROR CONDITIONS

- None

RUN-TIME ERROR CONDITIONS

- None

EXAMPLES

In the following example the ONERROR statement is used to trap the **Division by 0** error and the OFFERROR is used to disable error trapping after the division is performed:

```
ONERROR DIV0                ; Check for Division by 0
C=A/B
OFFERROR                    ; Turn off error check
```

3.22 ONERROR

FUNCTION

ONERROR enables trapping of run-time errors which would otherwise cause program termination.

FORMAT

ONERROR *label*

where:

label is a statement label where control is to be transferred when an error occurs.

RULES

- This statement may be written as ONERROR or ON ERROR.
- ONERROR remains in effect until one of the following occurs:
  - An ONERROR is executed which specifies a different *label*.
  - An XCALL is executed. ONERROR is suspended until control returns from the external subroutine.
  - An OFFERROR is executed.
  - The program terminates.
- The error detected by ONERROR may be determined either by using the ERROR external subroutine, or by knowing the nature of the statements executed after ONERROR was executed.
- ONERROR is disabled by OFFERROR.

COMPILER ERROR CONDITIONS

- Label out of context block: <label name>

RUN-TIME ERROR CONDITIONS

- None

EXAMPLES

In the following example the ONERROR statement is used to trap errors. If a trappable error occurs after the ONERROR has been executed, control will be transferred to the label IOERR:

```
        ONERROR IOERR
NEXT,   READS (1,CUST,EOF)      ; Read a customer record
        NAME=CUSNAM             ; Save customer name
        AMT=BALANC              ; Save the balance
        WRITES (6,PLINE)        ; Print name and balance
        GOTO NEXT
```

3.23 OPEN

FUNCTION

OPEN associates a channel number with a device or with a file on a device.

FORMAT

OPEN (*ch*,*mode*[:*submode*],*filespec*[,ALLOC:*dexp*][,BKTSIZ:*dexp*]

[,BLKSIZ:*dexp*][,BUFSIZ:*dexp*][,RECSIZ:*dexp*])

where:

ch is a decimal expression that evaluates to a channel number.

mode designates the data transfer method (Input, Output, or Update).

submode further defines, qualifies, or restricts mode.

filespec is an alpha field, alpha literal, or record that contains the file specification.

ALLOC:*dexp*
is a decimal expression that specifies the initial file allocation.

BKTSIZ:*dexp*
is a decimal expression that specifies the bucketsize in blocks.

BLKSIZ:*dexp*
is a decimal expression that specifies the block size (bytes) of magnetic tape.

BUFSIZ:*dexp*
is a decimal expression that specifies the size of the transfer buffer in blocks for this channel.

RECSIZ:*dexp*
is a decimal expression that specifies the length (bytes) of the records in the file.

GENERAL RULES

- A unique OPEN statement must be executed for each unique combination of device, file, and mode of operation.
- OPEN must be executed prior to any I/O operation and remains in effect until a corresponding CLOSE is executed.
- The maximum number of channels opened simultaneously is system dependent.
- The maximum channel number is equal to the maximum number of channels which can be opened simultaneously.
- Optional qualifiers prefaced by a keyword can occur in any order.
- The transfer of program control to an external subroutine does not affect the status of a channel.

RULES FOR MODE

- OPEN uses three data access methods: sequential, relative, and indexed.
- If a file is being opened, the modes of operation and file I/O statements are:

  INPUT (I) used to obtain input from an existing sequential, relative, or indexed file. Input mode is a read only mode.

  OUTPUT (O) used to create a file.

  UPDATE (U) used for input and output from an existing relative or indexed file.

- If a terminal is being opened, only the Input and Output modes of operation are used.

RULES FOR SUBMODE

- *Submodes* are Sequential (S), Print (P), Relative (R) or Indexed (I).
- Sequential submode is used with O mode and indicates that the file being created is a sequential file. Sequential submode is assumed if no *submode* is specified with O mode.
- Print submode is used with O mode and indicates that the file being created is a print file.
- Relative submode is used with the O mode and indicates that the file being created is a relative file. O:R is required when creating an RMS relative file.
- Indexed submode is used with I and U modes and indicates that the file being opened is an indexed file. All file volumes must be on-line simultaneously. SI is equivalent to I:I and SU is equivalent to U:I.

RULES FOR ALLOC

- ALLOC overrides any filesize specified with the *filespec*. The value specified is system dependent.
- ALLOC is used in O mode. It is ignored for other modes.

RULES FOR BKTSIZ

- BKTSIZ is used when creating an RMS relative file. Any other use of BKTSIZ is ignored.
- The value is system dependent.

RULES FOR BLKSIZ

- BLKSIZ is used when creating a file on magtape. Any other use of BLKSIZ is ignored.

RULES FOR BUFSIZ

- BUFSIZ overrides the buffer size designated by PROC for this OPEN.
- The value must be between 1 and 15.

RULES FOR RECSIZ

- RECSIZ is required when creating an RMS relative file.
- RECSIZ implies the records are fixed length.
- The range for RECSIZ is 1 to 16,383.

The following chart shows which statements are legal for a file organization, *mode*, and character device.

	File Organization							Character Device		
	Sequential		Relative			Indexed		Terminal		Printer
	I	O	I	O	U	I	U	I	O	O
READS	X		X	X	X	X	X	X	X	
WRITES		X		X	X			X	X	X
READ			X	X	X	X	X			
WRITE				X	X		X			
DELETE							X			
STORE							X			
ACCEPT								X	X	
DISPLAY		*						X	X	*
FORMS		X						X	X	X

* O:P ONLY

COMPILER ERROR CONDITIONS

- Invalid OPEN keyword: <keyword name>
- Invalid OPEN mode
- Invalid OPEN submode

RUN-TIME ERROR CONDITIONS

- 9 T Not enough memory
- 10 NT Illegal channel number
- 11 NT Channel not open
- 12 T Input from write-only device
- 16 NT DIBOL channel in use
- 17 T Bad file specification
- 18 T File not found
- 19 T Device handler not available
- 22 T I-O error
- 24 T No space for file
- 32 T Cannot supersede existing file
- 37 T Device in use
- 38 T File in use
- 39 T Output to read-only device
- 43 NT ?M-Dir IO error
- 56 T Not ISAM file
- 62 T Protection violation
- 87 T Argument missing
- 103 T Invalid file organization
- 107 T Device not ready
- 108 T Invalid OPEN mode value
- 112 T Error during file open
- 113 T Invalid ALLOC value in OPEN
- 114 T Invalid BKTSIZ value in OPEN
- 115 T Invalid BLKSIZ value in OPEN
- 116 T Invalid BUFSIZ value in OPEN
- 117 T Invalid RECSIZ value in OPEN

EXAMPLES

The following statement creates a new sequential file named RENEW.DDF and associates it with channel 5:

```
OPEN (5,O,'RENEW.DDF')
```

The following statement creates a new relative file named ARMAS.DDF and associates it with channel 2. All the records in the file will be 100 characters in length:

```
OPEN (2,O:R.'ARMAS.DDF',RECSIZ:100)
```

The following statement opens the terminal for both input and output and associates the terminal with channel 15:

```
OPEN (15,O,'TT:')
```

The following statement opens the relative file ARMAS.DDF for modification using channel 3. It also specifies an internal buffer size of 2 blocks. This buffer size overrides the size specified by PROC for this OPEN only:

```
OPEN (3,U,'ARMAS.DDF',BUFSIZ:2)
```

The following statement creates a new sequential file named AR.LIS and associates it with channel 5. Since the new file will eventually be printed, it is created with the P submode:

```
OPEN (5,O:P,'AR.LIS')
```

3.24 PROC-END

FUNCTION

PROC-END identifies the beginning and ending of the Procedure Division.

FORMAT

```
PROC [(dliteral)]
     statement
     .
     .
     .
[END]
```

where:

dliteral is a decimal literal that specifies the size of the I/O buffer allocated to every opened sequential file.

statement is a DIBOL Procedure Division statement.

RULES

- Only 1 PROC may be used in a program.
- PROC cannot have a statement label.
- The dliteral value is ignored in external subroutines.
- The dliteral value must be between 1 and 15 blocks (1 block = 512 bytes).
- If no *dliteral* is specified, it is assumed to be 1.
- The *dliteral* is meaningful only with sequential files. For relative and indexed files, the buffer sizes are determined by the bucket size specified when defining the file.
- The BUFSIZ used in an OPEN statement overrides the buffer size established by PROC just for that OPEN.
- END is not mandatory.
- Termination of the PROC-END block, either explicitly (END) or implicitly (end-of-file) causes an implicit STOP statement to be inserted into the compiled program.
- END can be used anywhere in a program except as the statement to be executed in an IF, IF-THEN-ELSE, FOR, WHILE, DO-UNTIL, or USING statement.
- Source lines following END are not compiled.

COMPILER ERROR CONDITIONS

- Invalid PROC statement
- No PROC statement

RUN-TIME ERROR CONDITIONS

- None

EXAMPLES

The following example copies 100 character records from INFILE.DDF to OUTFIL.DDF. The PROC statement specifies that a 3 block buffer is to be allocated for each opened channel. After the 2 OPEN statements are executed, 6 blocks will have been allocated for internal buffers.

```
RECORD  REC
        ,     A100
RECORD
        EOF,  D1, 0
PROC 3
        OPEN (1,I,'INFILE.DDF')      ; Open input file
        OPEN (2,O,'OUTFIL.DDF')      ; Open output file
        DO
            BEGIN
            READS (1,REC,EOF)        ; Read input record
            WRITES (2,REC)           ; Write output record
            END
        UNTIL EOF
 EOF,
        CLOSE 1                      ; Close input file
        CLOSE 2                      ; Close output file
        STOP
 END
```

3.25 READ (RELATIVE FILE)

FUNCTION

READ inputs a record from a relative file.

FORMAT

READ (*ch*,*record*,*dexp*)

where:

ch is a decimal expression that evaluates to a channel number as specified in a previous OPEN statement.

record is an alpha field or record which will contain the data.

dexp is a decimal expression that specifies the sequence number of the record to be read.

RULES

- READ is used in I and U modes.
- *Dexp* must be between 1 and the total number of records in the file.
- The record is read into *record* according to the rules for moving alpha data (see section 3.2.1).
- If the record is larger than *record*, a **Line too long** error is generated.
- When READ is executed in U mode, the blocks which contain the record are locked; other records that lie wholly or partially within these blocks are also locked. The lock remains in effect until one of the following occurs:
    - A WRITE or WRITES using the channel is executed.
    - A READ or READS using the channel is executed.
    - An UNLOCK using the channel is executed.
    - A CLOSE using the channel is executed.
    - The program terminates.

COMPILER ERROR CONDITIONS

- Invalid data type

RUN-TIME ERROR CONDITIONS

- 1 T End of file
- 8 NT Writing into a literal
- 10 NT Illegal channel number
- 11 NT Channel not open
- 21 T Bad OPEN
- 22 T I-O error
- 23 T Line too long
- 28 T Illegal record number
- 40 T Record locked
- 64 T Record not found
- 84 T Illegal block I/O record size

EXAMPLES

The following statement reads the 88th record of the relative file associated with channel 5 and places the record in the variable REX:

```
READ (5,REX,88)
```

The following statement reads the record specified by the value stored in the variable COUNT from the relative file associated with channel 6 and places the record in the variable BLT:

```
READ (6,BLT,COUNT)
```

3.26 READ (INDEXED FILE)

FUNCTION

READ inputs a record from an indexed file.

FORMAT

READ (*ch*,*record*,*keyfld*)

where:

ch is a decimal expression that evaluates to a channel number as specified in a previous OPEN statement.

record is an alpha field or record which will contain the data.

keyfld is an alpha field or record which identifies the record to be read.

RULES

- READ is used in I and U modes.
- If *keyfld* is less than the size of the key field defined for the indexed file, it is assumed to be a partial key. The system returns the first record whose initial characters match the specified key.
- If duplicate keys exist, READ retrieves the first occurrence of the key. READS is used to retrieve each additional occurrence of the key.
- *Keyfld* must occupy the same position within the record as does a key field defined for the indexed file. If it is a multi key file, any key may be used.
- If a record containing the specified key is not found, the record with the next higher key is returned and a **Key not same** error is generated.
- The record is read into *record* according to the rules for moving alpha data (see section 3.2.1).
- If the record is larger than *record*, a **Line too long** error is error generated.
- When a READ is executed in U mode, the blocks which contain the record are locked; other records that lie wholly or partially within these blocks are also locked. The lock remains in effect until one of the following occurs:
    - A WRITE using the channel is executed.
    - A READ or READS using the *ch*annel is executed.
    - A STORE using the *ch*annel is executed.
    - A DELETE using the *ch*annel is executed.

- An UNLOCK using the *ch*annel is executed.
- A CLOSE using the *ch*annel is executed.
- The program terminates.

