dpANS DIBOL
Language Reference Manual

Order Number: AA–KZ36A–TK

March 1988

Supersession Information:	This is a new manual.
Operating System:	VAX/VMS Version 5.0 or later RSTS/E DIBOL Version 9.5 or later
Software Version:	Applicable to all products containing dpANS DIBOL.

digital equipment corporation
maynard, massachusetts

March 1988

ALL-IN-1	DECtype	LN03	Q-bus	ULTRIX-32m
A-to-Z	DECUS	LVP16	Rainbow	UNIBUS
COMPACTape	DIBOL	LQP02	RDB/VMS	VAX
COS-310	DIBOL-11	LQP03	ReGIS	VAX CDD
CTS-300	DIBOL-83	MASSBUS	RMS-11	VAXcluster
DATASYSTEM	DMS	MicroPDP-11	RSTS	VMS
DEC	FMS	Micro/RSTS	RSTS/E	VNX
DECdx	FMS-11	Micro/RSX	RSX	VT100
DECFORM	GOLD KEY	MicroVAX	RSX-11	VT125
DECgraph	J-11	MicroVAX I	RSX-11M	VT220
DECmail	LA50	MicroVAX II	RSX-11M-PLUS	VT240
DECmate	LA100	MicroVMS	RT-11	VT241
DECnet	LA120	PDP-11	RX50	WPS
DECprinter	LA210	P/OS	TK50	WPS-8
DECspell	Letterprinter	Professional	ULTRIX-11	
DECsupport	Letterwriter	PRO/RT	ULTRIX-32	

54923

Contents

Preface

The dpANS DIBOL Language Reference Manual contains reference information on all aspects of the dpANS Proposed Standard for the DIBOL Programming Language. It does not include information on any particular operating systems or their specific effect on DIBOL.

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 dpANS DIBOL language elements such as the character set, statement types, program structure, syntax, labels, literals, and expressions.

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

Chapter 3 references all the Procedure Division statements, explains the Value Assignment Statements, and array subscripting.

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

Chapter 5 references all dpANS DIBOL External Subroutines.

Appendix A contains the Character Set for the dpANS Proposed DIBOL Standard.

Appendix B contains information on Error Handling.

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

MANUAL FORMAT

This manual provides the reader with fast information retrieval.

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 makeup 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 numeric expression that evaluates to an input/output channel number.
nexp	is a numeric expression that can be any valid combination of operands and operators. In its simplest cases, *nexp* can be a *nfield* or a *nliteral.*
nfield	is the name of a numeric field.
nliteral	is a numeric literal.
field	is the name of either an alpha or a numeric field.
label	is a Procedure Division statement label.
literal	is either an alpha or a numeric 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 braces and mean optional arguments.
[ ]	represent brackets and and mean a single choice **must** be made from a list of arguments.

Symbol	Definition
. . .	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

dpANS DIBOL Language Elements

This chapter contains information on the DIBOL Character Set, the various DIBOL statement types, program structure, statement line syntax, labels, literals, and expressions.

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

1.1 DIBOL Character Set

A DIBOL program consists of symbolic characters that form the elements of the language. A subset of the American Standard Code for Information Exchange (ASCII) characters comprises 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 language.

DIBOL 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 runtime.
- Most Compiler Directives may appear anywhere in the program.
- Declarations are limited to either the Data Division (SUBROUTINE) or Procedure Division (BEGIN-END or PROC-END). They are discussed in the chapters devoted to those respective program divisions.
- The Compiler Directives are:

.END	identifies the end of the Procedure Division.
.IFDEF-.ELSE-.ENDC	specifies conditional compilation based on the presence of a preceding definition of a named variable within the current compilation.
.IFNDEF-.ELSE-.ENDC	specifies conditional compilation based on the absence of a preceding definition of a named variable within the current compilation.
.INCLUDE	causes the compiler to open a specified file and continue the compilation using that file.
.LIST	enables the compiler to list source code.
.MAIN	identifies the beginning of the Data Division of the main program.
.NOLIST	inhibits the listing of compiler source code.
.PAGE	terminates the current listing page and begins a new listing page.
.PROC	identifies the beginning of the Procedure Division.
.SUBROUTINE	identifies the beginning of a source program that is an external subroutine.
.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. A BEGIN-END sequence of statements may be used anywhere a single statement may be used.
PROC-END	separates the 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. SUBROUTINE may be used instead of .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 program.
- The Data Specification Statements are:

COMMON	defines an area of memory where variable data is stored. This area can be accessed by both main program and external subroutines.
RECORD	defines an area of memory where variable data is stored. This area is accessible only by the declaring program.
field definition	describes the name, array count, data type, size, and initial value of a field in a RECORD or COMMON area.

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.
DECR	decreases a variable by one.
INCR	increases 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	provided for ANS compatibility. DETACH has no effect in VAX DIBOL.
DO-UNTIL	causes repetitive execution of a statement until a condition is true.
EXIT	terminates execution of a BEGIN-END block.
EXITLOOP	terminates execution within an iterative construct (FOR, DO-UNTIL, REPEAT, or WHILE) and transfers control to the statement immediately following the construct.
FOR	causes repetitive execution of a statement.
GOTO	transfers control to another statement.
GOTO (computed)	conditionally transfers program control based on the evaluation of an expression.
IF	executes a statement if a condition is true.
IF-THEN-ELSE	allows conditional execution of one of two statements.
NEXTLOOP	terminates execution with an iterative construct (DO-UNTIL, FOR, REPEAT, or WHILE) and begins executing the next iteration, if any, of the iterative construct.
OFFERROR	disables trapping of run-time errors.
ONERROR	enables trapping of run-time errors.
REPEAT	repetitively executes a statement.
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.
XRETURN	transfers program control to the statement logically following the XCALL statement that transferred control to the current external subroutine.

1.2.5 Intertask Communications Statements

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

LPQUE	requests a file to be printed.
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.
OPEN	initializes a file in preparation for input /output operations.
READ (Indexed File)	reads a record from an indexed file.
READ (Relative File)	reads a record from a relative file.

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 (Indexed File)	writes a record to an indexed file.
WRITE (Relative File)	writes a record to a relative file.
WRITES	writes the next record in sequence to a file.

1.3 Program Structure

A DIBOL program may contain two major parts: an optional 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 dpANS DIBOL program structure.

Figure 1–1: dpANS DIBOL Program Structure

```
Main Program
      RECORD statement 1
            field definitions
                  .                                       Data Division
                  .
                  .
      RECORD statement n
            field definitions
                  .                                  Procedure Division
                  .
                  .
      PROC
        .
        .
        .
      END
External Subroutine
      SUBROUTINE statement
            argument definitions
      RECORD statement n
            field definitions
                  .
                  .
                  .
      PROC
        .                                            Procedure Division
        .
        .
      END
```

MK-02719-00

1.4 Statement Line Syntax

General Rules

- Each division of a DIBOL program has one or more logical lines.
- A logical line consists of a physical line (data record) and can be followed by one or more continuation lines.
- A logical line cannot exceed 1023 characters in length. A physical line cannot exceed 255 characters in length.
- A program may contain no more than one statement per logical 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 nonspacing character position in the continuation line.
- A statement can be continued until it exceeds the limit of a logical line (which can contain 1023 characters including ampersand symbols, spaces, horizontal tabs, Carriage Return, and Line Feed characters).
- Comments **cannot** be continued by an ampersand. They must be preceded by a semicolon on each physical line.

Rules for Delimiters

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

Table 1–1: dpANS DIBOL Delimiters

Name	Symbol	Name	Symbol
Addition	+	Percent	%
Colon	:	Period	.
Comma	,	Pound	#
Division	/	Right Parenthesis	)
Double Quotes	"	Single Quote	'

Table 1–1 (Cont.): dpANS DIBOL Delimiters

Name	Symbol	Name	Symbol
Equal	=	Space	
Left Parenthesis	(	Subtraction	–
Multiplication	*	Tabs	<TAB>
Boolean and	.AND.	Relational equal	.EQ.
Boolean or	.OR.	Relational not equal	.NE.
Boolean xor	.XOR.	Relational greater than	.GT.
Boolean not	.NOT.	Relational greater than or equal to	.GE.
		Relational less than	.LT.
		Relational less than or equal to	.LE.

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 by themselves (full line comments).
- Comments **cannot** be continued by an ampersand. They must be preceded by a semicolon on each physical line.

Rules for Spacing Characters

- Spacing characters are spaces or horizontal tabs not contained in an alpha literal.
- Adjacent spacing characters occupy one character of a logical line.
- Spacing characters at either the beginning or end of a physical line are ignored and not considered part of a logical line.

Rules for Blank Lines

- A blank line is a physical line that contains no compilation information.
- Any number of blank lines may be placed between logical lines.
- A blank line cannot precede a continuation line.

Run-Time Error Conditions

None

Examples

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

Definition

A statement label is a unique symbolic name that identifies a statement in the Procedure Division of a DIBOL program.

Format

label,{statement}

label
is the statement label.

statement
is a DIBOL statement.

General Rules

- A *label* consists of up to 30 characters, the first of which must be alphabetic. The remaining characters can be alphabetic, numeric, dollar sign ("$"), or underscore ("_").
- 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* can be on a line by itself.
- A *label* cannot be used to identify more than one *statement.*
- Compiler Directives and Declarations (except for BEGIN-END) cannot have *labels.*

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 is legal but will be truncated to 30 characters (i.e., DO_PAYROLL_ON_THE_DAY_THAT_BEG):

```
DO_PAYROLL_ON_THE_DAY_THAT_BEGINS_FISCAL_YEAR, 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 Literals

Definition

Literals are alpha or numeric 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 apostrophes (') 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.

- Numeric literals can be any valid DIBOL number that does not exceed 18 digits.
- Literals can be used as passed arguments to subroutines but cannot be altered by the subroutine.

Run-Time Error Conditions

None

Examples

The following numbers are all legal numeric 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"
```

1.6.1 Error Mnemonics

Appendix B lists the dpANS standard error mnemonics for error conditions. Error mnemonics are treated as symbolic representations of numeric literals and can be used wherever a numeric literal is allowed.

1.7 Expressions

Definition

An expression is a construct composed of one operand and an optional unary operator or two operands joined by a binary operator. The operands of expressions may themselves be expressions.

General Rules

- The result of a resolved numeric expression shall replace all components of the expression and shall be treated as a single operand for any remaining phases of the expression evaluation.
- Expressions are typed after the data type of their value. Thus, there are two classes of expressions: numeric and alpha.

1.7.1 Alpha Expressions

Definition

An alpha expression is either an alpha variable or an alpha literal.

General Rules

- The value of an alpha expression is the character string defined by the field or literal.
- The truth value of an alpha expression is FALSE if all characters contained in the expression are ASCII spaces.
- The truth value of an alpha expression is TRUE if any of the characters contained in the expression are not an ASCII space.

1.7.2 Numeric Expressions

Definition

Numeric expressions are valid combinations of operands and operators.

General Rules

- If X and Y are operands, the following are numeric expressions:
  - X binary operator Y
  - unary operator X
  - (X)
- A numeric operand is a numeric field, numeric literal, or expression.
- The value of a numeric expression is the numeric result of the operations indicated by the operators specified within the expression.
- Operators in a numeric expression represent various arithmetic, relational, or Boolean functions of the dpANS DIBOL language.
- Unary operators require one operand.
- Binary operators require two operands.
- Operators require operands to be the correct data type.
- Numeric expressions are evaluated according to the order of precedence. 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 numeric expression in which they appear. Additional levels of precedence are achieved by nesting; the innermost expressions are evaluated first.
- A character within a numeric field that is a space shall be treated exactly as if it were a zero by all operators acting on that field.
- When the rightmost character of a numeric field contains a lowercase 'p' (the minus zero value), and all other characters are either zeroes or spaces, the field shall be considered to have a value of zero.
- The treatment of a character within a numeric field which is not a numeric digit, a blank, or a lowercase "p" through lowercase "y" alpha letter in the rightmost character position, is undefined.
- The result of a numeric expression cannot be minus zero.

- The truth value of a numeric expression shall be FALSE if the actual value of the expression is zero, and TRUE if the actual value of the expression is non-zero.
- The maximum size of the resolved value of any numeric expression shall be considered to be 18 digits for the purposes of subsequent operations.

Rules for +, –, *, and /

- Numeric expressions deal with integers only. So output data can be correctly formatted for printing, the position of an implied decimal point in a numeric value must be determined by the program.
- Numeric 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 and results in error $ERR_DIVIDE ATTEMPT TO DIVIDE BY ZERO.
- 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 numeric 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 one if the digit to its right is greater than or equal to five.

Rules for Relational Operators

- Relational operators are .EQ., .NE., .GT., .LT., .GE., and .LE.
- Relational expressions produce numeric 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., numeric/numeric 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.

