INTRODUCTION TO DIBOL-83

Order No. AA-P042B-TK

April 1984

Supersession: This is a revision.

Operating System: VAX/VMS, CTS-300, RSTS/E, Professional, RSX-11M-Plus, Micro/RSX, Professional CTS-300

Software Version: Applicable to all products containing DIBOL-83

First Revision, April 1984

TABLE OF CONTENTS

CONTENTS (CONT.)

PREFACE

PURPOSE

Introduction to DIBOL-83 is a precursor to the *DIBOL-83 Language Reference Manual*. It is intended for the following audiences:

The experienced programmer who is learning DIBOL-83 and needs to become productive as rapidly as possible in writing, maintaining, or modifying DIBOL-83 programs.

The programmer who knows at least one high-level computer language and wants to learn DIBOL-83

NON-GOALS

This manual is not intended to be used as a reference document and should not be construed as such The complete authority and reference document on the DIBOL-83 Language is the *DIBOL-83 Language Reference Manual*.

MANUAL ORGANIZATION

The manual is organized as follows:

Chapter 1 contains a run-down of the DIBOL-83 language; when and why it was developed and how it evolved into what it is today. Chapter 1 also describes the DIBOL Standards Organization.

Chapter 2 is a general introduction of DIBOL-83. This chapter gives a basic description of the language and some of the terms used when discussing the language. This chapter also includes information on compiler directives and statement labels.

Chapter 3 picks up where Chapter 2 leaves off; going from the general to the specific. Program structure is the main focus of this chapter and the parts that make up the whole

Chapter 4 discusses program control in DIBOL-83 and also introduces structured constructs available with DIBOL-83. Each construct is discussed and examples are shown

Chapter 5 contains information on the assignment statement, formatting, and operator precedence

Chapter 6 explains how DIBOL-83 uses external and internal subroutines

Chapter 7 introduces the concept of arrays and subscripting within a DIBOL-83 program.

Chapter 8 deals with Input/Output operations within DIBOL-83 and the statements used in performing these operations.

Appendix A contains the DIBOL-83 Character Set.

The Glossary defines terms used for DIBOL-83.

ASSOCIATED DOCUMENTS

DIBOL for Beginners

DIBOL-83 Language Reference Manuals
DIBOL-83 System User Guides
DIBOL-83 Language Reference Cards

CHAPTER 1

INTRODUCING DIBOL-83

This chapter introduces the DIBOL-83 language and explains how it evolved from a simple commercial language into what it is today. This chapter also discusses the DIBOL Standards Organization (DSO).

1.1 FROM DIBOL to DIBOL-83

The original DiBOL language was developed in the late 60's as a simple commercial language to serve as a development tool for writing applications for PDP-8 minicomputers. The first version of DIBOL was comprised of a collection of ASSEMBLY language routines. Although effective, these routines were complex and slow. The compiler printed its listings at a slow rate, even with a "fast" printer. Not satisfied with this, DIGITAL set forth to produce an improved version of the DIBOL language that would meet the high standards of both the corporation and its customers.

1.1.1 COS-300

A DIBOL development team was formed in 1971 faced with the task of improving DIBOL. After a concentrated effort, the project team produced a fast and efficient DIBOL Compiler, Interpreter, Sort program, Monitor, and assorted utilities. DIBOL was offered as a Commercial Operating System called COS-300. The initial customer reaction was encouraging and enhancements were made for succeeding versions.

In 1974 a version of COS-300 was released which allowed up to eight terminals to concurrently execute a single DIBOL program. Included was a 'foreground' data entry package which allowed data entry/inquiry to take place simultaneously with the operation of background DIBOL data processing.

1.1.2 PDP-11s and COS-350

The advent of the PDP-11s brought a need for a proven DIBOL-based system capable of running on such machines. This need was met with the development of an expanded version of the DIBOL language called DIBOL-11. The operating system for this line of PDP-11 processors became known as COS-350. In 1975 COS-350 became the first commercial timesharing DIBOL-11 operating system. This initial release of COS-350 offered multi-user facilities for two to four users.

1.1.3 COS-310 and DIBOL-8

COS-300 was renamed COS-310 and became a floppy disk based, single-user system. A highly competitive product, COS-310 became the first high volume operating system produced by DIGITAL. This single-user DIBOL operating system was used on the PDP-8A family of processors. COS-310 supported the original DIBOL language known today as DIBOL-8.

COS-310 became the first tightly integrated packaged system produced by DIGITAL and is the only DIBOL-8 operating system still under active development. COS-310 has continued to adapt to customer needs with each release.

1.1.4 CTS-300 and RSTS/E DIBOL

COS-350 eventually became known as CTS-300 (Commercial Timesharing System). CTS-300 serves as a DIBOL-11 timesharing operating system for the PDP-11 family.

RSTS/E is a timesharing operating system that functions in a much larger environment than CTS-300, supporting many terminal users concurrently. RSTS/E DIBOL enables development work to be done by multi-users, working simultaneously. RSTS/E contains a number of CTS-300 facilities, which, along with DIBOL-83 support, gives the two systems the capability of developing some applications on and for either system.

1.1.5 VAX, PRO, and RSX DIBOL

In the late 70's the VAX-11 system became available and soon after VAX DIBOL also became available The Professional-350, one of DIGITAL's entries into the personal computer market contained DIBOL as part of its package. DIBOL has recently become available for RSX-11M-PLUS systems to complete the availability of DIBOL across the family of systems now offered by DIGITAL. Today's DIBOL is known as DIBOL-83 and has evolved through the efforts of the DIBOL Standards Organization (DSO).

1.2 DIBOL STANDARDS ORGANIZATION

The DIBOL Standards Organization (DSO) was formed in 1979 to establish language standards for DIBOL and to eliminate the incompatibilities of DIBOL between systems. DSO is made up of DIBOL developers and customers who discuss enhancements and customer needs.

After countless meetings, discussions, and testing, DSO produced a new language standard. DIGITAL produced and released implementations based on this standard (DIBOL-83) for CTS-300, RSTS/E, PROFESSIONAL, VAX, and RSX-11M-PLUS.

Today DIBOL-83 is recognized as an efficient high-level programming language, supported by a proven family of operating systems and processors. This dictates a need for control and standardization which DSO offers.

1.3 DIBOL-83

Through the efforts of developers and customers DIBOL evolved into today's DIBOL-83, an interactive high-level business language. DIBOL-83 offers:

- Compatibility across DIGITAL's range of computers, from DIGITAL's Professional Personal Computer through the VAX systems
- Flexibility with record handling through subscripting, array handling, substring operations, and record redefinition.
- A simple-to-follow syntax that uses English action verbs as the first element characterizing an action to be performed.
- Sequence control statements that allow you to write structured programs and reduces development, testing, and maintenance time
- A large set of DIGITAL-supplied external subroutines that help you develop software as efficiently as possible

From DIBOL to DIBOL-83, this language has continued to adapt and develop according to its customer s needs and will continue to do so in the future

CHAPTER 2

TALKING DIBOL-83

This chapter briefly introduces the DIBOL-83 language and describes some of the terms that are used when talking about the language.

2.1 DIBOL-83 CHARACTER SET

The DIBOL-83 Character Set is comprised of a subset of symbolic characters from The American Standard Code for Information Exchange (ASCII) characters. Appendix A lists the ASCII characters and their associated numeric codes.

2.2 DIBOL-83 STATEMENT TYPES

DIBOL-83 uses six functional groups of statements. They are:

Compiler Directives and Declarations which are instructions that provide information on how to compile a program.

Data Specification Statements which identify and define all characteristics on data processed by the program.

Data Manipulation Statements which perform conversion and assignment tasks.

Control Statements which modify the order of statement execution within a program.

Intertask Communications Statements which allow communication between programs

Input/Output Statements which control transmission and reception of data between memory and input/output devices.

DIBOL-83 statements are English verbs which represent actions to be performed (such as READ, WRITE, SLEEP, OPEN, and CALL).

These statements may also contain arguments, expressions, or other statements. Arguments may be 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.

2.2.1 Statement Labels

A statement label is a unique, symbolic name which identifies a specific statement in the Procedure Division. Statement labels are used with the GOTO statement to transfer program control. The GOTO statement is discussed in Chapter 4.

Statement label names must begin with an alphabetic character followed by any combination of alphabetic or decimal characters, and the two special characters $, _ A label name may begin at any column on a line. It must precede the statement it identifies and must be followed by a comma. A statement label must be on the same line as the statement or be on a line by itself preceding the statement.