COMPILER ERROR CONDITIONS

- Invalid data type

RUN-TIME ERROR CONDITIONS

- 1 T End of file
- 8 NT Writing into a literal
- 10 NT Illegal channel number
- 11 NT Channel not open
- 21 T Bad OPEN
- 22 T I-O error
- 23 T Line too long
- 40 T Record locked
- 52 T Illegal key
- 53 T Key not same

EXAMPLES

Assuming that the indexed file has been defined with a key length of 5 characters and a key position of 16 and the Data Division contains:

```
RECORD    ADDR
          ,       A5
          ,       D10
          KEY,    A5, 'SMITH'
          ,       D20
```

then the following statement will return the record with the key SMITH from the indexed file opened on channel 1. The READ will place that record in ADDR. If more than one SMITH record exists, the first one is obtained and the remaining SMITH records can be read using the READS statement. If SMITH does not exist, the next higher keyed record will be retrieved, and a **Key not same** error will be generated. This error can be trapped by an ONERROR statement.

```
READ (1,ADDR,KEY)
```

3.27 READS

FUNCTION

READS inputs the next available record in sequence from a file.

FORMAT

READS (*ch*,*record*[,*label*])

where:

ch is a decimal expression that evaluates to a channel number as specified in a previous OPEN statement.

record is an alpha field or record which will contain the data.

label is a statement label where control is to be transferred when the logical end-of-file is detected.

GENERAL RULES

- READS is used in I mode with a sequential file; in I and U modes with a relative file and with an indexed file; and in I and O modes with a terminal.
- The record is read into *record* according to the rules for moving alpha data (see section 3.2.1).
- If the record is larger than *record*, a **Line too long** error is generated.
- When a READS is executed in U mode, record locking occurs in the same manner as when a READ is executed.

RULES FOR READS FROM AN INDEXED FILE

- When an indexed file is opened and the first I/O statement for that file is a READS, the record with the lowest primary key value is returned.

RULES FOR READS FROM A TERMINAL

- READS from a terminal may be affected by the FLAGS subroutine.
- All terminating characters except ESCAPE position the cursor or carriage at the beginning of the next line. ESCAPE terminates input but does not move the cursor or carriage.
- When *record* is full, additional characters are ignored and the terminal alarm sounds for each additional character typed.

COMPILER ERROR CONDITIONS

- Invalid data type

RUN-TIME ERROR CONDITIONS

- 1 T End of file
- 8 NT Writing into a literal
- 10 NT Illegal channel number
- 11 NT Channel not open
- 21 T Bad OPEN
- 22 T I-O error
- 23 T Line too long
- 40 T Record locked

EXAMPLES

The following statement transfers a record from the file associated with channel 3 to the variable INV. If the end of the file is reached, control branches to a statement labeled END.

```
READS (3,INV,END)
```

The next example is the same as the previous one, except that if the end of file is reached, an **End of file** error will be generated since no end of file label was specified. This error can be trapped by an ONERROR statement.

```
READS (3,INV)
```

3.28 RECV

FUNCTION

RECV accepts a message which was sent by another program.

FORMAT

RECV (*message*,*label*[,*size*])

where:

message is an alpha field or record which will contain the message.

label is a statement label where control is to be transferred if no message is pending.

size is a decimal field which will contain the size of the message received.

RULES

- The message is moved into *message* according to the rules for moving alpha data (see section 3.2.1).
- The message size is moved into *size* according to the rules for moving decimal data (see section 3.2.2).

COMPILER ERROR CONDITIONS

- Invalid data type
- Label out of context block: <label name>

RUN-TIME ERROR CONDITIONS

- 8 T Writing into a literal
- 23 T Line too long
- 118 T Unable to open message manager mailbox

EXAMPLES

The following program segments show how one program might pass the name of a data file to another program using the SEND and RECV statements. The PAYROL program sends the file name (TFIL.DDF) to the program BAT. The RECV statement in BAT accepts the file name. If the RECV statement is executed prior to the message having been sent, control is transferred to the statement labeled LOOP. At LOOP the program delays for 10 seconds and then attempts to receive the message again.

Program PAYROL

```
RECORD
        MSG,    A8, 'TFIL.DDF'
        PRONAM, A3, 'BAT"
PROC
         .
         .
         .
        SEND (MSG,PRONAM)               ; Send file name
         .
         .
         .
        STOP
```

Program BAT

```
RECORD
        FILE,     A9
PROC
GETM,   RECV (FILE,LOOP)                ; Receive file name
         .
         .
         .
        STOP
LOOP,   SLEEP 10                        ; Wait for 10 seconds
        GOTO GETM
```

3.29 RETURN

FUNCTION

RETURN transfers program control to the statement logically following the most recently executed CALL or XCALL statement.

FORMAT

RETURN

RULES

- RETURN must be placed at the logical exit of each internal and external subroutine.

COMPILER ERROR CONDITIONS

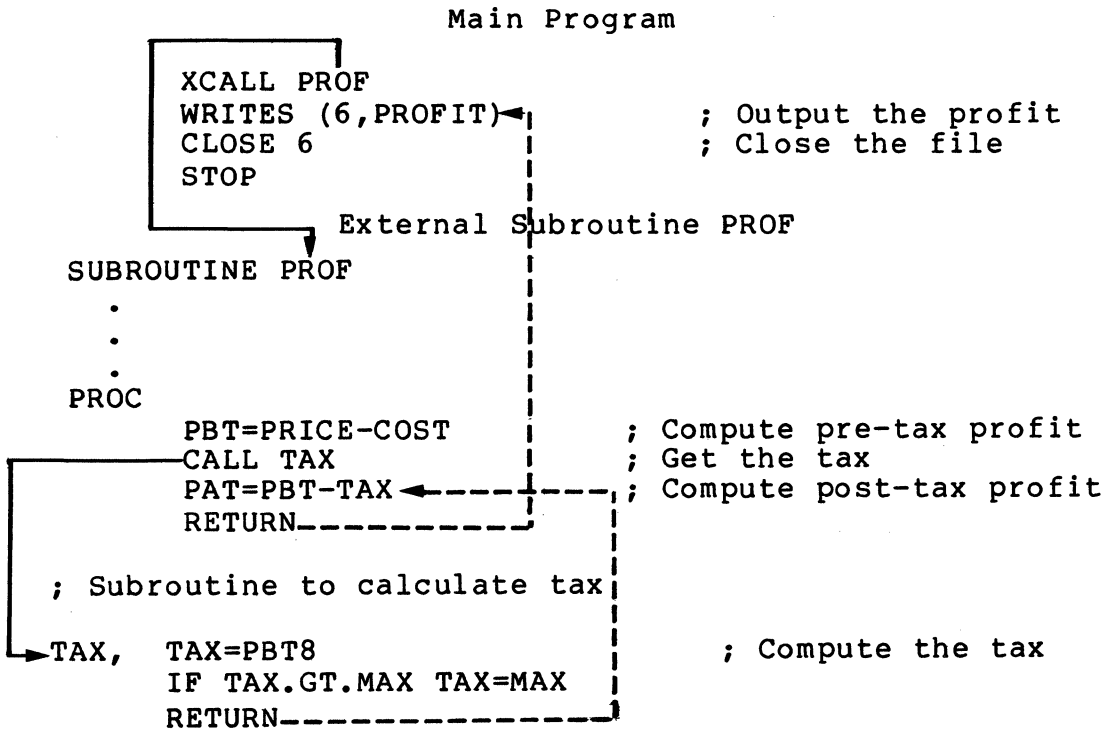
- None

RUN-TIME ERROR CONDITIONS

- 2 NT RETURN but no CALL or XCALL

EXAMPLES

The following example shows how program control branches when using external and internal subroutines. The solid lines show the control path upon execution of CALL and XCALL statements and the broken lines show the control path upon execution of RETURN statements:



3.30 SEND

FUNCTION

SEND transmits a message to another program.

FORMAT

SEND (*message*,*program*[,*terminal*])

where:

message is an alpha field, alpha literal, or record which contains the message to be sent.

program is an alpha field, alpha literal, or record which contains the name of the program that is to receive the message.

terminal is a decimal expression which specifies the terminal number associated with the receiving program.

RULES

- *Message* is stored for a subsequent RECV.
- Multiple messages can be stored.
- FIFO (First-In-First-Out) message processing ensures that the first message sent to a program is the first to be received by that program.
- Messages may be sent from one program in a chain to a program further along the chain.
- System resources (memory, disk, ...) can affect sending a message.
- Programs with the same name can be identified by specifying the terminal to which the program is attached.
- If the terminal number is not used, the first program with the correct name that executes a RECV will receive the message.
- Messages may be sent to a detached program by specifying a terminal number of -1.
- If two or more detached programs have the same name, the first to execute a RECV will receive the message.

RUN-TIME ERROR CONDITIONS

- 9 T Not enough memory
- 118 T Unable to open message manager mailbox

EXAMPLES

The following statement sends a message to the program CNCRNT which may be running concurrently or at some later time on any terminal or detached:

```
SEND (MSG,'CNCRNT')
```

The following example sends a message to the program NEXT which is designated as running on the same terminal as the current program:

```
RECORD
        TNUM,   D3                  ; Terminal number
          .
          .
          .
PROC
        XCALL TNMBR (TNUM)          ; Get terminal number
        SEND (MSG,'NEXT',TNUM)      ; Send message
        STOP 'NEXT'
```

3.31 SLEEP

FUNCTION

SLEEP suspends program execution for a specified period of time.

FORMAT

SLEEP *seconds*

where:

seconds is a decimal expression that specifies the number of seconds to suspend program execution.

RULES

- Program execution resumes only when the specified time has elapsed
- Specifying a negative number of seconds will generate a **Value out of range** error.

COMPILER ERROR CONDITIONS

- Invalid data type

RUN-TIME ERROR CONDITIONS

- 104 NT Value out of range

EXAMPLES

The following program sounds the terminal's alarm once every minute:

```
PROC
        OPEN (3,O,'TT:')        ; Open terminal
BEEP,   DISPLAY (3,7)           ; Sound terminal alarm
        SLEEP 60                ; Delay for 60 seconds
        GOTO BEEP
```

3.32 STOP

FUNCTION

STOP terminates program execution.

FORMAT

STOP [*filespec*]

where:

filespec is an alpha field, alpha literal, or record which contains a program or command file specification.

RULES

- STOP can appear as often as needed in a program, but the first STOP executed terminates the program.
- If *filespec* is used, the system automatically chains to the specified program.
- If *filespec* begins with a '@', it indicates that the filespec is for a command file.
- If no *filespec* is specified for a detached program, the program is logged out.
- When a detached program stops, no terminal output is generated (traceback, STOP message, etc.).
- If a *filespec* is specified by a detached program, the new program also runs detached.

COMPILER ERROR CONDITIONS

- Invalid data type

RUN-TIME ERROR CONDITIONS

- 18 T File not found

EXAMPLES

The following statement will stop execution of the current program and begin execution of the PROG2 program:

```
STOP 'PROG2'
```

The following statement will stop execution of the current program and begin execution of the CMDFIL command file:

```
STOP '@CMDFIL'
```

3.33 STORE

FUNCTION

STORE adds a record to an indexed file.

FORMAT

STORE (*ch*,*record*,*keyfld*)

where:

ch is a decimal expression that evaluates to a channel number as specified in a previous OPEN statement.

record is an alpha field or record which contains the data to be stored.

keyfld is an alpha field or record which identifies the record in which the data will be stored.

RULES

- STORE is used in U:I mode.
- *Keyfld* must occupy the same position within the record as does a key field defined for the indexed file. If it is a multi-key file, any key may be used.
- STORE locks the record which is being stored. The record is unlocked when STORE is completed.
- If duplicate key values are not allowed and a record with the specified key already exists, a **No duplicates** error is generated.

COMPILER ERROR CONDITIONS

- Invalid data type

RUN-TIME ERROR CONDITIONS

- 10 NT Illegal channel number
- 11 NT Channel not open
- 21 T Bad OPEN
- 25 T Output file full
- 26 T Field or record too long
- 40 T Record locked
- 52 T Illegal key
- 54 T No duplicates

EXAMPLES

The following example illustrates the use of STORE. On each iteration of the loop, this program stores an employee record with the key value contained in the field BADGE.

```
RECORD  NEWREC                          ; Employee record
        NAME,   A20                     ; Employee name
        BADGE,  A5                      ; Employee badge number
RECORD
        DONE,   A1
PROC
        OPEN (1,O,'TT:')                ; Open terminal
        OPEN (2,U:I,'EMPFIL')           ; Open employee file
        DO
            BEGIN
            WRITES (1,'Name?')          ; Prompt for name
            READS (1,NAME)              ; Get employee name
            WRITES (1,'Badge?')         ; Prompt for badge number
            READS (1,BADGE)             ; Get badge number
            STORE (2,NEWREC,BADGE)      ; Create employee record
            WRITES (1,'Done?')          ; Ask if finished
            READS (1,DONE)              ; Get response
            END
        UNTIL DONE.EQ.'Y'
        CLOSE 1                         ; Close terminal
        CLOSE 2                         ; Close employee file
        STOP
```

UNLOCK

3.34 UNLOCK

FUNCTION

UNLOCK clears the lock condition on a specified channel.

FORMAT

UNLOCK *ch*

where:

ch is a decimal expression that evaluates to a channel number as specified in a previous OPEN statement.

RULES

- Records in the locked blocks will become available for access by other programs.
- The specified channel is the one associated with the file containing the locked blocks.
- UNLOCK is ignored if no records are locked on the channel or if the channel is not opened.

