

# MDS Series 21

System Software Manual

## MOBOL REFERENCE MANUAL



**MDS**

**MOHAWK DATA SCIENCES**



**SERIES 21**

**MOBOL**

**REFERENCE MANUAL**

FIRST EDITION  
RELEASE 7.0

The following are Trademarks of Mohawk Data Sciences Corp., Parsippany, N.J.  
Mohawk Data Sciences—Canada, Ltd. Registered User.



MDS

MOBOL

SERIES 21

Form No. M-2612-1078 ©1978 Mohawk Data Sciences Corp. Printed in USA.



## FOREWORD

This reference manual is designed to introduce Series 21 programming personnel to the MOBOL programming language. Familiarity with Series 21 components as well as a general knowledge of computer programming principles are assumed throughout.

This publication consists of 8 major sections which are described below:

Section 1 presents an overview; it describes the function of the MOBOL compiler and presents the basic structure of a MOBOL source program.

Section 2 outlines the procedures involved in generating a MOBOL source program.

Sections 3 through 6 define the four main components of MOBOL source code and all respective statements.

Section 7 discusses Input/Output operations.

Section 8 discusses STATION Input/Output operations.

Other publications which may be of interest to the reader are:

*MDS Series 21 Operator's Guide* (Form No. M-2611)

*MDS Series 21 System Generation User's Guide* (Form No. M-3922)

*MDS Series 21 Display Messages Reference Manual* (Form No. M-3925)

*MDS Series 21 Binary Synchronous Communications Reference Manual* (Form No. M-3921).

These publications can be obtained through your local MDS representative, or from Mohawk Data Sciences Corp., Palisade Street, Herkimer, N.Y. 13350 (attention: Distribution Dept. 374).

Questions and comments related to the content of this publication should be submitted on the Reader Comment Form located in the back of this publication.



# MOBOL LANGUAGE REFERENCE MANUAL

INTRODUCTION .....	ix
MANUAL NOTATIONS .....	xi
SECTION 1:	
OVERVIEW .....	1-1
MOBOL Compiler .....	1-1
Source Program Structure .....	1-3
SECTION 2:	
CONSTRUCTING A MOBOL SOURCE PROGRAM .....	2-1
Designing Screen Displays .....	2-1
Generating MOBOL Source .....	2-2
Data Definition Section Requirements .....	2-3
Execution Section Requirements .....	2-4
SECTION 3:	
COMMENT STATEMENTS AND COMMENT FIELDS .....	3-1
Comment Statements .....	3-1
Comment Fields .....	3-1
SECTION 4:	
COMPILER DIRECTIVES .....	4-1
TITLE .....	4-1
EJECT .....	4-1
START .....	4-2
END .....	4-2
SECTION 5:	
DATA DEFINITION STATEMENTS .....	5-1
The Input/Output Descriptor Statement (IOD) .....	5-3
The Record Statement .....	5-5
RCD .....	5-5
RCD-Array .....	5-16
The Key Entry Table Statement (KET) .....	5-18
The Equate Statement (EQU) .....	5-28
Record Remapping .....	5-29
SECTION 6:	
EXECUTION STATEMENTS .....	6-1
Syntax Conventions .....	6-3
Semantic Conventions .....	6-5
Interpretation of Literal Values By	
Instruction Type .....	6-5
Indexed Data References .....	6-6
Explicit Notation .....	6-6
Implicit Notation .....	6-7
Non-Indexed Data References .....	6-8
Move Statements .....	6-11
MOVE LEFT AND FILL .....	6-12
MOVE RIGHT AND FILL .....	6-13
FILL .....	6-14
MOVE LEFT, NO FILL .....	6-15
MOVE RIGHT, NO FILL .....	6-16

## MOBOL LANGUAGE REFERENCE MANUAL (Cont'd.)

SECTION 6 (Cont'd.)	Decimal Arithmetic Statements .....	6-17
	DECIMAL EQUATE .....	6-18
	DECIMAL ADD .....	6-19
	DECIMAL SUBTRACT .....	6-20
	DECIMAL MULTIPLY .....	6-21
	DECIMAL DIVIDE .....	6-22
	Binary Arithmetic Statements .....	6-23
	BINARY ADD .....	6-24
	BINARY SUBTRACT .....	6-25
	BINARY MULTIPLY .....	6-26
	BINARY DIVIDE .....	6-27
	Boolean Statements .....	6-28
	AND .....	6-29
	OR .....	6-30
	XOR (Exclusive OR) .....	6-31
	Editing Statements .....	6-32
	BINARY .....	6-33
	DECIMAL .....	6-34
	JUSTIFY LEFT .....	6-35
	JUSTIFY RIGHT .....	6-36
	PICTURE/EDIT .....	6-37
	CURRENCY .....	6-42
	NUMBER EDIT .....	6-43
	TRANSLATE .....	6-44
	COMPRESS .....	6-45
	DECOMPRESS .....	6-49
	HEX .....	6-52
	UNHEX .....	6-53
	Control Statements .....	6-54
	UNCONDITIONAL GO .....	6-55
	STOP .....	6-56
	CONDITIONALS .....	6-57
	COMPUTED GO .....	6-66
	CASE .....	6-67
	ERROR TEST .....	6-68
	Subroutine Statements .....	6-69
	PERFORM .....	6-70
	ENTRY .....	6-71
	EXIT .....	6-73
	I/O Statements .....	6-74
	OPEN .....	6-75
	CLOSE .....	6-76
	READ .....	6-77
	WRITE .....	6-78
	CHECKEOD .....	6-79
	DELETE .....	6-80
	FREESPACE .....	6-81
	INSERT .....	6-82
	READLOCK .....	6-83
	READNEXT .....	6-84

## MOBOL LANGUAGE REFERENCE MANUAL (Cont'd.)

SECTION 6 (Cont'd.)	<ul style="list-style-type: none"> <li>RELEASE ..... 6-85</li> <li>SETEOD ..... 6-86</li> <li>BACKSPACE ..... 6-87</li> <li>MARK ..... 6-88</li> <li>REWIND ..... 6-89</li> <li>REWINDLOCK ..... 6-90</li> <li>SKIPFILE ..... 6-91</li> <li>CHECKFORMS ..... 6-92</li> <li>PRINT ..... 6-93</li> <li>SETFORMS ..... 6-94</li> <li>SENDEOF ..... 6-95</li> <li>STATION I/O Statements ..... 6-96               <ul style="list-style-type: none"> <li>KENTER ..... 6-97</li> <li>KVERIFY ..... 6-98</li> <li>RESUME ..... 6-99</li> <li>RESUMERR ..... 6-100</li> <li>ERROR ..... 6-101</li> <li>NOTIFY ..... 6-102</li> <li>READSCREEN ..... 6-103</li> <li>READKEY ..... 6-104</li> </ul> </li> <li>Special Purpose Statements ..... 6-105               <ul style="list-style-type: none"> <li>CHECKDIGIT ..... 6-106</li> <li>GETTIME ..... 6-107</li> <li>SETTIME ..... 6-108</li> <li>SAME ..... 6-109</li> <li>STRING ..... 6-110</li> </ul> </li> </ul>
SECTION 7:	<ul style="list-style-type: none"> <li>INPUT/OUTPUT OPERATIONS ..... 7-1</li> <li>I/O Statements In the Data Definition Section ..... 7-1</li> <li>I/O Statements In The Execution Section ..... 7-2</li> <li>KEYWORD CLASSIFICATION ..... 7-3               <ul style="list-style-type: none"> <li>Keywords Used In The Data Definition Section ..... 7-3</li> <li>Keywords Used In The Execution Section ..... 7-4</li> <li>Keywords Used In Both the Data Definition And The Execution Section ..... 7-6</li> </ul> </li> <li>DISK/DISKETTE I/O Operations ..... 7-8               <ul style="list-style-type: none"> <li>Basic Access Method ..... 7-8</li> <li>Sequential Index Access Method ..... 7-17</li> <li>Random Index Access Method ..... 7-18</li> </ul> </li> <li>TAPE I/O Operations ..... 7-22</li> <li>PRINTER I/O Operations ..... 7-25               <ul style="list-style-type: none"> <li>VFU (Vertical Forms Unit) ..... 7-27</li> </ul> </li> <li>Compatible Channel I/O Operations ..... 7-29</li> </ul>
SECTION 8:	<ul style="list-style-type: none"> <li>STATION I/O OPERATIONS ..... 8-1</li> <li>Keywords Used In The Data Definition Section ..... 8-2</li> <li>Keywords Used In The Execution Section ..... 8-3</li> <li>KENTER ..... 8-5               <ul style="list-style-type: none"> <li>Field Mode ..... 8-7</li> <li>Screen Mode ..... 8-9</li> <li>Control Keys ..... 8-11</li> </ul> </li> </ul>



## MOBOL LANGUAGE REFERENCE MANUAL (Cont'd.)

SECTION 8: (Cont'd.)	KVERIFY .....	8-16
	RESUME .....	8-17
	RESUMERR .....	8-20
	ERROR .....	8-22
	NOTIFY .....	8-23
	READSCREEN .....	8-24
	READKEY .....	8-25
APPENDIX A:	DIRECTORY OF FIGURES AND TABLES .....	A-1
APPENDIX B:	RESERVED WORDS IN MOBOL .....	B-1
APPENDIX C:	MOBOLIST .....	C-1
APPENDIX D:	MOBOL CROSS REFERENCE PROGRAM .....	D-1
APPENDIX E:	ERROR CODES .....	E-1
APPENDIX F:	GLOSSARY OF TERMS .....	F-1
APPENDIX G:	CHECKDIGIT ALGORITHMS .....	G-1
APPENDIX H:	INSTRUCTIONS FOR THE MOBOL COMPILER .....	H-1
APPENDIX I:	SYSTEM ERROR MESSAGES THAT OCCUR DURING EXECUTION OF USER-DEFINED APPLICATIONS .....	I-1
INDEX		

## INTRODUCTION

This manual defines Mohawk Data Science's MOBOL (Mohawk Business Oriented Language). MOBOL programs are developed by users of Series 21 Distributed Data Processing Systems to perform data entry, data verification, data validation and transaction processing.

MOBOL is a relatively free-form high level programming language designed to meet the needs of an operator-interactive environment. In this environment, the operator communicates interactively with the system through a keyboard and a video display screen (CRT). Using the keyboard, the operator enters data or makes selections in response to the programmed prompts displayed on the screen. Based on the data entered or the selection made, the program carries out a pre-programmed series of steps. This cycle continues until all data has been entered and all required selections are made.

Using MOBOL, A Series 21 System may be directed to perform the following functions:

### STATION HANDLING

- Prompting
- Accepting data
- Verifying data

### INPUT/OUTPUT

- Data recording
- Data retrieval
- Record update
- Report printing

### DATA HANDLING

- Compare
- Arithmetic
- Logical
- Edit
- Move



## MANUAL NOTATIONS

Conventions used to represent statement formats throughout this manual are described below. A sample statement is shown.

1. UPPER CASE words and punctuation characters must be coded exactly as shown in the statement formats.
2. { } are known as Braces. Braces enclose two or more statement elements where one and only one of the elements must be selected.
3. Spaces are not considered to be significant except when delimited in a literal.
4. [ ] are known as brackets. Any element enclosed in brackets is optional to the statement.
5. . . . are known as Horizontal Ellipses.  
.  
: are known as Vertical Ellipses. Ellipses indicate that the elements or statements may be repeated on subsequent lines.
6. Scripts 1, 2, . . . , n or 1, 2, . . . , m are used to:
  - a. indicate the number of subsequent elements that will follow;
  - b. indicate distinctions of unique elements of the same type.

$$\text{PRINT (IOD-name, buffer, ERR:sn-1, } \left. \begin{array}{l} \text{EOM:} \\ \text{EOF:} \end{array} \right\} \text{sn-2)}$$

**Sample MOBOL statement**



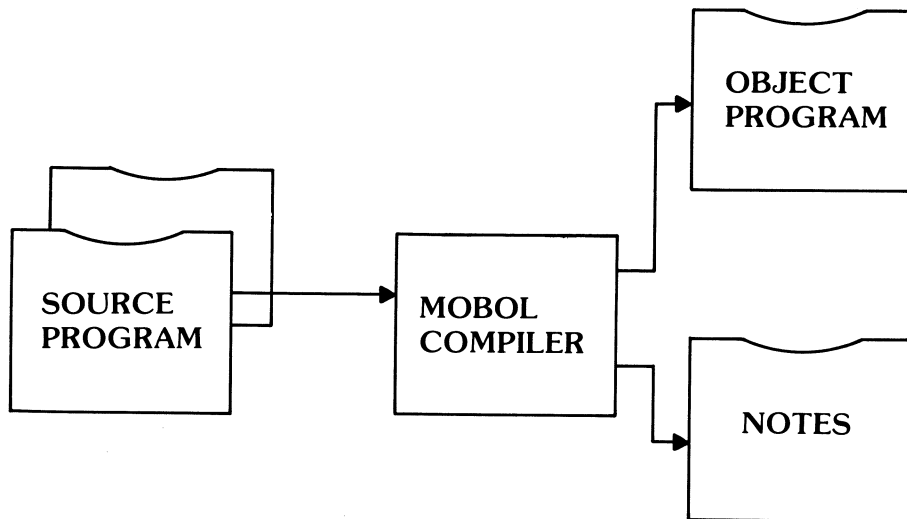
## SECTION 1: OVERVIEW

Once a distributed processing application for Series 21 is identified, the following steps are performed:

1. Gather information regarding application requirements.
2. Analyze requirements and produce system block diagrams.
3. Produce detailed flowchart of program steps.
4. Translate steps of the flowchart into MOBOL source statements (see Figure 2-2 of Section 2 for a sample of the MOBOL Coding Form).
5. Transcribe the resulting MOBOL program to DISKETTE. For exceptionally large applications, multiple diskettes may be required to hold the source program. (A data entry program may be used for this operation.)
6. Compile the source program using the MOBOL Compiler.

### MOBOL COMPILER

A source code statement is not directly executable; i.e., it is not in a form readily understood by the Series 21 System hardware. To be executed by the computer, a MOBOL source program must be translated by the compiler into an object program. An object program is a series of instructions especially designed for machine decoding. Using MOBOL, a programmer expresses a problem in a language he readily understands; the resulting source program is then translated by the compiler into the object language that the system understands. Compilation of MOBOL source language into object language is performed only once since the object program can be re-executed as often as required. The compilation process is diagrammed below in Figure 1-1.



**Figure 1-1** Compilation Process

The compiler produces an object file and a NOTES file. The object file is the primary compiler output; it contains the compiled object program. The NOTES file contains summary information about the compilation; also, it contains notes about detected errors (if any), which are called diagnostic messages.

A MOBOL source program and/or its NOTES file may be printed using the MOBOLIST utility program. A sample listing from the MOBOLIST utility is presented in Figure 1-2. Detailed explanation of MOBOLIST is presented in Appendix C of this manual.

```

M O B O L I S T      L E V E L  7 . 0      TITLE:  PRINT/SELECT UPDATE RECORDS

0013.  IOD:      KSTATION = STATION;          KEY STATION
0014.  IOD:      INPUT = DISKETTE            DISKETTE FILE
0015.      DATASET = "MASTER01"             DATASET NAME
0016.      UNIT = 2;                        DRIVE 2
0017.
0018.  IOD:      OUTPUT = PRINTER            PRINTER OUTPUT
0019.      UNIT = 1;                        DEFAULT PRINTER UNIT
0020.
0021.  RCD:      WORKAREA                     BUFFER
0022.      TYPE (1)                          FIRST CHARACTER
0023.      DATA (127);                       ALL THE REST
0024.
0025.  KET :      DISPLAY                     SCREEN DISPLAY
0026.      CRTSIZE = 480                      BIG SCREEN
0027.      BLANK = CRT SIZE                   BLANKS ENTIRE SCREEN
0028.      RELEASE = AUTOMATIC                'ENTER' NOT REQUIRED
0029.      (3,2) 'PRINT/SELECT UPDATE RECORDS' PROMPT MESSAGE
0030.      (4,2) '1 - LIST ALL'               PROMPT MESSAGE
0031.      (5,2) '2- LIST UPDATES'           PROMPT MESSAGE
0032.      (6,2) '3 - SIGN OFF'              PROMPT MESSAGE
0033.      (7,2) 'SELECTION:'                PROMPT MESSAGE
0034.      (7,14) SELECTION (1,N);           FIELD ENTRY
0035.
0036.  START      BEGIN CODE SECTION
0037.  10,          KENTER (KSTATION, DISPLAY)  DISPLAY
0038.          IF (SELECTION, CONTAINS, :3:) STOP END PROGRAM
0039.          IFNOT ('12', CONTAINS, SELECTION) RESUMERR DISALLOW BAD CHARS
0040.
0041.          OPEN (INPUT, WORKAREA, ERR:200)  OPEN BUFFER FILE
0042.          OPEN (OUTPUT,WORKAREA, ERR:200)  OPEN PRINT FILE
0043.
0044.  110,         READ (INPUT, WORKAREA, EOF:200, ERR:200)  READ INPUT
0045.          GO (SELECTION) 130, 130, 120    BRANCH ON SELECTION
0046.
0047.  120,         IFNOT (TYPE, CONTAINS, :U:) GO:110      NOT UPDATE: RECYCLE
0048.
0049.  130,         PRINT (OUTPUT, WORKAREA, EOM:110, ERR:200)  PRINT, IGNORE EOM
0050.          GO:110                                           CYCLE
0051.
0052.  200,         CLOSE (INPUT, WORKAREA, ERR:201)          CLOSE, IGNORE ERRS
0053.  201,         CLOSE (OUTPUT, WORKAREA, ERR:10)          CLOSE PRINTER
0054.          GO:10                                           CYCLE
0055.
0056.  END          END OF SOURCE FILE
<END>

```

Figure 1-2: MOBOLIST Sample Listing

## SOURCE PROGRAM STRUCTURE

A MOBOL source program has two major sections:

- 1) A Data Definition Section;
- 2) An Execution Section.

Data definition statements occur in the Data Definition Section and define each data element to be processed by the program. All data definition statements must be in the Data Definition Section. Complete information on constructing data definition statements is presented in Section 5.

Execution statements occur in the Execution Section and define the processing steps to be applied to the data elements. All execution statements must be in the Execution Section. Complete information on constructing execution statements is presented in Section 6.

In addition to data definition and execution statements, there are comment statements and compiler directives which may appear in both sections of a MOBOL program. Comment statements are used to provide program documentation and annotate the source listing. Compiler directives are used to delimit the Execution Section (START and END) and control the program source listing (TITLE and EJECT). Comments and compiler directives are documented in Sections 3 and 4, respectively.





## SECTION 2: CONSTRUCTING A MOBOL SOURCE PROGRAM

After analyzing the application and creating a flowchart of the processing logic, a MOBOL program may be produced by following the sequential steps outlined below:

1. Determine required screen displays for the application and create a layout of these displays using the Series 21 CRT Screen Layout Form. (See discussion on designing screen displays presented below.)
2. Generate MOBOL source code. Include coding for screen displays. A MOBOL coding form is provided by MDS to assist the programmer when writing the MOBOL source.
3. Enter the source, via the STATION keyboard, into the Series 21 System.

### DESIGNING SCREEN DISPLAYS

It is recommended that Series 21 CRT Screen Layout Form be used when designing the format and content of the screen display. A sample form is presented in Figure 2-1.

<b>MDS.</b>	SERIES 21" CRT SCREEN LAYOUT FORM FOR 480 AND 1920 CHARACTER SCREEN FORMATS	APPLICATION <u>MOBOL</u>
		JOB NAME <u>SAMPLE PROGRAM</u>
		DATE _____ PAGE <u>1</u> OF <u>1</u>

	10	20	30	40	50	60	70	80
1								
2								
3								
4	⊕PRINT/SELECT UPDATE RECORDS							
5	⊕1 - LIST ALL							
6	⊕2 - LIST UPDATES							
7	⊕3 - SIGN OFF							
8	⊕SELECTION: ⊕							
9								
10								
11								
12								
13								
14								
15								
16								
17								
18								
19								
20								
21								
22								
23								
24								

FORM NO. M-3798-1277 \*Trademarks of Mohawk Data Sciences Corp., Parsippany, N.J. 07054 Mohawk Data Sciences-Canada, LTD. Registered User.

⊕ = Denotes attribute position.

**Figure 2-1: Screen Layout Form**

Data may be presented in either the 480 or the 1920 character display. Error messages (if any) presented during program execution are normally displayed on line 2 of the screen. Therefore, when an error message is presented, any data appearing on line 2 will be temporarily displaced.

Every guide message is preceded by an attribute byte which uses one character space on the CRT screen. This space must be reflected in the source code. For example, if a guide message appears on the screen on line 3, beginning in column 3, the layout would be coded (3,2).

## GENERATING MOBOL SOURCE

The compiler accepts source programs recorded on DISKETTE as one or more datasets. The diskette records (sectors) are input in sequence so that the source program is recorded in proper sequence.

Each diskette sector may contain one line of source. A line of source may be one of the following:

1. One comment statement.

For example:

```
*THE INPUT ROUTINE FOLLOWS  
  (Column 1)
```

2. One entire compiler directive.

For example:

```
EJECT
```

3. One entire clause of a multi-clause data definition statement.

For example:

```
MASK(12) = '$$, $$ $V.99CR'
```

4. One entire execution statement.

For example:

```
EDITFIELD = PICTURE (FIELD, MASK)
```

- 5a. The beginning of a data definition clause of an execution statement.

For example:

```
IF (TOTAL HOURS 40)
```

- 5b. A continuation of a data definition clause of execution statement.

```
⊕OVERTIME = OVERTIME + 1  
  (Column 1)
```

(where the beginning of the statement is illustrated in 5a.)

In this manual, the format of a clause or a statement is presented as a line of source as illustrated in Steps 1-4 above, (that is, a printed line of format corresponds to a complete line of source).

The rules are given in this section so that long clauses and statements can be recorded as illustrated by Steps 5a and 5b above.

A MOBOL program is written on MOBOL coding forms for ease and accuracy of transcription to DISKETTE. Accordingly, the requirements for writing a line of source are described below in terms of columns on a coding form.

MOBOL compiler directives, data definition and execution statements may be written in columns 1-59 of the coding form. However, most programmers follow a formatting convention (such as beginning any statement number of an execution in column 1 and beginning the remaining part of the statement in column 10) to align statements for clarity.

When a clause or statement requires more than 59 positions available on a single line, multiple lines are used to record the entire clause or statement. The first line is the beginning line and subsequent lines are continuation.

A continuation line is recognized by the appearance of a plus (+) in column 1 of the current line or by a delimiting comma (,) on the previous line.

When the "plus method" is used, the plus character is discarded and columns 2-59 are used as continuation for the previous line. With the exception of strings, basic syntactic elements must not be split between lines. The basic syntactic elements that must not be split are:

- Fill Designators (e.g., :\*)
- Comparators (e.g., <=)

When a string is split, it is important to consider the fact that column 2 of the continuation line immediately follows column 59 of the previous line and not the last non-blank column of the previous line.

When the "comma method" is used, columns 1-59 of the line are used as continuation for the line terminated with the comma.

The "comma method" may be employed only with a comma which is a part of the syntax; that is, a comma may not be inserted merely to signal continuation.

When both methods are employed simultaneously, the "plus method" prevails; that is, only columns 2-59 are used for continuation.

Comment statements may appear anywhere within the source program. There are two formats for comment statements. The first format requires an asterisk (\*) in column 1, of the coding form, with the remainder of the statement containing description information or all blanks. The second format requires all blanks in columns 1-59 of the coding form, with the remainder of the statement containing descriptive information or all blanks.

## **DATA DEFINITION SECTION REQUIREMENTS**

1. Data definition statements are composed of multiple clauses, the first of which begins with an identifying header and the last of which ends with a semicolon (;). The individual clauses occurring between the header and semicolon terminator must be coded according to the rules associated with the specific type of data definition statement header (IOD, RCD, KET or EQU).
2. Individual clauses which constitute a data definition statement are to be distinguished from continuation lines. Continuation is employed in a data definition statement only when an individual clause requires more than 59 columns available on a single line.

## EXECUTION SECTION REQUIREMENTS

1. The Execution Section begins with the compiler directive `START` which must be positioned within columns 1-59 of the coding form. (See Section 4 for a detailed discussion of compiler directives.)
2. The Execution Section is concluded with the compiler directive `END` which must be positioned within columns 1-59 of the coding form.
3. Individual execution statements must be coded according to the format appropriate to the statement as covered in Section 6.
4. Execution statements may have none, one or several statement numbers (see Figure 1-2, line 37). Statement numbers are used to branch from one execution statement to another. The compiler associates each statement number with the first execution statement which follows the statement numbers.
  - a. Each statement number must be followed by a delimiting comma (,).
  - b. Statement numbers must be unique when regarded as a decimal number (for example, 32 is equivalent to 0032 and, therefore, would not be considered unique).
  - c. Generally, 3 or 4 digits are used for statement numbers. However, up to 32 digits may be used, if desired.

Figure 2-2 is an example of a MOBOL coding form completed during development of the sample program shown in Figure 1-2.

MOBOL CODING FORM		PAGE 1 OF 4
PROGRAM <b>MOBOL SAMPLE PROGRAM</b>	PROJECT	DATE
PROGRAMMER <b>P. LANGSTRAAT</b>		
<div style="display: flex; justify-content: space-between; font-size: small;"> <span>10</span><span>20</span><span>30</span><span>40</span><span>50</span><span>59</span> </div> <b>TITLE: PRINT/SELECT UPDATE RECORDS</b> <b>KEY: DISPLAY</b> <b>CRTSIZE = 480</b> <b>BLANK = CRTSIZE</b> <b>RELEASE = AUTOMATIC</b> <b>(3,2) 'PRINT/SELECT UPDATE RECORDS'</b> <b>(4,2) '1 - LIST ALL'</b> <b>(5,2) '2 - LIST UPDATES'</b> <b>(6,2) '3 - SIGN OFF'</b> <b>(7,2) 'SELECTION (1,N);'</b>		
<div style="display: flex; justify-content: space-between; font-size: small;"> <span>60</span><span>70</span><span>80</span><span>90</span><span>100</span><span>110</span><span>120</span><span>128</span> </div>		

FORM NO. M-3797-1277 \*Trademarks of Mohawk Data Sciences Corp., Parsippany, N.J. 07054 Mohawk Data Sciences - Canada, LTD. Registered User.

Figure 2-2: MOBOL Coding Form

## **SECTION 3: COMMENT STATEMENTS AND COMMENT FIELDS**

### **COMMENT STATEMENTS**

The following rules apply:

1. An asterisk (\*) is written in column 1 with text (or all blanks) written in columns 2-128.
2. All blanks in columns 1-59 with text (or all blanks) in columns 60-128.

Comment statements may be inserted wherever desired. However, they may not appear within a data definition statement.

### **COMMENT FIELDS**

Comment fields can be placed in columns 60-128. These will be printed on the same line as a source MOBOL statement without actually being within the statement. Comment fields may appear in columns 60-128, adjacent to any line.



## SECTION 4: COMPILER DIRECTIVES

There are four MOBOL compiler directives which provide the compiler with control information during compilation. These directives are presented in logical order. Compiler directives do not produce object code or affect other MOBOL source statements.

### TITLE

Purpose: To cause a top-of-form and print the title on a source listing when using MOBOLIST.

Format: **TITLE:** Up to 53 characters.

Description: TITLE causes an immediate top-of-form to occur in a printed listing of source statements. It establishes up to 53 characters to be printed on the top line of the first and any subsequent pages until a new TITLE directive is encountered. Several TITLE directives may be used in a source listing for documentation clarification. TITLE can appear between any two statements.

Example: **TITLE: PRINT/SELECT UPDATE RECORDS**

### EJECT

Purpose: To eject a page when printing a source statement listing when using MOBOLIST.

Format: **EJECT**

Description: EJECT causes an immediate top-of-form (eject a page) to occur in a printed listing of source statements. The most recently established title is printed at the top of the new page. EJECT can appear between any two statements.

Example: **EJECT**



## START

Purpose: To mark the beginning of the Execution Section.

Format: **START**

Description: START begins the Execution Section.

Example:

- }  
• } DATA DEFINITION SECTION  
• }

START

- }  
• } EXECUTION SECTION  
• }

END

- NOTE:
1. All data definition statements must precede the START statement.
  2. All execution statements must follow the START statement.

## END

Purpose: To mark the END of the Execution Section.

Format: **END**

Description: END signifies the end of the execution statements and concludes compilation.

Example: See example for START

NOTE: Every program must contain an END statement and END must be the last statement in the program.

## SECTION 5: DATA DEFINITION STATEMENTS

Data definition statements establish data elements and the names by which these data elements can be referenced. Names (IOD-names, RCD-names, INDEX-names, KET-names, and EQU-names) must be chosen according to the following rules:

1. The first character must be alphabetic.
2. Each subsequent character (if any) must be either alphabetic or numeric.
3. Space characters may be interspersed to improve readability.
4. The number of significant characters must not exceed 32.
5. The name must not be a reserved name (listed in Appendix B).
6. The name must not be defined earlier within the Data Definition Section.

The following are examples of valid names:

T  
ACCOUNT  
POLICY2  
BIN NUMBER

The following are examples of rule violations:

@BAT	(Violates rule 1)
TEN%	(Violates rule 2)
THISNAMEISMUCHTOOLONGTOBEPERMITTED	(Violates rule 4)
READ	(Violates rule 5)
TWICE DEFINED	(Valid name)
TWICEDEFINED	(Violates rule 6)

The last two examples illustrate that spaces relate only to readability; that is, the missing space in the last example does not make the name different from the preceding example.

There are four types of data definition statements in MOBOL:

1. The IOD statement designates access information for Input/Output operations. The IOD statement also designates an IOD-name that may be used to reference all of the associated access information\* or may be used to formulate a reference to a particular access parameter.
2. The RCD statement designates a record area as a collection of individual fields. The RCD statement also designates an RCD-name and field names that may be used to reference the entire record area and its individual fields. The RCD statement may also designate an INDEX-name.
3. The KET statement designates a record area and control information specific to entry, verification and display of the record data. The KET statement also designates a KET-name and field names that may be used to reference the entire record area and its individual fields. The record area and the control information are collectively referenced when the KET-name is used with the entry/verify operations. (The control information cannot otherwise be referenced by the program.)
4. The EQU statement designates one or more EQU-names that may be used for parameter passing within the Execution Section. An EQU-name may be made equivalent to a data item so that the EQU-name may be referenced instead of the original item. An EQU-name may be made equivalent to a sub-field of another data item to allow access to varying length fields or records.

A data definition statement consists of multiple source statements, each of which contributes a certain amount of related parametric information. The format of each source statement and the number of source statements required for data definition depends upon the number of parameters to be established within the data definition statement.

All data definition statements must be terminated by a semicolon (;) placed at the end of the last source statement. The placement of the semicolon is important; it ensures that subsequent individual source statements are not mistaken to be additional parametric information.

---

\*this block of information is a set of system provided parameters or specifications for access to the data contained in the file named by the IOD.

## THE INPUT/OUTPUT DESCRIPTOR STATEMENT

### IOD

Purpose: To define a File Control Block (FCB) of access information (to be used with Input/Output operations). The IOD is a data definition statement which is required for each dataset accessed by execution statements.

Format:           **IOD: IOD-name = device**  
                          **iod-specification-1**  
                          **iod-specification-2**  
                          •  
                          •  
                          •  
                          **iod-specification-n ;**

Description:       The components of this statement are:

IOD-name   The name used to reference the FCB or a specific parameter within the block. The IOD-name is used with Input/Output operations to identify the file to be operated upon.

device      The type of device used for data transfer.

Permitted selections are:

DISK  
DISKETTE  
COMP CHAN  
PRINTER  
STATION  
TAPE

**Note:** The selection TAPE is also used for DATA RECORDER .

iod-specification   A keyword expression which assigns a particular parameter. The keywords that are used depend upon the device specification; for example, the specification:

                  DATASET = 'ACCOUNTS '

is valid for DISK and DISKETTE devices only.

The keywords that may be used in the IOD statement are:

ACCESS	KEYBOARD	SIGNIF	UNIT
BOUND	KEYVALUE	SLEW	VOLUME
DATASET	MCSEQ	STATE	
FILTER	OPERATIONAL	TARGET	

In Section 6 the keywords are listed and defined. When a keyword is not explicitly assigned an initial-value, a default value is used. Most keywords have default values which correspond to normal use. Therefore, it is necessary to include iod-specifications only for those parameters whose defaults are unsuitable for the particular application program. A keyword that is available for reference in the Execution Section is referenced in the Execution Section by concatenating the IOD-name and the keyword.

For example:

IOD-name.DATASET

In the example above, the IOD-name identifies the block of information and DATASET identifies the specific parameter; in this case, the name of the dataset.

## RECORD STATEMENT

The RCD statement format can vary depending upon specific use. Therefore, for clarity, two main formats are presented:

The first format is the basic form of RCD. It is used to define a record area and its fields.

The second format is used to define an array of records (multiple records of the same format).

Additional formats of the Record statement are presented in this section under the heading "RECORD REMAPPING".

### **RCD**

Purpose: To define a record area and its fields.

Format: **RCD: RCD-name**  
**field-specification-1**  
**field-specification-2**  
**•**  
**•**  
**•**  
**field-specification-n ;**

Description: The components of this statement are:

RCD-name The name used to refer to the entire record area established by the various field-specifications that follow.

field-specification A clause which defines one field of a record area, its offset and length.

In the event that the individual fields are not related to each other, as when defining counters and intermediate result fields, RCD-name may be coded as a minus (-). When the RCD-name is coded as minus, reference to the overall record area (as a single data item) is not possible.

A field-specification consists of two main parts: a name and a value. The name is used to reference the individual field and the value establishes the initial-value of the field. The initial-value is the contents of the field after program load and prior to program execution.

Example:

```
RCD: -  
      DOLLAR LIMIT = 10000  
      MESSAGE = 'LIMIT EXCEEDED';
```

The field DOLLAR LIMIT is the first field of the record and consists of five (5) positions. The field MESSAGE is the second field of the record and consists of fourteen (14) positions. In both cases, the size of the field is determined by the number of positions appearing within the value. The value assigned to DOLLAR LIMIT is called a decimal number. The value assigned to MESSAGE is called an alphanumeric string.

## Decimal Numbers

A decimal number is a series of digits with an optional leading sign (+ or -). The value is taken as positive unless the minus (-) is explicitly coded. Space characters are not significant and may be interspersed to improve readability.

Example:

```
RCD: -  
    LARGE = 50 000  
    SMALL = 64  
    MIDDLE = 24 966;
```

The fields LARGE and MIDDLE are each five (5) positions and positive; field SMALL is two (2) positions and negative. Internally, the sign of a decimal number is carried in the unit's position along with the unit's digit. Having a sign does not increase the length of the resulting field.

## Alphanumeric Strings

An alphanumeric string is a sequence of characters enclosed within quote (') characters. The enclosing quotes do not contribute to the length of the field. Space characters are significant and must be coded only when a space data character is desired. If a quote character is to be part of the string value, two contiguous quotes must be coded for each quote data character desired.

Example:

```
RCD: -  
    UPPER    = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'  
    LOWER    = 'abcdefghijklmnopqrstuvwxyz'  
    NUMBERS  = '123456789'  
    RESPONSE = 'YOU''RE OK.';
```

The fields UPPER and LOWER are each 26 positions containing the alphabet in uppercase and lowercase, respectively. The field NUMBERS is ten (10) positions and contains the digit characters. The field RESPONSE is also ten (10) positions, even though eleven (11) characters are coded between the surrounding quotes. If RESPONSE is displayed on the CRT screen, it would appear as:

YOU'RE OK.

## Referencing Overall Record Areas

An RCD-name must be designated when the overall record area is to be referenced. A field-name may be coded as minus (-) if there is no requirement to reference the field individually.

Example:

```
RCD:  HEADER
      - = 'PAGE'
      CURRENT PAGE = 00
      - = 'OF'
      TOTAL PAGES = 00;
```

The field TOTAL PAGES is to contain the number of pages within the report and the field CURRENT PAGE is to be incremented for each new page. The record HEADER would be printed once per page; suitable coding in the Execution Section could be written to produce the following sequence of print images for a twelve-page report:

```
PAGE 01 OF 12
PAGE 02 OF 12
  •
  •
  •
PAGE 12 OF 12
```

Note that two zero digits are necessary for the initial-value of CURRENT PAGE and TOTAL PAGES so that a sufficient number of positions are allocated to handle reports of up to ninety-nine (99) pages in length.

## Stating An Explicit Size For A Field

An explicit size may be designated for a field. If the field is to be given a decimal number as an initial-value, the leading zeros do not have to be coded. The field length must not exceed 256 positions, if the field is named.

Example:

```
RCD:  HEADER
      - = 'PAGE'
      CURRENT PAGE(2) = 0
      - = 'OF'
      TOTAL PAGES(2) = 0;
```

Because CURRENT PAGE has an explicit size of two (2) positions, the value is established by moving the initial-value into the field, from right-to-left, with leading zeros to fill the field. The same considerations apply to TOTAL PAGES.



## Fill Characters And Alignment

The fill character and the direction of filling may be designated, provided that the field is explicitly given a size. The fill character itself is coded between two colon characters and the resulting three characters are placed to the left of the initial-value for left-filling or the right for right-filling. Coding a fill character and omitting an initial-value designates that every position of the field is to be filled.

Example:

```
RCD:  -
      LEFTFILL(10) = **: 'PAGE'
      RIGHTFILL(10) = 'PAGE' **:
      ALLFILL(10) = **: ;
```

The effect of these definitions is listed:

```
*****PAGE
PAGE*****
*****
```

Large displacements for printing at the right margin may be indicated through the use of filling.

Example:

```
RCD:  HEADER
      -(119) = : :
      - = 'PAGE'
      CURRENT PAGE = 00
      - = 'OF'
      TOTAL PAGES = 00;
```

or:

```
RCD:  HEADER
      -(124) = : : 'PAGE'
      CURRENT PAGE = 00
      - = 'OF'
      TOTAL PAGES = 00;
```

Either format would cause the page accounting information to appear to the far right of a full 132-position print line. Because of the choice of names, only one of the two examples could appear in a given program.

## Offset Notation

In all of the examples presented thus far, each field is placed within the record according to its order of definition; fields defined in this way are called sequential. Optionally, a field may be placed out of sequence using the offset notation; fields defined in this way are called non-sequential.

Examples:

```
RCD:  HEADER:
      -(119,4) = 'PAGE'
      -(127,2) = 'OF'
      CURRENT PAGE(124,2) = 0
      TOTAL PAGES(130,2) = 0;
```

For the offset notation, both the offset into the record and the size of the field are specified. The offset is coded first and represents the number of record positions to the left of the field being defined.

Note that only ten (10) positions of the record are assigned an initial-value. No additional values are required in this case, since all records are created with space characters for those positions not explicitly assigned an initial-value.

## Fields Without Initial-Values

When an initial-value is assigned to a field, the corresponding positions of the record are updated. When no initial-value is assigned, there is no effect on the corresponding positions of the record during compilation.

Example:

```
RCD:  -
      F1(0,5) = 'AAAAA'
      F2(1,3) = 'BBB'
      F3(2,1) = 'C'
      F4(0,3) ;
```

As the compiler processes each specification sequentially, the appearance of the record is shown below at the various stages of compilation:

```
AAAAA      (after F1)
ABBBA      (after F2)
ABCBA      (after F3)
ABCBA      (after F4)
```

Note that the first three positions of the record were not affected by the definition of field F4.

### Alignment Without Filling

An initial-value may designate that alignment without filling is required. In this case, the field positions that would normally receive a fill character are unchanged from their current value.

Example:

```
RCD:  -  
      F1(0,6) = 'AAAAAA'  
      F2(1,4) = '::'B'  
      F3(2,2) = 'C'::;
```

The two contiguous colons(::) signify that filling is to be suppressed. The appearance of the record is shown below at various stages of compilation:

```
AAAAAA      (after F1)  
AAAABA      (after F2 - only the fourth character  
             is changed since the initial-value is  
             only one character in length and  
             right-justified. The fill characters are  
             suppressed).
```

```
AACABA      (after F3 - only the third character is  
             changed since the initial-value is  
             only one character in length and left-  
             justified. The fill characters are  
             suppressed).
```

NOTE: The above example is for illustrative purposes only and would not normally be found in an application program.

## Sequential and Non-Sequential Fields

Both sequential and non-sequential fields may be employed in defining a record. A sequential field always begins at the first record position after the previous sequential field. A non-sequential field begins at the designated offset and has no influence on the placement of the next sequential field.

Example:

```
RCD:  -
      LAST NAME(15) = 'SMITH' : :
      FIRST NAME(15) = 'SAM' : :
      LAST INITIAL(0,1) ;
```

or:

```
RCD:  -
      LAST INITIAL(0,1)
      LAST NAME(15) = 'SMITH' : :
      FIRST INITIAL(15,1)
      FIRST NAME(15) = 'SAM' : : ;
```

Both examples produce equivalent records and fields. Note that in the second example, the placement of LAST NAME is not influenced by LAST INITIAL. Again, only one of the two examples could appear in a given program.

A non-sequential field may overlap other fields within the record.