Rules for Boolean Operators

- Boolean operators are .AND., .OR., and .XOR.
- The operands of binary Boolean operators are numeric expressions.
- The operand of the Boolean operator (.NOT.) may be an alpha or numeric expression.
- Boolean operators guarantee left-to-right evaluation. If the result is known from the evaluation of the left operand, the right operand will not be evaluated.

Run-Time Error Conditions

$ERR_BIGNUM	E	Arithmetic operand exceeds 18 digits
$ERR_DIVIDE	E	Attempt to divide by zero

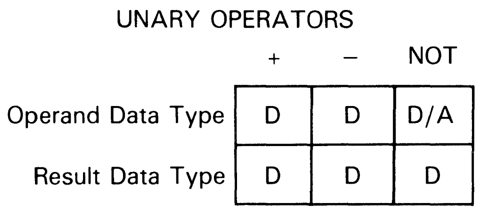
Table 1–2: Table of Operator Precedence

(from highest to lowest)

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

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.

Table 1–3: Unary Operator Table



UNARY OPERATORS

	+	–	NOT
Operand Data Type	D	D	D/A
Result Data Type	D	D	D

MK-02737-00

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

Table 1–4: Binary Operator Table

	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

MK-02738-00

The following Truth Table illustrates how truth values are determined for .OR., .AND., .XOR., and .NOT.:

Table 1–5: Truth Table

.AND.				.OR.				.XOR.				.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			

MK-02740-00

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
       E,      D5.3, 12.300
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 numeric 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

This chapter contains information on Data Division statements.

The Data Division is the first division of a dpANS DIBOL 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 and argument definitions if the program is an external subroutine.

The Data Division in a main program begins with a .MAIN compiler directive. The Data Division in an external subroutine begins with a .SUBROUTINE compiler directive. The Data Division is terminated by a .PROC compiler directive.

2.1 Record Statement

Function

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

Format

RECORD *{name}{,***X***}*

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 main program.
- RECORD must be followed by at least one field definition.
- The total size of a record is the sum of the sizes of its fields.
- 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 30 characters, the first of which must be alphabetic. Remaining characters can be alphabetic, numeric, $, or _(underscore).
- A *name* cannot be used to identify more than one 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.
- RECORD can redefine RECORD or COMMON.
- When the redefinition indicator is used, the RECORD statement redefines the memory area defined by the immediately preceding RECORD or COMMON statement not having a redefinition indicator.
- A 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 an external subroutine, the size of a RECORD redefining a COMMON area may be greater than the size of the COMMON being redefined.
- If a COMMON in an external subroutine is redefined by a RECORD, the COMMON has to be named.
- If a named COMMON is redefined in an external subroutine by a RECORD, the RECORD begins at the position of the redefined COMMON's name in the main program.
- Fields in a redefining RECORD cannot be assigned initial values.

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
```

2.2 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**}

name
is the COMMON name.

X
is the redefinition indicator.

General Rules

- COMMON must be followed by at least one field definition.
- Storage is allocated contiguously in memory in the order the COMMON statements appear in the main program.
- The size of the allocated memory area in the main program is the sum of the sizes of all the fields that comprise the COMMON in the main program.
- 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 will 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 30 characters for VAX DIBOL, 5 for PDP–11 DIBOL, the first of which must be alphabetic. Remaining characters can be alphabetic, numeric, $, or _(underscore).
- A *name* cannot be used to identify more than one RECORD area, COMMON area, or field.
- If a COMMON *name* is not specified, only named fields within that COMMON area can be referenced.

Rules for Redefinition Indicator

- The redefinition indicator permits redefinition of fields within the record being redefined.
- When the redefinition indicator is used in a main program, the COMMON statement redefines the memory area defined by the immediately preceding RECORD or COMMON statement not having a redefinition indicator.
- 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.
- In an external subroutine, the redefinition indicator on COMMON is ignored.
- Fields in a redefining COMMON cannot be assigned initial values.

Run-Time Conditions

None

Examples

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

```
COMMON REC6

COMMON A_REC

COMMON BAD$
```

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
```

2.3 Field Definitions

Function

Field definitions define variables within a RECORD or COMMON area.

Format

```
            ┌                                          ┐
            │ [ m] [ A ]  n { ,alit, . . . }           │
[ name],    │      [ D ]    { ,nlit, . . . }           │
            │      A*,alit                             │
            │      D*,nlit                             │
            └                                          ┘
```

name
is the field (or array) name.

m
is the array count.

A
declares the field to be alpha.

D
declares the field to be numeric decimal.

n
is the size of each element of the field.

*****
is the automatic sizing indicator.

alit
is the initial value for the alpha field.

nlit
is the initial value for the numeric field.

General Rules

- Each field *name* must be unique within the set of variable names used within the declaring program. The same field *name* may be used within a program and an external subroutine called by that program.

Rules for Field Name

- A field *name* in a RECORD consists of up to 30 characters, the first of which must be alphabetic. Remaining characters can be alphabetic, numeric, $, or _(underscore).
- Only the first 30 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 five characters (in PDP–11 DIBOL; 30 characters in VAX DIBOL), the first of which must be alphabetic. Remaining characters can be alphabetic, numeric, $, or _(underscore).
- Only the first five characters (in PDP–11 DIBOL; 30 in VAX DIBOL) of a field *name* in a COMMON area are significant; remaining characters are ignored.
- A *name* cannot be used to identify more than one RECORD area, COMMON area, or field.
- COMMON fields in an external subroutine may be defined in a different order than the COMMON fields in the main program.
- There must be an identically named COMMON variable in the main programs for the COMMON fields in an external subroutine.
- 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 Array Count

- The array count may be any non-zero positive numeric value.
- The array count default is one (1) unless the array count is specified.
- Array data is referenced by using the array variable name with a subscript.
- Fields defined with an array count of one are called simple variables.
- Fields defined with an array count of more than one are called arrays.

Rules for Field Size

- The minimum field size is one (1).
- The maximum field size for alpha fields is 16,383.
- The maximum field size for numeric fields is 18.

Rules for Automatic Sizing Indicator

- The initial value must be specified.
- The size of the field will be the length of either *alit* for alpha fields or *nlit* for numeric fields.
- The auto size indicator cannot be used when an array count is specified.

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 numeric literals).
- Leading signs (+ and –) in numeric literals, as well as delimiting apostrophes or quotation marks 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 numeric field.
- Initial values for COMMON fields are ignored in an external subroutine.
- Fields within an array may be initialized by specifying a series of initial values separated from each other by commas.
- It is unnecessary to initialize all fields of an array. Initialized array fields will reside at the beginning of the array and will be contiguous.
- Trailing unary operators (+/–) may be specified on *nlit*. This feature may be deleted from future standards.

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 record contains both named and unnamed fields. The three unnamed fields all have initial values (named fields can also have initial values). The third field is a two 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 two decimal fields which have initial values. The first field (LINE) is a two 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
```

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'
```

2.4 SUBROUTINE Statement

Function

SUBROUTINE identifies a program as an external subroutine.

Format

SUBROUTINE *name*

name
is the subroutine name.

Rules

- SUBROUTINE is identical in function to .SUBROUTINE. SUBROUTINE may be deleted from future standards.
- 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.
- An external subroutine is a completely self-contained program that is external to the calling program. An external subroutine is compiled separately from the calling program.
- SUBROUTINE may be followed by one or more argument definitions.

Rules for Subroutine Name

- A subroutine *name* consists of up to 30 characters (VAX DIBOL; 6 for PDP–11 DIBOL), the first of which must be alphabetic. Remaining characters can be alphabetic, numeric, $, or _(underscore).
- Only the first 30 characters (VAX DIBOL; 6 for PDP–11 DIBOL) of a subroutine *name* are significant; remaining characters are ignored.

Run-Time Error Conditions

None

2.4.1 Subroutine Argument Definition

Function

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

Format

$$\textit{name,}\ \begin{bmatrix} \textbf{A} \\ \textbf{D} \end{bmatrix}$$

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

A
declares the field to be alpha.

D
declares the field to be numeric 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 data type with the arguments specified in the XCALL statement in the calling program.
- The first argument definition specified refers to the data element referenced in the first argument in the XCALL statement. The second argument definition refers to the second XCALL argument, and so on.
- The number of subroutine arguments defined in the external subroutine must be equal to, or greater than, the number of arguments defined in the corresponding XCALL in the calling routine.
- The size of a declared subroutine argument not passed by the XCALL statement in the calling program is –1. The only valid reference to a declared argument that does not have a corresponding passed argument is a passed argument in a call to another external subroutine.

- If the argument specified in the calling program is either a literal or a numeric expression consisting of more than a single element, the data shall be considered to be a literal. The field shall not be allowed to be modified. Array access to the field with a value greater than one is not allowed. Substring access outside the bounds of the field is not allowed.
- Null arguments passed as parameters to a subroutine will be treated as placeholders of those parameters. An **argument missing** ($ERR_ARGMIS) error will be generated if the parameter is required by the subroutine.

Rules for Subroutine Argument Name

- An argument *name* consists of up to six characters (PDP-11 DIBOL; 30 for VAX DIBOL), the first of which must be alphabetic. Remaining characters can be alphabetic, numeric, $, or _(underscore).
- Only the first six characters (PDP-11 DIBOL; 30 for VAX DIBOL) of a subroutine argument *name* are significant; remaining characters are ignored.
- A subroutine argument *name* must be unique within the set of variable names used within the external subroutine. The *name* can be identical to another routine *name* or statement *label* used within that program or external subroutine.

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
        X_DATE, A11
PROC
        XCALL CNVRT (DATE,X_DATE)  ; Convert the date
        OPEN  (1,0,'TT:')          ; Open the terminal
        WRITES (1,X_DATE)          ; 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 dpANS DIBOL Procedure Division

3.1 Introduction

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

The dpANS DIBOL Procedure Division processes data and controls program execution. It contains procedural statements, statement labels, and compiler directives. It begins immediately following the .PROC compiler directive and ends with the .END compiler directive.

3.2 Value Assignment Statements

Function

Value Assignment statements:

- Move data.
- Store the results of arithmetic expressions.
- Convert and format data.
- Set destination values equal to explicit variables.

Format

destination = {source}

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 numeric.
- The *source* data is always converted to the data type defined for the *destination.*
- If the *source* is not specified, the alpha *destination* is set to spaces and the numeric *destination* is set to zeros.

3.2.1 Moving Alpha Data

Function

Value assignment statements move alpha data.

Format

name, $\begin{bmatrix} \textit{afield1} \\ \textit{aliteral} \end{bmatrix}$

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

$\begin{bmatrix} \textbf{\textit{aliteral}} \\ \textbf{\textit{afield1}} \end{bmatrix}$
is an alpha field, alpha literal, or record which is the source.

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.

Run-Time Error Conditions

$ERR_WRTLIT	F	Attempt to store data in a literal

Examples

In the following example, NAME2, which contains 'Johnson', is moved to NAME1. Since NAME1 is only four characters long, only the first four 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, 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 Numeric Data

Function

Value assignment statements move numeric data.

Format

nfield = nexp

nfield
is a numeric field which is the destination.

nexp
is a numeric 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.

Run-Time Error Conditions

$ERR_WRTLIT	F	Attempt to store data in a literal

Examples

In the following example, 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, 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,
        C,      D10
PROC
        C=A
```

In the following example, the result of A*B (1234*34=41956) is moved to C. Since C is only four digits long, 41956 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=A*B
```

3.2.3 Alpha-to-Numeric Conversion

Function

Value assignment statements convert alpha data to its numeric equivalent.

Format

$$\mathit{nfield},\ \left[\begin{array}{l} \mathit{afield1} \\ \mathit{aliteral} \end{array}\right]$$

nfield
is a numeric field which is the destination.

$\left[\begin{array}{l}\textbf{\textit{afield1}}\\ \textbf{\textit{aliteral}}\end{array}\right]$
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. No other characters are allowed.
- 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.

Run-Time Error Conditions

$ERR_DIGIT	E	Bad digit encountered
$ERR_WRTLIT	F	Attempt to store data in a literal

Examples

In the following example, 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 0111213141.

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

In the following example, 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, 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 Numeric-to-Alpha Conversion

Function

Value assignment statements convert numeric data to its alpha equivalent.

Format

afield = nexp

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

nexp
is a numeric 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.

Run-Time Error Conditions

$ERR_WRTLIT	F	Attempt to store data in a literal

Examples

In the following example, 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, 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, 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, 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, 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 numeric field can have a negative value, space must be made for the sign in the alpha field. In the following example, 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 numeric data to be converted to its alpha equivalent and formatted.

Format

afield = nexp, format_string

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

nexp
is a numeric expression which is the source.

format_string
is an alpha field, alpha literal, or record which contains format control characters.

Rules

- The source is formatted according to the *format_string* and moved to the destination.
- 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 dpANS DIBOL characters (except for the format control characters themselves) that are to be inserted in the formatted data. All non-format control characters are moved to the corresponding position in *afield.*
- 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.
- If *nexp* is zero and the field is zero suppressed, all format control characters except asterisk (*) are suppressed.