COMPILER ERROR CONDITIONS

- Invalid data type

RUN-TIME ERROR CONDITIONS

- 10 NT Illegal channel number

EXAMPLES

The following program will delete employee records from an indexed file. On each iteration of the loop, this program prompts for an employee badge number (the key field for the indexed record), reads the employee record (which locks the record), displays the associated name, and asks if the employee record should be deleted. If the record is not to be deleted, the UNLOCK statement makes the record available for other programs to read.

```
RECORD  EMPREC                          ; Employee record
        NAME,   A20                     ; Employee name
        BADGE,  A5                      ; Employee badge number
RECORD
        DELETE, A1
        DONE,   A1
PROC
        OPEN (1,O,'TT:')                ; Open terminal
        OPEN (2,U:I,'EMPFIL')           ; Open employee file
        DO
            BEGIN
            WRITES (1,'Badge?')         ; Prompt for badge number
            READS (1,BADGE)             ; Get badge number
            READ (2,EMPREC,BADGE)       ; Read employee record
            WRITES (1,NAME)             ; Display employee name
            WRITES (1,'Delete?')        ; Prompt for deletion
            WRITES (1,DELETE)           ; Get response
            IF DELETE.EQ.'Y'            ; Delete the record
              THEN                      ; Yes--
                DELETE (2,BADGE)        ; Delete the record
              ELSE
                UNLOCK 2                ; Unlock the record
            WRITES (1,'Done?')          ; Ask if finished
            READS (1,DONE)              ; Get response
            END
        UNTIL DONE.EQ.'Y'
        CLOSE 1                         ; Close terminal
        CLOSE 2                         ; Close employee file
        STOP
```

3.35 UPCASE

FUNCTION

UPCASE converts lowercase characters to corresponding uppercase characters.

FORMAT

UPCASE *afield*

where:

afield is an alpha field or record which contains the characters to be converted.

RULES

- The following non-alphabetic characters are converted by UPCASE:

Lowercase	Uppercase
\`	@
{	[
|	\
}	]
~	^

- Other non-alphabetic characters are unaffected.

COMPILER ERROR CONDITIONS

- Invalid data type

RUN-TIME ERROR CONDITIONS

- 8 NT Writing into a literal

EXAMPLES

In the following example the first UPCASE statement changes the first character in field A. After the first UPCASE statement is executed the contents of REC are 'This is a test {of upcase} '. The second UPCASE statement changes the characters 'This is a test {of upcase} ' to uppercase. After the second UPCASE statement is executed the contents of REC are 'THIS IS A TEST [OF UPCASE]'.

```
RECORD  REC
        A,          A14, 'this is a test'
        B,          A12, ' {of upcase}'
PROC
        UPCASE A(1,1)
        UPCASE REC
        STOP
```

The following example allows an operator to answer 'YES' without regard to uppercase or lowercase. The operator could type any of the following: yes, yeS, yEs, yES, Yes, YeS, YEs, or YES.

```
RECORD
        DONE, A3
PROC
          .
          .
          .
        WRITES (1,'Done?')          ; Prompt user
        READS (1,DONE)              ; Get response
        UPCASE DONE                 ; Make it uppercase
        IF DONE.EQ.'YES'            ; Did operator type 'YES'?
            STOP                    ; Yes--
          .
          .
          .
```

3.36 USING

FUNCTION

USING conditionally executes one statement from a list of statements based on the evaluation of an expression.

FORMAT

```
USING selection—value SELECT
    ([mexp[,...]]), statement
      .
      .
      .
    ENDUSING
```

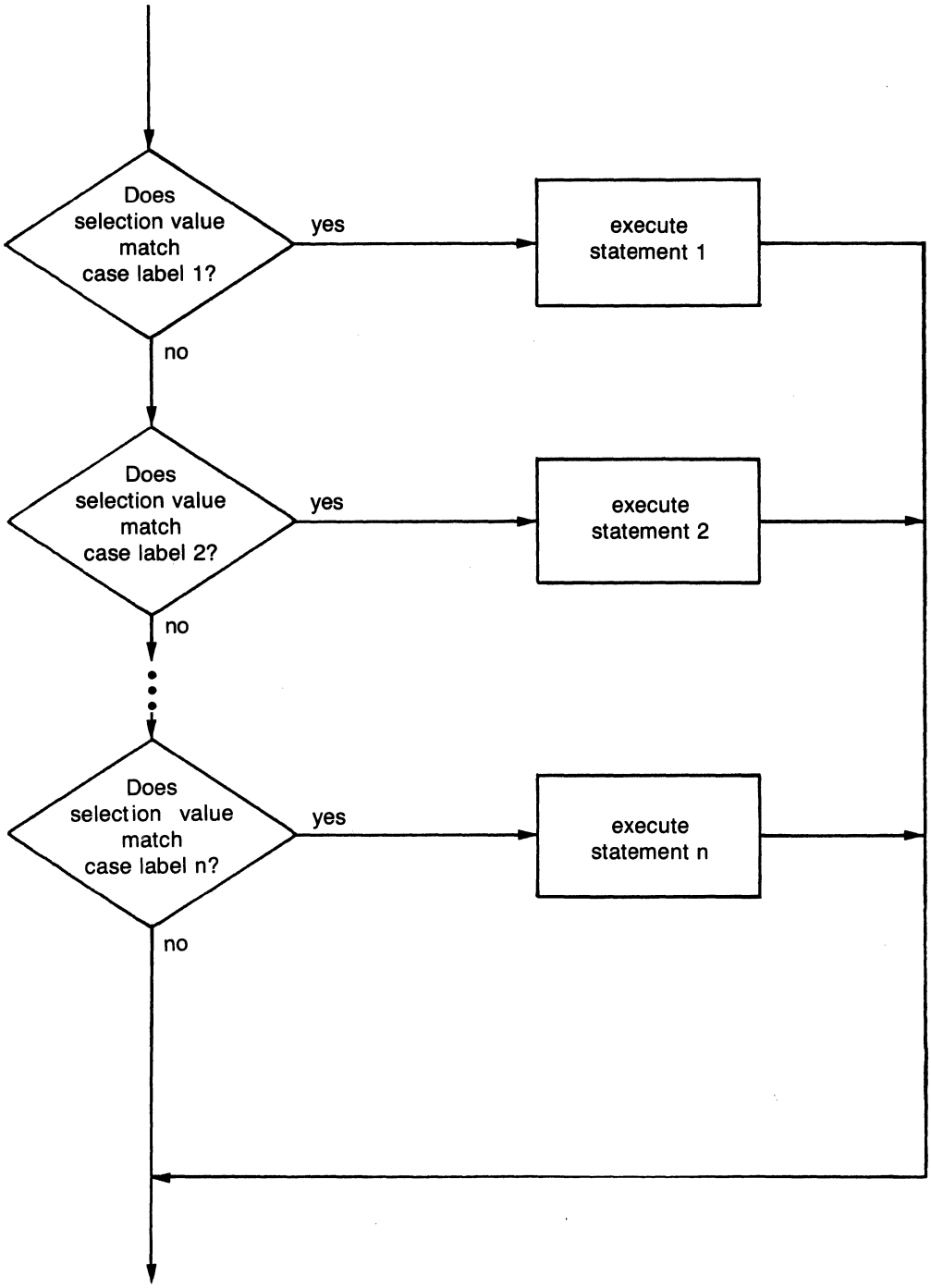
where:

selection—value
is an alpha field, alpha literal, decimal expression, or record.

mexp is one or more match expressions in the following format:

```
|value                  |
|value THRU value       |
```

statement is a DIBOL Procedure Division statement.


USING
Does selection value match case label 1?
yes
execute statement 1
no
Does selection value match case label 2?
yes
execute statement 2
no
Does selection value match case label n?
yes
execute statement n
no

RULES

- *Selection—value* is evaluated and compared with the match expressions ([*mexp*[,...]]).
- *Selection—value* cannot be an alpha double-subscripted variable.
- *Selection—value* cannot be an alpha substring because USING creates a temporary field for the value. The length of the alpha substring is not known at compile time, thus the compiler does not know how big to make the temporary field.
- The match expression list ([*mexp*[,...]]) is referred to as a case-label.
- An empty case-label (empty parentheses) is referred to as a null case-label.
- A null case-label matches any *selection—value*.
- The statement associated with the first matching case-label is executed and USING is exited.
- If no match is found, no *statement* within USING is executed.
- Each case-label must begin on a new line.
- *Statement* may be on a separate line.
- No match is found if the value to the left of THRU is greater than the value to the right of THRU.
- The data types of the values in the match expressions (*mexp*) must match.
- The data type of *selection—value* must match the data type of the match expression (*mexp*).

COMPILER ERROR CONDITIONS

- Invalid data type
- No SELECT in USING statement
- No ENDUSING in USING statement
- Stack overflow

RUN-TIME ERROR CONDITIONS

- None

EXAMPLES

In the following example, the USING statement is used to check for the decimal character codes for CTRL/U and DELETE.

```
    USING DCHAR SELECT
      (21),                 ; CTRL/U
        BEGIN
        COL=STOOL               ; Reset cursor position to
        CALL POSTN              ; ... start of field
        CALL CLEAR              ; Clear field
        END
      (127),                ; DELETE
        BEGIN
        IF COL.GT.STOOL         ; At beginning of field?
            BEGIN               ; No--
            COL=COL-1           ; Backup column number
            CALL POSTN          ; Reset cursor position
            DISPLAY (1,' ')     ; Erase the character
            CALL POSTN          ; Reset cursor position
            END
        END
    ENDUSING
```

The following program displays the message indicating which case of USING was selected.

```
RECORD
        CHARS,  A3                          ; Characters entered
PROC
        OPEN (1,I,'TT:')                    ; Open terminal
AGAIN,  WRITES (1,'Enter 3 characters') ; Display prompt
        READS (1,CHARS,EOF)                 ; Get response
        USING CHARS SELECT                  ; Branch based on CHARS
          ('AAA'),
            WRITES (1,'1st case selected')
          ('AAB' THRU 'AZZ')
            WRITES (1,'2nd case selected')
          ('BAA' THRU 'WZZ')
            WRITES (1,'3rd case selected')
          ('XXX', 'YYY', 'ZZZ'),
            WRITES (1,'4th case selected')
          (),
            WRITES (1,'Null case selected')
        ENDUSING
        GOTO AGAIN
EOF,    CLOSE 1                             ; Close terminal
        STOP
```

3.37 WHILE

FUNCTION

WHILE repetitively executes a statement as long as a condition is true.

FORMAT

WHILE *condition statement*

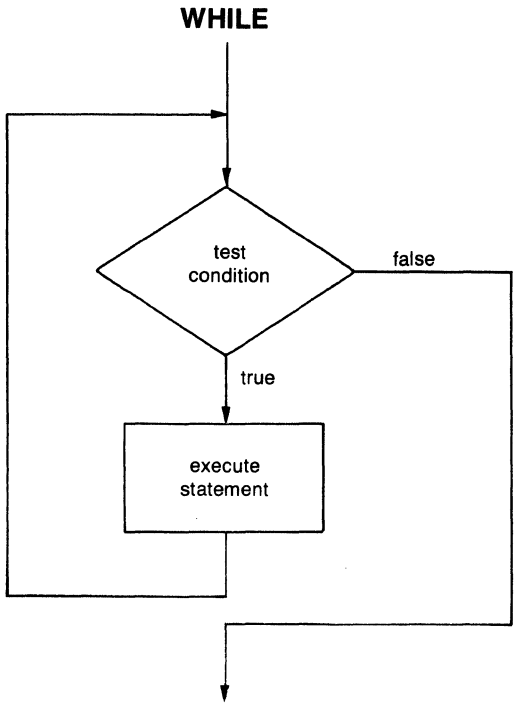
where:

condition is a decimal expression.

statement is a DIBOL Procedure Division statement.

RULES

- The *condition* is evaluated prior to each possible execution of *statement*.
- The *condition* is either true (non-zero) or false (zero).
- If the *condition* is true, *statement* is executed.
- *Statement* may be on a separate line.



COMPILER ERROR CONDITIONS

- Invalid data type
- Stack overflow

RUN-TIME ERROR CONDITIONS

- None

EXAMPLES

The following program segment accepts a line from the terminal. The WHILE statement is used to trim trailing spaces from the input line.

```
RECORD  INLINE
        CHR,     80A1                ; Characters input
RECORD
        SIZE, D2                     ; Number of characters
PROC
        OPEN (1,I,'TT:')             ; Open terminal
        READS (1,INLINE)             ; Accept terminal input
        SIZE=80                      ; Set size of line
        WHILE CHR(SIZE).EQ.' ' .AND. SIZE.GT.1 ; Trim line
            SIZE=SIZE-1
        .
        .
        .
```

3.38 WRITE (RELATIVE FILE)

FUNCTION

WRITE outputs a record into a specified position in a relative file.

FORMAT

WRITE (*ch*,*record*,*dexp*)

where:

ch is a decimal expression that evaluates to a channel number as specified in a previous OPEN statement.

record is an alpha field, alpha literal, or record which contains the data to be written.

dexp is a decimal expression that specifies the sequence number of the record to be written.

RULES

- WRITE is used in O and U modes.
- WRITE updates the *record* if it exists. If no *record* exists, WRITE creates one.
- WRITE locks the *record* it is writing and unlocks the *record* when the WRITE is completed.