In the following example the label AGAIN identifies the WRITES statement which follows it. When the GOTO statement is executed, control transfers to the WRITES statement which AGAIN identifies.

```
AGAIN,
   WRITES, (1,BLANK)
   .
   .
   .
   IF (COUNT .LT. 3) GOTO AGAIN
```

2.3 COMPILER DIRECTIVES

DIBOL-83 has a collection of Compiler Directives which provide instructions about the program to the compiler. They can be used anywhere in the program and they are not executable at run time.

The Compiler Directive PROC tells the compiler where the Data Division ends and the Procedure Division begins. PROC can be used only once in a program and only in one place. It is placed at the end of the Data Division.

The Compiler Directives serve many other functions. They can be used to enable (.LIST) or disable (.NOLIST) the listing of compiler source code. They can tell the compiler to include a top-of-page command (.PAGE) and place a new title in the page header (.TITLE). There is even a directive that informs the compiler to open a specified file and continue the compilation using that file (.INCLUDE).

There are two sets of Compiler Directives which can be used to conditionally compile statements; .IFDEF-.ENDC and .IFNDEF-.ENDC.

.IFDEF causes statements that follow it (up to .ENDC) to be compiled if a specified variable (field or record) is defined. In the following example the CALL statement is not compiled because the variable BNKNBR is not a defined variable.

```
RECORD
        NAME,       A12
        ADDR,       A15
PROC
.IFDEF BNKNBR
        CALL RECALC
.ENDC
```

.IFNDEF causes statements that follow it (up to .ENDC) to be compiled if a specified variable (field or record) is NOT defined. In the following example the CALL statement is compiled because the variable BNKNBR is not a defined variable.

```
RECORD
        NAME,       A12
        ADDR,       A15
PROC
.IFNDEF BNKNBR
        CALL RECALC
.ENDC
```

All the DIBOL-83 Compiler Directives are described in detail in the *DIBOL-83 Language Reference Manual*.

2.4 DIBOL-83 TERMS

DIBOL-83 has its own set of terms. These terms may or may not be familiar to you. Later chapters in this manual may use some of these terms. The purpose here is to familiarize you with these terms at this point in the book. Here are a few:

Alphanumeric
This is one of the two data types recognized by DIBOL. Alphanumeric fields may contain any characters from the character set.

Array
An array is a technique for specifying multiple occurrences of a field of a certain length and type

Channel
A channel is a number that represents a device or file that is associated with an input/output statement.

Comments
Comments are informative notes that can be included in a DIBOL program. Comments must be preceded by a semicolon (;).

Keyword
A keyword is part of a command operand and it consists of a specific character string

Mode
A mode is a designation used in an OPEN statement which indicates the purpose for which a file was opened.

Subscript
A subscript is a designation which specifies particular parts (characters, values records) within an array.

Trappable error
A trappable error is an error that can be handled by the executing program so that execution will not terminate.

2.5 LINE CONTINUATION

DIBOL code can be continued onto another line by using an ampersand (&). The ampersand specifies that a statement is continued. This is accomplished by placing the ampersand in the first character position on the line on which the statement will be continued Only one statement may appear on a line

2.6 PROGRAM DOCUMENTATION

In order to provide easier understanding of what a program is doing, comments are allowed in DIBOL programs. A semicolon (;) is used for this purpose

A semicolon indicates to the DIBOL compiler that what follows is a comment and is not executable, the compiler ignores any characters after a semicolon A semicolon may be contained in a literal without indicating a comment. A comment may begin at any column in a line of DIBOL source code If a comment is to be continued on a following line, the semicolon must be repeated

CHAPTER 3

DIBOL-83 PROGRAM STRUCTURE

A DIBOL-83 program is made up of two major parts a Data Division and a Procedure Division This chapter explains the characteristics of these two program sections and also discusses the DIBOL-83 assignment statement

3.1 DATA DIVISION

The Data Division of a DIBOL-83 program contains statements that define and identify data used by the program All variables to be used in the program must be declared in the Data Division

3.1.1 RECORD Statement

The RECORD statement is the most commonly used DIBOL-83 statement within the Data Division This RECORD statement is known as a Data Declaration statement The RECORD statement defines the record, data, or work area that will be processed via actions specified in the Procedure Division

RECORD names must be no larger than six characters and must begin with an alphabetic character This character may be followed by any combination of alphabetic, numeric and the two special characters $ and _. The following are examples of RECORD statement names

```
RECORD   PAYROL      RECORD   NUMBER      RECORD   OUTPUT
```

Each name begins with an alphabetic character

3.1.2 COMMON and SUBROUTINE Statements

The COMMON and SUBROUTINE statements are the other Data Declaration statements which can be used in the Data Division COMMON is similar to RECORD except that fields defined in COMMON can be also used by an external subrcutine An external subroutine is a unit of (DIBOL) code which is separately compiled and may be linked into many different programs Subroutines are discussed in detail in Chapter 6.

COMMON names must conform to the same rules as RECORD names When a COMMON area is to be used by a subroutine, that COMMON area must be defined identically in both the calling routine and the subroutine. The following are examples of COMMON statement names

```
COMMON   EMPNM      COMMON   BALNC      COMMON   SALRY
```

A SUBROUTINE statement identifies a program as an external subroutine This subroutine can be called into the main program via an XCALL statement XCALL will be explained in more detail later on in this book. The following are examples of SUBROUTINE statement names

```
SUBROUTINE   TOTAL      SUBROUTINE   SCALE      SUBROUTINE   PRODU
```

3.1.3 Field and Argument Definitions

RECORD and COMMON must be followed by one or more field definitions and SUBROUTINE must be followed by one or more argument definitions. Field definitions define fields or variables within a RECORD or COMMON area. Field and argument definitions (names) must begin with an alphabetic character. This character may be followed by any combination of alphabetic, numeric, and the two special characters $ and _. All field names must be followed by a comma.

All fields have a field size and data type. DIBOL-83 supports two data types: alphanumeric and decimal.

Fields do not require names. Only a comma is required before the data type specification. Unnamed fields cannot be directly referenced from the Procedure Division, but they can be accessed by referencing the record under which they are defined.

In the upcoming example, the field is named SUM and its data type is identified as being decimal (D). Decimal fields contain integer quantities represented by the digits 0-9. A negative value is encoded with the right-most digit.

If no sign appears in the field or if the plus sign is present, the value is positive. The value is negative only if the minus sign is present in the field.

```
SUM, D1
```

The number 1 following the D defines the field size. SUM may contain 1 character which may be a positive or negative integer in the range 0-9.

In the next example WORDS is an alphanumeric (A) field. Alphanumeric fields may contain any character from the character set. Appendix A contains the DIBOL-83 character set. WORDS has a field size of 23 characters.

```
WORDS, A23.  The sum of 2 plus 3 is
```

NOTE

> Both record and field names can be referred to as variables because the values they contain can change during program execution.

Subroutine argument definitions are similar to field definitions except that they specify the data passed between the external subroutine and the program that is calling the subroutine. An argument definition name must begin with an alphabetic character. This character may be followed by any combination of alphabetic, numeric, and the two special characters $ and _. The argument name must be followed by a comma.

3.1.4 Initial Values

All DIBOL-83 fields have initial values. An initial value is the original value a field has when program execution begins. DIBOL-83 supplies a default initial value for each field or the programmer may specify an initial value.

DIBOL-83 supplies an initial value of spaces for alphanumeric fields. In the following example ANSWER contains 1 space.

```
ANSWER, A1
```

Zeros are the initial value supplied for decimal fields. In the following example SUM contains 1 zero.

```
SUM, D1
```

The programmer may specify initial values after the field size, separated by a comma. Apostrophes (') and quotation marks ('') may be used to delimit initial values for alphanumeric fields; initial values for decimal fields have no delimiters.

In the following example WORDS has a programmer-specified initial value.

```
WORDS, A23, 'The sum of 2 plus 3 is '
```

Whatever is contained within the delimiters is taken as the initial value. The initial value in WORDS contains the following string of characters:

```
The sum of 2 plus 3 is
```

The following example shows a programmer-specified initial value for a decimal field:

```
NUMBER, D1, 5
```

This specifies a decimal field named NUMBER with a field size of 1 and an initial value of 5.

3.2 PROCEDURE DIVISION

The Procedure Division in a DIBOL-83 program always follows the Data Division and contains executable statements which work with the data declared in the Data Division. The PROC Compiler Directive separates the Data Division from the Procedure Division. PROC must appear in every program to serve this function. It does not affect the processing that occurs within the program.

Figure 3-1 shows a schematic drawing of a typical DIBOL-83 program structure.