Example:

```
RCD:  -
      LAST NAME(15) = 'SMITH' : :
      FIRST NAME(15) = 'SAM' : :
      IDENTIFIER(0,16) ;
```

The field IDENTIFIER includes all of the field LAST NAME and the first position of FIRST NAME.

For some Input/Output related applications, certain fields of a data record are not significant to the application and therefore do not require individual identification. In such cases, an unnamed field may be required to create a sufficiently large record area.

Example:

```
RCD:  MASTER
      TAX TO DATE(60,7)
      -(0,128) ;
```

This record is suitable for a selective copy of employee MASTER records based on certain TAX TO DATE criteria. The application is not specifically concerned with address information, salary, etc., but is obligated to provide room for this data so that it will be preserved on the output file. The unnamed field ensures that 128 positions are allocated for MASTER records.

Tables 5-1 and 5-2 present every variation of establishing a field and assigning its initial-value. In Table 5-1, the field name does not have an explicit size. In Table 5-2, the field length is specified.

**TABLE 5-1: ESTABLISHING FIELD SPECIFICATIONS WHERE FIELD NAME HAS NO SIZE SPECIFICATIONS**

Field-name

	<b>EXAMPLE</b>	<b>INTERNAL REPRESENTATION</b>
NO VALUE ASSIGNED	A	Contents not changed Contents=(-)
VALUE ONLY	A = 321	Alignment right-to-left; No fill characters. Contents=(321)
VALUE/NO FILL CHARACTERS	A = ::'ABC'	Alignment right-to-left; No fill characters. Diagnostic message. Contents=(ABC)
VALUE/FILL CHARACTERS	A = :X:'ABC'	Alignment right-to-left; No fill characters. Diagnostic message. Contents=(ABC)
FILL CHARACTERS ONLY	A = :X:	Fill with specified fill character. Diagnostic message. Contents (X)
EMPTY FILL	A = ::	Contents not changed. Diagnostic message. Contents=(-) .

Where (-) = One byte of previously established data (normally a space character).

**TABLE 5-2: ESTABLISHING FIELD SPECIFICATIONS  
WHERE FIELD NAME HAS A SIZE SPECIFICATION**

Field-name (n)  
Where n = field length

	<b>EXAMPLE</b>	<b>INTERNAL REPRESENTATION</b>
NO VALUE ASSIGNED	A(3)	Contents not changed. Contents=(---)
VALUE ONLY value length < field length	A(3) = 2	Alignment right-to-left; fill after value length with decimal zero. Contents=(002)
VALUE ONLY value length = field length	A(3) = 321	Alignment right-to-left; no fill characters required. Contents=(321)
VALUE ONLY value length > field length	A(3) = 4321	Alignment right-to-left; truncate once field is full. Diagnostic message. Contents=(321)
VALUE/NO FILL CHARACTERS value length < field length	A(3) = :: 'A'	Alignment right-to-left; remaining characters of field remain unchanged. Contents=(--A)
VALUE/NO FILL CHARACTERS value length = field length	A(3) = :: 'ABC'	Alignment right-to-left; no fill characters required. Contents=(ABC)
VALUE/NO FILL CHARACTERS value length > field length	A(3) = :: 'ABCD'	Alignment right-to-left; truncate once field is full. Diagnostic message. Contents=(BCD)
VALUE/FILL CHARS value length < field length	A(3) = :X: 'A'	Alignment right-to-left; fill with specified fill characters. Contents=(XXA)
VALUE/FILL CHARS value length = field length	A(3) = :X: 'ABC'	Alignment right-to-left; no fill characters required. Contents=(ABC)
VALUE/FILL CHARS value length > field length	A(3) = :X: 'ABCD'	Alignment right-to-left; truncate once field is full. Diagnostic message. Contents=(BCD)
FILL/CHARACTERS ONLY	A(3) = :X:	Fill with specified fill character Contents=(XXX)
EMPTY FILL	A(3) = ::	Contents not changed. Diagnostic message. Contents=(---)

Where (---) = three bytes of previously established data.

## Non-Graphic Fill Characters

If a non-graphic character (one that does not appear on the keyboard) is required as a fill character, it must be expressed in terms of its hexadecimal representation. In this case, the two hexadecimal digits of the fill character are coded between the colons.

Example:

```
RCD: -  
      ALL ONES (2) = :FF:  
      ALL NULL (8) = :00: ;
```

The field ALL ONES consists of two positions, or 16 bits; each bit is set. The field ALL NULL consists of eight positions, or 64 bits; each bit is clear. The left hexadecimal digit determines the leftmost four bits of the fill character and the right digit determines the rightmost four bits. The hexadecimal digits and their corresponding four-bit values are tabled as follows:

Hexadecimal Digit	4-Bit Value
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

## Non-Graphic Hexadecimal Strings

If a non-graphic string is required as an initial-value, it must be expressed as a hexadecimal string. In this case, two hexadecimal digits must be coded for each string character. All hexadecimal digits specified must be enclosed by quotes and preceded by an X.

Example:

```
RCD: -  
      BINARY HUNDRED = X'64'  
      BINARY THOUSAND = X'03E8'  
      ALPHA THREE = X'C1 C2 C3' ;
```

The field BINARY HUNDRED is one position in length and contains the binary representation of the number 100. The field BINARY THOUSAND is two positions and contains the binary 1000. The field ALPHA THREE is three positions and contains the EBCDIC characters ABC (more typically this field would be defined as 'ABC'). Space characters are not significant and may be used to improve readability.

## Composite String

A string requiring a mix of graphic and non-graphic characters may be specified using a composite specification. A composite specification is a sequence of alphanumeric strings, hexadecimal strings and unsigned decimal numbers coded as applicable.

Example:

```
RCD:  FILE COMMANDS
      - = 'UPDATE' X'01'
      - = 'DELETE' X'02'
      - = 'ADD    ' X'04'
      - = 'INSERT' X'08'
      - = X'00' ;
```

The record FILE COMMANDS contains a sequence of individual (unnamed) fields. It may be searched to transform a file command into a bit mask for program interpretation. The null terminates the record ('COMMAND NOT FOUND' condition). In this example, each source line represents one transformation entry. This type of encoding facilitates the addition and maintenance of commands and provides clarity of the source program.



## **RCD Array**

Purpose: To define an array of records so that multiple records of the same overall format are available for reference.

Format: **RCD: RCD-name (INDEX-name, iterations)**  
**field-specification-1**  
**field-specification-2**  
•  
•  
•  
**field-specification-n ;**

Description: The components of this statement are:

RCD-name	The name used to reference one record of the array. The field-specifications which follow establish the format of the records.
INDEX-name	The name of the index variable used to select a particular record of the array. When a reference to the array is made and a specific record number is not coded, the record implied by the current value of INDEX-name is used.
iterations	The number of record occurrences within the array. A maximum of 100 may be designated.
field-specification	A clause that defines one field of a record in the array, its offset and length. Also, the initial-value of the field is defined for the first record.

The various field-specifications establish field-names and field initial-values for the first element of the record array. The rules governing specification are identical to the ones described in the first format of RCD with the following exceptions:

- An individual record of the array must not exceed 256 positions. The entire array, however, may exceed 256 positions.
- An individual record of the array must not be defined as having more than 255 named fields.

Additional record areas are allocated so that the total number of records in the array corresponds to the iterations parameter.

The index variable is defined as a two-position field with an initial-value of decimal zero.

During execution, the assignment of a value to INDEX-name selects an element of the array for implied references. The value zero selects the first element, one selects the second, etc. Accordingly, each value assigned to an index variable must be:

- decimal;
- greater than or equal to zero; and
- less than the iterations parameter designated for the record.

Example:

```
RCD: RECORD (INDEX, 3)
      FIELD A(4) = 'ABCD'
      FIELD B(2) = 41;
```

The arrangement of record elements and their fields is shown below with explicit indexing notation used for identification.

RECORD (0) {	A	B	C	D	FIELD A(0)
	4	1			FIELD B(0)
RECORD (1) {	Ø	Ø	Ø	Ø	FIELD A(1)
	Ø	Ø			FIELD B(1)
RECORD (2) {	Ø	Ø	Ø	Ø	FIELD A(2)
	Ø	Ø			FIELD B(2)

Note that only the first iteration is assigned an initial-value.

## THE KEY ENTRY TABLE STATEMENT

### **KET**

Purpose: To define a record area and control information specific to entry, verification and display of the record data.

Format:           **KET: KET-name**  
                          **context-specification-1**  
                          **context-specification-2**  
                          •  
                          •  
                          •  
                          **context-specification-n**  
                          **display-specification-1**  
                          **display-specification-2**  
                          •  
                          •  
                          •  
                          **display-specification-m ;**

A variation of the above format for KET is presented later in this section under the heading "RECORD REMAPPING".

Description:       The components of this statement are:

KET-name	The name used to refer to the entire display record established by the various field-specifications which follow. The name refers to both the display record and its control information when used with entry/verify operations.
context-specification	A clause that establishes a parameter associated with the overall KET.
display-specification	A clause that establishes either a field within the record area or a fixed guide message.

The ordering of specifications within the KET is important:

- context-specifications must follow the KET-header; and
- display-specifications must follow the last context-specification.

### **Context-Specification**

Context-specifications are used to adjust operational details of entry, verify and display activities associated with the KET. Each specification is coded as an individual keyword assignment clause. In these clauses, the keyword identifies the parameter and the assignment value identifies the option selected.

A keyword specification may be omitted from the KET statement if its default value is appropriate. The ordering of the assignments is not significant, unless conflicting assignments are made to the same keyword. In this case, the last assignment prevails.

The parameters and their options are discussed in the following paragraphs.

## CRTSIZE

The display size to be used when the KET is active may be 480 or 1920 characters.

$$\text{CRTSIZE} = \left\{ \begin{array}{l} 480 \\ 1920 \end{array} \right\}$$

When 480 is selected, the display is configured to twelve (12) lines of forty (40) characters each. When 1920 is selected, there are twenty-four (24) lines of eighty (80) characters each. Regardless of the selection, the display position in the upper left corner is designated line one, column one (1,1).

An application may have a mix of '480' KET's and '1920' KET's as required.

The default value is 480.

## BLANK

An area of the display to be cleared upon activation of the KET may be designated.

$$\text{BLANK} = \left\{ \begin{array}{l} \text{CRTSIZE} \\ (11, \text{cc}) / (11, \text{cc}) \end{array} \right\}$$

When CRTSIZE is selected, the entire display area is designated. When the second option is selected, the starting line column and the ending line column are explicitly coded.

Example:

$$\text{BLANK} = (6, 1) / (12, 40)$$

For a '480' KET, the bottom half of the display is cleared.

The default value is computed by the compiler as the minimum extent required to clear the guide and field areas used by the KET. The keyword BLANK should not be omitted when a previous KET occupies more space on the screen, since data from this previous KET will remain in the subsequent display.

## MODE

Data fields of the KET are presented for entry/verify in their order of definition within the KET. When a KET is used for the creation of new data, the standard order of presentation cannot be superseded by the operator through the use of the cursor control keys. The MODE parameter designates the intended use of the KET.

Data fields of the KET are presented for entry/verify in their order of definition within the KET. When a KET is used for the creation of new data, the standard order of presentation cannot be superseded by the operator. When a KET is used for updating existing data, the standard order may be superseded by the operator through the use of the cursor control keys. The MODE parameter designates the intended use of the KET.

$$\text{MODE: } \left\{ \begin{array}{l} \text{FIELD} \\ \text{SCREEN} \end{array} \right\}$$

The FIELD option designates data creation and enables the use of the field positioning keys (  $\leftarrow$  and  $\rightarrow$  ) to move the cursor backwards or forwards (in order of definition) one field at a time. The SCREEN option designates data update and enables the use of the field positioning keys as well as the cursor positioning keys (  $\uparrow$   $\downarrow$   $\leftarrow$  and  $\rightarrow$  ). Each cursor positioning key moves the cursor one position on the screen, without regard to field ordering; a sequence of these keys may be used to place the cursor on any field requiring update.

Other effects of selecting FIELD or SCREEN mode are described later, in Section 8.

The default value is FIELD.

## NUMPAD

The numeric keypad (when installed) may be used for entry of numeric data or for selection of application-defined functions.

$$\text{NUMPAD} = \left\{ \begin{array}{l} \text{DATA} \\ \text{ESCAPE} \end{array} \right\}$$

The option determines whether input from the keypad is to be processed as numeric data by the entry/verify operations (DATA option) or is to be passed back for application-defined interpretation (ESCAPE option). Operational details associated with the ESCAPE option are covered in Section 8.

The default value is DATA.

## RELEASE

When the end of an entry/verify operation is reached, the display data may be held for final operator review or released immediately for processing.

$$\text{RELEASE} = \left\{ \begin{array}{l} \text{MANUAL} \\ \text{AUTOMATIC} \end{array} \right\}$$

The AUTOMATIC option designates that the display data is to be released immediately for processing. The MANUAL option designates that the display data is to be held until the operator depresses the ENTER key. Errors detected during the review may be corrected prior to the depression of ENTER.

The default is MANUAL.

## INIT

When a KET is presented for the entry or the verify process, the first field to receive the cursor is determined by specific rules discussed in Section 8.

$$\text{INIT} = \left\{ \begin{array}{l} \text{AUTOMATIC} \\ \text{ESCAPE} \end{array} \right\}$$

The AUTOMATIC option designates that the first field to be keyed is selected by the system. The ESCAPE option designates that a branch is to be taken to the application program before initial field selection. Operational details associated with the ESCAPE option are covered in Section 8.

The default is AUTOMATIC

## TYPE

Program-assigned value that is transcribed to the STATION IOD when the KET is activated. This one-byte parameter may be used to represent anything desired by the programmer. The format is:

$$\text{TYPE} = \text{initial-value}$$

If a decimal initial-value is specified, the range is 0-255. It is subsequently converted to binary representation.

## NOTE

Program-assigned value that is transcribed to the STATION IOD when the KET is activated. This one-byte parameter may be used to represent anything desired by the programmer. The format is:

$$\text{NOTE} = \text{initial-value}$$

If a decimal initial-value is specified, the range is 0-255. It is subsequently converted to binary representation.

## Display-Specification

Each display-specification designates either a fixed guide message (guide-specification) or a data field (field-specification). The field-specifications should be ordered in the desired entry/verify sequence. The guide-specifications may occur in any order and may be interspersed with field-specifications.

### Guide-Specification

A guide-specification designates one guide message which may be used to identify the overall display (title usage) or to identify a specific field (prompt usage).

The format of a guide-specification is:

**(line, column, attribute) guide-text**

The components are:

#### Line

The line number on which the message attribute is to be placed. The range is 1 to 12 for a '480' KET and 1 to 24 for a '1920' KET.

#### Column

The column number in which the message attribute is to be placed. The range is 1 to 40 for a '480' KET and 1 to 80 for a '1920' KET.

#### Attribute

The parameter which designates a particular attribute byte that is to precede the guide message. The attribute byte occupies one display position, appears as a blank and influences the display of the characters which follow.

Each attribute parameter and its effect on the display is listed below:

S or SUPPRESS for suppressed;  
L or LOW for low intensity;  
LB or LOWBLINK for blinking, low intensity;  
R or REVERSE for reverse image;  
RB or REVERSEBLINK for blinking, reverse image;  
H or HIGH for high intensity;  
HB or HIGHBLINK for blinking, high intensity.

The default is LOW if the attribute and its preceding comma (,) are omitted in a guide-specification.

### Guide-text

The actual guide message text, is coded as an alphanumeric string (see example). The string is placed on the screen immediately following the attribute position and continuing to the right.

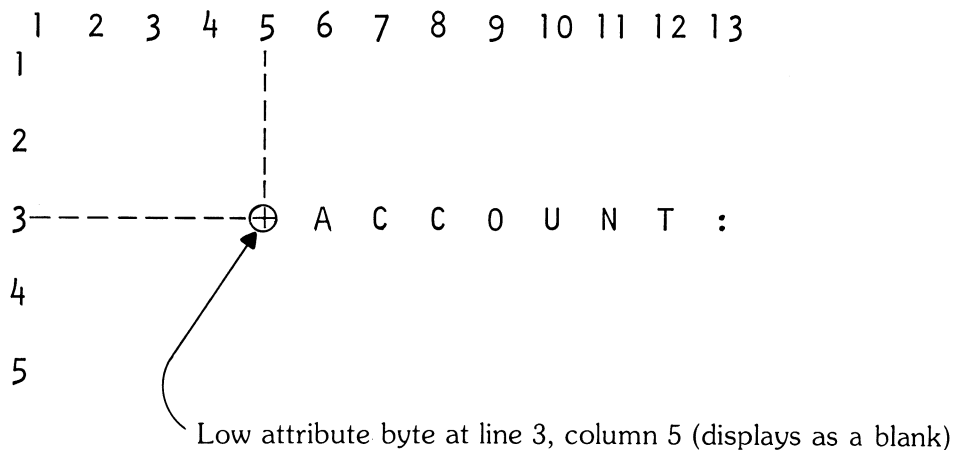
The total number of display positions used is equal to the length of the string plus one (for the leading position occupied by the attribute byte).

In the event of display line overflow, continuation begins on the first position of the next line.

Example:

```
KET:
      .
      .
      .
      CRTSIZE = 480
      .
      .
      .
      (3,5,LOW) 'ACCOUNT:'
      .
      .
      .
      ;
```

These fragments of a KET statement corresponds to the display results diagrammed below.





## Field-specification

A field-specification establishes one field for data entry/display. Field-specifications may be interspersed with guide-specifications. The ordering of field-specifications governs the standard entry sequence.

The format of a field-specification is:

**(line, column, attribute) field-name (position, shift, source, exit, verify) sn-post, sn-pre**

### Line

The line number on which the field attribute is to be placed. The range is 1 to 12 for a '480' KET and 1 to 24 for a '1920' KET.

### Column

The column number in which the field attribute is to be placed. The range is 1 to 40 for a '480' KET and 1 to 80 for a '1920' KET.

### Attribute

The parameter which controls the display of the field. The attribute appears as a blank position at the designated line and column with the field immediately following the attribute. Parameters are the same as those listed for guide-specifications.

The default is HIGH.

### Field-name

The name identifying the field within the record. Optionally, a minus (-) character may be coded in lieu of field-name when no direct program references to the field are required.

### Position

Position designates the location and length of the field within the record. Also, it designates certain entry control information. The form for this parameter is:

**field-offset, field-length/minimum/0**

The decimal number field-offset establishes the displacement of the field from the beginning of the record. The field-offset and its delimiting comma (,) may be omitted to designate that the field is to follow the last specified sequential field within the record.

The decimal number field-length establishes the number of contiguous characters comprising the field. Specification of this parameter is mandatory.

The decimal number minimum designates the minimum number of significant data characters that must exist within the field before a field entry operation can be considered complete. The terminating phrase '/0' is optional. It designates that a field entry operation can be considered complete when the minimum number of significant characters are present or when no significant characters are present. If the entire phrase '/minimum/0' is omitted there is no constraint on the number of significant characters within the field.

Table 5-3 illustrates the interpretation of the significance parameters for a five-position field.

**Table 5-3: Interpretation of Significance Parameters**

Source Specification	Number of Significant Characters in Field
...,5/4/0,...	0,4 or 5
...,5/4,...	4 or 5
...,5,...	0,1,2,3,4 or 5
...,5/0,...	0,1,2,3,4 or 5
...,5/0/0,...	0,1,2,3,4 or 5

**Shift**

Shift facilitates the entry of a specific class of characters (by minimizing the need to depress the shift keys). Also, it restricts the acceptable character class. The permitted values are:

- LETTER or L            to facilitate the entry of upper case A-Z and to impose no actual restriction.
- DIGIT or D            to facilitate the entry of the digits 0-9 and to impose no actual restriction.
- TEXT or T             to facilitate the entry of lower case A-Z and to impose no actual restriction.
- NUMERIC or N        to facilitate the entry of the digits 0-9 and to restrict data to the digits 0-9.
- ALPHA or A            to facilitate the entry of upper case A-Z and to restrict data to upper case A-Z.

The significance of L, D or T fields is determined by field-length less the number of leading and training spaces. The significance of Numeric fields is determined by field-length less the number of leading zeros. The significance of Alpha fields is determined by field-length alone. The default is LETTER for this parameter.

## Source

This parameter designates the conditions of allowed operator entry. The values are:

KEY or K	to designate that the field may be keyed whenever the cursor arrives at the field displayed.
GENERATED or G	to designate that the field has a program-generated initial-value which the operator seldom keys. The field may be keyed, however, if the cursor arrives at the field by means of a backspacing action, cursor positioning action or explicit program direction.
PROTECTED or P	to designate that the field is protected from operator entry unless the cursor arrives at the field under explicit program direction.

The default value for this parameter is KEY.

## Exit

The exit parameter is specified if a particular field release key (or key class) must be used to signal the end of keying to initiate alignment. If no parameter is coded, the field is automatically released when the rightmost position is correctly keyed. The field may be released prior to keying the rightmost position as follows:

Release Key	Shift	Action
SKIP	L, D, T	Left align; space fill
EXIT	L, T	Right align; space fill
EXIT	D, N	Right align; zero fill
- (minus)	N	Right align; zero fill; negate field

The following parameters specify that a particular release key must be used (even after keying the rightmost position).

SKIP or S to designate that the SKIP key must be used to release the field. If depressed at or prior to the rightmost position, the action is as tabled above. If depressed after the rightmost position, no alignment or filling is performed.

EXIT or E to designate that the EXIT key (or - key) must be used to release the field. If depressed at or prior to the rightmost position, the action is as tabled above. If depressed after the rightmost position, no alignment or filling is performed.

## Verify

Verify designates that the field is to be key-verified when a data verify operation is active. VERIFY or V means verification is required.

Ordering of the parameters shift, source, exit and verify is not important.

## sn-post, sn-pre

These are statement labels which designate processing routines associated with the field. The sn-post routine is executed upon a field-release condition (all of the data has been entered for the field); the exit allows for 'post-processing' of the data to occur before the next field is selected for entry/verify. The sn-pre routine is executed when the operator initiates field update (backspaces into a field); the exit allows for 'pre-processing' of the data to occur before the update actually proceeds.

Post-processing is used for validation, calculating extensions and selecting fields out of normal sequence. Pre-processing is used for "backing-out" previous extensions, inhibiting field update and selecting fields out of normal sequence.

The conditions under which these routines are executed is dependent upon the station operation and the MODE of the KET. These topics are covered in Section 8.

Either or both labels may be omitted to designate that the corresponding form of processing is not required for the field. Ordering is important: if sn-post is omitted and sn-pre is present, the separating comma (,) must precede the sn-pre label.

The following fragments of field-specifications illustrate all coding combinations:

...)		no exits
...)	100	post exits only
...)	100, 200	both exits
...)	, 200	pre exit only

## THE EQUATE STATEMENT

### **EQU**

Purpose: To establish names that may be used as alternate reference names for records and fields.

Format: **EQU: EQU-name-1,**  
**EQU-name-2,**  
**•**  
**•**  
**•**  
**EQU-name-n;**  
**or**  
**EQU: EQU-name-1, EQU-name-2,...EQU-name-n;**

Note: Commas are not required in the vertical syntax.

Description: An EQU statement establishes each EQU-name as a specialized data item having the following characteristics.

- Defining an EQU-name does not cause the name to be associated with any *particular* data field; that is, no memory is allocated and no already allocated memory is re-used.
- An EQU-name (and only an EQU-name) may appear as the "receiving" parameter in a SAME or STRING execution statement. These statements cause the EQU-name to be assigned as an alternate reference name for a record, field, or portion of a record of field.
- An EQU-name used elsewhere in the Execution Section, references the data area established by the last executed SAME or STRING statement that assigned the EQU-name.

## RECORD REMAPPING

The purpose of remapping is to re-use a previously specified record or field area with an alternate layout. One of the following formats may be used:

1. **RCD: RCD-name, REMAPS:previous**  
**field-specification-1**  
**field-specification-2**  
•  
•  
•  
**field-specification-n ;**
2. **RCD: RCD-name (INDEX-name, iterations), REMAPS:previous**  
**field-specification-1**  
**field-specification-2**  
•  
•  
•  
**field-specification-n ;**
3. **KET: KET-name, REMAPS:previous**  
**context-specification-1**  
**context-specification-2**  
•  
•  
•  
**context-specification-n**  
**display-specification-1**  
**display-specification-2**  
•  
•  
•  
**display-specification-m ;**

The new component introduced is:

**previous** The designation of the previously defined record or field that is to be reused with an alternate layout.

If format 1 or format 3 (above) is used, the area designated by "previous" must be of sufficient length to hold the entire record currently being defined. If format 2 (above) is used, the area designated by "previous" must be of sufficient length to hold the entire record currently being defined. If format 2 (above) is used, the area designated by "previous" must be of sufficient length to hold all iterations of the record being defined.

The coding of "previous" is dependent upon the record or field that is being remapped and the remapping objectives. Three types of remapped records are distinguished:

- normal** The record is 256 positions, or less, in length and is not iterated. A normal record may be defined by a KET or RCD.
- extended** The record is 257 positions, or more, in length (and therefore is not iterated). An extended record may be defined by a KET.
- indexed** The record is iterated (and each iteration, therefore, has 256 or fewer positions). The total number of positions for all iterations is not limited to 256 positions. An indexed record is defined only by an RCD array.

## REMAPPING A NORMAL RECORD

A normal record may be remapped by coding "previous" as follows:

- Record-name to remap the entire record;
- Record-name\* also to remap the entire record; or
- field-name to remap a single field of the record.

The syntax Record-name\* has little significance for a normal record. It is allowed for a normal record, however, for convenience when recoding or updating source code. For example, the syntax Record-name\* was used to name an extended record and if this record is changed to a normal record, the syntax Record-name\* need not be changed.

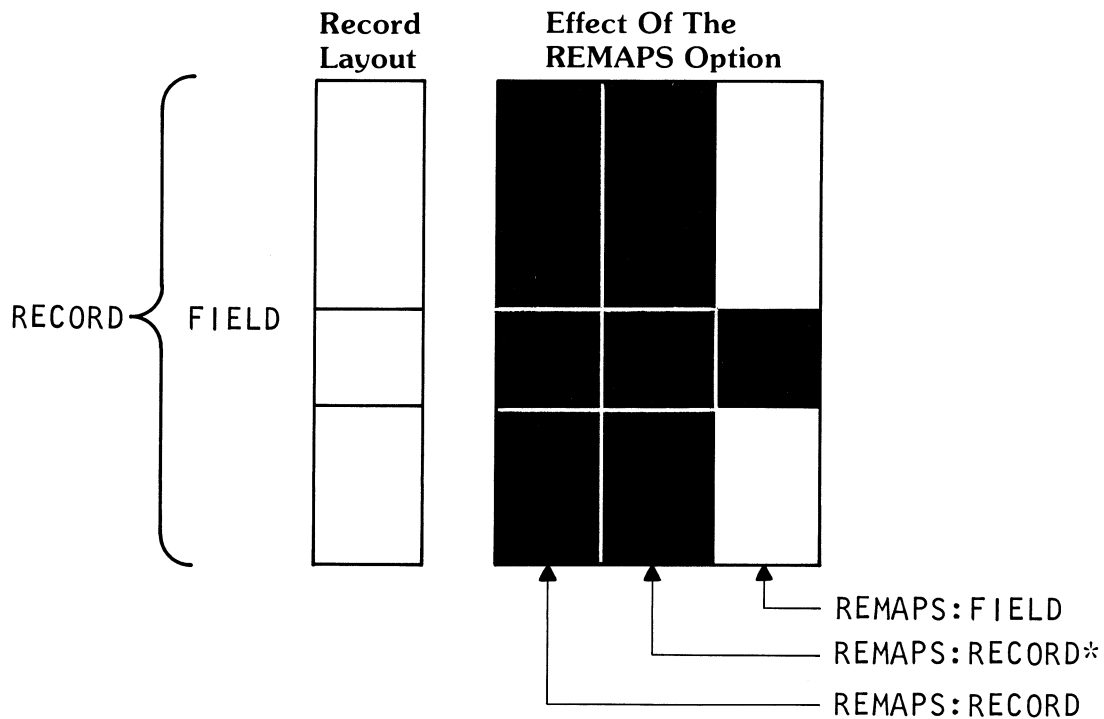
Example:

```

RCD:  RECORD
      .
      .
      .
      FIELD
      .
      .
      . ;
  
```

} length 256 positions, or less

The diagram below illustrates the layout of RECORD as a point of reference with individual shaded layouts to depict the areas that may be remapped by a subsequently defined RCD or KET. Corresponding results would occur if the example were a KET.



## REMAPPING AN EXTENDED RECORD

An extended record may be remapped by coding "previous" as follows:

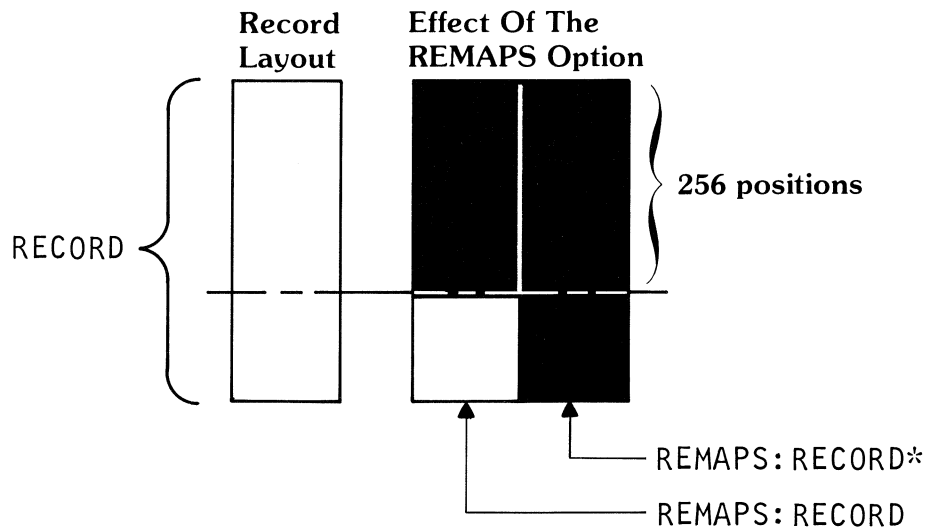
- Record-name to remap the first 256 positions;
- Record-name\* to remap the entire extended record; or
- field-name to remap a single field of the record.

Example:

```
RCD:  RECORD
      .
      .
      .
      FIELD
      .
      .
      .
      ;
```

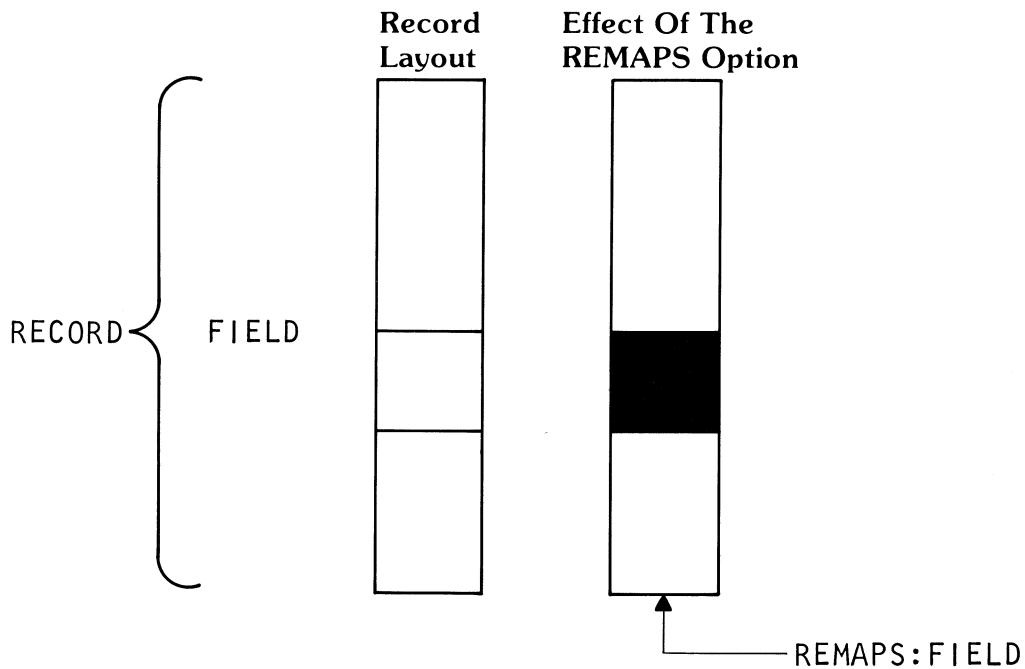
} length 257 positions, or more

The diagram below illustrates the layout of RECORD as a point of reference with individual shaded layouts to depict the areas that may be remapped by a subsequently defined RCD or KET. Corresponding results would occur if the example were a KET.





The diagram below illustrates the field level option for RCD and KET remapping.



### REMAPPING AN INDEXED RECORD

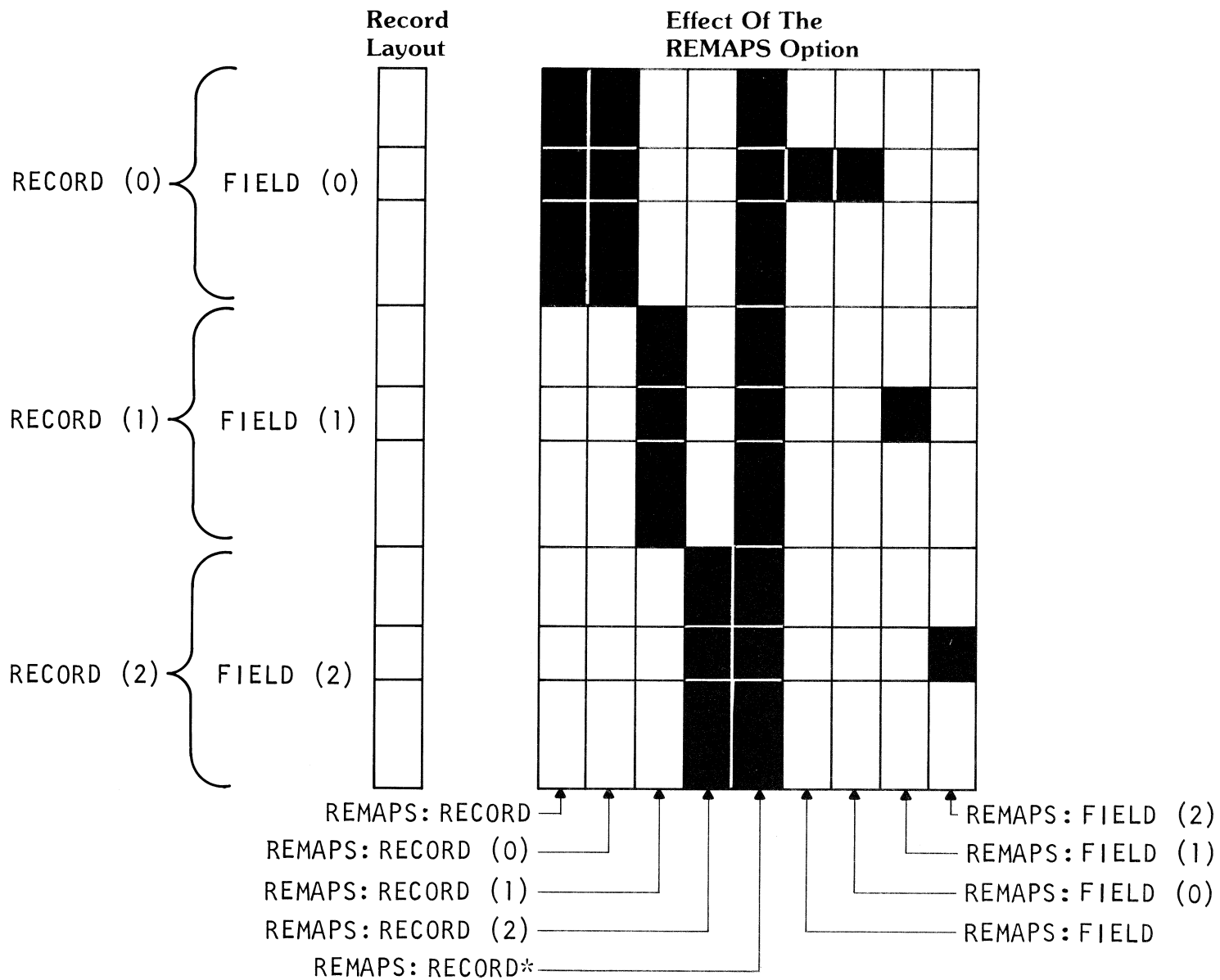
An indexed record may be remapped by coding "previous" as follows:

- RCD-name to remap the first iteration of the record;
- RCD-name (index) to remap a specific, fixed iteration of the record;
- RCD-name\* to remap all iterations of the record;
- field-name to remap a field within the first iteration of the record; or
- field-name (index) to remap a specific, fixed iteration within the record.

Example:

```
RCD:  RECORD (INDEX,3)
      .
      .
      .
      FIELD
      .
      .
      .
      ;
```

The diagram on the following page illustrates the layout of the three RECORD iterations as a point of reference with individual shaded layouts to depict the areas that may be remapped by a subsequently defined RCD or KET.





## SECTION 6: EXECUTION STATEMENTS

Execution statements indicate the processing steps to be applied to the data elements that are defined in the Data Definition Section. Execution statements may have none, one or several statement numbers. These numbers are used to branch from one execution statement to another. Specific rules governing the use of statement numbers and the coding requirements for execution statements are included in Section 2 of this manual.

Execution statements may be grouped into ten functional categories:

1. Move statements—move the contents of one field to another. These statements do not edit or manipulate the characters within the sending field. To prevent data truncation, the receiving field must be at least as long as the sending field. The programmer can also specify the alignment of data within the field; in addition, he may specify a fill character for excess positions in the receiving field. The Move statements are:

MOVE LEFT AND FILL  
MOVE RIGHT AND FILL  
FILL  
MOVE LEFT, NO FILL  
MOVE RIGHT, NO FILL

2. Decimal Arithmetic Statements — perform arithmetic calculations on decimal numbers. The Decimal Arithmetic statements are:

DECIMAL EQUATE  
DECIMAL ADD  
DECIMAL SUBTRACT  
DECIMAL MULTIPLY  
DECIMAL DIVIDE

3. Binary Arithmetic Statements — perform arithmetic calculations on binary numbers. The Binary Arithmetic statements are:

BINARY ADD  
BINARY SUBTRACT  
BINARY MULTIPLY  
BINARY DIVIDE

4. Boolean Statements — perform “logical” operations upon fields at the bit level. The Boolean statements are:

AND  
OR  
XOR (Exclusive OR)

5. Editing Statements—allow the programmer to alter the representation, format and alignment of fields. The Editing statements are:

BINARY  
DECIMAL  
JUSTIFY LEFT  
JUSTIFY RIGHT  
PICTURE EDIT  
CURRENCY  
NUMBER EDIT  
TRANSLATE  
COMPRESS  
DECOMPRESS  
HEX  
UNHEX

6. Control Statements—control the flow of program execution, either conditionally or unconditionally. The Control statements are:

UNCONDITIONAL GO  
STOP  
CONDITIONALS  
COMPUTED GO  
CASE  
ERROR TEST

7. Subroutine Statements—delimit and direct execution of a subroutine. A subroutine is a group of statements performing a specific function that may be performed as often as necessary. The subroutine is coded only once, eliminating repetitive statements. It may be accessed from anywhere within the Execution Section of the program. The Subroutine statements are:

PERFORM  
ENTRY  
EXIT

8. I/O Statements — direct Input/Output operations. I/O operations transfer data between the application program in main memory and the I/O device. Also, I/O statements control the I/O device. The I/O statements are:

OPEN  
CLOSE  
READ  
WRITE  
CHECKEOD  
DELETE  
FREESPACE  
INSERT  
READLOCK  
READNEXT  
RELEASE  
SETEOD  
BACKSPACE  
MARK  
REWIND  
REWINDLOCK  
SKIPFILE  
CHECKFORMS  
PRINT  
SETFORMS  
SENDEOF

9. STATION I/O Statements — provide the necessary interface between the operator and the application for data entry, validation, and display. The STATION I/O statements are:

KENTER  
KVERIFY  
RESUME  
RESUMERR  
ERROR  
NOTIFY  
READSCREEN  
READKEY

10. Special Purpose Statements — perform specialized functions to enhance application program capability. The Special Purpose statements are:

CHECKDIGIT  
GETTIME  
SETTIME  
SAME  
STRING

## **SYNTAX CONVENTIONS**

A format is presented with each statement in this section. Each statement must be coded according to its format. Listed below are conventions (additional to the ones stated in Manual Notations) associated with these formats.