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. Any Z or * format character to the right of an X is considered to be an 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 zero.
*	Each asterisk (*) represents a digit position. It replaces a leading zero with an * symbol in this position.

Table 3–1 (Cont.): Format Control Characters

Character	Description
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. Non-money characters to the left of the money sign are considered as non-format control characters except for embedded commas. 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.

NOTE

The following descriptions on the decimal point (.) and comma (,) are reversed when international data formatting is selected via the FLAGS external subroutine.

Character	Description
	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.
	If an asterisk precedes a comma and the position corresponding to the asterisk is not filled with a significant digit, the comma shall be considered an asterisk.
	If a Z precedes a comma and the position corresponding to the Z is not filled with a significant digit, the comma shall be considered a Z.
	If a money sign precedes a comma and the position corresponding to the money sign is not filled with a significant digit, the comma shall be considered to be a money sign.

Run-Time Error Conditions

$ERR_WRTLIT	F	Attempt to store data in 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, '$$$$$$$$'	' $123'
F= -1123, '-XXX,XXX'	'-001,123'
F= 123, '$$$$$.XX'	' $1.23'
F= -123, '$***.**-'	'$**1.23-'
F= 12345678,'X,XXX.XX'	'3,456.78'
F= 1234, '$,$$$.XX'	' $12.34'
F= 1234, 'X,XXX.X'	'***12.34'
F= 1234, 'Z,ZZZ.XX'	' 12.34'

3.2.6 Clearing Variables

Function

Value assignment statements clear variables.

Format

field =

field
is an alpha field, numeric 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 numeric field, it is cleared to zeros.
- If the destination is a record containing numeric fields, the entire record, including the numeric 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.

Run-Time Error Conditions

$ERR_WRTLIT	F	Attempt to store data in 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, then 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.

3.3 Array Subscripting

Definition

Array subscripting references a specific variable within an array of variables.

Format

array (subscript)

array
is an alpha array, numeric array, or a record being referenced.

subscript
is a numeric expression that refers to a field (element) 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 one 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.
- If *subscript* is a subroutine argument that is either a literal or an expression within the calling program, the only *subscript* value allowed is one.
- A reference to an array without *subscript* accesses the first field in the array.

Run-Time Error Conditions

$ERR_SUBSCR	E	Invalid subscript specified

Examples

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

```
RECORD
        NAME,  4A3, 'LAS', 'FIR', 'MID', 'ADD'
        CODE,  4D4, 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

3.4 Substrings

Definition

Substrings reference a portion of a variable.

Rules

- Substrings can be specified in any Procedure Division statement where a data field of the same type is allowed.
- The substring type must be the same as the reference variable.

3.4.1 Absolute Substring Specification

Definition

An absolute substring references a portion of a variable specified by a starting and ending position.

Format

field (start,end)

field
is an alpha field, numeric field, or record being referenced.

start
is a numeric expression that specifies the position of the first character of the substring.

end
is a numeric expression that specifies the position of the last character of the substring.

Rules

- The starting position must be greater than or equal to one.
- The starting position must be less than or equal to the ending position.
- 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 field is a subroutine argument that is a literal or an expression in the calling program, end cannot be greater than the length of field.

Run-Time Error Conditions

$ERR_BIGNUM	E	Arithmetic operand exceeds 18 digits
$ERR_SUBSCR	E	Invalid subscript specified

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 absolute 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)	478
NUM(10,10)	0
NZ(12,13)	yz

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

3.4.2 Relative Substring Specification

Definition

A relative substring references a portion of a variable specified by a starting position and a length.

Format

field(pos:length)

field
is an alpha field, numeric field, or record being referenced.

pos
is a numeric expression that specifies the first or last character of the substring.

length
is a numeric expression that specifies the substring.

Rules

- *length* cannot equal zero.
- *pos* must be a positive integer.
- *pos* must be greater than or equal to one.
- If the value of *length* is positive, the substring begins at *pos* and is *length* (absolute value) characters in length.
- If the value of *length* is negative, the substring ends at *pos* and is *length* (absolute value) characters in length.
- If the relative specification exceeds the declared size of *field,* portions of adjacent fields may be referenced. A **Subscript error** occurs when memory outside the Data Division that defines *name* is referenced.

- If *field* is a subroutine argument that is a literal or an expression in the calling program, the specification cannot refer to memory that is not defined by *field*.

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
```

The following examples illustrate the use of relative substrings:

```
Field             Data Accessed

AM(2:2)           bc
AM(4:1)           d
AM(13:-4)         jklm
REC(1:10)         abcdefghij
REC(28:-2)        12
NUM(4:5)          45678
NUM(10:-1)        0
NZ(12:2)          yz
AM(NUM(5,1):5)    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(12:4)        lmno
NZ(13:3)        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(10:2)      Subscript error
NZ(30:-4)      Subscript error
```

3.5 ACCEPT

Function

ACCEPT inputs a character from a terminal.

Format

ACCEPT *(ch,field{,label})*

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

field
is an alpha field, numeric 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

- Each character is received individually using the ACCEPT statement. The program can receive either the actual ASCII character or the decimal equivalent of that character.
- ACCEPT is used in I or O mode with a character-oriented device.
- 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.
- 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 Numeric Field

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

Run-Time Error Conditions

$ERR_EOF	E	End of file encountered
$ERR_ILLCHN	F	Illegal channel number specified
$ERR_IOMODE	E	Bad mode specified
$ERR_NOOPEN	F	Channel has not been opened
$ERR_WRTLIT	F	Attempt to store data in a literal

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,ACHR,END)
```

In the next example, ACCEPT puts the decimal character code for the next character into DCHR. When accepting into a numeric 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)
```

3.6 BEGIN-END Block

Function

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

```
BEGIN
  {statement
   .
   .
   .}
END
```

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.

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 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
```

3.7 CALL

Function

CALL transfers program control to an internal subroutine.

Format

CALL *label*

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.
- *label* must be defined within the current program.

Run-Time Error Conditions

$ERR_SYSTEM	F	System error

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=PBT*8                        ; Compute the tax
        IF TAX.GT.MAX TAX=MAX
        RETURN
```

3.8 CLEAR

Function

CLEAR sets variables to zeros or spaces.

Format

CLEAR *field{, . . . }*

field
is an alpha field, numeric field, or record to be cleared.

Rules

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

Run-Time Error Conditions

$ERR_WRTLIT	F	Attempt to store data in 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 numeric fields within the record, are cleared to spaces. The following statement will clear AFLD and DFLD to spaces:

```
CLEAR REC
```

3.9 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*

ch
is a numeric 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, 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.

Run-Time Error Conditions

$ERR_FILFUL	E	Output file is full
$ERR_ILLCHN	F	Illegal channel number specified
$ERR_IOFAIL	E	Bad data encountered during I/O operation
$ERR_LOCKED	E	Record is 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
```

3.10 DECR

Function

DECR decreases a numeric field by one.

Format

DECR *nvar*

nvar
is the field to be decreased.

Rules

- DECR is functionally equivalent to *nvar=nvar* – 1 but will not cause an overflow error if the value exceeds 18 digits.
- Underflow will result in *nvar* being set to minus zero.
- The field to be decreased can contain positive or negative numbers.
- If the size of the resulting value is larger than *nvar,* the leftmost digits causing overflow are truncated.

Run-Time Error Conditions

$ERR_WRTLIT	F	Attempt to store data in a literal

Examples

The following DECR statements are all valid (assuming that the fields being decreased are all numeric).

```
DECR CNTR

DECR A(3)

DECR C(H,6)

IF LNECTR.GT.MINTR DECR LINECTR
```

3.11 DELETE

Function

DELETE eliminates a record from an indexed file.

Format

DELETE *(ch,{keyfld})*

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

keyfld
is ignored.

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 clears any lock on the specified channel.
- DELETE serves as a signal to the file system that the record is no longer valid. The action taken is system dependent.

Run-Time Error Conditions

$ERR_ILLCHN	F	Illegal channel number specified
$ERR_IOFAIL	E	Bad data encountered during I/O operation
$ERR_IOMODE	E	Bad mode specified
$ERR_KEYNOT	E	Key not same
$ERR_LOCKED	E	Record is locked
$ERR_NOCURR	E	No current record
$ERR_NOOPEN	F	Channel has not been opened

Examples

In the following example, all 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
```

3.12 DETACH

Function

DETACH disconnects the program from its associated terminal.

Format

DETACH

Rules for PDP

- 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.

Rules for VAX

- DETACH is non-operative.

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
```

3.13 DISPLAY

Function

DISPLAY outputs (8-bit ASCII) characters to a device or file.

Format

$$\textbf{DISPLAY}\ \ (ch, \begin{bmatrix} afield \\ aliteral \\ nexp \end{bmatrix} \{, \ldots \})$$

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

$$\begin{bmatrix} \boldsymbol{afield} \\ \boldsymbol{aliteral} \\ \boldsymbol{nexp} \end{bmatrix}$$
is an alpha field, alpha literal, or numeric expression which contains characters to be output.

Rules

- DISPLAY is used in O:P mode with a sequential file in I and O modes with a printer or character-oriented device.
- DISPLAY uses the numeric ASCII character code (see APPENDIX A).
- If the data is alpha, the characters are output to the device as presented.
- If the data is numeric, it is evaluated modulo 256 and the number is treated as a single ASCII 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.

Run-Time Error Conditions

$ERR_ILLCHN	F	Illegal chain number specified
$ERR_IOMODE	E	Bad mode specified
$ERR_NOOPEN	F	Channel has not been opened
$ERR_OUTRNG	F	Value out of range

Examples

The following example outputs the message HELLO followed by a carriage return (numeric character code 13) and a Line Feed (numeric 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')
```

3.14 DO-UNTIL

Function

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

Format

DO *statement* **UNTIL** *condition*

statement
is a DIBOL Procedure Division statement.

condition
is a numeric 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.

Run-Time Error Conditions

None

Examples

In the following example, customer records (CUST) will be read until one is found with a balance (BAL) 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
```

3.15 EXIT

Function

EXIT terminates the execution of a BEGIN-END block.

Format

EXIT

Rules

- EXIT is specified within a BEGIN-END block.
- Control is transferred to the END statement of the current BEGIN-END block.

Examples

The following program segment reads customer records (CUST) and creates a list of customers with a balance over $1000. No entry is made in the list if the customer is allowed an extended line of credit (CREDIT).

```
DO
   BEGIN
   READS (1,CUST,EOF)
   IF BALANC .GT. 1000
       BEGIN
       IF CUSTYP .EQ. 'CREDIT' EXIT
       NAME = CUSNAM
       AMT = BALANC
       WRITES (6,PLINE)
       END
   END
UNTIL CUSNAM .EQ. SPACES
```

3.16 EXITLOOP

Function

EXITLOOP terminates execution within an iterative construct (FOR, DO-UNTIL, REPEAT, or WHILE) and transfers program control to the statement immediately following the iterative construct.

Format

EXITLOOP

Rules

- EXITLOOP must be physically contained within an iterative construct.

Examples

The following program segment totals month to date sales (MDTSLS). The loop is exited if sales for any month are negative.

```
CLEAR YTDSLS
FOR MONTH FROM 1 THRU 12
    BEGIN
    IF MTDSLS (MONTH) .LT. 0
    THEN EXITLOOP
    ELSE YTDSLS = YTDSLS + MTDSLS
```

3.17 FOR

Function

FOR repetitively executes a statement.

Format

FOR *nfield* **FROM** *initial* **THRU** *final* {**BY** *step*} *statement*

nfield
is a numeric field to be altered.

initial
is a numeric expression which specifies the initial value to be assigned to *nfield.*

final
is a numeric expression which specifies the final value for *nfield.*

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

statement
is a DIBOL Procedure Division statement.

Rules

- 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 one.
- 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, *nfield* must be less than, or equal to, *t_final* if *t_step* is positive; and *nfield* 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, nfield* is tested to determine if it has reached its limit. If *nfield* has not reached its limit, *statement* is executed.

- If *nfield* is not large enough to hold *final* plus the *step* value without truncation, an infinite loop may occur.
- *t_step* is added to *nfield* following each *statement* execution.
- If the loop is not executed, *nfield* is equal to the *initial* value.
- If the loop is exited normally, *nfield* will equal the previous value of *nfield* 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.

Run-Time Error Conditions

$ERR_ARGMIS	E	Argument missing
$ERR_BIGNUM	E	Arithmetic operand exceeds 18 digits

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 two 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
```

3.18 FORMS

Function

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

Format

FORMS *(ch,nexp)*

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

nexp
is a numeric 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 character-oriented device, 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).