COMPILER ERROR CONDITIONS

- Invalid data type

RUN-TIME ERROR CONDITIONS

- 1 T End of file
- 10 NT Illegal channel number
- 11 NT Channel not open
- 21 T Bad OPEN
- 22 T I-O error
- 26 T Field or record too long
- 28 T Illegal record number
- 40 T Record locked
- 84 T Illegal block I/O record size

EXAMPLES

The following statement writes the data in the variable REX into the 88th record of the relative file associated with channel 5.

```
WRITE (5,REX,88)
```

The following statement writes the data in the variable BLT into the relative file associated with channel 6. The record number is specified by the value stored in the variable COUNT.

```
WRITE (6,BLT,COUNT)
```

3.39 WRITE (INDEXED FILE)

FUNCTION

WRITE updates a record in an indexed file.

FORMAT

WRITE (*ch*,*record*,*keyfld*)

where:

ch is a decimal expression that evaluates to a channel number as specified in a previous OPEN statement.

record is an alpha field or record which contains the data to be written.

keyfld is an alpha field or record which identifies the record into which the data will be written.

RULES

- WRITE is used in U:I mode.
- *Keyfld* must occupy the same position within the record as does a key field defined for the indexed file. If it is a multikey file, any key field may be used.
- WRITE updates the *record* if the record to be replaced was the last record read and its key field has the same value as the last record read.
- The *record* to be written is the *record* most recently read on the specified channel and the record must still be locked. WRITE unlocks the *record* when the WRITE is completed.

COMPILER ERROR CONDITIONS

- Invalid data type

RUN-TIME ERROR CONDITIONS

- 1 T End of file
- 10 NT Illegal channel number
- 11 NT Channel not open
- 21 T Bad OPEN
- 22 T I-O Error
- 26 T Field or record too long
- 52 T Illegal key
- 53 T Key not same
- 61 T No current record

EXAMPLES

The following statement will update a record in the indexed file opened on channel 1. The data for the record is in ADDR and the key field is in KEY.

```
WRITE (1,ADDR,KEY)
```

3.40 WRITES

FUNCTION

WRITES outputs a record to the next available position in a file.

FORMAT

WRITES (*ch*,*record*)

where:

ch is a decimal expression that evaluates to a channel number as specified in a previous OPEN statement.

record is an alpha field, alpha literal, or record which contains the data to be written.

RULES

- WRITES is used in O mode with a sequential file; in O and U modes with a relative file; in I and O modes with a terminal; and in O mode with a printer.
- In U mode, WRITES locks the *record* it is writing and unlocks the *record* when the WRITES is completed.

COMPILER ERROR CONDITIONS

- Invalid data type

RUN-TIME ERROR CONDITIONS

- 10 NT Illegal channel number
- 11 NT Channel not open
- 21 T Bad OPEN
- 22 T I-O error
- 25 T Output file full
- 26 T Field or record too long

EXAMPLES

The following statement transfers the data in the array field PAY(EMPLNO) to the next sequential record in the file associated with channel 4. PAY(EMPLNO) must be an alpha field.

```
WRITES (4,PAY(EMPLNO))
```

Assuming that LPT contains the channel number associated with the printer, the following statement transfers the 2nd through the 9th characters in the variable MESSAG to the printer.

```
WRITES (LPT,MESSAG(2,9))
```

3.41 XCALL

FUNCTION

XCALL transfers program control to an external program.

FORMAT

XCALL *name* (*arg*[,...])

where:

name is the name of the external subroutine being called.

arg is an alpha field, alpha literal, decimal field, decimal literal, expression, or record which contains the subroutine arguments.

RULES

- Each argument is linked to a corresponding argument definition in the called subroutine to provide the logical connections necessary to pass data. The first XCALL argument is linked to the first *arg*ument in the subroutine, the second is linked to the second, etc.
- *Arg*uments in the argument list are separated by commas.
- A given *arg*ument may be omitted from the argument list. If more *arg*uments are needed, their place must be held by putting in the commas, e.g., XCALL SUB (A,,C).
- For decimal fields, the returned value is moved to the field according to the rules for moving decimal data (see section 3.2.2).
- For alpha fields and records, the returned value is moved to the field according to the rules for moving alpha data (see section 3.2.1).
- If the number of arguments passed exceeds the number expected by the subroutine, an error is generated.
- If the number of arguments is fewer than expected, no error is generated; it is the responsibility of the subroutine to check for the existence of each argument.
- XCALL causes information to be stored in an internal stack. This stack is of finite size; if too many XCALL statements are executed without an intervening RETURN, the stack will overflow. The exact size of the stack is system dependent and the exact number of XCALL statements which can be nested will vary. In general, programmers should limit subroutine nesting to 15 levels.
- Following the execution of the subroutine, execution of the calling routine begins with the statement which logically follows the XCALL.
- The size of a missing external subroutine argument is -1.
- An external subroutine cannot call itself.

RULES FOR SUBROUTINE NAME

- A subroutine *name* consists of up to 6 characters, the first of which must be alphabetic. Remaining characters can be alphabetic, numeric, or _(underscore).
- Only the first 6 characters of a subroutine *name* are significant; remaining characters are ignored.

COMPILER ERROR CONDITIONS

- None

RUN-TIME ERROR CONDITIONS

- 4 NT DIBOL stack overflow
- 60 NT R6 stack overflow

EXAMPLES

In the following example, the main program calls the external subroutine (CNVRT) to change the format of the date. It passes the arguments DATE and X—DATE. These arguments are represented in the subroutine as OLD and NEW.

Main Program

```
RECORD
        DATE,   D6, 010750
        XDATE, A11
PROC
        XCALL CNVRT (DATE,XDATE)            ; Convert the date
        OPEN (1,O,'TT:')                    ; Open the terminal
        WRITES (1,XDATE)                    ; Display the date
        CLOSE 1                             ; Close the terminal
        STOP
```

External Subroutine

```
SUBROUTINE CNVRT                ; Convert the date format
        OLD,    D                       ; Date (mmddyy)
        NEW,    A                       ; Date (dd-mmm-yy)

RECORD  ODATE                           ; Old date format
        MM,     D2                      ; Month
        DD,     D2                      ; Day
        YY,     D2                      ; Year

RECORD  NDATE                           ; New date format
        DAY,    A2                      ; Day
        ,       A1,'-'
        MONTH,  A3                      ; Month
        ,       A1,'-'
        YEAR,   D2                      ; Year

RECORD
        MNAME,  12A3, 'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun'
_                   , 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'

PROC
        ODATE=OLD
        DAY=DD                          ; Move day to new format
        YEAR=YY                         ; Move year to new format
        MONTH=MNAME(MM)                 ; Move month to new format
        NEW=NDATE                       ; Return new date
        RETURN
```

Arguments can also be made optional. This requires some coordination between the calling program and the external subroutine. The external subroutine must determine whether a given optional argument was passed. This is done by using the SIZE subroutine. If the SIZE subroutine returns a negative value, then the subroutine argument was not passed. The following external subroutine accepts up to 3 file names to delete.

```
SUBROUTINE DEL3                 ; Subroutine to delete 3 files
        FILE1, A                        ; First file
        FILE2, A                        ; Second file
        FILE3, A                        ; Third file
RECORD
        SIZE,  D3
PROC
        XCALL SIZE (FILE1,SIZE)         ; Get size of first name
        IF SIZE.GT.0                    ; Was argument passed?
            XCALL DELET (1,FILE1)       ; Yes--Delete file
        XCALL SIZE (FILE2,SIZE)         ; Get size of second name
        IF SIZE.GT.0                    ; Was argument passed?
            XCALL DELET (1,FILE2)       ; Yes--Delete file
        XCALL SIZE (FILE3,SIZE)         ; Get size of third name
        IF SIZE.GT.0                    ; Was argument passed?
            XCALL DELET (1,FILE3)       ; Yes--Delete file
        RETURN
```

The DEL3 external subroutine can be called using many different types of argument lists. Some of the possible argument lists are shown below. F1, F2, and F3 are assumed to be valid file specifications.

```
XCALL DEL3 (F1)

XCALL DEL3 (F1,F2)

XCALL DEL3 (F1,F2,)

XCALL DEL3 (F1,,F3)

XCALL DEL3 (,,F3)

XCALL DEL3 ()

XCALL DEL3
```

CHAPTER 4
THE DIBOL-83 COMPILER DIRECTIVES

4.1 INTRODUCTION

This chapter contains information about all the Compiler Directives. For easy reference, the Compiler Directives are arranged alphabetically.

Compiler Directives can be used anywhere in a program except as the statement to be executed in an IF, IF-THEN-ELSE, FOR,WHILE, DO-UNTIL, or USING statement.

Compiler Directives cannot have labels.

4.2 .IFDEF-.ENDC

FUNCTION

.IFDEF specifies conditional compilation based on the definition of a variable.

FORMAT

```
.IFDEF field
  statement
    .
    .
.ENDC
```

where:

field is an alpha field, decimal field, or record that must be defined if the statements that follow are to be compiled.

statement is a DIBOL statement.

RULES

- The statements between .IFDEF and .ENDC are compiled only if the *field* is defined in the Data Division before .IFDEF.
- Conditional compilation directives may be nested.
- Compiler Directives have no effect when they are within conditionally uncompiled code.

COMPILER ERROR CONDITIONS

- Directive error

EXAMPLES

In the following example the INCR statement is not compiled because the variable RT11 is not defined:

```
RECORD
        B,      D1
PROC
.IFDEF RT11
        INCR B
.ENDC
        STOP
```

4.3 .IFNDEF-.ENDC

FUNCTION

.IFNDEF specifies conditional compilation based on the non-definition of a variable.

FORMAT

```
.IFNDEF field
  statement
    .
    .
    .
.ENDC
```

where:

field is an alpha field, decimal field, or record that must not be defined if the statements that follow are to be compiled.

statement is a DIBOL statement.

RULES

- The statements between .IFNDEF and .ENDC are compiled only if the *field* is not defined in the Data Division before .IFNDEF.
- Conditional compilation directives may be nested.
- Compiler Directives have no effect when they are within conditionally uncompiled code.

COMPILER ERROR CONDITIONS

- Directive error

EXAMPLES

In the following example the INCR statement is compiled because the variable RSTS is not defined:

```
RECORD
        B,          D1
PROC
.IFNDEF RSTS
        INCR B
.ENDC
        STOP
```

.INCLUDE

4.4 .INCLUDE

FUNCTION

.INCLUDE directs the compiler to read source code from a specified file.

FORMAT

.INCLUDE filespec

where:

filespec is an alpha literal that contains the file specification of the file to be included.

RULES

- When the compiler encounters .INCLUDE, the compiler stops reading statements from the current file and reads the statements in the included file. When it reaches the end of the included file, the compiler resumes compilation with the next logical line after .INCLUDE.
- The *filespec* may contain only one specification.
- The default extension for the file is the same as the default extension for DIBOL program source files. Other system dependent information in the specification follows the system defaults.
- .INCLUDE may be nested to 3 levels.

COMPILER ERROR CONDITIONS

- Cannot open .INCLUDE file <filespec>
- .INCLUDE nested too deeply - file ignored <filespec>

EXAMPLES

.INCLUDE is particularly useful for including standard record descriptions. Assume the file EMPREC.DBL contains the following information:

```
RECORD  EMPREC                  ; Employee record
        NAME,   A20             ; Employee name
        BADGE,  A5              ; Employee badge number
```

The .INCLUDE in the following program will include the employee record description (stored in the file EMPREC.DBL):

```
.INCLUDE 'EMPREC.DBL'
RECORD
        DONE,  A1
PROC
        OPEN (1,O,'TT:')              ; Open terminal
        OPEN (2,U;I,'EMPFIL')         ; Open employee file
        DO
            BEGIN
            WRITES (1,'Name?')        ; Prompt for name
            READS (1,NAME)            ; Get employee name
            WRITES (1,'Badge?')       ; Prompt for badge number
            READS (1,BADGE)           ; Get badge number
            STORE (2,EMPREC,BADGE)    ; Create employee record
            WRITES (1,'Done?')        ; Ask if finished
            READS (1,DONE)            ; Get response
            END
        UNTIL DONE.EQ.'Y'
        CLOSE 1                       ; Close terminal
        CLOSE 2                       ; Close employee file
        STOP
```

The resulting program listing will contain:

```
   .INCLUDE 'EMPREC.DBL'
 1 RECORD EMPREC                      ; Employee record
 2        NAME,   A20                 ; Employee name
 3        BADGE,  A5                  ; Employee badge number
 4 RECORD
 5        DONE,   A1
 6 PROC
 7        OPEN (1,O,'TT:')             ; Open terminal
 8        OPEN (2,U;I,'EMPFIL')        ; Open employee file
 9        DO
10            BEGIN
11            WRITES (1,'Name?')       ; Prompt for name
12            READS (1,NAME)           ; Get employee name
13            WRITES (1,'Badge?')      ; Prompt for badge number
14            READS (1,BADGE)          ; Get badge number
15            STORE (2,EMPREC,BADGE)   ; Create employee record
16            WRITES (1,'Done?')       ; Ask if finished
17            READS (1,DONE)           ; Get response
18            END
19        UNTIL DONE.EQ.'Y'
20        CLOSE 1                      ; Close terminal
21        CLOSE 2                      ; Close employee file
22        STOP
```

LIST
4.5 .LIST

FUNCTION

.LIST enables the source code listing.