1. The notations :X: and :Y: denote any EBCDIC character delimited by colons. Two coding representations are permitted: graphic or hexadecimal. In the graphic representation, the single character between colons is used as an EBCDIC character. In the hexadecimal representation, two hexadecimal characters between the colons specify one EBCDIC character. (See “Non-Graphic Fill Characters” in Section 5.)
2. The notation sn represents a statement number. It indicates that a statement number will be specified to reference the associated statement.
3. The character A indicates the primary receiving field of an operation. Valid substitutions for this character are:

EQU-name  
INDEX-name  
IOD-name • keyword  
KET-name  
KET-field-name  
RCD-name  
RCD-field-name

4. The characters B and C denote a source field for an operation. Valid substitutions for this character are:

EQU-name  
INDEX-name  
IOD-name  
IOD-name • keyword  
KET-name  
KET-field-name  
literal  
RCD-name  
RCD-field-name

5. The character D denotes the secondary receiving field of an operation. Valid substitutions for this character are identical to those listed for character A.
6. The term EQU-name denotes the primary receiving field of an operation when the receiving field must be a data item defined by the EQU data definition statement.
7. The term buffer may be any one of the following. A buffer designates that the reference is not limited in size.

- EQU-name
- KET-name
- KET-field-name
- RCD-name
- RCD-field-name
- KET-name\*
- RCD-name\*
- literal

8. Symbols A, B, C, D and buffer shown in the statement formats are presented for descriptive clarity. They do not require that separate fields be individually substituted. For example, the actual statement COUNT=COUNT+1 is compatible with the format A=B+C called out for the DECIMAL ADD statement.
9. All fields denoted by the symbols A, B, C, and D are processed up to a maximum of 256 bytes unless otherwise specified in the individual statement descriptions.
10. The unary minus may be applied to numeric literals (e.g., -21) as shown in Table 6-1. The unary minus must not be used with non-numeric literals (e.g., -'ABC') or field-names (e.g., -COUNT) unless explicitly permitted in the individual statement descriptions.
11. The exception exit shown in a statement format may be coded as EOF (end-of-file), EOM (end-of-medium), or ESC (escape). All three notations are equivalent.
12. Ordering of the exception exit and ERR exit, within a format, is not significant.
13. Either or both abnormal exits may be omitted. The preceding comma (,) must also be omitted in these cases.

## SEMANTICS CONVENTIONS

### INTERPRETATION OF LITERAL VALUES BY INSTRUCTION TYPE

Operations deal with fields containing character strings and/or numbers.

A non-numeric literal (character string) is processed by using the exact value stated in the literal.

A numeric value is processed in one of two ways, depending upon the type of instruction specified. When a decimal oriented instruction is used, the value stated is coded directly as a literal and the compiler automatically creates a field containing the designated decimal value. When a binary oriented instruction is used, the value stated is converted to binary representation. The compiler then creates a field containing the converted binary value. An instruction that does not appear to be binary or decimal is treated as a decimal oriented instruction.

Table 6-1 illustrates the interpretations of character strings and numeric values made by the compiler.

**Table 6-1: INTERPRETATION OF LITERAL VALUES BY INSTRUCTION TYPE**

Decimal Oriented Instruction	NUMERIC			NON-NUMERIC Hexadecimal, composite or Alpha- numeric Strings
	Unsigned	Signed		
	1000	+1000	-1000	'ABCD'
	Internal Represent- ation=  F1 F0 F0 F0	Internal Represent- ation=  F1 F0 F0 F0	Internal Represent- ation=  F1 F0 F0 D0	Internal Representation=  C1 C2 C3 C4
Binary Oriented Instruction	NUMERIC			NON-NUMERIC Hexadecimal, composite or Alpha- numeric Strings
	Unsigned	Signed		
	1000	+1000	-1000	'ABCD'
	Internal Represent- ation=  03E8	Internal Represent- ation=  03E8	Diagnostic Message	Internal Representation=  C1 C2 C3 C4

NOTE: Numeric literals employed in any of the binary class of statements must be  $\geq 0$  and  $\leq 16,383$

Example: A = ADD(99,C)



## INDEXED DATA REFERENCES

When an indexed RCD-name or indexed RCD-field-name is to be referenced, the specific iteration of the record or field is selected at compile-time or at execution-time depending on how the reference is coded. The explicit indexing notation is used if the selection is to be made at compile-time and the implicit notation is used if the selection is to be made at execution-time.

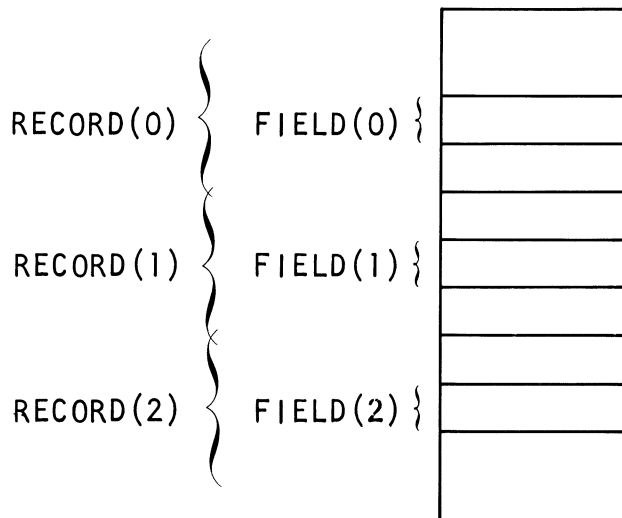
### Explicit Notation

An explicit reference is coded for an execution statement if the statement is used to access a fixed iteration of the record or field. The particular iteration to be selected is specified by means of an iteration number coded immediately after the Record-name or field-name, for example, ACCOUNT (1).

Example:

```
RCD:  RECORD (INDEX, 3)
      .
      .
      .
      FIELD
      .
      .
      .
      ;
```

The actual layout of the three iterations of RECORD is shown below. Explicit notations are listed to the left in order to designate the area referenced by each notation that may be coded within the Execution Section.



## Implicit Notation

An implicit reference is coded for an execution statement if the statement is used to access an iteration which may vary from one execution of the statement to another. The particular iteration selected for reference is determined by the current value of its associated index variable (INDEX-name). The iteration selected for implicit reference is changed each time its associated index variable is assigned a new value. An index value of zero selects the first iteration, one selects the second iteration, and so on.

An implicit reference is distinguished by the absence of an iteration number coded as a part of the reference, for example, ACCOUNT.

Example: The layout for the RCD example under Explicit Notation is shown next with shading used to designate the area referred to by the notations RECORD and FIELD for each value of INDEX. Explicit references are shown to the left for identification purposes.

		RECORD			FIELD		
		Current Index Value			Current Index Value		
		00	01	02	00	01	02
RECORD(0)	FIELD(0)						
RECORD(1)	FIELD(1)						
RECORD(2)	FIELD(2)						

The table below illustrates the various forms of coding and the interpretation placed upon an indexed RCD-name or indexed RCD-field-name reference.

**Table 6-2: INTERPRETATION OF INDEXED DATA REFERENCES**

Reference Notation	Categories of References	
	A,B,C, or D	buffer
RCD-name	Iteration of RCD-name designated by the current value of its index-name.	Iteration of RCD-name designated by the current value of its index-name.
RCD-field-name	RCD-field-name within the iteration of RCD-name designated by the current value of its index variable.	RCD-field-name within the iteration of RCD-name designated by the current value of its index variable.
RCD-name(n)	Specifically designated iteration of RCD-name.	Specifically designated iteration of RCD-name.
RCD-field-name(n)	Field-name within the specifically designated iteration of RCD-name.	Field-name within the specifically designated iteration of RCD-name.
RCD-name *	Not applicable	All iteration of RCD-name (overall length may exceed 265 bytes).

**NON-INDEXED DATA REFERENCES**

EQU variables and non-indexed records may correspond to fields with length in excess of 256 positions. The table below illustrates the various forms of coding and the interpretation placed upon the reference by the compiler.

**TABLE 6-3: INTERPRETATION OF NON-INDEXED DATA REFERENCES**

Reference Notation	Categories of Reference			
	A,B,C, or D		buffer	
EQU-name	First 256 positions	All positions	All positions	All positions
RCD-name or KET-name	First 256 positions	All positions	All positions	First 256 positions
RCD-name*or KET-name*	Not Applicable	Not Applicable	All positions	All positions
		Length 256 positions		
		Length in excess of 256 positions		

## OTHER DATA REFERENCES

All other data items are limited to 256 positions. For these types, the data-name used as a reference designates the entire field area.

Selected substitutions for A and/or D in a statement format cause additional processing to occur as outlined below:

- |                           |   |
|---------------------------|---|
| INDEX-name                | causes a new iteration of the associated record to be selected.   |
| KET-field-name            | causes the corresponding display area to be updated, provided that the KET containing the field is active.  |
| STATION-IOD-name • FLDNUM | causes the KET field identified by the FLDNUM parameter to be selected as the active field (see Section 8). |
| STATION-IOD-name • CURFLD | causes the currently active KET field to be updated on the display (see Section 8).                         |

Additional processing invoked due to a substitution for D occurs before any processing invoked due to substitution for A.

## DESCRIPTION OF STATEMENTS

Statements are presented by functional group. The following is presented for each statement:

- Purpose
- Format
- Description
- Example

## MOVE STATEMENTS

Move statements move the contents of one field to another. These statements do not edit or manipulate the characters within the sending field. To prevent data truncation, the receiving field must be at least as long as the sending field. The programmer can also specify the alignment of data within the field; in addition, he may specify a fill character for excess positions in the receiving field. The Move statements are:

MOVE LEFT AND FILL

MOVE RIGHT AND FILL

FILL

MOVE LEFT, NO FILL

MOVE RIGHT, NO FILL

## MOVE LEFT AND FILL

Purpose: To move data from one field to another, left-aligned, with remaining positions padded with a specified fill character.

Format: A = B:X:

Description: Beginning with the leftmost position, the contents of field B are moved, one byte at a time, to the leftmost position of field A. Any remaining positions of field A are padded with the fill character specified within the colons.

If field A is shorter than field B, the transfer of data is terminated when field A is full and no filling occurs.

Example: FLDA = FLDB:\*:

**BEFORE EXECUTION**

**AFTER EXECUTION**

FLDB = 123

FLDB = 123

FLDA = XXXXXX

FLDA = 123\*\*\*

FLDB = 1234

FLDB = 1234

FLDA = XXX

FLDA = 123

## MOVE RIGHT AND FILL

Purpose: To move data from one field to another, right-aligned, with remaining positions padded with a specified fill character.

Format: A = :X:B

Description: Beginning with the rightmost position, the contents of field B are moved, one byte at a time, to the rightmost positions of field A. Any remaining positions of field A are padded with the fill character specified within the colons.

If field A is shorter than field B, the transfer of data is terminated when field A is full and no filling takes place.

Example: FLDA = :\* : FLDB

**BEFORE EXECUTION**

FLDB = 123

FLDA = XXXXXX

FLDB = 1234

FLDA = XXX

**AFTER EXECUTION**

FLDB = 123

FLDA = \*\*\*123

FLDB = 1234

FLDA = 234



## FILL

Purpose: To fill a field with a specified fill character.

Format: A = :X:

Description: The fill character specified within the colons is moved to each position of field A.

Example: FLDA = :\*:

**BEFORE EXECUTION**

FLDA = (insignificant)

**AFTER EXECUTION**

FLDA = \*\*\*\*\*

## MOVE LEFT, NO FILL

Purpose: To move data from one field to another, left-aligned.

Format: A = B::

Description: Beginning with the leftmost position, the contents of field B are moved, one byte at a time, to the leftmost positions of field A. Any remaining positions of field A are unchanged.

If field A is shorter than field B, the transfer of data is terminated when field A is full.

Example: FLDA = FLDB::

**BEFORE EXECUTION**

**AFTER EXECUTION**

FLDB = 123

FLDB = 123

FLDA = XXXXXX

FLDA = 123XXX

FLDB = 12345

FLDB = 12345

FLDA = XXX

FLDA = 123

The two colons (::) must be coded to specify the direction of data movement (in this case, left-to-right).

These colons must be contiguous. If a space character is inserted between the colons, it will be considered a fill character; therefore, a MOVE LEFT AND FILL statement.

## MOVE RIGHT, NO FILL

Purpose: To move data from one field to another, right-aligned.

Format: A = ::B

Description: Beginning with the rightmost position, the contents of field B are moved, one byte at a time, to the rightmost positions of field A. Any remaining positions of field A are unchanged.

If field A is shorter than field B, the transfer of data is terminated when field A is full.

Example: FLDA = ::FLDB

### **BEFORE EXECUTION**

FLDB = 123

FLDA = XXXXXX

FLDB = 1234

FLDA = XXX

### **AFTER EXECUTION**

FLDB = 123

FLDA = XXX123

FLDB = 1234

FLDA = 234

The two colons (: :) must be coded to specify the direction of data movement. In this case, right-to-left.

These colons must be contiguous. If a space character is inserted between the colons, it will be considered a fill character, therefore, a MOVE RIGHT AND FILL statement.

## DECIMAL ARITHMETIC STATEMENTS

Decimal Arithmetic statements perform arithmetic calculations on decimal numbers. The Decimal Arithmetic statements are:

DECIMAL EQUATE

DECIMAL ADD

DECIMAL SUBTRACT

DECIMAL MULTIPLY

DECIMAL DIVIDE

## DECIMAL EQUATE

Purpose: To move the contents of a decimal field to another field.

Format:

1.  $A = B$
2.  $A = -B$

Description:

1. The contents of field B are moved to the rightmost positions of field A. The sign of the resulting field A is the same as the sign of field B.
2. The contents of field B are moved to the rightmost positions of field A. The sign of the resulting field A is the opposite of the sign of field B.

If field A is shorter than field B, the transfer of data is terminated when field A is full. If field A is longer than field B, the leftmost positions of field A are filled with zeros.

Example 1:  $FLDA \approx FLDB$

**BEFORE EXECUTION**

FLDB = 231<sup>+</sup>

FLDA = (insignificant)

**AFTER EXECUTION**

FLDB = 231<sup>+</sup>

FLDA = 231<sup>+</sup>

Example 2:  $FLDA = -FLDB$

**BEFORE EXECUTION**

FLDB = 231<sup>+</sup>

FLDA = (insignificant)

**AFTER EXECUTION**

FLDB = 231<sup>+</sup>

FLDA = 231

## DECIMAL ADD

Purpose: To obtain the sum of two decimal fields.

Format:  $A = B + C$   
Where B and/or C may be negative.

Description: The contents of field B and field C are added together with the result stored in field A.

If field A is not long enough to hold the maximum possible result, the error indicator is set and the result value in field A may not be correct. If field A is longer than required, the leftmost positions of A are zero filled.

Example: FLDA = FLDB + FLDC

**BEFORE EXECUTION**

**AFTER EXECUTION**

FLDB = 100<sup>+</sup>

FLDB = 100<sup>+</sup>

FLDC = 30<sup>-</sup>

FLDC = 30<sup>-</sup>

FLDA = (insignificant)

FLDA = 0070<sup>+</sup>

The sign control for DECIMAL ADD is as follows:

<u>Sign of Field B</u>	<u>Sign of Field C</u>	<u>Sign of Field A</u>
+	+	+
-	-	-
+	-	Sign of field of greater absolute value
-	+	Sign of field of greater absolute value

A negative decimal number is stored with the units position modified to reflect the negative value so that an additional sign position is not required. These numbers should be edited prior to printing. Printing an unedited negative decimal number causes the units position to be printed as J through R instead of 1 through 9. The graphic character associated with the modified zero is not a standard graphic. When a negative decimal number is displayed on the CRT screen in a NUMERIC KET field, the units position will display 0 through 9 and the display attribute byte is changed to REVERSE.

## DECIMAL SUBTRACT

Purpose: To obtain the difference of two decimal fields.

Format:  $A = B - C$

Description: The contents of field C are subtracted from the contents of field B with the result stored in field A.

If field A is not long enough to hold the minimum possible result, the error indicator is set and the result value in field A may not be correct. If field A is longer than required, the leftmost positions of field A are filled with zeros.

Example:  $FLDA = FLDB - FLDC$

**BEFORE EXECUTION**

$FLDB = 400^+$

$FLDC = 50^+$

$FLDA = (\text{insignificant})$

**AFTER EXECUTION**

$FLDB = 400^+$

$FLDC = 50^+$

$FLDA = 000350^+$

The sign control for DECIMAL SUBTRACT is as follows:

<u>Sign of Field B</u>	<u>Sign of Field C</u>	<u>Sign of Field A</u>
+	+	If field C is of greater absolute value, the result is negative. Otherwise, the sign is positive.
-	-	If field B is of greater absolute value, the result is negative. Otherwise, the result is positive.
+	-	+
-	+	-

## DECIMAL MULTIPLY

Purpose: To obtain the product of two decimal fields.

Format:  $A = B * C$

Description: The contents of field B are multiplied by the contents of field C, the result is stored in field A.

Fields A, B and C have the following length limitations:

A— 40 positions

B— 20 positions

C— 20 positions

If field A is not long enough to hold the result, field A is truncated on the left and only the least significant digits are stored. In this case, the error indicator is set.

If field A is longer than required, the leftmost positions of field A are filled with zeros.

Example:  $FLDA = FLDB * FLDC$

**BEFORE EXECUTION**

**AFTER EXECUTION**

FLDB = 60<sup>+</sup>

FLDB = 60<sup>+</sup>

FLDC = 8<sup>+</sup>

FLDC = 8<sup>+</sup>

FLDA = (insignificant)

FLDA = 00480<sup>+</sup>

The sign control for DECIMAL MULTIPLY is as follows:

<u>Sign of Field B</u>	<u>Sign of Field C</u>	<u>Sign of Field A</u>
+	+	+
-	-	+
+	-	-
-	+	-



## DECIMAL DIVIDE

Purpose: To obtain the quotient and remainder of one decimal field divided by another.

Format:

1.  $A = B/C$
2.  $A = B/C, D$

Description:

1. The contents of field B are divided by the contents of field C with the result stored in field A.
2. The contents of field B are divided by the contents of field C with the result stored in field A. The remainder is stored in field D.

Fields A, B, C and D have the following length limitations:

A— 20 positions  
B— 40 positions  
C— 20 positions  
D— 20 positions

If field A and/or field D is not long enough to hold the result, field A or field D is truncated on the left and only the least significant digits are stored. In this case, the error indicator is set. If field A (or field D) is longer than required, the leftmost positions of A (or D) are filled with zeros.

Division by zero is not defined. In this case, field A is zero filled and field D is set to the same value as field B. Also, the error indicator is set.

Example: FLDA = FLDB/FLDC

BEFORE EXECUTION	AFTER EXECUTION
FLDB = 20 <sup>+</sup> 6	FLDB = 20 <sup>+</sup> 6
FLDC = 3 <sup>+</sup>	FLDC = 3 <sup>+</sup>
FLDD = (insignificant)	FLDD = 2 <sup>+</sup>
FLDA = (insignificant)	FLDA = 00006 <sup>+</sup> 8

Sign control for DECIMAL DIVIDE is as follows:

<u>Sign of Field B</u>	<u>Sign of Field C</u>	<u>Sign of Field D</u>	<u>Sign of Field A</u>
+	+	+	+
-	-	-	+
+	-	+	-
-	+	-	-

NOTE: the relationship between fields A, B, C and D also may be expressed as:  $B=A*C+D$

## BINARY ARITHMETIC STATEMENTS

Binary Arithmetic statements perform arithmetic calculations on binary numbers. The Binary Arithmetic statements are:

BINARY ADD

BINARY SUBTRACT

BINARY MULTIPLY

BINARY DIVIDE

## **BINARY ADD**

Purpose: To obtain the sum of two binary fields.

Format: A = ADD(B,C)

Description: The contents of field B and field C are added together with the result stored in field A.

The operation proceeds right-to-left, one character (8 bits) at a time until field A is full.

Whenever a field is referenced as a source field in a BINARY ADD statement and the receiving field is longer than the source field, the source field is logically prefixed with a sufficient number of zero bits to force length concurrence. Similarly, if the receiving field is shorter than a source field, then the excess bits of the source field are ignored.

Example: FLDA = ADD(FLDB,FLDC)

### **BEFORE EXECUTION**

FLDB = 0000 0000 0111 1111

FLDC = 0000 0000 0011 1111

FLDA = (insignificant)

### **AFTER EXECUTION**

FLDB = 0000 0000 0111 1111

FLDC = 0000 0000 0011 1111

FLDA = 0000 0000 1011 1110

The error indicator is affected by this instruction but no significance can be associated with its setting.

## BINARY SUBTRACT

Purpose: To obtain the difference of two binary fields.

Format A = SUB(B,C)

Description: The contents of field C are subtracted from the contents of field B. The result is stored in field A.

The operation proceeds right-to-left, one character at a time, until field A is full. (In effect, BINARY SUBTRACT adds the two's complement of field C to field B.)

Whenever a field is referenced as a source field in a BINARY SUBTRACT statement and the receiving field is longer than the source field, the source field is logically prefixed with a sufficient number of zero bits to force length concurrence. Similarly, if the receiving field is shorter than a source field, then the excess bits of the source field are ignored.

In particular, if field C is shorter than field B, the prefixing of zero bits occurs prior to the two's complement operation.

Example: FLDA = SUB(FLDB,FLDC)

### **BEFORE EXECUTION**

FLDB = 0000 0000 0111 1111

FLDC = 0000 0000 0011 1111

FLDA = (insignificant)

FLDB = 0000 0000 0011 1111

FLDC = 0000 0000 0111 1111

FLDA = (insignificant)

### **AFTER EXECUTION**

FLDB = 0000 0000 0111 1111

FLDC = 0000 0000 0011 1111

FLDA = 0000 0000 0100 0000

FLDB = 0000 0000 0011 1111

FLDC = 0000 0000 0111 1111

FLDA = 1111 1111 1100 0000

The error indicator is affected by this instruction but no significance can be associated with its setting.

## BINARY MULTIPLY

Purpose: To obtain the product of two binary fields.

Format:  $A = \text{MPY}(B, C)$

Description: The contents of field B and field C are multiplied together with the result stored in field A.

The operation proceeds right-to-left, one character at a time, until field A is full.

Field A cannot exceed 8 bytes (64 bits) in length; field B and C are limited to 4 bytes each (32 bits).

Whenever a field is referenced as a source field in a BINARY MULTIPLY statement and the receiving field is longer than the source field, the source field is logically prefixed with a sufficient number of zero bits to force length concurrence. Similarly, if the receiving field is shorter than the source field, then the excess bits of the source field are ignored.

Example:  $\text{FLDA} = \text{MPY}(\text{FLDB}, \text{FLDC})$

### **BEFORE EXECUTION**

FLDB = 0000 0000 0011 1100

FLDC = 0000 0000 0000 1000

FLDA = (insignificant)

### **AFTER EXECUTION**

FLDB = 0000 0000 0011 1100

FLDC = 0000 0000 0000 1000

FLDA = 0000 0001 1110 0000

The error indicator is affected by this instruction but no significance can be associated with its setting.

## BINARY DIVIDE

Purpose: To obtain the quotient and remainder of one binary field divided by another.

Format:  
1.  $A = \text{DIVR}(B, C)$   
2.  $A = \text{DIVR}(B, C, D)$

Description:

1. The contents of field B are divided by the contents of field C with the result stored in field A. The operation proceeds right-to-left, one character at a time, until field A is full.
2. The contents of field B are divided by the contents of field C. The result is stored in field A. The remainder is stored in field D. The operation proceeds right-to-left, one character at a time, until field A is full.

Fields A, C and D cannot exceed 4 bytes (32 bits) in length; field B is limited to 8 bytes (64 bits).

Whenever a field is referenced as a source field in a BINARY DIVIDE statement and the receiving field is longer than the source field, the source field is logically prefixed with a sufficient number of zero bits to force length concurrence. Similarly, if the receiving field is shorter than a source field, then the excess bits of the source field are ignored.

Example:  $\text{FLDA} = \text{DIVR}(\text{FLDB}, \text{FLDC}, \text{FLDD})$

### **BEFORE EXECUTION**

FLDB = 0000 0001 1110 0010

FLDC = 0000 0000 0000 1000

FLDD = 0000 0000 0000 0000

FLDA = (insignificant)

### **AFTER EXECUTION**

FLDB = 0000 0001 1110 0010

FLDC = 0000 0000 0000 1000

FLDD = 0000 0000 0000 0010

FLDA = 0000 0000 0011 1100

The error indicator is affected by this instruction but no significance can be associated with its setting.

## BOOLEAN STATEMENTS

Boolean statements perform “logical” operations upon fields at the bit level. The Boolean Statements are:

AND

OR

XOR (Exclusive OR)

## AND

Purpose: To form the logical product of two fields.

Format:  $A = \text{AND}(B, C)$

Description: From right-to-left, each bit position of field B is logically AND'ed with the corresponding bit position of field C to produce a result for the corresponding position of field A. This operation terminates when field A is full or when all positions of field B or field C have been processed.

Example:  $\text{FLDA} = \text{AND}(\text{FLDB}, \text{FLDC})$

**BEFORE EXECUTION**

FLDB = 1101 0101

FLDC = 1010 1010

FLDA = (insignificant)

**AFTER EXECUTION**

FLDB = 1101 0101

FLDC = 1010 1010

FLDA = 1000 0000

The diagram below illustrates the result of logically ANDing a bit pair:

		C Bit Value	
		0	1
B Bit Value	0	0	0
	1	0	1

The intersection of the row and column indicates the appropriate A bit value.



## OR

Purpose: To form the logical sum of two fields.

Format:  $A = OR(B, C)$

Description: From right-to-left, each bit position of field B is logically OR'ed with the corresponding bit position of field C to produce a result for the corresponding position of field A. The operation terminates when field A is full or when all positions of field B or field C have been processed.

Example:  $FLDA = OR(FLDB, FLDC)$

**BEFORE EXECUTION**

FLDB = 1100 0111

FLDC = 1010 1010

FLDA = (insignificant)

**AFTER EXECUTION**

FLDB = 1100 0111

FLDC = 1010 1010

FLDA = 1110 1111

The diagram below illustrates the result of logically ORing a bit pair.

		C Bit Value	
		0	1
B Bit Value	0	0	1
	1	1	1

The intersection of the row and column indicates the appropriate A bit value.

## XOR (Exclusive OR)

Purpose: To perform a logical Exclusive OR comparison between two fields.

Format:  $A = \text{XOR}(B, C)$

Description: From right-to-left, each bit position of field B is logically EXCLUSIVE OR'ed with the corresponding bit position of field C to produce a result for the corresponding position of field A. The operation terminates when field A is full or when all positions of field B or field C have been processed.

Example:  $\text{FLDA} = \text{XOR}(\text{FLDB}, \text{FLDC})$

**BEFORE EXECUTION**

FLDB = 1100 0101

FLDC = 1010 1010

FLDA = (insignificant)

**AFTER EXECUTION**

FLDB = 1100 0101

FLDC = 1010 1010

FLDA = 0110 1111

The diagram below illustrates the result of performing a logical Exclusive OR on a bit pair.

		C Bit Value	
		0	1
B Bit Value	0	0	1
	1	1	0

The intersection of the row and column indicates the appropriate A bit value.

## EDITING STATEMENTS

Editing statements allow the programmer to alter the representation, format and alignment of fields. The Editing statements are:

BINARY

DECIMAL

JUSTIFY LEFT

JUSTIFY RIGHT

PICTURE EDIT

CURRENCY

NUMBER EDIT

TRANSLATE

COMPRESS

DECOMPRESS

HEX

UNHEX

## BINARY

Purpose: To convert a number field in decimal representation into a number field in binary representation.

Format A = BINARY(B)

Description: The decimal value of field B is converted to binary representation and moved right-to-left to field A. The conversion terminates when all positions of field B (up to a maximum of 10 positions) have been processed or when field A is full.

Termination due to the completion of field B causes the unused positions of field A to be filled with leading null characters.

Example: FLDA = BINARY(FLDB)

### BEFORE EXECUTION

FLDB (Hex Code) = F0 F0 F2 F5 F9

FLDA (Binary Code) = 0000 0000 0000 0000

### AFTER EXECUTION

FLDB (Hex Code) = F0 F0 F2 F5 F9

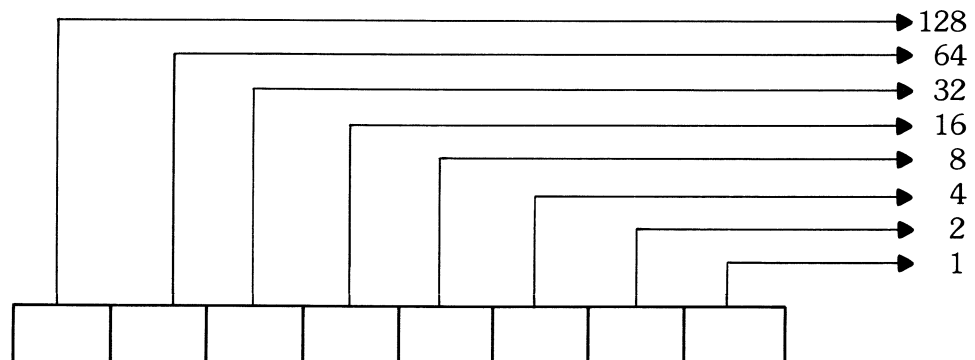
FLDA (Binary Code) = 0000 0001 0000 0011

Hex representation of FLDA=X'01 03'

**NOTE:** The contents of field B must be a positive decimal number or the results will be unpredictable.

The error indicator is affected by this instruction but no significance can be associated with its setting.

**NOTE:** A binary number field contains a binary digit in each bit. Each character in a field contains eight(8) binary digits. The rightmost bit is the unit's position, the next bit to the left is the two's position, the next to the left is the four's position, etc. This form of binary representation is unsigned.



## DECIMAL

Purpose: To convert a number field in binary representation into a number field in decimal representation.

Format: A = DECIMAL(B)

Description: The binary contents of field B are converted to decimal and moved to the rightmost positions of field A.

A decimal number field contains a decimal digit in each position. The rightmost position is the unit's position, the next to rightmost is the ten's position, etc.

The operation terminates when field A is full (maximum 10 positions) or when all positions of field B (or the four rightmost positions of field B) are converted. Unused positions of field A are filled with leading decimal zero characters.

The error indicator is affected by this instruction but no significance can be associated with its setting.

Example: FLDA = DECIMAL(FLDB)

### **BEFORE EXECUTION**

FLDB (Hex Code) = X'FFFF'

FLDA (Graphic Code) = 00000

### **AFTER EXECUTION**

FLDB (Hex Code) = X'FFFF'

FLDA (Graphic Code) = '65535'

**NOTE:** The leftmost bit of a binary field is not regarded as a sign bit.

## JUSTIFY LEFT

Purpose: To move significant characters from one field to the leftmost positions of another.

Format: A = JUSTIFY(B:X:)

Description: Field B is moved to the leftmost positions of field A, ignoring all leading spaces and zeros. The rightmost positions in field A that are not required for the edited data from field B are padded with the fill character specified within the colons.

Example: FLDA = JUSTIFY(FLDB:\*:)

### BEFORE EXECUTION

FLDB = 030

FLDA = XXXXXX

FLDB = 030

FLDA = XXXXXX

### AFTER EXECUTION

FLDB = 030

FLDA = 30\*\*\*\*

FLDB = 030

FLDA = 30\*\*

If field A is too small to hold all of the significant characters from the edited field B, the transfer of data is terminated when field A is full. Spaces and zeros following the leftmost significant character in field B are moved to field A.

## JUSTIFY RIGHT

Purpose: To move significant characters from one field to the rightmost positions of another.

Format: A = JUSTIFY(:X:B)

Description: Field B is moved to the rightmost positions of field A, ignoring all trailing spaces. The leftmost positions in field A that are not required for edited data from field B are padded with the fill character specified within the colons.

Example: FLDA = JUSTIFY(:\*:FLDB)

<b>BEFORE EXECUTION</b>	<b>AFTER EXECUTION</b>
FLDB = 030 <del> </del> <del> </del> <del> </del>	FLDB = 030 <del> </del> <del> </del> <del> </del>
FLDA = XXXXXXX	FLDA = ****030
FLDB = <del> </del> 030 <del> </del> <del> </del>	FLDB = <del> </del> 030 <del> </del> <del> </del>
FLDA = XXXXXXX	FLDA = ** <del> </del> 030

If field A is too small to hold all of the significant characters from the edited field B, the transfer of data terminates when field A is full.

Spaces to the left of the rightmost significant characters in field B are moved to field A.

Zero is always a significant character in this operation.

## PICTURE EDIT

Purpose: To edit a numeric field according to an edit mask.

Format: A = PICTURE (B,C)

Description: The contents of field B are moved to field A, right-to-left, according to the mask in field C.

The unit's position in field B is unique in that the sign of the field is recorded there as well as the unit's magnitude. For this reason, a negative digit from the unit's position is converted to its corresponding positive representation when transcribed. This conversion is not applicable to any other digit position.

Field B is equal to zero if all positions are zero.

Field B is greater than zero if any position is not zero and the unit's position does not contain a modified digit.

Field B is less than zero if any position is not zero and the unit's position does contain a modified digit.

The mask in field C is regarded as a string of editing characters which specify the editing details. The mask has either a right part only, or both a right part and a left part. The rightmost "V" character within the mask determines the end of the right part and the beginning of the left part. The essential difference between the right part and the left part is the change in function associated with certain edit characters and non-significant zeros.

The term currency-symbol refers to the '\$' character. Any character sequence of up to three characters may be defined as an insertion substitute for the standard currency symbol. The '\$' character always remains the edit character that must be coded in the mask itself.

The edit characters 'S', '+', '-', and '\$' are termed floating if they appear more than once within the mask.



Example:

FLDA = PICTURE(FLDB , FLDC)

7/04/76            070476    Z9/99/99

The following table defines the editing action associated with each edit character:

<u>EDIT CHARACTER</u>	<u>EDITING ACTION</u>
9	Transcribes one character from field B.
Z	Transcribes one character from field B if the character is not zero. If the zero is significant, then the zero is transcribed; otherwise, an asterisk is substituted.
*	Transcribes one character from field B if the character is not zero. If the zero is significant, then the zero is transcribed; otherwise, an asterisk is substituted.
Y	Transcribes one character from field B if the character is not zero. A space is substituted for a zero, whether or not the zero is significant.
∅	Transcribes the edit character itself (space).
V	Ends the right part of a mask and begins the left part.
, /	In right-part processing, the mask character itself is transcribed unconditionally.  In left-part processing, the mask character itself is transcribed if the character from field B is significant. If the character from field B is not significant, then one of the following three editing actions occurs:  <ol style="list-style-type: none"><li>1. an asterisk is substituted if the next edit character is an asterisk.</li><li>2. the current edit character is skipped and the floating insertion is effected immediately if the next edit character is a floating edit character.</li><li>3. a space is substituted if the next edit character is other than above.</li></ol>

**EDIT  
CHARACTER**

**EDITING ACTION**

S If the edit character occurs only once within the mask, a plus or a minus character is substituted in accordance with the positive or negative value of the field.

If the edit character is floating and the character from field B is significant, the field B character is transcribed; otherwise the appropriate sign character is substituted and the remainder of field A is space-filled without further reference to field B or the mask.

+ Identical to S, except the space character is substituted instead of a minus.

- Identical to S, except the space character is substituted instead of a plus.

\$ If the edit character occurs only once within the mask, the currency symbol is substituted.

If the character is floating and the character from field B is significant, then the field B character is transcribed; otherwise, the currency symbol is transcribed and field A is space-filled without further reference to field B or the mask.

other

Any edit character not explicitly listed above is inserted directly if the value of field B is less than zero; otherwise, a space is substituted. The mask symbols DB and CR are edited in this manner.

Operation is terminated when field A is full. If field B is exhausted first, characters of zero are selected for continued editing. If field C is exhausted, the edit character 'Z' is used for continued editing.

Example:

The sign associated with field B is shown here as a separate position only for clarity. In practice, the field B sign would be represented with a modified unit's position.

<u>FIELD B VALUE</u>	<u>FIELD C MASK</u>	<u>FIELD A RESULT</u>
0	9	0
1	9	1
10	9	10
0	Z	<del>0</del>
1	Z	1
10	Z	10
0	99	00
0	9Z	0 <del>0</del>
0	*	*
1	*	1
10	*	10
0	*9	*0
1	*9	*1
10	*9	10
0	9 9	0 <del>0</del> 0
1	9 9	0 <del>0</del> 1
10	9 9	1 <del>0</del> 0
0	.9	.0
1	.9	.1
10	.9	1.0
0	.V9	0
1	.V9	1
10	.V9	1.0
0	*,**9	*,**0
0	*,**V9	****0
100	\$\$,\$\$9	\$,100
100	\$\$,\$\$V9	\$100
1000	\$\$,\$\$V9	\$1,000
100000	Z,ZZZV.99	1,000.00

<u>FIELD B VALUE</u>	<u>FIELD C MASK</u>	<u>FIELD A RESULT</u>
10000	Z,ZZZV,99	100.00
10	Z,ZZZV.99	.10
100000	Z,ZZZV,99	1,000.00
10000	Z,ZZZV,99	100.00
10	Z.ZZZV,99	,10
1000	SS,SSV9	+1,000
-1000	SS,SSV9	-1,000
100	SS,SSV9	+100
-100	SS,SSV9	-100
1000	++,++V9	+1,000
-1000	++,++V9	1,000
100	--,--V9	100
-100	--,--V9	-100
0	9CR	0 <del>00</del>
1	9CR	1 <del>00</del>
-1	9CR	1CR
1	9 <sup>△</sup> DB	1 <del>00</del> <del>00</del>
-1	9 <sup>△</sup> DB	1 <del>00</del> <del>00</del>
0	\$\$,\$\$\$V.99CR	\$.00 <del>00</del>
100	\$\$,\$\$\$V.99CR	\$1.00 <del>00</del>
10000	\$\$,\$\$\$V.99CR	\$100.00 <del>00</del>
100000	\$\$,\$\$\$V.99CR	\$1,000.00 <del>00</del>
-100	\$\$,\$\$\$V.99CR	\$1.00CR
-10000	\$\$,\$\$\$V.99CR	\$100.00CR
-100000	\$\$,\$\$\$V.99CR	\$1,000.00CR
100000	\$\$,\$\$\$V.99S	\$1,000.00+
-10000	\$\$,\$\$\$V.99S	\$100.00-
10203	YYYYY	1 <del>02</del> <del>03</del>
070476	Z9/Y9/99	7/04/76
070476	Z9/99/99	7/04/76

NOTE: An all zero field has no significant positions.

Also see CURRENCY.

## CURRENCY

Purpose: To change the currency symbol for subsequent use by the PICTURE statement.

Format: CURRENCY(B)

Description: The contents of field B are used as the new currency symbol. The currency symbol may be altered as many times within a program as desired and may consist of up to 3 characters.

Example: CURRENCY(' \* ')

This will cause an asterisk to be substituted for '\$' until restored or changed.

CURRENCY(' \$ ')

This will cause the currency symbol to be restored to '\$'.

CURRENCY(' DOL ')

This will cause the letters "DOL" to assume the leftmost edit positions previously occupied by '\$'.

Also See PICTURE.

## NUMBER EDIT

Purpose: To create a decimal field from a field containing edit and decimal characters.

Format: A = NUMBER(B)

Description: From right-to-left, each position of field B is examined to determine if it is an edit character or a decimal digit. Edit characters are discarded but are used to determine the arithmetic sign. Decimal digits are moved to field A. If examination of field B terminates before field A is full, the remainder of the field is filled with zeros.

Example: FLDA = NUMBER(FLDB)

**BEFORE EXECUTION**

FLDB = ~~\$\$\$~~34.53

FLDA = XXXXXX

**AFTER EXECUTION**

FLDB = ~~\$\$\$~~34.53

FLDA = 003453

The sign of the resulting field A is adjusted to correspond with the net sign indicated by all edit characters. Interpretation of characters is as follows:

<u>Character</u>	<u>Action</u>
0-9	Sign unchanged, digit moved
+ . , / * <del>\$\$\$</del>	Sign unchanged, discarded
Others (including -)	Negative result, discarded

**NOTE:** Field B must contain only positive digits and edit characters.

## TRANSLATE:

Purpose: To translate data from one code set to another using a translate table.

Format: A = TRANSLATE(B,C,EFF:sn)

Description: From left-to-right, each byte of field B translated per field C and moved to the corresponding position in field A.

Field C is a translate table.

As each byte (character) is moved, the character is used as a binary offset into field C to select the contents of that position as the translated value.

If field A is shorter than field B, the operation terminates when A is full. If field A is longer than field B, the unused bytes of field A are not modified.

Example: FLDA = TRANSLATE(FLDB,FLDC,ERR:sn)

### **BEFORE EXECUTION**

FLDB (Hex Code) = 03 02 01 04

FLDC (Graphic Code) = \*ABCD

FLDA (Graphic Code) = XXXX

### **AFTER EXECUTION**

FLDB (Hex Code) = 03 02 01 04

FLDC (Graphic Code) = \*ABCD

FLDA (Graphic Code) = CBAD

**NOTE:** If the range of field C is exceeded by a particular value in field B, the error indicator is set; however, the translate process is not terminated.

## COMPRESS

Purpose: To convert a record into a compressed record.

Format: A = COMPRESS (B,C,ERR:sn)

Description: Compression is used to achieve improved recording characteristics for data on magnetic media. The improvement is derived from the reduced number of positions required by the compressed record and from the use of all positions of the physical records within the compressed file.

Field A is the buffer receiving the compressed characters. When filled with data, field A should be written to the compression file.

Field B is the record which is to be compressed.

Field C is the data control block (DCB) used to indicate when field A is full and when field B is fully compressed.

