

Addendum

**Corrections
To
TRS-XENIX COBOL Development System
Version 01.06.xx**

Important: The following command must be entered from the root prompt for the "PRINTER" option to work from the user level:

```
chmod 666 /dev/clp <ENTER>
```

The following corrections need to be made to your TRS-XENIX COBOL Development System Manual:

User's Guide, Page 3. The -x option should read:

"Indicates a cross-reference of RM/COBOL Procedure and Data Division names is to be produced. This option is effective only when a listing (-l option) has been specified.

The default is not to generate a cross-reference."

User's Guide, Page 34. The command line should read:

```
cc -n -O -F 800 -o runcobol runcobol.o sub.c -ltermlib
```

Editor, Page 6-5.

The arrow keys do not move the cursor when entered from the console. Use the "h", "j", "k", and "l" keys instead.

Radio Shack
Part # 8759274

IMPORTANT

Installing RMCOBOL On Your Hard Disk

Please read all these directions before installing RMCOBOL on your hard disk. This lets you become familiar with the entire procedure and help make the installation smoother. During the installation be sure to watch the screen and answer all the prompts.

To begin installing the package:

1. Turn on all peripherals and then turn on the computer.
2. Log in as root (to install RMCOBOL, you must be logged in as root).
3. At the root prompt (#) type:

install **ENTER**

The screen shows Installation Menu. It also shows the prompts

1) to install or q) to quit.

4. Enter **1** to install RMCOBOL.
5. At the prompt:

Insert diskette in Drive 0 and press <ENTER>

insert your TRS-XENIX RMCOBOL diskette in Drive 0 and press **ENTER**.

The next screen shows the name of the package being installed, the version number, and the catalog number. At the bottom of the screen a welcoming message appears.

When TRS-XENIX finishes, the message:

“Installation complete — Remove the diskette, then press <ENTER>” appears. Pressing **ENTER** returns the menu to the screen. You are then prompted to press:

- 1) to install another application, or
- q) to quit.

Your installation of RMCOBOL is now complete. Press **Q** to quit and return to TRS-XENIX.

Note: You should never write protect your hard disk when running TRS-XENIX or RMCOBOL.

Radio Shack®
A DIVISION OF TANDY CORPORATION
FORT WORTH, TEXAS 76102

Radio Shack®

TRS-XENIX™ Operating System

COBOL Development System

TERMS AND CONDITIONS OF SALE AND LICENSE OF RADIO SHACK COMPUTER EQUIPMENT AND SOFTWARE
PURCHASED FROM A RADIO SHACK COMPANY-OWNED COMPUTER CENTER, RETAIL STORE OR FROM A
RADIO SHACK FRANCHISEE OR DEALER AT ITS AUTHORIZED LOCATION

LIMITED WARRANTY

I. CUSTOMER OBLIGATIONS

- A. CUSTOMER assumes full responsibility that this Radio Shack computer hardware purchased (the "Equipment"), and any copies of Radio Shack software included with the Equipment or licensed separately (the "Software") meets the specifications, capacity, capabilities, versatility, and other requirements of CUSTOMER.
- B. CUSTOMER assumes full responsibility for the condition and effectiveness of the operating environment in which the Equipment and Software are to function, and for its installation.

II. RADIO SHACK LIMITED WARRANTIES AND CONDITIONS OF SALE

- A. For a period of ninety (90) calendar days from the date of the Radio Shack sales document received upon purchase of the Equipment, RADIO SHACK warrants to the original CUSTOMER that the Equipment and the medium upon which the Software is stored is free from manufacturing defects. THIS WARRANTY IS ONLY APPLICABLE TO PURCHASES OF RADIO SHACK EQUIPMENT BY THE ORIGINAL CUSTOMER FROM RADIO SHACK COMPANY-OWNED COMPUTER CENTERS, RETAIL STORES AND FROM RADIO SHACK FRANCHISEES AND DEALERS AT ITS AUTHORIZED LOCATION. The warranty is void if the Equipment's case or cabinet has been opened, or if the Equipment or Software has been subjected to improper or abnormal use. If a manufacturing defect is discovered during the stated warranty period, the defective Equipment must be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer for repair, along with a copy of the sales document or lease agreement. The original CUSTOMER'S sole and exclusive remedy in the event of a defect is limited to the correction of the defect by repair, replacement, or refund of the purchase price, at RADIO SHACK'S election and sole expense. RADIO SHACK has no obligation to replace or repair expendable items.
- B. RADIO SHACK makes no warranty as to the design, capability, capacity, or suitability for use of the Software, except as provided in this paragraph. Software is licensed on an "AS IS" basis, without warranty. The original CUSTOMER'S exclusive remedy, in the event of a Software manufacturing defect, is its repair or replacement within thirty (30) calendar days of the date of the Radio Shack sales document received upon license of the Software. The defective Software shall be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer along with the sales document.
- C. Except as provided herein no employee, agent, franchisee, dealer or other person is authorized to give any warranties of any nature on behalf of RADIO SHACK.
- D. Except as provided herein, **RADIO SHACK MAKES NO WARRANTIES, INCLUDING WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.**
- E. Some states do not allow limitations on how long an implied warranty lasts, so the above limitation(s) may not apply to CUSTOMER.

III. LIMITATION OF LIABILITY

- A. EXCEPT AS PROVIDED HEREIN, RADIO SHACK SHALL HAVE NO LIABILITY OR RESPONSIBILITY TO CUSTOMER OR ANY OTHER PERSON OR ENTITY WITH RESPECT TO ANY LIABILITY, LOSS OR DAMAGE CAUSED OR ALLEGED TO BE CAUSED DIRECTLY OR INDIRECTLY BY "EQUIPMENT" OR "SOFTWARE" SOLD, LEASED, LICENSED OR FURNISHED BY RADIO SHACK, INCLUDING, BUT NOT LIMITED TO, ANY INTERRUPTION OF SERVICE, LOSS OF BUSINESS OR ANTICIPATORY PROFITS OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OR OPERATION OF THE "EQUIPMENT" OR "SOFTWARE". IN NO EVENT SHALL RADIO SHACK BE LIABLE FOR LOSS OF PROFITS, OR ANY INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR IN ANY MANNER ARISING OUT OF OR CONNECTED WITH THE SALE, LEASE, LICENSE, USE OR ANTICIPATED USE OF THE "EQUIPMENT" OR "SOFTWARE".

NOTWITHSTANDING THE ABOVE LIMITATIONS AND WARRANTIES, RADIO SHACK'S LIABILITY HEREUNDER FOR DAMAGES INCURRED BY CUSTOMER OR OTHERS SHALL NOT EXCEED THE AMOUNT PAID BY CUSTOMER FOR THE PARTICULAR "EQUIPMENT" OR "SOFTWARE" INVOLVED.

- B. RADIO SHACK shall not be liable for any damages caused by delay in delivering or furnishing Equipment and/or Software.
- C. No action arising out of any claimed breach of this Warranty or transactions under this Warranty may be brought more than two (2) years after the cause of action has accrued or more than four (4) years after the date of the Radio Shack sales document for the Equipment or Software, whichever first occurs.
- D. Some states do not allow the limitation or exclusion of incidental or consequential damages, so the above limitation(s) or exclusion(s) may not apply to CUSTOMER.

IV. RADIO SHACK SOFTWARE LICENSE

RADIO SHACK grants to CUSTOMER a non-exclusive, paid-up license to use the RADIO SHACK Software on **one** computer, subject to the following provisions:

- A. Except as otherwise provided in this Software License, applicable copyright laws shall apply to the Software.
- B. Title to the medium on which the Software is recorded (cassette and/or diskette) or stored (ROM) is transferred to CUSTOMER, but not title to the Software.
- C. CUSTOMER may use Software on one host computer and access that Software through one or more terminals if the Software permits this function.
- D. CUSTOMER shall not use, make, manufacture, or reproduce copies of Software except for use on **one** computer and as is specifically provided in this Software License. Customer is expressly prohibited from disassembling the Software.
- E. CUSTOMER is permitted to make additional copies of the Software **only** for backup or archival purposes or if additional copies are required in the operation of **one** computer with the Software, but only to the extent the Software allows a backup copy to be made. However, for TRSDOS Software, CUSTOMER is permitted to make a limited number of additional copies for CUSTOMER'S own use.
- F. CUSTOMER may resell or distribute unmodified copies of the Software provided CUSTOMER has purchased one copy of the Software for each one sold or distributed. The provisions of this Software License shall also be applicable to third parties receiving copies of the Software from CUSTOMER.
- G. All copyright notices shall be retained on all copies of the Software.

V. APPLICABILITY OF WARRANTY

- A. The terms and conditions of this Warranty are applicable as between RADIO SHACK and CUSTOMER to either a sale of the Equipment and/or Software License to CUSTOMER or to a transaction whereby RADIO SHACK sells or conveys such Equipment to a third party for lease to CUSTOMER.
- B. The limitations of liability and Warranty provisions herein shall inure to the benefit of RADIO SHACK, the author, owner and/or licensor of the Software and any manufacturer of the Equipment sold by RADIO SHACK.

VI. STATE LAW RIGHTS

The warranties granted herein give the **original** CUSTOMER specific legal rights, and the **original** CUSTOMER may have other rights which vary from state to state.

TRS-XENIX™ Operating System Software: Copyright 1983 Microsoft Corporation.
All Rights Reserved. Licensed to Tandy Corporation.
XENIX is a trademark of Microsoft Corporation.

Restricted rights: Use, duplication, and disclosure are subject to the terms stated in the customer
Non-Disclosure Agreement.

"tsh" and "tx" Software: Copyright 1983 Tandy Corporation. All Rights Reserved.

RMCOBOL Software: Copyright 1983 Ryan-McFarland Corporation.
All Rights Reserved. Licensed to Tandy Corporation.

COBOL Run-Time Software: Copyright 1982 Ryan-McFarland Corporation.
All Rights Reserved. Licensed to Tandy Corporation.

COBOL Development System Manual: Copyright 1983 Tandy Corporation. All Rights Reserved.

Reproduction or use without express written permission from Tandy Corporation, of any
portion of this manual is prohibited. While reasonable efforts have been taken in the
preparation of this manual to assure its accuracy, Tandy Corporation assumes no
liability resulting from any errors or omissions in this manual, or from the use of the
information contained herein.

An Overview of the Model 16 COBOL Documentation Package

This binder contains the information you need to use the Radio Shack COBOL system. It assumes you are familiar with the general operation of the Computer, including use of the TRS-XENIX™ operating system. The package includes three manuals.

RMCOBOL System Operation

Provides information on the COBOL Compiler, Runtime, and Debugger. Also includes TRS-XENIX operating system considerations in writing a COBOL program.

RMCOBOL Language Reference Guide

A complete description of the Radio Shack version of the COBOL programming language. Newcomers to COBOL should consult a standard COBOL textbook for tutorial material.

Thank You,

Radio Shack®
A DIVISION OF TANDY CORPORATION
FORT WORTH, TEXAS 76102

System Users Guide

**COBOL Development System
TRS-XENIX™ Operating System**

RM/COBOL

TRS-80 Model 16

TRS-XENIX Operating System

RM/COBOL USER'S GUIDE

January, 1983

P R E F A C E

This document contains the information required to compile, run and debug RM/COBOL language programs on the Radio Shack TRS-80 Model 16 Microcomputer under the TRS-XENIX Disk Operating System.

It assumes the reader is familiar with the RM/COBOL Language, the general operation of TRS-80 Model 16 Microcomputer and the TRS-XENIX Operating System. The reader is specifically referred to the following publications:

TRS-XENIX RM/COBOL Language Manual
TRS-XENIX Operation Guide
TRS-XENIX Reference Manuals

This guide is organized such that each chapter fully describes a particular operational procedure. While the experienced user need only refer to the appropriate chapter, it is recommended that the first-time user read the complete guide prior to operation of the RM/COBOL system.

PROPRIETARY RIGHTS NOTICE

TRS-80 Model 16 COBOL (RMCOBOL) is a proprietary product of:

Ryan-McFarland Corporation
Language Products Group
Aptos, California 95003

licensed to:

Tandy Corporation
One Tandy Center
Fort Worth, Texas 76102
(817) 390-3583
Telex: 171562

The software described in this document is furnished to the user under a license for use on a single computer system and may be copied (with inclusion of the copyright notice) only in accordance with the terms of such license.

Copyright 1982 by Ryan-McFarland Corporation. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of Tandy Corporation.

TABLE OF CONTENTS

Section	Page
CHAPTER 1 THE RM/COBOL COMPILER	1
1.1 Compiler Overview	1
1.2 Device Assignments	1
1.3 Executing the Compiler	2
1.3.1 Compiler Options	2
1.3.2 Compiler Messages	4
1.3.3 Compiler Termination	6
1.3.4 Examples	6
1.4 File System Considerations	7
1.4.1 Compiler Source Input	7
1.4.2 Compiler Listing Output	7
1.4.3 Compiler Object Output	7
1.5 The Program Listing	8
1.5.1 Listing Diagnostics	8
1.5.2 Diagnostic Messages	9
CHAPTER 2 THE RM/COBOL RUNTIME	15
2.1 Runtime Overview	15
2.2 Device Assignments	15
2.3 Executing the Compiled Program	16
2.3.1 Runtime Options	16
2.3.2 Runtime Messages	17
2.3.3 Program Termination	18
2.3.4 Examples	18
2.4 Runtime Diagnostics	19
2.5 File System Considerations	23
2.5.1 RM/COBOL Sequential Files	23
2.5.2 RM/COBOL Relative Files	23
2.5.3 RM/COBOL Indexed Files	24
2.5.4 RM/COBOL Label Processing	25
2.5.5 Record and File Locking	26
2.6 Runtime Memory Usage	27
CHAPTER 3 INTERACTIVE DEBUG	28
3.1 Debug Overview	28
3.2 User Interaction and Display	28
3.3 Debug Commands	28
CHAPTER 4 SYSTEM CONSIDERATIONS	30
4.1 The ACCEPT and DISPLAY Statements	30
4.2 The CALL Statement	32
4.3 The COPY Statement	33
CHAPTER 5 INSTALLATION	34
5.1 Installation Procedures	34
5.2 Device Synonym Assignments	35

CHAPTER 1

THE RM/COBOL COMPILER

1.1 Compiler Overview

The RM/COBOL Compiler operates on a TRS-80 Model 16 Microcomputer under the TRS-XENIX Operating System.

Once executed, the Compiler makes a single pass on the source program, generating object and listing files concurrently. Upon completion it reports compilation results to the standard error file (default terminal) and returns control to TRS-XENIX.

Compilation always proceeds to the end of the program, regardless of the number of source errors found.

A listing of the program is generated showing the original RM/COBOL source statements, error information, data allocation, Interactive Debug information and, optionally, a Cross Reference of all program labels and data items. This listing is directed to the standard output file.

The generated object file is in a form ready for immediate execution by the RM/COBOL Runtime. Object code is produced such that an attempt to execute an erroneous statement will terminate execution with an appropriate error message.

1.2 Device Assignments

During operation, the Compiler will require one or more of the following files:

Standard error	compiler messages
User specified	source input file
Standard output	listing file (optional)
User specified	object file (optional)
User specified	COPY input file (optional)

1.3 Executing the Compiler

To compile an RM/COBOL source program, issue the following command to TRS-XENIX:

```
rmcobol file [<options>...]
```

where:

file

is the filename or pathname of the RM/COBOL source file to be compiled.

<options>

allows the user to specify compiler and/or file options. Each option must be specified as shown below, separated by spaces. Either upper or lower case is acceptable.

When no options are specified the compiler will automatically generate an object file but no listing output.

1.3.1 Compiler Options

-d

Instructs the compiler to compile all RM/COBOL "Debug" source lines, identified by a "D" in column 7. This allows the user selective compilation of RM/COBOL source statements.

This option has no relationship to the RM/COBOL Runtime Interactive Debug facility and need not be specified to allow such debugging.

The default is to treat such lines as comments.

-c nn

Indicates the maximum line length for the listing file is 'nn' characters in decimal.

The default value is 80.

-e

Instructs the compiler to generate an 'Error Only' listing instead of a full listing to the standard output file.

The listing generated will contain the page heading information, all source lines in error with their appropriate undermarks and messages, plus all summary information.

The default is not to generate an error listing.

-l

Indicates that the compiler listing is to be written to the standard output file.

The default is not to generate a listing file.

-n

Indicates that no object file is to be produced.

-o objfile

Indicates that the compiler object output is to be written to the file 'objfile', where 'objfile' is a filename or pathname.

The default is to output the object to the file "cbl.out" in the current directly.

-p nn

Indicates the page size of the listing file is 'nn' lines in decimal.

The default value is 66.

-x

Indicates a cross-reference of RM/COBOL Procedure and Data Division names is to be produced. This option is effective only when a listing ('-e' or '-l' option) has been specified.

The default is not to generate a cross-reference.

1.3.2 Compiler Messages

Messages which report the compiler's status, or its ability to complete the compilation process are reported on the standard error file as they are detected.

RM/COBOL Compiler (ver v.ra) for the TRS-80 Model-16
Copyright 1982 by Ryan-McFarland Corporation. All rights reserved.

Indicates that the compiler has been loaded and has begun to execute the specified program. 'v.ra' indicates the version (v) and revision (ra) level of the compiler.

Compilation complete: eeee errors, wwww warnings

Indicates that the compilation has been completed. The values of 'eeee' and 'www' indicate the number of errors and warnings, respectively, identified in the source program. This message is repeated on the listing.

usage: rmcobol file [-c n] [-d] [-e] [-l] [-o objfile] [-p n] [-x]

Indicates that an unrecoverable error was detected on the command to execute the compiler.

The user should reenter the command with the necessary corrections.

Compilation cancelled

Compiler cancelled by operator with DELETE key.

Compiler Error, no: nnnn

An internal error has occurred which prevents continued compilation. The value of 'nnnn' identifies the condition which caused the error.

1 Pointer overflow

The program has exceeded an internal compiler limit. The total number of user defined words must be reduced. This can be accomplished through segmentation or by breaking the program into a main program with multiple subprograms.

2 Roll Memory overflow

The program has exceeded the available workspace. The program size must be reduced. This can be accomplished by reducing the size of user defined words (characters are packed 3 to a word), by segmentation, or by breaking the program into a main program with multiple subprograms.

3 Program overflow

The program has exceeded an internal compiler limit. One of the object sections has run out of space. Segment the program or break it into a main program with multiple subprograms.

4 Compiler error

An internal compiler error has been encountered.

5 Empty source file

An end of file error was encountered when trying to read the first record of the RM/COBOL source file.

1.3.3 Compiler Termination

When the compile terminates, an appropriate message will be displayed on the terminal.

In addition, the TRS-XENIX system return code will be set to indicate the result of the compile. Return codes and their definitions are:

- 0 normal termination
- 1 IO error
- 2 cancelled (DEL)
- 3 compiler error
- 4 syntax error in source
- 5 command line usage error

1.3.4 Examples

```
rmcobol -l -x payroll
```

locates and compiles the source program 'payroll' in the current working directory, producing an object file 'cbl.out', and a listing with cross reference is written to the standard output file.

```
rmcobol /z/cbl/src/mortgage -o /z/cbl/obj/mortgage
```

compiles the source program 'mortgage' located in the directory '/z/cbl/src', producing an object file 'mortgage' in the directory '/z/cbl/obj'.

1.4 File System Considerations

1.4.1 Compiler Source Input

The compiler source input consists of lines (variable length records) of ASCII text with a line-feed character as the record separator. This format is compatible with the standard TRS-XENIX editors 'ed' and 'vi', and the RM/COBOL Runtime Sequential file format. Embedded tab characters are expanded as 1 or more spaces according to default column positions 8,12,16,20,24,...72 (every 4th column 8 through 72).

1.4.2 Compiler Listing Output

The RM/COBOL listing file can be directed to the standard output device by specifying -l in the compiler command line. The default standard output device is the user's console. You can redirect the standard output to a file or device by use of the '>' operator. In every case, the file consists of lines (variable length records) of ASCII text, terminated with a line-feed character as the record separator.

1.4.3 Compiler Object Output

The RM/COBOL object file is created as a binary data file. The COBOL object file is immediately executable by the RM/COBOL Runtime Package (see THE RM/COBOL RUNTIME).

1.5 The Program Listing

The compiler outputs 'source', 'allocation', and 'summary' listings if the list option '-l' is specified. When the '-x' option is specified, a 'cross-reference' listing is also produced.

The source listing includes a sequential line number, sentence address, source image, and interspersed diagnostics.

The allocation listing includes the address, size, order, type, and name of each identifier. The identifier names are indented to show the record structure. (The order of an identifier is the number of subscripts it requires).

The summary listing includes the number of errors, the number of warnings, and the size of the program.

The cross-reference listing includes all identifier names in alphabetical order, and the line number of each declaration, source, and destination reference. The line number is surrounded by slashes if the reference is a declaration; asterisks if the reference is a possible modification. References to all paragraphs and sections are included.

In all listings, numbers in decimal are represented as ddd...d, numbers in hexadecimal are represented as >dd...d.

1.5.1 Listing Diagnostics

Source constructs are checked for syntax and semantic errors as they are scanned. Errors may cause interruption in scanning. In this case, text is ignored until a recovery point is found and a resume message is printed. Recovery points are chosen to minimize the amount of unanalyzed text without producing irrelevant error messages. In any case, the constructs at fault are undermarked and error messages listed when the source line is printed. The error message includes either E's or W's indicating error or warning. For example:

Indicates a semantic number size error but

indicates a syntax error at the first undermark and a recover at the second undermark.

The number preceding the error message is the undermark number, counting from left to right. More than one message may refer to the same undermark.

Global errors such as undefined paragraph names and illegal control transfers are listed with the program summary at the end of the source listing.

1.5.2 Diagnostic Messages

ACCESS CLASH

Nonsequential access given for sequential file.

BLANK WHEN ZERO

BLANK WHEN ZERO clause given for nonnumeric or group item.

CLASS

The referenced identifier is not valid in a class condition.

COPY

COPY statement failed because of permanent error associated with the undermarked file-name.

CORRESPONDING

The CORRESPONDING phrase cannot be used with the referenced identifier.

DATA OVERFLOW

The data area (working-storage and literals) is larger than 65535 bytes in length.

DATA TYPE

Context does not allow data type of the referenced identifier.

DEVICE CLASH

Random characteristics given to nonrandom device.

DEVICE TYPE

OPEN or CLOSE mode inconsistent with device type.

DOUBLE DECLARATION

Multiple declaration of a file or identifier attribute.

DOUBLE DEFINITION

Multiple definition of an identifier.

DUPLICATE

Warning only. Multiple USE procedure declared for same function or file.

FILE DECL ERROR

The referenced file-name is SELECTed and has an invalid or missing file description (FD).

FILE NAME ERROR

The referenced file-name has an invalid external file name declaration.

FILE NAME REQUIRED

File name not given as referenced in I/O verb.

FILE RECORD KEY ERROR

The referenced file-name has a RECORD KEY which is incorrectly qualified or is not defined as a data item of the category alphanumeric within a record description entry associated with that file name.

FILE RECORD SIZE ERROR

The referenced file-name has a declared record size which conflicts with the actual data record descriptions or is a relative organization file with variable length records.

FILE RELATIVE KEY ERROR

The referenced file-name has a RELATIVE KEY which is incorrectly qualified, is defined in a record description associated with that file-name, or is not defined as an unsigned integer.

FILE STATUS ERROR

The referenced file-name has a status 'item which is incorrectly qualified, is not defined in the WORKING-STORAGE SECTION, or is not a two-character alphanumeric item.

FILE TYPE

Access or organization of file conflicts with undermarked statement.

FILLER LEVEL

A non-elementary FILLER item is declared.

GROUP CLASH

USAGE or VALUE clause of group member conflicts with same clause for group.

GROUP VALUE CLASH

Warning Only. An item subordinate to a group with the VALUE IS clause is described with the SYNCHRONIZED, JUSTIFIED, or USAGE (other than USAGE IS DISPLAY) clause.

IDENTIFIER

Identifier reference is incorrectly constructed or the identifier has an invalid or double definition.

ILLEGAL ALTER

An ALTER statement references an unalterable paragraph or violates the rules of segmentation.

ILLEGAL PERFORM

A PERFORM statement reference undefined or incorrectly qualified paragraph or the reference violates the rules of segmentation.

INVALID ID

The referenced identifier was not successfully defined.

INVALID PARAGRAPH

Context does not allow section name.

JUSTIFY

JUSTIFY clause given in conflict with other attributes.

KEY REQUIRED

Relative key not declared for random access relative file or record key not declared for indexed file.

LABEL

Presence or absence of label record conflicts with device standards.

LEVEL

Level-number given is invalid either intrinsically or because of position within a group.

LINKAGE

An identifier in the USING clause of the PROCEDURE title is not a linkage item or a statement references a linkage item not subordinate to an identifier in the USING clause of the PROCEDURE title.

LITERAL VALUE

Literal value given is incorrect in context.

MOVE

Operands of MOVE verb specify an invalid move.

MUST BE INTEGER

Context requires decimal integer.

MUST BE PROCEDURE

Context requires procedure name either as reference or definition, or the reference must be a nondeclarative procedure-name.

MUST BE SECTION

Context requires procedure-name to be section.

NESTING

Illegal nesting of condition that is not an IF condition.

NOT IN REDEFINE

 VALUE IS clause given in REDEFINES item.

OCCURS

 OCCURS clause given at invalid level or after three have been given for the same item.

OCCURS DEPENDING ERROR

 The referenced object of a DEPENDING phrase has not been defined correctly.

OCCURS-VALUE CLASH

 VALUE IS and OCCURS in effect for the same item.

PICTURE

 Invalid PICTURE syntax.

PICTURE-BWZ CLASH

 Zero suppression and BLANK WHEN ZERO cannot be in effect for the same item.

PICTURE-USAGE CLASH

 USAGE clause or implied usage conflicts with usage implied by picture.

PROCEDURE INDEPENDENCE

 PERFORM given for procedures in independent segments not in the current segment.

PROGRAM OVERFLOW

 The instruction area is larger than 32767 bytes in length.

RECORD KEY

 Record key declared for other than an indexed organization file or a START statement KEY phrase references a data item not aligned on the declared key's leftmost byte.

RECORD REQUIRED

 Context requires record name.

REDEFINE

 REDEFINES given within an OCCURS or not redefining the last allocated item.

REDEFINES ERROR

 The referenced data-name redefines an item which does not have the same number of character positions and is not level 01.

REFERENCE INVALID

 Reference given is not valid in context.

RELATION

 Operands of relation test are incompatible.

RELATIVE KEY

Relative key declared for other than a relative organization file or a START statement KEY phrase references a data item other than the declared key.

RESERVED WORD CONFLICT

A RM/COBOL reserved word or symbol is given where a user word is required. In the summary this is only a warning about an ANSI COBOL reserved word that is not an implemented RM/COBOL reserved word.

SCAN RESUME

Warning only. Scanning was terminated at previous error message and resumes at undermarked character.

SECTION CLASH

A VALUE IS clause appears in the FILE or LINKAGE section.

SEGMENT

Warning only. Segment number given in an independent segment is not the same as the current segment or the number of a new independent segment. The current segment number is used.

SEPARATOR

Warning only. Redundant punctuation or a separator is not followed by the required space.

SIGN

SIGN clause given in conflict with usage and picture.

SIZE

Warning only. Size of data referenced not correct for context.

SIZE ERROR

Declared size of record conflicts with present reference.

SUBSCRIPT

Incorrect number of subscripts or indices for a reference.

SYNC

Synchronized clause given for a group item

SYNTAX

Incorrect character or reserved word given for context.

UNDEFINED

File referenced in FD entry was not defined.

UNDEFINED DECLARATIVE PROCEDURE

A declarative statement references a procedure not defined within the DECLARATIVES.

UNDEFINED PROCEDURE

A GO TO statement references an undefined or incorrectly qualified paragraph.

USE REQUIRED

A DECLARATIVES section must begin with a USE statement.

USING COUNT

Warning only. The item count in the USING list of a CALL statement is different from that of the first reference to the same program name.

VALUE ERROR

Value given in VALUE IS required truncation of nonzero digits.

VALUE

VALUE IS clause given in conflict with other declared attributes.

VARIABLE RECORD

Warning only. The INTO phrase is not allowed with variable size records.

CHAPTER 2

THE RM/COBOL RUNTIME

2.1 Runtime Overview

The RM/COBOL Runtime operates on a TRS-80 Model 16 Microcomputer under the TRS-XENIX Operating System.

Once invoked, the Runtime loads and executes the compiled object program, automatically loading any required segments. Concurrently, it allocates memory for file buffers, and CALLED RM/COBOL subprograms. Upon completion appropriate messages are displayed and control is returned to the operating system.

2.2 Device Assignments

During operation the Runtime will require one or more of the following files:

Standard Input ACCEPT, Interactive Debug

Standard Output DISPLAY, Interactive Debug

Standard Error Runtime messages

Device Synonyms The external filename in the SELECT statement may specify a synonym for a system file or a pipe to a system program. These special files are accessed in sequential order. See the section on Device Synonym Assignments for particular synonym prefixes.

Note that CLOSE NO REWIND requests an implicit CLOSE (at subprogram EXIT time) made by the interface to a program pipe (for example, the line printer spooler) have no effect on the pipe. An explicit CLOSE in the user program is required to close the pipe.

2.3 Executing the Compiled Program

To execute a compiled RM/COBOL object program, issue the following command to TRS-XENIX:

```
runcobol file [<options>...]
```

where:

file

is the filename or pathname of the compiled RM/COBOL object program to be executed.

<options>

allows the user to specify Runtime options. Each option must be specified as shown below, separated by spaces. Either upper or lower case is acceptable.

When no options are specified, the Runtime will execute the User's program without Interactive Debug, with all switches set to 0.

2.3.1 Runtime Options

-a

Sets the automatic line-feed flag on. This tells the Runtime that the line printer generates a line-feed after each carriage return. This can affect printer slewing (see WRITE...ADVANCING statement). Incorrect specification will cause overprinting.

The default is to set the automatic line-feed flag on.

-d

Invokes the RM/COBOL Interactive Debug package. See RM/COBOL Interactive Debug discussion, below, for operating instructions.