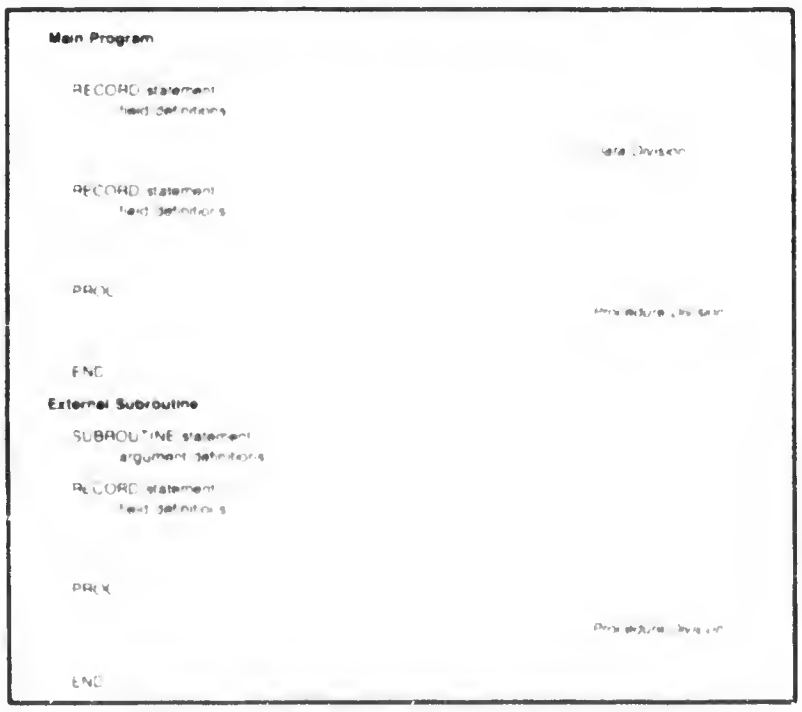

Figure 3-1 DIBOL-83 Program Structure

3.2.1 Procedure Division Statements

The Procedure Division is made up of four different groups of statements. They are:

Data Manipulation Statements which are used for conversion and value assignment

Control Statements which are used to modify the order of statement execution within a program

Intertask Communications Statements which allow communication between programs

Input/Output Statements which control the transmission and reception of data between memory and input/output devices.

Some of the statements in each of the above groups are introduced and discussed further on in this book. They are all discussed in detail in the *DIBOL-83 Language Reference Manual*

CHAPTER 4

PROGRAM CONTROL IN DIBOL-83

This chapter introduces DIBOL-83 program control and the elements which can be used to achieve program control.

4.1 DIBOL-83 RELATIONAL OPERATORS

DIBOL-83 offers a group of relational operators which can create relational expressions that affect program control. In the following example a relational expression containing the relational operator .LT. (for less than) tests to see if the value of COUNT is less than 3. The expression evaluates as true if the value of COUNT is less than 3 and false if the value of COUNT is not less than 3.

```
IF (COUNT .LT. 3) GOTO AGAIN
```

When IF is executed, if the value of COUNT is less than 3, the GOTO statement is executed. If the value of COUNT is equal to or greater than 3, GOTO is ignored and the next statement is executed.

Table 4-1 contains all the DIBOL-83 relational operators for forming relational expressions

Table 4-1

DIBOL-83 Relational Operators

Operator	Meaning
.EQ.	equal to
.NE.	not equal to
.LT.	less than
.LE.	less than or equal to
.GT.	greater than
.GE.	greater than or equal to
.NOT.	changes true to false and false to true
.AND.	Boolean AND
.OR. and .XOR.	Boolean OR and exclusive OR

4.2 GOTO STATEMENT

The GOTO statement transfers program control to a statement label specified as its argument. In the following example the GOTO statement transfers control to the statement identified by the label AGAIN. Once control is transferred to the specified statement, execution continues from that point.

```
IF (COUNT .LT. 3) GOTO AGAIN
CLOSE 1
```

In this example the GOTO statement allows the statements within the label AGAIN to be executed over and over until the value of COUNT becomes 3. At that time program control will pass to the CLOSE statement.

Program control can also be affected via a DIBOL-83 structured construct The following section introduces these constructs.

4.3 DIBOL-83 STRUCTURED CONSTRUCTS

DIBOL-83 offers an assortment of structured constructs that allow structured programming for all DIBOL-83 programs. This section introduces the DIBOL-83 constructs and illustrates their use

Currently there are six structured constructs in DIBOL-83. They are:

DO-UNTIL which repetitively executes a statement UNTIL a condition is true

FOR which repetitively executes a statement based on an index, with an initial, final, and step value.

IF which executes a statement IF a condition is true.

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

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

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

Each of these structured constructs is explained in detail in the following sections

4.3.1 Using BEGIN-END Blocks

A program is created to perform a certain task or job. In order to do this job the program may have to go through a number of smaller steps, tasks, or functions. Many times there will be a use for one or more of the DIBOL-83 structured constructs within these steps.

The BEGIN-END block groups individual statements into a single entity (unit) which can be conditionally executed or repeated. This results in a more controlled and readable program. Each one of these functions must be carried out for the program to complete its job The BEGIN-END block makes it easy for the programmer to easily identify a certain function and to quickly make a change

In the following example, the outer BEGIN-END block contains statements which are repetitively executed until CUSNAM equals spaces The inner BEGIN-END block contains statements which are executed only if the current customer's balance is over $100

```
DO
   BEGIN
   READS (1,CUSREC,EOF)
   IF BALANC.GT.100
        BEGIN
        NAME = CUSNAM
        AMT = BALANC
        WRITES (6,PLINE)
        END
   END
UNTIL CUSNAM.EQ.SPACES
```

The examples shown throughout this book will, at times, use BEGIN-END blocks to illustrate their use The purpose here is to show their value within a program

4.3.2 IF Statement

If executes a statement only if a condition is true The condition that is tested can be either true (non-zero) or false (zero). If the condition is true then the statement is executed If the statement is false (zero) then the statement is not executed

Figure 4-1 contains a flowchart of the IF statement

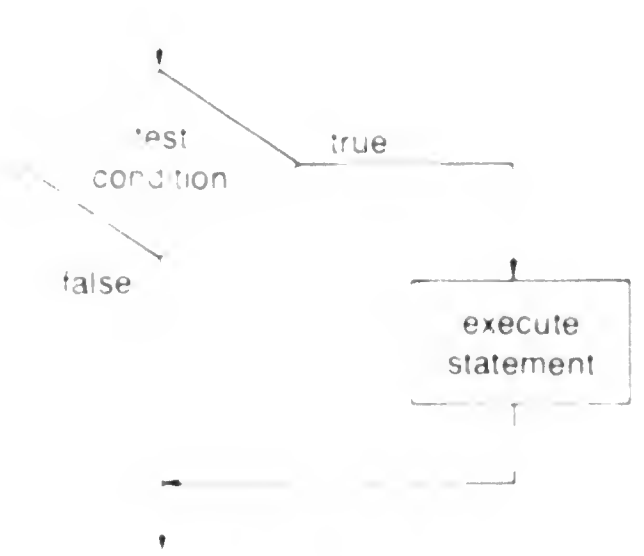

Figure 4-1 IF Flowchart

An example of an IF statement in a program would be

```
IF BALANC GT 1000
   BEGIN
   WRITES (6,CUSNAM)
   CLEAR BALANC
   END
READS (1,CUSFIL)
```

In this example the program is reading a field named BALANC. The program is checking to see whether BALANC is over $1000. If the program finds a BALANC greater than $1000 the WRITES statement is then executed printing out the contents of CUSNAM. The BALANC field is then cleared and then the program is instructed to read CUSFIL.

4.3.3 IF-THEN-ELSE Statement

IF-THEN-ELSE executes one of two statements based on a test condition. The condition is a decimal expression that determines which of the two statements is to be executed.

The test condition must be either true (non-zero) or false (zero). If the condition is true the statement following the THEN is executed. If the condition is false the statement following the ELSE is executed.

Figure 4-2 contains a flowchart for the IF-THEN-ELSE statement.