When the DCB indicates that field A is full by returning an External State Code (ESC) of X'01', the program should take the following steps:

1. Write the compression buffer (field A) to the compression file.
2. Execute the COMPRESS statement again so that the compression of the record (field B) may continue.

When the DCB indicates that field B is fully compressed by returning an External State Code (ESC) of X'00', the program should take the following steps:

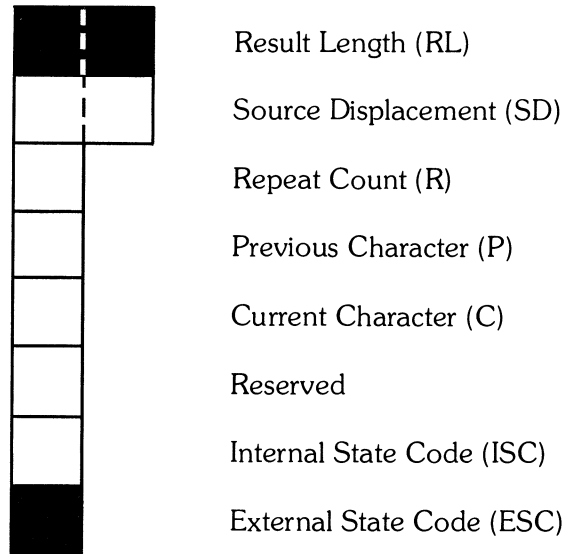
1. Obtain the next record (field B) which is to be compressed.
2. Execute the COMPRESS statement again so that compression of the new record may begin.

Field A and field B are each limited to 256 positions.

Field C (DCB) must be exactly 10 positions. The DCB must be initialized to all zeros (either binary or decimal) before the compression activity and should not be subsequently reset or altered for the duration of the activity.



The internal organization of the DCB is as follows:



The Result Length and External State Code subfields are significant to the application as described below. All others are for exclusive use of the COMPRESS operation.

The DCB indicates via the External State Code (ESC) field whether field B is fully compressed (ESC = X'00') or if field A is full prior to the end of the input segment (ESC = '01'). When ESC is X'00' and there is no "next record," the compression activity should be terminated. The program may accomplish this with the following steps:

1. Null-fill the unused portion of the compression buffer (field A).
2. Write the compression buffer to the compression file.
3. CLOSE the compression file.

The unused portion of field A to be null-filled can be determined from the DCB's Result Length (RL) field which designates the number of positions currently in use.

Null-filling provides for properly synchronized end-of-segment detection during decompression and simplifies later extension of the file, if necessary.

Compression involves left-to-right processing of field B as shown in the table below. The "FIELD B DATA" column enumerates the current situation in regards to the sequence of characters to be compressed. The first applicable entry in the column is the one which is used. The "Process" section enumerates the one or two values. (Repeat Value/Data Value) added to the compression buffer and the next compression step taken.

**Table 6-4: COMPRESSION PROCESS**

FIELD B DATA	PROCESS		
	Repeat Value	Data Value	Next Step
15 or more repeated characters in sequence	15 (X'0F')	Repeated Character	Continue scanning
2-14 repeated characters in sequence	Repetition Count (X'02'–X'0E')	Repeated Character	Continue scanning
1 character in the range X'00' to X'0F'	1 (X'01')	Data Character	Continue scanning
1 character in the range X'10' to X'FF'	Not applicable	Data Character	Continue scanning
No characters remaining	0 (X'00')	Not applicable	Return to program with the External State Code = X'00'

The ERR:sn branch is taken if the length of field C is incorrect or its contents are determined to be invalid (e.g., the Data Control Block (DCB) is not properly initialized to nulls (X'00') before execution of COMPRESS).

Example 1:

Contents of B before and after execution of COMPRESS

F0	F0	F0	F0	F2	F1	40	40	40	40	F0	F0	F0	F0	F6	F3	40	40	40	F0	F9
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

Contents of A after Execution of COMPRESS

04	F0	F2	F1	04	40	03	F0	F6	F3	03	40	F0	F9	00
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

**NOTE:**

Example 1 illustrates the compression of only one string.

Example 2:

```
RCD: B
      -(133);
RCD: A
      -(128);
RCD: DCB
      DCBLEN(0,2)
      -(0,10);
EQU: EQ;
      .
      .
      .
      START
      .
      .
      .
      DCB=:00:
5,    READ(-,B,EOF:30)
10,   A=COMPRESS(B,DCB)
      GO(DCB)5,20
20,   WRITE(-,A,-)
      GO:10
30,   EQ=STRING(A,0,DCBLEN,ERR:40)
      A=EQ:00:
      WRITE(-,A,-)
40,   CLOSE(-)
      .
      .
      .
      END
```

Example 2, which is a portion of an entire application, illustrates how the COMPRESS statement is used with other statements to compress an entire file.

Also see DECOMPRESS.

## DECOMPRESS

Purpose: To convert a compressed record into its original expanded representation.

Format: A = DECOMPRESS (B, C, ERR: sn)

Description: Field A is the buffer to receive the expansion of field B. When it contains a full segment of data, field A should be written to the decompression file.

Field B is the compressed record.

Field C is the Data Control Block (DCB) used to indicate when field A is full or when field B is completely decompressed. The internal organization of the DCB is shown under COMPRESS.

The ERR:sn branch is taken if the length of field C is incorrect or its contents are determined to be invalid (i.e., the Data Control Block (DCB) is not properly initialized to nulls (X'00') before execution of DECOMPRESS).

When the DCB indicates that a full segment has been expanded into field A, an External State Code of X'00' is returned, the program should take the following steps:

1. Write the record in the buffer (field A) to the decompression file. The record within field A occupies the first number of positions indicated by the RL in the DCB. (See the internal organization of the DCB.)
2. Execute the DECOMPRESS statement again so that expansion of another segment may begin.

When the DCB indicates that field B is fully decompressed by returning an External State Code of X'01', the program should take the following steps:

1. Obtain the next compressed record (field B) to be expanded.
2. Execute the DECOMPRESS statement again so that expansion of the compressed segment may continue.

When the DCB indicates that field A is full by returning an External State Code of X'02', the program has not provided a buffer of sufficient length to hold an entire decompressed segment. Programmer options are:

1. Change the source program to provide for a buffer of greater length.
2. Dispose of the current contents of field A and execute the DECOMPRESS statement again to obtain the remainder of the segment.

Field A and field B are each limited to 256 positions.

Field C (DCB) must be exactly 10 positions. The DCB must be initialized to all zeros (either binary or decimal) before the decompression activity and should not be subsequently reset or altered for the duration of the activity.

Tables ONE and TWO are used to outline character scanning during the decompression activity. Characters are scanned one or two at a time. The first character scanned is referred to as R; the second character simultaneously scanned (if applicable) is referred to as D.

When DECOMPRESS is executed and the External State Code is X'00', TABLE ONE applies. Otherwise the last active table is used.

**TABLE ONE**

Field B Value of R	Next Step
X'00'	Continue scanning with Table ONE
X'01' – X'FF'	Process according to Table TWO

**TABLE TWO**

Field B Value of R	Data Decompression	Next Step
X'00'	Not Applicable	Return to program with ESC=X'00'
X'01' – X'0F'	Produce R copies of character D	Continue scanning with Table TWO
X'10' – X'FF'	Produce one copy of character R	Continue scanning with Table TWO

Example:

```
RCD:  A
      -(133);
RCD:  B
      -(128);
RCD:  DCB
      DCBLEN(0,2)
      -(0,10);
EQU:  EQ;
      .
      .
      .
START
      .
      .
      .
      DCB=:00:
5,    READ(-,B,-)
10   A=DECOMPRESS(B,DCB)
      GO(DCB)20,5,30
20,  EQ=STRING(A,0,DCBLEN)
      PRINT(-,EQ,-)
      GO:10
30,  NOTIFY('DATA ATTRIBUTE(AE)',ANS)
      (ERROR RECOVERY)
      .
      .
      .
      END
```

The example above which is a portion of an entire application, illustrates how the DECOMPRESS statement is used with other statements to decompress an entire file.

## HEX

Purpose: To convert binary data into graphic hexadecimal data (EBCDIC 0-9, A-F).

Format: A = HEX(B)

Description: From right-to-left, each byte of field B is converted into two bytes of hexadecimal data and stored in field A; therefore, field A should be twice the size of field B.

The operation terminates when field A is full. If field A is more than twice as long as field B, the unused leftmost positions of field A are filled with EBCDIC zeros.

Example: FLDA = HEX(B)

### **BEFORE EXECUTION**

FLDB (Binary Code) = 1000 0100 0100 1111

FLDA (Graphic Code) = 'XXXX'

### **AFTER EXECUTION**

FLDB (Binary Code) = 1000 0100 0100 1111

FLDA (Graphic Code) = '844F'

## UNHEX

Purpose: To convert graphic hexadecimal data (EBCDIC 0–9, A–F) into binary data.

Format: A = UNHEX(B)

Description: From right-to-left, each successive pair of bytes from field B converted into one byte of binary data and stored in field A; therefore, field A need be only half as long as field B.

The operation terminates when field A is full. If field A is longer than required, the unused leftmost positions of field A are filled with binary zeros.

If field B contains an odd number of hexadecimal digits, a value of decimal zero is assumed to complete the leftmost hexadecimal pair.

If a character other than a hexadecimal digit is encountered in field B, a value of decimal zero is assumed for that character.

The error indicator is affected by this instruction but no significance can be associated with its setting.

### Example 1:

#### **BEFORE EXECUTION**

FLDB (Graphic Code) = '144F'

FLDA (Binary Code) = 0000 0000 0000 0000

#### **AFTER EXECUTION**

FLDB (Graphic Code) = '844F'

FLDA (Binary Code) = 0000 0000 0000 0000

### Example 2:

#### **BEFORE EXECUTION**

FLDB (Graphic Code) = '44F'

FLDA (Binary Code) = 0000 0100 0100 1111

#### **AFTER EXECUTION**

FLDB (Graphic Code) = '844F'

FLDA (Binary Code) = 1000 0100 0100 1111



## CONTROL STATEMENTS

Control statements control the flow of program execution, either conditionally or unconditionally. The Control statements are:

UNCONDITIONAL GO

STOP

CONDITIONALS

COMPUTED GO

CASE

ERROR TEST

## UNCONDITIONAL GO

Purpose: To branch unconditionally to a specified statement.

Format: GO: sn

Description: The GO statement unconditionally transfers control to the statement specified by sn.

Example:

```
      GO:25
22, A+B*C
      .
      .
      .
25, STOP
      END
```

If the statement immediately following a GO is not an END statement, then that statement must have a statement number so that it can be reached (via a branch or PERFORM statement) during program execution.

## STOP

Purpose: To terminate program execution.

Format: STOP

Description: The stop statement terminates program execution. It may appear anywhere in the Execution Section prior to the END statement. Although several STOP statements may appear in the program, only one STOP can be executed per program execution.

When a STOP statement is executed, all open logical files are closed. If an error condition occurs during closing of these files, the condition is not reported to the program.

Example: STOP

If the statement immediately following STOP is not an END statement, it must have a statement number so that it can be reached (via a branch or PERFORM statement) during program execution.

## CONDITIONALS

Purpose: To alter the flow of execution based upon a specified condition of program data.

Format: A conditional statement is composed of two parts: a conditional phrase and an executable statement. The test specified in the conditional phrase determines whether or not the executable statement is executed.

1. IF (condition) statement
2. IFNOT (condition) statement

Description:

1. When the specified condition is true, statement is executed; otherwise, the next in-line statement is executed.
2. When the specified condition is false, statement is executed; otherwise, the next in-line statement is executed.

The executable statement must not be an ENTRY statement.

Example 1: IF (condition) GO: 20

When condition is true, the GO statement is executed; otherwise, the next in-line statement is executed.

Example 2: IFNOT (condition) A = :\*:

When the condition is false, the FILL statement is executed; otherwise, the next in-line statement is executed.

Unless the executable statement affects program flow, the next in-line statement will also be executed.

NOTE: Since a conditional statement is also an executable statement, the conditional statement may be compounded. For example,

IF (condition-B) IFNOT (condition-C) GO: 20

When condition-B is true *and* condition-C is false, the GO statement is executed, otherwise, the next in-line statement is executed.

## CONDITIONAL PHRASES: ALPHANUMERIC

Purpose: To compare one alphanumeric field to another alphanumeric field or to a specified alphanumeric literal character.

Format:

1. (B, MATCHES, C)
2. (B, MATCHES, :X:)
3. (B, CONTAINS, :X:)
4. (B, CONTAINS, C)

Description:

1. From left to right, each position of field B is compared to the corresponding position of field C for as many positions as there are in the shorter of the two fields. If each position pair contains the same character value or either position contains the null character, the position pair matches. If every position pair matches, the condition is true; otherwise, false.
2. Each position of field B is compared to the specified literal character and to the null character. If each position of field B contains either the specified literal character, or the null character, the condition is true; otherwise, false.
3. Each position of field B is compared to the specified literal character. If at least one position of field B matches the specified literal character, the condition is true; otherwise, false.
4. Each position of field B is compared to the first position of field C. If at least one position of field B matches the character contained in the first position of field C, the condition is true.

Example 1: IF (FLDB, MATCHES , FLDC) GO: 10 RESULT

A	A	nl	A	B	A	TRUE
A	nl	A	B	A	B	TRUE
A	nl	A	A	nl	B	FALSE

Example 2: IF (FLDB, MATCHES , :A:) GO: 10 RESULT

A	nl	A	A	TRUE
A	A	A	TRUE	
A	nl	B	A	FALSE

Example 3: IF (FLDB, CONTAINS , :A:) GO: 10 RESULT

A	A	A	A	TRUE
B	B	B	A	TRUE
B	C	D	A	FALSE

Example 4: IF (FLDB, CONTAINS , FLDC) GO: 10 RESULT

A	nl	A	A	B	B	TRUE
A	B	C	C	B	A	TRUE
A	nl	A	C	B	A	FALSE

Where nl = null

The logical not character (  $\neg$  ) may prefix an alphanumeric comparator to designate inversion of the true/false condition. For example,  $\neg$  CONTAINS designates a true condition when the corresponding CONTAINS condition is false. Thus, the two samples below are functionally equivalent:

## CONDITIONAL PHRASES: BINARY

Purpose: To compare two binary fields.

Format:

1. (B, EQ, C)	4. (B, NE, C)
2. (B, LT, C)	5. (B, LE, C)
3. (B, GT, C)	6. (B, GE, C)

Description: The binary contents of field B are compared to the binary contents of field C. The condition is true for the following cases:

1. Field B equals field C;
2. Field B is less than field C;
3. Field B is greater than field C;
4. Field B is unequal to field C;
5. Field B is less than or equal to field C;
6. Field B is greater than or equal to field C.

In all cases, if one field is longer than another, the shorter field is regarded as filled with sufficient leading nulls to achieve equal length.

<u>Example:</u>		<u>RESULT</u>
	IF (FLDB, EQ, FLDC) GO: 10	
	1010 1111                      1010 1111	TRUE
0000 0000	1111 0000                      0000 1111 0000	TRUE
	1010 1111                      0000 0000	FALSE

The logical not character (7) may prefix a binary comparator to designate inversion of the true/false condition.

For example, 7EQ designates a true condition when the corresponding EQ condition is false. Thus the two samples below are functionally equivalent:

IF (FLDB, 7EQ, FLDC) GO: 10

IFNOT (FLDB, EQ, FLDC) GO: 10

NOTE: The leftmost bit of a binary field is not regarded as a sign bit.

## CONDITIONAL PHRASES: BIT

Purpose: To compare bit settings.

Format: 1. (B,HASBITS,C)  
2. (B,HASBITS,:X:)

Description: The leftmost byte of field B is compared, using the logical AND, with either the leftmost byte of field C (format 1) or the literal character X (format 2). If the result is not equal to zero, the condition is true.

<u>Example:</u>	IF (FLDB, HASBITS, FLDC) GO: 10	<u>RESULT</u>
	1000 0000            1100 0000	TRUE
	0100 0000            1100 0000	TRUE
	0010 0000            1000 0000	FALSE

The logical not character (7) may prefix a bit comparator to designate inversion of the true/false condition. For example, 7HASBITS designates a true condition when the corresponding HASBITS condition is false. Thus, the two samples below are functionally equivalent:

IF (FLDB, 7HASBITS, :80:) GO: 10

IFNOT (FLDB, HASBITS, :80:) GO: 10



## CONDITIONAL PHRASES: DECIMAL

Purpose: To compare two decimal fields.

Format:

1. (B = C)
2. (B < C)
3. (B > C)
4. (B <=C)
5. (B >=C)

Description: The decimal contents of field B are compared to the decimal contents of field C. The condition is true for the following cases:

1. Field B equals field C;
2. Field B is less than field C;
3. Field B is greater than field C;
4. Field B is less than or equal to field C;
5. Field B is greater than or equal to field C.

In all cases, if one field is longer than the other, the shorter field is regarded as filled with sufficient leading zeros to achieve equal length.

Example 1:

IF (FLDB = FLDC) GO:10	<u>RESULT</u>								
<table style="display: inline-table; border-collapse: collapse; margin-right: 20px;"> <tr><td style="border: 1px solid black; padding: 2px 5px;">0</td><td style="border: 1px solid black; padding: 2px 5px;">0</td><td style="border: 1px solid black; padding: 2px 5px;">1</td><td style="border: 1px solid black; padding: 2px 5px;">2</td><td style="border: 1px solid black; padding: 2px 5px;">5</td></tr> </table> <table style="display: inline-table; border-collapse: collapse;"> <tr><td style="border: 1px solid black; padding: 2px 5px;">1</td><td style="border: 1px solid black; padding: 2px 5px;">2</td><td style="border: 1px solid black; padding: 2px 5px;">5</td></tr> </table>	0	0	1	2	5	1	2	5	TRUE
0	0	1	2	5					
1	2	5							
<table style="display: inline-table; border-collapse: collapse; margin-right: 20px;"> <tr><td style="border: 1px solid black; padding: 2px 5px;">0</td><td style="border: 1px solid black; padding: 2px 5px;">0</td><td style="border: 1px solid black; padding: 2px 5px;">1</td><td style="border: 1px solid black; padding: 2px 5px;">2</td><td style="border: 1px solid black; padding: 2px 5px;">5̄</td></tr> </table> <table style="display: inline-table; border-collapse: collapse;"> <tr><td style="border: 1px solid black; padding: 2px 5px;">1</td><td style="border: 1px solid black; padding: 2px 5px;">2</td><td style="border: 1px solid black; padding: 2px 5px;">5̄</td></tr> </table>	0	0	1	2	5̄	1	2	5̄	TRUE
0	0	1	2	5̄					
1	2	5̄							
<table style="display: inline-table; border-collapse: collapse; margin-right: 20px;"> <tr><td style="border: 1px solid black; padding: 2px 5px;">0</td><td style="border: 1px solid black; padding: 2px 5px;">1</td><td style="border: 1px solid black; padding: 2px 5px;">2</td><td style="border: 1px solid black; padding: 2px 5px;">5</td></tr> </table> <table style="display: inline-table; border-collapse: collapse;"> <tr><td style="border: 1px solid black; padding: 2px 5px;">0</td><td style="border: 1px solid black; padding: 2px 5px;">2</td><td style="border: 1px solid black; padding: 2px 5px;">5</td><td style="border: 1px solid black; padding: 2px 5px;">0</td></tr> </table>	0	1	2	5	0	2	5	0	FALSE
0	1	2	5						
0	2	5	0						

Example 2:

IF (FLDB < FLDC) GO:10	<u>RESULT</u>					
<table style="display: inline-table; border-collapse: collapse; margin-right: 20px;"> <tr><td style="border: 1px solid black; padding: 2px 5px;">0</td><td style="border: 1px solid black; padding: 2px 5px;">5</td></tr> </table> <table style="display: inline-table; border-collapse: collapse;"> <tr><td style="border: 1px solid black; padding: 2px 5px;">1</td><td style="border: 1px solid black; padding: 2px 5px;">0</td></tr> </table>	0	5	1	0	TRUE	
0	5					
1	0					
<table style="display: inline-table; border-collapse: collapse; margin-right: 20px;"> <tr><td style="border: 1px solid black; padding: 2px 5px;">0</td><td style="border: 1px solid black; padding: 2px 5px;">2</td><td style="border: 1px solid black; padding: 2px 5px;">0̄</td></tr> </table> <table style="display: inline-table; border-collapse: collapse;"> <tr><td style="border: 1px solid black; padding: 2px 5px;">1</td><td style="border: 1px solid black; padding: 2px 5px;">0</td></tr> </table>	0	2	0̄	1	0	TRUE
0	2	0̄				
1	0					
<table style="display: inline-table; border-collapse: collapse; margin-right: 20px;"> <tr><td style="border: 1px solid black; padding: 2px 5px;">1</td></tr> </table> <table style="display: inline-table; border-collapse: collapse;"> <tr><td style="border: 1px solid black; padding: 2px 5px;">2</td><td style="border: 1px solid black; padding: 2px 5px;">0</td><td style="border: 1px solid black; padding: 2px 5px;">0̄</td></tr> </table>	1	2	0	0̄	FALSE	
1						
2	0	0̄				

Example 3:

IF (FLDB > FLDC) GO: 10					<u>RESULT</u>
0	2	0 <sup>+</sup>	1	0 <sup>+</sup>	TRUE
1̄	9	9	9̄		TRUE
1̄	9	9	9 <sup>+</sup>		FALSE

NOTE:

The values in the compared fields may be positive or negative. The contents of the B or C fields cannot be negated during the decimal compare operation.

The logical not character (⌈) may prefix a decimal comparator to designate inversion of the true/false condition. For example ⌈= designates a true condition when the corresponding = condition is false. Thus, the two samples below are functionally equivalent:

IF (FLDB⌈=FLDC) GO: 10

IFNOT (FLDB=FLDC) GO: 10

## CONDITIONAL PHRASES: SORT

Purpose: To compare two alphanumeric fields for collating sequence.

Format:

1. (B, EQUALS, C)
2. (B, BEFORE, C)
3. (B, AFTER, C)

If one field is longer than the other in the above formats, the shorter field is regarded as filled with sufficient trailing nulls to achieve equal length.

Description:

1. From left-to-right, each position of field B is compared to the corresponding position of field C for as many positions as there are in the longer of the two fields. If each pair contains the same character value, the condition is true.
2. From left-to-right, each position of field B is compared to the corresponding position of field C for as many positions as there are in the longer of the two fields or until a mis-match occurs. If a mis-match occurs and the character from field B is before the character from field C in the EBCDIC collating sequence, the condition is true.
3. From left-to-right, each position of field B is compared to the corresponding position of field C for as many positions as there are in the longer of the two fields, or until a mis-match occurs. If the character from field B is after the character from field C in the EBCDIC collating sequence, the condition is true.

Example 1: IF (FLDB, EQUALS, FLDC) GO: 10      RESULT

A	B	A	B	nl	TRUE
A	C	nl	A	C	TRUE
nl	A	A	A		FALSE

Example 2: IF (FLDB, BEFORE, FLDC) GO: 10      RESULT

A	A	A	B	B	B	TRUE
0	0	1	2	2	3	TRUE
B	C	D	*	5	6	FALSE

Where nl = null

<u>Example 3:</u>	IF (FLDB, AFTER, FLDC) GO: 10	<u>RESULT</u>
B   B   B	A   A   A	TRUE
3   0   1	2   0   1	TRUE
B   B   B	B   B   B	FALSE

The logical not character ( $\neg$ ) may prefix a sort comparator to designate inversion of the true/false condition. For example,  $\neg$ BEFORE designates a true condition when the corresponding BEFORE condition is false. Thus, the two samples below are functionally equivalent:

IF (FLDB,  $\neg$ BEFORE, FLDC) GO: 10

IFNOT (FLDB, BEFORE, FLDC) GO: 10

## COMPUTED GO

Purpose: To branch, based upon a value to a statement specified in a list of statement numbers.

Format: GO(B) sn-list

Description: The low order byte of field B determines which statement from the statement number list is to be branched to. If the low order byte contains zero (0), then the first statement number from the list is selected. If the low order byte contains 1, then the second statement number from the list is selected, and so on.

Only the rightmost four bits of field B are examined. All other bits of field B are ignored. Therefore, the values X'00', X'10', ..., X'F0' would each cause the first statement number in the list to be selected.

Example: GO(FLDB) 10,20,10

If the rightmost four bits of field B contain a value greater than that implied by the length of the statement number list, the next in-line statement is executed.

NOTE: Branches within the sn-list may not be omitted (for example: GO(FLDB) 10,,10 is considered invalid).

## CASE

Purpose:

To branch to a statement specified in a list of statements.

Format:

CASE(B) condition-1:sn-1, condition-2:sn-2, ... , condition-n:sn-n

Description:

The rightmost position of field B determines the statement number to which it is to branch. The coding of each condition includes a range of values for field B associated with the statement to be branched to if the condition is met.

A condition may be expressed as either an alphanumeric or hexadecimal string.

If a condition describes a single character, then that character must be matched exactly by the rightmost position of field B.

If the condition describes more than one character, then the rightmost position of field B must be greater than or equal to the leftmost condition character, and less than or equal to the rightmost condition character in order for the condition to be satisfied.

If more than two characters occur in the condition, only the first and last are used as described above.

The EBCDIC collating sequence constitutes the basis for the compare.

If multiple conditions are satisfied by field B, the first satisfied condition determines the branch.

Example:

CASE (FLDB) 'AF':10, '09':20

If the rightmost position of FLDB contains either A, B, C, D, E, or F, control is passed to the statement labeled 10. If FLDB contains a digit (0 through 9), control is passed to the statement labeled 20. For all other values, control is passed to the next in-line statement.

## ERROR TEST

Purpose: To branch to a specified statement on an error condition.

Format: ERR: sn

Description: ERR transfers control to the statement number specified when the error indicator is set. If the error indicator is not set, the next in-line statement is executed.

Example:

```
FLDA=FLDB+FLDC
ERR: 25
.
.
.
25, STOP
```

The ERR statement should immediately follow the statement being monitored.

## SUBROUTINE STATEMENTS

Subroutine statements delimit and direct execution of a subroutine. A subroutine is a group of statements performing a specific function that may be performed as often as necessary. The subroutine is coded only once, eliminating repetitive statements. It may be accessed from anywhere within the Execution Section of the program. The Subroutine statements are:

PERFORM

ENTRY

EXIT



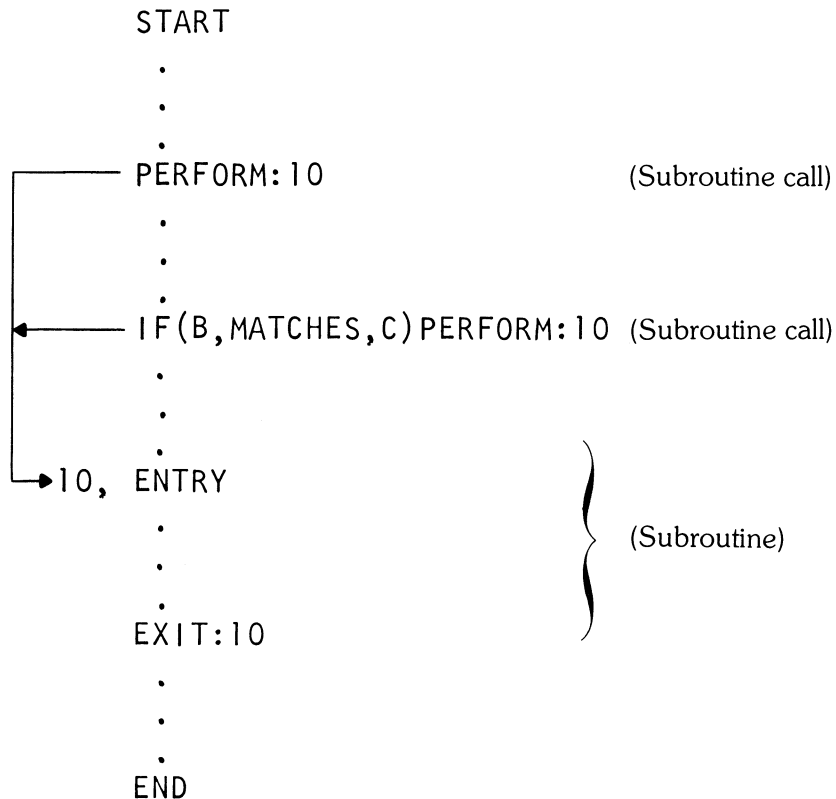
## PERFORM

Purpose: To invoke execution of a subroutine.

Format: PERFORM: sn

Description: To transfer control to a subroutine ENTRY specified by sn. Also, PERFORM causes the return location of the next statement in-line with the PERFORM to be saved for later continuation of the current program sequence.

Example:



Since subroutines can be referenced by other coding within the main programming module and since all fields of the main module are available to subroutines, they are usually reserved for calculations/operations which are frequently used, tedious to repeat, or subject to change. A significant reduction in both source program size and object program size can be realized with this technique for program structuring.

The PERFORM, ENTRY and EXIT statements provide a means of logically structuring the application program; therefore, branching to a statement within the subroutine is not recommended.

A subroutine should not contain a call to itself. However, if a call of this type is coded, a logical exit must be provided within the subroutine, such as a conditional or unconditional branch statement. A logical means of leaving such a subroutine is required, since the saved address of the first PERFORM statement is lost upon execution of the second PERFORM statement.

## ENTRY

Purpose: To designate the beginning of a subroutine.

Format: sn, ENTRY

Description: The ENTRY statement designates the beginning of the sequence of execution statements (subroutines) and establishes the ENTRY statement number.

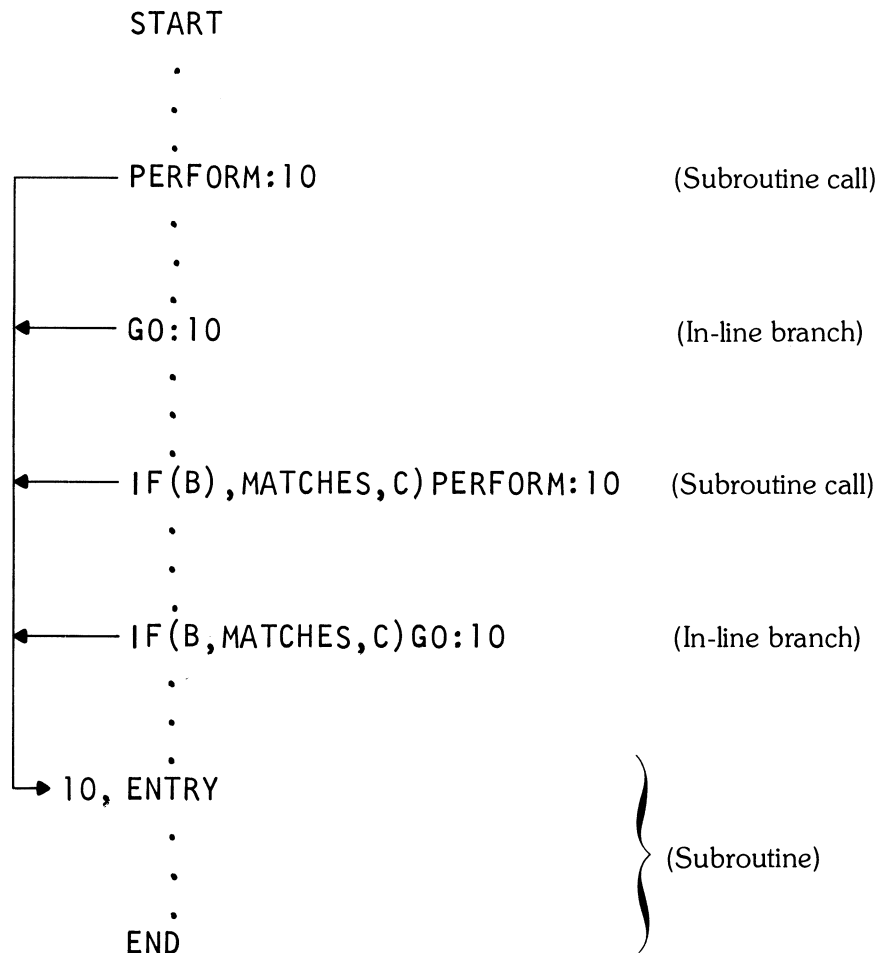
The subroutine is normally called through the PERFORM statement. However, the ENTRY statement can be reached logically as the next statement in sequence or as the result of a branch statement. When reached logically as an in-line statement, execution proceeds with the first statement following the ENTRY statement.

Since the subroutine is internal to the calling program, all fields within the calling program are common to the subroutine and may be used by statements within the subroutine.

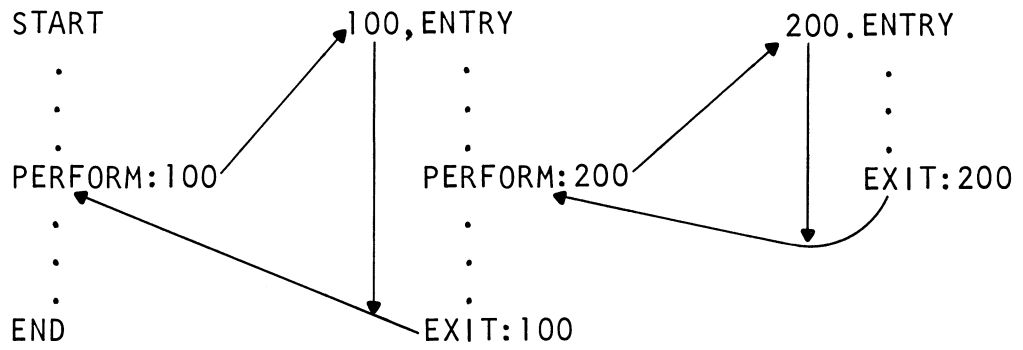
No data definition statements may appear within the subroutine. All data definition statements required for subroutine data must be included in the Data Definition Section of the calling program.

If multiple statement numbers are assigned to ENTRY, only the last may be referenced by PERFORM/EXIT.

Example:



Nesting of subroutine calls is permitted as long as no subroutine has two outstanding performs against it. For example:



Also see EXIT and PERFORM.

## EXIT

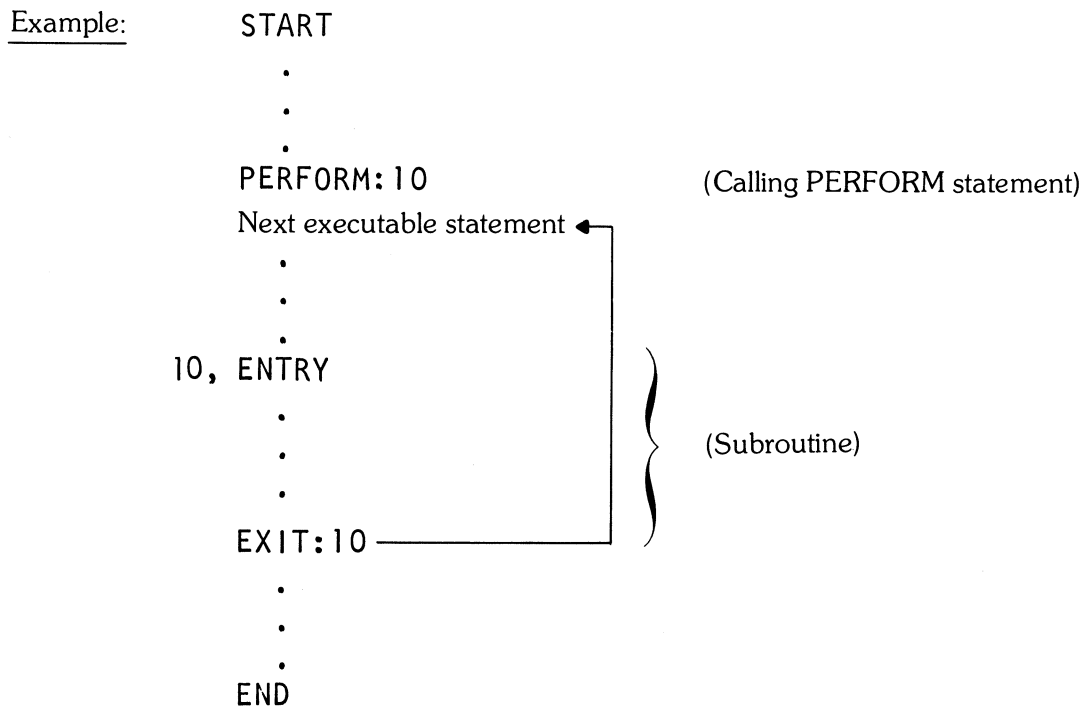
Purpose: To terminate execution of the subroutine by returning control to the calling program.

Format: EXIT:sn

Description: An EXIT statement returns control to the return location saved by the initiating PERFORM statement.

EXIT does not alter the return information established by PERFORM. If a PERFORM has been previously executed, control passes to the statement immediately following that PERFORM. However, if a PERFORM has never been executed, control passes to the statement following the EXIT.

An exit can be made from a PERFORM'ed sequence to a specified statement anywhere in the program by means of a branch statement. A labeled statement within a PERFORM'ed sequence can be the object of a branch from anywhere in the program.



Also see ENTRY and PERFORM.

## I/O STATEMENTS

I/O statements direct Input/Output operations. I/O operations transfer data between the application program in main memory and the I/O device. Also, I/O statements control the I/O device. The I/O statements are:

OPEN

CLOSE

READ

WRITE

CHECKEOD

DELETE

FREESPACE

INSERT

READLOCK

READNEXT

RELEASE

SETEOD

BACKSPACE

MARK

REWIND

REWINDLOCK

SKIPFILE

CHECKFORMS

PRINT

SETFORMS

SENDEOF

## OPEN

Purpose: To gain access to an I/O device.

Format: OPEN (IOD-name, buffer, ERR:sn)

Description: The dataset represented by the IOD is connected to the corresponding device and is made available for use. For DISKETTE, the dataset may be a named dataset or it may be the whole volume.

For all devices but DISK and DISKETTE, the buffer and its preceding comma may be omitted.

If an error condition occurs, the ERR exit is taken. If no ERR exit is coded, execution of the program is cancelled.

Example: OPEN(PAYROLFILE, BUFFER, ERR:200)

## CLOSE

Purpose: To terminate access to an OPEN'ed dataset.

Format: CLOSE ( IOD-name, buffer, ERR: sn)

Description: CLOSE releases the logical file defined by the IOD and all of the underlying resources. If the IOD is not OPEN, the operation is null.

For all devices but DISK and DISKETTE, the buffer and its preceding comma may be omitted.

If error occurs during the CLOSE operation, control is transferred to the statement labeled sn. Even in this case the logical file is no longer open.

Example: CLOSE (PAYROLFILE, BUFFER, ERR: 20)

NOTE: When a STOP statement is executed, all open logical files are closed. See the discussion of STOP in this section.

**READ (applies to all devices, except PRINTER)**

Purpose: To transfer a data record from a dataset (or the host processor).

Format: READ ( IOD-name, buffer, ERR:sn-1, EOF:sn-2)

Description: A data record is transferred by the device specified in the IOD.

The ERR exit is taken if an error condition results. If no ERR exit is coded, execution of the program is cancelled.

The EOF exit is taken if an end-of-file condition results. If no EOF exit is coded, the end-of-file condition is regarded as an error condition and handled as described in the paragraph above.

Example: READ (PAYROLFILE, BUFFER, ERR:200, EOF:201)



## WRITE

Purpose: To transfer a data record to a dataset (or the host processor).

Format: WRITE (IOD-name, buffer, ERR:sn-1, EOF:sn-2)

Description: A data record is transferred to the device specified in the IOD.

The ERR exit is taken if an error condition results. If no ERR exit is coded, execution of the program is cancelled.

The EOM exit is taken if an end-of-medium condition results. If no EOM exit is coded, the end-of-medium condition is regarded as an error condition and handled as described in the paragraph above.

Example: WRITE (PAYROLFILE, BUFFER, ERR:101, EOM:102)

## CHECKEOD

Purpose: To lock position of READ or WRITE operations at EOD (next record-space after the last record in the dataset) and to prevent other users from accessing that record-space.

Format: CHECKEOD ( IOD-name , ERR : sn )

Description: CHECKEOD positions the file to prepare for writing at EOD (i.e., to append records). The location of EOD is locked in order to serialize the use of EOD and avoid a conflict between two or more users attempting to write a new "last record."

The lock may be removed by RELEASE, WRITE, or CLOSE. One record at a time may be locked for an IOD.

The POSITION keyword\* is updated to contain the system-maintained EOD.

If the record located at EOD cannot be reserved, control is transferred to the statement labeled sn in the ERR clause. If no ERR clause is coded, program execution is cancelled.

\* See section 7 for a detailed discussion of the POSITION Keyword.

Example: CHECKEOD ( PAYROLFILE , ERR : 102 )

## DELETE

Purpose: To remove an entry from an index dataset.

Format: DELETE (IOD-name, ERR:sn)

Description: The index entry for the most recently referenced record of the target dataset is removed from the index dataset represented by this IOD. The index IOD must have just been used to reference the target record. The target dataset remains unchanged.

DELETE is valid only for index datasets for which ACCESS=SEQ or ACCESS=RAN.