The default is not to invoke Interactive Debug.

-s nn..n

Sets (or resets) the value of SWITCHES in the RM/COBOL program.

Each 'n' is a switch value, 0 for off, 1 for on, numbered 1 to 8, left to right. Trailing zeroes need not be specified.

The default is to set all switches off (0).

2.3.2 Runtime Messages

Messages which report the Runtime's status, or its ability to execute the RM/COBOL program, are reported on to the standard error file (default terminal).

RM/COBOL Runtime (ver v.ra) for the TRS-80 Model-16
Copyright 1983 by Ryan-McFarland Corporation. All rights reserved

Indicates the Runtime has been loaded and has begun to execute the specified program. 'v.ra' identifies the version (v) and revision (ra) of the Runtime.

COBOL STOP RUN AT xx yyyy IN nnnnnn

This is the normal termination message of a program.

'xxxxxx' identifies the overlay (xx) and statement address (yyyy) where the program terminated. 'nnnnnn' are the first six characters of the PROGRAM-ID.

If Debug was invoked on the command line, an 'S' Debug command may be used to cause Debug to exit to the operating system.

COBOL STOP literal AT xx yyyy IN nnnnnn
CONTINUE (Y/N)?

This message indicates that a STOP 'literal' statement has been encountered. 'xxxxxx' identifies the overlay (xx) and statement address (yyyy) where the program terminated. 'nnnnnn' are the first six characters of the PROGRAM-ID.

Responding with a 'y' will be the equivalent of a "pause" statement, returning control to the next RM/COBOL statement.

An 'n' response will cause all program files to be closed and control returned to the operating system.

2.3.3 Program Termination

When the RM/COBOL object program terminates, an appropriate message will be displayed on the terminal.

In addition, the TRS-XENIX system return code will be set to indicate the result of the program's execution. Return codes and their definitions are:

0	normal termination
1-255	User defined (STOP RUN)
-1	IO error
-2	cancelled (DEL)
-3	Runtime error
-4	program load fail
-5	command line usage error

2.3.4 Examples

runcobol PAYROLL -s 1011

locates, loads, and executes the RM/COBOL object program PAYROLL and sets the value of SWITCHES 1, 3, and 4 'on', all others 'off'.

runcobol /z/cbl/obj/mortgage -d

loads the RM/COBOL object program 'mortgage' from the directory '/z/cbl/obj' and invokes the Interactive Debug package. Control is passed directly to Debug.

2.4 Runtime Diagnostics

Diagnostic messages are published on the standard error device if an internal error occurs, or if an I/O error occurs that was not, or could not, be processed by an appropriate USE procedure.

If Debug was invoked, Debug will be entered to allow examination of program data values; otherwise, control will return to the operating system.

COBOL error AT xxYYYY IN nnnnnn

Indicates an internal error condition has occurred, where 'error' identifies the error condition. 'xxYYYY' identifies the overlay (xx) and statement address (YYYY) where the program terminated. 'nnnnnn' are the first six characters of the PROGRAM-ID.

COBOL filename IO ERROR = cc/dd AT xxYYYY IN nnnnnn

Identifies that an abnormal I/O condition, 'cc' has caused the program to be aborted. 'xxYYYY' identifies the overlay (xx) and statement address (YYYY) where the program terminated. 'nnnnnn' are the first 6 characters of the PROGRAM-ID.

The I/O error 'cc' has a different meaning depending on whether the file's organization is sequential, relative or indexed. 'dd' will always be zero and is reserved for use in a future release.

Sequential Files:

10 AT END.

The sequential READ statement was unsuccessfully executed as a result of an attempt to read a record when no next logical record exists in the file.

30 PERMANENT ERROR.

The input-output statement was unsuccessfully executed as the result of an input-output error, such as data check parity error, or transmission error. May also indicate attempted execution of an instruction not implemented in the Runtime (REWRITE, CLOSE REEL).

34 PERMANENT ERROR BOUNDARY VIOLATION.

The input-output statement was unsuccessfully executed as the result of a boundary violation for a sequential file.

- 90 INVALID OPERATION.
An attempt has been made to execute a READ, WRITE, or REWRITE statement that conflicts with the current open mode or a REWRITE statement was not preceded by a successful READ statement.
- 91 FILE NOT OPENED.
An attempt has been made to execute a DELETE, READ, START, UNLOCK, WRITE, REWRITE or CLOSE statement on a file which is not currently open.
- 92 FILE NOT CLOSED.
An attempt has been made to execute an OPEN statement on a file which is currently open.
- 93 FILE NOT AVAILABLE.
An attempt has been made to execute an OPEN statement for a file closed with LOCK.
- 94 INVALID OPEN.
An attempt has been made to execute an OPEN statement for a file with no external correspondence or a file having inconsistent parameters.
- 95 INVALID DEVICE.
An attempt has been made to execute a CLOSE REEL statement, in which case the file has been closed as if the REEL phrase had not been specified, or to execute an OPEN statement for a file which is assigned to a device in conflict with the externally assigned device.
- 96 UNDEFINED CURRENT RECORD POINTER STATUS.
An attempt has been made to execute a READ statement after the occurrence of an unsuccessful read statement without an intervening successful CLOSE and OPEN.
- 97 INVALID RECORD LENGTH.
An attempt has been made to execute a REWRITE statement when the new record length is different from that of the record to be rewritten, or to OPEN a file that was defined with a maximum record length different from the externally defined maximum record length, or to execute a WRITE statement that specifies a record with a length smaller than the minimum or larger than the maximum record size.
- 99 RECORD LOCKED
An attempt has been made to READ a record which is locked. This error is returned only if an applicable USE procedure and a FILE STATUS data item are declared for the file. Otherwise, the READ statement is retried until the record is unlocked.

Relative and Indexed Files:

02 SUCCESSFUL OPERATION BUT KEY HAS DUPLICATE.

For a read operation, it indicates that the next record associated with the current key of reference has the same key value. For a write or rewrite operation, it indicates that the record just written created a duplicate key value for at least one alternate key for which duplicates are allowed.

10 AT END.

The Format 1 READ statement was unsuccessfully executed as a result of an attempt to read a record when no next logical record exists in the file.

21 SEQUENCE ERROR FOR A SEQUENTIALLY ACCESSED INDEXED FILE.

The ascending sequence requirement of successive record key values has been violated or the record key value has been changed by the RM/COBOL program between the successful execution of a READ statement and the execution of the next REWRITE statement for that file.

22 DUPLICATE KEY VALUE.

An attempt has been made to WRITE a record that would create a duplicate key on a file that does not allow duplicates.

23 NO RECORD FOUND.

An attempt has been made to access a record, identified by a key, and that record does not exist in the file.

24 BOUNDARY VIOLATION.

An attempt has been made to WRITE beyond the externally-defined boundaries of a file.

30 PERMANENT ERROR.

The input-output statement was unsuccessfully executed as the result of an input-output error, such as data check, parity error, or transmission error.

90 INVALID OPERATION.

An attempt has been made to execute a DELETE, READ, REWRITE, START, or WRITE statement which conflicts with the current open mode of the file or a sequential access DELETE or REWRITE statement not preceded by a successful read statement.

91 FILE NOT OPENED.

An attempt has been made to execute a CLOSE, DELETE, READ, REWRITE, START, UNLOCK, or WRITE statement on a file which is not in an open mode.

92 FILE NOT CLOSED.

An attempt has been made to execute an OPEN statement on a file that is currently open.

- 93 FILE NOT AVAILABLE.
An attempt has been made to execute an OPEN statement on a file closed with lock.
- 94 INVALID OPEN.
An attempt has been made to execute an OPEN statement for a file with no external correspondence or a file having inconsistent parameters.
- 95 INVALID DEVICE.
An attempt has been made to execute an OPEN statement on a file whose device description conflicts with the externally assigned device. The device must be RANDOM and the external correspondence must be a disk.
- 96 UNDEFINED CURRENT RECORD POINTER.
An attempt has been made to execute a Format 1 READ statement when the current record pointer has an undefined state. This can occur only as the result of a preceding unsuccessful READ or START statement.
- 97 INVALID RECORD LENGTH.
An attempt has been made to execute a REWRITE statement and the new record length is different from that of the record to be rewritten, or to OPEN a file that was defined with a maximum record length different from the externally defined maximum record length, or to execute a WRITE statement that specifies a record with a length smaller than the minimum or larger than the maximum record size.
- 98 INVALID INDEX.
An input-output statement on an indexed organization file was unsuccessful as a result of invalid data in the index. This can result if the externally assigned file is not an index organization file or if an undetected input-output error has occurred.
- 99 RECORD LOCKED
An attempt has been made to READ a record which is locked. This error is returned only if an applicable USE procedure and a FILE STATUS data item are declared for the file. Otherwise, the READ statement is retried until the record is unlocked.

2.5 File System Considerations

Three types of files are supported in RM/COBOL: sequential, relative (random), and indexed sequential.

Files are specified in the user's program in a manner consistent with TRS-XENIX filenames or pathnames.

2.5.1 RM/COBOL Sequential Files

RM/COBOL sequential files consist of a serially accessible set of 'logical' records. These 'logical' records exist as variable length records, and are terminated with a line feed (>0A). Files created by the editor are of this form.

RM/COBOL sequential files are accessible only in serial order. Each 'logical' record can have a maximum length of 2048 bytes.

For sequential files the 'BLOCK CONTAINS' clause defines the number of bytes that will be read or written on each system I-O request. If this clause is omitted the default BLOCK size will be the maximum record size. In any case, the programmer need not worry about 'logical' records spanning across 'block' boundaries as this is handled by the Runtime system.

For sequential files, the device-type assignment has special meaning. If the file resides on disk, the device-type should be 'RANDOM' as this is the only device that supports 'REWRITE' and record-locking, which require that we reposition to the start of the block which contains the 'logical' record. For all other device-types, the record pointer is only advanced (for example tape files should be assigned to 'INPUT-OUTPUT', card-reader as 'INPUT' etc.) and therefore, do not support 'REWRITE' and record-locking.

2.5.2 RM/COBOL Relative Files

RM/COBOL relative files are addressable randomly by 'logical' record number. These files can only exist on the disk as fixed length records.

RM/COBOL relative file 'logical' records are internally formatted, and can be created and/or accessed only by RM/COBOL programs. Each 'logical' record can have a maximum length of 2048 bytes.

Relative files are dynamically allocated or extended as required. The number of bytes required for a relative file can be calculated as follows:

$$\text{NUMBER OF BYTES} = (\text{declared record size} + 2) * R$$

where

R = maximum number of records expected in the file.

For relative files the 'BLOCK CONTAINS' clause has no special meaning and the device-type must be 'RANDOM'.

2.5.3 RM/COBOL Indexed Files

RM/COBOL indexed files are created and maintained by the RM/COBOL Runtime.

RM/COBOL indexed files are internally formatted, and can be created and/or accessed only by RM/COBOL programs. Each 'logical' record can have a maximum of 2048 bytes.

Indexed files contain an index structure for each key specified interspersed with the data records. The use of ALTERNATE KEYS can have a geometric increase on the time required to create the file, however, access time will be relatively constant throughout the file.

Indexed files are dynamically allocated or extended as required by TRS-XENIX. If you want to estimate the size of the file, the formula shown below can be used to calculate the approximate number of 256 byte records (NRECS) that will be required.

Records and B-tree index formation reside in an indexed file, in no prescribed order, and the equation is in three parts to reflect this.

Variables - R = maximum number of records
 S = maximum size of record (in bytes)
 D = number of keys that allow duplicates
 K₁ = size of first key
 K₂ = size of second key
 K_n = size of nth key

Subtotals - A = Int((Int((S+33)/32)*R/8)+1)
 B = Int((R*D/8)+1)
 T₁ = Int((R*2/Int(252/(K₁+8)))+1)
 T_n = Int((R*2/Int(252/(K_n+8)))+1)

Best case usage NRECS = A+(B/3)+((T₁+T₂...+T_n)/2)
Worst case usage NRECS = A+B+T₁+T₂...+T_n

Int(A) is the whole part of A (eg: Int (3.1416)=3).

Example:

An indexed file has records 50 bytes in length with a primary key of 10 bytes, an alternate key with no duplicates of 5 bytes, and an alternate key of 4 bytes with duplicates, and is expected to have at most 1000 records.

Variables - R = 1000
 S = 50
 D = 1
 K1 = 10
 K2 = 5
 K3 = 4

Subtotals - A = Int((Int((50+33)/32)*1000/8+1)
 = Int((2*1000/8)+1)=251
 B = Int((1000*1/8)+1)=126
 T1 = Int((1000*2/Int(252/(10+8)))+1)
 = Int((1000*2/14)+1)=143
 T2 = Int((1000*2/Int(252/(5+8)))+1)
 = Int((1000*2/19)+1)=106
 T3 = Int((1000*2/Int(252/(4+8)))+1
 = Int((1000*2/21)+1)=96

Best case NRECS = 251+(126/3)+((143+106+96)/2) = 466
Worst case NRECS = 251+126+143+106+96 = 722

Note: This calculation provides an approximation for estimating the size of the file. TRS-XENIX will automatically create and/or extend the file as necessary to the physical limits of the disk.

For indexed files the 'BLOCK CONTAINS' clause has no special meaning and the device-type must be RANDOM.

2.5.4 RM/COBOL Label Processing

The RM/COBOL language allows the specification of the existence and processing of label records on file type devices, however, no additional label processing is performed unique to RM/COBOL programs or files.

References to label processing in the file description entry (FD), OPEN statement, and CLOSE statement, are checked for correct syntax by the compiler. They are largely ignored by the Runtime except that appropriate error codes will be returned, and any applicable USE procedures will be executed.

2.5.5 Record and File Locking

Currently only record locking is supported by the Runtime environment.

Record locking takes place as follows:

All READ requests test for a lock condition before reading the record. If the READ request is specified as without lock (READ WITH NO LOCK) the record will be released (unlocked) at the conclusion of the read operation. Otherwise the record will remain locked until a subsequent READ, WRITE, REWRITE or UNLOCK operation takes place to the same file.

Note that sequential record locking is accomplished by locking the BLOCK in which the logical record starts and thus may have the side-effect of locking a preceding or following logical record for the duration of the lock.

Sequential record locking is only in effect if the device-type is declared to be RANDOM (i.e. disk)! For this reason care should be taken not to mix device types when actually referring to the same file. For relative and indexed files, the device-type must be RANDOM, so record locking is always enforced.

Note also that for reads only, the information that a record is unavailable due to locking by another process can be obtained by means of the USE statement in the Declaratives Section of the Procedure Division associated with a given file or mode of access, as follows:

When no USE statement is present, the RM/COBOL Runtime will wait indefinitely until the lock-out situation is cleared.

When the USE statement applying to a particular READ operation is present, an error return code of 99, "record locked" will be returned. (See RM/COBOL Language Manual, the USE Statement).

Note that if the file organization is INDEXED, random REWRITES should be avoided as they can result in a deadlock situation. The REWRITE operation should be preceded by a successful READ operation.

2.6 Runtime Memory Usage

The runtime system, 'runcobol', is split into two segments, code and data. The size of these segments can be determined by using the TRS-XENIX 'size' command. Currently the code segment occupies about 37K bytes while the data segment requires about 14K bytes.

The object code produced by the RM/COBOL compiler is currently restricted to 16-bit addressing. This implies that the total space required by all COBOL object programs that must reside in memory concurrently (i.e. CALLED) can not exceed 64K bytes. In practice, this number is actually about 48K bytes, as the runtime data space is included in this 64K. Given the CALL and SEGMENTATION facilities of RM/COBOL, it should be realized that the virtual size of the application should not be constrained by this limitation.

Runtime Memory is managed as follows:

Before an RM/COBOL object program is loaded, its memory requirements are determined and a call is made to allocate the desired block of memory. Conversely, when the program has finished execution, the space is deallocated.

When a file is opened from within an RM/COBOL object program, any buffer space required for the file will be allocated at that time. Conversely, when the file is closed, the space is deallocated.

The amount of space required for an RM/COBOL object program is equal to:

Program Control Block (PCB) +
Total Byte Size +
Size of the largest INDEXED record for this program.

where the size of the PCB is equal to 28 bytes plus the string length of the external program name and the Total Byte Size corresponds to the total byte size given at the end of the compilation.

The amount of space required for an open data file is equal to:

File Control Block (FCB) +
BLOCK SIZE

where the size of the FCB is equal to 32 bytes plus the string length of the external file name and the BLOCK SIZE is zero for RELATIVE and INDEXED files. For SEQUENTIAL files, the BLOCK SIZE is the number of bytes defined by the 'BLOCK CONTAINS' clause, or if this clause is omitted, the maximum record size.

CHAPTER 3

INTERACTIVE DEBUG

3.1 Debug Overview

RM/COBOL Interactive Debug is invoked when the user specifies the '-d' option on the RUNCOBOL statement. Debug is given control and supervises the execution of the user's program.

3.2 User Interaction and Display

All Debug commands, and all resultant displays, are through standard input and standard output.

Debug will request command input by a prompt of the form

nnnnnnn xxyyyy

where 'nnnnnnn' are the first 6 characters of PROGRAM-ID, 'xx' is the overlay number, and 'yyyy' is the hexadecimal location within the specified overlay that will be executed next.

The values of 'xx' and 'yyyy' are taken directly from the Debug column in the source listing for program 'nnnnnnn'.

3.3 Debug Commands

All commands are specified by a single character, optionally followed by one or more arguments. All numeric arguments are in hexadecimal unless otherwise noted.

Invalid commands will be rejected with 'ERROR' displayed; corrected input will be requested with a reprompt.

A[xx]yyyy[,nnnnnn] Address stop.

Executes object instructions until overlay number 'xx' and location 'yyyy' in program nnnnnnn is to be executed. Debug will regain control immediately prior to the execution of the specified RM/COBOL sentence, and request further command input.

If 'xx' is specified, 'yyyy' must be fully four hexadecimal digits. If 'nnnnnnn' is omitted, it is assumed to be the first six characters of the program-id of the currently executing program.

S[n]

Single step sentence.

Execute 'n' RM/COBOL sentences and return to the Debug monitor.

The decimal argument 'n' specifies the number of RM/COBOL sentences to be executed before returning to Debug.

Dxxxx,yyyy[,tttt] Dump by type.

Display the RM/COBOL data item starting at hexadecimal location 'xxxx' of decimal length 'yyyy' and type 'tttt'. The values for 'xxxx', 'yyyy', and 'tttt' are directly from the first three columns of the allocation map. 'tttt' may be one of the following:

NSU	NPS
NTS	ABS
NCU	ANS
NCS	GRP
NBS	ANSE
NSE	HEX (hexadecimal)

Dump display has the format:

xxxx tttt dddd....

where dddd = data in the specified format

Note: Only items in the currently executing program can be displayed. This does not include linkage items.

Q

Quit Execution.

Terminate Debug and force an immediate STOP RUN. Enter 'S' to return to TRS-XENIX.

E

Exit

Exit the Debugger. Continue normal execution as if the debugger had not been invoked on the command line.

CHAPTER 4

SYSTEM CONSIDERATIONS

4.1 The ACCEPT and DISPLAY Statements

The ACCEPT and DISPLAY statements support the transfer of data between the console and the user's program data area. These statements allow the specification of general phrases which may not be supported on every CRT.

Phrases which are not supported will compile correctly, but will be ignored at Runtime, causing no operation to take place.

If the ON EXCEPTION phrase is specified on the ACCEPT statement, the exception variable receives the ASCII value of the key the operator entered to terminate the data. Values of 0, OAH, and ODH (respectively: automatic return, line feed, carriage return) are regarded as normal terminations and the exception imperative statement is not executed. Abnormal terminations are all other non-printable ASCII characters, below 020H and above 07EH, and terminations of this kind will execute the exception imperative statement.

Certain keys as defined by the TRS-XENIX stty system command affect the operation of the ACCEPT statement, including:

erase	Erases the current character and moves the cursor back one position.
kill	Backspace to the beginning of the field, erasing all characters in the field.

If no UNIT parameter is specified, the ACCEPT statement will read from standard input with character echo, and the DISPLAY statement output is directed through standard output. A specified UNIT is used as an index into the file /etc/ttys to locate an alternate device. UNIT 0 is the first entry in the table, UNIT 1 is the second, and so on. The only tty devices that can be referenced through the UNIT parameter are those that are not being used by the TRS-XENIX operating system for a login port. Login port entries are activated by the digit 1 being the first character of the /etc/ttys entry, hence the valid UNIT devices are the deactivated entries that begin with the digit 0.

Cursor motion and alternate video sequences are taken from the Termcap capability data base, modified to include sequences required by ACCEPT and DISPLAY. The terminal type used to access Termcaps is defined in the environment string TERM of the process that initiates the Runtime. The same sequences are used on all units.

It is not necessary to change /etc/termcap to use the Runtime, however, to support the HIGH, LOW, BLINK, and REVERSE modifiers of ACCEPT and DISPLAY the following additional capabilities must be added for individual terminal types in the file (capital letters are used to avoid ambiguities with existing capabilities):

NM - normal (high intensity, normal video, no blinking)
NB - normal blink (high intensity)
NR - normal reverse (high intensity)
NS - normal blink and reverse (high intensity)
AL - alternate (low intensity, normal video, no blinking)
AB - alternate blink (low intensity)
AR - alternate reverse (low intensity)
AS - alternate blink and reverse (low intensity)
OV - overhead = maximum number of screen positions occupied by the above defined field attribute modifiers
CF - sequence that makes the cursor invisible
CN - sequence that makes the cursor visible

The following is an example of how the capabilities can be added. These lines may be added to the adm31 entry in /etc/termcap to provide the correct attribute sequences for that terminal model:

```
:NM=\E(\EG0:NB=\E(\EG2:NR=\E(\EG4:NS=\E(\EG6:\
:AL=\E)\EG0:AB=\E)\EG2:AR=\E)\EG4:AS=\E)\EG6:OV#1:
```

All fields ACCEPTed or DISPLAYed on the CRT have an attribute sequence output at the beginning and end of each field.

The cursor is turned off when the Runtime begins and is turned on during an ACCEPT, and when the Runtime exits.

4.2 The CALL Statement

When 'CALLED' the first time, RM/COBOL programs are loaded by the Runtime and entered at their initial location. These 'CALLED' programs remain in memory as long as the 'CALLing' program does not reference another RM/COBOL object program. Therefore, subsequent CALLS from the 'CALLing' program will enter the 'CALLED' program directly without requiring the program to be reloaded.

Once the 'CALLing' program references another RM/COBOL object program, the previously CALLED program and any of its related programs, are discarded, and the sequence repeats itself.

Regardless of the 'CALLing' sequence, any data files not explicitly closed by a program are closed by the interface upon 'EXIT' from that program. The spooler is handled in a special manner, and is not closed at EXIT time by the interface. An explicit CLOSE in the program is necessary to release the spooled output.

A user written function 'sub' provides a manner to execute 'C' subroutines or assembler subroutines that are 'C' callable. This function is called before searching for an RM/COBOL object program, and the function returns three kinds of values:

- 1 - continue search for RM/COBOL object program
- 0 - execution complete, return to calling program
without error
- >0 - execution complete, signal a stop run with value
returned as the exit value

RM/COBOL programs that are to be 'CALLED' must have been previously compiled. The name specified in the CALL statement must be sufficient to search and locate the desired file. The default directory is the current directory.

The USING arguments of the CALL statement are made available to 'sub' as an argument count, argc, and an array of pointers to the actual arguments, argv. By convention, argv[0] is a pointer to the filename string referenced in the CALL statement, so argc is always greater than 0.

The format of each of the RM/COBOL arguments depends on its dataname PICTURE definition; see the RM/COBOL Language Manual, 'the PICTURE Clause'.

An example 'sub' function, written in 'C', has been supplied as part of the Runtime package. This function provides an interface into the TRS-XENIX system from within RM/COBOL. This function can be rewritten to perform any number of tasks, expanding the capability of the Runtime system. All that is required, is that the new function be linked with the other Runtime object module to produce the new Runtime system.

4.3 The COPY Statement

The COPY statement provides the facility to copy (include) RM/COBOL source text from a user-specified file into the source program. The complete file is copied into the program, without change, at the location of the COPY statement.

The file to be copied is identified in the RM/COBOL program by the statement

```
COPY file-name
```

or

```
COPY path-name
```

A filename consisting only of letters and numbers (first character must be letter) can be written without surrounding quotes. All other forms must be surrounded by quotes.

Examples:

```
IDENTIFICATION DIVISION.  
    COPY      STID.  
ENVIRONMENT DIVISION.  
    COPY      "/COBOL/LIBRARY/STDENVIR"  
DATA DIVISION.  
    COPY      "/any/directory/stddata"
```

CHAPTER 5

INSTALLATION

5.1 Installation Procedures

Installation of RM/COBOL requires that the object modules be copied from the Software Distribution Media to the appropriate directory. This is accomplished by following the instructions for installing an application as described in the TRS-XENIX Operations Guide. NOTE: 'nn' indicates the current release level, i.e., release 1.6 will be '16'.

The compiler modules are:

```
bin/rmcobol  
lib/rmcbl0nn  
lib/rmcbl1nn  
lib/rmcbl2nn  
lib/rmcbl3nn  
lib/rmcbl4nn
```

These are all the modules needed to compile RM/COBOL source programs. No further processing is required.

The Runtime modules are:

```
bin/runcobol  
runcobol.o  
sub.c
```

The module 'sub.c' contains the 'C' source for a dummy routine that signals the Runtime to continue searching for an RM/COBOL object subroutine. This module can be expanded as discussed in the section 'The CALL Statement' to provide 'C' or assembler subroutines. Any new version of this module must be linked into the Runtime using the following command:

```
cc -n -o bin/runcobol runcobol.o sub.c -ltermplib
```

At this point any outstanding patches to the system must be applied to the newly created executable file.

As with all Software Distribution Media, the user should save it in a secure location in case re-creation is required.

5.2 Device Synonym Assignments

To provide device name independence at Runtime when dealing with special files such as the line printer or tape device, a synonym table has been provided. The synonym table is used to map a standard name assigned to a particular device into the corresponding system file name at open time.

For example, you might always want to refer to the PRINT device by the external file name "PRINTER", even though on the system it might actually be called '/dev/tty7' or in the case of the spooler '/bin/lpr'. The synonym table gives you this capability. The synonym's assigned for this system are defined below.

Note that these names are prefixes in the sense that the external filename string "PRINTERSPOOLER" is matched by the name synonym prefix string "PRINTER".

Synonym Prefix	System Equivalent	Pipe/File
SPOOLER	/bin/lpr	pipe
PRINTER	/dev/lp	file

Language Reference Manual

COBOL Development System

RM/COBOL

COBOL LANGUAGE MANUAL

First Edition

PREFACE

This document contains the information required to develop COBOL language programs using the Ryan-McFarland Corporation RM/COBOL* Compiler. This document is reference in nature and assumes the user is familiar with the COBOL language.

The reader is specifically referenced to the RM/COBOL User's Guide applicable to the Operating System to be used.

COPYRIGHT NOTICE

Ryan-McFarland COBOL (RM/COBOL) is a proprietary product of:

Ryan-McFarland Corporation
Software Products Group
3233 Valencia Avenue
Aptos, California 95003
(408) 662-2522

The software described in this document is furnished to the user under a license for use on a single computer system and may be copied (with inclusion of the copyright notice) only in accordance with the terms of such license.

Copyright 1980 by Ryan-McFarland Corporation. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of Ryan-McFarland Corporation.

*RM/COBOL is a registered trademark of Ryan-McFarland Corporation.

ACKNOWLEDGEMENT

Much of the material in this manual is extracted from the ANSI X3.23-1974 COBOL Standard. Accordingly, the following acknowledgement is made as required in that document.

COBOL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations.

No warranty, expressed or implied, is made by any contributor or by the CODASYL Programming Language Committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contributor, or by the committee, in connection therewith.

The authors and copyright holders of the copyrighted material used herein

FLOW-MATIC (trademark of Sperry Rand Corporation), Programming for the UNIVAC I and II, Data Automation Systems copyrighted 1958, 1959, by Sperry Rand Corporation; IBM Commercial Translator Form No. F28-8013, copyrighted 1959 by IBM; FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell

have specifically authorized the use of this material in whole or in part, in the COBOL specifications. Such authorization extends to the reproduction and use of COBOL specifications in programming manuals or similar publications.