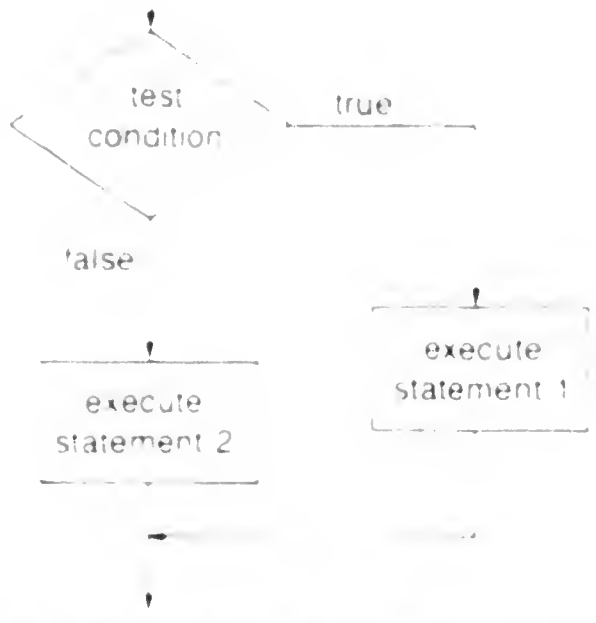

Figure 4-2 IF-THEN-ELSE Flowchart

In the following example BOOKS is tested to determine whether a book is overdue. The IF THEN ELSE statement determines what will be displayed on the terminal as a result of this test.

```
READS (1,BOOKS)
IF OVERDU GT 30
  THEN
    DISPLAY (15,TITLE,' is over a month late')
  ELSE
    DISPLAY (15,TITLE,' is less than a month late')
```

4.3.4 WHILE Statement

WHILE repetitively executes a statement as long as a condition is (remains) true. The condition is evaluated prior to each possible execution of the statement. The condition is either true (non-zero) or false (zero).

If the condition is true the statement is executed. If the condition is false the statement is not executed.

Figure 4-3 contains a flowchart illustrating how WHILE works

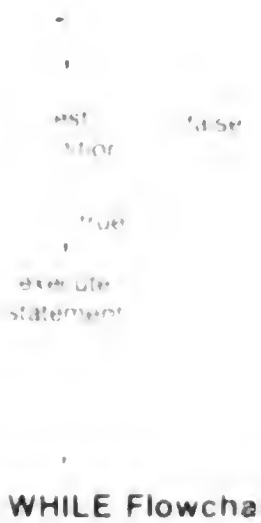

Figure 4-3 WHILE Flowchart

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 SIZE GT 9               Trim line
          SIZE = SIZE-1
```

4.3.5 DO-UNTIL Statement

DO-UNTIL is used to execute a statement until a test condition is true. Figure 4-4 shows a flowchart for the DO-UNTIL structured construct. Notice that the statement is executed repeatedly until the test condition results in a true condition.

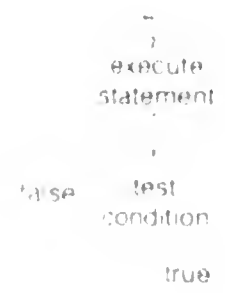

Figure 4-4 DO-UNTIL Flowchart

The following example illustrates the DO-UNTIL statement. In this example the program reads customer records (CUSREC) until one is found with a balance (BALANC) over $500.

```
DO
   READS (1,CUSREC,EOF)
UNTIL BALANC.GT.500
```

The sequential reading (READS) of the file is stopped once a record with a balance (BALANC) exceeding $500 is read. The program direction is then re-directed to perform another task, like creating a list of those records having a balance over $500. In the following example the program creates a list of the records having balances over $500 until CUSNAM is blank.

```
DO
   BEGIN
   READS (1,CUSREC,EOF)
   IF BALANC.GT.500
         BEGIN
         NAME = CUSNAM
         AMT = BALANC
         WRITES (6,PLINE)
         END
   END
UNTIL CUSNAM.EQ.SPACES
```

4.3.6 FOR Statement

FOR will repetitively execute a statement. To best explain how FOR works it is necessary to look at the format:

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

where:

dfield	is a decimal field to be incremented.
initial	is a decimal expression which specifies the *initial* value to be assigned to *dfield*.
final	is a decimal expression which specifies the *final* value for *dfield*.
step	is a decimal expression which specifies the value to add to *dfield* each time through the loop.
statement	is a DIBOL Procedure Division statement.

Prior to each execution of *statement*, *dfield* is tested to determine if it has reached its limit. If *dfield* has not reached its limit, *statement* is executed.

If the loop is exited normally, *dfield* will equal the previous value of *dfield* plus *step*. Modifying the *initial* value, *final* value, or *step* value in the FOR loop has no effect on the execution of the FOR loop.

Figure 4-5 contains a flowchart representation of the FOR statement.

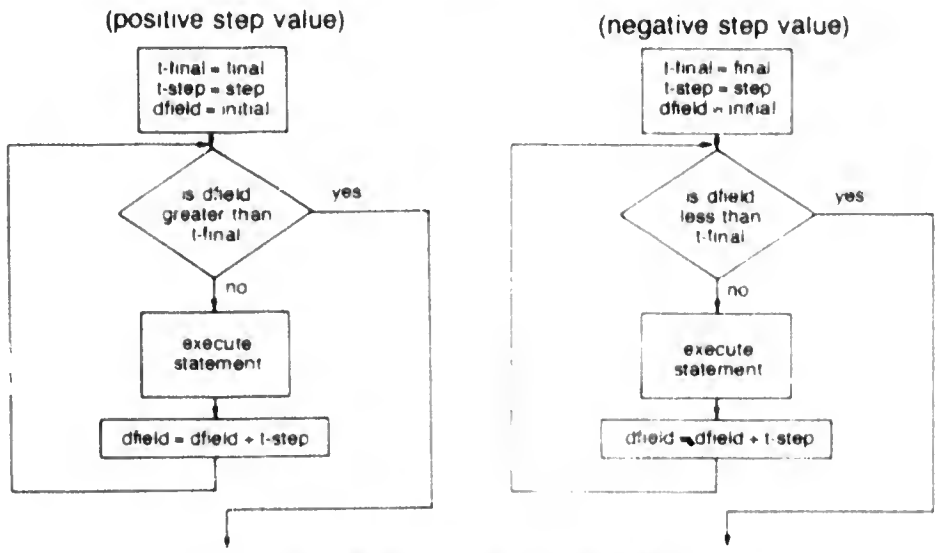

Figure 4-5 FOR Flowchart

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
```

4.3.7 USING Statement

USING conditionally executes a statement from a list of statements based on the evaluation of an expression. The expression is evaluated and then compared with each match expression within the case-label (list of match expressions). USING, like FOR, is best explained by first looking at the format:

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

where:

selection-value
is an alphanumeric field or literal, a decimal expression, or record.

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

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

statement is a DIBOL Procedure Division statement

The match expression *(mexp)* is referred to as a case-label. Each case-label has an accompanying statement which is executed if the case-label matches the *selection-value*. Once this happens, USING is exited. If no match is found, no *statement* within USING is executed.

Figure 4-6 shows a flowchart depicting the USING statement.

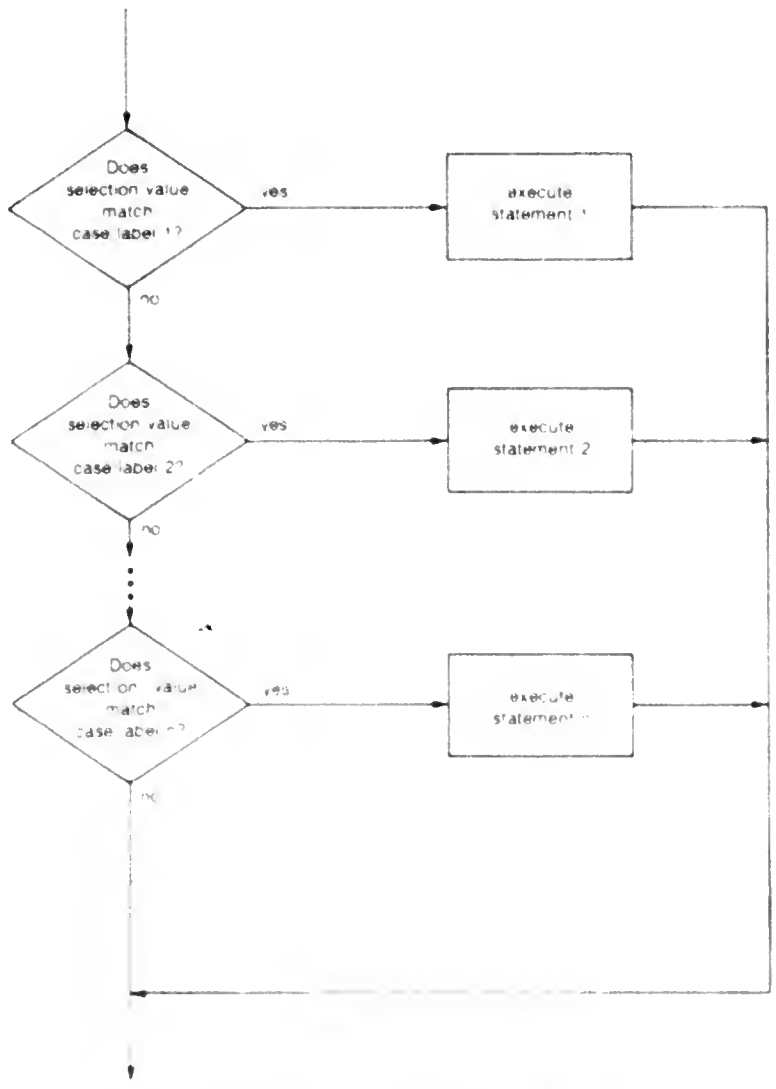

Figure 4-6 USING Flowchart

No match is found if the value to the left of THRU is greater than the value to the right of THRU. The data types of the values in the match expressions *(mexp)* must match. The data type of *selection-value* must match the data type of the match expression *(mexp)*.