Example: IOD:DATAFILE = DISK/DISKETTE  
                  ...;  
IOD:INDEX1 = DISK/DISKETTE

          ACCESS = RANDOM

          KEYVALUE = FIELDNAME

          TARGET = DATAFILE

          ...;  
START

  .

  .

  .

          READ(INDEX1,...)

          DELETE(INDEX1,...)

The record is not deleted from the target dataset, only its associated index dataset entry is deleted.

## FREESPACE

Purpose: To free allocated space beyond EOD (end-of-data).

Format: `FREESPACE ( IOD-name,ERR;sn)`

Description: Any space beyond EOD for the dataset is returned to the free space chain for the volume. The entire allocated space may be returned to the free space chain by setting EOD=X'00 00', then executing FREESPACE.

If an error condition occurs, the ERR exit is taken. If no ERR exit is coded, program execution is cancelled.

Example: `FREESPACE (PAYROLFILE,ERR:102)`

## INSERT

Purpose: To cause a new entry to be included in an index dataset for an existing but not previously indexed record in the target dataset.

Format: INSERT ( IOD-name , ERR : sn )

Description: The key position information for the most recently referenced record of the target dataset is added to the index dataset. The target IOD and the buffer for the reference operation must have been left undisturbed.

Prior to INSERT, a READ operation must be executed against the target record to establish the target .POSITION and the record value. The INSERT statement causes the Random Index Access Method (RIAM) to hash the keyfield(s) located in the READ buffer, determine a POSITION for the index entry and WRITE that index entry to the index dataset.

If an index for the particular target record already exists in the index dataset, the error branch is taken. If the index dataset is full, the index record cannot be inserted and the ERR branch is taken. If the ERR exit is not coded, program execution is cancelled.

Example: IOD:DATAFILE = DISK/DISKETTE

...;

IOD:INDEX1 = DISK/DISKETTE

ACCESS = RANDOM

KEYVALUE = FIELDNAME

TARGET = DATAFILE

...;

START

.

.

.

CHECKEOD (DATAFILE, ...)

WRITE (DATAFILE, ...)

INSERT (INDEX1, ...)

The above statements add a new record to the target dataset DATAFILE and enable this record to be accessed by INDEX1.

NOTE: The operation of INSERT is valid only for index datasets for which ACCESS=RAN.

## READLOCK

Purpose: To transfer a data record from a device and prevent any other user from accessing that record.

Format: READLOCK (IOD-name, buffer, ERR: sn-1, EOF: sn-2)

Description: A READLOCK operation is similar to a READ operation except as noted below. The record described in the .POSITION keyword is locked to prevent read access through other IOD's. The .POSITION keyword is not incremented (in anticipation of a subsequent WRITE operation which updates the record and clears the lock).

One record at a time may be locked for an IOD.

The ERR exit is taken if an error condition results. If no ERR exit is coded, program execution is cancelled.

The EOF exit is taken if an end-of-file condition results.

If an EOF condition arises and no EOF clause is specified, the ERR exit is taken.

Example: READLOCK (PAYROLFILE, BUFFER, EOF: 101, ERR: 102)

## READNEXT

Purpose: To transfer the next data record, which has the same keyvalue as the record previously read, from the target dataset to memory.

Format: READNEXT ( IOD-name, buffer, ERR:sn-1, EOF:sn-2)

Description: When more than one record contains the same keyvalue, this statement may be used to acquire the second record (and subsequent records) via repeated executions. When the series of duplicated keyvalues is exhausted, the READNEXT operation takes the EOF:sn-2 branch.

The ERR exit is taken if an error condition results; control passes to the associated sn. If no ERR exit is coded, program execution is cancelled.

The EOF exit is taken if an end-of-file condition results. If an EOF condition arises and no EOF clause is specified, the ERR exit is taken.

This statement is valid only for index files when ACCESS=RAN.

Example: READ ( INDEX1, ... )

READNEXT ( INDEX1, ... )

## RELEASE

Purpose: To remove restricted access (without transferring data) on an external dataset record which has been the object of the READLOCK or CHECKEOD operation.

Format: RELEASE ( IOD-name , ERR : sn )

Description: This statement unlocks a previously locked record without performing a WRITE operation.

Example: RELEASE ( PAYROLFILE , ERR : 102 )

The POSITION keyword\* is examined to determine which record is to be unlocked. Upon successful completion of the RELEASE operation, CONTROL is returned to the next statement; the POSITION keyword is incremented by one.

If the RELEASE request is not honored, control is transferred to the statement associated with the statement number specified in the ERR clause and the POSITION keyword is not changed. If no ERR clause is coded, program execution is cancelled.

\* See Section 7 for a complete discussion of POSITION and other keywords.



## SETEOD

Purpose: To update the system-maintained EOD (end-of-data).

Format: SETEOD (IOD-name, ERR: sn)

Description: The EOD (end-of-data) for a dataset is reset to the value obtained from the POSITION keyword of the IOD.

The EOD parameter is recorded in two locations: in the system-maintained EOD and in the dataset label located in the VTOC. The system-maintained EOD is updated following a successful completion of a SETEOD operation. The system-maintained EOD is also updated upon completion of WRITE operations which extend the file. The dataset label of the VTOC is updated from the system-maintained EOD when the dataset is closed.

If the record designated by the POSITION keyword exceeds the extent of the dataset, an error condition occurs and control is transferred to the statement labeled sn. If no ERR exit is coded, program execution is cancelled.

Example: SETEOD (PAYROLFILE, ERR: 102)

NOTE: SETEOD can only be used with DISK/DISKETTE if ACCESS=EXCLUSIVE.

## BACKSPACE

Purpose: To move the tape back one record.

Format: BACKSPACE (IOD-name, ERR: sn)

Description: The magnetic tape is backed-up one record. The tape is left positioned in the inter-record gap preceding the "backed-over" record. This operation is canceled if no previous record is encountered.

Example: BACKSPACE (PAYROLFILE, ERR: 100)

## REWINDLOCK

Purpose: To cause the magnetic tape to be positioned immediately prior to load point.

Format: REWINDLOCK( IOD-name, ERR: sn)

Description: The magnetic tape is positioned ahead of the load point. The tape drive is switched off-line, effectively preventing subsequent use without operator intervention (i.e., the tape drive must be manually placed on-line prior to issuing further positioning or data transfer commands).

If an error condition occurs, the ERR exit is taken. If no ERR exit is coded, program execution is cancelled.

Example: REWINDLOCK(PAYROLFILE, ERR: 101)

## SKIPFILE

Purpose: To position a magnetic tape beyond the next encountered tape mark.

Format: SKIPFILE( IOD-name, ERR: sn)

Description: The magnetic tape is positioned within the inter-record gap following the next tape mark encountered in the forward direction.

The operation is cancelled and the ERR exit is taken in the event that no forward positioned tape mark is encountered (end-of-reel). If no ERR exit is coded, program execution is cancelled.

Example: SKIPFILE(PAYROLFILE, ERR:101)

## CHECKFORMS

Purpose: To obtain the current printer control and positioning information.

Format: CHECKFORMS ( IOD-name, ERR: sn )

Description: The current printer control information is returned to the FORMS and POSITION parameters.\*

Control passes to the next statement unless the CHECKFORMS request cannot be honored. In this case, control is transferred to the statement labeled sn in the ERR clause. If no ERR clause is coded, program execution is cancelled.

Example: CHECKFORMS ( LISTFILE, ERR: 103 )

\* See Section 7, for an explanation of the FORMS and POSITION keywords.

## PRINT

Purpose: To effect printing with forms control.

Format: PRINT( IOD-name, buffer, ERR: sn-1, EOF: sn-2)

Description: The paper is advanced according to the value contained in the SLEW\* keyword. Up to 132 bytes of data may be transferred to the printer.

Example: PRINT(PAYROLFILE, BUFFER, EOF: 110, ERR: 200)

The ERR exit is taken if an error condition results. If no ERR exit is coded, program execution is cancelled.

The ERR exit is taken if an end-of-form condition results. If no EOF exit is coded, the end-of-form condition is disregarded and the next in-line statement is executed.

\* See Section 7, for a detailed description of SLEW

## SETFORMS

Purpose: To alter active printer control parameters.

Format: SETFORMS ( IOD-name, ERR: sn)

Description: SETFORMS sets the POSITION keyword to zero (0), notifying the system that the forms are positioned to top-of-forms. Also, it transfers the values specified for FORMS (in the IOD) to the System Control Block. See the Initial Program Load values for the system control table in Section 7, under the FORMS keyword.

The FORMS parameter is a four-byte field:

byte 1 = lines per page

byte 2 = characters per inch — horizontal, range 1-15

byte 3 = lines per inch — vertical, range 1-48

byte 4 = lines per form

For the horizontal measurement, only 1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, and 60 are exact measurements since the unit of escapement is 1/60th of an inch. Any other number would be approximate to its nearest value.

For the vertical measurement, only 1, 2, 3, 4, 6, 8, 12, 24, and 48 lines per inch are exact measurements since the vertical forms ratchet is 1/48th of an inch. Any other number would be approximated. After execution of SETFORMS, control passes to the next subsequent statement. If the SETFORMS request cannot be honored, control passes to the statement labeled sn in the ERR clause. If no ERR clause is coded, program execution is cancelled.

Example: SETFORMS (LISTFILE, ERR:103)

## SENDEOF

Purpose: To transfer an end-of-file indicator to the compatible channel.

Format: SENDEOF ( IOD-name, ERR: sn)

Description: A tape mark (end-of-file) is transmitted to the host System 2400. For a detailed description of SENDEOF, see Section 7.

Example: SENDEOF(PAYROLFILE, ERR: 101)



## STATION I/O STATEMENTS

STATION I/O statements provide the necessary interface between the operator and the application for data entry, validation and display. The STATION I/O statements are:

KENTER

KVERIFY

RESUME

RESUMERR

ERROR

NOTIFY

READSCREEN

READKEY

## KENTER

Purpose: To initiate a session of data entry/update from the keystation.

Format: KENTER ( IOD-name, KET-name, ESC:sn-1, ERR:sn-2)

Description: A session of entry/update is initiated with the KET serving as the focal point for station control and data display.

The IOD-name parameter must designate an IOD for which the device is STATION.

The "ERR:sn-2" phrase designates a program routine which is to receive control when a program or systems configuration error is encountered. If omitted and such an error condition occurs, program execution is cancelled.

The "ESC:sn-1" phrase designates a program routine which is to receive control when an application-level control key is depressed. The routine is also executed prior to operator input if the KET is defined with the "INIT=ESCAPE" clause. If omitted and either condition occurs, processing is continued as though the condition had not occurred.

For complete details of operation, see Section 8.

## KVERIFY

Purpose: To initiate a session of data verify/update from the keystation.

Format: KVERIFY (IOD-name, KET-name, ESC: sn-2, ERR: sn-2)

Description: A session of verify/update is initiated with the KET as the focal point for station control and data display.

The syntactic units are as described for KENTER.

For complete details of operation, see Section 8.

## RESUME

Purpose: To continue the active KENTER/KVERIFY session.

Format:

1. RESUME
2. RESUME(KET-field-name)

Description: For complete details of operation, see Section 8.

## RESUMERR

Purpose: To continue the active KENTER/KVERIFY session following display of an exception message.

Format:

1. RESUMERR
2. RESUMERR (B)
3. RESUMERR (B, KET-field-name)
4. RESUMERR (, KET-field-name)

Description: For Formats 2 and 3, the exception message (Field B) is edited and displayed. For Formats 1 and 4, an all blank operation message is edited and displayed.

Following operator acknowledgement, the session is continued at the point of lost interaction (Formats 1 and 2) or to a specific field (Formats 3 and 4).

If Format 3 or 4 is coded, field A must be a KET field-name within the currently active KET or a program error results.

For complete details of operation, see Section 8.

## ERROR

Purpose: To display an exception message for which no response, other than acknowledgement, is required.

Format: ERROR (B)

Description An exception message (field B) is displayed on line 2 of the display screen.

The RESET key is depressed to acknowledge the message. Control is then returned to the next in-line statement.

For details of operation, see Section 8.

Example: ERROR('RECORD NOT FOUND')

## NOTIFY

Purpose: To display an exception message and solicit a keyed response character.

Format: NOTIFY(B,C)

Description: An exception message (field B) is displayed on line 2 of the display screen.

The RESET key is depressed to acknowledge the message. A response is then keyed; the value of this key is placed in the leftmost position of field C. The control is then returned to the next in-line statement.

For details of operation, see Section 8.

Example: NOTIFY('RECORD NOT FOUND',FLDC)

## READSCREEN

Purpose: To obtain an individual line from the display.

Format: READSCREEN( IOD-name, buffer, ERR:sn-1, EOF:sn-2)

Description: A display line is edited for printing and transferred to the buffer.

The IOD-name must designate the station device.

The "ERR:sn-1" exit designates a program routine which is to receive control when an incorrect display line is referenced. If omitted, program execution is cancelled.

The "EOF:sn-2" exit designates a program routine which is to receive control when the successor to the last display line is referenced. If this exit is omitted, the reference is considered to be incorrect and is handled as described above.



## READKEY

Purpose: To obtain the next keyed data character or to be informed that no keyed data character is present.

Format: READKEY ( IOD-name, buffer, ERR:sn)

Description: The next keyed data character is returned in the leftmost position in the buffer; the program routine designated by “ERR:sn” is executed if no keyed character is present.

The IOD-name must designate the STATION device.

For details of operation, see Section 8.

## SPECIAL PURPOSE STATEMENTS

Special Purpose Statements perform specialized functions to enhance application program capability. The Special Purpose statements are:

CHECKDIGIT

GETTIME

SETTIME

SAME

STRING

## CHECKDIGIT

Purpose: To validate or generate the checkdigit of a self checking number.

Format:

<b>VALIDATE</b>	<b>GENERATE</b>
1. CK10 (B, ERR: sn)	5. B=CK10 (B, ERR: sn)
2. CK11 (B, ERR: sn)	6. B=CK11 (B, ERR: sn)
3. CKTB1 (B, ERR: sn)	7. B=CKTB1 (B, ERR: sn)
4. CKTB2 (B, ERR: sn)	8. B=CKTB2 (B, ERR: sn)

Description: The contents of field B, exclusive of the rightmost position, are evaluated to determine the checkdigit for the field.

1 — 4. The calculated checkdigit is compared with the rightmost position of field B. If they do not match or the field contents are not suitable for the selected algorithm, control is transferred to the statement labeled sn.

5 — 8. The calculated ccheckdigit is moved to the rightmost position of field B. If the calculated digit is not suitable for the selected algorithm, control is transferred to the statement labeled sn.

The selected algorithms are:

1,5 : IBM Modulus 10

2,6 : IBM Modulus 11

3,7 : Custom algorithm.

4,8 : Alternate custom algorithm.

Example 1: CK10 (FLDB, ERR: 101)

Example 2: FLDB=CK11 (FLDB, ERR: 101)

Example 3: FLDA=CK11 (FLDB, ERR: 101)

Example 3 illustrates that the receiving field for generation may be different from the source field.

**NOTE:** See Appendix G for CHECKDIGIT Algorithms.

## GETTIME

Purpose: To read the System Time-Of-Day Clock.

Format: GETTIME (B)

Description: The time of day is retrieved from the clock and stored in field B.

The System Time-Of-Day is a 24-hour clock. This field is a twelve-position decimal field containing:

YYMMDDhhmmss

Where:

YY = year

MM = month

DD = day

hh = hours

mm = minutes

ss = seconds

The system increments only hours, minutes and seconds. Space is allocated for year, month and day so that they may be optionally incremented by an application program. When the time of day is retrieved, it is moved right-to-left to field B. The operation terminates when field A is full. If field B is longer than twelve positions, it is filled with leading zeros.

During the Initial Program Load, the clock is filled with zeros. A value may be key-entered whenever the Selection Display Screen is conditioned to perform a program load. Care should be taken so that conflicting settings of the time-of-day clock are avoided.

Example: GETTIME (FLDB)

## SETTIME

Purpose: To set the System Time-Of-Day Clock.

Format: SETTIME (B)

Description: The time of day is moved from field B to the clock. The System Time-Of-Day Clock is a 24-hour clock. This field is a twelve-position decimal field containing:

YYMMDDhhmmss

Where:

YY = year

MM = month

DD = day

hh = hours

mm = minutes

ss = seconds

The system increments only hours, minutes and seconds. Space is allocated for year, month and day so that they may optionally be incremented by an application program. When the time of day is transferred to the clock, it is moved from right-to-left from field B until every position (up to the rightmost twelve) of field B has been transferred. If field B contains fewer than twelve positions, the Time-Of-Day Clock is filled with leading zeros.

Care should be taken so that conflicting settings of the Time-Of-Day Clock are avoided.

Example: SETTIME (FLDB)

## SAME

Purpose: To select a field or buffer for subsequent program reference.

Format: EQU-name=SAME (buffer)

Description: After assignment of EQU-name as an alternate reference name for buffer, EQU-name may be used to reference the contents of the selected buffer.

Example:

```
RCD:  .
      .
      .
      COUNT1 (5)
      .
      .
      .
      COUNT2 (8)
      .
      .
      .
      EQU: FLDE;                               (Subroutine Parameter)

      START
      .
      .
      .
      FLDE = SAME (COUNT1)                   (Setup Parameter)
      PERFORM: 100                             (Call Subroutine)
      .
      .
      .
      FLDE = SAME (COUNT2)                   (Setup Parameter)
      PERFORM: 100                             (Call Subroutine)
      .
      .
      .
      100, ENTRY
      .
      .
      .
      FLDE = FLDE + 1                           (Increment Parameter)
      .
      .
      .
      EXIT: 100
      END
```

**NOTE:** For interpretation of EQU-name where buffer is greater than 256 positions, see Table 6-3.

## STRING

<u>Purpose:</u>	To select a sub-field of a field or a buffer for subsequent program reference.
<u>Format:</u>	EQU-name=STRING(buffer, B, C, ERR: sn)
<u>Description:</u>	After assignment of EQU-name as an alternate reference name for a sub-field of the buffer, EQU-name may be used to reference the contents of the selected sub-field.

NOTE: For interpretation of EQU-name when the resulting sub-field of the buffer is greater than 256 positions, see Table 6-3.

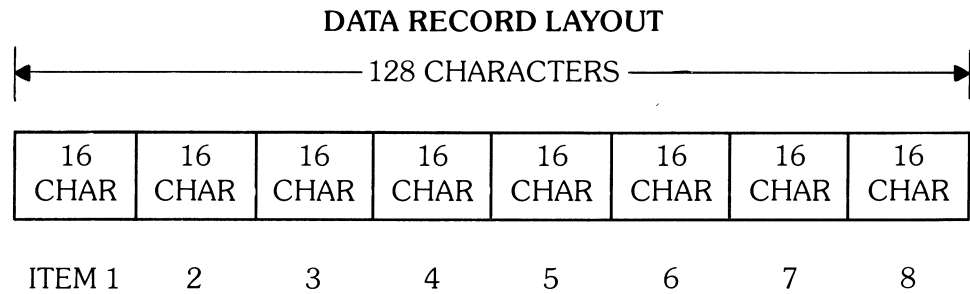
The binary contents of the rightmost two bytes of field B indicate the number of leading bytes of the specified buffer which precede the desired sub-field (offset). The binary contents of the rightmost two bytes of field C indicate the length of the sub-field (length).

The sub-field must be within the bounds of the specified buffer. If any part of the sub-field lies beyond those bounds:

1. The EQU-name is not changed.
2. The error indicator is set.
3. If the ERR exit is coded, control is transferred to the statement labeled sn.

Example:

The example program processes input records each of which contains eight 16 byte items.



```
RCD:  BUFFER                               (Input buffer of:
      -(128);                               8 * 16 byte items)

RCD:  PARAMS                               (Offset in buffer to item)
      OFFSET = X'00 00';

EQU:  SUBFIELD;                            (Active item for processing)

START
      .
      .
      .
10,  OFFSET = X'00 00'                      (Set (RESET) offset)
      READ (—,Buffer, EOF:30)              (Read (new) buffer)

20,  SUBFIELD = STRING (BUFFER, OFFSET,     (Select item until buffer ends)
      16, ERR:10)
      OFFSET = ADD (OFFSET,16)            (Setup for next item)
      .
      .
      .
      GO:20                                } Process this item
      .                                    } (Continue extracting)
      .
30,  .
      .
      .
      END
```



NOTE: The following operations are supported as synchronous operations by Release 7 compiler and system software:

BACKSPACEA  
MARKA  
PRINTA  
READA  
READLOCKA  
READNEXTA  
REWINDA  
REWINDLOCKA  
SENDEOFA  
SKIPFILEA  
WAIT

## SECTION 7: INPUT/OUTPUT OPERATIONS

Data is recorded at a device on some form of media. All or a portion of that media may be available for the programmer's use. Input/Output operations transfer data between the application program in main memory and the I/O device. Input operations receive data from a device to the application program. Output operations send data from the application program to the output device. Both operations involve some device. The data transferred is contained in datasets (files). A complete definition of a dataset is presented in Appendix F of this manual. A dataset may be thought of as the physical structure containing data. A file is a logical structure of data.

Statements involved with I/O operations may be divided according to the subsections of MOBOL.

### I/O STATEMENTS IN THE DATA DEFINITION SECTION

The Data definition statements define initial parameters for the I/O devices. They are:

- Key Entry Table Statement (KET)
- Input/Output Descriptor Statement (IOD)

The KET statement is documented in Sections 5 and 8. The IOD statement is used to identify an I/O device, its access characteristics and the datasets which may be contained on the device. The IOD specifies certain information regarding control parameters used by the operating system to access the data. When an IOD is declared, a File Control Block (FCB) containing the parameters for the IOD is created in the data section of memory by the compiler. The access method control program uses this information when it accesses the dataset. The FCB is an interface between the system and the MOBOL program. The IOD declaration is a means to provide certain default information to the File Control Block when the program is initially loaded into the system. Certain parameters may be changed at execution time by use of the IOD-name.keyword syntax. The syntax of the IOD statement is presented in Section 5 of this manual.

## I/O STATEMENTS IN THE EXECUTION SECTION

I/O execution statements direct operation of the I/O devices. Table 7-1 lists all statements and the devices to which they apply.

**Table 7-1: I/O Execution Statements by Device**

	Station	Disk/Diskette	Tape	Data Recorder	Printer	Comp Chan
BACKSPACE			X			
CHECKEOD		X				
CHECKFORMS					X	
CLOSE		X	X	X	X	X
DELETE		X				
ERROR	X					
FREESPACE		X(DISK only)				
INSERT		X				
KENTER	X					
KVERIFY	X					
MARK			X			
NOTIFY	X					
OPEN		X	X	X	X	X
PRINT					X	
READ		X	X	X		X
READKEY	X					
READLOCK		X				
READNEXT		X				
READSCREEN	X					
RELEASE		X				
RESUME	X					
RESUMERR	X					
REWIND			X			
REWINDLOCK			X			
SENDEOF						X
SETEOD		X				
SETFORMS					X	
SKIPFILE			X			
WRITE		X	X	X	X	X

The I/O statements listed above that are used for STATION are explained in Section 8. Statements pertaining to all other devices are discussed within each device operation subsection. General syntax for all statements is presented in Section 6.

## KEYWORD CLASSIFICATION

Keywords are classified according to the functional locations of their appearance in MOBOL source code. D keywords appear in the Data Definition Section. E keywords appear in the Execution Section. DE keywords may appear in either section.

### Keywords Used In The Data Definition Section (D Keywords)

Keywords appearing in the Data Definition Section (D keywords) either define parameters or error message text. They appear in the IOD statement in the form:

$$\text{Keyword} = \text{Value}$$

Keywords relating to STATION operations are discussed in Section 8. Keywords pertaining to I/O operations for DISK, DISKETTE, TAPE (DATA RECORDER) and COMP CHAN (Compatible Channel) are:

#### ACCESS

A parameter that specifies the desired access method, degree of exclusive use and whether a "READ after WRITE" check is to be performed. This parameter, available only in the iod-specification, is a compile-time parameter. It is not addressable at run-time. The format is:

$$\text{ACCESS} = \left\{ \begin{array}{l} \text{BASIC} \\ \text{RANDOM} \\ \text{SEQUENTIAL} \end{array} \right\}, \left[ \begin{array}{l} \text{EXCLUSIVE} \\ \text{SHARED} \end{array} \right], \left[ \begin{array}{l} \text{NVERIFY} \\ \text{VERIFY} \end{array} \right]$$

The default values are BASIC, EXCLUSIVE and NVERIFY. They are used in the absence of the ACCESS iod-specification or the absence of their respective selection group. RANDOM may be abbreviated using RAN, SEQUENTIAL may be abbreviated using SEQ.

#### KEYVALUE

A keyword that designates a data-name in a previously defined RCD. The previously defined RCD must not be an RCD array. The contents of the field associated with this data-name will be used by the RANDOM INDEX ACCESS METHOD to determine the location of a specific target record. KEYVALUE is valid only when ACCESS=RANDOM and must be specified after the ACCESS clause. See "RANDOM INDEX ACCESS METHOD" on subsequent pages of this section.

#### TARGET

A keyword which designates the IOD-name of the target dataset previously defined. TARGET appears in the IOD of an index dataset. The TARGET clause is required and considered valid only when ACCESS = RAN or ACCESS = SEQ. It must be specified after the ACCESS clause.

## Keywords Used In The Execution Section (E Keywords)

Keywords appearing in the Execution Section relate to the status and parameters of I/O operations. These keywords may appear in any of the execution statements where A, B, C or D is indicated in the syntax of the statements. The format used is:

IOD-name . Keyword

Keywords used for STATION operations are presented in Section 8. Keywords pertaining to I/O operations for TAPE (DATA RECORDER), DISK, DISKETTE and COMP CHAN are:

ACTUAL	A six-byte field corresponding to the volume-name. It contains the actual volume-name found on the unit when OPEN was executed.
BLKLEN	Is a two-byte binary parameter indicating the block length (used as a maximum record length) specified in the dataset label.
EOD	A two-byte binary READ-ONLY field that indicates the logical end of data (the number of records in the dataset). The EOD value is the relative record number of the next record to be appended to the file.
EOE	A two-byte binary READ-ONLY field indicating the dataset's last physically allocated record number.
ERRCODE	<p>A one-byte READ-ONLY field containing the completion code for the operation. For an abnormal completion, ERRCODE will contain the actual code used to designate the specific termination result (i.e., reason for abnormal completion). The bit encoding is:</p> <p style="margin-left: 40px;">7654 3210 SSSS MMMM</p> <p>M = Major status S = Sub-status</p> <p>See Appendix E.</p>
FORMS	<p>A four-byte parameter indicating the binary parameters (listed below) for the printer.</p> <p style="margin-left: 40px;">Byte 1 = lines per page Byte 2 = characters per inch Byte 3 = lines per page Byte 4 = lines per form</p> <p>The default values are 66 lines per page, ten characters per inch, six lines per inch and 58 lines per form. These values are established during Initial Program Load but may be altered by subsequent use of SETFORMS.</p>

MARK	<p>A one-byte parameter used to specify whether the current record is active or deleted. The values for this field are:</p> <p style="margin-left: 40px;">X'03' = Normal record X'00' = Deleted record</p> <p>This parameter is valid only for DISKETTE.</p>
OUTLEN	<p>A two-byte binary field that indicates the length of the next output buffer. This parameter applies to all devices (DISK, DISKETTE, PRINTER, TAPE (DATA RECORDER) and COMP CHAN).</p> <p>OUTLEN may be used to override the length attribute for a WRITE operation in lieu of specifying a variable length buffer. Prior to the WRITE operation, the intended length of the buffer is placed in IOD-name.OUTLEN. Then, when the WRITE operation takes place, this length is used instead of the buffer length. This override mechanism is a one-time execution (that is, the length must be placed in OUTLEN for each specific operation).</p>
POSITION	<p>For DISK/DISKETTE: A two-byte binary field that specifies the current relative record number (X'00 00' is the first record) of the active dataset. This parameter may be used to pass random location information to a READ or WRITE function causing the device to access the specified record.</p> <p>For PRINTER: A one-byte binary READ-ONLY parameter that indicates the current line position on the form. The value X'00' indicates top-of-form.</p>
STATUS	<p>A one-byte READ-ONLY field that indicates the current operational condition of the I/O device.</p> <p>Actual coding of the major and minor status bits indicate various conditions depending upon the I/O device used.</p> <p>NOTE: STATUS is provided for compatibility between system levels. It is recommended that ERRCODE be used in lieu of STATUS.</p>
XFERLEN	<p>A two-byte binary READ-ONLY field that indicates the length minus one of the data record last read into the buffer.</p>
XFERSTATUS	<p>A field that indicates the status of the Series 21 when it is connected (via compatible channel) to the System 2400. Bit encoding is presented in this section under "COMPATIBLE CHANNEL I/O OPERATIONS".</p>

## Keywords Used in Both The Data Definition And The Execution Section (DE Keywords)

The keywords that may be used in both sections of MOBOL are presented below. When used in the Data Definition Section, they define parameters in the IOD statement. The format for this section is:

Keyword = Value

In the Execution Section, these keywords are used to reference their respective parameters. They may appear in any of the execution statements where fields A, B, C or D is indicated in the syntax of the statements. The format for the Execution Section keyword reference is:

IOD-name.Keyword

Keywords that may be used in both sections of MOBOL are listed and defined below.

**DATASET**                    An eight-byte field\* containing the name of the dataset to be used by this file. For a DISKETTE, the whole volume will be used as the dataset if the leftmost byte of the field contains a X'FF' (e.g., DATASET = :FF:).

**SLEW**                        A one-byte parameter for directing forms control for a printer. Paper advances may be performed before or after printing according to the value set. The bit encoding is:

7654 3210  
TBnn nnnn

T = 0 - Slew  
     1 - Top of Forms

B = 0 - Before Print

N = number of lines in binary to slew when T = 0.

If the slew value chosen causes the line position to exceed the bottom of a form (lines per form), and end-of-form is generated. The print operation is performed before the EOF exit is taken. The slew value supplied by the application program is *not* changed.

\*The dataset-name must be exactly eight characters and must be properly aligned. The default value is "DATA**bbbb**".

## STATE

A one-byte parameter indicating the status of the file as it is referenced by the IOD (e.g., OPEN or CLOSED, SHARED or EXCLUSIVE). The STATE is specified by the values set in each bit. The bit encoding is:

7654 3210  
0EXX XXCD

O = 0 - If the file is CLOSED  
1 - If the file is OPEN

E = 0 - If the file is SHARED  
1 - If the file is EXCLUSIVE

X = Reserved for future use

C\* = 0 - New print operation  
1 - Continue previous print

D\* = 0 - If the SLEW information is contained in the IOD parameter  
1 - If the SLEW information is in the first position of the data buffer

## UNIT

A one-byte binary field indicating a specific drive (in the case of DISK or DISKETTE) or a specific printer (in the case of PRINTER). A value of X'00' has a special meaning in that the I/O system will search for the required unit (see OPEN in the device dependent sections). The default value of X'01' is used if UNIT is not specified in the IOD. A decimal number may be specified for UNIT in the data definition for the IOD. It is converted to binary representation by the compiler.

## VOLUME

A six-byte field\*\* specifying the name of the volume to be used. If this field contains nulls (X'00') at OPEN time (e.g., VOLUME = :00), the I/O system will search for the dataset on any volume (as specified by the contents of the UNIT field). The default value is all nulls.

\*Used only at OPEN time. Once set in the IOD, these bits can only be changed while the file is closed.

\*\*This field must be exactly six characters.



**Table 7-2: Keywords By Device**

	Diskette	Disk	Printer	Tape	Data-Recorder	Comp Chan
ACCESS	X	X				
ACTUAL	X	X				
BLKLEN	X	X				
DATASET	X	X				
EOD	X	X				
EOE	X					
ERRCODE	X	X	X	X	X	X
FORMS			X			
KEYVALUE	X	X				
MARK	X					
OUTLEN	X	X	X	X	X	X
POSITION	X	X	X			
SLEW			X			
STATE	X	X	X	X	X	X
STATUS	X	X	X	X	X	X
TARGET	X	X				
UNIT	X	X	X	X	X	
VOLUME	X	X				
VOLUME	X	X				
XFERLEN	X	X		X	X	X
XFERSTATUS						X

### **DISK/DISKETTE I/O OPERATIONS**

DISK and DISKETTE I/O operations are similar. Exceptions and/or major differences will be noted in the text.

In these operations there are three access methods: Basic, Sequential Index and Random Index.

### **BASIC ACCESS METHOD**

The Basic Access Method is the control program responsible for logical Input/Output operations to the I/O device DISKETTE or DISK.

## **DISKETTE Operations**

MDS supports diskettes in a manner which is consistent and compatible with IBM's definition of Basic Data Exchange. MDS does not support the alternate track assignment feature.

A diskette dataset is a series of contiguous records with a constant length (less than or equal to 128 bytes). The dataset has a label in the Volume Table Of Contents (VTOC) which describes the dataset configuration. The VTOC is recorded at the beginning of the diskette on the Index Track. This dataset label contains the dataset-name and the absolute addresses for beginning-of-extent (BOE), end-of-extent (EOE) and the relative address of end-of-data (EOD) as well as other information required by the IBM Basic Data Exchange standards.\* A diskette record is a series of bytes preceded by a mark byte (which is not part of the actual data). Data is recorded in the EBCDIC representation.

The diskette is physically configured in 128 byte sectors; however, the logical record size may be set to any length from one to 128 bytes. Record lengths of less than 128 bytes will be recorded leaving the unused bytes of the sector null-filled. For the Basic Access Method, diskette records (logical records) always begin at the first data byte of the sector; there can only be one record per sector.

Records are numbered sequentially, starting with zero. This relative record number (i.e., POSITION) is used to address the records. The Basic Access Method maintains a pointer to the sector following the last logical record of a dataset. This sector is called the EOD Sector. A dataset containing no data has an EOD = X'00 00'.

The concept of POSITION, EOD, BLKLEN, XFERLEN and OUTLEN are described within the keyword definitions on prior pages of this section.

## **DISK Operations**

MDS usage of the various types of supported disks is not intended to be compatible with any other manufacturer. The software support is designed so that differences between types of disk drives are transparent to the user.

A disk dataset is a series of logically connected records with a constant length (between 10 and 4096 bytes) having a label describing the dataset configuration in the DISK Volume Table of Contents (VTOC). This dataset label contains the dataset-name and the absolute address for BOE and the relative address of EOD as well as a maximum allocation indicating a size which the dataset may not exceed. Data is recorded in EBCDIC representation.

A DISK dataset is configured as logically connected records of a length which may be initially defined (or later redefined) using the Disk Utility Program. Records are numbered sequentially starting with zero (0). The concept of POSITION, EOD, BLKLEN, XFERLEN and OUTLEN are described within the keyword definitions on prior pages of this section.

\*See the "Diskette Organization Section" of the Series 21 Operator's Guide (Form No. M-3611).

Applicable operations for the Basic Access Method are:

## **OPEN**

Purpose: Obtain access to a dataset on DISK or DISKETTE.

Format: OPEN( IOD-name, buffer, ERR:sn)

Description: The OPEN operation searches for the dataset specified in the IOD. The .DATASET, .VOLUME and .UNIT keywords are relevant to the OPEN operation.

The programmer can specify this search in several ways according to the following table:

**Table 7-3: Search Operation By OPEN For DISK/DISKETTE**

	<b>VOLUME = nulls</b>	<b>VOLUME not = nulls</b>
UNIT = 0	search for dataset on any unit.	search for volume on any unit; then search for dataset.
UNIT not = 0	search specified unit for dataset.	verify volume on specified unit; then search for dataset.

NOTE: The buffer length must be equal to 128 bytes.

If the OPEN operation cannot be successfully completed, the error condition which caused the failure may be determined from the .ERRCODE keyword (see Appendix E).

When a data set is being reopened for the purpose of appending to the file, the .POSITION parameter should be reset after the OPEN to contain the current value of EOD (i.e., IOD-name.POSITION=IOD-name.EOD)

## **READ**

Purpose: To transfer a data record from a dataset on DISK or DISKETTE to main memory.

Format: READ ( IOD-name, buffer, ERR: sn-1, EOF: sn-2)

Description: The READ operation reads the record indicated by the .POSITION keyword and places it in the buffer supplied by the READ statement. The length of the record actually transferred may be referenced by means of the .XFERLEN keyword. Upon successful completion of the READ operation, the data is transferred from the DISK/DISKETTE record to the buffer and the value of the .POSITION keyword is incremented to point to the next record.

If the supplied buffer is too short to contain the record read, an error is generated which may be referenced by means of .ERRCODE. If the supplied buffer is longer than the actual record, the remainder of the buffer is filled with blanks. The minimum length of a record is 1 byte for DISKETTE and 10 bytes for DISK. The maximum length of a record is 128 bytes for DISKETTE and 4096 bytes for DISK.

If any errors are detected during the operation, the .POSITION keyword is not incremented.

If a whole volume on DISKETTE is specified (first byte of dataset-name is X'FF'), UNIT must be specified.

Accessibility to a diskette volume or dataset may be restricted according to the accessibility fields in the VTOC.

The OPEN operation also establishes the usage of the dataset as EXCLUSIVE or SHARED by reference to the STATE parameter in the IOD. Bit 6 of the STATE parameter determines usage ("1" - EXCLUSIVE; "0" - SHARED). If the dataset is requested for exclusive use, the OPEN will be successful only if no other user has an open file for this dataset. If shared use is requested, the OPEN will be successful if all other files open for this dataset have also requested shared use. An entire volume may be opened for exclusive use if there are no other users with OPEN datasets on the volume. Opening the entire volume with shared usage allows the entire volume to be read but not written.

At conclusion of OPEN, reference may be made to the volume-name, current record position, current EOD, record length, and EOE by means of the .ACTUAL, .POSITION, .EOD, .BLKLEN and .EOE keywords.

The OPEN operation gets the volume-name, EOD, BLKLEN, and EOE from the dataset label in the VTOC. The representations of EOD and EOE keywords are converted to relative displacement; BLKLEN is converted to binary. The current position is initially set to zero, to designate the first record in the dataset.

NOTE: There is no EOE keyword for DISK.

Prior to the READ, the user may alter the value of the .POSITION keyword to point to any valid record, allowing for random retrieval of records. If .POSITION points to the EOD Sector at the beginning of the operation, the EOF exit is taken. If an attempt is made to read a record beyond the EOD Sector, the ERR exit is taken.

Upon successful completion of READ, the .MARK keyword will contain a value derived from the mark byte of the diskette record. A value of X'00' indicates the record is deleted. Values other than X'00' indicate the record is not deleted. The mark byte and the .MARK keyword apply only to DISKETTE.

## **READLOCK**

Purpose: To transfer a data record from a dataset on DISK or DISKETTE to memory and to prevent any other user from reading that record.

Format: READLOCK( IOD-name,buffer,ERR:sn-1,EOF:sn-2)

Description: A READLOCK operation is similar to a READ operation. The record described by the .POSITION keyword is locked to prevent read access through other IOD's. The .POSITION keyword is not incremented (in anticipation of a subsequent WRITE operation which updates the record and clears the lock).

One record at a time may be locked for an IOD.

## **CHECKEOD**

Purpose: To lock the EOD record of a dataset on DISK or DISKETTE and to prevent other users from accessing that record concurrently.

Format: CHECKEOD(IOD-name,ERR:sn)

Description: CHECKEOD is described in detail in Section 6.

## WRITE

Purpose: To transfer a data record (128 bytes) of data from memory to a dataset on DISK or DISKETTE.

Format: WRITE (IOD-name, buffer, ERR:sn-1, EOF:sn-2)

Description: The WRITE operation transfers the data from the supplied buffer to the dataset record at the location indicated by .POSITION. The POSITION keyword is automatically incremented by one upon completion.

If the supplied POSITION is less than EOD, any existing data in the record is overwritten. If the POSITION is equal to EOD, the record is appended to the dataset and EOD is automatically incremented by one. If the POSITION is greater than EOD, the record is not written and the ERR exit is taken. If the record is shorter than the record length (defined in the dataset label of the VTOC), the remainder of the record is padded with blanks. For DISKETTE, if the logical record length (from the dataset label) is less than 128 bytes, the remainder of the 128 bytes is padded with nulls (X'00'). If the buffer length is longer than the record length, the data is truncated on the right.

OUTLEN can be used to override a buffer-size, record-size mismatch. See OUTLEN under "KEYWORDS USED IN THE EXECUTION SECTION". If the WRITE operation has been preceded by a CHECKEOD or a READLOCK, successful completion of the WRITE releases the record. Prior to executing a WRITE operation, the .MARK keyword may be adjusted to write a deleted or a non-deleted record. The WRITE operation transfers the value in .MARK to the mark byte of the diskette record. X'00' indicates a deleted record. Values other than null indicate a non-deleted record. For example, to write a deleted record:

```
IOD-name.MARK = X'00'  
WRITE (IOD-name, buffer, ERR:sn-1, EOF:sn-2)
```

The mark byte and the .MARK keyword apply only to DISKETTE. The effect of the WRITE operation on .POSITION and .EOD keywords is discussed under WRITE in Section 6. Normal or abnormal completion codes of a WRITE may be referenced via the .STATUS and .ERRCODE keywords.

If an error condition occurs, the ERR exit is taken. If an end-of-file condition occurs, the EOF exit is taken. If the EOF branch is taken on the completion of the WRITE operation, the last available sector of the dataset has just been used.