FORMAT

.LIST

RULES

- .LIST is the default condition when beginning a compilation.
- .LIST and all subsequent source file input is listed.
- Normal listing continues until the end of the program or until a .NOLIST directive is encountered.
- .LIST always enables the listing regardless of the number of .NOLIST directives that preceded the .LIST. .LIST/.NOLIST cannot be nested.
- .LIST does not affect the content of the listing beyond the last line of the source code.

COMPILER ERROR CONDITIONS

- None

EXAMPLES

The .LIST and .NOLIST directives in the following program will affect the listing of the program. The .NOLIST disables listing the EMPREC record description and the .LIST enables listing the remainder of the program.

```
.NOLIST
RECORD  EMPREC                          ; Employee record
        NAME,   A20                     ; Employee name
        BADGE,  A5                      ; Employee badge number
.LIST
RECORD
        DONE,   A1
PROC
        OPEN (1,O,'TT:')                ; Open terminal
        OPEN (2,U;I,'EMPFIL')           ; Open employee file
        DO
```

```
        BEGIN
        WRITES (1,'Name?')      ; Prompt for name
        READS (1,NAME)          ; Get employee name
        WRITES (1,'Badge?')     ; Prompt for badge number
        READS (1,BADGE)         ; Get badge number
        STORE (2,EMPREC,BADGE) ; Create employee record
        WRITES (1,'Done?')      ; Ask if finished
        READS (1,DONE)          ; Get response
        END
    UNTIL DONE.EQ.'Y'
    CLOSE 1                     ; Close terminal
    CLOSE 2                     ; Close employee file
    STOP
```

The resulting program listing will contain:

```
  .LIST
4 RECORD
5         DONE,  A1


6 PROC
7         OPEN (1,O,'TT:')            ; Open terminal
8         OPEN (2,U;I,'EMPFIL')       ; Open employee file
9         DO
10            BEGIN
11            WRITES (1,'Name?')      ; Prompt for name
12            READS (1,NAME)          ; Get employee name
13            WRITES (1,'Badge?')     ; Prompt for badge number
14            READS (1,BADGE)         ; Get badge number
15            STORE (2,EMPREC,BADGE) ; Create employee record
16            WRITES (1,'Done?')      ; Ask if finished
17            READS (1,DONE)          ; Get response
18            END
19        UNTIL DONE.EQ.'Y'
20        CLOSE 1                     ; Close terminal
21        CLOSE 2                     ; Close employee file
22        STOP
```

.NOLIST

4.6 .NOLIST

FUNCTION

.NOLIST disables the source code listing.

FORMAT

.NOLIST

RULES

- .NOLIST and all subsequent source file input is not listed.
- If an error is detected while the listing is disabled, the statement containing the error and the error message is listed.
- Normal listing continues only when a .LIST directive is encountered.
- .NOLIST ALWAYS inhibits the listing regardless of the number of .LIST directives that preceded the .NOLIST. .LIST/.NOLIST cannot be nested.
- .NOLIST does not affect the content of the listing beyond the last line of the source code.

COMPILER ERROR CONDITIONS

- None

EXAMPLES

See .LIST for example.

4.7 .PAGE

FUNCTION

.PAGE ends the current listing page and begins a new listing page.

FORMAT

.PAGE

RULES

- .PAGE is the last line listed on the page being completed.

COMPILER ERROR CONDITIONS

- None

EXAMPLES

The .PAGE directive in the following program will place the EMPREC record description on a page by itself:

```
RECORD  EMPREC                      ; Employee record
        NAME,   A20                 ; Employee name
        BADGE,  A5                  ; Employee badge number
RECORD
        DONE,  A1
PROC
        OPEN (1,O,'TT:')             ; Open terminal
        OPEN (2,U;I,'EMPFIL')        ; Open employee file
        DO
            BEGIN
            WRITES (1,'Name?')       ; Prompt for name
            READS (1,NAME)           ; Get employee name
            WRITES (1,'Badge?')      ; Prompt for badge number
            READS (1,BADGE)          ; Get badge number
            STORE (2,EMPREC,BADGE)   ; Create employee record
            WRITES (1,'Done?')       ; Ask if finished
            READS (1,DONE)           ; Get response
            END
        UNTIL DONE.EQ.'Y'
        CLOSE 1                      ; Close terminal
        CLOSE 2                      ; Close employee file
        STOP
```

The resulting program listing will contain:

```
 1 RECORD EMPREC                          ; Employee record
 2        NAME, A20                       ; Employee name
 3        BADGE, A5                       ; Employee badge number
   .PAGE

 4 RECORD
 5         DONE,  A1

 6 PROC
 7         OPEN (1,O,'TT:')               ; Open terminal
 8         OPEN (2,U;I,'EMPFIL')          ; Open employee file
 9         DO
10             BEGIN
11             WRITES (1,'Name?')         ; Prompt for name
12             READS (1,NAME)             ; Get employee name
13             WRITES (1,'Badge?')        ; Prompt for badge number
14             READS (1,BADGE)            ; Get badge number
15             STORE (2,EMPREC,BADGE)     ; Create employee record
16             WRITES (1,'Done?')         ; Ask if finished
17             READS (1,DONE)             ; Get response
18             END
19         UNTIL DONE.EQ.'Y'
20         CLOSE 1                        ; Close terminal
21         CLOSE 2                        ; Close employee file
22         STOP
```

4.8 .TITLE

FUNCTION

.TITLE changes the listing page header.

FORMAT

.TITLE [*text—string*]

where:

text—string is an alpha literal which is the page header text.

RULES

- .TITLE is the first source line listed on a new page.
- If the listing is already at the beginning of a page when .TITLE is encountered, no new page is generated.
- The *text—string* set by .TITLE is used in the page header of all pages until a new .TITLE directive is encountered.
- The *text—string* is moved to the page header area according to the rules for moving alpha data (see section 3.2.1).
- If no *text—string* is specified, the page header area is filled with spaces.

COMPILER ERROR CONDITIONS

- Directive error

EXAMPLES

The following .TITLE directive will set the title to 'Employee Update Program'. This title will appear at the top of all pages until another .TITLE is encountered.

```
.TITLE  'Employee Update Program'
```

The following .TITLE directive will clear the title for all pages that follow until another .TITLE is encountered.

```
.TITLE
```

CHAPTER 5

UNIVERSAL EXTERNAL SUBROUTINES

This chapter contains information on the DIBOL-83 Universal External Subroutines. Each subroutine is described and an example of its use is given. Some subroutines may differ when used under a particular operating system.

The appropriate operating system User's Guide should be referred to when using any of the subroutines contained in this document.

ASCII

5.1 ASCII

FUNCTION

ASCII returns the ASCII character for a decimal character code.

FORMAT

XCALL ASCII (*dfield*, *afield*)

where:

dfield is a decimal field or decimal literal that contains the decimal character code.

afield is an alpha field or record that is to contain the ASCII character.

RULES

- *Afield* should be a 1 character field.
- The ASCII character is moved to *afield* according to the rules for moving alpha data (see section 3.2.1).
- *Dfield* is treated as a single character code.
- *If dfield* exceeds the range of character codes, *dfield* is automatically converted by dividing the number by 256 and taking the remainder as the character code (258 becomes 2, 259 becomes 3, etc).

RUN-TIME ERROR CONDITIONS

- 6 NT Incorrect number of arguments
- 8 NT Writing into a literal

EXAMPLES

Since 87 is the decimal character code for 'W', CHAR will contain 'W' after executing the following example:

```
RECORD
        NUM,    D2, 87              ; Decimal character code
        CHAR,   A1                  ; ASCII character
PROC
        XCALL ASCII (NUM,CHAR)      ; Get ASCII character
        STOP
```

5.2 DATE

FUNCTION

DATE returns the current system date.

FORMAT

XCALL DATE (*afield*)

where:

afield is an alpha field or record that is to contain the date.

RULES

- *Afield* should be a 9 character field.
- The date is moved to the alpha field according to the rules for moving alpha data (see section 3.2.1).
- The date is returned in the form:

  dd-mmm-yy

  where:

  dd is the day of the month (01-31).

  mmm is the first three characters for the name of the month (JAN, FEB, MAR, JUN, JUL, AUG, SEP, OCT, NOV, and DEC).

  yy is the last two digits of the year (00-99).

RUN-TIME ERROR CONDITIONS

- 6 NT Incorrect number of arguments
- 8 NT Writing into a literal

EXAMPLES

Assuming the current system date is May 13, 1983, DAT will contain '13-MAY-83' upon execution of the following program:

```
RECORD
        DAT,    A9                      ; System date
PROC
        XCALL DATE (DAT)                ; Get system date
        STOP
```

5.3 DECML

FUNCTION

DECML returns the decimal character code for an ASCII character.

FORMAT

XCALL DECML (*afield*, *dfield*)

where:

afield is an alpha field, alpha literal, or record that contains the ASCII character.

dfield is a decimal field that is to contain the decimal character code.

RULES

- If *afield* is longer than one character, only the first (leftmost) character is used.
- The *dfield* should be a three digit field.
- The decimal character code is moved to *dfield* according to the rules for moving decimal data (see section 3.2.2).

RUN-TIME ERROR CONDITIONS

- 6 NT Incorrect number of arguments
- 8 NT Writing into a literal

EXAMPLES

After executing the following example NUM will contain 087, which is the decimal character code for 'W'.

```
RECORD
          NUM,  D3                    ; Decimal character code
          CHAR, A1, 'W'               ; ASCII character
PROC
          XCALL DECML (CHAR,NUM)      ; Get character code
          STOP
```

DELET

5.4 DELET

FUNCTION

DELET removes one or more versions of a file from a directory.

FORMAT

XCALL DELET ([*ch*,] *filespec*)

where:

ch is a decimal field or decimal literal that specifies a channel number.

filespec is an alpha field, alpha literal, or record that contains a file specification.

GENERAL RULES

- If the specified file does not exist, no error is given.
- The file is deleted unless it is protected by the system.

RULES FOR MULTI-VERSION FILE SYSTEMS

- The file specification may contain wildcards.
- If the file specification does not specify a version number, all versions are deleted.
- DELET will attempt to delete all the files before generating an error.

RUN-TIME ERROR CONDITIONS

- 6 NT Incorrect number of arguments

EXAMPLES

The following program will delete the file ARMAST.DDF:

```
RECORD
            NAME,     A10, 'ARMAST.DDF'
PROC
            XCALL DELET (NAME)          ; Delete ARMAST.DDF
            STOP
```

FUNCTION

ERROR returns the error number and the line number at which the last trappable error occurred.

FORMAT

XCALL ERROR (*errnum*, *line*)

where:

errnum is a decimal field that is to contain the error number.

line is a decimal field that is to contain the line number.

RULES

- *Errnum* should be a 3 digit field.
- The error number is moved to *errnum* according to the rules for moving decimal data (see section 3.2.2).
- The *line* field should be large enough to hold the largest line number in the program.
- The line number is moved to *line* according to the rules for moving decimal data (see section 3.2.2).

RUN-TIME ERROR CONDITIONS

- 6 NT Incorrect number of arguments
- 8 NT Writing into a literal

EXAMPLES

Assuming that the statement C = 5/0 is on line 7, LINE will contain 0007 and ERR will contain 0030, which is the **Divide by 0** error number.

```
RECORD
          LINE,   D4
          ERR,    D4
          C,      D4
PROC
          ONERROR BAD
          C=5/0
            .
            .
            .
BAD,      XCALL ERROR (ERR,LINE)        ; Get error and line #
```

5.6 FATAL

FUNCTION

FATAL specifies the action to be taken when a non-trappable error is detected by the run-time system.

FORMAT

XCALL FATAL (*action*[,*filespec*])

where:

action is a decimal field or decimal literal that directs the run-time system what to do in the event of a fatal error.

filespec is an alpha field, alpha literal or record which contains the file specification of a program to be run in place of this program if an untrapped error occurs. It may also be a record or alpha field that will receive the name of the default user-designated program.

RULES

- When an untrapped error occurs, the user-designated program is sent a message which contains error information. The format of the message is:

```
ERR1, D3          ;DIBOL fatal error number
ERR2, D10         ;Additional system information
ERLN, D10         ;Line number of statement causing the error
MODUL, A31        ;Name of module which caused the error
PRGNM, A31        ;Name of the 'root' module
```

- If the program that encounters the error is running detached, the user-designated program is started detached.

- If the program that encounters the fatal error is running at a terminal, the user-designated program is started at the terminal.

- Acceptable action values are:

  0 Return to system level on untrapped error.

  1 Use the default user-designated program on an untrapped error. If there is no default user-designated program, return to the system level.

  2 Use the user-designated program specified by the filespec on the untrapped error. This filespec designation remains in effect while the current program is running.

  3 Return, in the *filespec* field, the name of the default user-designated program. If none is defined, return spaces.

RUN-TIME ERROR CONDITIONS

- 6 NT Incorrect number of arguments
- 8 NT Writing into a literal
- 87 T Argument missing
- 104 NT Value out of range

EXAMPLES

The following statement designates the program BADERR as the program to execute when an untrapped error occurs:

```
XCALL FATAL (2,'BADERR')
```

The following statement specifies that no program is to be loaded when an untrapped error occurs. Instead, control will be returned to the system level.

```
XCALL FATAL (0)
```

FLAGS

5.7 FLAGS

FUNCTION

FLAGS alters operating parameters of the run-time system.

FORMAT

XCALL FLAGS (*parameters* [,*action*])

where:

parameters
is a decimal field or decimal literal which contains the FLAGS parameters.

action is a decimal field or decimal literal which alters the action of the subroutine to provide for flexibility.