Run-Time Error Conditions

$ERR_ILLCHN	F	Illegal channel number specified
$ERR_IOFAIL	E	Bad data encountered during I/O operation
$ERR_OUTRNG	F	Value out of range
$ERR_NOOPEN	F	Channel has not been opened

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)
```

3.19 GOTO

Function

An unconditional GOTO transfers program control.

Format

GOTO *(label)*

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

Rules

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

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
```

3.20 GOTO (Computed)

Function

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

Format

GOTO *(label{, . . . }),nexp*

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

nexp
is a numeric 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 *nexp* is one, to the statement identified by the second *label* if *nexp* is two, and so on.
- If *nexp* is negative, zero, or greater than the number of labels, control is transferred to the next logical statement in sequence.

Run-Time Error Conditions

None

Examples

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

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

3.21 IF

Function

IF executes a statement if a condition is true.

Format

IF *condition statement*

condition
is an 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.

Run-Time Error Conditions

None

Examples

In an alpha comparison, the operands are compared on a character basis from left to right according to the value of 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 three character alpha field to a five character alpha field. Since only the first three characters are compared, the result of the following statement is true:

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

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
```

3.22 IF-THEN-ELSE

Function

IF-THEN-ELSE executes one of two statements based on a condition.

Format

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

condition
is an 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.

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
```

3.23 INCR

Function

INCR increases a numeric field by 1.

Format

INCR *nfield*

nfield
is a numeric 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 *nfield,* the leftmost digits that cause overflow are truncated.

Run-Time Error Conditions

$ERR_WRTLIT	F	Attempt to store data in a literal

Examples

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

```
INCR CNTR

INCR A(3)

INCR C(H,6)

IF LNECTR.LT.MAXCTR INCR LNECTR
```

3.24 LOCASE

Function

LOCASE converts uppercase characters to corresponding lowercase characters.

Format

LOCASE *afield*

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

Rules

- LOCASE will convert each byte encountered in *afield* from an uppercase character to a corresponding lowercase character if the numeric ASCII value of the byte is between 65 and 90, inclusive. These characters represent the English uppercase alphabetic characters.
- LOCASE will convert each byte encountered in *afield* from an uppercase character to a corresponding lowercase character if the numeric ASCII value of the byte is between 192 and 222, inclusive. These characters represent the Multinational uppercase alphabetic characters.
- Other non-alphabetic characters are unaffected.

Run-Time Error Conditions

$ERR_WRTLIT	F	Attempt to store data in 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]'. 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
        LOCASE A(2,14)
        LOCASE REC
        STOP
```

3.25 LPQUE

Function

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

Format

LPQUE *(filespec{,***LPNUM:***dexp} {,***COPIES:***dexp}*

*{,***FORM:** $\left[\begin{array}{l} \textit{afield} \\ \textit{aliteral} \end{array}\right]$ *}{,***DELETE***{:nexp}})*

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

LPNUM:***nexp***
is a numeric expression that specifies the printer.

COPIES:***nexp***
is a numeric expression which specifies the number of copies to print.

FORM: $\left[\begin{array}{l} \textbf{\textit{afield}} \\ \textbf{\textit{aliteral}} \end{array}\right]$
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***{:nexp}***
is the deletion indicator and is a numeric expression which specifies whether or not the file is to be deleted.

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) is (are) used.
- If the deletion indicator is zero, the file is deleted.
- If the deletion indicator is non-zero, the file is not deleted.
- If the deletion indicator is not specified, the file is deleted.

- If no copy count is specified, or if it is less than one, it is assumed to be 1.
- If a form is specified, a system specific forms request is issued.

Run-Time Error Conditions

$ERR_FNF	E	File not found
$ERR_ILLCHN	F	Illegal channel number was specified
$ERR_NOOPEN	F	Channel has not been opened
$ERR_SYSTEM	F	System error

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
```

3.26 NEXTLOOP

Function

NEXTLOOP terminates execution within an iterative construct and begins executing the next iteration, if any, of the iterative construct.

Format

NEXTLOOP

Rules

- NEXTLOOP must be physically contained within a FOR loop, DO-UNTIL loop, WHILE loop, or REPEAT loop.
- NEXTLOOP transfers control to the test condition of the immediate iterative construct with a test condition.
- For a REPEAT iterative construct, control will be passed to the statement to be repeated.

Examples

In the following example, if a character cannot be printed, NEXTLOOP terminates the loop prior to processing the character.

```
FOR COUNTER FROM 1 THRU STR_LENGTH
   BEGIN
   ACCEPT (1, IN_CHAR)
   IF (IN_CHAR .LT. '') .OR. (IN_CHAR .GT. '~') NEXTLOOP
   INCR O_CTR
   OUT_FILE (O_OCTR,O_CTR) = IN_CHAR
   END
```

3.27 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.
- OFFERROR affects only an active ONERROR.

Run-Time Error Conditions

None

Examples

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

```
ONERROR DIVO        ; Check for $ERR_DIVIDE error
C=A/B
OFFERROR            ; Turn off error check
```

3.28 ONERROR

Function

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

Format

ONERROR *label*

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.

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
```

3.29 OPEN

Function

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

Format

$$\textbf{OPEN}\quad \textit{(ch,} \left[\begin{array}{l}\textit{mode\{:submode\}}\\ \textbf{MODE:}\textit{afield}\end{array}\right] \textit{,filespec\{,}\textbf{ALLOC:}\textit{nexp\}\{,}\textbf{BKTSIZ:}\textit{nexp\}}$$

$$\textit{\{,}\textbf{BLKSIZ:}\textit{nexp\}\{,}\textbf{BUFSIZ:}\textit{nexp\}\{,}\textbf{RECSIZ:}\textit{nexp\}\{,}\textbf{NUMREC:}\textit{nexp\})}$$

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

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

submode
further defines, qualifies, or restricts *mode.*

MODE:*afield*
specifies that *mode* and *submode* will be determined at execution time. *afield* is an alpha field or literal that contains the *mode* and the optional *submode* of the form *"mode{:submode}."*

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

ALLOC:*nexp*
is a numeric expression that specifies the initial file allocation.

BKTSIZ:*nexp*
is a numeric expression that specifies the bucketsize in blocks.

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

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

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

NUMREC:*nexp*
is a numeric expression that specifies the number of logical records in lengths as defined by RECSIZE that is to be used as the initial allocation of a 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 channel number can be between 1 and 31, inclusive.
- The maximum number of channels opened simultaneously is system dependent.
- 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.
- An attempt to OPEN a file on a channel currently open will result in an error.

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.

- A character-oriented device is being opened, only the Input and Output modes of operation are used.

- *:afield* opens the channel based on the contents of *afield*. The contents of *afield* are evaluated at execution time to determine the *mode* and optional *submode* for OPEN.

Rules for *submode*

- *submodes* are Sequential (S), Print (P), Relative (R), Indexed (I), or Character (C).
- Sequential *submode* is used with O *mode* and indicates that the file being created is a sequential file. Sequential *submode* is assumed for file-oriented devices if no *submode* is specified with O *mode*.
- Character *submode* is assumed for character-oriented devices 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.
- Character *submode* may be used with I and O *modes* and indicate that the file or device is to be treated as a character-oriented device.

Rules for ALLOC

- ALLOC reserves space on a device for a file at OPEN; *nexp* is the number of 512 byte units.
- 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*.
- *nexp* must equal a non-negative integer.
- If *nexp* in NUMREC:*nexp* is non-zero, then NUMREC overrides ALLOC.

Rules for BKTSIZ

- BKTSIZ specifies at file creation time the number of 512 byte I/O units to be considered as a logical group.
- *nexp* must be a positive integer.

Rules for BLKSIZ

- BLKSIZ specifies the block size, in bytes, for files opened on magnetic tape.
- BLKSIZ is used when creating a file on magtape. Any other use of BLKSIZ is ignored.
- *nexp* must be a positive integer.

Rules for BUFSIZ

- BUFSIZ defines the size of an internal buffer.
- The I/O buffer size must be large enough to contain the data record.
- BUFSIZ overrides the buffer size designated by PROC for this OPEN.
- The value must be between 1 and 15, inclusive.

Rules for RECSIZ

- RECSIZ defines the size of a logical record with *nexp* specifying the length in bytes of each logical record in the file.
- RECSIZ is required when creating an RMS relative file. Any other use of RECSIZ is ignored.
- RECSIZ implies the records are fixed length.
- *nexp* must be a positive integer.

Rules for NUMREC

- NUMREC specifies the number of logical records in a file to be used as the initial file allocation.
- NUMREC may be used in OUTPUT *mode*. It is ignored for other *modes*.
- *nexp* must be a positive integer.
- If NUMREC is specified, RECSIZ must also be specified.
- If *nexp* is a positive, nonzero integer, NUMREC will override ALLOC.
- If *nexp* is 0, the result is the same as not specifying NUMREC.

The following tables show which statements are legal for a file organization, mode, and character device:

Table 3–2: Shared File Access

File Type	Open Mode	Other Users	Access Status
Sequential	Input	none	Granted
		Input	Granted
		Output	Denied
	Output	none	Granted
		Input	Denied
		Output	Denied
Relative	Input	none	Granted
		Input	Granted
		Output	Denied
		Update	Granted
	Output	none	Granted
		Input	Denied
		Output	Denied
		Update	Denied
	Update	none	Granted
		Input	Granted
		Output	Denied
		Update	Granted
Index	Input	none	Granted
		Input	Granted
		Update	Granted
	Update	none	Granted
		Input	Granted
		Update	Granted

Table 3–3: Valid Combinations of Mode:Submode

	I:S	O:S	I:R	O:R	U:R	I:I	U:I	I:C	O:C	O:P
ACCEPT								X	X	
CLOSE	X	X	X	X	X	X	X	X	X	X
DELETE							X			
DISPLAY								X	X	X
FORMS		X						X	X	X
READ			X	X	X	X	X			
READS	X		X	X	X	X	X	X	X	
STORE							X			
WRITE				X	X		X			
WRITES		X		X	X			X	X	X

MK-02736-00

Run-Time Error Conditions

$ERR_ALLOC	E	Invalid value specified for ALLOC:
$ERR_ARGMIS	E	Argument missing
$ERR_BKTSIZ	E	Invalid value specified for BKTSIZ:
$ERR_BUFSIZ	E	Invalid value specified for BUFSIZ:
$ERR_CHNUSE	F	Channel is in use
$ERR_DEVUSE	E	Device in use
$ERR_FILORG	E	Invalid file organization specified
$ERR_FILSPC	E	Bad file name
$ERR_FINUSE	E	File in use by another user
$ERR_FNF	E	File not found

$ERR_ILLCHN	F	Illegal channel number specified
$ERR_IOFAIL	E	Bad data encountered during I/O operation
$ERR_IOMODE	E	Bad mode specified
$ERR_NOMEM	E	Not enough memory for desired operation
$ERR_NOOPEN	F	Channel has not been opened
$ERR_NOSPAC	E	No space exists for file on disk
$ERR_NUMREC	E	Invalid value specified for NUMREC:
$ERR_ONLYRO	E	Attempt to write to a read-only device
$ERR_ONLYWR	E	Attempt to open output device in input mode
$ERR_PROTEC	E	Protection violation
$ERR_RECSIZ	E	Invalid value specified for RECSIZ:
$ERR_REPLAC	E	Cannot supersede existing file
$ERR_SYSTEM	F	System error

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:C,'TT:')
```

The following statement opens the relative file ARMAS.DDF for modification using channel 3. It also specifies an internal buffer size of three 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')
```

The following example creates a new relative file, ARMAS.DDF and associates it with channel 15. All the records in the file will be 25 characters in length. The initial file allocation is 56 blocks.

```
OPEN (15,O:R,'ARMAS.DDF',RECSIZ:25,ALLOC:56)
```

The following example creates a new relative file, ARREC.DDF and associates it with channel 21. All the records will be 80 characters in length. The initial record allocation is 100 records which will cause an initial file allocation of 15 blocks (80x100).