## **RELEASE**

Purpose: To remove restricted access (without transferring data) on an external dataset record which has been the object of a READLOCK or CHECKEOD operation.

Format: RELEASE (IOD-name, ERR: sn)

Description: The RELEASE operation is described in detail in Section 6 of this manual.

## **SETEOD**

Purpose: To update the system-maintained EOD (end-of-data)

Format: SETEOD (IOD-name, ERR: sn)

Description: The SETEOD operation is described in detail in Section 6 of this manual.

NOTE: Only the records between the beginning of the dataset and the EOD are considered to be valid data records. Normally, the EOD is only advanced in response to WRITE operations at EOD. The program, however, can alter the value of EOD to any value between zero and EOE with the SETEOD statement. The value in POSITION at the time SETEOD is issued becomes the new EOD. Since, for DISK, the value of EOE can grow to expansion of the dataset, the SETEOD statement can also be used to preallocate space for the dataset.

SETEOD can only be used with DISK/DISKETTE if ACCESS = EXCLUSIVE.

## **FREESPACE**

Purpose: To free allocated space beyond EOD (end-of-data).

Format: FREESPACE (IOD-name, ERR: sn)

Description: FREESPACE (which applies only to DISK) is described in detail in Section 6.



## **CLOSE**

Purpose: To terminate access to an OPENed dataset on a DISK or DISKETTE.

Format: CLOSE ( IOD-name, buffer , ERR:sn)

Description: When CLOSE is executed, the system-maintained EOD (which is updated with each operation) is written to the dataset label of the VTOC. The VTOC EOD parameter is updated only at this time. When a MOBOL program signs off (see STOP in Section 6) any remaining open datasets for that program are automatically closed by the system.

In addition to updating the EOD in the dataset label of the VTOC, the CLOSE operation resets the leftmost bit of the STATE keyword to zero. Thus, the STATE parameter may be examined to determine whether a dataset is OPEN or CLOSED.

NOTE: The buffer length must be equal to 128 bytes.

## SEQUENTIAL INDEX ACCESS METHOD

The Sequential Index Access Method is a means of accessing a dataset (the target dataset) via an index. The index dataset consists of record pointers indicating the sort sequence (the record pointers are sorted in ascending or descending order of the contents of the keyfields within the target records). Any number of index datasets (with differing sort sequences or keys) may reference the same target dataset.

DMU (Data Management Utility Program) is used to create the index dataset. The user specifies sort fields which reference fields in the target dataset records.

The index dataset is later used by a MOBOL application program to retrieve records, in a sorted order, from a target dataset. Using a sequential index requires two IOD's in the MOBOL program: one for the dataset containing the data records (the target dataset) and one for the index dataset. The index IOD must contain the TARGET keyword referencing the target IOD (TARGET=IOD-name of the target dataset). The index IOD must also include ACCESS=SEQ (see ACCESS described under "KEYWORDS USED IN THE DATA DEFINITION SECTION" on previous pages of this section).

The operations that may be used with the Sequential Index Access Method include:

- OPEN
- CLOSE
- DELETE
- READ
- READLOCK
- RELEASE
- CHECKEOD
- SETEOD
- WRITE

These operations function similarly to operations under the Basic Access Method except that all references for I/O operations are made through the index dataset. Operations using the Sequential Index Access Method affect the parameters of the IOD associated with the index dataset in the same manner as operations for the Basic Access Method. Parameters of the target IOD are adjusted by these operations to effect access to the target dataset.

## RANDOM INDEX ACCESS METHOD

The Random Index Access Method (RIAM) provides a means to access a target dataset by record content (keyfield) rather than relative record position. Using this access method requires the use of an index of the target dataset. This index is created by the Data Management Utility Program (DMU). DMU allows a user to specify keyfields by which the records are to be accessed. DMU creates an index dataset, which is separate from the target dataset, and must be identified by a separate IOD. The target dataset itself is not affected by DMU. The index dataset created by DMU consists of short records (four bytes each) containing pointers to the target dataset records. These short index records are positioned in the index dataset according to the result of a hashing of the key values contained in the user-specified keyfields. (The term hashing refers to a process of generating an arithmetic summation of the contents of the keyvalue.)

The MOBOL application program supplies a particular KEYVALUE to RIAM. Then, RIAM hashes that value, locates the short record in the index (based on the hash value) and obtains the POSITION of the desired record in the target dataset.

A MOBOL program may use RIAM by supplying certain IOD and keyword declarations in its Data Definition Section. These declarations include an IOD for the index dataset which specifies ACCESS=RAN, TARGET=IOD-name of the target dataset and KEYVALUE=data-name. The parameters of these keywords are detailed in "KEYWORDS USED IN THE DATA DEFINITION SECTION" on previous pages of this section. Applicable operations are:

### OPEN

Purpose: Obtain access to a dataset on DISK or DISKETTE.

Format: OPEN ( IOD-name , buffer , ERR: sn )

Description: Operations are similar to those described for the Basic Access Method. Both the index dataset and the target dataset must be OPENed.

### READ

Purpose: To transfer a data record from a dataset on DISK or DISKETTE to main memory.

Format: READ ( Index- IOD-name , buffer , ERR: sn-1 , EOF: sn-2 )

Description: The field designated by the KEYVALUE keyword contains the desired search parameter. Since this data field is an RCD field, a MOBOL move statement (FIELD=Value : :) may be used to set the search parameter.

Once this data field is set, the READ is issued. RIAM will read the index dataset, find the correct target record position and transfer the data from the target record to the buffer.

NOTE: The relationship between buffer length and record length is as described for the READ operation of the Basic Access Method.

## **READNEXT**

Purpose: To transfer the next data record, having a keyvalue previously read, via an index dataset.

Format: READNEXT ( Index-IOD-name, buffer, ERR:sn-1, EOF:sn-2)

Description: When more than one record contains the same key value, this statement may be used to acquire the second record (and subsequent records, via repeated executions). When the series of duplicated key values is exhausted, the READNEXT operation takes the EOF:sn-2 branch.

## **READLOCK**

Purpose: To transfer a data record from a dataset on DISK or DISKETTE to memory and to prevent any other user from reading that record.

Format: READLOCK ( Index-IOD-name, buffer, ERR:sn-1, EOF:sn-2)

Description: Operation of READLOCK for RIAM is identical to the operation described for the Basic Access Method.

## **RELEASE**

Purpose: To remove restricted access (without transferring data) on an external dataset record which has been the object of a READLOCK operation.

Format: RELEASE ( Index-IOD-name, ERR:sn)

Description: Operation for RELEASE for RIAM is identical to the operation described for the Basic Access Method.

## **DELETE**

Purpose: To remove a target record position entry from an index dataset.

Format: DELETE ( Index- IOD-name, ERR:sn)

Description: The DELETE operation is described in detail in Section 6 of this manual.

## **WRITE**

Purpose: To transfer a data record of data from memory to a dataset on DISK or DISKETTE.

Format: WRITE ( Index- IOD- name, buffer, ERR: sn-1, EOF: sn-2)

Description: The WRITE statement does not go through the hashing process, but instead uses information in the target IOD provided by a prior READ to the index dataset. In RIAM, WRITE uses the relative record position of the target dataset just as in the Basic Access Method. That is, to WRITE a record to a target dataset, Target IOD- name. POSITION must contain the correct position value. This is accomplished as follows: READLOCK the record using the KEYVALUE method. This action will generate the correct target record position in the target IOD.

The subsequent WRITE (e.g., WRITE (Index-IOD-name,buffer,ERR:sn)) will transfer the data from the buffer to the correct record position in the target dataset.

For every record added to the target dataset, a new index is required. For every keyvalue modified in the target dataset, a new index entry is required (the existing index entry to this record should be deleted). The new record is created using a WRITE operation to the target dataset to establish the target .POSITION and the record value. The INSERT operation, then is executed.

## **INSERT**

Purpose: To cause a new entry to be included in an index dataset for an existing (but not previously indexed) record in the target dataset.

Format: INSERT ( Index- IOD- name, ERR: sn)

Description: The INSERT statement is described in detail in Section 6.

## **CLOSE**

Purpose: To terminate access to an OPENed dataset on a DISK or DISKETTE.

Format: CLOSE ( IOD- name, buffer, ERR: sn)

Description: Operations are similar to those described for the Basic Access Method. A CLOSE must be executed against both the index dataset and the target dataset.

The keywords used for I/O operations for DISK and DISKETTE are tabled below.

**Table 7-4: Keywords Used For I/O Operations**

<b>Keyword</b>	<b>Basic Access Method</b>	<b>Sequential Index</b>	<b>Random Index</b>
ACCESS	X	X	X
ACTUAL	X	X	X
BLKLEN	X	X	X
DATASET	X	X	X
EOD	X	X	X
EOE (DISKETTE only)	X	X	X
ERRCODE	X	X	X
KEYVALUE			X
MARK (DISKETTE only)	X	X	X
OUTLEN	X	X	X
POSITION	X	X	X
STATE	X	X	X
STATUS	X	X	X
TARGET		X	X
UNIT	X	X	X
VOLUME	X	X	X
XFERLEN	X	X	X

## **TAPE I/O OPERATIONS**

Magnetic tape consists of a series of datasets (files) each of which is terminated by a single tape mark. The end of the last dataset (i.e., multifile) is terminated by double tape marks. Only one OPEN is required to access all datasets (files) on the tape. Following the OPEN, the program may process all or selected datasets on the magnetic tape volume.

Applicable operations for TAPE are:

### **OPEN**

Purpose: To gain access to a magnetic tape drive.

Format: OPEN ( IOD-name, buffer, ERR:sn)

Description: When the tape dataset is OPENed, the leftmost bit of the .STATE keyword is set to one (1) indicating the open state of the dataset.

The OPEN operation establishes the usage of the tape as EXCLUSIVE or SHARED by reference to the STATE parameter in the IOD. Bit 6 of the STATE parameter determines usage ("1"-EXCLUSIVE; "0"-SHARED). If the tape is requested for exclusive use, the OPEN will be successful only if no other user has the tape open as a file. If shared use is requested, the OPEN will be successful if all other files open for this tape have also requested shared use.

If the OPEN operation cannot successfully be completed, the error condition which caused the failure may be referenced by means of the .ERRCODE keyword (see Appendix E).

### **READ**

Purpose: Transfer a record from a magnetic tape file to main memory.

Format: READ ( IOD-name, buffer, EOF:sn, ERR:sn)

Description: After a READ operation, .XFERLEN contains the length of the data record just read.

ERRCODE and STATUS receive the completion code after READ is executed.

### **WRITE**

Purpose: Transfer a record from memory to a magnetic tape file.

Format: WRITE ( IOD-name, buffer, EOF:sn-1, ERR:sn-2)

Description: In a statement prior to the WRITE, the IOD-name.OUTLEN parameter may be used to override the buffer length for one WRITE operation (see "KEYWORDS IN THE EXECUTION SECTION" in this section).

ERRCODE and STATUS receive the completion code after WRITE is executed.

## **MARK**

Purpose: Write a tapemark.

Format: MARK ( IOD-name, ERR: sn)

Description: The TAPE is positioned to the interrecord gap following the tapemark.  
ERRCODE and STATUS receive the completion code after MARK is executed.

## **BACKSPACE**

Purpose: Move the tape back one record or one tapemark.

Format: BACKSPACE ( IOD-name, ERR: sn)

Description: The BACKSPACE operation positions the tape back one record (or tape mark).  
ERRCODE and STATUS receive the completion code after BACKSPACE is executed.

## **REWIND**

Purpose: Position the tape to load-point.

Format: REWIND ( IOD-name, ERR: sn)

Description: The REWIND operation is described in detail in Section 6.

## **REWINDLOCK**

Purpose: Position the tape to load-point and make the tape unavailable for use.

Format: REWINDLOCK ( IOD-name, ERR: sn)

Description: The REWINDLOCK operation is described in detail in Section 6.

## **CLOSE**

Purpose: Release access to the magnetic tape drive.

Format: CLOSE ( IOD-name, buffer, ERR: sn)

Description: When the tape file is closed, the leftmost bit of the STATE keyword is reset to zero (0) to indicate that the file is no longer OPEN.  
ERRCODE and STATUS receive the completion code after execution of CLOSE.



## **SKIPFILE**

Purpose: To position a magnetic tape beyond the next encountered tape mark.

Format: SKIPFILE (IOD-name, ERR:sn)

Description: The SKIPFILE operation is described in detail in Section 6.

NOTE: Use of a data recorder as the magnetic tape drive is the same as for standard TAPE except that the only applicable operations are: OPEN, CLOSE, READ and WRITE.

Keywords used or referenced in I/O operations for TAPE are:

ERRCODE  
OUTLEN  
STATE  
STATE  
STATUS  
UNIT  
XFERLEN

## **PRINTER I/O OPERATIONS**

There are two modes of controlling forms on the printer I/O device: Basic and VFU. The basic mode is the default value and is used for any file that has not been explicitly opened for VFU use. The VFU mode is explained in this subsection on subsequent pages.

Applicable operations for PRINTER in either mode are as follows:

### **OPEN**

Purpose: Obtain access to the printer.

Format: OPEN ( IOD-name, buffer, ERR: sn)

Description: When OPEN is executed, the printer denoted by IOD-name is made available for data transfer. The leftmost bit of the STATE keyword value is set to one (1) to indicate that the printer is OPEN. Also, bit 6 is read to establish the state of the device (see the STATE keyword under "KEYWORDS USED IN BOTH THE DATA DEFINITION AND THE EXECUTION SECTION" on previous pages of this section). Bit 0 is read to determine the source of SLEW information for individual print operations.

ERRCODE and STATUS receive the completion code after execution of OPEN.

### **CHECKFORMS**

Purpose: Obtain the current printer control and positioning information.

Format: CHECKFORMS ( IOD-name, ERR: sn)

Description: The CHECKFORMS operation is described in detail in Section 6.

### **SETFORMS**

Purpose: To reset active printer control parameters.

Format: SETFORMS ( IOD-name, ERR: sn)

Description: The SETFORMS operation is described in detail in Section 6.

## **PRINT**

Purpose: To activate printing with forms control.

Format: PRINT ( IOD-name, buffer, EOF: sn-1, ERR: sn-2)

Description: If, during a PRINT or WRITE operation, the slew value chosen causes the line position to exceed the bottom of a form (lines per form), an end-of-form is generated. The PRINT operation is performed before the EOF exit is taken. The slew value supplied by the application program is not changed.

The POSITION parameter is adjusted to reflect the relative line number following execution of PRINT.

In a statement prior to PRINT, .OUTLEN may be assigned a value to override the buffer length for one PRINT operation.

ERRCODE and STATUS receive the completion code after execution of PRINT.

## **WRITE**

Purpose: To activate printing with forms control.

Format: WRITE ( IOD-name, buffer, EOF: sn-1, ERR: sn-2)

Description: The WRITE operation is identical to the PRINT operation.

## **CLOSE**

Purpose: Release access to the printer.

Format: CLOSE ( IOD-name, buffer, ERR: sn)

Description: When CLOSE is executed, the leftmost bit of the .STATE parameter value is reset to zero (0) to indicate that access to the printer is no longer available.

ERRCODE and STATUS receive the completion code after execution of CLOSE.

## VFU (VERTICAL FORMS UNIT)

The VFU mode is used when the forms control information as well as the print data is contained in the buffer. The forms control information (slew value) is kept in the first position of the data buffer. In the standard mode, the forms control information is contained in the IOD (.SLEW keyword) and the buffer contains print data only.

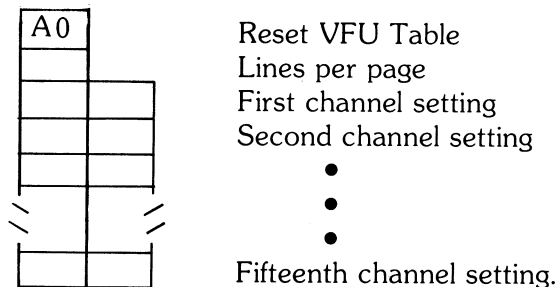
Prior to OPEN time, selection of the VFU mode is made by setting the rightmost bit of the .STATE parameter. The selection remains active until a CLOSE is executed on the device.

In the VFU mode, the program can skip or slew lines before or after PRINT. Slewing is to move paper a specified number of lines relative to the current position. Skipping to a channel is controlled on many printers by a carriage control tape which terminates paper motion upon detection of a punched hole in the designated channel. With the Series 21 System, the carriage control tape is simulated to support channels 1 through 12 for 132 lines.

The bit encoding for the slew value is:

011n	nnnn	Skip to Channel N - then print
010n	nnnn	Slew N lines - then print
100n	nnnn	Print - then skip to Channel N
000n	nnnn	Print - then Slew N lines
101x	xxxx	Reset the VFU Table

The format of the buffer used to reset the VFU Table is shown below:



The first byte of the VFU Table contains the value X'A0' to indicate that the remaining contents of the buffer specify printing control information. The second byte contains the value for lines per page (the total number of lines between perforations on the paper). The subsequent 15 byte-pairs describe the channel settings. Each channel setting consists of a channel number (1-12) and the corresponding line number (1 to lines per page).

Subsequent to establishing values for the VFU Table, slew values in the data buffer are interpreted accordingly. Whenever a slewing operation causes the paper to exit a line containing either Channel 9 or Channel 12, a skip to Channel 1 (top-of-form) occurs.

Example: The example below illustrates opening the printer for VFU mode and establishing particular VFU Table values.

```
*
* SAMPLE PROGRAM SEGMENT TO MODIFY VFU TABLE
*
IOD: LIST = PRINTER;           DEFINE PRINT DEVICE
RCD: VFUTBL                    RESET VFU TBL SLEW VALUE
    CONTROL(1)=X'A0'           SET LINES PER PAGE = 66
    LINESPP(1)=X'42'           SET CHANNEL 1 TO LINE 1 (TOP-OF-FORM)
    DEF1 (2)=X'0101'           SET CHANNEL 2 TO LINE 7
    DEF2 (2)=X'0207'           SET CHANNEL 3 TO LINE 16
    DEF3 (2)=X'0310'
    .
    .
    .
DEF12 (2)=X'0C42';             SET CHANNEL 12 TO LINE 66 (BOTTOM OF
                                FORM)
    .
    .
    .
START
    .
    .
    .
LIST.STATE=X'41'              SET EXCLUSIVE USE/VFU MODE
OPEN(LIST,BUFFER,ERR:999)     OPEN PRINT DEVICE
PRINT(LIST,VFUTBL,EOM:810,ERR:999) REPLACE VFU TABLE
    .
    .
    .
END
```

Keywords used or referenced in I/O operations for PRINTER are:

```
ERRCODE
FORMS
OUTLEN
POSITION
SLEW
STATE
STATUS
UNIT
```

## COMPATIBLE CHANNEL I/O OPERATIONS

Compatible Channel (COMP CHAN) allows the interchange of data between the Series 21 System and the MDS System 2400.

Using this channel as an interface, the Series 21 System acts as an I/O device for the System 2400. The System 2400 views the Series 21 as a magnetic tape drive. For data interchange to successfully occur, the System 2400 must be operating under the control of a program whose operations are compatible with the MOBOL program that is directing the Series 21; that is, operations between the two must correspond. For example, when the 2400 is executing a READ, the corresponding operation for the Series 21 is a WRITE.

Both the Series 21 and the System 2400 issue error and status information. Error information is passed between each system and its corresponding application program. Status information is exchanged between the Series 21 and the System 2400.

When the System 2400 issues a command to an I/O device, it receives two status values: 'Oldstatus' and 'Newstatus'. Oldstatus reflects the result of the last operation. Newstatus reflects the ability of the device to carry out the current operation.

On the Series 21, Newstatus is determined by the system. When transmitted to the System 2400, it will reflect READY and WRITE ENABLED. Oldstatus is maintained by the system and passed to the MOBOL application keyword .XFERSTATUS.

If required, this parameter may be modified by the application program by referencing .XFERSTATUS prior to the next COMP CHAN operation.

XFERSTATUS has the following bit assignments which are compatible with Oldstatus and Newstatus:

### Byte 1

2<sup>7</sup> - INTERRUPT  
2<sup>6</sup> - NOT BUSY  
2<sup>5</sup> - BUSY  
2<sup>4</sup> - TAPE MARK DETECTED  
2<sup>3</sup> - DATA CHECK  
2<sup>2</sup> - EQUIPMENT CHECK  
2<sup>1</sup> - OVERRUN/RUNAWAY  
2<sup>0</sup> - READY

### Byte 2

2<sup>7</sup> - SHORT TRANSFER  
2<sup>6</sup> - 7 CHANNEL  
2<sup>5</sup> - WRITE ENABLED  
2<sup>4</sup> - RUNAWAY  
2<sup>3</sup> - REWINDING  
2<sup>2</sup> - EOM DETECTED  
2<sup>1</sup> - LOADPOINT  
2<sup>0</sup> - LOCAL

Applicable operations for COMP CHAN are:

### **OPEN**

Purpose: To obtain access to a compatible channel for the purpose of data interchange with the MDS 2400.

Format: OPEN ( IOD-name, buffer, ERR: sn)

Description: When the COMP CHAN dataset is OPENed, the leftmost bit of the .STATE parameter is set to one indicating the open state of the dataset. ERRCODE and STATUS receive the completion code after OPEN is executed. Newstatus and the parameter .XFERSTATUS are both initialized to reflect READY, LOADPOINT and WRITE ENABLED. (The system 2400 will continue to receive a status of LOCAL until the next Series 21 is issued.)

## READ

Purpose: To transfer data from the compatible channel to the Series 21 System.

Format: READ(IOD-name, buffer, ERR:sn-1, EOF:sn-2)

Description: Oldstatus is updated from .XFERSTATUS and the Series 21 is available for a System 2400 command to be received. The next action executed depends upon the System 2400 command received.

<b>2400 Command</b>	<b>Action</b>
WRITE	<p>Data is transferred through the channel to the application buffer.</p> <p>.XFERSTATUS is set to reflect normal WRITE termination (the System 2400 continues to receive a busy status until the next Series 21 operation is issued).</p> <p>Newstatus is updated to cancel the LOAD-POINT status bit. .XFERLEN contains the length of the data record just read by the Series 21.</p> <p>After the READ, control is returned to the next inline statement.</p>
WRITE TAPE MARK	<p>.XFERSTATUS is set to the normal WRITE TAPE MARK termination (the System 2400 continues to receive a busy status until the next Series 21 operation is issued).</p> <p>Newstatus is updated to cancel the LOAD-POINT status bit and control is returned to the EOF:sn-2 exit.</p>
READ	<p>The Series 21 READ is suspended and control is returned to the ERR:sn-1 exit.</p> <p>(If the Series 21 executes a WRITE operation in response to this exit, the System 2400 operation receives the data from the application buffer and the .XFERSTATUS parameter is set to reflect normal READ termination. If the Series 21 executes any other operation, processing begins as described for that operation. The System 2400 READ operation is cancelled. Subsequent System 2400 activity is application dependent.)</p>
Other Commands (Or No Command i.e., Time Out)	<p>ERRCODE will indicate the command received from the System 2400 and control is returned to the ERR:sn-1 exit.</p>

## SENDEOF

Purpose: To transfer an end-of-file indicator (tape mark) to the compatible channel.

Format: SENDEOF ( IOD-name, ERR:sn)

Description: Oldstatus is updated from .XFERSTATUS and the Series 21 is available for a System 2400 command to be received. The next action executed depends upon the System 2400 command received:

<b>2400 Command</b>	<b>Action</b>
READ	<p>The Series 21 sends the standard EOF data block (which is a special one-byte record) to the System 2400. .XFERSTATUS is set to reflect TAPE MARK DETECTED; this combination will cause the System 2400 to recognize an end-of-file condition. (The system 2400 continues to receive a busy status until the next Series 21 operation is issued.)</p> <p>Newstatus is updated to cancel the LOADPOINT status bit.</p>
WRITE	<p>The Series 21 SENDEOF is suspended and control is returned to the ERR:sn-1 exit.</p> <p>(If the Series 21 executes a READ operation in response to this exit, the System 2400 operation transmits its data to the application buffer and the .XFERSTATUS parameter is set to reflect normal WRITE termination. If the Series 21 executes any other operation, processing begins as described for that operation. The System 2400 WRITE operation is cancelled. Subsequent System 2400 activity is application dependent.)</p>
WRITE MARK	<p>The Series 21 SENDEOF is suspended and control is returned to the ERR:sn-1 exit.</p> <p>(If the Series 21 executes a READ operation in response to this exit, the System 2400 operation is considered complete and the EOF:sn-2 exit associated with the Series 21 READ is taken. If the Series 21 executes any other operation, processing begins as described for that operation. The System 2400 WRITE TAPEMARK operation is cancelled. Subsequent System 2400 activity is application dependent.)</p>
Other Commands (Or No Command, i.e., Time Out)	<p>Control is returned to the ERR:sn-1 exit.</p> <p>ERRCODE will indicate the command received from the System 2400 and control is returned to the ERR:sn-1 exit.</p>



## WRITE

Purpose: To transfer data to the compatible channel from the Series 21 System.

Format: WRITE (IOD-name, buffer, ERR:sn-1, EOF:sn-2)

Description: In a statement prior to WRITE, .OUTLEN may be used to override the buffer length for one WRITE operation.

OLDSTATUS is updated from .XFERSTATUS and the Series 21 is available for a System 2400 command to be received. The next action executed depends upon the System 2400 command received:

<b>2400 Command</b>	<b>Action</b>
READ	<p>Data is transferred through the channel from the application buffer.</p> <p>.XFERSTATUS is set to reflect normal READ termination (the System 2400 continues to receive a busy status until the next Series 21 operation is issued).</p> <p>Newstatus is updated to cancel the LOAD-POINT status bit.</p> <p>.XFERLEN will be updated to indicate the length of the data record just transferred and control is returned to the next in-line statement.</p>
WRITE	<p>The Series 21 WRITE is suspended and control is returned to the ERR:sn-1 exit.</p> <p>(If the Series 21 executes a READ operation in response to this exit, the System 2400 operation transmits its data to the application buffer and the .XFERSTATUS parameter is set to reflect normal WRITE termination. If the Series 21 executes any other operation, processing begins as described for that operation. The System 2400 WRITE operation is cancelled. Subsequent System 2400 activity is application dependent.)</p>

2400 Command	Action
WRITE TAPE MARK	<p>The Series 21 WRITE is suspended and control is returned to the ERR:sn-1 exit.</p> <p>(If the Series 21 executes a READ operation in response to this exit, the System 2400 operation is considered complete and the Series 21 application EOF:sn-2 (associated with the READ) exit is taken. If the Series 21 executes any other operation, processing begins as described for that operation. The system 2400 WRITE TAPEMARK operation is cancelled. Subsequent System 2400 activity is application dependent.)</p>
Other Commands (Or No Command i.e., Time Out)	<p>Control is returned to the ERR:sn-1 exit. ERRCODE will indicate the command received from the System 2400 and control is returned to the ERR:sn-1 exit.</p>

## CLOSE

Purpose: To release access to the compatible channel.

Format: CLOSE ( IOD-name, buffer, ERR:sn )

Description: OLDSTATUS is updated from .XFERSTATUS and Newstatus is updated to show LOCAL. The Series 21 is available to allow the System 2400 to take status. ERRCODE receives the completion code after execution of CLOSE.

When the compatible channel is closed, the leftmost bit of the .STATE keyword is reset to zero.

MOBOL keywords used by the Series 21 System in I/O operations for COMP CHAN are:

ERRCODE  
OUTLEN  
STATE  
STATUS  
XFERLEN  
XFERSTATUS



## SECTION 8: STATION I/O OPERATIONS

The STATION is an interactive I/O device consisting of a keyboard and a video display screen (CRT). Station operations may be directed by MOBOL application programs to perform the following functions:

1. Solicit data from the keyboard.
2. Format the entry of data using screen displays, guide messages, etc.
3. Check correctness of entered data by comparing to predefined parameters.
4. Report run-time status during execution of application programs.
5. Accept operational information from the keyboard operator (e.g., SEL MODE).
6. Display error messages and obtain responses (and differentiate these responses from normal entry).
7. Read data on the display screen and transform the images into data that can be processed by the application program.

The components involved with station operations may be divided into two categories:

1. Structural or definition components which define parameters for the station device and define the format for data to be entered through the keyboard. These components appear in the Data Definition Section. They are:

IOD

KET

2. Operational or execution components which direct operations of the station during program execution. These components appear as statements in the Execution Section. They are:

KENTER          RESUMERR          READSCREEN

KVERIFY          ERROR          READKEY

RESUME                          NOTIFY

Keywords associated with these components are described in Section 7.

## **KEYWORDS USED IN THE DATA DEFINITION SECTION**

The keywords that may be used in the Data Definition Section in the IOD statements for the station relate to error message text. This text is used in the error-sequence associated with the internally detected keying errors.

When an error is detected and the corresponding message has been established by the IOD, the message text is displayed on line 2. Also, the Error Tone/Flasher is activated; RESET must be keyed to acknowledge the message. Following depression of RESET, the original line 2 display is restored and operation continues. (For a description of the resulting message display, see the ERROR statement later in this section.)

When the corresponding message has not been established by the IOD, the Error Tone/Flasher is activated and RESET must be keyed to acknowledge the message.

An error message may be given a display value by means of a keyword assignment in the IOD. The format is:

KEYWORD = alphanumeric string

Each keyword and the condition under which its value is displayed are tabled below:

<b>KEYWORD</b>	<b>ERROR CONDITION</b>
BOUND	Incorrect key used to release a field (e.g., SKIP keyed for an EXIT field).
FILTER	Incorrect type of data keyed for a field (e.g., an alphabetic character keyed for a NUMERIC field).
KEYBOARD	Hardware error, queue overflow, unassigned key or invalid dead key sequence (e.g., the application is more than 15 keystrokes behind the operator when the 16th key is struck).
MCSEQ	Multi-code error (i.e., the first or second key after depression of multi-code is not a hexadecimal digit).
MISCOMP	Character miscompare during a field verify operation.
OPERATIONAL	Inappropriate control or data key (e.g., data keyed when the cursor is positioned between two fields in screen mode).
SIGNIF	Incorrect number of significant data characters present when field release is attempted.

The length of the string assigned to a keyword is limited to 38 characters.

Example:

IOD: CRT = STATION

·  
·  
·

MCSEQ = 'MULTI-CODE ERROR'

·  
·  
· ;

Keywords appearing in the KET statement are presented under context specifications for the KET in Section 5.

### KEYWORDS USED IN THE EXECUTION SECTION

Keywords that may be used in the Execution Section relate to the status and parameters of station operations. The format for a reference is:

**IOD-name • keyword**

<b>KEYWORD</b>	<b>DEFINITION</b>
KETNUM	A one-byte binary number designating the currently active KET. The first defined KET in the Data Definition Section is numbered zero (X'00') and each subsequently defined KET is assigned a number in sequence.
NOTE TYPE	Each is a one-byte binary number corresponding to the TYPE and NOTE values associated with the currently active KET. TYPE and NOTE values may be used to represent anything desired or required by the programmer.
FLDCOUNT	A one-byte binary number designating the number of fields within the currently active KET.
FLDNUM	A one-byte binary number designating the currently active field of the currently active KET. The first field of the KET is always assigned a zero (0); each subsequently defined field within the KET is assigned a number in sequence.
CURFLD	An alternate name for the currently active field of the currently active KET. See the RESUME statement for details.

**KEYWORD****DEFINITION**

**SHIFT** A one-byte binary number designating the preferred shift. The value normally reflects the preferred shift for the current field but may be set by the application as a parameter for the READKEY operation. SHIFT is described in detail in Section 5 of this manual. The shift parameter encoding as tabled:

PREFERRED SHIFT	IOD PARAMETER
LETTER	BINARY ZERO (X'00')
DIGIT	BINARY ONE (X'01')
TEXT	BINARY TWO (X'02')

**KEYSTROKE** A three-byte binary number which is incremented by one for each keystroke.

**MATCH** A three-byte binary number which is incremented by one for each data key that results in no change to the current data character.

**CHANGE** A three-byte binary number which is incremented by one for each data key that results in a change to the current data character.

**CONTROL  
NUMPAD  
NOLABEL  
XCONTROL** These status parameters are set when a station operation is interrupted for application processing. Each parameter is one-byte in length. The value of the control key, if any, that caused the interruption is recorded in its appropriate keyword parameter. The other parameters will contain a code of X'FF' indicating that they do not record the control key value.

**LINE** A one-byte binary number used in conjunction with READSCREEN. The initial-value is one.

**UNIT** A one-byte value designating the source for key data. The values *currently* assigned signify that the source for key data is constant for the execution of the application and designate the appropriate station as tabled:

UNIT Value	Source for Key Data
X'01'	Station No. 1
X'02'	Station No. 2
X'03'	Station No. 3
X'04'	Station No. 4

## KENTER

Purpose: To activate a KET for key entry.

Format: KENTER (IOD-name, KET-name, ESC:sn-1, ERR:sn-2)

Description: The KET specified by KET-name is activated and the key entry process is executed in accordance with the specifications contained in KET-name. The IOD-name must designate the STATION device.

When the KENTER statement is executed, display and keying operation are primarily controlled by a key entry process which is defined by the active KET. This process is outlined as follows:

1. The KETNUM, TYPE and NOTE keyword fields of the IOD are transcribed from the corresponding fields of the KET.
2. The CRT is reconfigured, if necessary, to correspond to the CRTSIZE keyword setting of the KET. The section of the display described by the BLANK keyword of the KET is space filled.
3. The operator prompting messages are displayed.
4. The contents of all data fields are displayed.
5. The FLDNUM keyword is set to zero. (If the clause INIT=ESCAPE is coded in the KET, the initial-value of FLDNUM may be set by the application. The statement in the ESC clause is executed at this point so that a specific field may be designated as the initial field.) Beginning with the field identified by FLDNUM, each data field is selected for processing in the order in which the field appears in the KET. The MODE parameter of the associated KET statement determines which of the two variations of the key entry process is to be employed. See Field Mode and Screen Mode on subsequent pages.

When the last field has been selected and processed, execution of the KENTER statement is complete and control passes to the next statement. Transfer to the next statement occurs immediately if the RELEASE=AUTOMATIC clause is present in the KET; otherwise, ENTER must be pressed.

During the execution of the KENTER statement, the key entry process may be interrupted and control may be transferred to a labeled statement. This can be effected either by specifying a field post-processing or pre-processing routine in the KET, or the operator can signal an escape request which causes a transfer of control to the associated statement labeled sn-1.

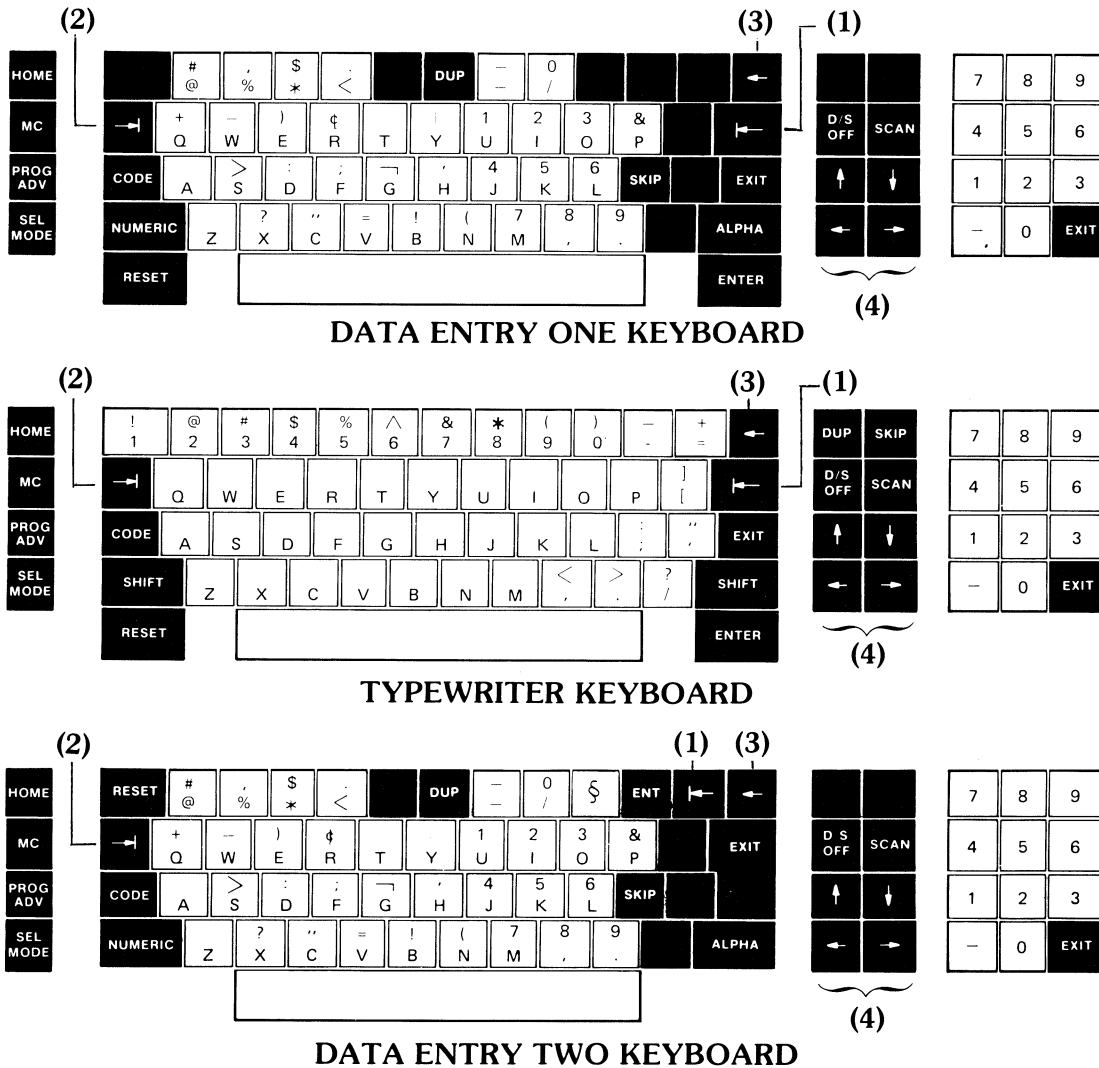
The operator signals the intention to escape by pressing certain labeled control keys, a numeric pad key when the NUMPAD=ESCAPE clause is present in the KET, or any unlabeled key. These keys are called user-defined keys. When control passes to the statement labeled sn-1, the CONTROL, NUMPAD, or NOLABEL keyword indicates which user-defined key caused the escape. If the "ESC:sn-1" clause is omitted from the KENTER statement, depression of any user-defined key is ignored and the execution of the key-entry process is continued.

Control can be returned to an interrupted execution of the key-entry process using the RESUME or RESUMERR statement.



The action of both system and user-defined control keys is presented in Tables 8-1 and 8-2 on subsequent pages.

If the KENTER statement cannot be executed due to parameter errors or logically inconsistent operation sequences (e.g., directing that execution be RESUMED at a non-existing field), an error condition results and the ERR exit is taken. If no ERR exit is coded, execution of the program is cancelled.



**Figure 8-1: Identification Of Operator STATION Control Keys**

Operator STATION symbolic control keys are identified as follows for all keyboards: .

1.     ← = FIELD BACKSPACE
2.     → = FIELD FORWARD
3.     ← = CHARACTER BACKSPACE
4.     = CURSOR CONTROL KEYS:
 

A	↓	A.	DOWN
B	↑	B.	UP
C	→	C.	RIGHT
D	←	D.	LEFT

## FIELD MODE

When the MODE=FIELD clause is specified in the KET, the order of field selection is determined primarily by the ordering of the field-specifications within the KET or by explicit program direction.

The program may interact with the key entry process by using field processing routines identified in the KET. A field post-processing routine, designated within a field specification by "sn-post", is executed following the completion of field key entry to allow program examination of the field. A field pre-processing routine, designated by "sn-pre" is executed upon re-selection of a field in anticipation of re-keying the field (e.g., Field Backspace).

Field completion is operator-initiated by the depression of one of the following:

1. An appropriate field-release key (SKIP, EXIT or -);
2. A data key in the rightmost position of a field for which mandatory boundary checking is not specified; or,
3. The Field Forward key.

The field post-processing routine (sn-post) is then executed, provided that the following field validation tests are successful:

1. Each character within the field is of the proper character class; and,
2. The field contains the required number of significant characters.

Otherwise, the key entry process signals an error to the operator by means of the error-sequence and the appropriate error message text.

Field post-processing routines can:

1. Inspect the completed field to accept or to reject the field according to application-dependent criteria.
2. Alter or generate the contents of related KET fields.
3. Perform application-related computations (e.g., increment an accumulator).

Field re-selection is operator-initiated by the depression of the Field Backspace key when the cursor is at the first position of a field. The field pre-processing routine (sn-pre) is then executed for the preceding field (provided that such a field exists).

Field pre-processing routines can:

1. Accept or reject the operator implied intention to re-key the field.
2. Alter the contents of related KET fields.
3. Perform application-related computations (e.g., decrement an accumulator).

An omitted field-processing-routine designator for the field signifies that no program interaction is required for the corresponding event.