RULES

The digits in the *parameters* field are right-justified.

- Each digit corresponds to a parameter.
- The digits are numbered from right to left.
- Acceptable action values are:

Not specified
Parameters where a non-zero appears are enabled and remaining parameters are disabled.

0 *Parameters* where a non-zero appears are disabled and remaining parameters are unchanged.

1 *Parameters* where a non-zero appears are enabled and remaining parameters are unchanged.

2 The current value for the parameters is moved into the *parameters* field and is right-justified.

RULES WHEN ACTION IS 2

- *Parameters* must be a decimal field; it cannot be a literal.
- *Parameters* should be a 10 digit field.
- The parameters are moved to the *parameters* field according to the rules for moving decimal data (see section 3.2.2).

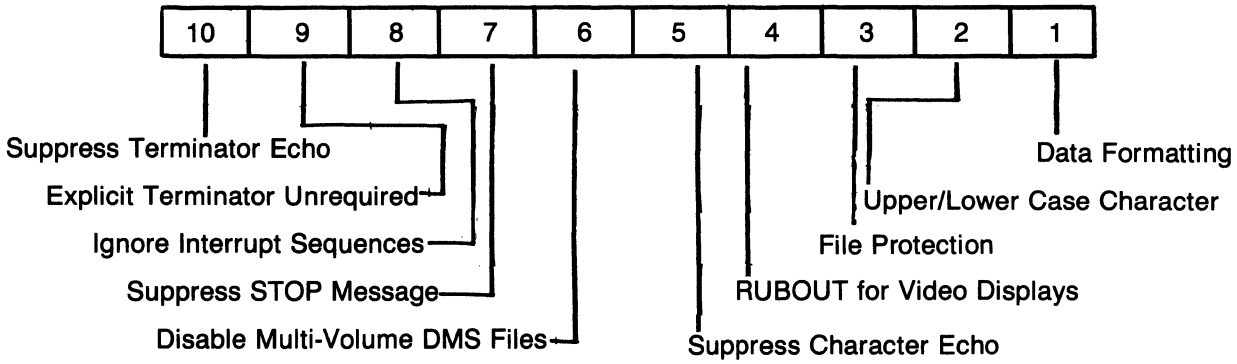


Figure 5-1 FLAGS Option Fields

Table 5-1 FLAGS Argument Assignments

Digit Position	Meaning When Digit is a Non-Zero Value
10	Do not echo the terminator of terminal input.
9	Do not require an explicit terminator for terminal input using READS. Entry into the field is terminated implicitly when it is filled. Entry may be terminated prior to the field being filled by use of a terminator character.
8	Ignore interrupt sequences input from the terminal. This prevents an operator from terminating a program by typing an interrupt sequence (CTRL/C).
7	Suppress STOP message which is displayed on the screen at the end of each program.
6	Do not recognize DMS multi volume files. This means that an input file returns an EOF at the end of the last block, even if a CTRL/Z is not present. This allows you to read files that are not terminated by a CTRL/Z, such as a single volume of multi volume file, files written with an editor, or files produced by the DIBOL compiler. Output files are closed with the unused portion of the final block set to nulls, but no CTRL/Z is added. For DMS DIBOL the **Output file full** error is generated when the file is full.
5	Do not echo characters input from a terminal.
4	Backspace the video terminal's cursor one character position in response to each operation of the DELETE key, and delete the character instead of displaying a backslash (\).
3	Detect an attempt to open an output file when a file of the same name and version already exists on the device. When this condition occurs, the error message Superseding existing file is generated, or, if the Superseding existing file ONERROR trap is enabled, the error is trapped to ensure that no files are accidentally deleted and replaced.
2	Do not convert lowercase characters entered from a terminal to the uppercase equivalents.
1	Enable international data formatting by deleting leading periods rather than leading commas.

RUN-TIME ERROR CONDITIONS

- 6 NT Incorrect number of arguments
- 8 NT Writing into a literal
- 87 T Argument missing
- 104 NT Value out of range

EXAMPLES

Disabling character echo is particularly useful when accepting passwords as in the following example. FLAGS digit 5 is used to control character echo.

```
RECORD
        PASS,   A10                     ; Password
PROC
        OPEN (1,I,'TT:')                ; Open terminal
        DISPLAY (1,'Enter password: ') ; Display password prompt
        XCALL FLAGS (0000010000,1)      ; Disable character echo
        READS (1,PASS)                  ; Accept password
        XCALL FLAGS (0000010000,0)      ; Re-enable character echo
         .
         .
         .
```

5.8 INSTR

FUNCTION

INSTR searches a string of data for another string.

FORMAT

XCALL INSTR (*start*,*string1*,*string2*,*position*)

where:

start is a decimal field or decimal literal which specifies the character position within string1 where the search begins.

string1 is an alpha field, alpha literal, or record to be searched.

string2 is an alpha field, alpha literal, or record to be searched for in string1.

position is a decimal field that is to contain the starting character position of *string2* within *string1*.

RULES

- The starting position specifies the position within *string1* where the search begins. The starting position indicates the leftmost boundary for *string1*.
- If the starting position is less than 1 or is greater than the length of *string1*, no search takes place and the *position* field is set to zero.
- The *position* field is set to a decimal value indicating the leftmost position of *string2* within *string1*. The complete *string2* (all characters in the order specified) must be found within *string1*.
- If the search is unsuccessful, the *position* field is set to zero.
- The value indicating the leftmost position of *string2* within string1 is moved to the *position* field according to the rules for moving decimal data (see section 3.2.2).

RUN-TIME ERROR CONDITIONS

- 6 NT Incorrect number of arguments
- 8 NT Writing into a literal

EXAMPLES

The following program reads in a file name and, if the file name contains '.ISM', opens the file in I:I (indexed) mode. Otherwise, the file is opened in I mode.

```
RECORD
        POS,      D3                          ; Where .ISM was found
        NAME,     A80                         ; File name

PROC
        OPEN (1,I,'TT:')                      ; Open terminal
        DISPLAY (1,'Enter file name: ')       ; Display file name promp
        READS (1,NAME)                        ; Get file name
        XCALL INSTR (1,NAME,'.ISM',POS)       ; Check for indexed file
        IF POS.NE.0                           ; Indexed file?
          THEN                                ; Yes--
            OPEN (3,I:I,NAME)                 ; Open indexed file
          ELSE
            OPEN (3,I,NAME)                   ; Open non-indexed file
              .
              .
              .
```

5.9 JBNO

FUNCTION

JBNO returns the job number.

FORMAT

XCALL JBNO (*dfield*)

where:

dfield is a decimal field that is to contain the job number.

RULES

- The job number is moved to *dfield* according to the rules for moving decimal data (see section 3.2.2).

RUN-TIME ERROR CONDITIONS

- 6 NT Incorrect number of arguments
- 8 NT Writing into a literal

EXAMPLES

The following program will display the job number:

```
RECORD  MSG
        ,       A11, 'Job number '
        JOB,    D3                      ; Job number
PROC
        OPEN (1,O,'TT:')                ; Open terminal
        XCALL JBNO (JOB)                ; Get job number
        WRITES (1,MSG)                  ; Display 'Job number 000'
        CLOSE 1                         ; Close terminal
        STOP
```

MONEY

5.10 MONEY

FUNCTION

MONEY specifies a currency symbol as either the dollar symbol ($) or some other symbol.

FORMAT

XCALL MONEY (*afield*)

where:

afield is an alpha field, alpha literal, or record which contains the currency symbol.

RULES

- The currency symbol may be any ASCII character except comma (,), period (.), asterisk (*), hyphen (-), or the letters X and Z.
- If *afield* is longer than one character, only the first (leftmost) character is used.

RUN-TIME ERROR CONDITIONS

- 6 NT Incorrect number of arguments
- 87 T Argument missing

EXAMPLES

In the following example the MONEY subroutine is used to change the currency symbol to '#'. The example will display '#1234567.89'.

```
RECORD
        A,    D10, 0123456789
        B,    A15
PROC
        OPEN (1,O,'TT:')          ; Open terminal
        XCALL MONEY ('#')         ; Change currency symbol
        B=A,'############.XX'     ; Format value
        WRITES (1,B)              ; Display formatted value
        CLOSE 1                   ; Close terminal
        STOP
```

5.11 PAK

FUNCTION

PAK converts zoned decimal fields to packed decimal.

FORMAT

XCALL PAK (*record*,*size*,*dfield*[,...])

where:

record is an alpha field or record which contains the unpacked source data. This also identifies the destination of the packed result.

size is a decimal field where the size of the packed record is returned.

dfield is one or more decimal fields to be packed.

RULES

- In packed form there are 2 digits per byte plus an extra byte for the sign.
- The size of a packed field (in terms of bytes) is equal to the number of digits in the zoned decimal field divided by 2 plus 1. If the result is not an integer, round down to the next lower integer. The formula for calculating the packed decimal field size is the following:

  packed decimal field size = (number of digits/2) + 1

  Using this formula a zoned decimal field of 16 bytes requires a packed decimal field of 9 bytes or 9 = (16/2) + 1.
- Fields to be packed must be listed in ascending order by position within the record.
- Fields between the packed fields are shifted forward in the resulting packed record.

RUN-TIME ERROR CONDITIONS

- 8 NT Writing into a literal
- 20 T Bad digit
- 77 T Arguments out of order
- 78 T Argument not defined in the record
- 79 T Incorrect argument count
- 80 T Field not packed

EXAMPLES

In the following example the employee record (EMPREC) is packed using the PAK subroutine and is then written to a file. The WRITES statement specifies the substring of EMPREC which contains the packed record.

```
RECORD
        SIZE,   D3                          ; Size of packed field
RECORD  EMPREC                              ; Employee record
        NAME,   A20                         ; Employee name
        BGN,    D6                          ; Beginning date
        SAL,    D10                         ; Current salary
        TITLE,  A10                         ; Current title
        DEP,    D2                          ; Number of dependents
PROC
          .
          .
          .
        XCALL PAK (EMPREC,SIZE,BGN,SAL,DEP) ; Pack employee record
        WRITES (1,EMPREC(1,SIZE))           ; Output packed record
          .
          .
          .
```

5.12 RENAM

FUNCTION

RENAM changes the name of an existing file.

FORMAT

XCALL RENAM ([*ch*,] *newfile*, *oldfile*)

where:

ch is a decimal field or decimal literal that specifies a channel number.

newfile is an alpha field, alpha literal, or record that contains the new file specification.

oldfile is an alpha field, alpha literal, or record that contains the current file specification.

RULES

- The rename operation follows the flowchart in Figure 5-2.
- A file can be renamed from one directory to another, but not from one device to another.
- If *oldfile* does not exist, a **File not found** error is generated and the rename operation is terminated.
- If *newfile* exists and specifies a file different from *oldfile*, but digit position 3 in the FLAGS subroutine is set to prevent the superseding of an existing file, a **Cannot supersede existing file** error is generated and the rename operation is terminated.
- If *newfile* specifies the same file as *oldfile*, the results are system dependent.

  On RSTS/E, if digit position 3 in the FLAGS subroutine is clear, the file is deleted and a **File not found** error is generated. If FLAG 3 is set, the file will not be deleted and no error occurs.

  On VAX and PRO, the file will not be deleted and no error occurs. If FLAG 3 is clear, old versions of the specified file may be deleted.

RULES FOR MULTI-VERSION FILE SYSTEMS

- If an error occurs during the processing of multiple versions of a file (such as a file protection error), processing continues if possible and an error is generated upon completion.
- If the version number of *newfile* is omitted or the version number is a wildcard (specified with an asterisk (*)), all versions of *newfile* will be deleted prior to the actual rename operation.

- If the version number of *oldfile* is omitted or the version number is wild, all versions of *oldfile* wil be renamed.
- If the version number of *oldfile* is 0 or blank, the latest version of *oldfile* will be renamed.
- If the version number of *oldfile* is explicit, that version of *oldfile* will be renamed.
- If the version number of *oldfile* is omitted or the version number is wild, and the version number of *newfile* is explicitly specified, unpredictable results may occur.
- The order of the versions of *oldfile* will be retained when the fields are renamed to *newfile*.