TABLE OF CONTENTS

	Page
I. INTRODUCTION.....	1
INTRODUCTION TO COBOL.....	3
What is COBOL?.....	3
The History of COBOL.....	4
The Standardization of COBOL.....	5
CONVENTIONS USED IN THIS MANUAL.....	6
Words.....	6
Brackets and Braces.....	6
Ellipses.....	6
Punctuation.....	7
Special Characters.....	7
System Dependent Information.....	7
II. THE STRUCTURE OF THE COBOL LANGUAGE.....	9
THE LANGUAGE STRUCTURE.....	11
Character Set.....	11
Separators.....	13
Character-Strings.....	14
COBOL Words.....	14
User Words.....	15
Reserved Words.....	18
Literals.....	21
Picture String.....	22
Comment-Entry.....	22
System Names.....	22
THE PROGRAM STRUCTURE.....	23
Source Format.....	23
Statements.....	25
Sentences.....	26
Clauses and Entries.....	26
Paragraphs.....	27
Sections.....	27
Divisions.....	27
THE COPY STATEMENT.....	28

III. IDENTIFICATION DIVISION.....	31
INTRODUCTION.....	33
PROGRAM IDENTIFICATION.....	33
The PROGRAM-ID Paragraph.....	34
The AUTHOR, INSTALLATION, DATE-WRITTEN, SECURITY Paragraphs.....	34
IV. ENVIRONMENT DIVISION.....	35
INTRODUCTION.....	36
CONFIGURATION SECTION.....	36
The SOURCE-COMPUTER Paragraph.....	38
The OBJECT-COMPUTER Paragraph.....	38
The SPECIAL-NAMES Paragraph.....	39
INPUT-OUTPUT SECTION.....	41
The FILE-CONTROL Paragraph.....	41
The Sequential File Control Entry.....	42
The Relative File Control Entry.....	44
The Indexed File Control Entry.....	46
The I-O CONTROL Paragraph.....	49
V. DATA DIVISION.....	51
INTRODUCTION.....	53
FILE SECTION.....	55
The File Description Entry.....	56
The BLOCK CONTAINS Clause.....	57
The RECORD CONTAINS Clause.....	58
The LABEL RECORD Clause.....	59
The VALUE OF Clause.....	59
The DATA RECORDS Clause.....	60
WORKING-STORAGE SECTION.....	61
LINKAGE SECTION.....	61
RECORD DESCRIPTION ENTRY.....	62
Level-Numbers.....	62
Elementary Items.....	62
77 LEVEL DESCRIPTION ENTRY.....	63

THE DATA DESCRIPTION ENTRY.....	64
The Level-Number.....	67
The Data Name or FILLER Clause.....	68
The REDEFINES Clause.....	69
The PICTURE Clause.....	71
The USAGE Clause.....	82
The SIGN Clause.....	84
The OCCURS Clause.....	85
The SYNCHRONIZED Clause.....	87
The JUSTIFIED Clause.....	89
The BLANK WHEN ZERO Clause.....	90
The VALUE IS Clause.....	91
The RENAMES Clause.....	94
 DATA STRUCTURES.....	96
Classes of Data.....	96
Representation of Numeric Items.....	97
Representation of Algebraic Signs.....	97
Standard Alignment Rules.....	98
 QUALIFICATION.....	99
 SUBSCRIPTING.....	101
 INDEXING.....	102
 IDENTIFIER.....	103
 CONDITION-NAME.....	104
 TABLE HANDLING.....	105
 VI. PROCEDURE DIVISION.....	109
 THE PROCEDURE DIVISION.....	111
Structure.....	112
Declaratives.....	113
Procedures.....	113
Execution.....	113
 PROCEDURE REFERENCES.....	114
 SEGMENTATION.....	116
Segments.....	116
Segmentation Classification.....	117
Segmentation Control.....	117
Restrictions on Program Flow.....	117
 THE USE STATEMENT.....	119

ARITHMETIC STATEMENTS.....	121
Arithmetic Expressions.....	121
Arithmetic Operators.....	122
Formation and Evaluation Rules.....	122
CONDITIONALS.....	124
Relation Condition.....	124
Class Condition.....	127
Condition-name (Conditional Variable).....	128
Switch-Status Condition.....	128
Complex Conditions.....	128
Negated Simple Conditions.....	129
Combined and Negated Combined Conditions.....	129
Condition Evaluation Rules.....	130
SEQUENTIAL ORGANIZATION INPUT-OUTPUT.....	132
Function.....	132
Organization.....	132
Access Mode.....	132
Current Record Pointer.....	132
I-O Status.....	133
RELATIVE ORGANIZATION INPUT-OUTPUT.....	135
Function.....	135
Organization.....	135
Access Modes.....	135
Current Record Pointer.....	136
I-O Status.....	136
The INVALID KEY Condition.....	138
The AT END Condition.....	139
INDEXED ORGANIZATION INPUT-OUTPUT.....	140
Function.....	140
Organization.....	140
Access Modes.....	140
Current Record Pointer.....	141
I-O Status.....	141
The INVALID KEY Condition.....	144
The AT END Condition.....	144

PROCEDURAL STATEMENTS.....	145
ACCEPT...FROM Statement.....	145
ACCEPT Statement (Terminal I-O).....	146
ADD Statement.....	152
ALTER Statement.....	156
CALL Statement.....	157
CLOSE Statement (Sequential I-O).....	159
CLOSE Statement (Relative and Indexed I-O) ..	161
COMPUTE Statement.....	162
DELETE Statement (Relative and Indexed I-O) ..	164
DISPLAY Statement (Terminal I-O).....	165
DIVIDE Statement.....	168
EXIT Statement.....	171
GO TO Statement.....	172
IF Statement.....	173
INSPECT Statement.....	175
MOVE Statement.....	183
MULTIPLY Statement.....	188
OPEN Statement (Sequential I-O).....	190
OPEN Statement (Relative and Indexed I-O) ...	194
PERFORM Statement.....	198
READ Statement (Sequential I-O).....	209
READ Statement (Relative and Indexed I-O) ...	211
REWRITE Statement (Sequential I-O).....	215
REWRITE Statement (Relative and Indexed I-O) ..	216
SET Statement.....	218
START Statement (Relative and Indexed I-O) ..	220
STOP Statement.....	222
SUBTRACT Statement.....	223
UNLOCK Statement.....	227
WRITE Statement (Sequential I-O).....	228
WRITE Statement (Relative and Indexed I-O) ..	231
 APPENDIX A: ERROR MESSAGES.....	235
 APPENDIX B: RESERVED WORDS.....	245
 APPENDIX C: GLOSSARY.....	251
 APPENDIX D: COMPOSITE LANGUAGE SKELETON.....	277
 APPENDIX E: SAMPLE PROGRAMS.....	305

TABLE OF CONTENTS

	Page
I. INTRODUCTION.....	1
INTRODUCTION TO COBOL.....	3
What is COBOL?.....	3
The History of COBOL.....	4
The Standardization of COBOL.....	5
CONVENTIONS USED IN THIS MANUAL.....	6
Words.....	6
Brackets and Braces.....	6
Ellipses.....	6
Punctuation.....	7
Special Characters.....	7
System Dependent Information.....	7
II. THE STRUCTURE OF THE COBOL LANGUAGE.....	9
THE LANGUAGE STRUCTURE.....	11
Character Set.....	11
Separators.....	13
Character-Strings.....	14
COBOL Words.....	14
User Words.....	15
Reserved Words.....	18
Literals.....	21
Picture String.....	22
Comment-Entry.....	22
System Names.....	22
THE PROGRAM STRUCTURE.....	23
Source Format.....	23
Statements.....	25
Sentences.....	26
Clauses and Entries.....	26
Paragraphs.....	27
Sections.....	27
Divisions.....	27
THE COPY STATEMENT.....	28

I

INTRODUCTION

INTRODUCTION TO COBOL

What is COBOL?

COBOL (COmmon Business Oriented Language) is an English oriented programming language designed primarily for developing business applications on computers. It is described as English oriented because its free form enables a programmer to write in such a way that the final result can be read easily and the general flow of the logic can be understood by persons not necessarily as closely allied with the details of the problem as the programmer himself.

Because COBOL is a programming language it can be translated to serve as communication between the programmer and the computer. The COBOL program (the source program) which has been written by the programmer is input to the COBOL compiler. The COBOL compiler then translates the COBOL program into a machine readable form (the object program).

Although each computer has its own unique COBOL compiler program, an industry-wide COBOL effort has resulted in a degree of compatibility so that a COBOL source program can be exchanged among different computers of one manufacturer or among computers of different manufacturers.

A COBOL program is both a readable document and an efficient computer program. Throughout the study of the COBOL language, it is important to keep these two basic capabilities of COBOL in mind and to observe the close relationship between them.

The readability factor of the COBOL language facilitates communication not only between programmer and management, but also among programmers, with a minimum of additional documentation. The readability factor need not affect the other equally important capability of constituting an efficient computer program. It is precisely here that the attention of a good COBOL programmer is centered. He can produce a solution in the form of a well-integrated COBOL program by combining the following: knowledge of the problem, programming technique, capability of the equipment, and familiarity with the available elements of the COBOL language.

The History of COBOL

Development of the COBOL programming language is a continuing process performed by the Programming Language Committee (PLC) of the Conference on Data Systems Languages (CODASYL). This committee is made up of representatives of computer manufacturers and computer users.

The first version of the COBOL programming language to be published by CODASYL was called COBOL-60. The second version, called COBOL-61, contained changes in the organization of the Procedure Division and thus was not completely compatible with COBOL-60.

In 1963 the third version, called COBOL-61 Extended, was released. It was basically COBOL-61 with the addition of the sort feature, the addition of the report writer feature, and the modification of the arithmetics to include multiple receiving fields and the CORRESPONDING option.

The fourth version of the COBOL programming language, COBOL-65, consists of COBOL-61 Extended with the inclusion of a series of options to provide for the reading, writing, and processing of mass storage files and the addition of table handling features.

Beginning in 1968 the CODASYL COBOL Programming Language Committee began to report its developmental work in a Journal of Development. The first report to be published was the CODASYL COBOL Journal of Development -- 1968. This journal is the official report of the CODASYL COBOL Programming Language Committee and it documents the developmental activities of CODASYL through July 1968. COBOL-68 is based on COBOL-65 with certain additions and deletions.

Additional COBOL Journal of Development reports were published in 1969, 1970 and 1973. Each documented the developmental activities of CODASYL from the previous report, resulting in continually varying COBOL definitions.

The Standardization of COBOL

In September 1962 the American National Standards Institute (ANSI) set up a committee to work on the definition of a standard COBOL programming language. This standardization effort was based on the technical content of COBOL as defined by CODASYL. In August 1968 an American National Standard COBOL was approved which was based upon the developmental work of CODASYL through January 1968. This first version was called American National Standard COBOL 1968.

In May 1974 a revision of American National Standard COBOL was approved. This revision, called American National Standard COBOL 1974, is based upon the developmental work of CODASYL through December 1971. The COBOL programming language and compiler described in this document is based on the American National Standard COBOL 1974.

CONVENTIONS USED IN THIS MANUAL

This manual presents the language definition and capabilities of COBOL in a generally accepted syntax consistent with the 1974 American National Standard COBOL document. As a result, COBOL Syntax is specified by formats employing special notation.

Words

All underlined uppercase words are key words and are required when the functions of which they are a part are used. Uppercase words which are not underlined are optional and may or may not be present in the source program. Uppercase words, whether underlined or not, must be spelled correctly.

Lowercase words are generic terms used to represent COBOL words, literals, PICTURE character-strings, comment-entries, or a complete syntactical entry that must be supplied by the user. When generic terms are repeated in a general format, a number or letter appendage to the term serves to identify that term for explanation or discussion.

Brackets and Braces

When a portion of a general format is enclosed in brackets, [], that portion may be included or omitted at the user's choice. Braces, {}, enclosing a portion of a general format means a selection of one of the options contained within the braces must be made. In both cases, a choice is indicated by vertically stacking the possibilities. When brackets or braces enclose a portion of a format, but only one possibility is shown, the function of the brackets or braces is to delimit that portion of the format to which a following ellipsis applies. If an option within braces contains only reserved words that are not key words, then the option is a default option (implicitly selected unless one of the other options is explicitly indicated).

Ellipsis

The ellipsis (...) represents the position at which repetition may occur at the user's option.

Punctuation

The punctuation characters comma and semicolon are shown in some formats. Where shown in the formats, they are optional and may be included or omitted by the user. In the source program these two punctuation characters are interchangeable and either may be used anywhere one of them is shown in the formats. Neither one may appear immediately preceding the first clause of an entry or paragraph.

If desired, a semicolon or comma may be used between statements in the Procedure Division.

Paragraphs within the Identification and Procedure Divisions, and the entries within the Environment and Data Divisions must be terminated by the separator period.

Special Characters

The characters '+', '- ', '>','<', '=' , when appearing in formats, although not underlined, are required when such formats are used.

System Dependent Information

Selected features in ANSI COBOL are intended for definition by the implementor, to accomodate the capabilities and restrictions of the host system. These system dependent items are summarized in the COBOL Users Guide.

II

THE STRUCTURE OF THE COBOL LANGUAGE

THE LANGUAGE STRUCTURE

The smallest element in the COBOL language is the character. A character is a digit, a letter of the alphabet, or a symbol. A COBOL word is one possible result obtained when one or more COBOL characters are joined in a sequence of contiguous characters. Just as English words are determined by rules of spelling, so COBOL words are formed by following a specific set of rules.

Using the COBOL rules of grammar, the COBOL words and COBOL punctuation characters are combined into statements, sentences, paragraphs, and sections. When writing normal English, a failure to follow the rules of grammar and sentence structure may cause misunderstanding; the same is true when writing COBOL. It must be emphasized that a thorough knowledge of the rules of COBOL structure is a prerequisite to writing a workable COBOL program.

Character Set

The COBOL character set consists of fifty-one characters:

Digits	0 through 9
Letters	A through Z
Punctuation	Blank (or space)
,	Comma
;	Semicolon
.	Period
"	Quote
(Left parenthesis
)	Right parenthesis
Special	
>	Greater than
<	Less than
+	Plus
-	Minus (or hyphen)
*	Asterisk
/	Slash (or Stroke)
=	Equal
\$	Currency

These characters determine the structure of a COBOL program. In some constructs, such as comments, other characters may be used but they have no grammatical meaning.

Characters are combined to form either a separator or a character-string.

Character Set

The COBOL character set is a proper subset of the ASCII character code set native to the computer. The complete character set may be used only within non numeric literals and comments. The chart below gives the hexadecimal and decimal codes for the complete character set.

Character	Hexadecimal		Decimal		Hexadecimal	Decimal	
	Value	Value	Character	Value		Character	Value
Space	20	32	@	40	64		
!	21	33	A	41	65		
"	22	34	B	42	66		
#	23	35	C	43	67		
\$	24	36	D	44	68		
%	25	37	E	45	69		
&	26	38	F	46	70		
'	27	39	G	47	71		
(28	40	H	48	72		
)	29	41	I	49	73		
*	2A	42	J	4A	74		
+	2B	43	K	4B	75		
,	2C	44	L	4C	76		
-	2D	45	M	4D	77		
.	2E	46	N	4E	78		
/	2F	47	O	4F	79		
0	30	48	P	50	80		
1	31	49	Q	51	81		
2	32	50	R	52	82		
3	33	51	S	53	83		
4	34	52	T	54	84		
5	35	53	U	55	85		
6	36	54	V	56	86		
7	37	55	W	57	87		
8	38	56	X	58	88		
9	39	57	Y	59	89		
:	3A	58	Z	5A	90		
;	3B	59	[5B	91		
<	3C	60	\	5C	92		
=	3D	61]	5D	93		
>	3E	62	^	5E	94		
?	3F	63	-	5F	95		

Separators

A separator is a string of one or more punctuation characters.

Punctuation characters belong to the following set:

'	Space
,	Comma
=	Equal sign
(Left parenthesis
:	Period
"	Quotation mark (double)
)	Right parenthesis
;	Semicolon

Separators are formed according to the following rules:

1. A space is a separator. Anywhere a space is used as a separator, more than one space may be used.
2. Comma, semicolon, and period are separators when immediately followed by a space. These separators may appear only when explicitly permitted.
3. Parentheses are separators which may appear only in balanced pairs of left and right parentheses delimiting subscripts, indices, arithmetic expressions or conditions.

Left parentheses must be preceded by a separator space or left parenthesis.

Right parenthesis must be followed by one of the separators: space, period, semicolon, comma or right parenthesis.

4. Quotes are separators which may appear only in balanced pairs delimiting the nonnumeric literals except when the literal is continued.

An opening quotation mark must be immediately preceded by a space or left parenthesis.

A closing quotation mark must be immediately followed by one of the separators: space, comma, semicolon, period or right parenthesis.

Separators

5. The separator space may optionally immediately precede all separators except:

As specified by reference format rules.

As the separator closing quotation mark. In this case, a preceding space is considered as part of the nonnumeric literal and not as a separator.

The separator space may optionally immediately follow any separator except the opening quotation mark. In this case, a following space is considered as part of the nonnumeric literal and not as a separator.

Any punctuation character which appears as part of the specification of a PICTURE character-string or numeric literal is not considered as a punctuation character, but rather as a symbol used in the specification of that PICTURE character-string or numeric literal. PICTURE character-strings are delimited only by the separators space, comma, semicolon, or period.

These rules do not apply to the characters within nonnumeric literals, picture strings, or comments.

Character-Strings

A character-string is a sequence of one or more characters that form a COBOL word, literal, picture string, or comment. A character-string is delimited by separators.

COBOL Words

A COBOL word is a character-string of not more than 30 characters which form either a user word or a reserved word. All words are one or the other.

User Words

User words are composed of the alphabetic characters, the numbers, and the hyphen character. A user word must not begin or end with a hyphen. With the exception of paragraph-name, section-name, level-number and segment-number, all user-defined words must contain at least one alphabetic character. There are twelve types of user words:

program-name	condition-name
file-name	index-name
record-name	alphabet-name
data-name	text-name
paragraph-name	level-number
section-name	segment-number

Program-Name

The program-name identifies the COBOL source and object program. The name must contain at least one alphabetic character. Only the first 6 characters are associated with the object program.

File-Name

File-names are the internal names for files accessed by the source program. They are not necessarily the same as the external names given to the files. File-names must contain at least one alphabetic character and must be unique.

Record-Name

Record-names are used to name data records within a file. They must contain at least one alphabetic character and, if not unique, must be made unique by qualification with the file name.

Data-Name

A group of contiguous characters or a word of binary data treated as a unit of data is called a data item, named by a data-name. A data-name must contain at least one alphabetic character. References to data items must be made unique by qualification or the appending of subscripts (or indices) or both. Complete unique references to data items are called identifiers.

User Words

Paragraph-Name

A paragraph-name is a procedure name that identifies the beginning of a set of COBOL procedural sentences. If not unique, a paragraph-name must be made unique by qualification with a section-name.

Section-Name

A section-name is a procedure name that identifies the beginning of a set of paragraphs. Section-names must be unique.

Condition-Name

A condition-name may be defined in the SPECIAL-NAMES paragraph within the Environment Division or in a level-number 88 description within the Data Division.

A SPECIAL-NAMES condition-name is assigned to ON STATUS or OFF STATUS of one of eight system software switches.

A level-number 88 condition-name is assigned to a specific value, set of values, or range of values within a complete set of values that a data item may assume. The data item itself is called a conditional variable.

A condition-name is used only in conditions as an abbreviation for the relation condition which assumes that the associated switch or conditional variable is equal to one of the set of values to which that condition-name is assigned.

Index-Name

An index-name names an index associated with a specific table. It must contain at least one alphabetic character and must be unique.

Alphabet-Name

An alphabet-name is used to specify a character code set. It must contain at least one alphabetic character and must be unique.

Text-Name

A text-name is the name of a COBOL library text file. It must correspond exactly to a valid file access-name as described in the operating system documentation.

Level-Number

A level-number is used to specify the position of a data item within a data hierarchy. A level-number is a one- or two-digit number in the range 01-49, 66, 77 or 88.

Level-numbers 66, 77 and 88 identify special properties of a data description entry.

Segment-Number

A segment-number specifies the segmentation classification of a section. It is a one- to two-digit number in the range 01-99.

Reserved Words

Reserved Words

The structure of COBOL governs the use of certain COBOL words called reserved words. Reserved words, recognized by the COBOL compiler, aid the compiler in determining how to generate a program. A programmer cannot devise a reserved word for a COBOL program; he must use the word designated by the format of the language. A reserved word must not appear as a user-defined word within a program. A list of all reserved words recognized by the compiler is shown in Appendix B.

Five kinds of reserved words are recognized by the compiler:

- Key words
- Optional words
- Connectives
- Figurative constants
- Special-characters

Key Words

Key words are required elements of COBOL formats. Their presence indicates specific compiler action.

Optional Words

Optional words are optional elements of COBOL formats. Their presence has no effect on the object program.

Connectives

The connectives OF and IN are used interchangeably to connect qualifiers to a user word. The words AND and OR are logical connectives and are used in the formation of conditions.

Figurative Constants

Figurative constants identify commonly used constant values. These constant values are generated by the compiler according to the context in which the references occur. Note that figuratives represent values, not literal occurrences. Thus QUOTE cannot be used to delimit a nonnumeric literal, SPACE is not a separator, and so forth. Singular and plural forms of figuratives are equivalent and may be used interchangeably.

ZERO
ZEROS
ZEROES

Represents the value 0 or one or more zero (0) characters, depending on context.

SPACE
SPACES

Represents one or more space () characters.

HIGH-VALUE
HIGH-VALUES

Represents one or more of the highest characters in the collating sequence (hexadecimal FF).

LOW-VALUE
LOW-VALUES

Represents one or more of the lowest characters in the collating sequence (hexadecimal 00).

QUOTE
QUOTES

Represents one or more quote ("") characters.

Reserved Words

ALL literal

Represents one or more of the characters comprising the literal. The literal must be either a nonnumeric literal or a figurative constant. When a figurative constant is used, the word ALL is redundant.

When a figurative constant represents a string of one or more characters, the length of the string is determined by the compiler from context according to the following rules:

1. When a figurative constant is associated with another data item, as when the figurative constant is moved to or compared with another data item, the string of characters specified by the figurative constant is repeated character-by-character on the right until the size of the resultant string is equal to the size in characters of the associated data item. This is done prior to and independent of the application of any JUSTIFIED clause that may be associated with the data item.
2. When a figurative constant is not associated with another data item, as when the figurative constant appears in a DISPLAY or STOP statement, the length of the string is one character.

A figurative constant may be used wherever a literal appears in a format, except that whenever the literal is restricted to having only numeric characters in it, the only figurative constant permitted is ZERO (ZEROS, ZEROES).

Each reserved word which is used to reference a figurative constant value is a distinct character-string with the exception of the construction 'ALL literal' which is composed of two distinct character-strings.

Special Characters

The special character words are the arithmetic operators and relation characters:

- + Plus sign (indexing)
- Minus sign (indexing)
- > Greater than
- < Less than
- = Equal to

Literals

A literal is a character-string whose form determines its value. Literals are either nonnumeric or numeric.

Nonnumeric Literals

A nonnumeric literal is a character-string enclosed in quotes. Any characters in the COBOL character set may be used. Quote characters within the string are represented by two contiguous quotes. The value of the literal is the string itself excluding the delimiting quotes and one of each contiguous pair of embedded quotes. The value of the literal may contain from 1 to 2047 characters.

Examples:

<u>Literal</u>	<u>Value</u>
"AGE?"	AGE?
""TWENTY""?"	"TWENTY"?
""""	illegal (odd number of quotes)

Numeric Literals

A numeric literal represents a numeric value, not a character-string. Numeric literals are composed according to the following rules:

1. The literal must contain from 1 to 18 digits.
2. The literal may contain a single plus or minus sign if it is the first character.
3. The literal may contain a single decimal point if it is not the last character. The decimal point must be represented with a comma if the DECIMAL-POINT IS COMMA phrase is specified in the SPECIAL-NAMES paragraph.

Examples:

```

1234
+1234
-1.234
.1234
+.1234

```

Picture String

Picture String

A picture string consists of certain combinations of characters from the COBOL character set used as symbols. Any punctuation character appearing as part of a picture string is considered to be a symbol, not a punctuation character.

Comment-Entry

A comment-entry is an entry in the Identification Division that may contain any characters from the computer's character set.

System Names

System names identify certain hardware or software system components. System names consist of device-names and switch-names.

<u>Device-Names</u>	<u>Component</u>
PRINT	printer or print file
INPUT	input only device
OUTPUT	output only device
INPUT-OUTPUT	input-output device
RANDOM	disc

<u>Switch-Names</u>	<u>Component</u>
SWITCH-1	
.	
.	
SWITCH-8	software switches

THE PROGRAM STRUCTURESource Format

COBOL programs are accepted as a sequence of formatted lines (or records) of 80 characters or less. Each line is divided into five areas:

<u>Columns</u>	<u>Area</u>
1-6	sequence number
7	indicator
8-11	A
12-72	B
73-80	identification

The sequence number and identification areas are used for clerical and documentation purposes. They are ignored by the compiler.

The indicator area is used for denoting line continuation, comments, and debugging.

Areas A and B contain the actual program according to the following rules:

1. Division headers, section headers, paragraph headers, section-names, and paragraph-names must begin in area A.
2. The Data Division level indicator FD and level-numbers 01 and 77 must begin in area A. Other level-numbers may begin in area A or area B, although B is preferable.
3. The key word DECLARATIVES and the key words END DECLARATIVES, precede and follow, respectively, the declaratives portion of the Procedure Division. Each must appear on a line by itself and each must begin in area A and be followed by a period and a space.
4. Any other language element must begin in area B unless it immediately follows, on the same line, an element in area A.

Source Format

Continuation of Lines

Whenever a sentence, entry, phrase, or clause requires more than one line, it may be continued by starting subsequent line(s) in area B. These subsequent lines are called the continuation line(s). The line being continued is called the continued line. Any word or literal may be broken in such a way that part of it appears on a continuation line, according to the following rules:

1. A hyphen in the indicator area of a line indicates that the first nonblank character in area B of the current line is the successor of the last nonblank character of the preceding line without any intervening space. However, if the continued line contains a nonnumeric literal without closing quotation mark, the first nonblank character in area B on the continuation line must be a quotation mark, and the continuation starts with the character immediately after that quotation mark. All spaces at the end of the continued line are considered part of the literal. Area A of continuation line must be blank.
2. If there is no hyphen in the indicator area of a line, it is assumed that the last character in the preceding line is followed by a space.

Blank Lines

A blank line is one that is blank in the indicator, A and B areas. A blank line can appear anywhere in the source program, except immediately preceding a continuation line with a hyphen in the indicator area.

Comment Lines

A comment line is any line with an asterisk (*) in the indicator area of the line. A comment line can appear as any line in a source program after the Identification Division header. Any combination of characters from the computer's character set may be included in area A and area B of that line. The asterisk and the characters in area A and area B will be produced on the listing but serve as documentation only.

Successive comment lines are allowed. Continuation of comment lines is permitted, except that each continuation line must contain an asterisk in the indicator area.

A special form of comment line represented by a slash (/) in the indicator area of the line causes page ejection prior to printing the comment.

Debugging Lines

A debugging line is any line with a D in the indicator area of the line. Any debugging line that consists solely of spaces from area A to the identifier area is considered to be a blank line.

A program that contains debugging lines must be syntactically correct with or without the debugging lines.

A debugging line will be considered to have all the characteristics of a comment line if the debug option is not specified at compiler invocation.

Successive debugging lines are allowed. Continuation of debugging lines is permitted, except that each continuation line must contain a D in the indicator area, and character strings may not be broken across two lines.

Statements

COBOL statements always begin with a key word called a verb. There are three kinds of statements: directive, conditional, and imperative.

A directive statement specifies action to be taken by the compiler during compilation. The directive statements are:

The COPY and USE statements.

A conditional statement specifies that the truth value of a condition is to be determined and that the subsequent action of the object program is dependent on this truth value. The conditional statements are:

An IF statement.

A READ statement with the AT END or INVALID KEY phrase.

A DELETE, REWRITE or START statement with the INVALID KEY phrase.

A WRITE statement with the INVALID KEY phrase.

An arithmetic statement (ADD, COMPUTE, DIVIDE, MULTIPLY, SUBTRACT) with the SIZE ERROR phrase.

Statements

An imperative statement specifies an unconditional action to be taken by the object program. The imperative statements are:

A READ statement without the AT END or INVALID KEY phrase.

A DELETE, REWRITE or START statement without the INVALID KEY phrase.

A WRITE statement without the INVALID KEY phrase.

An arithmetic statement (ADD, COMPUTE, DIVIDE, MULTIPLY, SUBTRACT) without the ON SIZE ERROR phrase.

An ACCEPT, ALTER, CLOSE, DISPLAY, EXIT, GO, INSPECT, MOVE, OPEN, PERFORM, SET or STOP statement.

Whenever the term imperative-statement appears in the format of a COBOL verb, it refers to one or more consecutive imperative statements. The sequence ends with a period separator or an ELSE associated with an IF verb.

Sentences

A sentence is a sequence of one or more statements terminated by the period separator. There are three kinds of sentences: directive, conditional, and imperative.

A directive sentence may contain only a single directive statement.

A conditional sentence is a conditional statement, optionally preceded by a sequence of imperative statements, terminated by a period followed by a space.

An imperative sentence is one or more imperative statements terminated by a period separator.

Clauses and Entries

An entry is an item of descriptive or declaratory nature composed of consecutive clauses. Each clause specifies an attribute of the entry. Clauses are separated by space, comma, or semicolon separators. The entry is terminated by a period separator.

Paragraphs

A paragraph is a sequence of an arbitrary number, which may be zero, of sentences or entries. In the Identification and Environment Divisions, each paragraph begins with a reserved word called a paragraph header. In the Procedure Division, each paragraph begins with a user-defined paragraph-name.

Sections

A section is a sequence of an arbitrary number, which may be zero, of paragraphs in the Environment and Procedure Divisions and a sequence of an arbitrary number, which may be zero, of entries in the Data Division. In the Environment and Data Divisions, each section begins with reserved words called a section header. In the Procedure Division, each section begins with a user-defined section-name.

Divisions

Each COBOL program consists of four divisions; each is composed of paragraphs or sections. These are the Identification, Environment, Data, and Procedure divisions, in that order. All divisions are required. Each division begins with a group of reserved words called a division header.

COPY Statement

THE COPY STATEMENT

The COPY statement provides the facility for copying text from user-specified files into the source program. Text is copied from the file without change. The effect of the interpretation of the COPY statement is to insert text into the source program, where it will be treated by the compiler as part of the source program.

COBOL library text is placed on the COBOL library as a function independent of the COBOL program and according to operating system techniques.

FORMAT

COPY text-name.

The COPY statement must be preceded by a space and terminated by the separator period. There must not be any additional text in area B following the separator period.

Text-name is the external identification of the file containing the text to be copied. Its format conforms to the rules for filename (or pathname) construction of the host operating system. If the external identification contains any characters that are not letters or digits, or if the first character is not a letter, then the text-name must be written as a nonnumeric literal and enclosed in quotation marks.

A COPY statement may occur in the source program anywhere a characterstring or separator may occur except that a COPY statement must not occur within a COPY statement.

The compilation of a source program containing COPY statements is logically equivalent to processing all COPY statements prior to the processing of the resulting source program.

The effect of processing a COPY statement is that the library text associated with text-name is copied into the source program, logically replacing the entire COPY statement, beginning with the reserved word COPY and ending with the punctuation character period, inclusive.

The library text is copied unchanged.

Debugging lines are permitted within library text. If a COPY statement is specified on a debugging line, then the COPY statement will be processed only if the debug option has been specified in the compiler invocation options.

The text produced as a result of processing a COPY statement may not contain a COPY statement.

The syntactic correctness of the library text cannot be independently determined. The syntactic correctness of the entire COBOL source cannot be determined until all COPY statements have been completely processed.

Library text must conform to the rules for COBOL source format.