Because of the serial nature of field selection, either forward or backward, the *operator* may not select fields for entry in an order which comprises the integrity of balanced sn-post and sn-pre field processing exits.

Two macro control keys are provided to facilitate operator-initiated restart or premature termination: HOME and ENTER. HOME corresponds to the number of repeated depressions of the Field Backspace key ( ← ) required to position the cursor to the first field of the KET which allows key entry (i.e., not a PROTECTED field).

ENTER corresponds to the number of repeated depressions of the Field Forward key ( → ) required to position the cursor after the last field of the KET; the next statement following the KENTER statement is then executed.

When the HOME or ENTER keys are used, the field processing routines are activated (as described above) for each field affected by the positioning. The macro control-key sequence terminates prematurely if one of the following occurs:

1. A field validation test is failed;
2. The field processing routine rejects the action (i.e., issues a RESUMERR); or,
3. The field processing routine explicitly selects a field out of sequence (i.e., a directed RESUME).

The four cursor control keys ( ↓↑→← ) and the Character Backspace key ( ← ) can be used to position the cursor within a field for individual character correction. Any depression of these keys which would result in the cursor being positioned outside of the field is ignored.

## SCREEN MODE

When the `MODE=SCREEN` clause is specified in the `KET`, the order of field selection is determined primarily by the ordering of the field-specifications within the `KET`. Operator-controlled cursor positioning or explicit program direction may be used to alter the order of field selection.

The program may interact with the key entry process by using field processing routines identified in the `KET`. In a manner similar to `FIELD` mode, the post-processing routine (`sn-post`) is executed following completion of field key entry. The field pre-processing routine (`sn-pre`) is executed upon re-selection of a field in anticipation of re-keying the field.

In `SCREEN` mode, the criteria for completion and re-selection differ slightly from those used in `FIELD` mode.

In `SCREEN` mode, the operator may position the cursor anywhere on the display whenever a field is not selected. A field is selected from the time that field re-selection takes place (described below) until field completion occurs (also described below).

Field re-selection is operator-initiated by the depression of a:

1. Data key;
2. Field release key (`SKIP`, `EXIT`, or `-`); or,
3. User-defined key.

In these cases, the cursor must be positioned within a field. The pre-processing routine (`sn-pre`) for the field is executed when one of the above keys is first depressed within a field. The re-selection key itself is retained in anticipation of the response from the field pre-processing routine.

The field pre-processing routine can:

1. Either accept or reject the intention to re-key the field.
2. Alter the contents of related `KET` fields.
3. Perform application-related computations.

Because the re-selection key has not yet caused the field contents to be modified, the field pre-processing routine can access the original field contents. If the field pre-processing routine rejects the re-selection, the original re-selection key is not further processed.

Field completion is possible only if a field has been successfully re-selected and has not already been completed. In this case, field completion is operator-initiated by the depression of one of the following:

1. An appropriate field release key (SKIP, EXIT, or -);
2. A data key in the rightmost position of a field for which mandatory boundary checking is specified;
3. The Field Forward key (  $\rightarrow$  );
4. The Field Backspace key (  $\leftarrow$  ), if the cursor is at the first position of the field; or,
5. A cursor control key (  $\downarrow$   $\uparrow$   $\rightarrow$   $\leftarrow$  ) which would result in a new cursor position outside the field display area.

The field post-processing routine (sn-post) is then executed, provided that the following field validation tests are successful:

1. Each character within the field is of the proper character class; and,
2. The field contains the required number of significant characters.

The field post-processing routine can:

1. Inspect the completed field and either accept or reject the field according to application-dependent criteria.
2. Alter or generate the contents of related KET fields.
3. Perform application-related computations.

An omitted field-processing-routine designator for the field signifies that no program interaction is required for the corresponding event.

The operator may not compromise the integrity of balanced sn-pre and sn-post field processing exits, even though the fields can be selected in a non-serial order.

When no field is currently selected and the cursor is moved through a field as an intermediate step in using the cursor positioning keys or field positioning keys, the field processing routines for these fields are not executed. The operator must explicitly re-select the field as described earlier.

Two macro control keys are provided to facilitate operator-initiated restart or premature termination: HOME and ENTER. HOME corresponds to the repeated depression of the field backspace key (  $\leftarrow$  ) required to position the cursor to the first field of the display.

ENTER corresponds to the repeated depression of the Field Forward key (  $\rightarrow$  ) required to position the cursor after the last field of the display; the next statement following the KENTER statement is then executed.

When the HOME or ENTER keys are used, the field not processed routine and field validation tests are performed (as described for FIELD mode) only for the originating field (if it is re-selected but not yet completed). The macro control key sequence is terminated prematurely in the same way as described for FIELD mode.

## CONTROL KEYS

Table 8-1 describes all labeled control keys. Unlabeled keys are user-defined. Table 8-2 indicates the values returned in the CONTROL, NUMPAD, NOLABEL and XCONTROL keywords which may be used in field processing routines.

**TABLE 8-1: Labeled Control Keys**

KEY	KENTER	KVERIFY
CODE	Pressed simultaneously with another key to modify its meaning. Produces a 'Third Shift' capability when used with data keys.	Same as KENTER
DUP CODE/DUP	User-defined	User-defined
DS OFF CODE/DS OFF	User-defined	User-defined
ENTER	Used to acknowledge completion of the active KET and signal a release. Also serves as a macro control key which performs the analogous function of repeated depression of the Field Forward key until the cursor is extinguished (after the last field) and a release is signaled.	Same as KENTER
CODE/ENTER	User-defined	User-defined
EXIT (LEFT ZERO OR RIGHT ADJUST	Exit key which causes data to be adjusted to the right boundary with shift specified (zero or space) fill characters to the left. The '-' (minus) key is used in lieu of 'EXIT' for negative numbers.	Same as KENTER; while verifying, the cursor stays in the first position of the field until the first non-fill data key is pressed. The fill characters are compared; then the first data character keyed is compared with the first non-fill character. If neither agree, an error occurs. When fill characters are in error, the cursor is at the start of the field.

**TABLE 8-1: Labeled Control Keys (Cont'd.)**

KEY	KENTER	KVERIFY
FIELD BACKSPACE	Screen Mode — When the cursor is between two fields, the Field Backspace key provides cursor movement to the first position of the nearest field to the left (or above) the current cursor position. When the cursor is within a field, the operation is identical to field mode. Field Backspace is ignored if cursor is in the first display position of the first field of the KET.	
CODE/FIELD BACKSPACE	User-defined	User-defined
CHARACTER CORRECT	<p>Provides a destructive (space fill) cursor movement one position to the left. Character is ignored when the cursor is in the first position of a field.</p> <p>Screen Mode — Allows cursor positioning, a character at a time.</p>	<p>Same as KENTER: Initiates a verified character correction sequence.</p> <p>Ignored</p>
CURSOR POSITION	Field Mode — Same as Screen Mode except ignored if resulting cursor position lies outside of the current field.	
EXIT (LEFT ZERO OR RIGHT ADJUST)		<p>When data does not match, the cursor is at the first non-fill character.</p> <p>After verifying the last character, either 'EXIT' or '-' is pressed. 'EXIT' and the '-' key also verify the sign in the case of a numeric field. If a verification error occurs while verifying the sign, a 'SIGN' error-sequence is initiated.</p>
HOME	Serves as a macro control key. It performs the analogous function of repeated depression of the Field Backspace key until the cursor is in the first field (of the KET) that is not a protected field.	Same as KENTER, except only fields marked Verify are considered. The verified field correction sequence is associated only with the final field.

**TABLE 8-1: Labeled Control Keys (Cont'd.)**

<b>KEY</b>	<b>KENTER</b>	<b>KVERIFY</b>
CODE/HOME	User-defined	User-defined
MC (MULTICODE)	The MC key provides a method of keying any of 256 EBCDIC codes. EBCDIC codes are entered by pressing 'MC' and then keying two hexadecimal digits. Valid hexadecimal digits are: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Non-displayable graphics will appear as a multiple slash character on the display.	Same as KENTER: the character is verified after its last keystroke.
PROG ADV CODE/PROG ADV	User-defined	User-defined
RESET	Used to acknowledge an error condition.	Used to acknowledge an error condition.
CODE/RESET	User-defined	User-defined
SCAN CODE/SCAN	User-defined	User-defined
SEL MODE CODE/SEL MODE	User-defined	User-defined
SKIP	Field Exit key which causes data to remain aligned with the left boundary and remaining unkeyed positions to be space filled. When Skip boundary checking is mandatory, failing to exit with the 'SKIP' key results in a 'BOUNDARY' error-sequence.	Same as KENTER; pressing 'SKIP' during verification causes the remainder of the field to be checked for spaces. If a non-space character is encountered, the cursor stops in that position and a 'MISCOMPARE' error-sequence is initiated. After pressing 'RESET', the non-space character can be replaced with a space by pressing 'SKIP'. KVERIFY, then, continues to check the remainder of the field for spaces.
FIELD FORWARD	Field Mode — Moves the cursor, left-to-right, to the first position of the next data field (in the KET) whose source is key. If pressed in the last field, the cursor is extinguished and subsequent depressions are ignored.	Field Escape; the verification Compare Test is considered successful for the current field.



**TABLE 8-1: Labeled Control Keys (Cont'd.)**

KEY	KENTER	KVERIFY
FIELD FORWARD (cont'd.)	Screen Mode — When the cursor is within a field, the operation is identical to field mode. When the cursor is between two fields, the Field Forward key moves the cursor to the first position of the next key field to the right (or below) the current position. If pressed in the last field, cursor is extinguished and subsequent depressions are ignored.	
CODE/FIELD FORWARD	User-defined	User-defined
FIELD BACKSPACE	Field Mode — When the cursor is located in the first position of a field, this key provides cursor movement to the first position of the first non-protected field previously specified. If the cursor is originally located in other than the first position of the field, this key provides cursor movement to the first position of the current field. Field Backspace is ignored if the cursor is in the first position of the first non-protected field.	Same as KENTER: Initiates a verified field correction sequence.

**TABLE 8-2: Values Returned In CONTROL, NUMPAD, NOLABEL And XCONTROL**

KEY	KEYWORD	HEXADECIMAL VALUE	
		WITHIN FIELD	OUTSIDE FIELD
SEL MODE	CONTROL	00	80
PROG ADV	CONTROL	01	81
SCAN	CONTROL	02	82
DUP	CONTROL	03	83
DS OFF	CONTROL	04	84
CODE/SEL MODE	CONTROL	05	85
CODE/PROG ADV	CONTROL	06	86
CODE/SCAN	CONTROL	07	87
CODE/DUP	CONTROL	08	88
CODE/DS OFF	CONTROL	09	89
CODE/RESET	CONTROL	0A	8A
CODE/ENTER	CONTROL	0B	8B
CODE/	CONTROL	0C	8C
CODE/HOME	CONTROL	0D	8D
CODE/	CONTROL	0E	8E
NONE	CONTROL	FF	FF
0	NUMPAD	10	90
1	NUMPAD	11	91
2	NUMPAD	12	92
3	NUMPAD	13	93
4	NUMPAD	14	94
5	NUMPAD	15	95
6	NUMPAD	16	96
7	NUMPAD	17	97
8	NUMPAD	18	98
9	NUMPAD	19	99
NONE	NUMPAD	FF	FF
INSTALLATION DEFINED	NO LABEL	20 - 2E	A0 - AE
NONE	NO LABEL	FF	FF
→	XCONTROL	30	B0
←	XCONTROL	31	B1
HOME	XCONTROL	32	B2
ENTER	XCONTROL	33	B3
↑	XCONTROL	34	B4
↓	XCONTROL	35	B5
←	XCONTROL	36	B6
↔	XCONTROL	37	B7
↖	XCONTROL	38	B8
(INIT=ESCAPE)	XCONTROL	39	B9
NONE	XCONTROL	3F	FF

## KVERIFY

Purpose:

To activate a KET for key verify

Format:

KVERIFY (IOD-name, KET-name, ESC:sn-1, ERR:sn-2)

Description:

The KET specified by KET-name is activated and the key-verification process is executed in accordance with the specifications contained in KET-name. The IOD-name must designate the STATION device.

When the KVERIFY statement is executed, display and keying operations are primarily controlled by a key-verification process which is defined by the active KET. The process is identical to KENTER with the following exceptions:



1. Only fields for which VERIFY is specified in the KET are acted upon by the key-verification process unless a directed RESUME or RESUMERR specifically indicates that a field is to be verified.
2. FIELD mode is always employed even if the MODE=SCREEN clause is present in the KET.
3. The contents of all data fields are initially displayed. After the first data key is depressed, those verify fields defined later in the KET are blanked on the display pending their verification.
4. Keying into a field does not alter the field contents (except as described below), instead the keyed data is used as a source for comparison with the field content.

During key verification, each character is compared with the corresponding character in the data field.

If the characters do not compare, a miscompare error is posted as follows:

1. The MISCOMP error message is displayed on line 2;
2. The cursor remains at the position in question;
3. All verify fields are displayed.

The RESET key is used to acknowledge the error condition and restore the display to the pre-error state. The first character following RESET is accepted if the character keyed agrees with either the character in the data field or the character that caused the error. If the first character following RESET is not accepted, a miscompare error again occurs and the cycle is repeated.

A verified-field-correction sequence can be used to correct an entire field with a minimum number of keystrokes. The sequence is initiated by pressing Field Backspace (  ). The cursor moves to the first position of the field and the correct data is keyed. Data field content is displayed only to the left of the cursor. When the corrected data has been keyed, the field is blanked on the display and the cursor is moved back to the first position from which point key verification takes place. Similarly, a verified-character-correction sequence is initiated by pressing Character Backspace (  ) invoking an analogous correction sequence (i.e., for a field of one character).

## RESUME

Purpose: To return control to key entry or key verify.

Format:

1. RESUME:
2. RESUME: (KET-field-name)

Description: To allow a field pre-processing or post-processing routine to return control to an interrupted key entry or key verification process.

When a branch to the statement labeled sn-pre or sn-post occurs, key entry or key verification processing is suspended, pending further direction from a field-processing routine that has the ability to specify the next field to be processed. Keystrokes are buffered during the suspension. When the field-processing routine specifies the next field to be processed, the buffered keystrokes are retrieved and processed in accordance with the specification of the selected field.

The operation is as follows:

1. If the FLDNUM keyword is not changed by the field processing routine, then the next cursor position is determined by: the MODE clause of the KET; the field-processing-routine type; and, the condition which caused the routine to be executed. Details concerning cursor positioning are tabled later in this section.

If the FLDNUM keyword is changed, the cursor moves to the first position of the corresponding field, regardless of MODE, field-processing-routine type and/or cause of execution. This is called a directed RESUME.

2. The cursor is moved to the first position of the designated field, regardless of MODE, field-processing-routine type and/or cause of execution. This is also a directed RESUME.

The RESUME statement may be used to return control to key entry or key verification from a control-key-processing routine (sn-1), an error routine (sn-2) and/or the next in-line statement to the KENTER or KVERIFY.

A processing routine may reference the contents of the current field using an alternate reference notation:

IOD-name.CURFLD

When this form of reference is coded, the processing routine may be used for more than one field within the application. The field which is reference through the CURFLD keyword corresponds to the KET field which contains the cursor and, therefore, repeatedly changes.

A processing routine may change the field to which CURFLD corresponds by changing the value of FLDNUM. For example, the following sequence would select the first field of the active KET, asterisk-fill the field and finally continue processing with that field:

```
IOD-name.FLDNUM = :00:
```

```
IOD-name.CURFLD = :*:
```

```
RESUME
```

If a processing routine assigns a value to FLDNUM and the value is less than FLDCOUNT, the designated field is activated and processing continues in sequence. If the value is equal to FLDCOUNT, the entry or verify process is continued at KET end; that is:

- the next processing statement within the exit routine is not executed; and,
- the statement following the KENTER/KVERIFY is executed next if RELEASE=AUTOMATIC is coded; or,
- the ENTER key is solicited if RELEASE=MANUAL is coded.

If the value is greater than FLDCOUNT, the next processing statement within the exit routine is not executed and an error condition results.

Table 8-3 indicates the cursor position following execution of the RESUME statement (as described for Format A.)

**Table 8-3: Cursor Positions Following Execution of Non-Directed RESUME**

<b>MODE</b>	<b>PROCESSING ROUTINE TYPE</b>	<b>CAUSE OF EXECUTION</b>	<b>NEXT CURSOR POSITION</b>
FIELD	SN-POST SN-PRE SN-1 SN-2 IN-LINE	Field Release key Field Backspace key User-defined key Error Resume after KENTER or KVERIFY statement	First position, next field First position, new field Current position, current field Program terminated First position, first field of last active KET
SCREEN	SN-POST SN-POST SN-POST SN-POST SN-PRE  SN-PRE  SN-PRE SN-1  SN-1  SN-2 IN-LINE	Field Release key Field Forward key Field Backspace key Cursor keys Data key (to initiate field re-selection) Field Release key (to initiate field re-selection) User-defined key User-defined key (when cursor is within data field) User-defined key (when cursor is outside of a data field) Error Resume after KENTER or KVERIFY statement	First position, next field First position, next field First position, new field New cursor position Current position, current field  Current position, current field  Transfer control to SN-1 Current position, current field  Current position, key ignored  Program terminated First position, first field of last active KET

## RESUMERR

Purpose: To return to key-entry or key-verify.

Format:

1. RESUMERR
2. RESUMERR (B)
3. RESUMERR (B, KET-field-name)
4. RESUMERR (,KET-field-name)

Description: Identical to RESUME except that an error is also posted.

The operation is as follows:

1. If the FLDNUM keyword is not changed by the exit routine, then an error-sequence is initiated at the field which contains the cursor. The next cursor position is determined by: the MODE clause of the KET; the field-processing-routine type; and, the condition which caused the routine to be executed. Details concerning cursor positioning are tabled later in this section.

If the FLDNUM keyword is changed by the field-processing-routine, the error sequence is initiated in the first position of the field indicated by FLDNUM. This is called a directed RESUMERR.

2. Same as described for Format 1, except that field B is used as an error message on the second line of the display during the error sequence.
3. Same as described for Format 2, except that the cursor position relates to the explicitly designated field. This is also a directed RESUMERR.
4. Same as described for Format 1, except that the cursor position  
RESUMERR.

The error sequence arising from the use of RESUMERR is as described for internally-detected keying errors, except that the message text is derived from field B.

Table 8-4 indicates the cursor position following execution of the RESUMERR statement (as described for Format 1).

**Table 8-4: Cursor Positions Following Execution Of Non-Directed RESUMERR**

<b>MODE</b>	<b>PROCESSING ROUTINE TYPE</b>	<b>CAUSE OF EXECUTION</b>	<b>NEXT CURSOR POSITION</b>
FIELD	SN-POST SN-PRE SN-1 SN-2 IN-LINE	Field Release key Field Backspace key User-Defined key Error Resume after KENTER or KVERIFY statement	First position, current field First position, current field Current position, current field Program terminated First position, first field of last active KET
SCREEN	SN-POST  SN-POST SN-POST SN-PRE  SN-PRE  SN-PRE SN-1  SN-1  SN-2 IN-LINE	Field Release key SN-POST Field Backspace key Cursor key Data key (to initiate Field re-selection) Field release key (to initiate field re-selection) User-defined key User-defined key (when cursor is within a data field) User-defined key (when cursor is outside of a data field) Error Resume after KENTER or KVERIFY statement	First position, current field First position, current field First position, current field First position, current field Current position, key ignored  Current position, key ignored  Current position, key ignored Current position, current field  Current position, key ignored  Program terminated First position, first field of last active KET



## **ERROR**

**Purpose:** To display an exception message for operator acknowledgement.

**Format:** ERROR (B)

**Description:** The operation proceeds in several steps as detailed:

1. The Error Line (line of the CRT is saved).
2. The field B data is displayed on line 2, columns 2-39, with a HIGH-BLINK attribute in column 1.
3. Column 40 or columns 40-80, depending upon the current CRTSIZE, are set to display blanks and to preserve the appearance of line 3.
4. The Error Tone is sounded once.
5. The Error Flasher is initiated.
6. All pre-keyed data and control keys are discarded.
7. Execution is suspended until RESET is depressed. (All other keys are discarded.)
8. After RESET, the Error Flasher is extinguished.
9. The Error Line is restored to its previous condition.
10. The next in-line statement is executed.

For example, the statement:

ERROR ('FILE NOT FOUND')

would produce the Error Line shown:

(line 2) = ⊕ FILE NOT FOUND  
          ↑                     └───┬───┘  
                                  FIELD  
                                  B  
                                  HIGHBLINK ATTRIBUTE

- NOTES:
1. If the length of field B is less than 38, trailing-space characters are used to fill the Error Line.
  2. If the length of field B is greater than 38, only the first 38 positions are transcribed to the Error Line.

## NOTIFY

Purpose:

To display an exception message for operator acknowledgement and to obtain a one-character response.

Format:

NOTIFY (B,C)

Description:

The operation proceeds in several steps as detailed:

- |            |  |
|------------|--|
| 1. thru 8. | As per ERROR   |
| 8a.        | Execution is suspended until the next key is depressed.  |
| 8b.        | The key is interpreted according to a preferred shift of LETTER and the resulting value is placed in the leftmost position of field C. |
| 9-10.      | As per ERROR.  |

NOTES: See NOTES for the ERROR statement.

## READSCREEN

Purpose: To transfer a CRT display line to a buffer.

Format: READSCREEN (IOD-name, B, ERR:sn-1, EOF:sn-2)

Description: The display line designated by the LINE keyword parameter is translated to EBCDIC and is moved to buffer.

If the current display uses the 480 character format, LINE can contain a value in the range of 1 to 12; if the 1920 format is used, the value of LINE can fall within the range of 1 to 24.

LINE is default-initialized to one (X'01') and is automatically incremented by 1 after the READSCREEN statement is executed. If the resulting value of LINE is greater than the number of lines of the display (i.e., either 13 or 125 for 480 or 1920 format), LINE is set to X'01' and control is transferred to the statement sn-2. If no EOF exit is coded, the end-of-file condition is regarded as an error condition.

If an error condition results (invalid value of LINE), the ERR exit is taken. If no error exit is coded, execution of the program is cancelled.

Display attribute bytes and other non-displayable graphics are translated to the EBCDIC space character.

The READSCREEN statement can be used in conjunction with the PRINT statement to produce a hardcopy representation of the display. If it is desired, the display image can be edited by the application prior to being printed. READSCREEN also provides read access to the prompting messages.

## READKEY

Purpose:

To allow an operator to interrupt a program during a non-interaction phase of execution. This command will cause the system to determine whether a key has been pressed to interrupt execution and, if so, to return the key value.

Format:

READKEY (IOD-name, B, ERR:sn)

Description:

If no data or control keys are available at the time of the request, execution continues with the statement labeled sn. If a data or control key is available, the key value is returned in the leftmost position of field B.

The preferred shift used for key interpretation is in accordance with IOD-name.SHIFT as established prior to the READKEY request.

NOTES: 1. If the application requires key input before processing can continue, the following format should be used:

READKEY (IOD-name, B)

In this instance, the absence of an available key causes execution of the application to be suspended until a key is struck.

2. READKEY allows the application to sense operator input without suspending mainline execution when no data is available.

Example:

\*

\* SAMPLE PROGRAM PORTION FOR READKEY FUNCTION

\*

```
IOD: CRT=STATION;          OPERATOR STATION DEFINITION
RCD: WORKAREA              WORK RECORD DEFINITION
    KEYSAVE(1)             SAVE AREA FOR READKEY
    CODERESET(2)=X'8A0A';   CODE RESET KEYBOARD VALUES
    KET: SCREEN            CRT DISPLAY DEFINITION
    .
    .
    .
    START ;                BEGIN EXECUTION SECTION
    .
    .
10, READKEY(CRT,KEYSAVE,ERR:20)  READ KEYBOARD
    IF(CODERESET,CONTAINS,KEYSAVE) STOP TEST FOR CODE RESET PRESENT
20, READ FROM DEVICE1
    .
    .
    .
    PROCESS INFORMATION
    .
    .
    .
    WRITE TO DEVICE2
GO:10                       REPEAT PROCESSING LOOP UNTIL
END                          OPERATOR INTERVENTION
```

## APPENDIX A: DIRECTORY OF FIGURES AND TABLES

<u>FIGURES</u>	<u>PAGE</u>
1-1 : Compilation Process	1-1
1-2 : MOBOLIST Sample Listing	1-2
2-1 : Series 21 CRT Screen Layout Form	2-1
2-2 : MOBOL Coding Form	2-4
8-1 : Identification of Operator Station Control Keys	8-6

<u>TABLES</u>	<u>PAGE</u>
5-1 : Establishing Field Specifications Where Field Name Has No Size Specifications	5-12
5-2 : Establishing Field Specifications Where Field Name Has A Size Specification	5-13
5-3 : Interpretation of Significance Parameters	5-25
6-1 : Interpretation of Literal Values of Institution Type	6-5
6-2 : Interpretation of Indexed Data References	6-8
6-3 : Interpretation of Non-Indexed Data References	6-8
6-4 : Compression Process	6-47
7-1 : I/O Execution Statements By Device	7-2
7-2 : Keywords By Device	7-8
7-3 : Search Operation Executed By OPEN For DISK/DISKETTE	7-10
7-4 : Keywords Used For I/O Operations	7-21
8-1 : Labeled Control Keys	8-12
8-2 : Values Returned In CONTROL, NUMPAD, NOLABEL And XCONTROL	8-15
8-3 : Cursor Positions Following Execution of Non-Directed RESUME	8-19
8-4 : Cursor Positions Following Execution of Non-Directed RESUMERR	8-21



## APPENDIX B RESERVED WORDS IN MOBOL

The following words are reserved for system use and must NOT be defined as programmer-assigned data names:

ADD	ERROR	OPEN	SEARCH
AND	ERR	PERFORM	SETEOD
BINARY	EQU	PICTURE	SETFORMS
BACKSPACE	EXIT	PRINT	SETTIME
BACKSPACEA	FREESPACE	PRINTA	SKIPFILE
CASE	FREESACEA	READ	SKIPFILEA
CHECKEOD	GETTIME	READA	SKIPRECORD
CHECKFORMS	GO	READKEY	SKIPRECORDA
CK10	HEX	READLOCK	SENDEOF
CK11	IF	READLOCKA	SENDEOFA
CKTB1	IFNOT	READNEXT	START
CKTB2	INSERT	READNEXTA	STOP
CLOSE	IOD	RELEASE	STRING
COMPRESS	JUSTIFY	READSCREEN	SUB
CURRENCY	KENTER	RESUME	TERMINATE
DECIMAL	KET	RESUMERR	TERMINATEA
DECOMPRESS	KEYIN	REWIND	TITLE
DELETE	KVERIFY	REWINDA	TRANSFORM
DISPLAY	MPY	REWINDLOCK	TRANSLATE
DIV	MARK	REWINDLOCKA	UNHEX
DIVR	MARKA	RCD	WAIT
EJECT	NOTIFY	RESUME	WRITE
END	NUMBER	RESUMERR	WRITEA
ENTRY	OR	SAME	XOR





## APPENDIX C: MOBOLIST

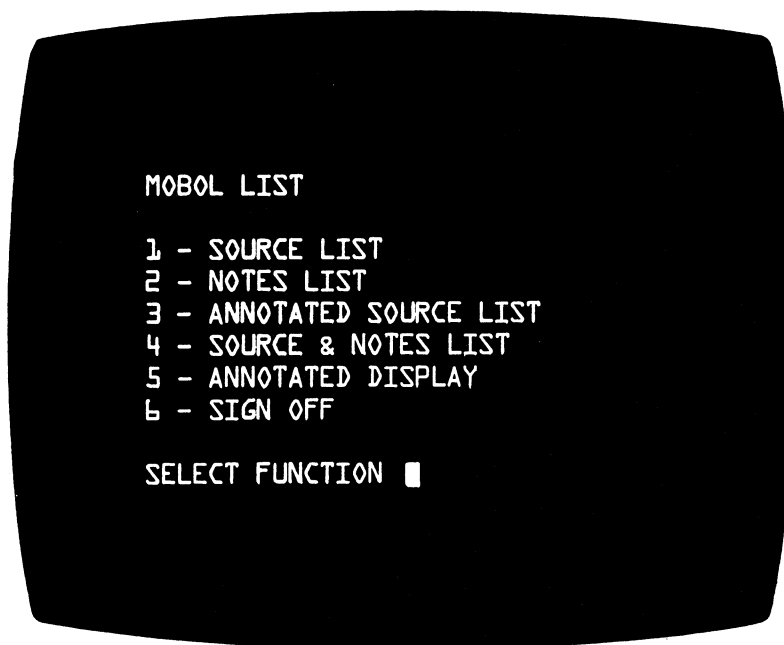
MOBOLIST is a utility program which can be used to generate a source listing of a MOBOL program and/or the associated NOTES file. The NOTES file contains any syntax errors detected by the MOBOL compiler during program compilation.

This utility program is accessed by inserting a library diskette containing MOBOLIST into a diskette drive of the Series 21 System.

A diskette containing MOBOL source code, the OBJECT FILE and the NOTES FILE should be inserted into a system diskette drive.

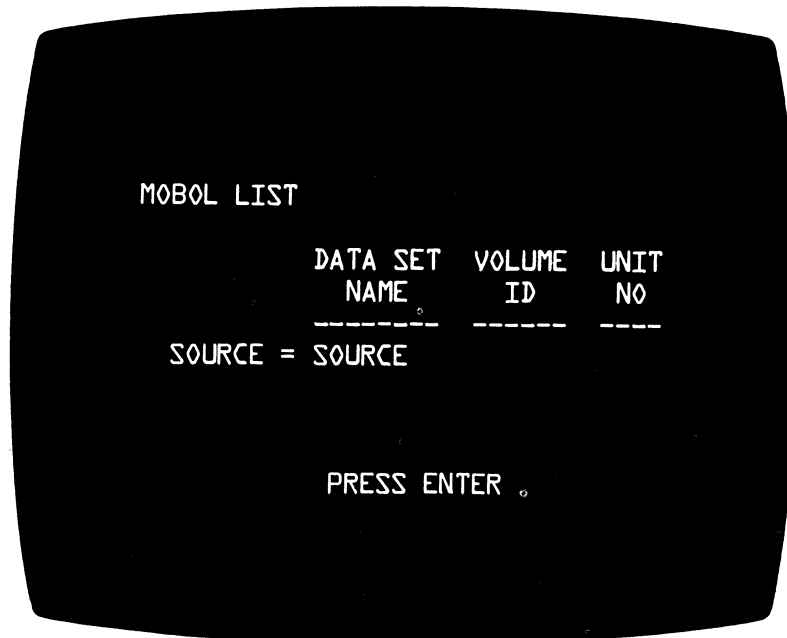
When the program selection display appears on the screen, select MOBOLIST, then press the ENTER key on the console keyboard.

The message LOADING MOBOLIST appears briefly, followed by the MOBOLIST function selection display.



Select the desired function:

1. If a listing of only the source code is desired, select 1. The following display will appear:



- a. Key-in the dataset name containing the source program, the Volume ID of the diskette and the unit number of the diskette drive in which the diskette is inserted. Press ENTER.
- b. If only a portion of a source program is to be listed, press the PROG ADV key. At the top of the screen;

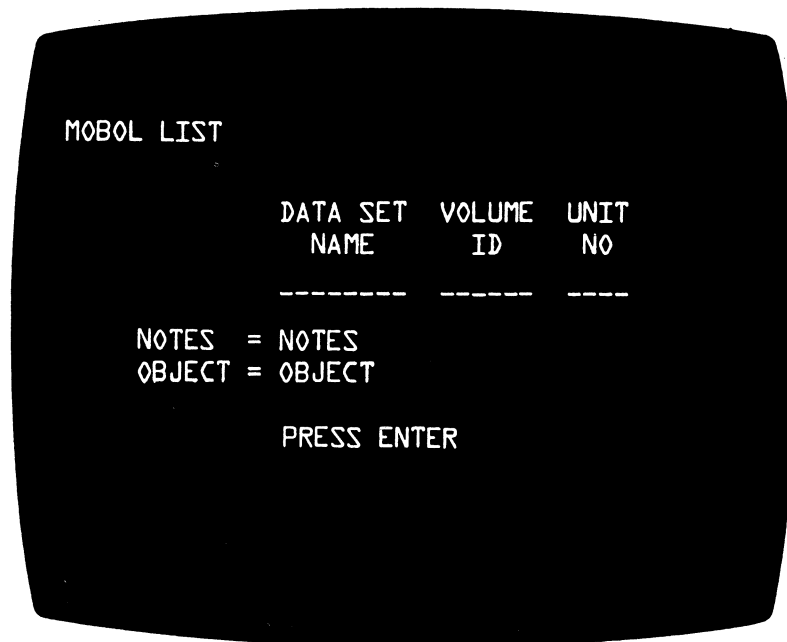
START =

LAST =

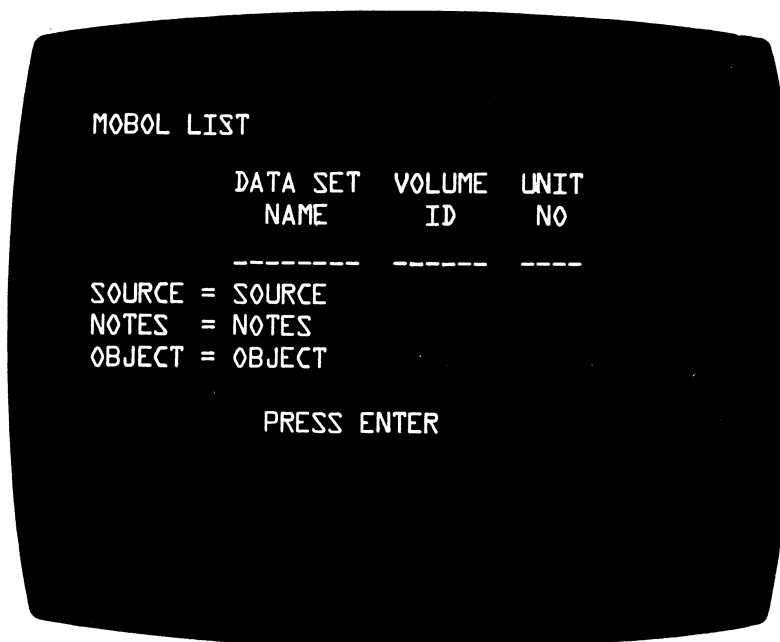
will appear. Enter the line numbers of the source program at which printing should begin and end.

- c. The printer will print the source listing. When the function is complete, the MOBOLIST Function Selection Display returns. Select 6 if no further listings are required.

2. If a listing of only the NOTES file is desired, select 2. The following display will appear:



- a. Key-in the dataset-name containing the OBJECT program, the Volume ID of the diskette and the unit number of the diskette drive into which the diskette is inserted. Press ENTER.
- b. When the function is complete, the MOBOLIST Function Selection Display returns. Select 6 if no further listings are required.
3. For an annotated source list, select 3.  
For a SOURCE and NOTES list, select 4.  
For an annotated display, select 5.
- a. After any of the above selections are made, the following display will appear on the screen:



- b. Key-in the dataset-names, Volume ID's and the unit numbers for SOURCE, OBJECT and NOTES files. Press ENTER.
  - c. When the function is complete, the MOBOLIST Function Display returns. Select 6 if no further listings are required.
4. A source program dataset contained on more than one diskette may be printed. MOBOLIST will print the contents of a diskette, then wait for the diskette containing the next subsequent portion of the source program to be inserted into the diskette drive. This cycle continues until the entire source program is printed. For all function selections but the annotated display, as many diskettes as required may be used. For the annotated display, the source program is limited to one primary and one extended dataset.
  5. Press the SEL MODE key, at any time during the MOBOLIST operation, to return to the Function Selection Display.

Following are examples of the MOBOL source listing, the NOTES file, the annotated source listing and the annotated display.

0013.	IOD:	KSTATION = STATION;	KEY STATION
0014.	IOD:	INPUT = DISKETTE	DISKETTE FILE
0015.		DATASET = 'MASTER01'	DATASET NAME
0016.		UNIT = 2;	DRIVE 2
0017.			
0018.	IOD:	OUTPUT = PRINTER	PRINTER OUTPUT
0019.		UNIT = 1;	DEFAULT PRINTER UNIT
0020.			
0021.	RCD:	WORKAREA	BUFFER
0022.		TYPE (1)	FIRST CHARACTER
0023.		DATA (127);	ALL THE REST
0024.			
0025.	KET :	DISPLAY	SCREEN DISPLAY
0026.		CRTSIZE = 400	BIG SCREEN
0027.		BLANK = CRT SIZE	BLANKS ENTIRE SCREEN
0028.		RELEASE = AUTOMATIC	'ENTER' NOT REQUIRED
0029.		(3,2) 'PRINT/SELECT UPDATE RECORDS'	PROMPT MESSAGE
0030.		(4,2) '1 - LIST ALL'	PROMPT MESSAGE
0031.		(5,2) '2- LIST UPDATES'	PROMPT MESSAGE
0032.		(6,2) '3 - SIGN OFF'	PROMPT MESSAGE
0033.		(7,2) 'SELECTION:'	PROMPT MESSAGE
0034.		(7,14) SELECTION (1,N);	FIELD ENTRY
0035.			
0036.	START		BEGIN CODE SECTION
0037.	10,	KENTER (KSTATION, DISPLAY)	DISPLAY
0038.		IF (SELECTION, CONTAINS, :3:) STOP	END PROGRAM
0039.		IFNOT ('12', CONTAINS, SELECTION) RESUMERR	DISALLOW BAD CHARS
0040.			
0041.		OPEN (INPUT, WORKAREA, ERR:200)	OPEN BUFFER FILE
0042.		OPEN (OUTPUT, WORKAREA, ERR:200)	OPEN PRINT FILE
0043.			
0044.	110,	READ (INPUT, WORKAREA, EOF:200, ERR:200)	READ INPUT
0045.		GO (SELECTION) 130, 130, 120	BRANCH ON SELECTION
0046.			
0047.	120,	IFNOT (TYPE, CONTAINS, :U:) GO:110	NOT UPDATE: RECYCLE
0048.			
0049.	130,	PRINT (OUTPUT, WORKAREA, EOM:110, ERR:200)	PRINT, IGNORE EOM
0050.		GO:110	CYCLE
0051.			
0052.	200,	CLOSE (INPUT, WORKAREA, ERR:201)	CLOSE, IGNORE ERRS
0053.	201,	CLOSE (OUTPUT, WORKAREA, ERR:10)	CLOSE PRINTER
0054.		GO:10	CYCLE
0055.			
0056.	END		END OF SOURCE FILE
<END>			

Figure C-1: MOBOL Source Listing

```

NO ERRORS DETECTED

OBJECT DATA SET = OBJECT

CODE AREA SIZE = 512 BYTES
DATA AREA SIZE = 512 BYTES

1 STATION SIZE = 1024 BYTES
2 STATION SIZE = 1536 BYTES
3 STATION SIZE = 2048 BYTES
4 STATION SIZE = 2560 BYTES

***** END NOTES LIST *****

```

Figure C-2: NOTES File

```

0017.
0018. IOD: OUTPUT = PRINTER PRINTER OUTPUT
*0018 00 MISPELLED DEVICE NAME

0036. START BEGIN CODE SECTION
0037. 10 KENTER (KSTATION, DISPLAY) DISPLAY
*0037 16 SYNTAX ERROR

0040.
0041. OPEN (INPET, WORKAREA, ERR:200) OPEN BUFFER FILE
*0041 00 REQUIRED IOD REFERENCE

0046.
0047. 120, IFNOT (TYPE, CONTAINS, :U ) GO:110 NOT UPDATE: RECYCLE
*0047 33 SYNTAX ERROR

0049. 130, PRINT (OUTPUT, WORKAREA, EOM:110, ERR:200) PRINT, IGNORE EOM
0050. GO:110 CYCLE
*0050 00 INAPPROPRIATE DEVICE OR OPERATION

0053. 201, CLOSE (OUTPUT, WORKAREA, ERR:10) CLOSE PRINTER
0054. GU:10 CYCLE
*0054 10 MISPELLED KEYWORD
*0054 00 UNDEFINED LABEL OR INCORRECT REFERENCE

THE COMPILER DETECTED 7 ERROR(S)

OBJECT DATA SET = OBJECT

CODE AREA SIZE = 512 BYTES
DATA AREA SIZE = 512 BYTES

1 STATION SIZE = 1024 BYTES
2 STATION SIZE = 1536 BYTES
3 STATION SIZE = 2048 BYTES
4 STATION SIZE = 2560 BYTES

*** END ANNOTATED SOURCE LIST ***

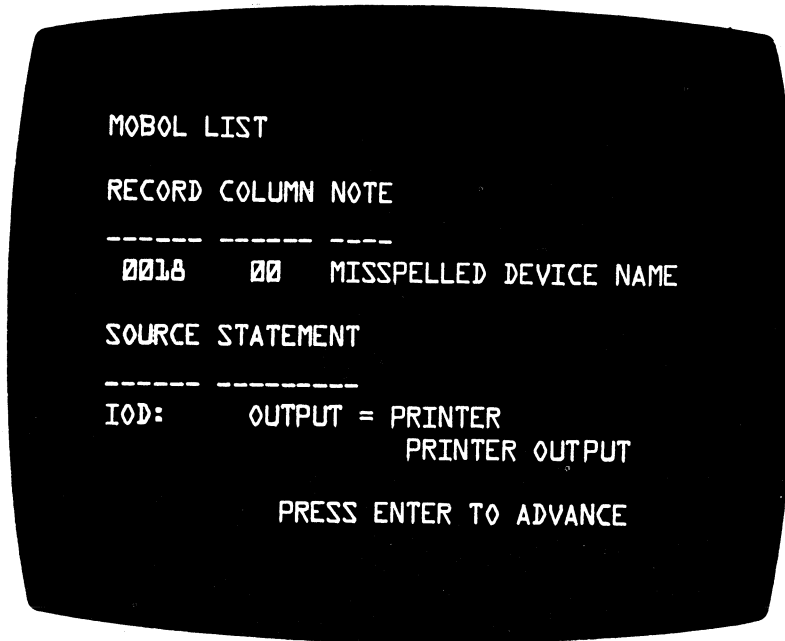
```

Figure C-3: Annotated Source Listing

**NOTE:**

In the annotated display, all statements are displayed individually (i.e., one at a time). The SCAN key is used to advance display of statements. The CODE and SCAN keys are used to reverse order of statement display.