The following program displays a message indicating which case of 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
```

All the DIBOL-83 structured constructs, as well as all DIBOL-83 statements, are completely described in The *DIBOL-83 Language Reference Manual*.

CHAPTER 5

ASSIGNMENT, FORMATTING, AND MATH

This chapter introduces the assignment statement, formatting techniques, and DIBOL-83 math Areas such as operator precedence and the INCR statement are also included in this chapter

5.1 ASSIGNMENT STATEMENT

One of the most often used statements in DIBOL is the assignment statement. The assignment statement in DIBOL-83 assigns a value to a field or record and has the following general format

```
field or record to receive the value = source of the value
```

The source may be another field or record, a literal, or an expression.

In the following example, SUM receives a value (5) from the expression 2 + 3

```
SUM = 2 + 3
```

The following example introduces an assignment statement with a variable as part of the source expression. This assignment increments the value of the count by 1

```
COUNT = COUNT + 1
```

There are cases where assignment statements move data between a source and destination of different sizes.

Where the destination has more character positions than the source, excess positions are filled with default values. If the destination field is alphanumeric, the excess positions are filled with spaces. If the destination field is decimal, the excess positions are filled with zeros.

When data is moved into a numeric field, the data is right-justified and truncation occurs on the left-most digits. When data is moved into an alphanumeric field, the data is left-justified and truncation occurs on the right-most characters.

In the following example RATE is a six character field, and NEWRTE is a five character field. Data moved between these fields will have a zero added in the left-most character position of RATE. If NEWRTE contains 05000, RATE will contain 005000.

```
RATE = NEWRTE
```

If 05000 was moved into a three character decimal field, the field would accommodate the three zeros on the right; 05 would be truncated. If ABC were moved into a two character alphanumeric field, the field would accommodate AB; C would be truncated.

5.1.1 Justifying Data in a Field

In cases where there are fewer characters in a field than the field size allows, DIBOL positions the characters in the field by justifying them.

Characters in alphanumeric fields are left-justified. DIBOL positions the characters beginning with the left-most character position and, when all characters have been placed, fills any remaining positions with spaces.

If the name 'William' were entered to the field REPLY, which can accept a name up to 30 characters, REPLY would contain the seven characters W i l l i a m followed by 23 spaces.

REPLY, A30

Characters are right-justified in decimal fields. DIBOL positions the characters beginning with the right-most character position and, when all characters have been placed, fills any remaining positions with zeros.

If the field COUNT contained the value 2, it would contain the character 2 preceded by one leading zero (02).

There is one exception to the above general case: data moved from a decimal field to an alphanumeric field is right-justified in the alphanumeric field and any leading zeros in the decimal field become spaces in the alphanumeric field. For example, if 01 were moved from a decimal to an alphanumeric field, the alphanumeric field would contain a blank and a 1.

Sometimes there is a need to move data from an alphanumeric field into a decimal field. When moving data from an alphanumeric field into a decimal field the data is right-justified. In the following example RECS is a decimal field and ANUM is an alphanumeric field.

```
RECS = ANUM
```

If ANUM contained a 1 and a 2 (12), it would be left-justified.

```
12
```

If this data was moved into RECS, it would be right-justified and filled with leading zeros.

```
00012
```

When DIBOL encounters the spaces in the alphanumeric field ANUM they are ignored and only the characters 1 and 2 are moved.

5.1.2 Formatting Decimal Data for Output

DIBOL provides a way to format decimal data that is to be output to a file or to a terminal. The structure which specifies the format is called an edit mask. An edit mask is a picture of how the output data is to look.

First, the decimal data must be moved to an alphanumeric field because the output field will contain non-numeric characters, and an edit mask can be specified only as part of an assignment statement.

In the following example the data in the decimal field RATE is moved to the output alphanumeric field ARATE. As part of that assignment statement, an edit mask is specified as an alphanumeric literal separated from the source field, RATE, by a comma.

```
ARATE = RATE,'$$$$X XX
```

Each time the program outputs a record to the print file the data in ARATE is formatted according to this edit mask. Each $ and each X can be replaced by a digit moved from RATE into ARATE. When the data is moved, the symbols $ and X are replaced by digits with replacement beginning on the right.

For example, the following table shows several examples of what the result would be for data moved from RATE to ARATE using '$$$$X XX'.

If RATE contains:	**ARATE will contain:**
2345	$23 45
1000	$10 00
34	$0 34
12345	$123 45
123456	$1234 56
0	$0 00

The X is fixed, it is always replaced by a digit. The dollar sign, however, floats, a maximum of one will appear in the output. Digits replace the dollar signs until all digits have been moved from RATE into ARATE. When all data is moved, one dollar sign will remain on the left to show that the output indicates dollars.

If the alphanumeric output field is too small to accommodate the maximum number of characters that could be moved from the decimal field and the formatting characters, then some of the formatting characters could be lost and/or data could be truncated on the left and lost. RATE's field size is six and ARATE's field size is eight. The fields are different sizes because allowance must be made for the decimal point and dollar sign of which each occupies a character position in the alphanumeric output field. ARATE, therefore, must have an edit mask of eight characters to be sure that a dollar sign will always appear in the output.

The following table shows several examples of what the result would be for data moved from RATE to ARATE if both fields had a size of six characters and the edit mask was '$$X XX'.

If RATE contains:	**ARATE will contain:**
2345	$23 45
1000	$10 00
34	$0 34
12345	123 45
123456	234 56
0	$0 00

Study the shaded examples. Where the data is 12345, all of the characters were output because the edit mask can accommodate up to five characters. The dollar sign, however, is lost because the left-most dollar sign is replaced by the 1. Where the data is 123456, the 1 is truncated and lost, because the mask can accommodate only five of the characters.

There are other symbols and options available with edit masks; for a complete discussion of this topic, see the *DIBOL-83 Language Reference Manual*.

5.2 INTEGER ARITHMETIC

DIBOL-83 processes all numeric data as integer data. Legal DIBOL-83 numeric data consists of the integers 0 through 9. Integers do not include decimal points, implied or otherwise.

DIBOL-83 can perform arithmetic using the following operators:

+	addition	8 + 2 = 10
-	subtraction	8 - 2 = 6
*	multiplication	8 * 2 = 16
/	division	8 / 2 = 4

5.2.1 Integer Division

DIBOL-83 performs Integer Division. When data is processed by DIBOL-83 only the integer part of the quotient is retained; remainders are discarded. For instance, the result of the DIBOL-83 operation 1/3 is 0; 5/2 is 2; 3/4 is 0; and 9999/10000 is 0.

If resulting data is to be output via a terminal or printer the data must be formatted so that it (the data) gives a correct result. The rounding operator (#) is used to round off numeric values. This operator has two operands. The first operand specifies the numeric value to be rounded and the second 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 equal to or greater than five.

The following table contains fields and their values and results from expressions using the rounding operator.

Field	Value	Expression	Results
SUM	1357	SUM#2	14
MONEY	-456	MONEY#1	-46
TOTAL	9999	TOTAL2#2	100

The following example uses the values given for SUM and MONEY in the table above to illustrated another way of using the rounding operator.

Expression	Results
(SUM + MONEY)#2	9
(TOTAL/33)#1	30
SUM#2/2	7

In the following program salary is calculated and the result is formatted so as to indicate the actual salary in dollars and cents.