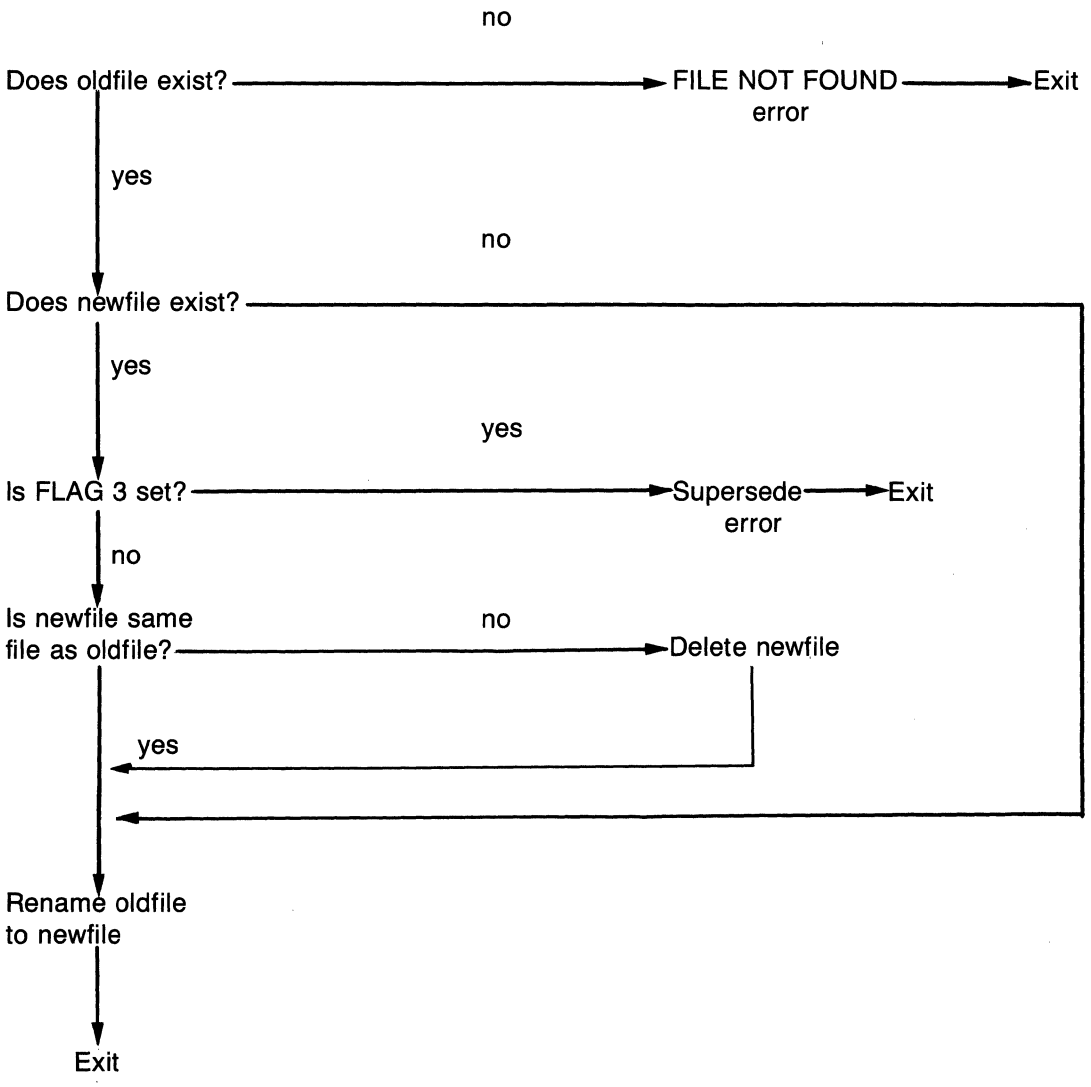


Figure 5-2 RENAM Flowchart

RUN-TIME ERROR CONDITIONS

- 6 NT Incorrect number of arguments
- 18 T File not found
- 32 T Cannot supersede existing file
- 62 T Protection violation
- Various other file-related errors may also occur which are not specific to RENAM.

EXAMPLES

The following statement will rename the file OLDFIL.DDF to NEWFIL.DDF:

```
XCALL RENAM ('NEWFIL.DDF','OLDFIL.DDF')
```

5.13 RSTAT

FUNCTION

RSTAT returns the size and terminating character for the last record read by a READ or READS statement.

FORMAT

XCALL RSTAT (*size*[,*char*])

where:

size is a decimal field that is to contain the record size.

char is an alpha field or record that is to contain the terminating character.

RULES

- The record size is moved to *size* according to the rules for moving decimal data (see section 3.2.2).
- *Char* should be a 1 character field.
- The terminating character is moved to *char* according to the rules for moving alpha data (see section 3.2.1).

RUN-TIME ERROR CONDITIONS

- 6 NT Incorrect number of arguments
- 8 NT Writing into a literal
- 87 T Argument missing

EXAMPLES

The program that follows creates a sequential file called NEWFIL.DDF and fills the file with records from the file called OLDFIL.DDF (i.e., a copy operation). Since the size of the input records may vary, RSTAT is used to obtain the record size following each READS. The WRITES is then done by specifying a substring.

```
RECORD
        IN,     A256                    ; Input record
        SIZE,   D3                      ; Input record size
PROC
        OPEN (1,I,'OLDFIL.DDF')         ; Open old file
        OPEN (2,O,'NEWFIL.DDF')         ; Create new file
LOOP,   READS (1,IN,DONE)               ; Read a record
        XCALL RSTAT (SIZE)              ; ...and get its size
        WRITES (2,IN(1,SIZE))           ; Copy record to new file
        GOTO LOOP
DONE,
        CLOSE 1
        CLOSE 2
        STOP
```

5.14 RUNJB

FUNCTION

RUNJB starts another program without terminating the current program.

FORMAT

XCALL RUNJB (*filespec*, *terminal*)

where:

filespec is an alpha field, alpha literal, or record which contains the file specification of the program to be executed.

terminal is a decimal field or decimal which specifies the terminal number.

RULES

- The *terminal* number specifies the terminal to which the program is to be attached.
- If the *terminal* number is -1, no terminal is attached to the job.
- RUNJB cannot be used in a non multi tasking environment.

RUN-TIME ERROR CONDITIONS

- 6 NT Incorrect number of arguments
- 58 T Job startup error

EXAMPLES

The following program starts a program called MENU on terminals 1 and 2 and starts a detached program called UPDT:

```
PROC
        XCALL RUNJB ('MENU',1)   ; Start program on terminal 1
        XCALL RUNJB ('MENU',2)   ; Start program on terminal 2
        XCALL RUNJB ('UPDT',-1) ; Start detached program
        STOP
```

5.15 SIZE

FUNCTION

SIZE returns the size of a field.

FORMAT

XCALL SIZE (*field*,*size*)

where:

field is an alpha field, alpha literal, decimal field, decimal literal, or record whose size is to be returned.

size is a decimal field which is to contain the size.

RULES

- The size of a subroutine argument which is not passed is -1.
- The size of an alpha field or decimal field is the number of characters as specified in the Data Division.
- The size of a record is the sum of the size of the fields which are part of the record.
- The size of an alpha literal is the number of characters required to store it.
- The size of a decimal literal is equal to the actual number of digits in the literal. Plus and minus signs are not counted.
- The size is moved to *size* according to the rules for moving decimal data (see section 3.2.2).

RUN-TIME ERROR CONDITIONS

- 6 NT Incorrect number of arguments
- 8 NT Writing into a literal
- 87 T Argument missing

EXAMPLES

Creating a relative file requires that the size of the records that will be placed in the file be specified in the OPEN. This can be done by counting the characters and hard-coding the value in the OPEN. This can also cause maintenance problems when new fields are added to the record. A better method is to use the SIZE subroutine to determine the size of the records as in the following example:

```
RECORD
        SIZE,  D3                       ; Size of packed field
RECORD  EMPREC                          ; Employee record
        NAME,  A20                      ; Employee name
        BGN,   D6                       ; Beginning date
        SAL,   D10                      ; Current salary
        TITLE, A10                      ; Current title
        DEP,   D2                       ; Number of dependents
PROC
        XCALL SIZE (EMPREC,SIZE)    ; Get employee record size
        OPEN (1,O;R,'EMPFIL.DDF',RECSIZ:SIZE) ; Create file
          .
          .
          .
```

5.16 TIME

FUNCTION

TIME returns the current system time of day.

FORMAT

XCALL TIME (*dfield*)

where:

dfield is a decimal field that is to contain the time.

RULES

- *Dfield* should be a 6 digit field.
- The time is moved to *dfield* according to the rules for moving decimal data (see section 3.2.2).
- The time is returned in a 24-hour notation in the format:

  hhmmss

  where:

  hh is the number of hours elapsed since midnight.
  mm is the number of minutes elapsed since the last hour.
  ss is the number of seconds elapsed since the last minute.

RUN-TIME ERROR CONDITIONS

- 6 NT Incorrect number of arguments
- 8 NT Writing into a literal

EXAMPLES

Assuming that the current time is 2:45:57 P.M., CURTIM will contain 144557 in the following example:

```
RECORD
        CURTIM, D6                  ; Current time
PROC
        XCALL TIME (CURTIM)         ; Get current time
        STOP
```

TNMBR

5.17 TNMBR

FUNCTION

TNMBR returns the number of the terminal to which the program is attached.

FORMAT

XCALL TNMBR (*terminal*)

where:

terminal is a decimal field which is to contain the terminal number.

RULES

- The terminal number is moved to *terminal* according to the rules for moving decimal data (see section 3.2.2).
- If the program is running detached, TNMBR returns -1.

RUN-TIME ERROR CONDITIONS

- 6 NT Incorrect number of arguments
- 8 NT Writing into a literal

EXAMPLES

The following program calls TNMBR to determine if it is running detached. If it is not detached, the program displays the terminal number.

```
RECORD MSG
       ,     A9, 'Terminal '
       TT,   D3
PROC
       XCALL TNMBR (TT)              ; Get terminal number
       IF TT.GE.0                    ; Is program detached?
           BEGIN                     ; No--
           OPEN (1,0,'TT:')          ; Open terminal
           WRITES (1,MSG)            ; Display terminal #
           CLOSE 1                   ; Close terminal
           END
        STOP
```

5.18 TTSTS

FUNCTION

TTSTS returns an indication of waiting terminal input.

FORMAT

XCALL TTSTS (*dfield*[,*ch*])

where:

dfield is a decimal field which is to contain the number of characters waiting to be input.

ch is a decimal field or decimal literal that evaluates to a channel number as specified in a previous OPEN statement.

RULES

- TTSTS indicates the status by returning one of the following in *dfield*:

  zero (0) if no characters are waiting

  non-zero if one or more characters are waiting.

- The status is moved to *dfield* according to the rules for moving decimal data (see section 3.2.2).
- If there is at least one character in the buffer, execution of an ACCEPT will not cause an I/O wait.
- If there is nothing in the buffer and an ACCEPT is done, the program will wait for keyboard input.

RUN-TIME ERROR CONDITIONS

- 6 NT Incorrect number of arguments
- 8 NT Writing into a literal
- 21 T Bad OPEN

EXAMPLES

The following example continuously displays a counter at the terminal. However, the program is designed to stop if a carriage return is entered.

```
RECORD    NUMBER
          ,       A15, 'Loop counter = '
          CTR,    D5
RECORD
          ARG,    D1
          CHAR,   D3
PROC
          OPEN (1,I,'TT:')              ; Open terminal
          DO
              BEGIN
              INCR CTR                  ; Increment loop counter
              WRITES (1,NUMBER)         ; Display loop counter
              XCALL TTSTS (ARG,1)       ; See if a key was typed
              IF ARG                    ; Was a character entered?
                  ACCEPT (1,CHAR)       ; Yes--Get the character
              END
          UNTIL CHAR.EQ.13              ; Carriage return?
          CLOSE 1                       ; Close terminal
          STOP
```

5.19 UNPAK

FUNCTION

UNPAK converts packed decimal fields to zoned decimal.

FORMAT

XCALL UNPAK (*record*,*dfield*[,...])

where:

record is an alpha field or record that contains the packed data. This also identifies the destination of the unpacked result.

dfield is one or more decimal fields to receive the unpacked data.

RULES

- The packed record base address is used as the reference point and the unpacked record overlays the packed record.
- Fields to be unpacked must be listed in ascending order by position within the record.
- If all the fields are unpacked, the fields between the packed fields are shifted back to their original positions.

RUN-TIME ERROR CONDITIONS

- BAD DIGIT
- ARGUMENTS OUT OF ORDER
- ARGUMENT OUT OF RECORD LIMIT
- ARGUMENT COUNT
- FIELD NOT PACKED

EXAMPLES

In the following example the employee record (EMPREC), which contains packed data, is read from a file and is unpacked using the UNPAK subroutine. The READS statement specifies the substring of EMPREC which contains the packed record. This size was returned by the PAK subroutine when the record was written.

```
RECORD
        SIZE,    D3                    ; Size of packed field
RECORD  EMPREC                         ; Employee record
        NAME,    A20                   ; Employee name
        BGN,     D6                    ; Beginning date
        SAL,     D10                   ; Current salary
        TITLE,   A10                   ; Current title
        DEP,     D2                    ; Number of dependents
PROC
          .
          .
          .
        XCALL PAK (EMPREC,SIZE,BGN,SAL,DEP) ; Pack employee record
        WRITES (1,EMPREC(1,SIZE))            ; Output packed record
          .
          .
          .
        READS (1,EMPREC(1,SIZE))             ; Read packed record
        XCALL UNPAK (EMPREC,BGN,SAL,DEP)     ; Unpack record
          .
          .
          .
```

5.20 VERSN

FUNCTION

VERSN returns the DIBOL version number.

FORMAT

XCALL VERSN (*afield*)

where:

afield is an alpha field or record which is to contain the version number.

RULES

- The version number is returned in the following format:

  vvvvvvvssssstmm.nnpp

  where:

vvvvvvv	is the operating system.
ssss	is the operating system subsystem.
t	is the release status of the product. V = Released Version Y = Field Test Version X = Internal Version
mm	is the major version number.
nn	is the minor version number.
pp	is the patch level.

  Table 5-2 contains the returned character string for each system.

- The version number is moved to *afield* according to the rules for moving alpha data (see section 3.2.1).