```
OPEN (21,O:R,'ARREC.DDF',RECSIZ:80,NUMREC:100,ALLOC:50)
```

3.30 READ (Indexed File)

Function

READ inputs a record from an indexed file.

Format

READ *(ch,record,keyfld{,***KEYNUM:***nexp})*

ch
is a numeric 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.

nexp
is a numeric expression that specifies which key of reference is to be used.

Rules

- READ is used in I and U *modes.*
- The data record read is the first one with a key value equal to the key of reference.
- If the size of *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.
- If *keyfld* is contained within the record, it is assumed to be in the same position as a key field defined for the file.
- 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.
- If the data record is larger than *record*, an **Input data size exceeds destination size** error is generated and the data record is read into *record* according to the rules for moving data to an alpha field.

- If the data record is smaller than *record*, the data record is read into *record* according to the rules for moving data to an alpha field.

Rules for KEYNUM

- KEYNUM specifies the key number to be used in a READ from an indexed file.
- KEYNUM:0 indicates that the primary key is to be used. KEYNUM:1 indicates that the first alternate key is to be used. KEYNUM:2 indicates that the second alternate key is to be used, and so on.
- If KEYNUM is not specified and the key field corresponds to a key position defined in the record, that key position determines which key number will be used.
- If KEYNUM is not specified and the key field does not correspond to a key position defined in the record, or is outside the record, the primary key number will be used.
- 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 channel is executed.
  - A STORE using the channel is executed.
  - A DELETE using the channel is executed.
  - An UNLOCK using the channel is executed.
  - A CLOSE using the channel is executed.
  - The program terminates.

Run-Time Error Conditions

$ERR_BADKEY	E	An illegal key was specified
$ERR_EOF	E	End of file encountered
$ERR_ILLCHN	F	Illegal channel number specified
$ERR_IOFAIL	E	Bad data encountered during I/O operation
$ERR_IOMODE	E	Bad mode specified
$ERR_KEYNOT	E	Key not same

$ERR_LOCKED	E	Record is locked
$ERR_NOOPEN	F	Channel has not been opened
$ERR_TOOBIG	E	Input data size exceeds destination size
$ERR_WRTLIT	F	Attempt to store data in a literal

Examples

Assuming that the indexed file has been defined with a key length of five 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 exisits, 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)
```

In the following example, the READ statement will return a record from the indexed file opened on channel 1. The READ will place that record in ADDR. Using the record definition above, KEYNAM specifies that the first alternate key be used.

```
READ (1,ADDR,ADDR(1,5),KEYNUM:1)
```

3.31 READ (Relative File)

Function

READ inputs a record from a relative file.

Format

READ *(ch,record,nexp)*

ch
is a numeric 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.

nexp
is a numeric expression that specifies the sequence number of the record to be read.

Rules

- READ is used in I and U *modes.*
- *nexp* must be between one and the total number of records in the file.
- The record is read into *record* according to the rules for moving alpha data.
- If the data record is larger than *record,* an **Input data size exceeds destination size** 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.

Run-Time Error Conditions

$ERR_EOF	E	End of file encountered
$ERR_ILLCHN	F	Illegal channel number specified
$ERR_IOFAIL	E	Bad data encountered during I/O operation
$ERR_IOMODE	E	Bad mode specified
$ERR_LOCKED	E	Record is locked
$ERR_NOOPEN	F	Channel has not been opened
$ERR_RECNUM	E	Illegal record number specified
$ERR_RNF	E	Record not found
$ERR_TOOBIG	E	Input data size exceeds destination size
$ERR_WRTLIT	F	Attempt to store data in a literal

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)
```

3.32 READS

Function

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

Format

READS *(ch,record{,label})*

ch
is a numeric 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 character-oriented device.
- The record is read into *record* according to the rules for moving alpha data.
- If the record is larger than *record,* an **Input data size exceeds destination size** error is generated and the record is read into *record* according to the rules for moving data to an alpha field.
- 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 Character-Oriented Device

- READS from a character-oriented device 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.

Run-Time Error Conditions

$ERR_EOF	E	End of file encountered
$ERR_ILLCHN	F	Illegal channel number specified
$ERR_IOFAIL	E	Bad data encountered during I/O operation
$ERR_IOMODE	E	Bad mode specified
$ERR_LOCKED	E	Record is locked
$ERR_NOOPEN	F	Channel has not been opened
$ERR_TOOBIG	E	Input data size exceeds destination size
$ERR_WRTLIT	F	Attempt to store data in a literal

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 encountered** error will be generated since no end of file label was specified. This error can be trapped by an ONERROR statement.

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

3.33 RECV

Function

RECV accepts a message which was sent by another program.

Format

RECV *(message,label{,size})*

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 numeric 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.
- If *message* is shorter than the actual message, an error will be returned, and the data is moved to *message* according to the rules for moving alpha data.
- The message size is moved into *size* according to the rules for moving numeric data.

Run-Time Error Conditions

$ERR_SYSTEM	F	System error
$ERR_TOOBIG	E	Input data size exceeds destination size
$ERR_WRTLIT	F	Attempt to store data in a literal

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
```

3.34 REPEAT

Function

REPEAT repetitively executes a statement until a condition occurs to transfer control to another statement.

Format

REPEAT *statement*

statement
is any DIBOL statement.

Rules

- *statement* is executed until control is transferred explicitly to a label or to the end of the iterative construct.

Examples

In the following example, records are read and a routine is called to process the records until the end of file is reached. When an end of file is reached, control is transferred to the label specified in the READS statement which is outside of the REPEAT block.

```
REPEAT
  BEGIN
  READS (1,FILE, DONE)
  XCALL SUB1 (FILE)
  END
DONE,
  .
  .
  .
```

3.35 RETURN

Function

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

Format

RETURN

Rules

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

Run-Time Error Conditions

$ERR_NOCALL	F	RETURN with 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.

Main Program

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

External Subroutine PROF

```
SUBROUTINE PROF
  .
  .
  .
PROC
       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=PBT*8                       ; Compute the tax
      IF TAX.GT.MAX TAX=MAX
      RETURN
```

3.36 SEND

Function

SEND transmits a message to another program.

Format

SEND *(message,program{,terminal})*

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 numeric 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

$ERR_NOMEM	E	Not enough memory for desired operation
$ERR_SYSTEM	F	System error

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'
```

3.37 SLEEP

Function

SLEEP suspends program execution for a specified period of time.

Format

SLEEP *seconds*

seconds
is a numeric 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.

Run-Time Error Conditions

$ERR_OUTRNG	F	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
```

3.38 STOP

Function

STOP terminates program execution.

Format

STOP *{filespec}*

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 an '@', 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.

Run-Time Error Conditions

$ERR_FNF	E	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'
```

3.39 STORE

Function

STORE adds a record to an indexed file.

Format

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

ch
is a numeric 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 ignored.

Rules

- STORE is used in U:I *mode.*
- If *record* is longer than the record length defined for the file, an error is generated and STORE is not performed.
- The data is moved according to the rules for moving data to an alpha field.
- 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 **Duplicate key specified** error is generated.

Run-Time Error Conditions

$ERR_BADKEY	E	An illegal key was specified
$ERR_FILFUL	E	Output file is full
$ERR_ILLCHN	F	Illegal channel number specified
$ERR_IOMODE	E	Bad mode specified
$ERR_LOCKED	E	Record is locked

$ERR_NODUPS	E	Duplicate key is specified
$ERR_NOOPEN	F	Channel has not been opened
$ERR_TOOBIG	E	Input data size exceeds destination size

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
```

3.40 UNLOCK

Function

UNLOCK clears the lock condition on a specified channel.

Format

UNLOCK *ch*

ch
is a numeric 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.
- If the specified channel is not open, a **Channel not open** error will be generated.

Run-Time Error Conditions

$ERR_ILLCHN	F	Illegal channel number specified
$ERR_NOOPEN	F	Channel has not been opened

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
```

3.41 UPCASE

Function

UPCASE converts lowercase characters to corresponding uppercase characters.

Format

UPCASE *afield*

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

Rules

- UPCASE will convert each byte encountered in *afield* from a lowercase character to a corresponding uppercase character if the numeric ASCII value of the byte is between 97 and 122, inclusive.
- UPCASE will convert each byte encountered in *afield* from a lowercase character to a corresponding uppercase character if the numeric ASCII value of the byte is between 224 and 254, inclusive.
- Other non-alphabetic characters are unaffected.

Run-Time Error Conditions

$ERR_WRTLIT	F	Attempt to store data in 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--
         .
         .
         .
```

3.42 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
```

selection_value
is an alpha field, alpha literal, decimal expression, or record.

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

$$\left[\begin{array}{l} exp \\ exp \text{ THRU } exp \end{array}\right]$$

statement
is a DIBOL Procedure Division statement.

Rules

- *selection_value* is evaluated and compared with the match expressions. Comparisons are done in the order they appear.
- *selection_value* cannot be an alpha substring.
- 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 type of *selection_value* must match the data type of the match expression (*mexp*).

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 a message indicating which case of the USING was selected:

```
RECORD
        CHARS,  D3                          ; 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
```

3.43 WHILE

Function

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

Format

WHILE *condition statement*

condition
is a numeric 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.

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
        .
        .
        .
```

3.44 WRITE (Indexed File)

Function

WRITE updates a record in an indexed file.

Format

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

ch
is a numeric 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 ignored.

Rules

- WRITE is used in U:I *mode.*
- 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.
- If record is longer than the record length defined for the file, an error condition is generated and WRITE is not performed.
- The data is moved according to the rules for moving data to an alpha field.

Run-Time Error Conditions

$ERR_BADKEY	E	An illegal key was specified
$ERR_EOF	E	End of file encountered
$ERR_ILLCHN	F	Illegal channel number specified

$ERR_IOFAIL	E	Bad data encountered during I/O operation
$ERR_IOMODE	E	Bad mode specified
$ERR_KEYNOT	E	Key not same
$ERR_NOCURR	E	No current record
$ERR_NOOPEN	F	Channel has not been opened
$ERR_TOOBIG	E	Input data size exceeds destination size

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)
```

3.45 WRITE (Relative File)

Function

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

Format

WRITE *(ch,record,nexp)*

ch
is a numeric 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 numeric 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.
- If the file is opened in output *mode* and the RECSIZ qualifier is specified when the channel is opened, and record is longer than the value specified for RECSIZ, an error condition is generated and WRITE is not performed.
- If the file is opened in update *mode* and *record* is larger than the record in the file, an error condition is generated and WRITE is not performed.
- The data is moved according to the rules for moving data to an alpha field.

Run-Time Error Conditions

$ERR_EOF	E	End of file encountered
$ERR_ILLCHN	F	Illegal channel number specified
$ERR_IOMODE	E	Bad mode specified
$ERR_IOFAIL	E	Bad data encountered during I/O operation
$ERR_LOCKED	E	Record is locked
$ERR_NOOPEN	F	Channel has not been opened
$ERR_RECNUM	E	Illegal record number specified
$ERR_TOOBIG	E	Input data size exceeds destination 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)
```

3.46 WRITES

Function

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

Format

WRITES *(ch,record)*

ch
is a numeric 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 character-oriented device 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.
- If the file is opened in update *mode* and *record* is longer than the defined record size, an error condition is generated and WRITES is not performed.
- The data is moved according to the rules for moving data to an alpha field.

Run-Time Error Conditions

$ERR_FILFUL	E	Output file is full
$ERR_ILLCHN	F	Illegal channel number specified
$ERR_IOFAIL	E	Bad data encountered during I/O operation
$ERR_IOMODE	E	Bad mode specified
$ERR_TOOBIG	E	Input data size exceeds destination size

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))
```

3.47 XCALL

Function

XCALL transfers program control to an external program.

Format

XCALL *name (arg{, . . . })*

name
is the name of the external subroutine being called.

arg
is an alpha field, alpha literal, numeric field, numeric 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 argument in the subroutine, the second is linked to the second, and so on.
- Arguments in the argument list are separated by commas.
- A given argument may be omitted from the argument list. If more arguments are needed, their place must be held by putting in the commas, e.g., XCALL SUB (A,,C).
- For numeric fields, the returned value is moved to the field according to the rules for moving numeric data.
- For alpha fields and records, the returned value is moved to the field according to the rules for moving alpha data.
- 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 or XRETURN, 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.

- 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 on PDP

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

Rules for Subroutine Name on VAX

- A subroutine name consists of up to 30 characters, the first of which must be alphabetic. Remaining characters can be alphabetic, numeric, or _ (underscore).
- Only the first 30 characters of a subroutine name are significant; remaining characters are ignored.

Run-Time Error Conditions

$ERR_SYSTEM	F	System error

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
        X_DATE, A11
PROC
        XCALL CNVRT (DATE,X_DATE)           ; Convert the date
        OPEN (1,O,'TT:')                    ; Open the terminal
        WRITES (1,X_DATE)                   ; 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 three 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
```

3.48 XRETURN

Function

XRETURN transfers program control to the statement logically following the XCALL statement that transferred control to the current external subroutine.

Format

XRETURN

Rules

- XRETURN is placed at the logical end of an external subroutine.
- XRETURN is not allowed in a main routine.
- When XRETURN is encountered within an internal subroutine of an external subroutine, the external subroutine is exited.
- XRETURN is useful within an external subroutine to distinguish between the return from an internal subroutine and the return from an external subroutine.

Examples

Main Program

```
XCALL GETONE (NAME)
WRITES (15, NAME)
. . .
```

Subroutine GETONE

```
           CALL READIT
           . . .
           XRETURN                 ; Return to main ROUTINE

READIT,    REPEAT
           BEGIN
           READS (2,BUF,EOF)
           IF BUF.EQ.NAME XRETURN  ; When one is found, return
           END                     ; ... to the main routine
EOF,       RETURN                  ; Return from internal routine
```

Chapter 4

THE dpANS DIBOL COMPILER DIRECTIVES

4.1 General Introduction

This chapter contains information about dpANS Compiler Directives. Compiler Directives are non-executable instructions to the compiler and cannot be part of any executable statement.