COPY Examples:

FILE-CONTROL.
COPY FLCTRL.

PROCEDURE DIVISION.
COPY "INPUTP.COBOL".

III

IDENTIFICATION DIVISION

INTRODUCTION

The Identification Division must be included in every COBOL source program. This division identifies both the source program and the resultant object program. In addition, the user may include other commentary information.

FORMAT

IDENTIFICATION DIVISION.

PROGRAM-ID. program-name.

[**AUTHOR.** [comment-entry] ...]

[**INSTALLATION.** [comment-entry] ...]

[**DATE-WRITTEN.** [comment-entry] ...]

[**SECURITY.** [comment-entry] ...]

PROGRAM IDENTIFICATION

The Identification Division must begin with the reserved words IDENTIFICATION DIVISION followed by a period and a space.

Paragraph headers identify the type of information contained in the paragraph. The name of the program must be given in the first paragraph, which is the PROGRAM-ID paragraph. The other paragraphs are optional and may be included at the user's choice, in the order of presentation shown.

PROGRAM-ID Paragraph

The PROGRAM-ID Paragraph

The PROGRAM-ID paragraph, containing the program-name, identifies the source program, the object program and all listings pertaining to a particular program. A program-name is a user-defined word made up of only those characters from the word set.

A program-name cannot exceed 8 characters in length, and must contain at least one alphabetic character located in any position within the program-name. Each program-name must be unique.

The AUTHOR, INSTALLATION, DATE-WRITTEN, SECURITY Paragraphs

The AUTHOR, INSTALLATION, DATE-WRITTEN, and SECURITY paragraphs are optional. The programmer may use these paragraphs to document information pertaining to the paragraph header.

The comment-entry may be any combination of characters from the computer's character set. The continuation of the comment-entry by the use of the hyphen in the indicator area is not permitted; however, the comment-entry may be contained on one or more lines.

IV

ENVIRONMENT DIVISION

INTRODUCTION

The Environment Division describes the hardware configuration of the compiling computer (source computer) and the computer on which the object program is run (object computer). It also describes the relationship between the files and the input/output media.

The Environment Division must be included in every COBOL source program.

There are two sections in the Environment Division: the Configuration Section and the Input-Output Section.

FORMAT

ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

SOURCE-COMPUTER. computer-name.

OBJECT-COMPUTER. computer-name.

[SPECIAL-NAMES. special-names-entry].

[INPUT-OUTPUT SECTION.

FILE-CONTROL. {file-control-entry} ...

[I-O-CONTROL. input-output-control-entry]].

CONFIGURATION SECTION

The Configuration Section deals with the characteristics of the source computer and the object computer. This section is divided into three paragraphs:

the SOURCE-COMPUTER paragraph, which describes the computer configuration on which the source program is compiled

the OBJECT-COMPUTER paragraph, which describes the computer configuration on which the object program produced by the compiler is to be run

the SPECIAL-NAMES paragraph, which relates names used by the compiler to user-names in the source program.

SOURCE-COMPUTER Paragraph

The SOURCE-COMPUTER Paragraph

The SOURCE-COMPUTER paragraph identifies the computer upon which the program is to be compiled.

FORMAT

SOURCE-COMPUTER. computer-name.

Computer-name is a user-defined word and is only commentary.

The OBJECT-COMPUTER Paragraph

The OBJECT-COMPUTER paragraph identifies the computer on which the program is to be executed.

FORMAT

OBJECT-COMPUTER. computer-name

[,MEMORY SIZE integer {WORDS }]
 {CHARACTERS}
 {MODULES }]

[,PROGRAM COLLATING SEQUENCE IS alphabet-name].

Computer-name is a user-defined word and is only commentary.

The MEMORY SIZE definition is treated as commentary.

The PROGRAM COLLATING SEQUENCE clause specifies the program collating sequence to be used in determining the truth value of any nonnumeric comparisons. The Program Collating Sequence clause is treated as commentary; the collating sequence is always ASCII.

The SPECIAL-NAMES Paragraph

The SPECIAL-NAMES paragraph relates names used by the compiler to user-names in the source program.

FORMAT

```
[SPECIAL-NAMES. [,switch-name
  {ON STATUS IS cond-name-1 [,OFF STATUS IS cond-name-2]}]...
  {OFF STATUS IS cond-name-2 [,ON STATUS IS cond-name-1 ]}

  [,alphabet-name IS {STANDARD-1}]...
    {NATIVE      }

  [,CURRENCY SIGN IS literal-1]

  [,DECIMAL-POINT IS COMMA] .]
```

Switch-name may be SWITCH-1, ..., SWITCH-8.

At least one condition-name must be associated with each switch-name given. The status of the switch is specified by condition-names and interrogated by testing the condition-names.

The alphabet-name clause provides a means for relating a name to a specified character code set and/or collating sequence. The alphabet-name definition is treated as commentary; the collating sequence is always ASCII.

SPECIAL-NAMES Paragraph

The literal which appears in the CURRENCY SIGN IS literal clause is used in the PICTURE clause to represent the currency symbol. The literal is limited to a single character and must not be one of the following characters:

digits 0 through 9;

alphabetic characters A, B, C, D, L, P, R, S, V, X, Z, or the space;

special characters '*', '+', '-', ',', '.', ';', '(', ')', ''', '/', '='.

If this clause is not present, only the currency sign (\$) is used in the PICTURE clause.

The clause DECIMAL-POINT IS COMMA means that the function of comma and period are exchanged in the character-string of the PICTURE clause and in numeric literals.

INPUT-OUTPUT SECTION

The INPUT-OUTPUT section names the files and external media required by an object program and provides information required for transmission and handling of data during execution of the object program. This section is divided into two paragraphs:

the FILE-CONTROL paragraph which names and associates the files with external media.

the I-O-CONTROL paragraph which defines special control techniques to be used in the object program.

FORMAT

[INPUT-OUTPUT SECTION.

FILE-CONTROL.

{file-control-entry} ...

[I-O-CONTROL.

I-O-control-entry]]

The FILE-CONTROL Paragraph

The FILE-CONTROL paragraph names each file and allows specification of other file-related information.

FORMAT

FILE-CONTROL. {file-control-entry} ...

The content of the file-control-entry is dependent upon the organization of the file named.

Sequential File Control Entry

The Sequential File Control Entry

FORMAT

SELECT file-name

```
  ASSIGN TO device-type, {"external-file-name"}  
                                {data-name-1} }  
  
  [;ORGANIZATION IS SEQUENTIAL]  
  
  [;ACCESS MODE IS SEQUENTIAL]  
  
  [;FILE STATUS IS data-name-2].
```

The SELECT clause must be specified first in the file control entry. The clauses which follow the SELECT clause may appear in any order.

Each file described in the Data Division must be named once and only once as file-name in the FILE-CONTROL paragraph. Each file specified in the file control entry must have a file description entry in the Data Division.

The ASSIGN clause specifies the association of the file referenced by file-name to a storage medium.

Device-type must be one of the device names INPUT, INPUT-OUTPUT, OUTPUT, PRINT, or RANDOM according to the operations to be performed.

External-file-name specifies the file access name. It can be from one to thirty characters in length and must be enclosed in quotation marks. A name longer than thirty characters will be diagnosed as an error. The name may contain any sequence of characters supported by the operating system for file access names.

Data-name-1 must be defined in the Data Division as a data item of category alphanumeric and must not be defined in the Linkage Section. Its value at the time of an OPEN statement execution will be used as the file access name. Data-name-1 may be qualified.

The ORGANIZATION clause specifies the logical structure of a file. The file organization is established at the time a file is created and cannot subsequently be changed.

Records in the file are accessed in the sequence dictated by the file organization. This sequence is specified by predecessor-successor record relationships established by the execution of WRITE statements when the file is created or extended.

When the ORGANIZATION clause is not specified, ORGANIZATION IS SEQUENTIAL is implied.

The ACCESS MODE clause specifies the order in which records are read or written.

If the ACCESS MODE clause is not specified, ACCESS MODE IS SEQUENTIAL is implied.

When the FILE STATUS clause is specified, a value will be moved by the operating system into the data item specified by data-name-2 after the execution of every statement that references that file either explicitly or implicitly. This value indicates the status of execution of the statement.

Data-name-2 must be defined in the Data Division as a two-character data item of the category alphanumeric and must not be defined in the File Section. Data-name-2 may be qualified.

Relative File Control Entry

The Relative File Control Entry

FORMAT

SELECT file-name

ASSIGN TO RANDOM, {"external-file-name"}
 {data-name-1 }

;ORGANIZATION IS RELATIVE

[;**ACCESS MODE IS** { **SEQUENTIAL** [,**RELATIVE KEY IS** data-name-2]}]
 {{**RANDOM**} ,**RELATIVE KEY IS** data-name-2 }
 {{**DYNAMIC**} }

[;**FILE STATUS IS** data-name-3].

The SELECT clause must be specified first in the file control entry. The clauses which follow the SELECT clause may appear in any order.

Each file described in the Data Division must be named once and only once as file-name in the FILE-CONTROL paragraph. Each file specified in the file control entry must have a file description entry in the Data Division.

The ASSIGN TO RANDOM clause specifies the association of the file referenced by file-name to a storage medium.

External-file-name specifies the file access name and must be enclosed in quotation marks. It can be from one to thirty characters in length. A name longer than thirty characters will be diagnosed as an error. The name may contain any characters supported by the operating system for file access names.

Data-name-1 must be defined in the Data Division as a data item of category alphanumeric and must not be defined in the Linkage Section. Its value at the time of an OPEN statement execution will be used as the file access name. Data-name-1 may be qualified.

The ORGANIZATION IS RELATIVE clause specifies the logical structure of a file. The file organization is established at the time a file is created and cannot subsequently be changed.

All records stored in a relative file are uniquely identified by relative record numbers. The relative record number of a given record specifies the record's logical ordinal position in the file. The first logical record has a relative record number of one (1), and subsequent logical records have relative record numbers of 2, 3, 4, ...n.

The ACCESS MODE clause specifies the order in which records are to be accessed.

When the ACCESS MODE IS SEQUENTIAL, records in the file are accessed in the sequence dictated by the file organization. This sequence is the order of ascending relative record numbers of existing records in the file.

If the ACCESS MODE IS RANDOM, the value of the RELATIVE KEY data item indicates the record to be accessed.

If a relative file is to be referenced by a START statement, the RELATIVE KEY phrase must be specified for that file.

When the ACCESS MODE IS DYNAMIC, records in the file may be accessed sequentially and/or randomly.

Data-name-2 must not be defined in a record description entry associated with that file-name. The data item referenced by data-name-2 must be defined as an unsigned integer. Data-name-2 may be qualified.

If the ACCESS MODE clause is not specified, ACCESS MODE IS SEQUENTIAL is implied.

When the FILE STATUS clause is specified, a value will be moved by the operating system into the data item specified by data-name-3 after the execution of every statement that references that file either explicitly or implicitly. This value indicates that status of execution of the statement.

Data-name-3 must be defined in the Data Division as a two-character data item of the category alphanumeric and must not be defined in the File Section.

Indexed File Control Entry

The Indexed File Control Entry

FORMAT

```
SELECT file-name  
ASSIGN TO RANDOM, {"external-file-name"}  
          {data-name-1} }  
  
[;ORGANIZATION IS INDEXED  
[;ACCESS MODE IS {SEQUENTIAL}]  
          {RANDOM }  
          {DYNAMIC }  
  
;RECORD KEY IS data-name-2  
[; ALTERNATE RECORD KEY IS data-name-3 [WITH DUPLICATES]]...  
[;FILE STATUS IS data-name-4].
```

The SELECT clause must be specified first in the file control entry. The clauses which follow the SELECT clause may appear in any order.

Each file described in the Data Division must be named once and only once as file-name in the FILE-CONTROL paragraph.

Each file specified in the file control entry must have a file description entry in the Data Division.

The ASSIGN TO RANDOM clause specifies the association of the file referenced by file-name to a storage medium.

External-file-name specifies the file access name and must be enclosed in quotation marks. It can be from one to thirty characters in length. A name longer than thirty characters will be diagnosed as an error. The name may contain any characters supported by the operating system for file access names.

Data-name-1 must be defined in the Data Division as a data item of category alphanumeric and must not be defined in the Linkage Section. Its value at the time of an OPEN statement execution will be used as the file access name. Data-name-1 may be qualified.

The ORGANIZATION IS INDEXED clause specifies the logical structure of a file. The file organization is established at the time a file is created and cannot subsequently be changed.

The ACCESS MODE clause specifies the order in which records are to be accessed.

When the ACCESS MODE IS SEQUENTIAL, records in the file are accessed in the sequence dictated by the file organization. For indexed files this sequence is the order of ascending record key values within a given key of reference.

If the ACCESS MODE IS RANDOM, the value of the RECORD KEY data item indicates the record to be accessed.

When the ACCESS MODE IS DYNAMIC, records in the file may be accessed sequentially and/or randomly.

If the ACCESS MODE clause is not specified, ACCESS MODE IS SEQUENTIAL is implied.

The RECORD KEY clause specifies the record key that is the prime record key for the file. This prime record key provides an access path to records in an indexed file.

An ALTERNATE RECORD KEY clause specifies a record key that is an alternate record key for the file. This alternate record key provides an alternate access path to records in an indexed file.

The data description of data-name-2 and data-name-3 as well as their relative locations within a record must be the same as that used when the file was created. The number of alternate keys for the file must also be the same as that used when the file was created.

The data items referenced by data-name-2 and data-name-3 must each be defined as a data item of the category alphanumeric within a record description entry associated with that file-name.

Neither data-name-2 nor data-name-3 can describe an item whose size is variable.

Data-name-3 cannot reference an item whose leftmost character position corresponds to the leftmost character position of an item referenced by data-name-2 or by any other data-name-3 associated with this file.

Data-name-2 and data-name-3 may be qualified.

Indexed File Control Entry

The WITH DUPLICATES phrase specifies that the value of the associated alternate record key may be duplicated within any of the records in the file. If the WITH DUPLICATES phrase is not specified, the value of the associated alternate record key must not be duplicated among any of the records in the file.

When the FILE STATUS clause is specified, a value will be moved by the operating system into the data item specified by data-name-4 after the execution of every statement that references that file either explicitly or implicitly. This value indicates the status of execution of the statement.

Data-name-4 must be defined in the Data Division as a two-character data item of the category alphanumeric and must not be defined in the File Section.

The I-O CONTROL Paragraph

The I-O CONTROL paragraph specifies the memory area which is to be shared by different files.

FORMAT

I-O-CONTROL.

```
[; SAME AREA FOR file-name-1 [, file-name-2] ...] ...
```

The I-O-CONTROL paragraph is optional.

The SAME AREA clause specifies that two or more files are to use the same memory area during processing. The area being shared includes all storage area assigned to the files specified; therefore, it is not valid to have more than one of the files open at the same time.

More than one SAME clause may be included in a program; however, a file-name must not appear in more than one SAME AREA clause.

The files referenced in the SAME AREA clause need not all have the same organization or access.

V

DATA DIVISION

INTRODUCTION

The Data Division describes the data that the object program is to accept as input, to manipulate, to create, or to produce as output. Data to be processed falls into three categories:

That which is contained in files and enters or leaves the internal memory of the computer from a specified area or areas.

That which is developed internally and placed into intermediate or working storage, or placed into specific format for output reporting purposes.

Constants which are defined by the user.

The Data Division, which is one of the required divisions in a program, is subdivided into three sections:

The FILE SECTION defines the structure of data files. Each file is defined by a file description entry and one or more record descriptions. Record descriptions are written immediately following the file description entry.

The WORKING-STORAGE SECTION describes records and noncontiguous data items which are not part of external data files but are developed and processed internally. It also describes data items whose values are assigned in the source program and do not change during the execution of the object program.

The LINKAGE SECTION in a program is meaningful if and only if the object program is to function under the control of a CALL statement, and the CALL statement in the calling program contains a USING phrase.

The Linkage Section is used for describing data that is available through the calling program but is to be referred to in both the calling and the called program. No space is allocated in the program for data items referenced by data-names in the Linkage Section of that program. Procedure Division references to these data items are resolved at object time by equating the reference in the called program to the location used in the calling program. In the case of index-names, no such correspondence is established. Index-names in the called and calling program always refer to separate indices.

Introduction

Data items defined in the Linkage Section of the called program may be referenced within the Procedure Division of the called program only if they are specified as operands of the USING phrase of the Procedure Division header or are subordinate to such operands, and the object program is under the control of a CALL statement that specifies a USING phrase.

FORMAT

DATA DIVISION.

[FILE SECTION.

```
[file-description-entry  
[record-description-entry] ...] ...]
```

[WORKING-STORAGE SECTION.

```
[77-level-description-entry] ...]  
[record-description-entry ]
```

[LINKAGE SECTION.

```
[77-level-description-entry] ...]  
[record-description-entry ]
```

FILE SECTION

The File Section header is followed by a file description entry consisting of a level indicator (FD), a file-name and a series of independent clauses. The FD clauses specify the size of the logical and physical records, the presence or absence of label records, the value of label items, and the names of the data records which comprise the file. The entry itself is terminated by a period.

In a COBOL program the file description entry (FD) represents the highest level or organization in the File Section.

FORMAT**FILE SECTION.**

```
[file-description-entry  
[record-description-entry] ...] ...
```

File Description Entry

The File Description Entry

The File Description furnishes information concerning the physical structure, identification, and record name pertaining to a given file.

FORMAT

FD file-name

```
[ ;BLOCK CONTAINS [integer-1 TO] integer-2 {RECORDS } ]  
{CHARACTERS}  
  
[ ;RECORD CONTAINS [integer-3 TO] integer-4 CHARACTERS]  
  
;LABEL {RECORD IS } {STANDARD}  
{RECORDS ARE} {OMITTED}  
  
[ ;VALUE OF LABEL IS [literal-1]]  
  
[ ;DATA {RECORD IS } data-name-1 [,data-name-2]...].  
{RECORDS ARE}
```

The level indicator FD identifies the beginning of a file description and must precede the file-name.

The clauses which follow the name of the file are optional in many cases, and their order of appearance is not significant.

One or more record description entries must follow the file description entry.

A file description entry must end with a period separator.

The BLOCK CONTAINS Clause

The BLOCK CONTAINS clause specifies the size of a physical record.

FORMAT

```
BLOCK CONTAINS [integer-1 TO] integer-2 {RECORDS }
{CHARACTERS}
```

This clause is required except when:

A physical record contains only one complete logical record.

The device assigned to the file has only one physical record size.

The device assigned to the file has a standard record size, although the device may have more than one physical record size. In this case, the absence of this clause denotes the standard physical record size.

The size of the physical record may be stated in terms of RECORDS, unless one of the following situations exist, in which case the RECORDS phrase must not be used:

In mass storage files where logical records may extend across physical records.

The physical record contains padding.

Logical records are grouped in such a manner that an inaccurate physical record size would be implied.

When the word CHARACTERS is specified, the physical record size is specified in terms of the number of character positions required to store the physical record, regardless of the types of characters used to represent the items within the physical record.

If only integer-2 is shown, it represents the exact size of the physical record. If integer-1 and integer-2 are shown, they refer to the minimum and maximum size of the physical record, respectively.

RECORD CONTAINS Clause

The RECORD CONTAINS Clause

The RECORD CONTAINS clause specifies the size of the data records.

FORMAT

RECORD CONTAINS [integer-1 TO] integer-2 CHARACTERS

The size of each data record is completely defined with the record description entry, therefore this clause is never required. When present, however, the following notes apply:

Integer-2 may not be used by itself unless all the data records in the file have the same size. In this case integer-2 represents the exact number of characters in the data record.

If integer-1 and integer-2 are both shown, they refer to the minimum number of characters in the smallest size data record and the maximum number of characters in the largest size data record, respectively.

The size is specified in terms of the number of character positions required to store the logical record, regardless of the types of characters used to represent the items within the logical record. The size of a record is determined by the sum of the number of characters in all fixed length elementary items plus any filler characters generated between elementary items because of the SYNCHRONIZED clause.

The LABEL RECORD Clause

The LABEL RECORD clause specifies whether labels are present.

FORMAT

```
LABEL {RECORD IS } {STANDARD}  
{RECORDS ARE} { OMITTED }
```

This clause is required in every file description entry.

STANDARD specifies that labels exist for the file or the device to which the file is assigned and the labels conform to the operating system specification. STANDARD must be specified for files assigned to a RANDOM device.

OMITTED specifies that no explicit labels exist for the file or the device to which the file is assigned.

The VALUE OF Clause

The VALUE OF clause particularizes the description of an item in the label records associated with a file.

FORMAT

```
VALUE OF LABEL IS literal-1
```

This clause is treated as commentary.

This clause must not be specified if OMITTED is specified in the LABEL RECORDS clause.

DATA RECORDS Clause

The DATA RECORDS Clause

The DATA RECORDS clause serves only as documentation for the names of data records with their associated file.

FORMAT

```
DATA {RECORD IS }    data-name-1 [,data-name-2]...
{RECORDS ARE}
```

Data-name-1 and data-name-2 are the names of data records and must have 01 level-number record descriptions, with the same name, associated with them.

The presence of more than one data-name indicates that the file contains more than one type of data record. These records may be of differing sizes, different formats, etc. The order in which they are listed is not significant.

Conceptually, all data records within a file share the same area. This is in no way altered by the presence of more than one type of data record within the file.

WORKING-STORAGE SECTION

The Working-Storage Section is composed of the section header, followed by data description entries for 77 level description entries and/or record description entries.

The data-name of a 01-level data description entry in the Working-Storage Section must be unique since it cannot be qualified. Subordinate data-names need not be unique if they can be made unique by qualification.

FORMAT**WORKING-STORAGE SECTION.**

```
[77-level-description-entry] ...
[record-description-entry ]
```

LINKAGE SECTION

The structure of the Linkage Section is the same as for the Working-Storage Section, beginning with a section header, followed by data description entries for noncontiguous data items and/or record description entries.

Each Linkage Section record-name and noncontiguous item name must be unique within the called program since it cannot be qualified.

FORMAT**LINKAGE SECTION.**

```
[77-level-description-entry] ...
[record-description-entry ]
```

Record Description Entry

RECORD DESCRIPTION ENTRY

A record description entry consists of a set of data description entries which describe the characteristics of a particular record. Each data description entry consists of a level-number followed by a data-name and a series of independent clauses, as required.

FORMAT

```
{data-description-entry} ...
```

Level-Numbers

The first data description of a record must have a level-number of 01 or 1, and must start in area A of a source line.

Each data description entry can be subdivided into multiple data description entries, each having the same level-number; which must be greater than the level-number of the subdivided entry, but less than 50. Level-numbers do not necessarily have to be successive. Thus, a record is a hierarchy of data description entries.

Elementary Items

Any data description entry which is not further subdivided is called an elementary item. A record itself may be an elementary item, consisting of a single level 01 data description entry. A subdivided data description entry with its subdivisions is called a group and is non-elementary. Therefore, a group includes all group and elementary items following it until a level-number less than or equal to the level-number of that group is encountered.

Note that certain clauses of the data description entry may occur only in elementary items. They may not occur in 01-level entry as they may affect the subdivisions of that entry. An elementary item must have either a PICTURE clause or INDEX usage; it may not have both.

77 LEVEL DESCRIPTION ENTRY

In the Working-Storage and Linkage Sections, a special level-number of 77 can be used in data description entries which are not subdivisions of other items, and are not themselves subdivided. These data description entries specify noncontiguous data items. Such a data description entry is elementary.

A 77 level description entry must contain a data name and either the PICTURE clause or the USAGE IS INDEX clause, but cannot contain an OCCURS clause. Other clauses are optional and can be used to complete the description of the item if necessary.

FORMAT**data-description-entry**

Data Description Entry

THE DATA DESCRIPTION ENTRY

A data description entry specifies the characteristics of a particular item of data.

FORMAT 1

```
level-number {data-name-1}
    {FILLER      }

[ ;REDEFINES data-name-2]

[ ;{PICTURE} IS character-string]
    {PIC      }

[ ;{USAGE IS}    {COMPUTATIONAL   }
    {COMP        }
    {COMPUTATIONAL-1}
    {COMP-1       }
    {COMPUTATIONAL-3}
    {COMP-3       }
    {DISPLAY     }
    {INDEX       } ]

[ ;{SIGN IS} {TRAILING} {SEPARATE CHARACTER}]

[ ;{Occurs {integer-1 TIMES
                {integer-1 TO integer-2 TIMES DEPENDING ON data-name-3}
                [INDEXED BY index-name-1 [,index-name-2] ... ]}

[ ;{SYNCHRONIZED} [LEFT ]
    {SYNC        } [RIGHT] ]

[ ;{JUSTIFIED} RIGHT]
    {JUST        }

[ ;BLANK WHEN ZERO]

[ ;VALUE IS literal] .
```

FORMAT 2

```
66 data-name-1; RENAMES data-name-2 [{THROUGH} data-name-3].
{THRU }
```

FORMAT 3

```
88 condition-name; {VALUE IS } literal-1 [{THROUGH} literal-2]
{VALUES ARE} {THRU }

[,literal-3 [{THROUGH} literal-4]] ... .
{THRU }
```

The clauses may be written in any order with two exceptions:

the data-name-1 or FILLER clause must immediately follow the level-number;

the REDEFINES clause, when used, must immediately follow the data-name-1 clause.

The PICTURE clause must be specified for every elementary item except an index data item, in which case use of this clause is prohibited.

The words THRU and THROUGH are equivalent.

The clauses SYNCHRONIZED, PICTURE, JUSTIFIED, and BLANK WHEN ZERO, must not be specified except for an elementary data item.

Data Description Entry

Format 3 is used for each condition-name. Each condition-name requires a separate entry with level-number 88. Format 3 contains the name of the condition and the value, values, or range of values associated with the condition-name. The condition-name entries for a particular conditional variable must follow the entry describing the item with which the condition-name is associated. A condition-name can be associated with any data description entry which contains a level-number except the following:

Another condition-name.

A group containing items with descriptions including JUSTIFIED, SYNCHRONIZED or USAGE (other than USAGE IS DISPLAY).

An index data item.

A level 66 item.

Each data description entry must end with a period separator.

The Level-Number

The level-number shows the hierarchy of data within a logical record. In addition, it is used to identify entries for working storage items, linkage items, condition-names and the RENAMES clause.

FORMAT

level-number

A level-number is required as the first element in each data description entry.

Data description entries subordinate to an FD entry must have level-numbers with the values 01 through 49, 66 or 88.

Data description entries in the Working-Storage Section and Linkage Section must have level-numbers with the values 01 through 49, 66, 77 or 88.

The level-number 01 identifies the first entry in each record description.

Level-number 66 is assigned to identify RENAMES entries.

Level-number 77 is assigned to identify noncontiguous working storage data items and noncontiguous linkage data items.

Level-number 88 is assigned to identify condition-names associated with a conditional variable.

Multiple level 01 entries subordinate to any given level indicator FD, represent implicit redefinitions of the same area.

The Data-Name or FILLER Clause

A data-name specifies the name of the data being described. The word FILLER specifies an elementary item of the logical record that cannot be referred to explicitly.

FORMAT

```
{data-name}  
{FILLER} }
```

A data-name or the key word FILLER must be the first word following the level-number in each data description entry.

The key word FILLER may be used to name an elementary item in a record. Under no circumstances can a FILLER item be referred to explicitly. However, the key word FILLER may be used as a conditional variable because such use does not require explicit reference to the FILLER item, but to its value.

The key word FILLER may not be used in data description entries with a 1, 01, 77, or 88 level-number.

The REDEFINES Clause

The REDEFINES clause allows the same computer storage area to be described by different data description entries.

FORMAT

```
level-number data-name-1; REDEFINES data-name-2
```

NOTE: Level-number, data-name-1 and the semicolon are shown in the above format to improve clarity. Level-number and data-name-1 are not part of the REDEFINES clause.

The REDEFINES clause, when specified, must immediately follow data-name-1.

The level-numbers of data-name-1 and data-name-2 must be identical but must not be 66 or 88.

This clause must not be used in level 01 entries in the File Section.

The data description entry for data-name-2 cannot contain a REDEFINES clause. Data-name-2 may be subordinate to an entry which contains a REDEFINES clause. The data description entry for data-name-2 cannot contain an OCCURS clause. However, data-name-2 may be subordinate to an item whose data description entry contains an OCCURS clause. In this case, the reference to data-name-2 in the REDEFINES clause may not be subscripted or indexed. Neither the original definition nor the redefinition can include an item whose size is variable as defined in the OCCURS clause.

No entry having a level-number numerically lower than the level-number of data-name-2 and data-name-1 may occur between the data description entries of data-name-2 and data-name-1.

Redefinition starts at data-name-2 and ends when a level-number less than or equal to that of data-name-2 is encountered.

When the level-number of data-name-1 is other than 01, it must specify the same number of character positions that the data item referenced by data-name-2 contains. It is important to observe that the REDEFINES clause specifies the redefinition of a storage area, not of the data items occupying the area.

REDEFINES Clause

Multiple redefinitions of the same character positions are permitted. The entries giving the new descriptions of the character positions must follow the entries defining the area being redefined without intervening entries that define new character positions. Multiple redefinitions of the same character positions must all use the data-name of the entry that originally defined the area.

The entries giving the new description of the character positions must not contain any VALUE clauses except in condition-name entries.

Multiple level 01 entries subordinate to any given level indicator represent implicit redefinitions of the same area.

The PICTURE Clause

The PICTURE clause describes the general characteristics and editing requirements of an elementary item.

FORMAT

```
{PICTURE} IS character-string  
{PIC }
```

A PICTURE clause can be specified only at the elementary item level.

A character-string consists of certain allowable combinations of characters in the COBOL character set used as symbols. The allowable combinations determine the category of the elementary item.

The maximum number of characters allowed in the character-string is 30.

The PICTURE clause must be specified for every elementary item except an index data item, in which case use of this clause is prohibited.

PIC is an abbreviation for PICTURE.

There are five categories of data that can be described with a PICTURE clause:

- alphanumeric
- numeric
- alphanumeric
- alphanumeric edited
- numeric edited

To define an item as alphabetic:

Its PICTURE character-string can only contain the symbols 'A', and/or 'B'.