```
RECORD      WORKER
SALARY,             D12             ;Salary carried out to 8 places.
AMOUNT,             A12             ;Field for calculated result.
HRRATE,             D4,1235         ;Hourly rate carried to 2 decimal places.
HRSWKD,             D4,3555         ;Hours worked carried to 2 decimal places.
            PROC
            OPEN (1,O,'TT:')                        ;Open channel 1 to the terminal.
            SALARY = (HRRATE * HRSWKD)#2            ;Calculate the wages and round off.
            AMOUNT = SALARY, '$******XX.XX'         ;Place the result in AMOUNT.
            WRITES (1,AMOUNT)                       ;Write out the contents of AMOUNT.
            CLOSE 1
            END
```

Notice that AMOUNT is a n A (alphanumeric) field. Only alphanumeric fields can be output to a terminal. Other formatting techniques are described in the *DIBOL-83 Language Reference Manual*.

5.3 OPERATOR PRECEDENCE

Like other languages DIBOL-83 maintains operator precedence, i.e., operators are processed from left to right with the exception that multiplication and division are done before addition and subtraction. Operations within parentheses are done first. The following examples illustrate how precedence affects the outcome of a mathematical operation.

For 6 + 4 - 2 the result is 8

For 6 * 4 - 2 the result is 22

For 6 * (4 - 2) the result is 12

For 8/4 - 2 the result is 0

For 8/(4 - 2) the result is 4

These examples do not include all the DIBOL-83 operators and illustrate only a few instances where one operator (for example, parentheses) can affect a mathematical operation. Refer to the *DIBOL-83 Language Reference Manual* for the complete table on operator precedence.

5.4 INCR STATEMENT

It is often necessary to use a counter when doing various types of programming tasks. These tasks can be best served via the INCR statement. The INCR statement increments the value in a decimal field (serving as a counter) by one.

In the following example the field COUNT is incremented by 1 each time the statement is executed. The first time the statement is executed the value of count is incremented from 0 to 1, the second time from 1 to 2, etc.

```
INCR COUNT
```

An assignment statement is required to increment a decimal field by more than 1. For example, COUNT = COUNT + 2.

CHAPTER 6

SUBROUTINES

There are two types of subroutines; internal and external. This chapter will explain the differences between the two and will explain how they are used within a DIBOL-83 program.

6.1 INTERNAL SUBROUTINES

Internal subroutines are units of DIBOL code, present within the program, and used repeatedly throughout the program. An internal subroutine allows lines to be referenced as a unit so that the lines do not have to be repeated within the program each time they are needed. The calling of the subroutine has the same effect as repeating the lines of code in the subroutine. The CALL statement transfers program control to an internal subroutine. When the subroutine executes the RETURN statement, control is returned to the statement logically following the CALL.

In the following example the CALL statement calls the PROFIT subroutine to perform a function that will need to be repeated throughout the program. At the end of the subroutine PROFIT the RETURN statement returns program control to the line beginning with the WRITES statement. At the end of the subroutine TAX, the RETURN statement returns program control to the line which reads PAT = PBT-TAX.

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

            ;Subroutine to calculate profit

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

            ;Subroutine to calculate tax

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

6.2 EXTERNAL SUBROUTINES

External subroutines are units of code which are not contained in the program. These subroutines may be used repeatedly throughout the program via an XCALL statement. This procedure eliminates the need to repeat the lines of code wherever they are needed.

Arguments in the main program may be passed to subroutines. The size of the arguments are specified in the calling program. Also, when passing arguments the argument definitions within the subroutine must correspond in both number and data type with the arguments specified in the XCALL statement in the calling program. The first argument definition specified in the subroutine refers to the first data item specified in the XCALL statement; the second argument definition specified in the subroutine refers to the second XCALL argument, etc. These rules are explained in detail in the *DIBOL-83 Language Reference Manual*.

XCALL transfers control to a subroutine that is external to the calling routine

The format for the XCALL is:

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

The *name* refers to the name of the subroutine being called and *arg* refers to the alpha field, alpha literal, decimal field, decimal literal, expression, or record which contains the subroutine arguments.

The external subroutine is compiled separately, but linked to the calling routine. DIBOL-83 supplies an external subroutine library which contains many useful subroutines. External subroutines may be user-written and compiled using the SUBROUTINE statement.

6.2.1 Universal External Subroutines

DIGITAL supplies an external subroutine library with all of its DIBOL-83 implementations. This library, known as the Universal External Subroutine Library (UESL) contains many useful subroutines which provide similar capabilities under each operating system. These subroutines are described in the *DIBOL-83 Language Reference Manual*.

The following example uses the UESL subroutine JBNO to illustrate how an external subroutine is used JBNO returns the job number and the following program will display the job number

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

CHAPTER 7

ARRAYS AND SUBSTRINGS

This chapter introduces the concept of arrays and subscripting in DIBOL-83

7.1 DIBOL-83 ARRAYS

An array is a group of related fields that share the same data type, field size, and symbolic name (array name). Arrays hold large areas of information from which you can use over and over again during program execution. The number of fields in the array is identified by the array field count. The array field count is specified before the data type specification

The following example specifies an array of four fields, each field has the decimal data type (D) and a field size of five. NEWRTE, therefore, is an array of four, five-character decimal fields

```
NEWRTE, 4D5
```

NOTE

The method for identifying specific fields in an array is called array subscripting. This is explained later in this chapter

Arrays may contain initial values as do fields. If no initial value is specified by the programmer, each of the fields in the array will contain spaces if the array is alphanumeric or zeros if the array is decimal. If initial values are to be programmer-specified, each field must have its value specified separately

In the following example, the array NEWRTE has four fields. Each field's initial value is specified by the programmer. The following table shows how the initial values correspond to the array's fields

```
NEWRTE,4D5, 05000,05500,
            06000,07000
```

NEWRTE	VALUE
Field 1	05000
Field 2	05500
Field 3	06000
Field 4	07000

If the first field of NEWRTE is referenced in the Procedure Division, it will have the value 05000, which in this case represents 50. The second field will have the value 05500, etc

Where the programmer specifies initial values, fields may not be skipped. If the second field in an array is to have an initial value specified, the first field must also have an initial value specified. If there were a third field, it could be left unspecified, and it would contain the DIBOL-supplied initial value for its data type. For example consider the following array definition:

```
ARRAY ,3A5,'ABCDE','FGHIJ'
```

The array would contain the following:

ARRAY	VALUE
Field 1	ABCDE
Field 2	FGHIJ
Field 3	(5 blanks)

7.2 ARRAY SUBSCRIPTING

Array subscripting identifies specific fields within an array. Arrays are referenced in the Procedure Division using the array name and a single subscript. The subscript is specified in parentheses after the array name and must be a decimal literal or a decimal field name. The subscript's value determines which field in the array is referenced.

For example, if there were an array named COST containing three fields, the first field would be referenced as COST(1) and the second and third fields as COST(2) and COST(3). If no subscript is specified, the first field is accessed. COST and COST(1) reference the same field. COST, therefore, is the array name and COST(1), COST(2), and COST(3) are the names of fields in the array.

```
RATE = NEWRTE(RATECD)
```

The example refers to array NEWRTE. This example uses the decimal field RATECD as the subscript. NEWRTE contains four fields so RATECD should equal 1, 2, 3, or 4. A value of 1 in RATECD references the first field in NEWRTE. A value of 2 in RATECD references the second field, etc.

The values in NEWRTE are programmer initialized. If RATECD is 1, NEWRTE(RATECD) contains 05000; if RATECD is 2, NEWRTE(RATECD) contains 05500, etc.

7.3 SUBSTRING SUBSCRIPTING

Substring subscripting is a method for directly referencing a specific substring residing within a record or field. Subscripts are decimal values specified as literals or field names, which are enclosed in parentheses following the subscripted field or name.

The subscripts identify specific character positions in the record or field being subscripted. The first subscript specifies a substring's beginning character position. The second subscript specifies the substring's ending character position.

A simple way of understanding substrings is to think of the alphabet. In the following example the field STRING contains the alphabet. The subscripts are referencing beginning and ending character positions.

```
STRING, A26, "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

STRING(1,4) is "ABCD"
STRING(5,5) is "E"
STRING(9,14) is "IJKLMN
STRING(25,26)is "YZ"
```

In the following example, the first subscript specifies the first character position in ANSWER (defined in the Data Division as A20). The second subscript specifies the third character position. This reference accesses positions 1-3 in ANSWER; positions 4-20 are ignored. If the subscripts were 5 and 15, positions 5 through 15 inclusive would be accessed, and all others would be ignored.

```
READS (1,ANSWER(1:3))
```

If only one specific position is to be referenced, then the number for that position is specified for both subscripts.