Compiler Directives always begin a new line and are identified with a period as their first non-spacing character. For easy reference, the Compiler Directives are arranged alphabetically.

4.2 .END

Function

.END identifies the end of the Procedure Division.

Format

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

Rules

- The Procedure Division is terminated by .END.

4.3 .IFDEF-.ELSE-.ENDC

Function

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

Format

```
.IFDEF field
   statement1
      .
      .
      .
{.ELSE
   statement2
      .
      .
      .}
.ENDC
```

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

statement1
is a DIBOL statement to be compiled if *field* is defined.

statement2
is a DIBOL statement to be compiled if *field* is not defined.

Rules

- Each .IFDEF must have a matching .ENDC.
- The statements between .IFDEF and .ELSE (or .ENDC if .ELSE is not specified) are compiled only if the *field* is defined in the Data Division before .IFDEF.
- The statements between .ELSE and .ENDC are compiled only if *field* is not 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.

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
```

4.4 .IFNDEF-.ELSE-.ENDC

Function

.IFNDEF specifies conditional compilation based on the absence of a preceding definition of a named variable within the compilation.

Format

```
.IFNDEF field
  statement1
     .
     .
     .
{.ELSE
  statement2
     .
     .
     .}
.ENDC
```

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

statement1
is a DIBOL statement that is compiled if *field* **is not** defined.

statement2
is a DIBOL statement that is compiled if *field* **is** defined.

Rules

- Each .IFNDEF must have a matching .ENDC.
- The statements between .IFNDEF and .ELSE (or .ENDC if .ELSE is not specified) are compiled only if the *field* is not defined in the Data Division before .IFNDEF.
- The statements between .ELSE and .ENDC are compiled only if the *field* is 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.

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
```

4.5 .INCLUDE

Function

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

Format

.INCLUDE *filespec*

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 eight (8) levels.

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
         ---------- new page ----------
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
```

4.6 .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.

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
         ---------- new page ----------
 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
```

4.7 .MAIN

Function

.MAIN identifies the beginning of the Data Division of the main program.

Format

.MAIN *name*

name
is a valid identifier.

Rules

- Only one .MAIN is allowed within a source file.
- *name* must be unique among routine names in the DIBOL program.
- *name* may be identical to the name of a variable, statement label, or keyword used within the routine.
- The rules for *name* are the same as the rules for subroutine names.

4.8 .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.

Examples

See .LIST for example.

4.9 .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.

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

         ---------- new page ----------
 4 RECORD
 5         DONE,  A1

         ---------- new page ----------

 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
```

4.10 .PROC

Function

.PROC identifies the beginning of the Procedure Division.

Format

.PROC

Rules

- Only one .PROC may be used for each .MAIN or .SUBROUTINE.

4.11 .SUBROUTINE

Function

.SUBROUTINE identifies the beginning of a source program that is an external subroutine.

Format

.SUBROUTINE *name*

name
is a valid identifier.

Rules

- .SUBROUTINE indicates the beginning of the Data Division for an external subroutine. Termination of the subroutine Data Division is indicated by the .PROC directive.
- *name* must be unique among routine names in the DIBOL program.
- *name* may be identical to the name of a variable, statement label, or keyword used within the subroutine.
- The rules for *name* are the same as those for subroutine *names*.

4.12 .TITLE

Function

.TITLE changes the listing page header.

Format

.TITLE *{text_string}*

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.
- If no *text_string* is specified, the page header area is filled with spaces.

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

External Subroutines

This chapter contains information on the dpANS DIBOL 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.

This chapter also defines argument usage for each subroutine. Each argument definition will contain a usage indicator at the beginning of the argument definition section. These four indicators are:

(R)	stands for "read only." The external subroutine expects to use the data contained in the argument.
(W)	stands for "write only." The external subroutine expects to return data in this argument.
(N)	stands for "neither." The argument is neither read nor written to, and all arguments with this indicator may possibly be deleted in the future.
(RW)	stands for "read and write." The external subroutine expects to use both the data contained in this argument and to return data in the argument.

5.1 ASCII

Function

ASCII returns the ASCII character for a decimal character code.

Format

XCALL ASCII *(nexp, afield)*

nexp	(R)	is a numeric field, literal, or expression that contains the decimal character code.
afield	(W)	is an alpha field or record that is to contain the ASCII character.

Rules

- The ASCII character is moved to *afield* according to the rules for moving alpha data.
- *nexp* is treated as a single character code.
- If *nexp* exceeds the range of character codes, *nexp* is automatically converted by dividing the number by 256 and taking the remainder as the character code (258 becomes 2, 259 becomes 3, and so on).

Run-Time Error Conditions

$ERR_ARGMIS	E	Argument missing
$ERR_ARGNUM	F	Incorrect number of arguments passed
$ERR_CANCEL	E	Cancel character detected
$ERR_INTRPT	E	Interrupt character detected
$ERR_WRTLIT	F	Attempt to store data in 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)*

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

Rules

- *afield* should be a nine character field.
- The date is moved to the alpha field according to the rules for moving alpha data.
- The date is returned in the form:

  dd-mmm-yy

dd	is the day of the month (01–31).
mmm	is the first three characters for the name of the month (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, and DEC).
yy	is the last two digits of the year (00–99).

Run-Time Error Conditions

$ERR_ARGMIS	E	Argument missing
$ERR_ARGNUM	F	Incorrect number of arguments passed
$ERR_CANCEL	E	Cancel character detected
$ERR_INTRPT	E	Interrupt character detected
$ERR_WRTLIT	F	Attempt to store data in 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 numeric character code for an ASCII character.

Format

XCALL DECML *(afield, nfield)*

afield	(R)	is an alpha field, alpha literal, or record that contains the ASCII character.
nfield	(W)	is a numeric 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 decimal character code is moved to *dfield* according to the rules for moving decimal data.
- The returned character code can have up to 3 digits.

Run-Time Error Conditions

$ERR_ARGMIS	E	Argument missing
$ERR_ARGNUM	F	Incorrect number of arguments passed
$ERR_CANCEL	E	Cancel character detected
$ERR_INTRPT	E	Interrupt character detected
$ERR_WRTLIT	F	Attempt to store data in 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
```

5.4 DELET

Function

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

Format

XCALL DELET *({ch,} filespec)*

ch	(N)	is a numeric field or numeric literal that specifies a channel number.
filespec	(R)	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

$ERR_ARGNUM	F	Incorrect number of arguments passed
$ERR_CANCEL	E	Cancel character detected
$ERR_INTRPT	E	Interrupt character detected
$ERR_WRTLIT	F	Attempt to store data in a literal

Examples

The following program will delete all versions of the file ARMAST.DDF:

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

5.5 ERROR

Function

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

Format

XCALL ERROR *(errnum{, line})*

errnum	(W)	is a numeric field that is to contain the error number.
line	(W)	is a numeric field that is to contain the line number.

Rules

- *errnum* should be a three digit field.
- The error number is moved to *errnum* according to the rules for moving numeric data.
- 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 numeric data.

Run-Time Error Conditions

$ERR_ARGMIS	E	Argument missing
$ERR_ARGNUM	F	Incorrect number of arguments passed
$ERR_CANCEL	E	Cancel character detected
$ERR_INTRPT	E	Interrupt character detected
$ERR_WRTLIT	F	Attempt to store data in 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{,* $\begin{bmatrix} \textit{afield} \\ \textit{avar} \end{bmatrix}$ *})*

action	(R)	is a numeric field or decimal literal that directs the run-time system what to do in the event of a fatal error.
afield	(R)	is an optional argument which contains the file name of a program to run in place of this program if an untrapped error occurs.
avar	(W)	is an optional record or field that receives the name of the default user-designated program if *action* is equal to three (3).

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 routine which caused the error
PRGNM, A31    ;Name of the main program
```

- 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. The second argument is optional and ignored.
1	Use the default user-designated program on an untrapped error. If there is no default user-designated program, return to the system level. The second argument is optional and ignored.
2	Use the user-designated program specified by *afield* on the untrapped error. This filespec designation remains in effect while the current program is running. The second argument is required.
3	Return, in *avar,* the name of the default user-designated program. If none is defined, return spaces. The second argument is required and must be *avar.*

Run-Time Error Conditions

$ERR_ARGMIS	E	Argument missing
$ERR_ARGNUM	F	Incorrect number of arguments passed
$ERR_CANCEL	E	Cancel character detected
$ERR_INTRPT	E	Interrupt character detected
$ERR_OUTRNG	F	Value out of range
$ERR_WRTLIT	F	Attempt to store data in a literal

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)
```

5.7 FILEC

Function

FILEC allows the creation of files.

Format

XCALL FILEC *(afield1,afield2)*

afield1	(R)	contains the name of the file to be created.
afield2	(R)	contains the name of the file containing the necessary information to create the file.

Rules

- For VAX, the second argument will be the name of an FDL (File Description Language) file.
- For RSTS/E, the second argument will be the name of a DES (RMSDES) file.

5.8 FLAGS

Function

FLAGS alters operating parameters of the run-time system.

Format

XCALL FLAGS *(parameters{,action})*

parameters	(R)	is a numeric field or numeric literal which contains the FLAGS parameters.
action	(R)	is a numeric field or numeric which alters the action of the subroutine.

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:

Value	Meaning
Not specified/passed	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 moved according to the rules for moving data to a numeric field.

Rules When Action is 2

- *parameters* must be a numeric 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 numeric data.

Figure 5–1: FLAGS Option Fields

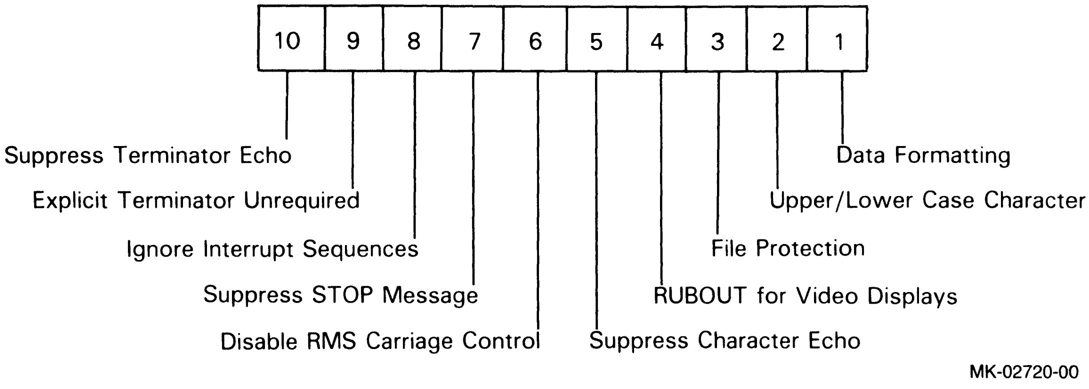

10
9
8
7
6
5
4
3
2
1
Suppress Terminator Echo
Explicit Terminator Unrequired
Ignore Interrupt Sequences
Suppress STOP Message
Disable RMS Carriage Control
Data Formatting
Upper/Lower Case Character
File Protection
RUBOUT for Video Displays
Suppress Character Echo
MK-02720-00

Table 5–1: FLAGS Argument Parameter Assignments

Pos	Value	Definition
1	False	Enable U.S. data formatting by using commas and a period to separate money units (123,456.78).
	True	Enable international data formatting by using periods and a comma to separate money units (123.456,78).
2	False	Perform an UPCASE operation on all characters entered from a character oriented device.
	True	Do not perform an UPCASE operation on characters entered from a character oriented device.
3	False	Permit a file to be opened in Output mode regardless of the existence of a file with the same file name.

Table 5–1 (Cont.): FLAGS Argument Parameter Assignments

Pos	Value	Definition
	True	Detect an attempt to open a file in output mode when one having the same file name already exists and generate an error.
4	False	In response to a delete character sequence entered from a character oriented device during a READS operation, remove the previously entered character from the associated device buffer and echo the character delete confirmation sequence for a hard copy device.
	True	In response to a delete character sequence entered from a character oriented during a READS operation, remove the previously entered character from the associated device buffer and echo the character delete confirmation sequence for video display device.
5	False	Echo character oriented device input.
	True	Do not echo character oriented device input.
6	Any	Implementation dependent.
7	False	Do not suppress messages issued upon normal completion of a program.
	True	Suppress messages issued upon normal completion of a program.
8	False	Do not ignore any program termination sequence entered from a character oriented device.
	True	Ignore any program termination sequence entered from a character oriented device.
9	False	Require an explicit termination character sequence for input from a READS operation from a character oriented device regardless of the number of characters entered.
	True	Implicitly terminate a READS operation from a character oriented device when the input field is filled.
10	False	Echo termination characters or sequences entered from a character oriented device.
	True	Do not echo termination characters or sequences entered from a character oriented device.