Its contents when represented in standard data format must be any combination of the twenty-six (26) letters of the Roman alphabet and the space from the COBOL character set.

PICTURE Clause

To define an item as numeric:

Its PICTURE character-string can only contain the symbols '9', 'P', 'S', and 'V'. The number of digit positions that can be described by the PICTURE character-string must range from 1 to 18 inclusive; and

If unsigned, its contents when represented in standard data format must be a combination of the Arabic numerals '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'; if signed, the item may also contain a '+', '-', or other representation of an operational sign.

To define an item as alphanumeric:

Its PICTURE character-string is restricted to certain combinations of the symbols 'A', 'X', '9', and the item is treated as if the character-string contained all X's. A PICTURE character-string which contains all A's or all 9's does not define an alphanumeric item; and

Its contents, when represented in standard data format, are allowable characters in the computer's character set.

To define an item as alphanumeric edited:

Its PICTURE character-string is restricted to certain combinations of the following symbols: 'A', 'X', '9', 'B', '0', and '/' (stroke);

The character-string must contain at least one 'B' and at least one 'X' or at least one '0' (zero) and at least one 'X' or at least one '/' (stroke) and at least one 'X'; or

The character-string must contain at least one '0' (zero) and at least one 'A' or at least one '/' (stroke) and at least one 'A'; and

The contents when represented in standard data format are allowable characters in the computer's character set.

To define an item as numeric edited:

Its PICTURE character-string is restricted to certain combinations of the following symbols: 'B', '/' (stroke), 'P', 'V', 'Z', '0', '9', ',', '.', '*', '-', '+', 'CR', 'DB', and the currency symbol. The allowable combinations are determined from the order of precedence of symbols and the editing rules; and

The number of digit positions that can be represented in the PICTURE character-string must range from 1 to 18 inclusive; and

The character-string must contain at least one '0', 'B', '/' (stroke), 'Z', '*', '+', ',', '.', '-', 'CR', 'DB', or currency symbol.

The contents of the character positions of these symbols that are allowed to represent a digit in standard data format, must be one of the numerals.

The size of an elementary item, where size means the number of character positions occupied by the elementary item in standard data format, is determined by the number of allowable symbols that represent character positions. An integer which is enclosed in parentheses following the symbols 'A', ',', 'X', '9', 'P', 'Z', '*', 'B', '/' (stroke), '0', '+', '-', or the currency symbol indicates the number of consecutive occurrences of the symbol. Note that the following symbols may appear only once in a given PICTURE: 'S', 'V', '.', 'CR', and 'DB'.

The functions of the symbols used to describe an elementary item are explained as follows:

Each 'A' in the character-string represents a character position which can contain only a letter of the alphabet or a space.

Each 'B' in the character-string represents a character position into which the space character will be inserted.

PICTURE Clause

Each 'P' indicates an assumed decimal scaling position and is used to specify the location of an assumed decimal point when the point is not within the number that appears in the data item. The scaling position character 'P' is not counted in the size of the data item. Scaling position characters are counted in determining the maximum number of digit positions (18) in numeric edited items or numeric items. The scaling position character 'P' can appear only to the left or right as a continuous string of 'P's within a PICTURE description; since the scaling position character 'P' implies an assumed decimal point (to the left of 'P's if 'P's are leftmost PICTURE characters and to the right if 'P's are rightmost PICTURE characters), the assumed decimal point symbol 'V' is redundant as either the leftmost or rightmost character within such a PICTURE description. The character 'P' and the insertion character '.' (period) cannot both occur in the same PICTURE character-string. If, in any operation involving conversion of data from one form of internal representation to another, the data item being converted is described with the PICTURE character 'P', each digit position described by a 'P' is considered to contain the value zero, and the size of the data item is considered to include the digit positions so described.

The letter 'S' is used in a character-string to indicate the presence, but neither the representation nor, necessarily, the position of an operational sign; it must be written as the leftmost character in the PICTURE. The 'S' is counted in determining the size (in terms of standard data format characters) of elementary items having DISPLAY or COMPUTATIONAL usage.

The 'V' is used in a character-string to indicate the location of the assumed decimal point and may only appear once in a character-string. The 'V' does not represent a character position and therefore is not counted in the size of the elementary item. When the assumed decimal point is to the right of the rightmost symbol in the string the 'V' is redundant.

Each 'X' in the character-string is used to represent a character position which contains any allowable character from the computer's character set.

Each 'Z' in a character-string may only be used to represent the leftmost leading numeric character positions which will be replaced by a space character when the contents of that character position is zero. Each 'Z' is counted in the size of the item.

Each '9' in the character-string represents a character position which contains a numeral and is counted in the size of the item.

Each '0' (zero) in the character-string represents a character position into which the numeral zero will be inserted. The '0' is counted in the size of the item.

Each '/' (stroke) in the character-string represents a character position into which the stroke character will be inserted. The '/' (stroke) is counted in the size of the item.

Each ',' (comma) in the character-string represents a character position into which the character ',' will be inserted. This character position is counted in the size of the item. The insertion character ',' must not be the last character in the PICTURE character-string.

When the character '.' (period) appears in the character-string it is an editing symbol which represents the decimal point for alignment purposes and in addition, represents a character position into which the character '.' will be inserted. The character '.' is counted in the size of the item. For a given program the functions of the period and comma are exchanged if the clause DECIMAL-POINT IS COMMA is stated in the SPECIAL-NAMES paragraph. In this exchange the rules for the period apply to the comma and the rules for the comma apply to the period wherever they appear in a PICTURE clause. The insertion character '.' must not be the last character in the PICTURE character-string.

+, -, CR, DB. These symbols are used as editing sign control symbols. When used, they represent the character position into which the editing sign control symbol will be placed. The symbols are mutually exclusive in any one character-string and each character used in the symbol is counted in determining the size of the data item.

Each '*' (asterisk) in the character-string represents a leading numeric character position into which an asterisk will be placed when the contents of that position is zero. Each '*' is counted in the size of the item.

The asterisk when used as the zero suppression symbol and the clause BLANK WHEN ZERO may not appear in the same entry.

PICTURE Clause

The currency symbol in the character-string represents a character position into which a currency symbol is to be placed. The currency symbol in a character-string is represented by either the currency sign or by the single character specified in the CURRENCY SIGN IS clause in the SPECIAL-NAMES paragraph. The currency symbol is counted in the size of the item.

There are two general methods of performing editing in the PICTURE clause, either by insertion or by suppression and replacement. There are four types of insertion editing available:

- Simple insertion
- Special insertion
- Fixed insertion
- Floating insertion

There are two types of suppression and replacement editing:

- Zero suppression and replacement with spaces
- Zero suppression and replacement with asterisks

The type of editing which may be performed upon an item is dependent upon the category to which the item belongs. The following table specifies which type of editing may be performed upon a given category:

CATEGORY	TYPE OF EDITING
Alphabetic	Simple insertion 'B' only
Numeric	None
Alphanumeric	None
Alphanumeric Edited	Simple insertion '0', 'B', and '/' (stroke)
Numeric Edited	All, subject to rules below

Floating insertion editing and editing by zero suppression and replacement are mutually exclusive in a PICTURE clause. Only one type of replacement may be used with zero suppression in a PICTURE clause.

Simple Insertion Editing

The ',' (comma), 'B' (space), '0' (zero), and '/' (stroke) are used as the insertion characters. The insertion characters are counted in the size of the item and represent the position in the item into which the character will be inserted.

Special Insertion Editing

The '.' (period) is used as the insertion character. In addition to being an insertion character it also represents the decimal point for alignment purposes. The insertion character used for the actual decimal point is counted in the size of the item. The use of the assumed decimal point, represented by the symbol 'V' and the actual decimal point, represented by the insertion character, in the same PICTURE character-string is disallowed. The result of special insertion editing is the appearance of the insertion character in the item in the same position as shown in the character-string.

Fixed Insertion Editing

The currency symbol and the editing sign control symbols, '+', '- ', 'CR', 'DB', are the insertion characters. Only one currency symbol and only one of the editing sign control symbols can be used in a given PICTURE character-string. When the symbols 'CR' or 'DB' are used they represent two character positions in determining the size of the item and they must represent the rightmost character positions that are counted in the size of the item.

The symbol '+' or '-', when used, must be either the leftmost or rightmost character position to be counted in the size of the item.

The currency symbol must be the leftmost character position to be counted in the size of the item except that it can be preceded by either a '+' or a '-' symbol.

Fixed insertion editing results in the insertion character occupying the same character position in the edited item as it occupied in the PICTURE character-string.

PICTURE Clause - Editing

Editing sign control symbols produce the following results depending upon the value of the data item:

EDITING SYMBOL IN PICTURE CHARACTER-STRING	RESULT	
	DATA ITEM POSITIVE OR ZERO	DATA ITEM NEGATIVE
+	+	-
-	space	-
CR	2 spaces	CR
DB	2 spaces	DB

Floating Insertion Editing

The currency symbol and editing sign control symbols, '+' or '-', are the floating insertion characters and as such are mutually exclusive in a given PICTURE character-string.

Floating insertion editing is indicated in a PICTURE character-string by using a string of at least two of the floating insertion characters. This string of floating insertion characters may contain any of the fixed insertion symbols or have fixed insertion characters immediately to the right of this string. These simple insertion characters are part of the floating string.

The leftmost character of the floating insertion string represents the leftmost limit of the floating symbol in the data item. The rightmost character of the floating string represents the rightmost limit of the floating symbols in the data items.

The second floating character from the left represents the leftmost limit of the numeric data that can be stored in the data item. Nonzero numeric data may replace all the characters at or to the right of this limit.

In a PICTURE character-string, there are only two ways of representing floating insertion editing. One way is to represent any or all of the leading numeric character positions on the left of the decimal point by the insertion character. The other way is to represent all of the numeric character positions in the PICTURE character-string by the insertion character.

If the insertion characters are only to the left of the decimal point in the PICTURE character-string, the result is that a single floating insertion character will be placed into the character position immediately preceding either the decimal point or the first nonzero digit in the data represented by the insertion symbol string, whichever is farther to the left in the PICTURE character-string. The character positions preceding the insertion character are replaced with spaces.

If all numeric character positions in the PICTURE character-string are represented by the insertion character, the result depends upon the value of the data. If the value is zero the entire data item will contain spaces. If the value is not zero, the result is the same as when the insertion character is only to the left of the decimal point.

To avoid truncation, the minimum size of the PICTURE character-string for the receiving data item must be the number of characters in the sending data item, plus the number of non-floating insertion characters being edited into the receiving data item, plus one for the floating insertion character.

Zero Suppression Editing

The suppression of leading zeroes in numeric character positions is indicated by the use of the alphabetic character 'Z' or the character '*' (asterisk) as suppression symbols in a PICTURE character-string. These symbols are mutually exclusive in a given PICTURE character-string. Each suppression symbol is counted in determining the size of the item. If 'Z' is used the replacement character will be the space and if the asterisk is used, the replacement character will be '*'.

In a PICTURE character-string, there are only two ways of representing zero suppression. One way is to represent any or all of the leading numeric character positions to the left of the decimal point by suppression symbols. The other way is to represent all of the numeric character positions in the PICTURE character-string by suppression symbols.

If the suppression symbols appear only to the left of the decimal point, any leading zero in the data which corresponds to a symbol in the string is replaced by the replacement character. Suppression terminates at the first nonzero digit in the data represented by the suppression symbol string or at the decimal point, whichever is encountered first.

PICTURE Clause - Editing

If all numeric character positions in the PICTURE character-string are represented by suppression symbols and the value of the data is not zero the result is the same as if the suppression characters were only to the left of the decimal point. If the value is zero and the suppression symbol is 'Z', the entire data item will be spaces. If the value is zero and the suppression symbol is '**', the data item will be all '*' except for the actual decimal point.

The symbols '+', '−', '*', 'Z', and the currency symbol, when used as floating replacement characters, are mutually exclusive within a given character-string.

The picture precedence chart shows the order of precedence when using characters as symbols in a character-string. An 'X' at an intersection indicates that the symbol(s) at the top of the column may precede, in a given character-string, the symbol(s) at the left of the row. Arguments appearing in braces indicate that the symbols are mutually exclusive. The currency symbol is indicated by the symbol 'cs'.

At least one of the symbols 'A', 'X', 'Z', '9', or '**', or at least two of the symbols '+', '−', or 'cs' must be present in a PICTURE string.

Nonfloating insertion symbols '+' and '−', floating insertion symbols 'Z', '**', '+', '−', and 'cs', and other symbol 'P' appear twice in the PICTURE character precedence chart. The leftmost column and uppermost row for each symbol represents its use to the left of the decimal point position. The second appearance of the symbol in the chart represents its use to the right of the decimal point position.

\lst \Sym-	Non-Floating Insertion Symbols	Floating Insertion Symbols	Other Symbols
2nd\bol Sym-\bol	B 0 / ' . {+} {+} {CR} CS {-} {-} {DB}	{z} {z} {+} {+} CS CS { *} { *} { -} { -}	9 A S V P P X
B	X X X X X X X	X X X X X X X X X X X X X X	
O	X X X X X X X	X X X X X X X X X X X X X X	
N	/ X X X X X X X	X X X X X X X X X X X X X X	
O	, X X X X X X X	X X X X X X X X X X X X X X	
L	. X X X X X X	X X X X X X X X	
A	+ -		
I	-		
N	+ - X X X X X X	X X X X X X X X X X X	
G	CR DB X X X X X X	X X X X X X X X X X X	
C	CS X		
S	Z * X X X X X X	X	
F	Z * X X X X X X X	X X X X	
L	+ - X X X X X	X	
O	+ - X X X X X	X	
A	+ - X X X X X X	X X	
I	+ - X X X X X X	X X	
N	CS X X X X X	X	
G	CS X X X X X X	X X X X	
C	9 X X X X X X X	X X X X X X X X X	
A	A X X X X	X X	
O	S		
T	V X X X X X X	X X X X X X X X X	
H	P X X X X X X	X X X X X X X X X	
E	P X X	X X X	

PICTURE Character Precedence Chart

USAGE Clause

The USAGE Clause

The USAGE clause specifies the format of a data item in the computer storage.

FORMAT

```
[USAGE IS] {COMPUTATIONAL}  
    {COMP}  
    {COMPUTATIONAL-1}  
    {COMP-1}  
    {COMPUTATIONAL-3}  
    {COMP-3}  
    {DISPLAY}  
    {INDEX}
```

This clause specifies the manner in which a data item is represented in the storage of a computer. It does not affect the use of the data item, although the specifications for some statements in the Procedure Division may restrict the USAGE clause of the operands referenced.

The USAGE clause can be written at any level. If the USAGE clause is written at a group level, it applies to each elementary item in the group. The USAGE clause of an elementary item cannot contradict the USAGE clause of a group to which the item belongs.

If the USAGE clause is not specified for an elementary item, or for any group to which the item belongs, the usage is implicitly DISPLAY.

A COMPUTATIONAL (COMPUTATIONAL-1, COMPUTATIONAL-3) item represents a value to be used in computations and must be numeric. If a group is described as COMPUTATIONAL, then the elementary items in the group are COMPUTATIONAL. The group itself is not COMPUTATIONAL (cannot be used in computations.)

The format of a COMPUTATIONAL item is one decimal digit per character position (hexadecimal 00-09). If an 'S' appears in the PICTURE character-string, a trailing byte contains the sign with > 2B being generated for positive and > 2D being generated for negative. COMPUTATIONAL items will be treated as negative if the sign character is > 2D; otherwise they will be considered positive.

The format of a COMPUTATIONAL-1 item (abbreviated COMP-1) is 16 bit two's complement signed binary, independent of the number of nines or appearance of 'S' in the PICTURE character-string. The number of nines is significant when the value is converted to decimal during data manipulation. The value of a COMPUTATIONAL-1 item ranges between -32768 and 32767.

The format of a COMPUTATIONAL-3 item is two decimal digits per character position.

The PICTURE character-string of a COMPUTATIONAL, COMPUTATIONAL-1 or COMPUTATIONAL-3 item can contain only '9's, the operational sign character 'S', the implied decimal point character 'V', one or more 'P's. Since a COMPUTATIONAL-1 item must have zero scale it cannot contain any 'P's in its PICTURE character string and if it has a 'V' in its PICTURE character-string the 'V' must be the rightmost character.

The USAGE IS DISPLAY clause indicates that the format of the data is ASCII.

An elementary item described with the USAGE IS INDEX clause is called an index data item and contains a value which must correspond to an occurrence number of a table element. If a group item is described with the USAGE IS INDEX clause the elementary items in the group are all index data but the group item name cannot be used in the SET statement or in a relation condition.

An index data item can be referenced explicitly only in a SET statement or a relation condition.

The initial value of an index item is undefined.

The SYNCHRONIZED, JUSTIFIED, PICTURE, VALUE and BLANK WHEN ZERO clauses cannot be used to describe group or elementary items described with the USAGE IS INDEX clause.

An index data item can be part of a group which is referred to in a MOVE or input-output statement, in which case no conversion will take place.

The external and internal format of an index data item is the same as a COMPUTATIONAL-1 item.

SIGN Clause

The SIGN Clause

The SIGN clause specifies the position and the mode of representation of the operational sign when it is necessary to describe these properties explicitly.

FORMAT

[SIGN IS] {TRAILING} [SEPARATE CHARACTER]

The optional SIGN clause, if present, specifies the position and the mode of representation of the operational sign for the numeric data description entry to which it applies, or for each numeric data description entry subordinate to the group to which it applies. The SIGN clause applies only to numeric data description entries whose PICTURE contains the character 'S'.

The operational sign will be presumed to be the trailing character position of the elementary numeric data item; this character position is not a digit position.

The letter 'S' in a PICTURE character-string is counted in determining the size of the item (in terms of standard data format characters).

The operational signs for positive and negative are the standard data format characters '+' and '-', respectively.

The numeric data description entries to which the SIGN clause applies must be described as usage is DISPLAY.

At most one SIGN clause may apply to any given numeric data description entry.

The OCCURS Clause

The OCCURS clause eliminates the need for separate entries for repeated data items and supplies information required for the application of subscripts or indices.

FORMAT 1

```
OCCURS integer-1 TIMES
    [INDEXED BY index-name-1 [,index-name-2] ...]
```

FORMAT 2

```
OCCURS integer-1 TO integer-2 TIMES DEPENDING ON data-name-1
    [INDEXED BY index-name-1 [,index-name-2] ...]
```

The OCCURS clause is used in defining tables and other homogeneous sets of repeated data items. Whenever the OCCURS clause is used, the data-name which is the subject of this entry must be either subscripted or indexed whenever it is referred to in a statement. Further, if the subject of this entry is the name of a group item, then all data-names belonging to the group must be subscripted or indexed whenever they are used as operands, except as the object of a REDEFINES clause.

The OCCURS clause cannot be specified in a data description entry that:

Has an 01, 66, 77, or an 88 level-number.

Describes an item whose size is variable. The size of an item is variable if the data description of any subordinate item contains Format 2 of the OCCURS clause.

Except for the OCCURS clause itself, all data description clauses associated with an item whose description includes an OCCURS clause apply to each occurrence of the item described.

OCCURS Clause

The number of occurrences of the subject entry is defined as follows:

In Format 1, the value of integer-1 represents the exact number of occurrences.

In Format 2, the current value of the data item referenced by data-name-1 represents the number of occurrences.

This format specifies that the subject of this entry has a variable number of occurrences. The value of integer-2 represents the maximum number of occurrences and the value of integer-1 represents the minimum number of occurrences. This does not imply that the length of the subject of the entry is variable, but that the number of occurrences is variable.

The value of the data item referenced by data-name-1 must fall within the range integer-1 through integer-2. Reducing the value of the data item referenced by data-name-1 makes the contents of data items, whose occurrence numbers now exceed the value of the data item referenced by data-name-1, unpredictable.

Where both integer-1 and integer-2 are used, the value of integer-1 must be less than the value of integer-2.

The data description of data-name-1 must describe a positive integer. Data-name-1 may be qualified.

A data description entry that contains Format 2 of the OCCURS clause may only be followed, within that record description, by data description entries which are subordinate to it.

When a group item, having subordinate to it an entry that specifies Format 2 of the OCCURS clause, is referenced, only that part of the table area that is specified by the value of data-name-1 will be used in the operation.

An INDEXED BY phrase is required if the subject of this entry, or an entry subordinate to this entry, is to be referred to by indexing. The index-name identified by this clause is not defined elsewhere since its allocation and format are dependent on the hardware, and not being data, cannot be associated with any data hierarchy.

The SYNCHRONIZED Clause

The SYNCHRONIZED clause specifies the alignment of an elementary item on an even byte boundary.

FORMAT

```
{SYNCHRONIZED} [LEFT ]
{SYNC } [RIGHT]
```

This clause specifies that the subject data item is to be aligned in the computer such that no other data item occupies any of the character positions between the leftmost and rightmost natural boundaries delimiting this data item. If the number of character positions required to store this data item is less than the number of character positions between those natural boundaries, the unused character positions (or portions thereof) must not be used for any other data item. Such unused character positions, however, are included in:

The size of any group item(s) to which the elementary item belongs; and

The character positions redefined when this data item is the object of a REDEFINES clause.

SYNCHRONIZED LEFT specifies that the elementary item is to be positioned such that it will begin at the left character position of the next available even byte. If the data item contains an odd number of bytes, one trailing byte of FILLER is implied.

SYNCHRONIZED not followed by either RIGHT or LEFT is equivalent to the clause SYNCHRONIZED LEFT.

SYNC is an abbreviation for SYNCHRONIZED.

This clause may only appear with an elementary item.

SYNCHRONIZED RIGHT specifies that the elementary item is to be positioned such that it will terminate on the right character position of an integral even byte boundary. If the data item contains an odd number of bytes, a leading byte of FILLER is implied.

SYNCHRONIZED Clause

Whenever a SYNCHRONIZED item is referenced in the source program, the original size of the item, as shown in the PICTURE clause, is used in determining any action that depends on size, such as justification, truncation or overflow.

If the data description of an item contains the SYNCHRONIZED clause and an operational sign, the sign of the item appears in the normal operational sign position, regardless of whether the item is SYNCHRONIZED LEFT or SYNCHRONIZED RIGHT.

When the SYNCHRONIZED clause is specified in a data description entry of a data item that also contains an OCCURS clause, or in a data description entry of a data item subordinate to a data description entry that contains an OCCURS clause, then:

Each occurrence of the data item is SYNCHRONIZED.

Any implicit FILLER generated for other data items within that same table are generated for each occurrence of those data items.

Records of a file and index data items are automatically synchronized left. Records and noncontiguous data-items in working-storage begin on the next available byte unless the first elementary item is synchronized.

The format on external media of records or groups containing elementary items described with the SYNCHRONIZED clause includes any implied FILLER bytes.

When the data item preceding a data item described with the SYNCHRONIZED clause does not terminate on a byte whose address is even, then one implied FILLER byte is generated. Such automatically generated FILLER positions are included in:

The size of any group to which the FILLER item belongs; and

The number of character positions allocated when the group item of which the FILLER item is a part appears as the object of a REDEFINES clause.

The JUSTIFIED Clause

The JUSTIFIED clause specifies nonstandard positioning of data within a receiving data item.

FORMAT

```
{JUSTIFIED} RIGHT  
{JUST }
```

When a receiving data item is described with the JUSTIFIED clause and the sending data item is larger than the receiving data item, the leftmost characters are truncated. When the receiving data item is described with the JUSTIFIED clause and it is larger than the sending data item, the data is aligned at the rightmost character position in the data item with space-fill for the leftmost character positions.

When the JUSTIFIED clause is omitted, the standard rules for aligning data within an elementary item apply.

The JUSTIFIED clause cannot be specified for any data item described as numeric or for which editing is specified.

The JUSTIFIED clause can be specified only at the elementary item level.

JUST is an abbreviation for JUSTIFIED.

BLANK WHEN ZERO Clause

The BLANK WHEN ZERO Clause

The BLANK WHEN ZERO clause permits the blanking of an item when its value is zero.

FORMAT

BLANK WHEN ZERO

The BLANK WHEN ZERO clause can be used only for an elementary item whose PICTURE is specified as numeric or numeric edited.

The BLANK WHEN ZERO clause cannot appear in the same entry with a PICTURE clause having an asterisk as the zero suppression symbol.

When the BLANK WHEN ZERO clause is used, the item will contain nothing but spaces if the value of the item is zero.

When the BLANK WHEN ZERO clause is used for an item whose PICTURE is numeric, the category of the item is considered to be numeric edited.

The VALUE IS Clause

The VALUE IS clause defines the initial value of working storage items, and the values associated with a condition-name.

FORMAT 1

VALUE IS literal

FORMAT 2

```
{VALUE IS } literal-1 [{THROUGH} literal-2]
{VALUES ARE} {THRU }

[,literal-3 [{THROUGH} literal-4]] ...
{THRU }
```

The VALUE clause cannot be stated for any items whose size is variable.

A signed numeric literal must have associated with it a signed numeric PICTURE character-string.

All numeric literals in a VALUE clause of an item must have a value which is within the range of values indicated by the PICTURE clause, and must not have a value which would require truncation of nonzero digits. Nonnumeric literals in a VALUE clause of an item must not exceed the size indicated by the PICTURE clause.

The words THRU and THROUGH are equivalent.

The VALUE clause must not conflict with other clauses in the data description of the item or in the data description within the hierarchy of the item. The following rules apply:

1. If the category of the item is numeric, all literals in the VALUE clause must be numeric. If the literal defines the value of a working storage item, the literal is aligned in the data item according to the standard alignment rules.

VALUE IS Clause

2. If the category of the item is alphabetic, alphanumeric, alphanumeric edited or numeric edited, all literals in the VALUE clause must be nonnumeric literals. The literal is aligned in the data item as if the data item had been described as alphanumeric. Editing characters in the PICTURE clause are included in determining the size of the data item but have no effect on initialization of the data item. Therefore, the VALUE of an edited item is presented in an edited form.

Initialization takes place independent of any BLANK WHEN ZERO or JUSTIFIED clause that may be specified.

A figurative constant may be substituted in both Format 1 and Format 2 wherever a literal is specified.

Condition-Name Rules

In a condition-name entry, the VALUE clause is required. The VALUE clause and the condition-name itself are the only two clauses permitted in the entry. The characteristics of a condition-name are implicitly those of its conditional variable.

Format 2 can be used only in connection with condition-names. Wherever the THROUGH (THRU) phrase is used, literal-1 must be less than literal-2, literal-3 less than literal-4, etc.

Data Description Entries Other Than Condition-Names

Rules governing the use of the VALUE clause differ with the respective sections of the Data Division:

In the File Section, the VALUE clause may be used only in condition-name entries.

In the Working-Storage Section, the VALUE clause must be used in condition-name entries. The VALUE clause may also be used to specify the initial value of any other data item; in which case the clause causes the item to assume the specified value at the start of the object program. If the VALUE clause is not used in an item's description, the initial value is undefined.

In the Linkage Section, the VALUE clause may be used only in condition-name entries.

The VALUE clause must not be stated in a data description entry that contains an OCCURS clause, or in an entry that is subordinate to any entry containing a REDEFINES clause. This rule does not apply to condition-name entries.

If the VALUE clause is used in an entry at the group level, the literal must be a figurative constant or a nonnumeric literal, and the group area is initialized without consideration for the individual elementary or group items contained within this group. The VALUE clause cannot be stated at the subordinate levels within this group.

The VALUE clause must not be written for a group containing items with descriptions including JUSTIFIED, SYNCHRONIZED, or USAGE (other than USAGE IS DISPLAY).

RENAMES Clause

The RENAMES Clause

The RENAMES clause permits alternative, possibly overlapping, groupings of elementary items.

FORMAT

66 data-name-1;

RENAMEs data-name-2 [{THROUGH} data-name-3].
 {THRU }

NOTE: Level-number 66, data-name-1 and the semicolon are shown in the above format to improve clarity. Level-number and data-name-1 are not part of the RENAMES clause.

All RENAMES entries referring to data items within a given logical record must immediately follow the last data description entry of the associated record description entry.

Data-name-2 and data-name-3 must be names of elementary items or groups of elementary items in the same logical record, and cannot be the same data-name. A 66 level entry cannot rename another 66 level entry nor can it rename a 77, 88, or 01 level entry.

Data-name-1 cannot be used as a qualifier, and can only be qualified by the names of the associated level 01 or FD entries. Neither data-name-2 nor data-name-3 may have an OCCURS clause in its data description entry nor be subordinate to an item that has an OCCURS clause in its data description entry.

The beginning of the area described by data-name-3 must not be to the left of the beginning of the area described by data-name-2. The end of the area described by data-name-3 must be to the right of the end of the area described by data-name-2. Data-name-3, therefore, cannot be subordinate to data-name-2.

Data-name-2 and data-name-3 may be qualified.

None of the items within the range, including data-name-2 and data-name-3, if specified, can be an item whose size is variable as defined in the OCCURS clause.

One or more RENAMES entries can be written for a logical record.

When data-name-3 is specified, data-name-1 is a group item which includes all elementary items starting with data-name-2 (if data-name-2 is an elementary item) or the first elementary item in data-name-2 (if data-name-2 is a group item), and concluding with data-name-3 (if data-name-3 is an elementary item) or the last elementary item in data-name-3 (if data-name-3 is a group item).

When data-name-3 is not specified, data-name-2 can be either a group or an elementary item; when data-name-2 is a group item, data-name-1 is treated as a group item, and when data-name-2 is an elementary item, data-name-1 is treated as an elementary item.

The words THRU and THROUGH are equivalent.

DATA STRUCTURES

Classes of Data

The five categories of data items (see the PICTURE Clause) are grouped into three classes:

alphabetic
numeric
alphanumeric

For alphabetic and numeric, the classes and categories are synonymous.

The alphanumeric class includes the categories of alphanumeric edited, numeric edited and alphanumeric (without editing).

Every elementary item except for an index data item belongs to one of the classes and further to one of the categories. The class of a group item is treated at object time as alphanumeric regardless of the class of elementary items subordinate to that group item.