Table 5-2 VERSN Returned Formats

Operating Sub-System	Returned Format
CTS-300 Single User DIBOL	RT11 SUD V08.0000
CTS-300 Timeshare DIBOL	RT11 TSD V08.0000
CTS-300 Extended Memory Timeshare	RT11 XMT V08.0000
RSTS/E DMS	RSTS/E DMS V05.00
RSTS/E RMS	RSTS/E RMS V05.00
VAX/VMS DIBOL	VAX/VMS V02.00
Professional DIBOL	Pro-DIBOL V01.00

RUN-TIME ERROR CONDITIONS

- 6 NT Incorrect number of arguments
- 8 NT Writing into a literal

EXAMPLES

The following program displays the DIBOL version number:

```
RECORD    REPLY
          OPSYS,    A7                    ; Operating system
          SUB,      A4                    ; Subsystem
          VERSN,    A8                    ; tmm.nnpp
PROC
          OPEN (1,O,'TT:')                ; Open terminal
          XCALL VERSN (REPLY)             ; Get version number
          WRITES (1,REPLY)                ; Display response
          DISPLAY (1,'Operating system is ',OPSYS,13,10)
          DISPLAY (1,'Subsystem is ',SUB,13,10)
          DISPLAY (1,'DIBOL version is ',VERSN,13,10)
          CLOSE 1                         ; Close terminal
          STOP
```

5.21 WAIT

FUNCTION

WAIT suspends program execution pending the occurrence of an event.

FORMAT

XCALL WAIT ([*seconds*],*parameters*[,*event*])

where:

seconds is a decimal field or decimal literal that specifies the number of seconds to suspend program execution.

parameters
is a decimal field or decimal literal which specifies the events to wait for.

event is a decimal field which is to contain the digit corresponding to the event that occurred.

RULES

- The digits in *parameters* are right-justified.
- Each digit corresponds to a specific event; a non-zero digit enables the event.
- If more than one event is specified, the first event to occur will cause the program to resume execution.
- The digits are numbered from right to left.
- The event number that occurred is moved to *event* according to the rules for moving decimal data (see section 3.2.2).
- The *seconds* argument is required only when digit 1 in *parameters* is used.

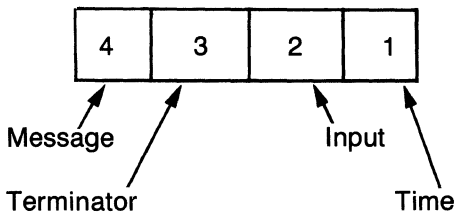


Figure 5-3 WAIT Option Fields

Table 5-3

WAIT Argument Assignments

Digit Position	Meaning When Digit is Non-zero
4	Wait for a message.
3	Wait for a terminator (CR, LF,FF, VT, ESC) to be typed at any of the keyboards opened by the program.
2	Wait for terminal input at any one of the keyboards opened by the program.
1	Wait for specified time to expire.

RUN-TIME ERROR CONDITIONS

- 87 T Argument missing
- 118 T Unable to open message manager mailbox

EXAMPLES

The following example waits for a message to be sent to the program. If a message hasn't been received within 30 seconds, the program stops.

```
RECORD
        MSG,    A512                    ; Message
        SIZE,   D3                      ; Message size
        REASON, D1                      ; WAIT expiration reason
PROC
AGN,    XCALL WAIT (30,1001,REASON)     ; Wait for message
        IF REASON.NE.4                  ; Is there a message?
            STOP                        ; No--
        RECV (MSG,AGN,SIZE)             ; Yes--Get the message
          .
          .
          .
```

APPENDIX A

DIBOL-83 CHARACTER SET

Table A-1 shows the 128-character ASCII character set and the corresponding decimal codes used by DIBOL-83 for data and program statements. The order of the character set, as shown, establishes the collating sequence.

All characters may be used for data input from the terminal and output to the terminal and printer.

DIBOL-83 stores both alpha and decimal data in character code form. To distinguish between positive and negative numbers, the negative numbers are stored with a character in the place of the least significant digit. The characters p through y are used to represent the least significant digit (0-9) in a negative number. Thus, the negative value -1234 (or 1234-) is stored internally as 123t. This means that any program that neglects to perform decimal-to-alpha conversion before output to a device may produce numeric values that contain an alphabetic character as the least significant digit.

Table A-1
DI80L-83 Character Set

DEC	HEX	OCT	ASC	DEC	HEX	OCT	ASC	DEC	HEX	OCT	ASC	DEC	HEX	OCT	ASC
000	00	000	NUL	032	20	040	<SPACE>	064	40	100	@	096	60	140	`
001	01	001		033	21	041	!	065	41	101	A	097	61	141	a
002	02	002		034	22	042	"	066	42	102	B	098	62	142	b
003	03	003		035	23	043	#	067	43	103	C	099	63	143	c
004	04	004		036	24	044	$	068	44	104	D	100	64	144	d
005	05	005		037	25	045	%	069	45	105	E	101	65	145	e
006	06	006		038	26	046	&	070	46	106	F	102	66	146	f
007	07	007	^G(BEL)	039	27	047	'	071	47	107	G	103	67	147	g
008	08	010	^H(BS)	040	28	050	(	072	48	110	H	104	68	150	h
009	09	011	^I(HT)	041	29	051	)	073	49	111	I	105	69	151	i
010	0A	012	^J(LF)	042	2A	052	*	074	4A	112	J	106	6A	152	j
011	0B	013	^K(VT)	043	2B	053	+	075	4B	113	K	107	6B	153	k
012	0C	014	^L(FF)	044	2C	054	,	076	4C	114	L	108	6C	154	l
013	0D	015	^M(CR)	045	2D	055	-	077	4D	115	M	109	6D	155	m
014	0E	016		046	2E	056	.	078	4E	116	N	110	6E	156	n
015	0F	017		047	2F	057	/	079	4F	117	O	111	6F	157	o
016	10	020		048	30	060	0	080	50	120	P	112	70	160	p -0
017	11	021		049	31	061	1	081	51	121	Q	113	71	161	q -1
018	12	022		050	32	062	2	082	52	122	R	114	72	162	r -2
019	13	023		051	33	063	3	083	53	123	S	115	73	163	s -3
020	14	024		052	34	064	4	084	54	124	T	116	74	164	t -4
021	15	025		053	35	065	5	085	55	125	U	117	75	165	u -5
022	16	026		054	36	066	6	086	56	126	V	118	76	166	v -6
023	17	027		055	37	067	7	087	57	127	W	119	77	167	w -7
024	18	030		056	38	070	8	088	58	130	X	120	78	170	x -8
025	19	031		057	39	071	9	089	59	131	Y	121	79	171	y -9
026	1A	032		058	3A	072	:	090	5A	132	Z	122	7A	172	z
027	1B	033	<ESC>	059	3B	073	;	091	5B	133	[	123	7B	173	{
028	1C	034		060	3C	074	<	092	5C	134	\	124	7C	174	|
029	1D	035		061	3D	075	=	093	5D	135	]	125	7D	175	}
030	1E	036		062	3E	076	>	094	5E	136	^	126	7E	176	~
031	1F	037		063	3F	077	?	095	5F	137	—	127	7F	177	DEL

GLOSSARY

alpha
A character set that contains letters, digits, and other characters, such as punctuation marks.

alphabetic
A character set that contains only letters.

array
A DIBOL technique for specifying more than one field of the same length and type. The array 5D3 reserves space for five numeric fields, each to be three digits long. The array 2A10 describes two alpha fields, each to be ten characters long.

ASCII
American Standard Code for Information Interchange. This is one method of coding alpha characters.

binary operator
An operator, such as * or /, which acts upon two or more constants or variables (e.g., B*C).

branch
A change in the sequence of execution of DIBOL-83 program statements.

byte
A group of eight bits considered as a unit.

channel
A number used to associate an input/output statement with a specified device.

character
A letter, digit, or other symbol used to control or to represent data. One character is equivalent to one byte.

character string
A connected linear sequence of characters.

clear
Setting an alpha field to spaces or a numeric field to zeros.

comments
Notes for people to read. They do not affect program execution or size.

data
A representation of information in a manner suitable for communication, interpretation, or processing by either people or machines. In DIBOL-83 systems, data is represented by characters.

DEC
Acronym for Digital Equipment Corporation.

decimal
Refers to a base ten number.

DIBOL-83
Digital's Interactive Business Oriented Language is used to write business application programs. It is based on the 1983 Standard.

direct access
The process of obtaining data from, or placing data into, a storage device where the availability of the data requested is independent of the location of the data most recently obtained or placed in storage.

dump
To copy the contents of all or part of storage, usually from memory to external storage.

end-of-file mark
A control character which marks the physical end of a file.

expressions
Variables, constants, or arithmetic expressions made up of variables, constants, and the operators #, +, —, *, and /.

fatal error
An error which terminates program execution.

field
A specified area in a data record used for alpha or numeric data; cannot exceed the specified character length.

file
A collection of records, treated as a logical unit.

file specification (filespec)
The general file name.

flowchart
A pictorial technique for analysis and solution of data flow and data processing problems. Symbols represent operations, and connecting flowlines show the direction of data flow.

illegal character
A character that is not valid according to the DIBOL-83 design rules.

indexed files
Indexed files are Indexed Sequential Access Method files.

input
Data flowing into the computer.

input/output
Either input or output, or both. I/O.

jump
A departure from the normal sequence of executing instructions in a computer.

justify
The process of positioning data in a field whose size is larger than the data. In alpha fields, the data is left-justified and any remaining positions are space-filled; in numeric fields, the digits are right-justified and any remaining positions to the left are zero-filled.

key
One or more fields within a record used to match or sort a file. If a file is to be arranged by customer name, then the field that contains the customers' names is the key field. In a sort operation, the key fields of two records are compared and the records are resequenced when necessary.

keyword
A part of a command operand that consists of a specific character string.

location
Any place where data may be stored.

loop
A sequence of instructions that is executed repeatedly until a terminal condition prevails. A commonly used programming technique in processing data records.

machine-level programming
Programming using a sequence of binary instructions in a form executable by the computer.

mass storage device
A device having large storage capacity.

master file
A data file that is either relatively permanent or that is treated as an authority in a particular job.

memory
The computer's primary internal storage.

merge
To combine records from two or more similarly ordered strings into another string that is arranged in the same order. The latter phases of a sort operation.

mnemonic
Brief identifiers which are easy to remember. Example: ch.

mode
A designation used in OPEN statements to indicate the purpose for which a file was opened or to indicate the input/output device being used.

modulo
A condition where the specified number exceeds the maximum condition in a variable. The maximum allowable number is then subtracted from the specified number, and the remainder is used by the processor. In modulo 16, if 17 were specified (maximum is 15), 16 would be subtracted from 17 and the processor would use 1 as the value.

nest
To embed subroutines, loops, or data in other subroutines or programs.

object program
A file which is output by the compiler or assembler.

output
Data flowing out of the computer.

parameter
A variable that is given a constant value for a specific purpose or process.

primary key
See key.

pushdown stack
A list of items where the last item entered becomes the first item in the list and where the relative position of the other items is pushed back one.

random access
Same as to direct access.

RECORD
A statement that reserves memory.

record redefinition
The technique of specifying several different record formats for the same data. Special rules apply.

screen column number
The number which indicates the order of the vertical lines on thescreen.

screen line number
The number which indicates the order of the horizontal lines on the screen.

sequential operation
Operations performed, one after the other.

serial access
The process of getting data from, or putting data into, storage, where the access time is dependent upon the location of the data most recently obtained or placed in storage.

sign
Indicates whether a number is negative or positive.

significant digit
A digit that is needed or recognized for a specified purpose.

source program
A program written in the DIBOL-83 language.

statement
An instruction in a source program.

string
A connected linear sequence of characters.

subscript
A designation which clarifies the particular parts (characters, values, records) within a larger grouping or array.

syntax
The rules governing the structure of a language.

system configuration
The combination of hardware and software that make up a usable computer system.

trappable error
An error condition which may be trapped.

unary operator
An operator, such as + or —, which acts upon only one variable or constant (e.g., A = —C).

variable
A quantity that can assume any one of a set of values.

variable-length record
A file in which the data records are not uniform in length. Direct access to such records is not possible.

verify
To determine if a transcription of data has been accomplished accurately.

zero fill
To fill the remaining character positions in a numeric field with zeros.

zoned decimal
A contiguous sequence of up to 18 bytes interpreted as a string of decimal digits (1 digit per byte). The sign is stored as a bit in the low order byte.

INDEX

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

DIBOL-83 Language Reference Manual
Order No. AA-U066A-TK
Includes Update Notice No. AD-U066A-T1

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. Problems with software should be reported on a Software Performance Report (SPR) form. If you require a written reply and are eligible to receive one under SPR service, submit your comments on an SPR form.

Did you find errors in this manual? If so, specify by page.

_____

_____

_____

_____

_____

_____

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

_____

_____

_____

_____

_____

_____

Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

_____

_____

_____

_____

_____

_____

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Non-programmer interested in computer concepts and capabilities

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code or Country _____

Please cut along this line.

—Do Not Tear - Fold Here and Tape —

digital

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Applied Commercial Engineering MK1-2/H32
Continental Boulevard
Merrimack N.H. 03054

ATTN: Documentation Supervisor

— — — Do Not Tear - Fold Here and Tape — — — — — — — — — — — — — — — — — —

Cut Along Dotted Line