Run-Time Error Conditions

$ERR_ARGMIS	E	Argument missing
$ERR_ARGNUM	F	Incorrect number of arguments passed
$ERR_CANCEL	E	Cancel character detected
$ERR_INTRPT	E	Interrupt character detected
$ERR_OUTRNG	F	Value out of range
$ERR_WRTLIT	F	Attempt to store data in a literal

Examples

Disabling character echo is particularly useful when accepting passwords as in the following example. FLAGS digit five (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
          .
          .
          .
```

5.9 INSTR

Function

INSTR searches a string of data for another string.

Format

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

start	(R)	is a numeric field or numeric literal which specifies the character position within *string1* where the search begins.
string1	(R)	is an alpha field, alpha literal, or record to be searched.
string2	(R)	is an alpha field, alpha literal, or record to be searched for in *string1*.
position	(W)	is a numeric 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 one 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 numeric 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 numeric data.

Run-Time Error Conditions

$ERR_ARGMIS	E	Argument missing
$ERR_ARGNUM	F	Incorrect number of arguments passed
$ERR_CANCEL	E	Cancel character detected
$ERR_INTRPT	E	Interrupt character detected
$ERR_WRTLIT	F	Attempt to store data in 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 prompt
       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
             .
             .
             .
```

5.10 MONEY

Function

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

Format

XCALL MONEY *(afield)*

afield	(R)	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.
- The currency symbol shall remain in effect until an XCALL MONEY is executed specifying a different character or the program terminates.
- If *afield* is longer than one character, only the first (leftmost) character is used.

Run-Time Error Conditions

$ERR_ARGMIS	E	Argument missing
$ERR_ARGNUM	F	Incorrect number of arguments passed
$ERR_CANCEL	E	Cancel character detected
$ERR_INTRPT	E	Interrupt character detected

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 RENAM

Function

RENAM changes the name of an existing file.

Format

XCALL RENAM *({ch,}newfile, oldfile)*

ch	(N)	is a numeric field or numeric literal that is ignored (vestigial argument).
newfile	(R)	is an alpha field, alpha literal, or record that contains the new file specification.
oldfile	(R)	is an alpha field, alpha literal, or record that contains the current file specification.

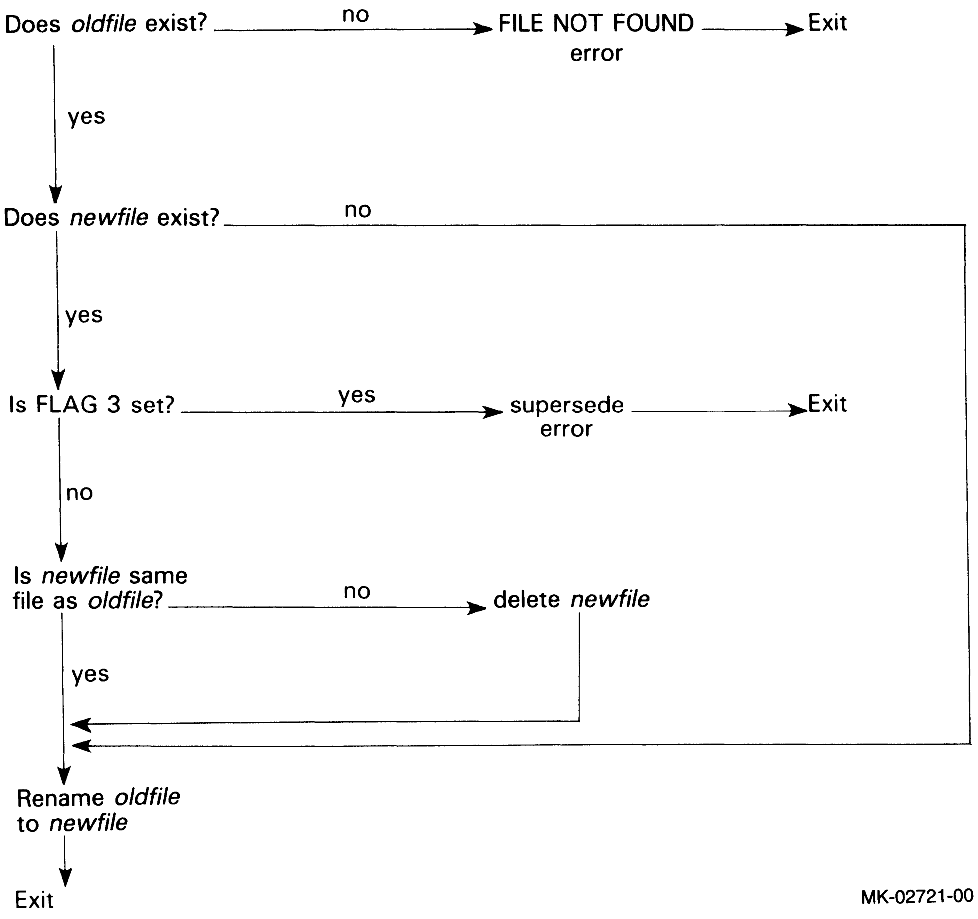
Rules

- 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 three 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 three 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* will be renamed.
- If the version number of *oldfile* is zero 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

$ERR_ARGMIS	E	Argument missing
$ERR_ARGNUM	F	Incorrect number of arguments passed
$ERR_CANCEL	E	Cancel character detected
$ERR_FNF	E	File not found
$ERR_INTRPT	E	Interrupt character detected
$ERR_PROTEC	E	Protection violation
$ERR_WRTLIT	F	Attempt to store data in a literal
$ERR_REPLAC	E	Cannot supersede existing file

Examples

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

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

5.12 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})*

size	(W)	is a numeric field that is to contain the record size.
char	(W)	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 numeric data.
- *char* should be a one character field.
- The terminating character is moved to char according to the rules for moving alpha data.

Run-Time Error Conditions

$ERR_ARGMIS	E	Argument missing
$ERR_ARGNUM	F	Incorrect number of arguments passed
$ERR_CANCEL	E	Cancel character detected
$ERR_INTRPT	E	Interrupt character detected
$ERR_WRTLIT	F	Attempt to store data in a literal

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
```

5.13 SIZE

Function

SIZE returns the size of a field.

Format

$$\textbf{XCALL SIZE} \ \left(\begin{bmatrix} \textit{field} \\ \textit{exp} \end{bmatrix} \textit{,size)} \right.$$

field	(R)	is a variable or literal to be measured.
exp	(R)	is an expression to be measured.
size	(W)	is a numeric field which is to contain the size in characters or digits.

Rules

- The size of a subroutine argument which is not passed is –1.
- The size of an alpha field or numeric 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.

Run-Time Error Conditions

$ERR_ARGMIS	E	Argument missing
$ERR_ARGNUM	F	Incorrect number of arguments passed
$ERR_CANCEL	E	Cancel character detected
$ERR_INTRPT	E	Interrupt character detected
$ERR_WRTLIT	F	Attempt to store data in a literal

Examples

Creating a relative file requires that the size of the records to 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
          .
          .
          .
```

5.14 TIME

Function

TIME returns the current system time of day.

Format

XCALL TIME *(nfield)*

nfield	(W)	is a numeric field that is to contain the current system time.

Rules

- The time is moved to *nfield* according to the rules for moving numeric data.
- The time is returned in a 24-hour notation in the format:

  hhmmss

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

$ERR_ARGMIS	E	Argument missing
$ERR_ARGNUM	F	Incorrect number of arguments passed
$ERR_CANCEL	E	Cancel character detected
$ERR_INTRPT	E	Interrupt character detected
$ERR_WRTLIT	F	Attempt to store data in 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
```

5.15 TTSTS

Function

TTSTS returns an indication of waiting terminal input.

Format

XCALL TTSTS *(nfield{,ch})*

nfield	(W)	is a numeric field which is to contain the number of characters waiting to be input.
ch	(R)	is a numeric field or numeric literal that evaluates to a channel number as specified in a previous OPEN statement.

Rules

- If *ch* is not passed, the default character oriented device associated with the program is assumed. If no default terminal exists, a zero (0) is returned.
- 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 *nfield* according to the rules for moving numeric data.
- 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

$ERR_ARGMIS	E	Argument missing
$ERR_ARGNUM	F	Incorrect number of arguments passed
$ERR_CANCEL	E	Cancel character detected
$ERR_INTRPT	E	Interrupt character detected
$ERR_WRTLIT	F	Attempt to store data in a literal

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
```

Appendix A

dpANS DIBOL Character Set

Table A–1 shows the 128-character ASCII character set and the corresponding decimal codes used by dpANS DIBOL 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.

(dpANS) DIBOL stores both alphanumeric 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-alphanumeric conversion before output to a device may produce numeric values that contain an alphabetic character as the least significant digit.

Table A–1: dpANS DIBOL Character Set

DEC	HX	OCT	ASC	DEC	HX	OCT	ASC	DEC	HX	OCT	ASC	DEC	HX	OCT	ASC
000	00	000	<NUL>	032	20	040	<SPACE>	064	40	100	@	096	60	140	
001	01	001	^A	033	21	041	!	065	41	101	A	097	61	141	a
002	02	002	^B	034	22	042	"	066	42	102	B	098	62	142	b
003	03	003	^C	035	23	043	#	067	43	103	C	099	63	143	c
004	04	004	^D	036	24	044	$	068	44	104	D	100	64	144	d
005	05	005	^E	037	25	045	%	069	45	105	E	101	65	145	e
006	06	006	^F	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	^N	046	2E	056	.	078	4E	116	N	110	6E	156	n
015	0F	017	^N	047	2F	057	/	079	4F	117	O	111	6F	157	o
016	10	020	^P	048	30	060	0	080	50	120	P	112	70	160	p (–0)
017	11	021	^Q	049	31	061	1	081	51	121	Q	113	71	161	q (–1)
018	12	022	^R	050	32	062	2	082	52	122	R	114	72	162	r (–2)
019	13	023	^S	051	33	063	3	083	53	123	S	115	73	163	s (–3)
020	14	024	^T	052	34	064	4	084	54	124	T	116	74	164	t (–4)
021	15	025	^U	053	35	065	5	085	55	125	U	117	75	165	u (–5)
022	16	026	^V	054	36	066	6	086	56	126	V	118	76	166	v (–6)
023	17	027	^W	055	37	067	7	087	57	127	W	119	77	167	w (–7)
024	18	030	^X	056	38	070	8	088	58	130	X	120	78	170	x (–8)
025	19	031	^Y	057	39	071	9	089	59	131	Y	121	79	171	y (–9)
026	1A	032	^Z	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>

MK-02739-00

Appendix B

Error Handling

B.1 Introduction

DIBOL provides for the reporting to the executing program, conditions that result from the execution of various statements. The mechanism that accomplishes this communication detects the condition and provides a value that is available to the executing program for internal determination.

If error trapping is enabled, a branch in program execution shall take place to the specified label, and processing is resumed at that point.

If error trapping is not enabled or if the condition is one which does not allow recovery, the program execution shall be terminated.

B.2 Error Numbers

Error numbers shall be implementation dependent. Each given implementation may return error numbers for any condition or state that is appropriate to the environment.

B.3 Error Mnemonics

Each implementation shall resolve the defined error mnemonics for American National Standard DIBOL consistent with the error number returned at execution time. The mechanism for resolution shall be implementation dependent.

B.4 Error Conditions

Standard error conditions for American National Standard DIBOL, associated mnemonics and the condition which generates the error shall be as described in Table B–1.

Table B–1: dpANS Error Mnemonics

Mnemonic	Severity Level	Definition
$ERR_ALLOC	E	**Invalid value specified for ALLOC** The ALLOC value specified in an open statement is outside the permitted range of values.
$ERR_ARGMIS	E	**Argument missing** An argument expected by the external subroutine was not passed.
$ERR_ARGNUM	F	**Incorrect number of arguments passed** The number of arguments passed to the external subroutine was greater than the required number of arguments defined in the external subroutine.
$ERR_ARGSIZ	F	**Argument specified with wrong size** An argument was passed in a call to an external subroutine that did not match the size expected.
$ERR_BADCMP	F	**Compiler not compatible with execution system** This routine was compiled with a compiler that is not supported by this runtime system.
$ERR_BADKEY	E	**An illegal key was specified** The key specified does not match any of the keys defined for the file in an indexed I/O operation.

Table B–1 (Cont.): dpANS Error Mnemonics

Mnemonic	Severity Level	Definition
$ERR_BIGNUM	E	**Arithmetic operand exceeds 18 digits** An operand in an arithmetic operation exceeds the allowable number of digits.
$ERR_BKTSIZ	E	**Invalid value specified for BKTSIZ** The BKTSIZ: value specified in an open statement is outside the permitted range of values.
$ERR_BLKSIZ	E	**Invalid value specified for BLKSIZ** The BLKSIZ: value specified in an open statement is outside the permitted range of values.
$ERR_BUFSIZ	E	**Invalid value specified for BUFSIZ** The BUFSIZ: value specified in an open statement is outside the permitted range of values.
$ERR_CANCEL	E	**Cancel character detected** The user entered the system specific cancel character.
$ERR_CHNEXC	E	**Too many channels open** An attempt to open more channels than is supported by this system.
$ERR_CHNUSE	F	**Channel is in use** An open statement was executed which specified a channel number currently in use.
$ERR_DEVUSE	E	**Device in use** An OPEN statement attempted to open a non-shareable device that was in use.