The following chart depicts the relationship of the class and categories of data items:

LEVEL OF ITEM	CLASS	CATEGORY
Elementary	Alphabetic	Alphabetic
	Numeric	Numeric
	Alphanumeric	Numeric Edited Alphanumeric Edited Alphanumeric
Nonelementary (Group)	Alphanumeric	Alphabetic Numeric Numeric Edited Alphanumeric Edited Alphanumeric

Representation of Numeric Items

The value of a numeric item may be represented in either binary, decimal or packed decimal form depending on the USAGE clause associated with the item. There are two ways of expressing decimal: DISPLAY and COMPUTATIONAL. Binary is COMPUTATIONAL-1. Packed decimal is COMPUTATIONAL-3.

The selection of the proper representation is dependent upon the usage of the numeric item. Items which must be used for input and output should be of DISPLAY usage to eliminate conversions to external forms. For efficiency of arithmetic operations, COMPUTATIONAL, COMPUTATIONAL-1, or COMPUTATIONAL-3 should be used. To reduce conversions and increase efficiency, types should not be mixed in operations except where required by program needs.

Representation of Algebraic Signs

Algebraic signs fall into two categories:

operational signs which are associated with signed numeric data items, and signed numeric literals to indicate their algebraic properties; and

editing signs which appear to identify the sign of the item.

For DISPLAY, COMPUTATIONAL, and COMPUTATIONAL-3, an unsigned numeric item is assumed to have an operational sign which is positive and will receive the absolute value of signed items. A signed numeric item maintains the operational sign as a separate trailing character.

For COMPUTATIONAL-1 (which is always signed), the operational sign is maintained as part of the item in two's complement signed binary form.

Editing signs are inserted into a data item through the use of the sign control symbols of the PICTURE clause.

Standard Alignment Rules

The standard rules of positioning data within an elementary item depend on the category of the receiving item:

If the receiving data item is described as numeric:

- a. The data is aligned by decimal point and is moved to the receiving character positions with zero fill or truncation on either end as required.
- b. When an assumed decimal point is not explicitly specified, the data item is treated as if it had an assumed decimal point immediately following its rightmost character and is aligned as in a. above.

If the receiving data item is a numeric edited data item, the data moved to the edited data item is aligned by decimal point with zero-fill or truncation at either end as required within the receiving character positions of the data item, except where editing requirements cause replacement of the leading zeros.

If the receiving data item is alphanumeric (other than a numeric edited data item), alphanumeric edited or alphabetic, the sending data is moved to the receiving character positions and aligned at the leftmost character position in the data item with space-fill or truncation to the right, as required.

If the JUSTIFIED clause is specified for the receiving item, these standard rules are modified as described in the JUSTIFIED clause.

QUALIFICATION

Every user-specified name that defines an element in a COBOL source program must be unique, either because no other name has the identical spelling and hyphenation, or because the name exists within a hierarchy of names such that references to the name can be made unique by mentioning one or more of the higher levels of the hierarchy. The higher levels are called qualifiers and this process that specifies uniqueness is called qualification. Enough qualification must be mentioned to make the name unique; however, it may not be necessary to mention all levels of the hierarchy. Within the Data Division, all data-names used for qualification must be associated with a level indicator or a level-number. Therefore, two identical data-names must not appear as entries subordinate to a group item unless they are capable of being made unique through qualification.

In the hierarchy of qualification, names associated with a level indicator are the most significant, then those names associated with level-number 01, then names associated with level-number 02, ..., 49. The most significant name in the hierarchy must be unique and cannot be qualified.

Qualification is performed by following a data-name, by one or more phrases composed of a qualifier preceded by IN or OF. IN and OF are logically equivalent.

FORMAT 1

```
{data-name-1} [{OF} data-name-2 ]...  
{condition-name} {IN}
```

FORMAT 2

```
paragraph-name [{OF} section-name]  
              {IN}
```

Qualification

The rules for qualification are as follows:

1. Each qualifier must be of a successively higher level and within the same hierarchy as the name it qualifies.
2. The same name must not appear at two levels in a hierarchy.
3. If a data name is assigned to more than one data item in a source program, the data-name must be qualified each time it is referred to in the Procedure, Environment, and Data Divisions (except in the REDEFINES clause where qualification is unnecessary and must not be used.)
4. A paragraph-name must not be duplicated within a section. When a paragraph-name is qualified by a section-name, the word SECTION must not appear. A paragraph-name need not be qualified when referred to from within the same section.
5. A data-name cannot be subscripted when it is being used as a qualifier.
6. A name can be qualified even though it does not need qualification: if there is more than one combination of qualifiers that ensures uniqueness, then any such set can be used. The complete set of qualifiers for a data-name must not be the same as any partial set of qualifiers for another data-name. Qualified data-names may have any number of qualifiers up to a limit of 49.

SUBSCRIPTING

Subscripts can be used only when reference is made to an individual element within a list of a table of like elements that have not been assigned individual data-names (see The OCCURS Clause).

The subscript can be represented either by a numeric literal that is an integer or by a data-name. The data name must be a numeric elementary item that represents an integer. When the subscript is represented by a data-name, the data-name may be qualified but not subscripted.

The subscript may be signed and, if signed, it must be positive. The lowest possible subscript value is 1. This value points to the first element of the table. The next sequential elements of the table are pointed to by subscripts whose values are 2, 3, ...n. The highest permissible subscript value, in any particular case, is the maximum number of occurrences of the item as specified in the OCCURS clause.

The subscript, or set of subscripts, that identifies the table element is delimited by the balanced pair of separators, left parenthesis and right parenthesis, following the table element data-name. The table element data-name appended with a subscript is called a subscripted data-name or an identifier. When more than one subscript is required, they are written in the order of successively less inclusive dimensions of the data organization.

FORMAT

```
{data-name      } (subscript-1 [subscript-2 [,subscript-3]])  
{condition-name}
```

INDEXING

References can be made to individual elements within a table of like elements by specifying indexing for that reference. An index is assigned to that level of the table by using the INDEXED BY phrase in the definition of a table. A name given in the INDEXED BY phrase is known as an index-name and is used to refer to the assigned index. The value of an index corresponds to the occurrence number of an element in the associated table. An index-name must be initialized before it is used as a table reference. An index-name can be given an initial value by a SET statement, or a FORMAT 4 PERFORM statement.

Direct indexing is specified by using an index-name in the form of a subscript. Relative indexing is specified when the index-name is followed by the operator + or -, followed by an unsigned integer numeric literal all delimited by the balanced pair of separators, left parenthesis and right parenthesis, following the table element data-name. The occurrence number resulting from relative indexing is determined by incrementing (where the operator + is used) or decrementing (when the operator - is used), by the value of the literal, the occurrence number represented by the value of the index. When more than one index-name is required, they are written in the order of successively less inclusive dimensions of the data organization.

At the time of execution of a statement which refers to an indexed table element, the value contained in the index referenced by the index-name associated with the table element must neither correspond to a value less than one (1) nor to a value greater than the highest permissible occurrence number of an element of the associated table. This restriction also applies to the value resultant from relative indexing.

FORMAT

```
{data-name}      {{index-name-1 [{+} literal-2]}}
{condition-name} {literal-1      {-}          }
                  [,{{index-name-2 [{+} literal-4]}}
                   {literal-3      {-}          }
                  [,{{index-name-3 [{+} literal-6]}]])
                   {literal-5      {-}          }
```

IDENTIFIER

An identifier is a term used to reflect that a data-name, if not unique in a program, must be followed by a syntactically correct combination of qualifiers, subscripts or indices necessary to ensure uniqueness. The general formats for identifiers are:

FORMAT 1

```
data-name-1 [{OF} data-name-2] ... [(subscript-1
    {IN}
    [,subscript-2 [,subscript-3]])]
```

FORMAT 2

```
data-name-1 [{OF} data-name-2] ... [{({index-name-1 [+]} literal-2)}
    {literal-1      {-}}           }
    {IN}
    [, {({index-name-2 [+]} literal-4)}
        {literal-3      {-}}          ]
    [, {({index-name-3 [+]} literal-6)}]]
    {literal-5      {-}}
```

Restrictions on qualification, subscripting and indexing are:

A data-name must not itself be subscripted nor indexed when that data-name is being used as an index, subscript or qualifier.

Indexing is not permitted where subscripting is not permitted.

An index may be modified only by the SET and PERFORM statements. Data items described by the USAGE IS INDEX clause permit storage of the values associated with index-names as data in a form specified by the compiler. Such data items are called index data items.

Literal-1, literal-3, literal-5 in the above format must be positive numeric integers. Literal-2, literal-4, literal-6, must be unsigned numeric integers.

CONDITION-NAME

Each condition-name must be unique, or be made unique through qualification and/or indexing, or subscripting.

If qualification is used to make a condition-name unique, the associated conditional variable may be used as the first qualifier. If qualification is used, the hierarchy of names associated with the conditional variable or the conditional variable itself must be used to make the condition-name unique.

If references to a conditional variable require indexing or subscripting, then references to any of its condition-names also require the same combination of indexing or subscripting.

The format and restrictions on the combined use of qualification, subscripting, and indexing of condition-names is exactly that of 'identifier' except that data-name-1 is replaced by condition-name-1.

In the general formats, 'condition-name' refers to a condition-name qualified, indexed or subscripted, as necessary.

TABLE HANDLING

Tables of data are common components of business data processing problems. Although items of data that make up a table could be described as contiguous data items, there are two reasons why this approach is not satisfactory. First, from a documentation standpoint, the underlying homogeneity of the items would not be readily apparent; and second, the problem of making available an individual element of such a table would be severe when there is a decision as to which element is to be made available at object time.

Tables composed of contiguous data items are defined in COBOL by including the OCCURS clause in their data description entries. This clause specifies that the item is to be repeated as many times as stated. The item is considered to be a table element and its name and description apply to each repetition or occurrence. Since each occurrence of a table element does not have assigned to it a unique data-name, reference to a desired occurrence may be made only by specifying the data-name of the table element together with the occurrence number of the desired table element. Subscripting and indexing are the two methods that are used to specify the occurrence number of a desired table element.

Table Definition

To define a one-dimensional table, the programmer uses an OCCURS clause as part of the data description of the table element, but the OCCURS clause must not appear in the description of group items which contain the table element.

Example 1:

```
01 TABLE-1.  
02 TABLE-ELEMENT OCCURS 20 TIMES.  
03 NAME .....  
03 SSAN .....
```

Table Handling

Defining a one-dimensional table within each occurrence of an element of another one-dimensional table gives rise to a two-dimensional table. To define a two-dimensional table, then, an OCCURS clause must appear in the data description of the element of the table, and in the description of only one group item which contains that table. In the description of a three-dimensional table, the OCCURS clause should appear in the data description of 2 nested group items which contain the element. In COBOL, tables of up to 3 dimensions are permitted.

Example 2 shows a table which has one dimension for CONTINENT-NAME, two dimensions for COUNTRY-NAME, and three dimensions for CITY-NAME and CITY-POPULATION. The table includes 100,510 data items--10 for CONTINENT-NAME, 500 for COUNTRY-NAME, 50,000 for CITY-NAME, and 50,000 for CITY-POPULATION. Within the table there are ten occurrences of CONTINENT-NAME. Within each CONTINENT-NAME there are 50 occurrences of COUNTRY-NAME and within each COUNTRY-NAME there are one hundred occurrences of CITY-NAME and CITY-POPULATION.

Example 2:

```
01 CENSUS-TABLE.  
  05 CONTINENT-TABLE OCCURS 10 TIMES.  
    10 CONTINENT-NAME PIC XXXXXX.  
    10 COUNTRY-TABLE OCCURS 50 TIMES.  
      15 COUNTRY-NAME PIC XXXXXXXX.  
      15 CITY-TABLE OCCURS 100 TIMES.  
        20 CITY-NAME PIC XXXXXXXXXXXX.  
        20 CITY-POPULATION PIC 999999999999.
```

References to Table Items

Whenever the user refers to a table element, the reference must indicate which occurrence of the element is intended. For access to a one-dimensional table, the occurrence number of the desired element provides complete information. For access to tables of more than one dimension, an occurrence number must be supplied for each dimension of the table accessed. In Example 2 then, a reference to the 4th CONTINENT-NAME would be complete, whereas a reference to the 4th COUNTRY-NAME would not. To refer to COUNTRY-NAME, which is an element of a two-dimensional table, the user must refer to, for example, the 4th COUNTRY-NAME within the 6th CONTINENT-TABLE.

One method by which occurrence numbers may be specified is to append one or more subscripts to the data-name. A subscript is an integer whose value specifies the occurrence number of an element. The subscript can be represented either by a literal which is an integer or by a data-name which is defined elsewhere as a numeric elementary item with no character positions to the right of the assumed decimal point. In either case, the subscript, enclosed in parentheses, is written immediately following the name of the table element. A table reference must include as many subscripts as there are dimensions in the table whose element is being referenced. That is, there must be a subscript for each OCCURS clause in the hierarchy containing the data-name, including the data-name itself. In Example 2, references to CONTINENT-NAME require only one subscript, reference to COUNTRY-NAME requires two, and references to CITY-NAME and CITY-POPULATION require three.

When more than one subscript is required, they are written in order of successively less inclusive dimensions of the data organization. When a data-name is used as a subscript, it may be used to refer to items in many different tables. These tables need not have elements of the same size. The data-name may also appear as the only subscript with one item and as one of two or three subscripts with another item. Also, it is permissible to mix literal and data-name subscripts, for example: CITY-POPULATION (10, NEWKEY, 42).

Another method of referring to items in a table is indexing. To use this technique, the programmer assigns one or more index-names (defined with the INDEXED-BY phrase of the OCCURS clause) to an item whose data description contains an OCCURS clause. There is no separate entry to describe the index-name since its definition is completely hardware-oriented and it is not considered data per se. At object time the contents of the index-name will correspond to an occurrence number for that specific dimension of the table to which the index-name was assigned. The initial value of an index-name at object time is not determinable and the index-name must be initialized by the SET statement before use.

When a reference is made to a table element, or to an item within a table element, and the name of the item is followed by its related index-name or names in parentheses, then each occurrence number required to complete the reference will be obtained from the respective index-name. The index-name thus acts as a subscript whose value is used in any table reference that specifies indexing.

VI

PROCEDURE DIVISION

THE PROCEDURE DIVISION

The Procedure Division must be included in every COBOL source program. This division may contain declaratives and nondeclarative procedures.

The Procedure Division is identified by and must begin with the following header:

PROCEDURE DIVISION [USING data-name-1 [,data-name-2] ...] .

The USING phrase is present if and only if the object program is to function under the control of a CALL statement, and the CALL statement in the calling program contains a USING phrase.

Each of the operands in the USING phrase of the Procedure Division header must be defined as a data item in the Linkage Section of the program in which this header occurs, and it must have a 01 or 77 level-number.

Within a called program, Linkage Section data items are processed according to their descriptions given in the called program. Of those items defined in the Linkage Section only data-name-1, data-name-2, items subordinate to these data-names, and condition-names and/or index-names associated with such data-names and/or subordinate data items, may be referenced in the Procedure Division.

When the USING phrase is present, the object program operates as if data-name-1 of the Procedure Division header in the called program and data-name-1 in the USING phrase of the CALL statement in the calling program refer to a single set of data that is equally available to both the called and calling programs. Their definitions must contain the same data descriptions; however, they need not be the same name. In like manner, there is an equivalent relationship between data-name-2, ..., in the USING phrase of the called program and data-name-2, ..., in the USING phrase of the CALL statement in the calling program. A data-name must not appear more than once in the USING phrase in the Procedure Division header of the called program; however, a given data-name may appear more than once in the same USING phrase of a CALL statement.

PROCEDURE DIVISION

Structure

The body of the Procedure Division must conform to one of the following formats:

FORMAT 1

PROCEDURE DIVISION [USING data-name-1 [,data-name-2]...].

[DECLARATIVES.

{section-name SECTION [segment-number]. declarative-sentence
[paragraph-name. [sentence] ...] ...} ...

END DECLARATIVES.]

{section-name SECTION [segment-number].
[paragraph-name. [sentence] ...] ...} ...

[END PROGRAM].

FORMAT 2

PROCEDURE DIVISION [USING data-name-1 [,data-name-2]...].

{paragraph-name. [sentence] ...} ...

[END PROGRAM].

The segment-number must be an integer ranging in value from 0 through 127.

If the segment-number is omitted from the section header, the segment-number is assumed to be 0.

Sections in the declaratives must contain segment-numbers less than 50.

All sections which have the same segment-number constitute a program segment. Sections with the same segment-number must be physically contiguous in the source program.

Segments with segment-numbers 0 through 49 belong to the fixed portion of the object program. Segments with segment-numbers 50 through 127 are independent segments. Independent segments must follow fixed segments.

Declaratives

Declarative sections must be grouped at the beginning of the Procedure Division preceded by the key word DECLARATIVES and followed by the key words END DECLARATIVES.

Procedures

A procedure is composed of a paragraph, or group of successive paragraphs, or a section, or a group of successive sections within the Procedure Division. If one paragraph is in a section, then all paragraphs must be in sections. A procedure-name is a word used to refer to a paragraph or section. It consists of a paragraph-name (which may be qualified), or a section-name.

A section consists of a section header followed by zero, or more successive paragraphs. A section ends immediately before the next section or at the end of the Procedure Division or, in the declaratives portion of the Procedure Division, at the key words END DECLARATIVES.

A paragraph consists of a paragraph-name followed by a period and a space and by zero, or more successive sentences. A paragraph ends immediately before the next paragraph-name or section-name or at the end of the Procedure Division or, in the declaratives portion of the Procedure Division, at the key words END DECLARATIVES. A paragraph-name must not be duplicated within a section.

Execution

Execution begins with the first statement of the Procedure Division, excluding declaratives. Statements are then executed in the order in which they are presented for compilation, except where the rules indicate some other order.

PROCEDURE REFERENCES

A procedure is referred to by its paragraph-name or section-name. Paragraph-names may be qualified by the section-name of the section containing the paragraph, whether or not it needs qualification. When referring to a section-name or when using a section-name as a qualifier, the word SECTION must not appear. Qualification is performed by following a paragraph-name with a section-name preceded by IN or OF. IN and OF are logically equivalent. The general format for paragraph qualification is:

```
paragraph-name [{OF} section-name]  
           {IN}
```

A paragraph-name need not be qualified when referred to from within the same section or when the paragraph-name is unique.

Explicit and Implicit Transfers of Control

The mechanism that controls program flow transfers control from statement to statement in the sequence in which they were written in the source program unless an explicit transfer of control overrides this sequence or there is no next executable statement to which control can be passed. The transfer of control from statement to statement occurs without the writing of an explicit Procedure Division statement, and therefore, is an implicit transfer of control.

COBOL provides both explicit and implicit means of altering the implicit control transfer mechanism.

In addition to the implicit transfer of control between consecutive statements, implicit transfer of control also occurs when the normal flow is altered without the execution of a procedure branching statement. COBOL provides the following types of implicit control flow alterations which override the statement-to-statement transfers of control:

If a paragraph is being executed under control of another COBOL statement (for example, PERFORM and USE) and the paragraph is the last paragraph in the range of the controlling statement, then an implied transfer of control occurs from the last statement in the paragraph to the control mechanism of the last executed controlling statement. Further, if a paragraph is being executed under the control of a PERFORM statement which causes iterative execution and that paragraph is the first paragraph in the range of that PERFORM statement, an implicit transfer of control occurs between the control mechanism associated with that PERFORM statement and the first statement in that paragraph for each iterative execution of the paragraph.

When any COBOL statement is executed which results in the execution of a declarative section, an implicit transfer of control to the declarative section occurs. Note that another implicit transfer of control occurs after execution of the declarative.

An explicit transfer of control consists of an alteration of the implicit control transfer mechanism by the execution of a procedure branching or conditional statement. An explicit transfer of control can be caused only by the execution of a procedure branching or conditional statement. The execution of the procedure branching statement ALTER does not in itself constitute an explicit transfer of control, but affects the explicit transfer of control that occurs when the associated GO TO statement is executed.

In this document, the term 'next executable statement' is used to refer to the next COBOL statement to which control is transferred according to the rules above and the rules associated with each language element in the Procedure Division.

There is no next executable statement following:

The last statement in a declarative section when the paragraph in which it appears is not being executed under the control of some other COBOL statement. In COBOL, the result would be an implicit transfer of control to the first nondeclarative statement.

The last statement in a program when the paragraph in which it appears is not being executed under the control of some other COBOL statement. The result would be as if an implicit STOP RUN statement were executed.

Segmentation

SEGMENTATION

COBOL segmentation is a facility that provides a means by which the user may communicate with the compiler to specify object program overlay requirements. COBOL segmentation deals only with segmentation of procedures.

Segments

When segmentation is used, the entire Procedure Division must be in sections. In addition, each section must be classified as belonging either to the fixed portion or to one of the independent segments of the object program as determined by the assignment of segment numbers. All source paragraphs which contain the same segment-numbers can range from 00 through 127, it is possible to subdivide any object program into a maximum of 128 segments. Segmentation in no way affects the need for qualification of procedure-names to insure uniqueness.

Fixed Portion

The fixed portion is defined as that part of the object program which is always in memory. This portion of the program is composed of segments with segment-numbers 0 through 49.

Independent Segments

An independent segment is defined as part of the object program which can overlay, and can be overlaid by, another independent segment. An independent segment has a segment-number 50 through 127.

An independent segment is in its initial state whenever control is transferred (either implicitly or explicitly) to that segment for the first time during the execution of a program.

On subsequent transfers of control to the segment, an independent segment is also in its initial state when:

Control is transferred to that segment as a result of the implicit transfer of control between consecutive statements from a segment with a different segment-number.

Control is transferred explicitly to that segment from a segment with a different segment-number.

On subsequent transfer of control to the segment, an independent segment is in its last-used state when control is transferred implicitly to that segment from a segment with a different segment-number.

Segmentation Classification

Sections which are to be segmented are classified using a system of segment-numbers and the following criteria:

Logic Requirements--Sections which must be available for reference at all times, or which are referred to very frequently, are normally classified as belonging to one of the permanent segments; sections which are used less frequently are normally classified as belonging to one of the independent segments, depending on logic requirements.

Frequency of Use--Generally, the more frequently a section is referred to, the lower its segment-number; the less frequently it is referred to, the higher its segment-number.

Relationship to Other Sections -- Sections which frequently communicate with one another should be given the same segment-numbers.

Segmentation Control

The logical sequence of the program is the same as the physical sequence except for specific transfers of control. Control may be transferred within a source program to any paragraph in a section; that is, it is not mandatory to transfer control to the beginning of a section.

Restrictions on Program Flow

When segmentation is used, the following restrictions are placed on the ALTER and PERFORM statements.

The ALTER STATEMENT

A GO TO statement in a section whose segment-number is greater than or equal to 50 must not be referred to by an ALTER statement in a section with a different segment-number.

The PERFORM STATEMENT

A PERFORM statement that appears in a section that is not in an independent segment can have within its range, in addition to any declarative sections whose execution is caused within that range, only one of the following:

Sections and/or paragraphs wholly contained in one or more fixed segments, or

Sections and/or paragraphs wholly contained in a single independent segment.

A PERFORM statement that appears in an independent segment can have within its range, in addition to any declarative sections whose execution is caused within that range, only one of the following:

Sections and/or paragraphs wholly contained in one or more fixed segments, or

Sections and/or paragraphs wholly contained in the same independent segment as that PERFORM statement.

THE USE STATEMENT

The USE statement specifies procedures for input-output error handling that are in addition to the standard procedures provided by the input-output control system. It is a compiler directing statement required in each declarative section.

FORMAT

```
USE AFTER STANDARD {EXCEPTION}
    {ERROR      }

PROCEDURE ON {file-name-1 [,file-name-2] ...}
    {INPUT          }
    {OUTPUT         }
    {I-O            }
    {EXTEND        }
```

A USE statement, when present, must immediately follow a section header in the declaratives section and must be followed by a period followed by a space. The remainder of the section must consist of zero, one or more procedural paragraphs that define the procedures to be used.

The USE statement itself is never executed; it merely defines the conditions calling for the execution of the USE procedure.

The same file-name can appear in only one USE statement.

The words ERROR and EXCEPTION are synonymous and may be used interchangeably.

The designated procedures can be executed by the input-output system after completing the standard input-output error routine, or upon recognition of the INVALID KEY or AT END conditions, when the INVALID KEY phrase or AT END phrase, respectively, has not been specified in the input-output statement.

After execution of a USE procedure, control is returned to the invoking routine.

USE Statement

Within a USE procedure, there must not be any reference to any nondeclarative procedures. Conversely, in the nondeclarative portion there must be no reference to procedure-names that appear in the declarative portion, except that PERFORM statements may refer to a USE statement or to the procedures associated with such a USE statement.

Within a USE procedure, there must not be the execution of any statement that would cause the execution of a USE procedure that had previously been invoked and had not yet returned control to the invoking routine.

USE Example:

```
PROCEDURE DIVISION.  
DECLARATIVES.  
IO-ERROR SECTION.  
    USE AFTER STANDARD ERROR PROCEDURE ON I-O.  
IO-ERROR-ROUTINE.  
    DISPLAY "INPUT-OUTPUT ERROR OCCURRED".  
    ACCEPT CONTINUE-FLAG POSITION ZERO.  
    IF CONTINUE-FLAG = "NO" STOP RUN.  
END DECLARATIVES.
```

ARITHMETIC STATEMENTS

The arithmetic statements ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT have several common features:

The data descriptions of the operands need not be the same; any necessary conversion and decimal point alignment is supplied throughout the calculation.

Arithmetic operations are calculated in either binary, decimal, packed decimal, or mixed depending on the USAGE of the operands and receiving item according to the following rules:

If the receiving data item of a divide operation is DISPLAY or COMPUTATIONAL, the operation is always calculated in decimal with any necessary conversions.

Intermediate and final results are calculated in binary if all preceding intermediate results are binary and the next operand has COMPUTATIONAL-1 usage (except as noted in previous paragraph). Otherwise, the remaining intermediate and final results are calculated in decimal with any necessary conversions.

The maximum size of each operand is eighteen (18) decimal digits. The composite of operands, which is a hypothetical data item resulting from the super-imposition of specified operands in a statement aligned on their decimal points, must not contain more than eighteen decimal digits.

Arithmetic Expressions

An arithmetic expression can be an identifier of a numeric elementary item, a numeric literal, such identifiers and literals separated by arithmetic operators, two arithmetic expressions separated by an arithmetic operator, or an arithmetic expression enclosed in parentheses. Any arithmetic expression may be preceded by a unary operator. The permissible combinations of variables, numeric literals, arithmetic operator and parentheses are given in Combination of Symbols in Arithmetic Expressions Table.

Those identifiers and literals appearing in an arithmetic expression must represent either numeric elementary items or numeric literals on which arithmetic may be performed.

Arithmetic Statements

Arithmetic Operators

There are four binary arithmetic operators and two unary arithmetic operators that may be used in arithmetic expressions. They are represented by specific characters that must be preceded by a space and followed by a space.

<u>Binary Arithmetic Operators</u>	<u>Meaning</u>
+	Addition
-	Subtraction
*	Multiplication
/	Division

<u>Unary Arithmetic Operators</u>	<u>Meaning</u>
+	The effect of multiplication by numeric literal +1
-	The effect of multiplication by numeric literal -1.

Formation and Evaluation Rules

Parentheses may be used in arithmetic expressions to specify the order in which elements are to be evaluated. Expressions within parentheses are evaluated first, and within nested parentheses, evaluation proceeds from the least inclusive set to the most inclusive set. When parentheses are not used, or parenthesized expressions are at the same level of inclusiveness, the following hierarchical order of execution is implied:

- 1st - Unary plus and minus
- 2nd - Multiplication and division
- 3rd - Addition and subtraction

Parentheses are used either to eliminate ambiguities in logic where consecutive operations of the same hierarchical level appear or to modify the normal hierarchical sequence of execution in expressions where it is necessary to have some deviation from the normal precedence. When the sequence of execution is not specified by parentheses, the order of execution of consecutive operations of the same hierarchical level is from left to right.

The ways in which operators, variables, and parentheses may be combined in an arithmetic expression are summarized in the following table, where:

The letter 'P' indicates a permissible pair of symbols.

The character '-' indicates an invalid pair.

'Variable' indicates an identifier or literal.

FIRST SYMBOL	SECOND SYMBOL				
	Variable	*/--	Unary + or -	()
Variable	-	P	-	-	P
* / + -	P	-	P	P	-
Unary +or-	P	-	-	P	-
(P	-	P	P	-
)	-	P	-	-	P

An arithmetic expression may only begin with the symbol '(', '+', '−', or a variable and may only end with a ')' or a variable. There must be a one-to-one correspondence between left and right parentheses of an arithmetic expression such that each left parenthesis is to the left of its corresponding right parenthesis.

Arithmetic expressions allow the user to combine arithmetic operations without the restrictions on composite of operands and/or receiving data items.

Conditionals

CONDITIONALS

The conditions are relation, class, condition-name, and switch-status. A condition has a truth value of 'true' or 'false'.

Relation Condition

A relation condition causes a comparison of two operands, each of which may be the data item referenced by an identifier or a literal. A relation condition has the truth value of 'true' if the relation exists between the operands.

Comparison of two numeric operands is permitted regardless of the formats specified in their respective USAGE clauses. However, for all other comparisons the operands must have the same usage. If either of the operands is a group item, the nonnumeric comparison rules apply.

The general format of a relation condition is as follows:

```
{identifier-1} {IS [NOT] GREATER THAN}{identifier-2      }
{literal-1    } {IS [NOT] LESS THAN     }{literal-2      }
{index-name-1} {IS [NOT] EQUAL TO      }{index-name-2   }
              {IS [NOT] >          }
              {IS [NOT] <          }
              {IS [NOT] =          }
```

The first operand (identifier-1, literal-1 or index-name-1) is called the subject of the condition; the second operand (identifier-2, literal-2 or index-name-2) is called the object of the condition. The relation condition must contain at least one reference to a variable.