Also, error messages (contained in the NOTES file) are individually displayed. Press ENTER to advance display of error messages.



**Figure C-4: Annotated Display**

## APPENDIX D: MOBOL CROSS REFERENCE PROGRAM

The MOBOL Cross Reference Program (MOBOLXRF) is a Series 21 application program, written in MOBOL, designed to generate cross reference listings for MOBOL programs residing on diskette.

MOBOLXRF should only be executed for those programs whose SOURCE contains no syntax errors.

### PROGRAM LOADING AND INITIATION

MOBOLXRF is loaded by selection from the VTOC menu display using procedures required to load any MOBOL program. The EXEC (Control Program) must be configured for at least a printer, 1 diskette, and an operation station. After the program is loaded, the 'SELECT OPTION' display is presented, whereupon the user enters the dataset-name of the MOBOL program to be used. The unit number and volume name are not requested and all diskettes are searched.

### DESCRIPTION OF THE 'SELECT OPTION' DISPLAY

The 'SELECT OPTION' display is represented in the following:



```
LINE
 1 MOBOLXRF
 2
 3
 4 OPTIONS:          MODES:
 5 1-XREF & SOURCE LIST B=BOTH
 6 2-XREF LIST         N=NAMES
 7 3-SIGN OFF          L=LABELS
 8
 9 SELECT OPTION : k
10 DATA SET NAME : dddddddddd
11 MODE : m
12
```

#### OPERATOR ENTERED FIELDS:

dddddddd DATA SET NAME (SOURCE is default)  
k OPTION, 1 through 3  
m MODE KEY (B is a default value)

The operator selects one of the above options according to the type of output desired.

Large programs may require two steps to complete the cross reference process. First, mode "N" produces the complete data name references followed by mode "L" to produce the label references.



## XREF & SOURCE LISTING

In this mode ('1' entered at k field, in line 9) all source lines are listed starting with record zero and terminating when the END statement is encountered. The MOBOLXRF program is designed to automatically add a binary 1 to the right-hand position of the dataset-name currently processed when EOD is encountered and the END statement has not been encountered. This automatic dataset-name increment is compatible with the manner in which the MOBOL compiler continues from one dataset to the next if an END statement has not been encountered. The user specifies the starting dataset-name on line 10 in the 'ddddddd' field.

The current relative record position of the record read is displayed on line 12.

The MOBOLXRF program reads the DATA section of a MOBOL program from record number zero to the occurrence of the START statement, and builds DATA NAMES and LABELS (if B is selected) in memory in array 'NAMES'.

Certain types of cross references are also noted such as 'REMAPS' AAA, where AAA is cross referenced as indicated by the statement number in which the REMAPS occurred. Also, pre-processing and post-processing labels are cross referenced if they should occur in KET field processing statements.

After the START statement is encountered, the program switches to the 'BUILDING CROSS REFERENCES' mode in line 3.

While printing the source statements, TITLE statements are placed in the MOBOLXRF page header and a top-of-form is executed and the header line printed. Similarly, 'EJECT' statements result in top-of-form and the header contents is printed. XREF listing will use first TITLE statement as a page header.

After source listing is finished, the total number of compilable statements are printed.

The user specifies the type of cross reference desired by entering an 'L', an 'N', or a 'B' in the 'm' field of line 11. Entering 'L' causes the program to search just for statement labels and cross references to the labels. Entering a 'N' causes the program to search just for data-names and references. Entering a 'B' causes the program to search for both data names and labels. The cross references for data-names are printed first followed by labels, providing either or both are specified in 'm' on line 11.

The cross reference listing shows the record number of the occurrence of a data-name or label in the far left margin. The next item is the actual data-name or label.

## XREF LISTING (ONLY)

This mode is identical to that described above except that the source lines are not printed at the time the MOBOLXRF program is extracting data-names, labels and cross references from the source. After completing the reading of all records in the starting dataset, and continuation datasets, the program prints the cross reference data requested under MODE, field 'm', line 11.

## ERROR MESSAGE SUMMARY

Errors which may occur in the operation of the MOBOLXRF program are tabulated below and may appear on line 2.

### Error Display Format:

```
FU NNNNNNNN  XX-Message text
P              XX-Message text
                (XX : Hex Error Code
                  U  : Unit number
NNNNNNNN     : File name      )
```

### Error Messages:

I/O ERROR (RSAE)  
DATA ERROR (AE)  
UNEXPECTED CODE  
NOT FOUND/AVAIL (RE)  
MEDIA CHANGED (E)  
FAULT

I/O ERROR (RSAC)  
DATA ERROR (SA)  
UNEXPECTED CODE  
NOT FOUND/AVAIL (RSC)  
FAULT

### Response Meaning:

R : Retry  
S : Retry previous record (Printer)  
Operator should set forms to the top-of-form and then respond 'S'  
Skip to next record and Retry (Diskette)  
A : Accept  
E : Treat as EOF/EOM condition  
C : Continue printing previous line

## OPERATING MESSAGE SUMMARY

Operating messages which may occur in the operation of the MOBOLXRF are tabulated below and may appear on line 3.

SCANNING FOR DATA NAMES ('B' & 'N' Option)  
SCANNING KET'S FOR LABELS ('L' Option)  
BUILDING CROSS REFERENCES  
PRINTING CROSS REFERENCES  
DONE  
ENTER NEXT DATA SET OR SEL MODE (in case END statement has not  
occurred, and the next dataset-name can't be found.)  
CONTINUING WITH NEXT DATA SET  
WILL CONTINUE — PLEASE WAIT  
LOADING BINARY SEARCH ARRAYS

## DESCRIPTION OF THE LISTING

In the name listing, character '-' is marked at IOD.KEYWORDS (Case A:1,3) and character '\*' is marked at Names whose cross references are not referenced. (Case A:2)

In the Label listing, character '\*' is marked at Labels which are declared but not referenced (Case B:4) and Labels are sorted like Case B showing.

When overflow occurs, MOBOLXRF prints the cross reference listing to the point of overflow. A '-' character is printed on Labels that are referenced but not yet declared; a statement number is printed where the Label is cross referenced (Case C). MOBOLXRF then continues to process the remaining source.

A

CROSS REFERENCE LISTING OF : XXXXXXXX	TITLE : YYYYYY ----	PAGE ZZZZ
364. -DKS.POSITION	1452	_____ ①
132. DSNAME	877 428 299	_____ ②
137. XMSGCONSTANT		_____ ③
322. -PRT.POSITION	520 150	_____ ④
900. COME	999 277	

B

316. *13		_____ ④
263. 100	252	
424. 1014	429 417	
320. 2	131	
222. 25	523 521	
563. 2020	888 827 520	
388. 3	290	

C

1588. -10200	1588
1587. -10500	1587
1575. -30000	1575

## APPENDIX E: ERROR MESSAGES

### FOR DISK:

I/O ERROR	01	HARDWARE ERROR
DATA ERROR	02	SHORT XFER
DATA ERROR	12	EOF
NOT FOUND/AVAIL	04	VOLUME NOT FOUND
NOT FOUND/AVAIL	14	FILE NOT FOUND
NOT FOUND/AVAIL	34	NOT SYSGENED
NOT FOUND/AVAIL	44	DISK NOT READY
NOT FOUND/AVAIL	54	FILE IN USE
NOT FOUND/AVAIL	64	VOLUME IN USE
NOT FOUND/AVAIL	74	RECORD LOCKED
NOT FOUND/AVAIL	84	NO MORE FREESPACE IN VOLUME. THE LAST RECORD WRITTEN IS LOST
MEDIA CHANGED	06	DISK DOOR WAS OPENED
FAULT	07	FILE NOT OPEN
FAULT	17	FILE ALREADY OPEN
FAULT	27	INVALID POSITION POINTER
FAULT	37	C-LIST FAILURE
FAULT	67	INVALID COMMAND
FAULT	77	DEVICE NOT A DISK
FAULT	97	MULT LOCK REQUESTED
FAULT	A7	DSCB FAILURE

### FOR DISKETTE:

I/O ERROR	01	HARDWARE ERROR
DATA ERROR	02	SHORT XFER
DATA ERROR	12	EOF/EOM
DATA ERROR	22	ACCESSIBILITY ERROR
DATA ERROR	32	ERMAP NON-BLANK
WRITE PROTECTED	03	TRIED TO WRITE A READ-ONLY FILE
NOT FOUND/AVAIL	04	VOLUME NOT FOUND
NOT FOUND/AVAIL	14	FILE NOT FOUND
NOT FOUND/AVAIL	24	UNIT IN USE
NOT FOUND/AVAIL	34	UNIT # NOT SYSGENED
NOT FOUND/AVAIL	44	NO DISKETTE READY
NOT FOUND/AVAIL	54	FILE IN USE
NOT FOUND/AVAIL	64	VOLUME IN USE
NOT FOUND/AVAI:	74	RECORD LOCKED
MEDIA CHANGED	06	DKT DRIVE DOOR WAS OPENED

**Cont. DISKETTE :**

FAULT	07	FILE NOT OPEN
FAULT	17	FILE ALREADY OPEN
FAULT	27	INVALID POSITION POINTER
FAULT	37	C-LIST FAILURE
FAULT	67	INVALID COMMAND
FAULT	77	DEVICE NOT A DKT
FAULT	97	MULT LOCK REQUESTED
FAULT	A7	DSCB FAILURE

**FOR INDEXED ACCESS METHOD SEQUENTIAL:**

DATA ERROR	02	BUFFER SIZE NOT 128 ON OPEN
DATA ERROR	12	EOF ON READ
FAULT	67	INVALID COMMAND

**FOR INDEXED ACCESS METHOD RANDOM:**

DATA ERROR	02	BUFF SIZE NOT 128 ON OPEN
DATA ERROR	12	EOF ON INSERT
FAULT	67	INVALID COMMAND
(INFORMATION CODE)	18	NO RECORD ON RD OR RD NEXT (RECORD NOT FOUND)
(INFORMATION CODE)	28	DUPLICATE INSERT

**FOR TAPE:**

I/O ERROR	01	HARDWARE ERROR
I/O ERROR	11	PARITY ERROR
I/O ERROR	31	POS. ERROR DURING RECOVERY
I/O ERROR	41	OVERRUN
I/O ERROR	51	TIMEOUT
DATA ERROR	02	SHORT XFER
DATA ERROR	12	EOF/EOM
DATA ERROR	22	ILLEGAL 7-TRACK CHAR
WRITE PROTECTED	03	WRITE PROTECTED
NOT FOUND/AVAIL	34	NOT SYSGENED
NOT FOUND/AVAIL	44	NOT READY
NOT FOUND/AVAIL	54	TAPE IN USE
FAULT	07	FCB FAILURE
FAULT	17	FILE ALREADY OPEN
FAULT	37	C-LIST FAILURE
(INFORMATION CODE)	18	TAPE BACKSPACED TO LOAD POINT

**FOR PRINTER:**

I/O ERROR	01	HARDWARE ERROR
I/O ERROR	11	PARITY ERROR
DATA ERROR	12	BOTTOM OF FORMS HAS BEEN DETECTED
DATA ERROR	22	INVALID VFU CHAR
NOT FOUND/AVAIL	34	PRINTER NOT SYSGENED
NOT FOUND/AVAIL	44	PRINTER NOT READY
NOT FOUND/AVAIL	54	PRINTER IN USE
FAULT	07	FILE NOT OPEN
FAULT	17	FILE ALREADY OPEN
FAULT	37	C-LIST FAILURE



## APPENDIX F: GLOSSARY OF TERMS

Definitions of terms used in this manual are presented, in this section, to provide convenient reference and facilitate understanding of MOBOL. These terms are listed alphabetically.

**Abnormal Condition** — The occurrence of an Exception Condition or an Error Condition.

**Access Method** — A technique for moving data between a computer and its peripheral devices, e.g., serial access, random access.

**Address** — An identification for a location in which data is stored.

**Algorithm** — A procedure for performing a function described in terms of program operations.

**Application** — Programs written to produce specific reports and/or to update or create specific files.

**Attribute Byte** — The byte that controls the display characteristics of the field immediately following.

**Automatic Release** — Designates that the display data is to be released immediately for processing without intervention.

**Binary** — A numbering system based on the powers of two.

**Bit** — A binary digit.

**Buffer** — An area in storage which temporarily holds data that will subsequently be processed.

**Byte** — A sequence of adjacent binary digits operated upon as a unit.

**Cathode Ray Tube (CRT)** — An electronic device that can be used to display graphic images.

**Comment** — A statement used to include information that may be helpful in documenting a program for clarity.

**Comparator** — The comparator is an operator in a conditional phase (e.g.,  $A \leq B$ ;  $\leq$  is the comparator).

**Compile** — To transform a human readable program into a machine readable form.

**Compiler** — A program that compiles.

**Data Compression** — A technique that saves storage space by eliminating redundant data to shorten the length of records or blocks.

**Data Control Block (DCB)** — A control block used by access routines in storing and retrieving data.

**Data Element** — See **Field**

**Data Item** — See **Field**



**Data Management** — A general term that collectively describes those functions of the system that provide creation of and access to stored data, enforce data storage conventions and regulate the use of peripheral devices.

**Dataset** — A named area on a physical recording medium.

**Default** — The choice made by the system when no explicit choice is specified by the user.

**EOD** — End-of-Data.

**EOE** — End-of-Extent.

**EOM** — End-of-Medium.

**Error Condition** — The occurrence of a termination condition which prevents the detection of a Normal Condition or an Exception Condition.

**Exception Condition** — The occurrence of a secondary anticipated result of a function.

**Field** — The smallest unit of data that has meaning in describing information; the smallest unit of named data. Synonymous with Data Item and Data Element.

**File** — A group of related logical records as perceived by an application program; it may be in a different form from that in which it is stored on peripheral devices.

**Fill Character** — To insert the representation of a specific character in a storage medium, usually for the purpose of deleting unwanted data.

**Format** — The predefined arrangement for data.

**Hexadecimal** — A numbering system based on the powers of sixteen.

**Index** — A value indicating a specific record of several identically structured records.

**Index Dataset** — A control dataset on DISK or DISKETTE specifying the relative positions of records located within another (Target) dataset.

**Instruction** — A statement that specifies an operation and the values or locations of its operands.

**Iterate** — To repeat.

**Keystation** — A collection of hardware components accessible to the operator of a computer system which allows selection of, and interaction with, an application. The collection of hardware includes at least a keyboard and a screen.

**Keywords** — Predefined names recognized by the compiler to identify a particular access parameter for definition or reference.

**Labeled Volume** — A volume whose name is recorded on the medium in such a way as to be readable by the computer system.

**Line Number** — A number associated with a line in a printout or display.

**List** — An ordered collection of data items.

**Literal** — A symbol or a quantity in a source program that is itself data, rather than a reference to data.

**Loading** — The process of transferring an application from an external medium to a computer system's memory.

**Mask** — A pattern of characters that is used to control the retention or elimination of portions of another pattern of characters.

**Normal Condition** — The occurrence of the primary anticipated result of a function.

**Notes File** — A dataset produced by the compiler which contains summary information about the compilation and notes of detected error messages (if any).

**Object Code** — Output from a compiler or assembler which is itself executable machine code or is suitable for processing to produce executable machine code.

**Offset** — The number of measuring units from an arbitrary starting point in a record, area or control block, to some other point.

**Parameter** — A variable that is given a value for a specific purpose or process.

**Peripheral Device** — A part of the computer system external to the CPU and RAM, e.g., magnetic tape drive and diskette drive.

**Physical Record** — A collection of bits that are physically recorded on the storage medium and which are read or written by one access to the device.

**Protected Field** — A display field in which the operator can enter or modify data only under specific application conditions.

**Random Access** — To obtain data directly from a storage location based on a key or address. Also called Direct Access.

**Record** — A group of related fields of information treated as a unit by an application program.

**Serial** — Occurring successively in time, or by physical position.

**Sequential** — Occurring in a logical order not necessarily related to physical position.

**String** — A sequence of entities such as characters or physical elements.

**Subroutine** — A group of statements performing a specific function that may be performed as often as necessary.

**Target Dataset** — A dataset consisting specifically of application data referenced through one or more index datasets each of which defines an organization for the target dataset.

**Unit** — An addressable device attached to a computer system.

**Volume** — A physical instance of a medium (e.g., one diskette, one reel of tape).

**Volume ID** — The name recorded on a Labeled Volume.

**VTOC (Volume Table of Contents)** — The volume resident directory for the datasets recorded on the volume.

**Zero Fill** — To character fill with the representation of zero.

## APPENDIX G: CHECKDIGIT ALGORITHMS

A check digit, which is placed in the unit's position of a self-checking number, is developed by calculations made on the base number (the original number without the check digit). The calculated digit is then included with the base number to create the self-checking number. When a self-checking number is entered into the system, the calculations originally performed are repeated, and the generated digit is compared to the check digit in the number entered. If the numbers are not identical, a check digit error occurs and the error indicator is set.

The system calculates the check digit using either, the Modulus 10 or Modulus 11 Algorithm.

### MODULUS 10 ALGORITHM

The operation for Mod 10 is as follows:

1. Ignoring the check digit, multiply the base number's unit's position, and every alternate position moving right-to-left, by two. For example,

$$\begin{array}{rcccc}
 4 & 6 & 9 & 2 \\
 & \underline{\times 2} & & \underline{\times 2} \\
 & 12 & & 4
 \end{array}$$

2. Add the digits of these products to the digits of the base number which were *not* multiplied by two.

$$4 + 1 + 2 + 9 + 4 = 20$$

3. Divide this sum by 10:

$$\begin{array}{r}
 2 \\
 10 \overline{) 20} \\
 \underline{20} \\
 0 \text{ remainder}
 \end{array}$$

4. This remainder (0) is the offset.
5. Using the offset in the table below, a zero offset designates a check digit of zero (0).

MODULUS 10 CHECK DIGITS	
Offset	0 1 2 3 4 5 6 7 8 9
Check digit	0 9 8 7 6 5 4 3 2 1

## MODULUS 11 ALGORITHM

The operation for Mod 11 is as follows:

1. Ignoring the check digit, multiply the unit's position of the base number by two, the tens position by three, the hundreds position by four, and so on. Continue this procedure up to multiplication by seven (if the number contains that many digits), then begin multiplying by two again.

$$\begin{array}{cccc}
 4 & 6 & 9 & 2 \\
 \times 5 & \times 4 & \times 3 & \times 2 \\
 \hline
 20 & 24 & 27 & 4
 \end{array}$$

2. Add the products to each other.

$$20 + 24 + 27 + 4 = 75$$

3. Divide the sum by 11:

$$\begin{array}{r}
 6 \\
 11 \overline{) 75} \\
 \underline{66} \\
 9 \text{ remainder}
 \end{array}$$

4. The remainder nine (9) is the offset.
5. Using the offset in the table below, a nine offset designates a check digit of 2.

MODULUS 11 CHECK DIGITS	
Offset	0 1 2 3 4 5 6 7 8 9 10
Check digit	0 X 9 8 7 6 5 4 3 2 1

Note that a remainder (offset) of one (1) corresponds to an X which is stored as X'FF'. This is considered an illegal check digit.

For both algorithms, numerals 0 through 9 and all other EBCDIC characters with the values 0 through 9 in the low-order hex digit are totaled by the low-order hex digit value. All other characters are totaled as the value zero.

## APPENDIX H: INSTRUCTIONS FOR THE MOBOL COMPILER

The MOBOL Compiler is an MDS-supplied software package which converts user-written source programs into object programs. The Compiler is provided on diskette.

The MOBOL Compiler needs three datasets which can be allocated via the MDS DSKETTEU program. (Refer to the MDS Series 21 Release 7.0 Operator's Guide (Form No. M-2611) for a discussion of this program.)

They are:

- SOURCE — Input statements written by the user in MOBOL language.
- OBJECT — The executable MOBOL program, output by the Compiler.
- NOTES — Syntax errors in source statements, detected during compilation. Using an MDS-supplied utility called MOBOLIST, these error or warning messages can be merged with the original input statements to produce an annotated source listing on a printer or display screen.

The allocation requirements of SOURCE and NOTES are decided by the size and complexity of the source program, while the OBJECT requires a minimum of 300 records. The user is responsible for satisfying the minimum allocation requirement.

NOTE: The MOBOL Compiler is prohibited from more than one operator access at a time.

### COMPILER OPERATION

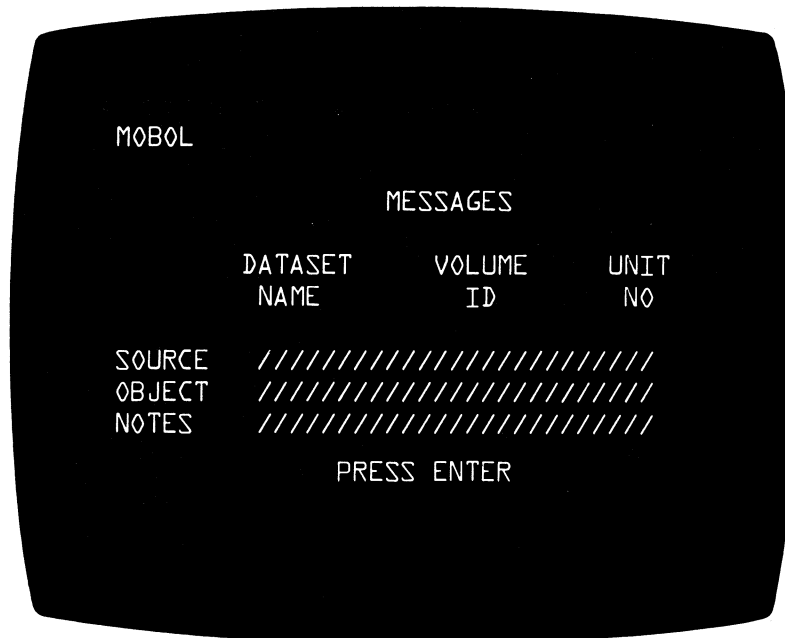
1. Controller — POWER On.  
Operator Station — Power ON.
2. Insert the Library Diskette containing the MOBOL Compiler.

NOTE: If Library Diskette has been previously loaded, select SIGN OFF from the Function Selection Display; otherwise, press RESET button on Controller Console to load Library Diskette.

- Wait for the Program Selection Display to appear on the display screen.



- Key the two-digit selection number for MOBOL Compiler; then press ENTER. "LOADING MOBOL" appears briefly, followed by the Data Set Display when loading is complete.



messages — presents exception conditions while searching datasets as explained in Step 7.

ddddddd — dataset-name.

vvvvv — Volume ID.

u — device unit number.

All these fields are first blanked by the Compiler.

5. The order of keying the fields is arranged as follows:

1. SOURCE dataset-name
2. OBJECT dataset-name
3. NOTES dataset-name
4. SOURCE Volume ID
5. SOURCE unit no.
6. OBJECT Volume ID
7. OBJECT unit no.
8. NOTES Volume ID
9. NOTES unit no.

But the operator can use the two field move keys (  $\rightarrow$  and  $\leftarrow$  ) to change the order of input, or use the HOME key to position the cursor to the first field (i.e., SOURCE dataset-name).

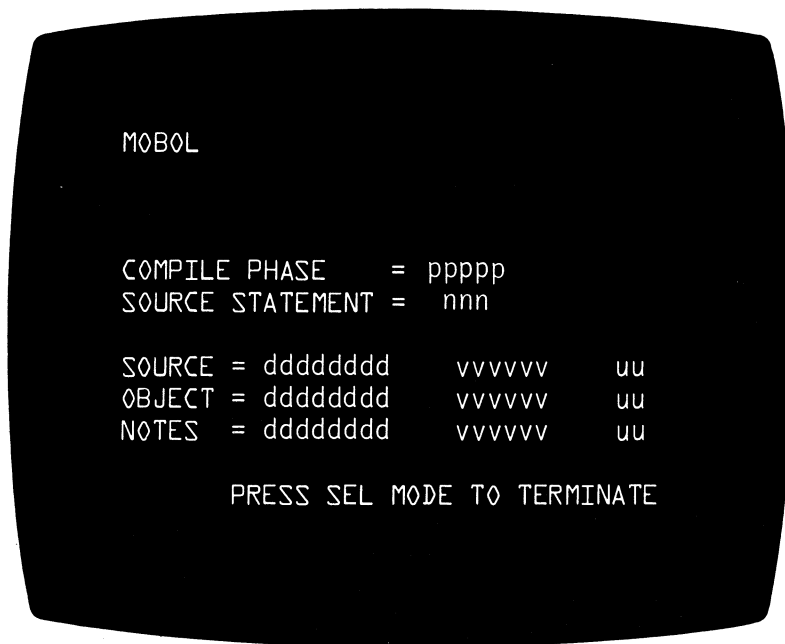
6. After the three dataset-names have been defined, the operator can press the ENTER key any time to signal the end of input. Then, the Compiler tries to locate the dataset-names on the volumes of the devices. If the unit number is left blank by the operator, the search starts with the lowest numbered unit and proceeds sequentially through all the units or until the dataset is found. Similarly, if the Volume ID is left blank by the operator, the Compiler looks for the first volume containing the requested dataset-name.
7. If a dataset can be found, its Volume ID and device unit number are temporarily fixed and are used to replace the blank volume or unit (if any) on the display; if not found, an error message is displayed in the message field. For instance, the message:

"SOURCE OBJECT — NOT FOUND"

is displayed if neither can be found. Only under the condition that the search for all the datasets succeeds, the Compiler proceeds to Step 8; otherwise Step 5 is re-entered; meanwhile, all the temporarily fixed datasets are unfixed and subject to changes. However, the operator can use some of them as inputs again simply by leaving them on display.

8. All three temporarily fixed datasets become permanently fixed (with the exception stated in Step 9) and the compilation takes place. The Compiler also presents the MOBOL Compilation Display, which remains active during the compilation process.





- pppp — the current compile phase is presented as one of the following:
  - DATA: processing the Data Definition Section
  - START: wrap-up of data definitions
  - CODE: processing the Execution Section
  - BUSY: compile wrap-up
  - CAT: searching for concatenated source file
- nnn — the number of the current source statement being compiled
- ddddddd — dataset - name
- vvvvvv — Volume ID
- uuu — device unit number

Pressing SEL MODE terminates the compilation prematurely, and returns the Program Selection Display. The contents of both OBJECT and NOTES from this termination is unpredictable.

9. If the Compiler reaches EOD (end-of-data) prior to detecting an END source statement, it assumes that there are multiple source datasets. The Compiler then searches for the next SOURCE dataset. The new dataset-name is generated by using the previous dataset-name with the EBCDIC value of its rightmost character incremented by 1. The Volume ID and unit number of the new dataset are considered as unrestricted in this case. If the new dataset is found, its name, volume, and unit number reflect on the display accordingly; then, the compilation continues with the new SOURCE dataset.

NOTE: The Compiler will wait for either the proper dataset to be mounted or the depression of SEL MODE to terminate.

Since the number of SOURCE datasets for one compilation is not limited, a SOURCE program can consist of a multi-volume dataset (with names incremented as in DATASETA, DATASETB, et.) on different device units.

10. The Compiler returns control to the system upon completion. The system displays the Program Selection Display for next operation. The pertaining syntax errors diagnosed by the Compiler can now be printed with or without the SOURCE statements via an MDS-supplied utility called MOBOLIST. (For discussion of MOBOLIST, see Appendix C.) If no error message is given, the OBJECT is ready for execution.

Error messages for the MOBOL Compiler (listed below) are displayed on line 2 of the display screen in the following format:

⊕##/DD

DD	DESCRIPTION
09 —	Terminal lockout
10 —	I/O error (source file)
11 —	I/O error (note file)
12 —	SYMBOL TABLE OVERFLOW
13 —	Label Reference overflow
14 —	I/O error (object file)
15 —	Symbol table space not allocated
16 —	Call your MDS service representative
17 —	I/O error (patch file)
18 —	Dictionary file overflow



## APPENDIX I: SYSTEM ERROR MESSAGES THAT OCCUR DURING EXECUTION OF USER-DEFINED APPLICATIONS

System error messages are displayed on line 2 of the display screen in the following format:

⊕ ###DD

(Column 1)

<u>DD</u>	<u>DESCRIPTION</u>
02	An index variable has been assigned an out of range value.
04	No error statement number was coded on an I/O operation and an error was encountered.
19	The processing routine assigned a value to FLDNUM that is greater than FLDCOUNT; no KENTER or KVERIFY is active or there is no ERR clause in that statement.
20	No error statement number was coded in a READSCREEN operation and the LINE parameter specified is beyond the maximum limit for the display screen.
21	During System Generation, a 480 character screen size was specified and a 1920 character screen size is required.
22	A directed RESUME field-name is not defined in the currently active KET and there is no ERR clause in the KENTER or KVERIFY statement.

If a system error message not listed above is displayed, contact your local MDS representative.



## INDEX

- ACCESS 7-3
- ACTUAL 7-4
- Alignment 5-8
  - without filling 5-10
- Alphanumeric strings 5-6
- AND
  - syntax 6-29
- BACKSPACE 7-23
  - syntax 6-87
- BINARY ARITHMETIC STATEMENTS 6-23 - 6-27
- BINARY
  - syntax 6-33
- BINARY ADD
  - syntax 6-24
- BINARY DIVIDE
  - syntax 6-27
- BINARY MULTIPLY
  - syntax 6-26
- BINARY SUBTRACT
  - syntax 6-25
- BLKLEN 7-4
- BOOLEAN STATEMENTS
  - syntax 6-28 - 6-31
- BOUND 8-2
- CASE
  - syntax 6-67
- CHANGE 8-4
- CHECK DIGIT
  - syntax 6-106
- CHECKEOD 7-13
  - syntax 6-79
- CHECKFORMS 7-25
  - syntax 6-92
- CLOSE 7-16
  - syntax 6-76
- COMPATIBLE CHANNEL I/O OPERATIONS 7-29
  - OPEN 7-29
  - READ 7-30
  - SENDEOF 7-31
  - WRITE 7-32
  - CLOSE 7-33
- COMP CHAN 7-29
- COMPRESS
  - syntax 6-45
- COMPUTED GO
  - syntax 6-66
- CONDITIONAL PHRASES
  - syntax 6-57
  - alphanumeric 6-58
  - binary 6-60
  - bit 6-61
  - decimal 6-62
  - sort 6-64
  - computed go 6-66
  - case 6-67
  - error test 6-68
- CONDITIONALS
  - syntax 6-57
- CONTROL 8-4
- CONTROL STATEMENTS 6-54 - 6-68
- CONTROL KEYS 8-11
- CROSS REFERENCE PROGRAM D-1
- CRTSIZE 5-19
- CURFLD 8-3
- CURRENCY 6-42
- Data Definition Statements
  - formats 2-2, 2-3
  - requirements 2-3, 5-1
- DATASET 7-6
- DECIMAL ARITHMETIC STATEMENTS 6-17 - 6-22

DECIMAL  
syntax 6-34  
ADD 6-19  
DIVIDE 6-22  
EQUATE 6-18  
MULTIPLY 6-21  
SUBTRACT 6-20

Decimal Numbers 5-6

DECOMPRESS  
syntax 6-49

DELETE 7-19  
syntax 6-80

DISK I/O OPERATIONS 7-8, 7-9  
OPEN 7-10  
READ 7-11  
READLOCK 7-13  
CHECKEOD 7-13  
WRITE 7-14  
RELEASE 7-15  
SETEOD 7-15  
FREESPACE 7-15  
CLOSE 7-16

DISKETTE I/O OPERATIONS 7-8, 7-9

DISPLAYS. *See Screen displays.*

EDITING STATEMENTS 6-32 - 6-53

EJECT 4-1

END 4-2

ENTRY  
syntax 6-71

EOD 7-4

EOE 7-4

EQU 5-28

ERRCODE 7-4

ERROR 8-22  
syntax 6-101

ERROR MESSAGES. (*See Series 21 Display Messages Manual M-3925.*)

ERROR TEST  
syntax 6-68

EXECUTION Statements 2-2, 2-4  
requirements 2-4, 6-1

EXIT Parameter 5-26  
Statement 6-73

Explicit Field Size 5-7

Fields  
non-sequential 5-11  
sequential 5-11  
without Initial-Values 5-9

Field Specifications 5-12, 5-13, 5-24

FIELD MODE 8-7

File 5-3

FILL  
syntax 6-14

Fill Characters  
alignment of 5-8  
non-graphic 5-14

FILTER 8-2

FLDCOUNT 8-3

FLDNUM 8-3

FORMS 7-4

FREESPACE 7-15  
syntax 6-81

GETTIME  
syntax 6-107

Guide messages/specifications 5-22

HEX  
syntax 6-52

Indexed Data 6-6

INPUT/OUTPUT OPERATIONS 7-1

INSERT  
syntax 6-82

IOD 5-2, 5-3

I/O Statements 7-1, 7-2  
syntax 6-74

I/O Execution Statements by Device (Table) 7-2  
I/O Statements in the Data Definition Section 7-1  
I/O Statements in the Execution Section 7-2  
JUSTIFY LEFT  
syntax 6-35  
JUSTIFY RIGHT  
syntax 6-36  
KENTER 8-5  
syntax 6-97  
KEYBOARD 8-2  
KEYSTROKE 8-4  
KEYVALUE 7-3  
KEYWORDS (*See also specific keyword name.*) 7-3  
Keywords By Device (Table) 7-8  
Keywords Used For I/O Operations (Table) 7-21  
Keywords Used In The Execution Section 8-3  
KET 5-18  
KETNUM 8-3  
KVERIFY 8-16  
syntax 6-98  
LINE 8-4  
Literal values,  
interpretation of 6-5  
MARK 7-5  
syntax 6-88  
MATCH 8-4  
MBLXREF D-1  
MCSEQ 8-2  
MISCOMP 8-2  
MOBOL CROSS REFERENCE PROGRAM D-1  
MOBOLIST C-1  
MOVE STATEMENTS 6-11  
MOVE LEFT AND FILL  
syntax 6-12  
MOVE LEFT, NO FILL  
syntax 6-15  
MOVE RIGHT AND FILL  
syntax 6-13  
MOVE RIGHT, NO FILL  
syntax 6-16  
NOLABEL 8-4  
Non-graphic  
fill characters 5-14  
hexadecimal strings 5-14  
Non-Indexed Data 6-8  
Non-sequential 5-11  
NOTE TYPE 8-3  
NOTIFY 8-23  
syntax 6-102  
NUMBER EDIT  
syntax 6-43  
NUMPAD 8-4  
Offset notation 5-9  
OPEN 7-10  
syntax 6-75  
OPERATION 8-2  
OR  
syntax 6-30  
OUTLEN 7-5  
Overall Record Areas 5-7  
PERFORM  
syntax 6-70  
PICTURE EDIT  
syntax 6-37  
POSITION 7-5  
PRINT 7-26  
syntax 6-93



PRINTER I/O OPERATIONS 7-25

OPEN 7-25  
CHECKFORMS 7-25  
SETFORMS 7-25  
PRINT 7-26  
WRITE 7-26  
CLOSE 7-26

RANDOM INDEX ACCESS METHOD 7-18

OPEN 7-18  
READ 7-18  
READNEXT 7-19  
READLOCK 7-19  
RELEASE 7-19  
DELETE 7-19  
WRITE 7-20  
INSERT 7-20  
CLOSE 7-20

RCD 5-2, 5-5

arrays 5-16  
statement formats 5-5 - 5-17

READ 7-11, 7-18

syntax 6-77

READKEY 8-25

syntax 6-104

READLOCK 7-13, 7-19

syntax 6-83

READNEXT 7-19

syntax 6-84

READSCREEN 8-24

syntax 6-103

RELEASE 7-15, 7-19

syntax 6-85

REMAPS 5-29

Remapping, Record 5-29

format 5-29  
extended records 5-31  
indexed records 5-32  
normal records 5-30  
types defined 5-29

Requirements 2-3, 2-4

Data Definition Section 2-3  
Execution Section 2-4

Reserved Words B-1

RESUME 8-17

syntax 6-99

RESUMERR 8-20

syntax 6-100

REWIND 7-23

syntax 6-89

REWINDLOCK 7-23

syntax 6-90

SAME

syntax 6-109

Screen displays 2-1

designing 2-1  
error messages 2-1  
guide messages 5-22  
sample layout form 2-1

SCREEN MODE 8-9

Semantics Conventions 6-5

SENDEOF 7-2

syntax 6-95

Sequential fields 5-11

SEQUENTIAL INDEX ACCESS METHOD 7-17

SETEOD 7-15

syntax 6-86

SETFORMS 7-25

syntax 6-94

SETTIME

syntax 6-108

SHIFT 8-4

defined 5-25

SIGNIF 8-2

SKIPFILE

syntax 6-91

SLEW 7-6

Source Program Structure 1-3

example of 1-2

SOURCE

defined 5-26

SPECIAL PURPOSE STATEMENTS 6-105

START 4-2

STATE 7-7

Statement Syntax and Descriptions 6-1 - 6-112

STATION I/O OPERATIONS 8-1

Data Definition Keywords 8-2

BOUND 8-2

FILTER 8-2

KEYBOARD 8-2

MCSEQ 8-2

MISCOMP 8-2

OPERATIONAL 8-2

SIGNIF 8-2

Execution Keywords

CHANGE 8-4

CONTROL 8-4

CURFLD 8-3

FLDCOUNT 8-3

FLDNUM 8-3

KETNUM 8-3

KEYSTROKE 8-4

MATCH 8-4

NOLABEL 8-4

NUMPAD 8-4

SHIFT 8-4

TYPE/NOTE 8-3

XCONTROL 8-4

STATION I/O STATEMENTS 6-96

STATUS 7-5

STOP

syntax 6-56

STRING

syntax 6-11

SUBROUTING STATEMENTS 6-69

Syntax and Descriptions, Statement 6-1 - 6-112

Syntax Conventions 6-3

TAPE I/O OPERATIONS 7-22

BACKSPACE 7-23

CLOSE 7-23

MARK 7-23

OPEN 7-22

READ 7-22

READLOCK 7-23

REWIND 7-23

REWINDLOCK 7-23

SKIPFILE 7-24

WRITE 7-22

TARGET 7-3

TITLE 4-1

TRANSLATE

syntax 6-44

TYPE/NOTE 8-3

UNCONDITIONAL GO

syntax 6-55

UNHEX

syntax 6-53

UNIT 7-7, 8-4

VERIFY

defined 5-27

VERTICAL FORMS UNIT

VFU 7-27

VOLUME 7-7

WRITE 7-14

syntax 6-78

XCONTROL 8-4

XFERLEN 7-5

XFERSTATUS 7-5

XOR

syntax 6-31



## READERS COMMENT FORM

**SERIES 21 MOBOL Reference Manual**  
**First Edition, Release 7.0**  
**Form No. M-2612-1078**

"Please restrict remarks to the publication itself. Comments pertaining to hardware/software difficulties should be referred to the local sales office. Please give specific page and line references with your comments when appropriate.

Requests for system assistance or publications should be directed to your MDS representative or to the MDS Branch Office serving your area.

---

ERRORS NOTED:

---

SUGGESTIONS FOR IMPROVEMENT:

---

How do you use this document?

- As an operator's Reference Manual
- As an introduction to the subject
- As an aid to instruction in a class
- As a student text book
- For advanced knowledge of subject

Do you wish a reply?

- Yes
- No

Your Name \_\_\_\_\_ Date \_\_\_\_\_

Occupation \_\_\_\_\_

Company \_\_\_\_\_

Address \_\_\_\_\_

Street

City

State

Zip Code

Fold

Fold



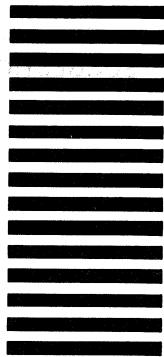
NO POSTAGE  
NECESSARY  
IF MAILED  
IN THE  
UNITED STATES

**BUSINESS REPLY MAIL**  
FIRST CLASS PERMIT NO. 73 HERKIMER, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE

**MOHAWK DATA SCIENCES CORP.**  
**LOS GATOS DEVELOPMENT LAB**  
**985 University Avenue**  
**Los Gatos, California 95030**

**ATTN: Technical Publications Department**



Cut A  
line

Fold

Fold