```
READS (1,ANSWER(1:1))
```

CHAPTER 8

INPUT/OUTPUT OPERATIONS

The purpose of this chapter is to familiarize you with the common record and file operations performed in DIBOL-83. This information is presented in a general light and it is recommended that you familiarize yourself with the file structures available with your system before reading this chapter.

The statements commonly used in record and file operations are OPEN, CLOSE, READ, READS, WRITE, WRITES, STORE, and DELETE. These statements will be discussed in this chapter.

8.1 OPEN STATEMENT

The OPEN statement establishes a line of communication between a program and either a disk file or an I/O device. OPEN may be followed by as many as three arguments.

The first argument specifies the channel number. This number may be a literal or field name with a decimal value that is system specific. The effect of this argument is to associate the channel number with the file or device being opened. Other I/O statements use this association by referring to the opened file or device with the channel number.

The second argument specifies the mode. The mode determines which I/O operations will be allowed over the channel being opened. The mode is specified by one of a group of mode indicators, such as I for input, O for output, U for update.

The third argument specifies the file or I/O device that is being opened. This argument may be an alphanumeric record, field name, or literal.

In the following example OPEN opens the terminal for input and/or output. When the terminal is the device being opened, either I or O may be specified. The example opens channel one for output to the terminal. The channel specification is the decimal literal 1. O specifies output mode. 'TT:' is an alphanumeric literal which specifies the terminal.

```
OPEN (1,O,'TT:')
```

The following example for the OPEN statement opens channel two. In this example, one is the channel for the terminal; the same number cannot be used more than once for concurrently opened files or devices. Channel one can be used again in this program only if an intervening CLOSE statement disassociates it from the terminal.

```
OPEN (2,O,FILNAM)
```

The mode indicator is O. This specifies that the file specified in the third argument will be opened for output. Opening a file in this mode causes a new file to be created.

The file specification argument (FILNAM) is the name of an alphanumeric field defined in the Data Division. The desired file specification is entered into FILNAM, possibly by an initial value, or read from the terminal. When the program terminates execution, that new file will reside on the system and contain the program's output.

A second OPEN statement opens a channel to the line printer.

If the terminal channel was closed before this OPEN statement the channel number (1) associated with the terminal is available for use. Output mode is specified and the device is the line printer (LP:).

```
OPEN (1,O,'LP:')
```

8.1.1 Relative and Indexed Files

Records in a relative file each have an ordinal position within the file and may be accessed directly by their record numbers. The record number is the same as the ordinal position; the 10th record is record number 10.

When a relative file is created, it must be identified as a relative file and the record size must be specified. The mode indicator is followed by a qualifier; the qualifier :R specifies a relative file. Record size is specified with a keyword argument, RECSIZ:n. The keyword RECSIZ is followed by a colon (:) and a positive integer. The integer (n) indicates the record size.

In this example, the relative file is opened for access via channel two.

```
OPEN (2,O:R,'ROOMS.DDF',RECSIZ:67)
```

Since the file is being created, output (O) mode is specified, and the :R specifies it will be a relative file. The third argument specifies that the file is to be named ROOMS.DDF.

The fourth argument indicates that this file's record size is 67; each record written to ROOMS.DDF must be 67 characters long. 67 is the collected field sizes of RECORD ROOM.

The OPEN statement can open an existing file for update. Update mode permits data to be added, deleted, or modified.

In the following example a relative file is opened for update by specifying the letter U as the second argument to OPEN and R (for relative file) as the submode.

```
OPEN (2,U:R,'ROOMS.DDF')
```

In this next example the OPEN statement introduces Indexed files. Indexed files can be opened for input and update only; the mode indicators are I:I for input and U:I for update.

This example opens the Indexed file RESERV.DAT for update.

```
OPEN (2,U:I,'RESERV.DAT')
```

Indexed files require much more information about the file before it can be created. Because of this DIBOL does not allow the creation of Indexed files directly with the OPEN statement. Consult your system's DIBOL User's Guide for information concerning the creation of indexed files on your particular system.

8.2 CLOSE STATEMENT

The CLOSE statement closes channels opened by an OPEN statement. CLOSE completes all pending operations to the file or device being closed, and disassociates the channel number from the opened file or device.

A single CLOSE statement can close only one channel. A separate CLOSE statement is required for each channel to be closed. CLOSE is followed by a decimal field or literal which specifies the channel to be closed. The value must be the same as was specified in the OPEN statement when the channel was opened.

All open channels should be closed before program execution terminates. It is possible for data to be lost or corrupted if the channel is not closed.

This next example closes channel one. If processing were to continue, channel one could be used again to refer to another device via another OPEN statement.

```
CLOSE 1
```

8.3 INDEXED VS. SEQUENTIAL

Files are Relative, Indexed, or Sequential. READ and WRITE perform random access I/O and may be used only on Indexed and Relative files. READS and WRITES are used to read and output sequential files.

8.4 READ AND WRITE STATEMENTS

READ is a random-access input statement. It transfers the specified data record from an input file to a record or field in the program. READ may be used on either Relative or Indexed files. READ is followed by three arguments.

The first argument specifies the channel number; the second specifies the record or field to receive the input. When accessing a Relative file the third argument is required to specify a record number. The record number may be either a decimal literal or field name. The value of the record number cannot be less than 1 or greater than the total number of records in the file being accessed.

```
READ (2,ROOM,RECNUM)
```

In this example, READ reads input via channel 2 into the record ROOM. The third argument, RECNUM, contains a value interpreted as the number of a record to be read into ROOM. If the value of RECNUM is 6, then the 6th record in the file will be read into ROOM.

The READ statement can also be used to read records from an Indexed file based upon a specified key. The first argument is the channel number associated with the file; the second argument specifies the record area the record is to be read into; and the third argument specifies the name of the key field. When the READ statement executes, this argument must contain the same value as the value of the desired record's key field.

```
READ (2,RES,NAME)
```

In the example, RES is being read via channel 2. When READ is executed, the record whose key-field value matches the data in NAME is read into RES. If the field name contains "JONES", READ will read into RES the record where JONES is the value in the key field.

WRITE is a random-access output statement. It transfers data from a field or record in the program to a specified record in a file. It may also be used on either Relative or Indexed files. WRITE is followed by three arguments. The first argument specifies the channel number; the second argument specifies the record to be written; and for Relative files, the third argument specifies the position in the file where the record is to be written. This third argument may be either a decimal literal or a decimal field name where the field's value is the record number. This argument allows access directly to any record in the file by specifying the record's number.

In the following example WRITE writes HEADER via channel 2 to record number 1, the first record in the file.

```
WRITE (2,HEADER,1)
```

The WRITE statement, when used with an Indexed file, modifies existing records and is followed by three arguments. The first argument is the channel number; the second argument is the record name; and the third argument specifies the key field in the record. The value in the key field is used to write the modified record back to the file.

The record must have previously been read with a READ statement.

```
WRITE (2,RES,NAME)
```

In this example, WRITE is writing RES via channel 2. The key value used to determine the record's place in the file is contained in NAME.

8.5 READS AND WRITES STATEMENTS

The READS statement does sequential input. It transfers the next available data record or field from an input file or device to a record or field in the program.

READS is followed by two arguments. The first argument may be either a decimal field or literal which specifies the channel number over which the input will occur. This channel must have been opened previously by an OPEN statement. The second argument specifies a record or field which is to receive the data being input.

In this example, READS reads each record from the input file. READS can detect when all records have been read, but there must be provision to specify where to transfer control when READS has read the last record. This can be handled with a third argument to READS.

When a third argument is used with READS, the argument must be a statement label separated from the second argument by a comma. The statement label identifies where control is to be transferred when the last record has been read.

When READS has read the last record, control transfers to the statement identified by NOMORE

```
READS (2,CUSREC,NOMORE)
```

The WRITES statement writes an alphanumeric record, field, or literal via a specified channel to the next available space in a file or to a device This is called sequential output. WRITES is followed by 2 arguments enclosed by parentheses

The first argument specifies the channel number over which the output will occur. The second argument is the name of the alphanumeric record, field, or literal which contains the information to be output.

In a sequential file opened for output, WRITES adds records to the file in sequence by writing them to the end of the file. A pointer is maintained to the end of the file so WRITES will know where to add the next record.

The following code writes three records to the file opened on channel 1.

```
WRITES (1,REPLY)
WRITES (1,"This is the first record")
WRITES (1,"This is the second record")
WRITES (1,"This is the third record")
```

In this example, WRITES uses the channel (1) associated with the line printer so that the records written by WRITES will be printed directly on the printer.

```
WRITES (1,CUSREC)
```

8.6 STORE AND DELETE STATEMENTS

The STORE statement adds new records to Indexed files. STORE is followed by three arguments.

The first argument specifies the channel number, the second argument specifies the name of the record to be written to the file; the third argument specifies the name of the key field in the record. The value in the key field is used to determine where the record will be stored in the file.

In the following example, STORE is writing the record RES via channel 2, and the key value is in the field NAME. If the value in NAME were JONES, for example, the record would be placed in the file after records having names beginning with I and before records having names beginning with K.