The relational operator specifies the type of comparison to be made in a relation condition. A space must precede and follow each reserved word comprising the relational operator. When used, 'NOT' and the next key word or relation character are one relational operator that defines the comparison to be executed for truth value; e.g., 'NOT EQUAL' is a truth test for an 'unequal' comparison; 'NOT GREATER' is a truth test for an 'equal' or 'less' comparison. The meaning of the relational operators is as follows:

<u>Meaning</u>	<u>Relational Operator</u>
Greater than or not greater than	IS [NOT] <u>GREATER THAN</u> IS [NOT] >
Less than or not less than	IS [NOT] <u>LESS THAN</u> IS [NOT] <
Equal to or not equal to	IS [NOT] <u>EQUAL TO</u> IS [NOT] =

NOTE: The required relational characters '>', '<', and '=' are not underlined to avoid confusion with other symbols such as '≥' (greater than or equal to).

Comparison of Numeric Operands

For operands whose class is numeric a comparison is made with respect to the algebraic value of the operands. The length of the literals or operands, in terms of number of digits represented, is not significant. Zero is considered a unique value regardless of the sign.

Comparison of these operands is permitted regardless of the manner in which their usage is described. Unsigned numeric operands are considered positive for purposes of comparison.

Comparison of Nonnumeric Operands

For nonnumeric operands, or one numeric and one nonnumeric operand, a comparison is made with respect to a specified collating sequence of characters. If one of the operands is specified as numeric, it must be an integer data item or an integer literal and:

If the nonnumeric operand is an elementary data item or a nonnumeric literal, the numeric operand is treated as though it were moved to an elementary alphanumeric data item of the same size as the numeric data item (in terms of standard data format characters), and the contents of this alphanumeric data item were then compared to the nonnumeric operand.

Conditionals

If the nonnumeric operand is a group item, the numeric operand is treated as though it were moved to a group item of the same size as the numeric data item (in terms of standard data format characters), and the contents of this group item were then compared to the nonnumeric operand.

A noninteger numeric operand cannot be compared to a nonnumeric operand.

The size of an operand is the total number of standard data format characters in the operand. Numeric and nonnumeric operands may be compared only when their usage is the same. There are two cases to consider: operands of equal size and operands of unequal size.

Operands of equal size: If the operands are of equal size, comparison effectively proceeds by comparing characters in corresponding character positions starting from the high order end and continuing until either a pair of unequal characters is encountered or the low order end of the operand is reached, whichever comes first. The operands are determined to be equal if all pairs of characters compare equally through the last pair, when the low order end is reached.

The first encountered pair of unequal characters is compared to determine their relative position in the collating sequence. The operand that contains the character that is positioned higher in the collating sequence is considered to be the greater operand.

Operands of unequal size: If the operands are of unequal size, comparison proceeds as though the shorter operand were extended on the right by sufficient spaces to make the operands of equal size.

Comparisons of Index-Names and/or Index Data Items

If two index-names are compared the result is the same as if the corresponding occurrence numbers were compared.

For an index-name and a data item (other than an index data item) or literal, the comparison is made between the occurrence number that corresponds to the value of the index-name and the data item or literal.

When a comparison is made between an index data item and an index-name or another index data item, the actual values are compared without conversion.

The result of the comparison of an index data item with any data item or literal not specified above is undefined.

Class Condition

The class condition determines whether the operand is numeric, that is, consists entirely of the characters '0', '1', '2', '3', ..., '9', with or without the operational sign; or alphabetic, that is, consists entirely of the characters 'A', 'B', 'C', ..., 'Z', space. The general format for the class condition is as follows:

```
identifier IS [NOT] {NUMERIC }
{ALPHABETIC}
```

The usage of the operand being tested must be described as display. When used, 'NOT' and the next key word specify one class condition that defines the class test to be executed for truth value, e.g., 'NOT NUMERIC' is a truth test for determining that an operand is nonnumeric.

The NUMERIC test cannot be used with an item whose data description describes the item as alphabetic or as a group item composed of elementary items whose data description indicates the presence of operational sign(s). If the data description of the item being tested does not indicate the presence of an operational sign, the item being tested is determined to be numeric only if the contents are numeric and an operational sign is not present. If the data description of the item does indicate the presence of an operational sign, the item being tested is determined to be numeric only if the contents are numeric and a valid operational sign is present. Valid operational signs for data items are the standard data format characters, '+' and '-'.

The ALPHABETIC test cannot be used with an item whose data description describes the item as numeric. The item being tested is determined to be alphabetic only if the contents consist of any combination of the alphabetic characters 'A' through 'Z' and the space.

Condition-name (Conditional Variable)

In a condition-name condition, a conditional variable is tested to determine whether or not its value is equal to one of the values associated with a condition-name. The general-format for the condition-name conditioning as follows:

condition-name

If the condition-name is associated with a range of values, then the conditional variable is tested to determine whether or not its value falls in this range, including the end values.

The rules for comparing a conditional variable with a condition-name value are the same as those specified for relation conditions.

The result of the test is true if one of the values corresponding to the condition-name equals the value of its associated conditional variable.

Switch-Status Condition

A switch-status condition determines the 'on' or 'off' status of a software switch. The switch-name and the 'on' or 'off' value associated with the condition must be named in the SPECIAL-NAMES paragraph of the Environment Division. The general format for the switch-status condition is as follows:

condition-name

The result of the test is true if the switch is set to the specified position corresponding to the condition-name.

Complex Conditions

A complex condition is formed by combining simple conditions, combined conditions and/or complex conditions with logical connectors (logical operators 'AND' and 'OR') or negating these conditions with logical negation (the logical operator 'NOT'). The truth value of a complex condition, whether parenthesized or not, is that truth value which results from the interaction of all the stated logical operators on the individual truth values of simple conditions, or the intermediate truth values of conditions logically connected or logically negated. The logical operators and their meanings are:

<u>Logical Operator</u>	<u>Meaning</u>
AND	Logical conjunction; the truth value is 'true' if both of the conjoined conditions are true; 'false' if one or both of the conjoined conditions is false.
OR	Logical inclusive OR; the truth value is 'true' if one or both of the included conditions is true; 'false' if both included conditions are false.
NOT	Logical negation or reversal of truth value; the truth value is 'true' if the condition is false; 'false' if the condition is true.

The logical operators must be preceded by a space and followed by a space.

Negated Simple Conditions

A simple condition is negated through the use of the logical operator 'NOT'. The negated simple condition effects the opposite truth value for a simple condition. Thus the truth value of a negated simple condition is 'true' if and only if the truth value of the simple condition is 'false'; the truth value of a negated simple condition is 'false' if and only if the truth value of the simple condition is 'true'. The inclusion in parentheses of a negated simple condition does not change the truth value.

The general format for a negated simple condition is:

NOT simple-condition

Combined and Negated Combined Conditions

A combined condition results from connecting conditions with one of the logical operators 'AND' or 'OR'. The general format of a combined condition is:

```
condition {{AND} condition} ...
{OR }
```

Conditionals

Where 'condition' may be:

A simple condition, or

A negated simple condition, or

A combined condition, or

A negated combined condition; i.e., the 'NOT' logical operator followed by a combined condition enclosed within parentheses, or

Combinations of the above.

Although parentheses need never be used when either 'AND' or 'OR' (but not both) is used exclusively in a combined condition, parentheses may be used to affect the final truth value when a mixture of 'AND', 'OR' and 'NOT' is used.

Condition Evaluation Rules

Condition Evaluation Rules indicate the ways in which conditions and logical operators may be combined and parenthesized. There must be a one-to-one correspondence between left and right parentheses such that each left parenthesis is to the left of its corresponding right parenthesis.

Parentheses may be used to specify the order in which individual conditions of complex conditions are to be evaluated when it is necessary to depart from the implied evaluation precedence. Conditions within parentheses are evaluated first, and, within nested parentheses, evaluation proceeds from the least inclusive condition to the most inclusive condition. When parentheses are not used, or parenthesized conditions are at the same level of inclusiveness, the following hierarchical order of logical evaluation is implied until the final truth value is determined:

Truth values for simple conditions are established.

Truth values for negated simple conditions are established.

Truth values for combined conditions are established:

'AND' logical operators, followed by
'OR' logical operators.

Truth values for negated combined conditions are established.

When the sequence of evaluation is not completely specified by parentheses, the order of evaluation of consecutive operations of the same hierarchical level is from left to right.

SEQUENTIAL ORGANIZATION INPUT-OUTPUT

The sequential organization input-output statements in the Procedure Division are the CLOSE, OPEN, READ, REWRITE, UNLOCK, USE, and WRITE statements.

Function

Sequential organization input-output provides a capability to access records of a file in established sequence. The sequence is established as a result of writing the records to the file.

Organization

Sequential files are organized such that each record in the file except the first has a unique predecessor record, and each record except the last has a unique successor record. These predecessor-successor relationships are established by the order of WRITE statements when the file is created. Once established, the predecessor-successor relationships do not change except in the case where records are added to the end of the file.

Access Mode

In the sequential access mode, the sequence in which records are accessed is the order in which the records were originally written.

Current Record Pointer

The current record pointer is a conceptual entity used in this document to facilitate specification of the next record to be accessed within a given file. The concept of the current record pointer has no meaning for a file opened in the output mode. The setting of the current record pointer is affected only by the OPEN and READ statements.

I-O Status

If the FILE STATUS clause is specified in a file control entry, a value is placed into the specified two-character data item during the execution of an OPEN, CLOSE, READ, WRITE, or REWRITE statement and before any applicable USE procedure is executed, to indicate to the COBOL program the status of that input-output operation.

Status Key 1

The leftmost character position of the FILE STATUS data item is known as status key 1 and is set to indicate one of the following conditions upon completion of the input-output operation:

'0' - Successful Completion. The input-output statement was successfully executed.

'1' - At End. The sequential READ statement was unsuccessfully executed as a result of an attempt to read a record when no next logical record exists in the file.

'3' - Permanent Error. The input-output statement was unsuccessfully executed as the result of a boundary violation for a sequential file or as the result of an input-output error, such as data check parity error, or transmission error.

'9' - General Error. The input-output statement was unsuccessfully executed as a result of a condition that is specified by the value of status key 2.

Status Key 2

The rightmost character position of the FILE STATUS data item is known as status key 2 and is used to further describe the results of the input-output operation. This character will contain a value as follows:

If no further information is available concerning the input-output operation, then status key 2 contains a value of '0'.

When status key 1 contains a value of '3' indicating a permanent error condition, status key 2 may contain a value of '4' indicating a boundary violation. This condition indicates that an attempt has been made to write beyond the externally defined boundaries of a sequential file.

When status key 1 contains a value a '9' indicating an operating system error condition, the value of status key 2 may contain a:

'0' indicating an invalid operation. This condition indicates that an attempt has been made to execute a READ, WRITE, or REWRITE statement that conflicts with the current open mode or a REWRITE statement not preceded by a successful READ statement.

'1' indicating file not opened. This condition indicates that an attempt has been made to execute a DELETE, START, UNLOCK, READ, WRITE, REWRITE or CLOSE statement on a file which is not currently open.

'2' indicating file not closed. This condition indicates that an attempt has been made to execute an OPEN statement on a file which is currently open.

'3' indicating file not available. This condition indicates that an attempt has been made to execute an OPEN statement for a file closed WITH LOCK or to OPEN a file LOCKed by another user.

'4' indicating an invalid open. This condition indicates that an attempt has been made to execute an OPEN statement for a file with no external correspondence or a file having inconsistent parameters.

'5' indicating invalid device or not next reel. This condition indicates that an attempt has been made to open a file having parameters (e.g., open mode or organization) which conflict with the device assignment (RANDOM, INPUT, PRINT, ...) or that an attempt has been made to execute a CLOSE REEL statement for the last reel/unit of a multi-reel file. In the case of a CLOSE REEL, the file has been closed.

'6' indicating an undefined current record pointer status. This condition indicates that an attempt has been made to execute a READ statement after occurrence of an unsuccessful READ statement without an intervening successful CLOSE and OPEN.

'7' indicating an invalid record length. This condition indicates an attempt has been made to open a file that was defined with a maximum record length different from the externally defined maximum record length, or to execute a WRITE statement that specifies a record with a length smaller than the minimum or larger than the maximum record size, or a REWRITE statement when the new record length is different from that of the record to be rewritten.

Sequential Organization Input-Output

```
| '9' indicating an attempt has been made to READ a record  
| which is locked. This error is returned only if an  
| applicable USE procedure and a FILE STATUS data item are  
| declared for the file. Otherwise the read statement is  
| retried until the record is unlocked.
```


RELATIVE ORGANIZATION INPUT-OUTPUT

The Relative input-output statements in the Procedure Division are the CLOSE, DELETE, OPEN, READ, REWRITE, START, UNLOCK and WRITE statements.

Function

Relative input-output provides a capability to access records of a mass storage file in either a random or sequential manner. Each record in a relative file is uniquely identified by an integer value greater than zero which specifies the record's logical position in the file.

Organization

Relative file organization is permitted only on mass storage devices (RANDOM device).

A relative file consists of records which are identified by relative record numbers. The file may be thought of as composed of a serial string of areas, each capable of holding a logical record. Each of these areas is denominated by a relative record number, an integer value greater than zero. Records are stored and retrieved based on this number. For example, the tenth record is the one addressed by relative record number 10 and is the tenth record area, whether or not records have been written in the first through the ninth record areas.

Access Modes

In the sequential access mode, the sequence in which records are accessed is the ascending order of the relative record numbers of all records which currently exist within the file.

In the random access mode, the sequence in which records are accessed is controlled by the programmer. The desired record is accessed by placing its relative record number in a relative key data item.

In the dynamic access mode, the programmer may change at will from sequential access to random access using appropriate forms of input-output statements.

Current Record Pointer

The current record pointer is a conceptual entity used in this document to facilitate specification of the next record to be accessed within a given file. The concept of the current record pointer has no meaning for a file opened in the output mode. The setting of the current record pointer is affected only by the OPEN, READ, and START statements.

I-O Status

If the FILE STATUS clause is specified in a file control entry, a value is placed into the specified two-character data item during the execution of an OPEN, CLOSE, READ, WRITE, REWRITE, DELETE, or START statement and before any applicable USE procedure is executed, to indicate to the COBOL program the status of that input-output operation:

Status Key 1

The leftmost character position of the FILE STATUS data item is known as status key 1 and is set to indicate one of the following conditions upon completion of the input-output operation:

'0' - Successful Completion. The input-output was successfully executed.

'1' - At End. The statement was unsuccessfully executed as a result of an attempt to read a record when no next logical record exists in the file.

'2' - Invalid Key. The input-output statement was unsuccessfully executed as a result of one of the following:

Duplicate Key
No Record Found
Boundary Violation

'3' - Permanent Error. The input-output statement was unsuccessfully executed as the result of an input-output error, such as data check, parity error, or transmission error.

'9' - General Error. The input-output statement was unsuccessfully executed as a result of a condition that is specified by the value of status key 2.

Status Key 2

The rightmost character position of the FILE STATUS data item is known as status key 2 and is used to further describe the results of the input-output operation. This character will contain a value as follows:

If no further information is available concerning the input-output operation, then status key 2 contains a value of '0'.

When status key 1 contains a value of '2' indicating an INVALID KEY condition, status key 2 is:

'2' indicating a duplicate key value. An attempt has been made to write a record that would create a duplicate key.

'3' indicating no record found. An attempt has been made to access a record, identified by a key, and that record does not exist in the file.

'4' indicating a boundary violation. An attempt has been made to write beyond the externally-defined boundaries of a file.

When status key 1 contains a value of '9' indicating an operating system error condition, the value of status key 2 is:

'0' indicating invalid operation. An attempt has been made to execute a DELETE, READ, REWRITE, START, or WRITE statement which conflicts with the current open mode of the file or a sequential access DELETE or REWRITE statement not preceded by a successful READ statement.

'1' indicating file not opened. This condition indicates that an attempt has been made to execute a DELETE, START, UNLOCK, READ, WRITE, REWRITE, or CLOSE statement on a file which is not currently open.

'2' indicating file not closed. An attempt has been made to execute an OPEN statement on a file that is currently open.

'3' indicating file not available. An attempt has been made to execute an OPEN statement for a file closed WITH LOCK or to OPEN a file LOCKed by another user.

Relative Organization Input-Output

'4' indicating invalid OPEN. An attempt has been made to execute an OPEN statement for a file with no external correspondence or a file having inconsistent parameters.

'5' indicating invalid device. This condition indicates that an attempt has been made to open a file having parameters (e.g., open mode or organization) which conflict with the device assignment (RANDOM, INPUT, PRINT,...).

'6' indicating an undefined current record pointer status. This condition indicates that an attempt has been made to execute a sequential READ statement after the occurrence of an unsuccessful READ or START statement without an intervening successful CLOSE and OPEN.

'7' indicating an invalid record length. This condition indicates that an attempt has been made to OPEN a file that was defined with a maximum record length different from the externally defined maximum record length, or to execute a WRITE statement that specifies a record with a length smaller than the minimum or larger than the maximum record size, or a REWRITE statement when the new record length is different from that of the record to be rewritten.

'9' indicating an attempt has been made to READ a record which is locked. This error is returned only if an applicable USE procedure and a FILE STATUS data item are declared for the file. Otherwise the read statement is retried until the record is unlocked.

The INVALID KEY Condition

The INVALID KEY condition can occur as a result of the execution of a START, READ, WRITE, REWRITE, or DELETE statement.

When the INVALID KEY condition is recognized, the System takes these actions in the following order:

A value is placed into the FILE STATUS data item, if specified for this file, to indicate a INVALID KEY condition.

If the INVALID KEY phrase is specified in the statement causing the condition, control is transferred to the INVALID KEY imperative statement. Any USE procedure specified for this file is not executed.

If the INVALID KEY phrase is not specified, but a USE procedure is specified, either explicitly or implicitly, for this file, that procedure is executed.

When the INVALID KEY condition occurs, execution of the input-output statement which recognized the condition is unsuccessful and the file is not affected.

The AT END Condition

The AT END condition can occur as a result of the execution of a READ statement. When the AT END condition occurs, execution of the READ statement is unsuccessful.

INDEXED ORGANIZATION INPUT-OUTPUT

Indexed input-output statements in the Procedure Division are the CLOSE, DELETE, OPEN, READ, REWRITE, START, UNLOCK and WRITE statements.

Function

Indexed input-output provides a capability to access records of a mass storage file in either a random or sequential manner. Each record in a nonsequential organization file is uniquely identified by a key.

Organization

A file whose organization is indexed is a mass storage file in which data records may be accessed by the value of a key. A record description may include one or more key data items, each of which is associated with an index. Each index provides a logical path to the data records according to the contents of a data item within each record which is the recorded key for that index.

The data item named in the RECORD KEY clause of the file control entry for a file is the prime record key for that file. For purposes of inserting, updating and deleting records in a file, each record is identified solely by the value of its prime record key. This value must, therefore, be unique and must not be changed when updating the record.

Access Modes

In the sequential access mode, the sequence in which records are accessed is the ascending order of the keys of all records which currently exist within the file.

In the random access mode, the sequence in which records are accessed is controlled by the programmer. For indexed files, the desired record is accessed by placing the value of its record key in a record key data item.

In the dynamic access mode, the programmer may change at will from sequential access to random access using appropriate forms of input-output statements.

Current Record Pointer

The current record pointer is a conceptual entity used in this document to facilitate specification of the next record to be accessed within a given file. The concept of the current record pointer has no meaning for a file opened in the output mode. The setting of the current record pointer is affected only by the OPEN, READ, and START statements.

I-O Status

If the FILE STATUS clause is specified in a file control entry, a value is placed into the specified two-character data item during the execution of an OPEN, CLOSE, READ, WRITE, REWRITE, DELETE, or START statement and before any applicable USE procedure is executed, to indicate to the COBOL program the status of that input-output operation:

Status Key 1

The leftmost character position of the FILE STATUS data item is known as status key 1 and is set to indicate one of the following conditions upon completion of the input-output operation:

'0' - Successful Completion. The input-output was successfully executed.

'1' - At End. The Format 1 READ statement was unsuccessfully executed as a result of an attempt to read a record when no next logical record exists in the file.

'2' - Invalid Key. The input-output statement was unsuccessfully executed as a result of one of the following:

- Sequence Error
- Duplicate Key
- No Record Found
- Boundary Violation

'3' - Permanent Error. The input-output statement was unsuccessfully executed as the result of an input-output error, such as data check, parity error, or transmission error.

'9' - General Error. The input-output statement was unsuccessfully executed as a result of a condition that is specified by the value of status key 2.

Status Key 2

The rightmost character position of the FILE STATUS data item is known as status key 2 and is used to further describe the results of the input-output operation. This character will contain a value as follows:

If no further information is available concerning the input-output operation, then status key 2 contains a value of '0'.

When status key 1 contains a value of 0, indicating a successful completion, status key 2 may contain a value of 2, indicating a duplicate key. This condition indicates:

For a READ statement, the key value for the current key of reference is equal to the value of that same key in the next record within the current key of reference.

For a WRITE or REWRITE statement, the record just written created a duplicate key value for at least one alternate record key for which duplicates are allowed.

When status key 1 contains a value of '2' indicating an INVALID KEY condition, status key 2 is:

'1' indicating a sequence error for a sequentially accessed indexed file. The ascending sequence requirement of successive record key values has been violated or the record key value has been changed by the COBOL program between the successful execution of a READ statement and the execution of the next REWRITE statement for that file.

'2' indicating a duplicate key value. An attempt has been made to write a record that would create a duplicate key.

'3' indicating no record found. An attempt has been made to access a record, identified by a key, and that record does not exist in the file.

'4' indicating a boundary violation. An attempt has been made to write beyond the externally-defined boundaries of a file.

When status key 1 contains a value of '9' indicating an operating system error condition, the value of status key 2 is:

'0' indicating invalid operation. An attempt has been made to execute a DELETE, READ, REWRITE, START, or WRITE statement which conflicts with the current open mode of the file or a sequential access DELETE or REWRITE statement not preceded by a successful READ statement.

'1' indicating file not opened. This condition indicates an attempt has been made to execute a delete, start, unlock, read, write, rewrite, or close statement on a file that is not currently open.

'2' indicating file not closed. An attempt has been made to execute an OPEN statement on a file that is currently open.

'3' indicating file not available. An attempt has been made to execute an OPEN statement for a file closed with LOCK or to OPEN a file LOCKed by another user.

'4' indicating invalid open. An attempt has been made to execute an OPEN statement for a file with no external correspondence or a file having inconsistent parameters.

'5' indicating invalid device. This condition indicates that an attempt has been made to open a file having parameters (e.g., open mode or organization which conflict with the device assignment (RANDOM, INPUT, PRINT, ...)).

'6' indicating an undefined current record pointer status. This condition indicates that an attempt has been made to execute a sequential READ statement after the occurrence of an unsuccessful READ or START statement without an intervening successful CLOSE and OPEN.

'7' indicating an invalid record length. This condition indicates that an attempt has been made to open a file that was defined with a maximum record length different from the externally defined maximum record length, or to execute a write statement that specifies a record with a length smaller than the minimum or larger than the maximum record size, or a REWRITE statement when the new record length is different from that of the record to be rewritten.

'8' indicating an invalid indexed file. This condition indicates that the indexed file contains inconsistent data. This is a catastrophic error from which there is no recovery at the present time.

| '9' indicating an attempt has been made to READ a record
| which is locked. This error is returned only if an applicable USE procedure and a FILE STATUS data item are declared for the file. Otherwise the read statement is retried until the record is unlocked.

The INVALID KEY Condition

The INVALID KEY condition can occur as a result of the execution of a START, READ, WRITE, REWRITE, or DELETE statement.

When the INVALID KEY condition is recognized, the System takes these actions in the following order:

A value is placed into the FILE STAUS data item, if specified for this file, to indicate an INVALID KEY condition.

If the INVALID KEY phrase is specified in the statement causing the condition, control is transferred to the INVALID KEY imperative statement. Any USE procedure specified for this file is not executed.

If the INVALID KEY phrase is not specified, but a USE procedure is specified, either explicitly or implicitly, for this file, that procedure is executed.

When the INVALID KEY condition occurs, execution of the input-output statement which recognized the condition is unsuccessful and the file is not affected.

The AT END Condition

The AT END condition can occur as a result of the execution of a READ statement. When the AT END condition occurs, execution of the READ statement is unsuccessful.

PROCEDURAL STATEMENTSThe ACCEPT ... FROM Statement

The ACCEPT statement causes the information requested to be transferred to the data item specified by identifier-1 according to the rules of the MOVE statement. DATE, DAY, and TIME are conceptual data items and, therefore, are not described in the COBOL program.

FORMAT

```
ACCEPT identifier-1 FROM {DATE}
          {DAY }
          {TIME}
```

DATE is composed of the data elements year of century, month of year, and day of month. The sequence of the data element codes is from high order to low order (left to right), year of century, month of year, and day of month. Therefore, July 1, 1979 would be expressed as 790701. DATE, when accessed by a COBOL program behaves as if it had been described in the COBOL program as an unsigned elementary numeric integer data item six digits in length.

DAY is composed of the data elements year of century and day of year. The sequence of the data element codes is from high order to low order (left to right) year of century, day of year. Therefore, July 1, 1979 would be expressed as 79181. DAY, when accessed by a COBOL program as an unsigned elementary numeric integer data item five digits in length.

TIME is composed of the data elements hours, minutes, seconds and hundredths of a second. TIME is based on elapsed time after midnight on a 24-hour clock basis--thus, 2:41 p.m. would be expressed 14410000. TIME, when accessed by a COBOL program behaves as if it had been described in a COBOL program as an unsigned elementary numeric integer data item eight digits in length. The minimum value of TIME is 00000000; the maximum value of TIME is 23595999.

ACCEPT ... FROM Examples

```
ACCEPT YEAR-DAY FROM DAY.
ACCEPT CLOCK FROM TIME.
```

The ACCEPT Statement (Terminal I-O)

The ACCEPT statement causes low volume data to be accepted from the CRT terminal and transferred to the specified data item. ACCEPT statement phrases allow the specification of position, form and format of the accepted data.

FORMAT

```
ACCEPT {identifier-1 [,UNIT {identifier-2}]
           {literal-1}
           [,LINE {identifier-3}] [,POSITION {identifier-4}]
           {literal-2} {literal-3}
           [,SIZE {identifier-5}] [,PROMPT [literal-5]]
           {literal-4}
           [,ECHO [,CONVERT] [,TAB] [,ERASE] [,NO BEEP]
           [,OFF] [,HIGH] [,BLINK] [,REVERSE]] ...
           [,LOW]
           [,ON EXCEPTION identifier-6 imperative-statement]
```

The ACCEPT statement causes the transfer of data from the CRT device.. This data replaces the contents of the data item named by identifier-1. The receiving data time must have usage DISPLAY if ECHO is specified; otherwise, it may have any usage except INDEX. The receiving field will ACCEPT data as though a group move is being performed, thus an ACCEPT into an edited field will result in the omission of editing.

The size of a field ACCEPTed is limited by the rightmost column position of the CRT. At execution, a field that would be split between two lines will be truncated at the end of the first line.

When an ACCEPT statement contains more than one operand, the values are transferred in the sequence in which the operands are encountered. ACCEPT phrases apply to the previously specified identifier-1 only. A subsequent identifier-1 in the same ACCEPT statement will be treated as if no previous phrases have been specified.

An ACCEPT statement may contain no more than one ON EXCEPTION phrase, and if present it must be associated with the last (or only) identifier-1.

NOTE: Features which require support of the host operating system and/or terminal hardware may not be supported on all systems. Any features which are not supported will compile correctly, but will be ignored at runtime. See the User's Guide for specific details.

The UNIT Phrase

The UNIT phrase must be the first phrase if used. The other phrases may be written in any order.

The value of identifier-2 or literal-1 in the UNIT phrase specifies the station identifier of the CRT form which the data is to be accepted. If the UNIT phrase is omitted, the CRT which executed the program will be accessed.

The LINE Phrase

The value of identifier-3 or literal-2 in the LINE phrase specifies the line number from which the data is to be accepted from the screen of the CRT terminal, with 1 being the top line. If the value is greater than the number of lines on the CRT screen, it is adjusted to the bottom line after the screen has been scrolled to the maximum line number +1.

If the value is zero or the LINE phrase is not present in an ACCEPT statement, then data is to be accepted from the next line below the current position of the cursor on the CRT screen unless the value specified in the POSITION phrase is also zero, in which case the data is to be accepted from the line at the current position of the cursor on the CRT screen.

The POSITION Phrase

The value of identifier-4 or literal-3 in the POSITION phrase specifies the number of the character positions to which the cursor is to be positioned within the specified line prior to the accepting of data from the CRT terminal, with 1 being the leftmost character position within a line. If the value is greater than the maximum number of characters within a line on the CRT screen, it is adjusted to the next row and decremented by the maximum number of characters within a line.

If the POSITION phrase is not specified, a value of 1 is assumed for the first accepted operand and 0 for each additional operand accepted in the same statement. If a value of 0 is specified, the data is to be accepted starting at the next field on the CRT screen (starting character position plus size of last ACCEPT or DISPLAY).

The SIZE Phrase

The value of identifier-5 or literal-4 in the SIZE phrase specifies the maximum number of characters to be accepted from the CRT terminal, overriding the Data Division definition of the field. If the SIZE phrase is not present or a value of 0 is specified, then the size of identifier-1, (identifier-5, ...) is used. A size greater than 80 is treated as equal to 80.

The size of the accepted field is determined by the SIZE phrase. The number of characters transferred from the CRT is less than or equal to the size of the accepted field. Input is terminated by depression of the return key (which is not considered part of the input). The number of characters actually input is the size of the source in the following:

If the receiving item is not numeric, the accepted input is stored according to the rules of the MOVE statement for an alphanumeric source and destination. If the receiving item is described JUSTIFIED RIGHT, the clause will apply to the MOVE rules.

If the receiving item is numeric, the accepted input is stored according to the rules of the MOVE statement for a numeric source and destination. If the CONVERT phrase is not specified, the source has the same scale as the receiving item. If the receiving item has a trailing sign and the CONVERT phrase is not specified, the input must contain digits followed by a sign character. If the CONVERT phrase is specified, then the input is converted according to the rules of the CONVERT phrase. The CONVERT phrase is recommended when accepting numeric items.

The PROMPT Phrase

The presence of the key word PROMPT in an ACCEPT statement causes the data to be accepted with prompting. The action of prompting is to display fill characters on the CRT screen in the positions from which data is to be accepted. Literal-5 must be a single character nonnumeric literal which specifies the fill character to be used in prompting. If literal-5 is omitted in the PROMPT phrase, then an underscore will be used as the fill character.