Table B–1 (Cont.): dpANS Error Mnemonics

Mnemonic	Severity Level	Definition
$ERR_DIGIT	E	**Bad digit encountered** The alpha value converted to a numeric value contained a character other than a numeric digit, a space, or a sign character (+ or –).
$ERR_DIVIDE	E	**Attempt to divide by zero** An arithmetic operation attempted to divide by zero.
$ERR_EOF	E	**End of file encountered** The end of file has been detected on a READS or READ or the system specific end of file indicator has been entered from a character oriented device during an ACCEPT or READS.
$ERR_FILFUL	E	**Output file is full** All space allocated for a file has been filled, and the file cannot be extended.
$ERR_FILOPT	E	**An invalid operation for file type** An I/O statement was issued for a file which was not allowed by the mode in which the file was opened.
$ERR_FILORG	E	**Invalid file organization specified** The mode specified in an OPEN statement did not match the organization of the file being opened.
$ERR_FILSPC	E	**Bad file name** The file name contained a syntactical error.
$ERR_FINUSE	E	**File in use by another user** The file specified in an OPEN is in use by another user and is not available as a shared file.

Table B–1 (Cont.): dpANS Error Mnemonics

Mnemonic	Severity Level	Definition
$ERR_FNF	E	**File not found** The file name used does not match an existing file name.
$ERR_ILLCHN	F	**Illegal channel number specified** A channel number was specified that is outside the legal range of channel numbers.
$ERR_INTRPT	E	**Interrupt character detected** The user entered the system specific interrupt character.
$ERR_IOFAIL	E	**Bad data encountered during I/O operation** A system error was returned during an I/O operation that indicates the data transfer was incorrect.
$ERR_IOMODE	E	**Bad mode specified** A mode was specified in an OPEN that conflicts with the file organization or is invalid.
$ERR_KEYNOT	E	**Key not same** The key value specified does not match an existing record in the file.
$ERR_LOCKED	E	**Record is locked** This record or group of records is in use by another user.
$ERR_NOCALL	F	**Return with no CALL or XCALL** A RETURN or XRETURN was executed in a program without a previous CALL or XCALL.
$ERR_NOCURR	E	**No current record** There is not a current record specified on the channel on which the I/O operation was executed.

Table B-1 (Cont.): dpANS Error Mnemonics

Mnemonic	Severity Level	Definition
$ERR_NODUPS	E	**Duplicate key specified** A duplicate key was specified in a file that was declared as not allowing duplicate keys.
$ERR_NOMEM	E	**Not enough memory for desired operation** This operation could not be performed with available memory.
$ERR_NOOPEN	F	**Channel has not been opened** An I/O operation was attempted on a channel that has not been opened.
$ERR_NOSPAC	E	**No space exists for file on disk** There is not enough room on the specified disk for the output file.
$ERR_NOTDIB	F	**Caller not DIBOL** A call was made by a non-DIBOL program.
$ERR_NUMREC	E	**Invalid value specified for NUMREC** The NUMREC: value specified in an open statement is outside the permitted range of values.
$ERR_ONLYRD	E	**Attempt to write to read only device** A write operation was attempted on a read only device.
$ERR_ONLYWR	E	**Attempt to open output device in Input mode** An attempt to open a write only device in input mode.

Table B–1 (Cont.): dpANS Error Mnemonics

Mnemonic	Severity Level	Definition
$ERR_OUTRNG	F	**Value out of range** A statement parameter or argument is outside the range of permitted values.
$ERR_PROTEC	E	**Protection violation** An attempt has been made to access a resource that is protected from this user.
$ERR_RECNUM	E	**Illegal record number specified** An illegal record number has been specified.
$ERR_RECSIZ	E	**Invalid value specified for RECSIZ** The RECSIZ value specified in an OPEN statement is outside the permitted range of values.
$ERR_REPLAC	E	**Cannot supersede existing file** An attempt was made to supersede a file which has been protected against deletion.
$ERR_RNF	E	**Record not found** The record specified does not exist.
$ERR_SUBSCR	E	**Invalid subscript specified** A value specified as a subscript is outside the allowable range of values.
$ERR_SYSTEM	F	**System error** An error occurred during an operating system or other external system service call.

Table B–1 (Cont.): dpANS Error Mnemonics

Mnemonic	Severity Level	Definition
$ERR_TOOBIG	E	**Input data size exceeds destination size** An attempt to store data into a destination that is not sufficient to contain the complete data transfer.
$ERR_WRTLIT	F	**Attempt to store data in a literal** An attempt has been made to store data into a literal.
$ERR_ARGCOUNT	E	**Needs minimum of three arguments** A call to PAK requires a minimum of three arguments.
$ERR_ARGOUTLIM	E	**Argument out of record** An argument specified in a call to PAK/UNPAK is outside of the specified record area.
$ERR_ARGOUTORD	E	**Arguments out of order** The arguments specified in a call to PAK/UNPAK are not in the correct order.
$ERR_CHADEFERR	F	**Channel definition error** An internal error occurred in the DIBOL Run-Time Library. Please submit an SPR.
$ERR_CMPERR	F	**Compilation error** Execution of a statement which contained a compilation error was attempted.
$ERR_DBLRTLERR	F	**DIBOL Run Time Library internal error** An error occurred in the DIBOL Run-Time Library software. Please submit an SPR.

Table B-1 (Cont.): dpANS Error Mnemonics

Mnemonic	Severity Level	Definition
$ERR_DEVNOTRDY	E	**Device not ready** The device accessed by a I/O statement was off-line or otherwise not ready.
$ERR_EXQUOTA	E	**Exceeded quota**
$ERR_FLDNOTPAK	E	**Field not packed** A field specified in a call to UNPAK was not packed.
$ERR_FLDRECLNG	F	**Field or record too long** The size of a field or record exceeds the maximum allowed size.
$ERR_HANNOTAVA	E	**Device handler not available** The device specified is not known to the operating system.
$ERR_ILLBIOSIZ	E	**Illegal block I/O record size** The record size specified in a block I/O statement was not a multiple of 512 bytes.
$ERR_ILLTRMNUM	E	**Illegal terminal number** The VMS terminal identification could not be translated into a DIBOL terminal number, or a DIBOL terminal number could not be translated into a VMS terminal identification.
$ERR_INVARGTYP	F	**Invalid argument type** An argument passed to a library routine was of the wrong type (alpha instead of decimal, or decimal instead of alpha).
$ERR_INVKEYNUM	E	**Invalid KEYNUM value** Invalid key of reference specified with KEYNUM keyword on READ or FIND.

Table B–1 (Cont.): dpANS Error Mnemonics

Mnemonic	Severity Level	Definition
$ERR_JOBSTAERR		
$ERR_NOMSGMGR	E	**Unable to open message manager mailbox** The mailbox used by the DIBOL SEND/RECV message manager could not be opened.
$ERR_NO_RECSIZ	E	**RECSIZ keyword required** Record size is required by OPEN modes O:R, O:I and OPEN keyword NUMREC.
$ERR_NO_SUCOPR	F	**No such operation (RMS)** Support for the VAX RMS service or option is not present in the system.
$ERR_NOTISMFIL	E	**Not ISAM file** An OPEN statement attempted to open a non-indexed file using SI or SU mode.
$ERR_OPENERROR	E	**Error during file open** An unexpected error occurred while attempting to open a file.
$ERR_QUENOTAVA	E	**Queue not available or invalid queue name** The queue specified in an LPQUE statement was either not available or invalid.
$ERR_RECEXTCAL	F	**Recursive external call**

Table B–1 (Cont.): dpANS Error Mnemonics

Mnemonic	Severity Level	Definition
$ERR_RMSERROR	F	**Unexpected RMS error** An unanticipated RMS error occurred.
$ERR_TIMOUT	E	**Time out detected**
$ERR_UNDOPC	F	**Undefined opcode** An internal error was detected by the DIBOL Run-Time Library. Please verify that compatible versions of the DIBOL compiler and RTL are being used. If the compiler and RTL are compatible, please submit an SPR.

Glossary

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

alpha expression A valid combination of operands and operators where all operands are alpha data type.

alphabetic A character set that contains only letters.

arithmetic expression An expression that consists entirely of arithmetic operators and their numeric operands. It evaluates to a numeric value.

array A DIBOL technique for specifying more than one field of the same length and type.

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

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

branch change in the sequence of execution of DIBOL 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 alphanumeric field to spaces or a numeric field to zeros.

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

compiler directive A DIBOL statement that is an instruction to the compiler.

compound statement None or more procedural statements preceded by a BEGIN declarative and followed by a matching END declarative.

conditional statement A DIBOL statement that consists of one or more keywords, a logical expression, and a simple compound statement that is executed based upon the truth value of the logical expression.

continuation line A physical line whose first non-spacing character is an ampersand. Text following the ampersand is considered part of the current logical line.

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

Data Division DIBOL program portion that defines data areas which can be referenced and manipulated during program execution.

data record A record within a DIBOL data file.

DEC Acronym for DIGITAL Equipment Corporation.

decimal DIBOL data type used to store numeric-only data in zoned decimal format.

decimal expression A combination of one or more operands and operators that is evaluated by a prescribed set of rules to yield a single decimal value.

delimiter A separator between identifiers, keywords, and literals.

DIBOL DIGITAL'S Interactive Business Oriented Language is used to write business application programs.

dynamic access The facility to obtain data from or to enter data into a file using both sequential and direct access methods.

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 A logical or mathematical statement made up of operands and operations. Also, any combination of variables and constants with arithmetic operators which can be evaluated to produce a result.

external subroutine A subprogram with its own Data and Procedure Divisions that is compiled independently, but can only be run from a calling program or other external subroutine with which it is linked.

fatal error An error which terminates program execution.

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

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

file specification (filespec) An implementation specific alpha character string used to uniquely identify a file.

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.

identifier A character string that is a symbol name.

illegal character A character that is not valid according to the DIBOL 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 alphanumeric 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.

line See logical line, continuation line, physical line, blank line. Unless otherwise specified, line will mean logical line.

literal An alpha, numeric, or user-defined value permanently defined in a program.

logical expression A decimal expression, alpha field, or decimal field that is evaluated by a prescribed set of rules to yield a truth value.

logical line A component of a DIBOL program.

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 (channel).

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.

physical line A record within a text file that is a DIBOL source program.

primary key See key.

Procedure Division DIBOL program portion that defines the processing logic to be performed at execution time.

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.

record A memory area composed of one or more fields.

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 the screen.

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.

simple statement A DIBOL statement that performs a single function.

source program A program written in the DIBOL 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 the high order bit in the low order byte.

Index

HOW TO ORDER ADDITIONAL DOCUMENTATION

DIRECT TELEPHONE ORDERS

In Continental USA and Puerto Rico call **800-DIGITAL**	In Canada call **800-267-6146**	In New Hampshire, Alaska or Hawaii call **800-DIGITAL**

ELECTRONIC ORDERS (U.S. only)

Dial 800-DEC-DEMO with any VT100 or VT200 compatible terminal and a 1200/2400 baud modem. If you need assistance, call 1-800-DIGITAL.

DIRECT MAIL ORDERS (U.S. and Puerto Rico*)

DIGITAL EQUIPMENT CORPORATION
P.O. Box CS2008
Nashua, New Hampshire 03061

DIRECT MAIL ORDERS (Canada)

DIGITAL EQUIPMENT OF CANADA LTD.
100 Herzberg Road
P.O. Box 13000,
Kanata, Ontario, Canada K2K 2A6
Attn: DECDIRECT OPERATIONS

INTERNATIONAL

DIGITAL EQUIPMENT CORPORATION
A&SG Business Manager
c/o Digital's local subsidiary
or approved distributor

Internal orders should be placed through the
Software Distribution Center (SDC) Digital Equipment Corporation,
Westminster, Massachusetts 01473-0471

*Any prepaid order from Puerto Rico must be placed
with the Local Digital Subsidiary:
800-754-7575

Reader's Comments

dpANS DIBOL
Language Reference Manual
AA–KZ36A–TK

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less ____________________

____

What I like best about this manual is ____________________

____

What I like least about this manual is ____________________

____

I found the following errors in this manual:

Page Description

______ ____________________

______ ____________________

______ ____________________

Additional comments or suggestions to improve this manual:

____

____

____

I am using **Version** ______ of the software this manual describes.

Name/Title ____________________ Dept. ____________

Company ____________________ Date ____________

Mailing Address ____________________

____________________ Phone ____________

-- 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
Corporate User Publications—Spit Brook
ZK01-3/J35
110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

-- Do Not Tear - Fold Here

Cut Along Dotted Line