```
STORE (2,RES.NAME)
```

The DELETE statement removes records from an Indexed file. DELETE is followed by two arguments.

The first argument specifies the channel number associated with the file, the second argument specifies the name of the key field. For DELETE to execute, the record to be deleted must have been read with a READ or READS statement, the record that is deleted is the one most recently read.

```
IF (REPLY .EQ. 'Y') DELETE (2,NAME)
```

In this example, DELETE removes whichever record has the same value in its key field as is contained in NAME.

8.7 TERMINAL I/O

DIBOL-83 is an interactive language and thus terminal I/O is commonplace when using DIBOL-83. The ACCEPT and DISPLAY statements are used for terminal character I/O. READS and WRITES may also be used for terminal I/O, their use is essentially similar to file I/O operations.

8.7.1 ACCEPT Statement

The ACCEPT statement reads single characters from an input device

ACCEPT is followed by two arguments. The first argument is the channel number (as specified in a previous OPEN statement) associated with the device where the character is being input, and the second argument is the name of a field or record where the input character is to be placed

In the following example ACCEPT reads the first available character from the terminal input buffer into the first character position of the field REPLY

```
ACCEPT (1,REPLY(1,1))
```

ACCEPT does not require a terminator, it is complete as soon as a character is read from the input buffer

When accepting into an alphanumeric field (or record), the character is moved to the leftmost character position of the field according to the rules for moving alphanumeric data. When accepting into a decimal field the character is moved into the field according to the rules for decimal data

8.7.2 DISPLAY Statement

The DISPLAY statement performs output to character-oriented devices, such as terminals and line printers. Under certain conditions DISPLAY may also be used to write files which can be printed at a later time. One of the most common uses of DISPLAY is to format output to a video terminal. The examples in this section use the American National Standards Institute (ANSI) terminal control sequences

The first argument in the DISPLAY statement is a channel number (as specified in a previous OPEN statement) which determines where all output for that statement will occur

Subsequent arguments may be records, fields, or literals of either data type. If the data is alphanumeric the contents of a record, field or literal are displayed on the output device. Decimal data, however whether specified as a literal or as a decimal field name, is interpreted as a decimal code representing a character in the character set

In the following example, the channel number is 1. The second argument is the decimal literal 27 which is the decimal code for the escape character. Here the escape character is part of an escape sequence

```
DISPLAY (1,27,'[2J')
```

An escape sequence consists of an escape character followed by one of a group of unique function codes. The codes are terminal dependent and are not displayed but specify where output will occur on the terminal screen or invoke certain control functions of the terminal

NOTE

Consult the users guide of your terminal to determine which escape sequences and terminal functions are available to you. The escape sequence used in the example is for a VT100 terminal

The third argument in the example is the function code [2J. This code follows the escape character completing the escape sequence. codes may be specified as alphanumeric literals, as in this case, or may be the contents of a specified alpha field or record.

This sequence specifies that the screen is to be cleared. this DISPLAY statement therefore clears the entire screen and consists only of the escape sequence itself.

In the next example, the second argument also specifies the escape character (27). The third argument ('[3,5H') completes the escape sequence. The numbers 3 and 5 indicate row and column respectively. This escape sequence positions the cursor to row 3, column 5. The specifications for row and column can vary from 1 to the total number of rows or columns available on the terminal.

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

The fourth argument is the alphanumeric field ENTREC. The data it contains is displayed at row 3, column 5 on the terminal screen. A fifth argument is an alphanumeric literal containing a space so the data in ENTREC will be displayed followed by a space.

APPENDIX A

DIBOL-83 CHARACTER SET

Table A-1 shows the DIBOL-83 Character Set. The DIBOL-83 Character Set contains 128 ASCII characters. The table shows corresponding decimal codes. All characters may be used for data input from the terminal and output to the terminal and printer. DIBOL-83 stores both alphanumeric and decimal data in character code form.

Table A-1
DIBOL-83 CHARACTER SET

DEC	ASC	DEC	ASC	DEC	ASC	DEC	ASC	
000	^ @ <NUL>	032	< SPACE >	064	@	096	`	
001	^ A	033	!	065	A	097	a	
002	^ B	034	"	066	B	098	b	
003	^ C	035	#	067	C	099	c	
004	^ D	036	$	068	D	100	d	
005	^ E	037	%	069	E	101	e	
006	^ F	038	&	070	F	102	f	
007	^ G <BEL>	039	'	071	G	103	g	
008	^ H <BS>	040	(	072	H	104	h	
009	^ I <HT>	041	)	073	I	105	i	
010	^ J <LF>	042	*	074	J	106	j	
011	^ K <VT>	043	+	075	K	107	k	
012	^ L <FF>	044	,	076	L	108	l	
013	^ M <CR>	045	-	077	M	109	m	
014	^ N	046	.	078	N	110	n	
015	^ O	047	/	079	O	111	o	
016	^ P	048	0	080	P	112	p	(-0)
017	^ Q	049	1	081	Q	113	q	(-1)
018	^ R	050	2	082	R	114	r	(-2)
019	^ S	051	3	083	S	115	s	(-3)
020	^ T	052	4	084	T	116	t	(-4)
021	^ U	053	5	085	U	117	u	(-5)
022	^ V	054	6	086	V	118	v	(-6)
023	^ W	055	7	087	W	119	w	(-7)
024	^ X	056	8	088	X	120	x	(-8)
025	^ Y	057	9	089	Y	121	y	(-9)
026	^ Z	058	:	090	Z	122	z	
027	^ [<ESC>	059	;	091	[	123	{	
028	^ \	060	<	092	\	124	|	
029	^]	061	=	093	]	125	}	
030	^ ^	062	>	094	^	126	~	
031	^ _	063	?	095	_	127	< DEL >	

GLOSSARY

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

alphabetic
A character set that contains only letters.

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

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., B*C).

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

byte
A group of eight bits considered as a unit.

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

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

character string
A connected linear sequence of characters.

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

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

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

DEC
Acronym for Digital Equipment Corporation.

decimal
Refers to a base ten number.

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

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

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

expressions
Variables, constants, or arithmetic expressions made up of variables, constants, and the operators –, + , -, , and /.,

fatal error
An error which terminates program execution.

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

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

file specification (filespec)
The general file name.

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

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

indexed files
Indexed files are Indexed Sequential Access Method files.

input
Data flowing into the computer.

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

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

justify
The process of positioning data in a field whose size is larger than the data. In 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.

location
Any place where data may be stored.

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

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

mass storage device
A device having large storage capacity.

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

memory
The computer's primary internal storage.

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

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

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

nest
To embed subroutines, loops, or data in other subroutines or programs.

object program
A file which is output by the compiler or assembler.

output
Data flowing out of the computer.

parameter
A variable that is given a constant value for a specific purpose or process

primary key
See key.

random access
Same as to direct access.

RECORD
A statement that reserves memory.

record redefinition
The technique of specifying several different record formats for the same data. Special rules apply.

screen column number
The number which indicates the order of the vertical lines on 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.

source program
A program written in the DIBOL-83 language.

statement
An instruction in a source program.

string
A connected linear sequence of characters.

subscript
A designation which clarifies the particular parts (characters, values, records) within a larger grouping or array.

syntax
The rules governing the structure of a language.

system configuration
The combination of hardware and software that make up a usable computer system.

trappable error
An error condition which may be trapped

unary operator
An operator, such as + or -, which acts upon only one variable or constant (e.g., A = -C)

variable
A quantity that can assume any one of a set of values

variable-length record
A file in which the data records are not uniform in length. Direct access to such records is not possible

verify
To determine if a transcription of data has been accomplished accurately

zero fill
To fill the remaining character positions in a numeric field with zeros

zoned decimal
A contiguous sequence of up to 18 bytes interpreted as a string of decimal digits (1 digit per byte). The sign is stored as the high order bit in the low order byte

INDEX

INDEX (CONT.)

Please cut along this line.

INTRODUCTION TO DIBOL-83
Order No. AA-P042B-TK
April 1984

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. Problems with software should be reported on a Software Performance Report (SPR) form. If you require a written reply and are eligible to receive one under SPR service, submit your comments on an SPR form.

Did you find errors in this manual? If so, specify by page.

_____

_____

_____

_____

_____

_____

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

_____

_____

_____

_____

_____

_____

Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

_____

_____

_____

_____

_____

_____

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Non-programmer interested in computer concepts and capabilities

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code or Country _____

— — — — —Do Not Tear - Fold Here and Tape —

digital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO 33 MAYNARD MASS

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Applied Commercial Engineering MK1-2/H32
Continental Boulevard
Merrimack N H 03054

ATTN Documentation Supervisor

— — — — Do Not Tear - Fold Here and Tape —

Cut Along Dotted Line