When the PROMPT phrase is not specified, then the data is to be accepted without prompting; the original contents of the field on the CRT will be undisturbed before accepting input.

The ECHO Phrase

The presence of the key word ECHO within an ACCEPT statement causes the contents of identifier-1 to be displayed on the screen of the CRT terminal. Conversion (see CONVERT Phrase), decimal alignment, and justification are performed prior to display. If the specified size is greater than the size of the receiving data-item, the data-item is displayed right justified in the accept field with leading blanks. If the specified size is less than the size of the receiving data-item, the display is truncated on the right. When the ECHO phrase is not specified, the original input data remains in the accept field.

The CONVERT Phrase

If the receiving data-item is numeric, the presence of the key word CONVERT within an ACCEPT statement causes the conversion of an accepted field to a trailing-signed decimal field. The trailing-sign decimal field is then stored in identifier-1. The conversion is accomplished by a left-to-right scan and the rules:

Set the sign according to the rightmost sign given in the input or positive if no sign is present.

Set the scale according to the rightmost period given in the input or to zero if no period is present. If the DECIMAL POINT IS COMMA clause was specified in the source program, a comma replaces the period in determining the scale.

Delete all nonnumeric characters from the accepted field.

When the CONVERT phrase is not specified, or the receiving data-item is not numeric, then the data is to be stored without the above conversion.

The TAB Phrase

The presence of the key word TAB in an ACCEPT statement causes a wait for a return, backspace, or field delete key (or other terminator in conjunction with an EXCEPTION) in reaching the end of the input field. If the key word TAB is omitted, input will automatically be terminated if the end of the input field is encountered.

The ERASE Phrase

The presence of the key word ERASE within an ACCEPT statement causes the screen of the CRT to be erased prior to cursor positioning. When the ERASE phrase is not specified, then the screen is not erased prior to cursor positioning.

The NO BEEP Phrase

The presence of the key words NO BEEP in an ACCEPT statement causes suppression of the beep signal upon cursor positioning. If the key words NO BEEP are omitted, a beep signal will occur upon cursor positioning prior to data input.

The OFF Phrase

The presence of the key word OFF within an ACCEPT statement causes data to be input from the terminal keyboard but not displayed to the screen. Blank characters are displayed to the screen in lieu of data characters.

The HIGH/LOW Phrase

The presence of the key word HIGH or LOW causes the PROMPT character and the accepted data (if CONVERT and/or ECHO was specified) to be displayed at the specified intensity.

When HIGH or LOW is not specified, the default display is HIGH.

The BLINK Phrase

The presence of the key word BLINK causes the PROMPT character, and any displayed data, to be BLINKed. When BLINK is not specified, no BLINK is provided.

The REVERSE Phrase

The presence of the key word REVERSE causes the PROMPT character, and any displayed data, to be displayed in a reverse image mode. When REVERSE is not specified, normal display is provided.

The ON EXCEPTION Phrase

The presence of ON EXCEPTION causes the imperative-statement to be executed if an invalid character is entered. The invalid character (in ASCII format) will be placed in identifier-6 prior to execution of the imperative-statement. The invalid character may be determined by declaring identifier-6 as USAGE COMP-1 and testing for its ASCII value.

When ON EXCEPTION and CONVERT are both specified and a conversion error occurs, an error code of "98" is returned in identifier-6.

ACCEPT Examples

ACCEPT ANSWER-1, ANSWER-2.

ACCEPT START-VALUE LINE 1, POSITION K,
PROMPT, ECHO, CONVERT.

ACCEPT NEXT-N POSITION 0,
PROMPT, ECHO.

ACCEPT YEAR, LINE YR-LN, POSITION YR-POS;
MONTH, LINE MN-LN, POSITION MN-POS.

The ADD Statement

The ADD statement causes two or more numeric operands to be summed and the result to be stored.

FORMAT 1

```
ADD {identifier-1} [,identifier-2] ...
{literal-1 } [,literal-2 ]
TO identifier-m [ROUNDED]
[;ON SIZE ERROR imperative-statement]
```

FORMAT 2

```
ADD {identifier-1}, {identifier-2} [,identifier-3] ...
{literal-1 } {literal-2 } [,literal-3 ]
GIVING identifier-m [ROUNDED]
[;ON SIZE ERROR imperative-statement]
```

FORMAT 3

```
ADD {CORRESPONDING} identifier-1 TO identifier-2 [ROUNDED]
{CORR }
[; ON SIZE ERROR imperative-statement]
```

In Format 1, the values of the operands preceding the word TO are added together, then the sum is added to the current value of identifier-m storing the result immediately into identifier-m.

In Format 2, the values of the operands preceding the word GIVING are added together, then the sum is stored as the new value of identifier-m.

In Formats 1 and 2, each identifier must refer to an elementary numeric item, except that in Format 2 identifier-m following the word GIVING must refer to either an elementary numeric item or an elementary numeric edited item.

In Format 3, data items in identifier-1 are added to and stored in the corresponding data items in identifier-2.

In Format 3, each identifier must refer to a group item.

Each literal must be a numeric literal.

The ROUNDED Phrase

The ADD statement may optionally include the ROUNDED phrase.

If, after decimal point alignment, the number of places in the fraction of the result of the arithmetic operation is greater than the number of places provided for the fraction of the resultant-identifier, truncation is relative to the size provided for the resultant-identifier. When rounding is requested, the absolute value of the resultant-identifier is increased by one (1) whenever the most significant digit of the excess is greater than or equal to five (5).

When the low-order integer positions in a resultant identifier are represented by the character 'P' in the picture for that resultant-identifier, rounding or truncation occurs relative to the rightmost integer position for which storage is allocated.

The SIZE ERROR Phrase

If, after appropriate decimal point alignment, the absolute value of the result exceeds the largest value that can be contained in the associated resultant-identifier, a size error condition exists. If the ROUNDED phrase is specified, rounding takes place before checking for size error.

If the CORRESPONDING phrase is specified, and any of the individual additions produces a size error condition, the imperative-statement is not executed until all of the individual additions are completed.

If the resultant-identifier has COMPUTATIONAL-3 usage, size error is correctly detected only for data items declared with an odd length picture clause. Therefore all COMP-3 data items should be declared with an odd number of character positions.

If the SIZE ERROR phrase is not specified and a size error condition exists, the value of the resultant-identifier is undefined.

If the SIZE ERROR phrase is specified and a size error condition exists, the value of the resultant-identifier is not altered and the imperative statement of the SIZE ERROR phrase is executed.

The CORRESPONDING Phrase

If the CORRESPONDING phrase is used, selected items within identifier-1 are ADDED to, and the result stored in, the corresponding items in identifier-2.

Data items referenced by the CORRESPONDING phrase must adhere to the following rules:

A data item in identifier-1 and a data item in identifier-2 must not be designated by the key word FILLER and must not have the same data-name and the same qualifiers up to, but not including, identifiers-1 and identifier-2.

Both of the data items must be elementary numeric data items.

The description of identifier-1 and identifier-2 must not contain level-number 66, 77, or 88 or the USAGE IS INDEX clause.

A data item that is subordinate to identifier-1 or identifier-2 and contains a REDEFINES, RENAMES, OCCURS or USAGE IS INDEX clause is ignored, as well as those data items subordinate to the data item that contains the REDEFINES, OCCURS, or USAGE IS INDEX clause. However, identifier-1 and identifier-2 may have REDEFINES or OCCURS clauses or be subordinate to data items with REDEFINES or OCCURS clauses.

CORR is an abbreviation for CORRESPONDING.

ADD Examples

ADD SALARY TO SALARY.
(doubles the value of SALARY)

ADD JOHNS-PAY, PAULS-PAY, ALBERTS-PAY
GIVING COMPANY-PAY.

ADD ACCELERATION TO VELOCITY ROUNDED
ON SIZE ERROR GO TO SOUND-BARRIER.

ADD CORRESPONDING ELEMENT (X)
TO ELEMENT (Y).

ADD CORR SUB-TOTAL-RECORD TO TOTAL-RECORD ROUNDED
ON SIZE ERROR GO TO ERR.

The ALTER Statement

The ALTER statement modifies a predetermined sequence of operations.

FORMAT

```
ALTER procedure-name-1 TO [PROCEED TO] procedure-name-2  
[,procedure-name-3 TO [PROCEED TO] procedure-name-4]...
```

Each procedure-name-1, procedure-name-3, ..., is the name of a paragraph that contains a single sentence consisting of a GO TO statement without the DEPENDING phrase.

Each procedure-name-2, procedure-name-4, ..., is the name of a paragraph or section in the Procedure Division.

Execution of the ALTER statement modifies the GO TO statement in the paragraph named procedure-name-1, procedure-name-3,..., so that subsequent executions of the modified GO TO statements cause transfer of control to procedure-name-2, procedure-name-4,..., respectively. Modified GO TO statements in independent segments may, under some circumstances, be returned to their initial states.

A GO TO statement in a section whose segment-number is greater than or equal to 50 must not be referred to by an ALTER statement in a section with a different segment-number.

The CALL Statement

The CALL statement causes control to be transferred from one object program to another, within the run unit.

FORMAT

```
CALL {identifier-1} [USING data-name-1 [,data-name-2] ...]  
      {literal-1    }
```

The execution of a CALL statement causes control to pass to the program whose name is specified by the value of literal-1 or identifier-1, the 'called' program.

Literal-1 must be a nonnumeric literal.

Identifier-1 must be defined as an alphanumeric data item such that its value can be a program name.

The called program can be another COBOL program or an assembly language program. Refer to the User's Guide for specific details.

Called programs may contain CALL statements. However, a called program must not contain a CALL statement that directly or indirectly calls the calling program.

The CALL statement may appear anywhere within a segmented program. When a CALL statement appears in a section with a segment-number greater than or equal to 50, the EXIT PROGRAM statement returns control to the calling program.

The USING Phrase

The data-names specified by the USING phrase of the CALL statement indicate those data items available to a calling program that may be referred to in the called program. The order of appearance of the data-names in the USING phrase of the CALL statement and the USING phrase in the Procedure Division header is critical. Corresponding data-names refer to a single set of data which is available to the called and calling program. The correspondence is positional, not by name. In the case of index-names, no such correspondence is established. Index-names in the called and calling program always refer to separate indices.

CALL

The USING phrase is included in the CALL statement only if there is a USING phrase in the Procedure Division header of the called program, and the number of operands in each USING phrase must be identical.

Each of the operands in the USING phrase must have been defined as a data item in the File Section, Working-Storage Section, or Linkage Section, and must have a level-number of 01 or 77. Data-name-1, data-name-2, ..., may be qualified when they reference data items defined in the File Section.

CALL Examples:

```
CALL "SUBPRG1".
```

```
CALL REORDER  
      USING TABLE, INDEX-1, RESULT.
```

The CLOSE Statement (Sequential I-O)

The CLOSE statement terminates the processing of files.

FORMAT

```
CLOSE file-name-1 [{REEL} [WITH NO REWIND] ]
                  {UNIT}
                  [WITH {NO REWIND}      ]
                  {LOCK        }
[,file-name-2 [{REEL} [WITH NO REWIND] ] ] ...
                  {UNIT}
                  [WITH {NO REWIND}      ]
                  {LOCK        }]
```

The function of a CLOSE statement (with no options) is to cause the operating system to close the file. For files opened for OUTPUT, the operating system also writes an EOF as it closes the file.

If a STOP RUN statement is executed prior to closing the file, the operating system will close the file without an EOF.

A CLOSE statement may only be executed for a file in an open mode.

Once a CLOSE statement has been executed for a file, no other statement can be executed that references that file, either explicitly or implicitly, unless an intervening OPEN statement for that file is executed.

The execution of a CLOSE statement causes the value of the FILE STATUS data-item, if any, associated with file-name-1 (file-name-2, ...) to be updated.

The REEL and UNIT Phrases

The CLOSE REEL and CLOSE UNIT statements are documentary only and may be included or omitted at the user's discretion.

CLOSE (Sequential I-O)

The NO REWIND Phrase

CLOSE WITH NO REWIND prevents page advancing on files assigned to the printer. It has no effect on other files.

The LOCK Phrase

The function of the CLOSE WITH LOCK statement is to perform the CLOSE function and set a flag to prevent the file from being OPENed again during execution of this program.

CLOSE Examples

CLOSE TRANSACTION-FILE.

CLOSE DATA-BASE WITH LOCK.

CLOSE PRINT-FILE WITH NO REWIND.

The CLOSE Statement (Relative and Indexed I-O)

The CLOSE statement terminates the processing of files.

FORMAT

```
CLOSE file-name-1 [WITH LOCK]  
[,file-name-2 [WITH LOCK]] ...
```

The function of a CLOSE statement (with no options) is to cause the operating system to close the file. For files opened for OUTPUT, the operating system also writes an EOF prior to closing the file.

If a STOP RUN statement is executed prior to closing the file, the operating system will close the file without an EOF.

The files referenced in the CLOSE statement need not all have the same organization or access.

A CLOSE statement may only be executed for a file in an open mode.

If a CLOSE statement has been executed for a file, no other statement can be executed that references that file, either explicitly or implicitly, unless an intervening OPEN statement for that file is executed.

The execution of the CLOSE statement causes the value of the specified FILE STATUS data item, if any, associated with file-name-1 (file-name-2, ...) to be updated.

The LOCK Phrase

The function of the CLOSE WITH LOCK statement is to perform the CLOSE function and set a flag to prevent the file from being OPENed during the execution of the program.

CLOSE Examples:

CLOSE TRANSACTION-FILE.

CLOSE DATA-BASE WITH LOCK.

The COMPUTE Statement

The COMPUTE statement assigns the value of an arithmetic expression to a data item.

FORMAT

```
COMPUTE identifier-1 [ROUNDED] = arithmetic-expression
[; ON SIZE ERROR imperative-statement]
```

Identifier-1 must refer to either an elementary numeric item or an elementary numeric edited item.

An arithmetic expression consisting of a single identifier or literal provides a method of setting the value of identifier-1 equal to the value of the single identifier or literal.

The COMPUTE statement allows the user to combine arithmetic operations without the restrictions on composite operands and/or receiving data items imposed by the arithmetic statements ADD, SUBTRACT, MULTIPLY and DIVIDE.

Note: Exponentiation is not supported.

The ROUNDED Phrase

The COMPUTE statement may optionally include the ROUNDED phrase. If, after decimal point alignment, the number of places in the fraction of the result of an arithmetic operation is greater than the number of places provided for the fraction of the identifier-1, truncation is relative to the size provided for the identifier-1. When rounding is requested, the absolute value of the resultant-identifier is increased by one (1) whenever the most significant digit of the excess is greater than or equal to five (5).

When the low-order integer positions in an identifier-1 are represented by the character 'P' in the picture for that identifier, rounding or truncation occurs relative to the rightmost integer position for which storage is allocated.

The SIZE ERROR Phrase

If, after appropriate decimal point alignment, the absolute value of the result exceeds the largest value that can be contained in identifier-1, a size error condition exists. If the ROUNDED phrase is specified, rounding takes place before checking for size error.

If identifier-1 has COMPUTATIONAL-3 usage, size error is detected only for data items declared with an odd length picture clause. Therefore all COMP-3 data items should be declared with an odd number of character positions.

Division by zero always causes a size error condition.

If the SIZE ERROR phrase is not specified and a size error condition exists, the value of the identifier-1 is undefined.

If the SIZE ERROR phrase is specified and a size error condition exists, the value identifier-1 is not altered and the imperative-statement in the SIZE ERROR phrase is executed.

COMPUTE Examples

```
COMPUTE SALARY ROUNDED = WAGES * HOURS.
```

```
COMPUTE SECONDS = (((HRS * 60) + MIN) * 60) + SEC.
```

```
COMPUTE AVERAGE = TOTAL / KOUNT
ON SIZE ERROR MOVE 0 TO AVERAGE.
```

```
COMPUTE PAY (DATE) ROUNDED
= RATE * 8.
```

The DELETE Statement (Relative and Indexed I-O)

The **DELETE** statement logically removes a record from a mass storage file.

FORMAT

DELETE file-name RECORD [;INVALID KEY imperative-statement]

After the successful execution of a **DELETE** statement, the identified record has been logically removed from the file and can no longer be accessed.

The execution of a **DELETE** statement does not affect the contents of the record area associated with **file-name**.

The associated file must be opened in the I-O mode at the time of execution of this statement.

For files in the sequential access mode, the last input-output statement executed for **file-name** prior to the execution of the **DELETE** statement must have been a successfully executed **READ** statement. The system logically removes from the file the record that was accessed by that **READ** statement.

For a file in random or dynamic access mode, the system logically removes from the file that record identified by the contents of the key data item associated with **file-name**. If the file does not contain the record specified by the key, an **INVALID KEY** condition exists.

The execution of the **DELETE** statement causes the value of the specified **FILE STATUS** data item, if any, associated with **file-name** to be updated.

The INVALID KEY Phrase

The **INVALID KEY** phrase must not be specified for a **DELETE** statement which references a file which is in sequential access mode.

The **INVALID KEY** phrase must be specified for a **DELETE** statement which references a file which is not in sequential access mode and for which an applicable **USE** procedure is not specified.

The current record pointer is not affected by the execution of a **DELETE** statement.

The DISPLAY Statement

The DISPLAY statement causes low volume data to be displayed on the specified CRT terminal. DISPLAY statement phrases allow the specification of position, form and format of the displayed data.

FORMAT

```
DISPLAY {{identifier-1} [,UNIT {identifier-2}]
           {literal-1}          {literal-2}
           [,LINE {identifier-3}] [,POSITION {identifier-4}]
           {literal-3}          {literal-4}
           [,SIZE {identifier-5}] [,BEEP] [,ERASE]
           {literal-5}
           [, {HIGH}] [,BLINK] [,REVERSE] ...
           {LOW}}
```

The DISPLAY statement causes the contents of each operand (identifier-1 or literal-1) to be transferred to the CRT device in the order listed. The sending data item must have DISPLAY usage.

When a DISPLAY statement contains more than one operand, the values of the operands are transferred in the sequence in which the operands are encountered.

Note: Features which require support of the host operating system and/or terminal hardware may not be supported on all systems. Any features which are not supported will compile correctly, but will be ignored at runtime. See the User's Guide for specific details.

The UNIT Phrase

The UNIT phrase, if specified, must be written first. The other phrases may be written in any order.

The value of identifier-2 or literal-2 in the UNIT phrase specifies the station identifier of the CRT upon which the data is to be displayed. If the UNIT phrase is omitted, the CRT which executed the program will be accessed.

The LINE Phrase

The value of identifier-3 or literal-3 in the LINE phrase specifies the line number upon which the data is to be displayed on the screen of the CRT terminal, with one being the top line. If the value is greater than the number of lines on the CRT screen, it is adjusted to the maximum line number after the screen has been scrolled one line. If the value is zero or the LINE phrase is not present in a DISPLAY statement, then data is to be displayed on the next line below the current position of the cursor on the CRT screen unless the value specified in the POSITION phrase is also zero, in which case the data is to be displayed on the line at the current position of the cursor on the CRT screen. If incrementing to the next line generates a line number greater than the maximum number of lines on the CRT screen, the new line is displayed at the bottom.

The POSITION Phrase

The value of identifier-4 or literal-4 in the POSITION phrase specifies the number of the character to which the cursor is to be positioned within the specified line prior to the displaying of data on the screen on the CRT terminal, with 1 being the leftmost character position within a line. If the value is greater than the maximum number of characters within a line on the CRT screen, it is adjusted to the next row and decremented by the maximum number of characters within a line.

If the POSITION phrase is not specified, a value of one is assumed for the first displayed operand and zero for each additional operand displayed in the same statement. If a value of zero is specified, the data is to be displayed starting at the next field on the CRT screen (starting character position plus size of the last ACCEPT or DISPLAY).

The SIZE Phrase

The value of identifier-5 or literal-5 in the SIZE phrase specifies the number of characters to be displayed on the screen of the CRT terminal, overriding the Data Division definition of the field. If the SIZE phrase is not present or a value of zero is specified, the size of identifier-1 or literal-1 is used. If literal-1 is a figurative constant, the literal has a size of one. A size greater than 80 is treated as equal to 80.

If the size of the display field is less than the size of the sending data item, only the leftmost characters are displayed. If the specified size is greater than the size of the sending date item, the results are unpredictable. If the sending item is a figurative constant, the constant fills the display field. No conversions are made in the transfer to the display field.

The BEEP Phrase

The presence of the key word BEEP within a DISPLAY statement causes a beep signal to occur on cursor positioning prior to the display of the data. If the BEEP key word is omitted, no signal is given on cursor positioning.

The ERASE Phrase

The presence of the key word ERASE within a DISPLAY statement causes the screen of the CRT terminal to be erased before the content of identifier-1 or literal-1 is displayed on the screen. When the ERASE phrase is not specified, then the screen is not erased prior to the display of the data.

The HIGH/LOW Phrase

The presence of HIGH or LOW causes the data to be displayed at the specified intensity. When HIGH or LOW is not specified, the default display is HIGH.

The BLINK Phrase

The presence of the key word BLINK causes the displayed data to be BLINKed. The normal mode is no blink.

The REVERSE Phrase

The REVERSE key word causes the data to be displayed in REVERSE video. The normal mode is no reverse.

DISPLAY Examples

DISPLAY "FLIGHT ARRIVING AT GATE", LINE FLT-LN,
POSITION 1, ERASE; GATE-NUMBER, HIGH, BLINK.

DISPLAY "ENTER JOB CODE: ".

DISPLAY CRT-HEADER LINE 1 ERASE.

DISPLAY ZEROES SIZE 5.

DISPLAY QUOTE.

The DIVIDE Statement

The DIVIDE statement divides one numeric data item into another and stores the quotient.

FORMAT 1

```
DIVIDE {identifier-1} INTO identifier-2 [ROUNDED]  
      {literal-1    }  
      [;ON SIZE ERROR imperative-statement]
```

FORMAT 2

```
DIVIDE {identifier-1} INTO {identifier-2}  
      {literal-1    }      {literal-2    }  
GIVING identifier-3 [ROUNDED]  
      [;ON SIZE ERROR imperative-statement]
```

FORMAT 3

```
DIVIDE {identifier-1} BY {identifier-2}  
      {literal-1    }      {literal-2    }  
GIVING identifier-3 [ROUNDED]  
      [;ON SIZE ERROR imperative-statement]
```

In Format 1, the value of identifier-1 or literal-1 is divided into the value of identifier-2. The value of the dividend (identifier-2) is replaced by this quotient.

In Format 2, the value of identifier-1 or literal-1 is divided into the value of identifier-2 or literal-2 and the result is stored in identifier-3.

In Format 3, the value of identifier-1 or literal-1 is divided by the value of identifier-2 or literal-2 and the result is stored in identifier-3.

Each identifier must refer to an elementary numeric item, except that any identifier associated with the GIVING phrase must refer to either an elementary numeric item or an elementary numeric edited item.

Each literal must be a numeric literal.

The ROUNDED Phrase

The DIVIDE statement may optionally include the ROUNDED phrase.

If, after decimal point alignment, the number of places in the fraction of the result of an arithmetic operation is greater than the number of places provided for the fraction of the resultant-identifier, truncation is relative to the size provided for the resultant-identifier. When rounding is requested, the absolute value of the resultant-identifier is increased by one (1) whenever the most significant digit of the excess is greater than or equal to five (5).

When the low-order integer positions in a resultant identifier are represented by the character 'P' in the picture for that resultant-identifier, rounding or truncation occurs relative to the rightmost integer position for which storage is allocated.

The SIZE ERROR Phrase

If, after appropriate decimal point alignment, the absolute value of the result exceeds the largest value that can be contained in the associated resultant-identifier, a size error condition exists. If the ROUNDED phrase is specified, rounding takes place before checking for size error.

If the resultant-identifier has COMPUTATIONAL-3 usage, size error is detected only for data items declared with an odd length picture clause. Therefore all COMP-3 data items should be declared with an odd number of character positions.

Division by zero always causes a size error condition.

If the SIZE ERROR phrase is not specified and a size error condition exists, the value of the resultant-identifier is undefined.

If the SIZE ERROR phrase is specified and a size error condition exists, the value of the resultant-identifier is not altered and the imperative statement in the SIZE ERROR phrase is executed.

DIVIDE

DIVIDE Examples

DIVIDE 10 INTO TOTAL-WORK-LOAD
GIVING MORRISS-WORK-LOAD

DIVIDE TOTAL-WORK-LOAD BY 2.5
GIVING ALFREDS-WORK-LOAD ROUNDED
ON SIZE ERROR GO TO ALFRED-QUIT.

DIVIDE 2.5 INTO TOTAL.

The EXIT Statement

The EXIT statement provides a common end point for a series of procedures or the logical end of a called program.

FORMAT

EXIT [PROGRAM].

The EXIT statement must appear in a sentence by itself.

The EXIT sentence must be the only sentence in the paragraph.

An EXIT statement without the word PROGRAM serves only to enable the user to assign a procedure-name to a given point in a program. Such an EXIT statement has no other effect on the compilation or execution of the program.

An execution of an EXIT PROGRAM statement in a CALLED program causes control to be passed to the calling program. Execution of an EXIT PROGRAM statement in a program which is not called behaves as if the statement were an EXIT statement without the word PROGRAM.

The GO TO Statement

The GO TO statement causes control to be transferred from one part of the Procedure Division to another.

FORMAT 1

GO TO procedure-name-1.

FORMAT 2

GO TO procedure-name-1 [,procedure-name-2] ...,
procedure-name-n DEPENDING ON identifier-1.

If a Format 1 GO TO statement appears in a consecutive sequence of imperative statements within a sentence, it must appear as the last statement in that sequence.

When a Format 1 GO TO statement is executed, control is transferred to procedure-name-1 or to another procedure-name if the GO TO statement has been modified by an ALTER statement.

When a paragraph is referenced by an ALTER statement, that paragraph can consist only of a paragraph header followed by a Format-1 GO TO statement.

The DEPENDING ON Phrase

When a Format 2 GO TO statement is executed, control is transferred to procedure-name-1, procedure-name-2, etc., depending on the value of the identifier-1 being 1, 2, ..., n. If the value of the identifier-1 is anything other than the positive or unsigned integers 1, 2, ..., n, then no transfer occurs and control passes to the next statement in the normal sequence for execution.

Identifier-1 is the name of a numeric integer elementary item.

The IF Statement

The IF statement causes a specified condition to be evaluated. The subsequent action of the object program depends on whether the value of the condition is true or false.

FORMAT

```
IF condition; {statement-1 } {;ELSE statement-2 }
               {NEXT SENTENCE} {;ELSE NEXT SENTENCE}
```

Statement-1 and statement-2 represent either an imperative statement or a conditional statement, and either may be followed by a conditional statement.

When an IF statement is executed, the following transfers of control occur:

If the condition is true, statement-1 is executed if specified. If statement-1 contains a procedure branching or conditional statement, control is explicitly transferred in accordance with the rules of that statement. If statement-1 does not contain a procedure branching or conditional statement, the ELSE phrase, if specified, is ignored and control passes to the next executable sentence.

If the condition is true and the NEXT SENTENCE phrase is specified instead of statement-1, the ELSE phrase, if specified, is ignored and control passes to the next executable sentence.

If the condition is false, statement-1 or its surrogate NEXT SENTENCE is ignored, and statement-2, if specified, is executed. If statement-2 contains a procedure branching or conditional statement, control is explicitly transferred in accordance with the rules of that statement. If statement-2 does not contain a procedure branching or conditional statement, control passes to the next executable sentence. If the ELSE statement-2 phrase is not specified, statement-1 is ignored and control passes to the next executable sentence.

If the condition is false, and the ELSE NEXT SENTENCE phrase is specified, statement-1 is ignored, if specified, and control passes to the next executable sentence.

Statement-1 and/or statement-2 may contain an IF statement. In this case the IF statement is said to be nested.

IF statements within IF statements may be considered as paired IF and ELSE combinations, proceeding from left to right. Thus, any ELSE encountered is considered to apply to the immediately preceding IF that has not been already paired with an ELSE.

The ELSE NEXT SENTENCE phrase may be omitted if it immediately precedes the terminal period of the sentence.

IF Examples

```
IF CHAR-STR IS ALPHABETIC,  
    MOVE CHAR-STR TO ALPHA-STR;  
ELSE IF CHAR-STR IS NUMERIC,  
    MOVE CHAR-STR TO NUM;  
    DISPLAY NUM;  
ELSE NEXT SENTENCE.
```

```
IF NUM = OLD-NUM GO TO RE-SET.
```

```
IF ALPHA-STR NOT = "TEST"  
    ADD 1 TO ERROR-CNT.
```

```
IF NUM < LIMIT, ADD 1 TO NUM.
```

```
IF NUM IS LESS THAN LIMIT  
    ADD 1 TO NUM.
```

```
IF PRINT-SWITCH PERFORM PRINT-ROUTINE.
```

The INSPECT Statement

The INSPECT statement provides the ability to tally (Format 1), replace (Format 2), or tally and replace (Format 3) occurrences of single characters or groups of characters in a data item.

FORMAT 1

INSPECT identifier-1

```
TALLYING identifier-2 FOR {{ALL      } {identifier-3}}
                           {literal-1    }
                           {{LEADING}          }
                           {      CHARACTERS      }
                           [{BEFORE} INITIAL {identifier-4}]}
                           {literal-2    }
                           {AFTER }
```

FORMAT 2

INSPECT identifier-1

```
REPLACING {{ALL      } {identifier-5}} BY {identifier-6}
                           {literal-3    } {literal-4    }
                           {{LEADING}          }
                           {{FIRST   }          }
                           {      CHARACTERS      }
                           [{BEFORE} INITIAL {identifier-7}]}
                           {literal-5    }
                           {AFTER }
```

FORMAT 3

```

INSPECT identifier-1

TALLYING identifier-2 FOR {{ALL } {identifier-3}}
                           {literal-1 }
                           {{LEADING} }
                           { CHARACTERS }

[{{BEFORE} INITIAL {identifier-4}]
                           {literal-2 }

{AFTER }

REPLACING {{ALL } {identifier-5}} BY {identifier-6}
                           {literal-3 } {literal-4 }
                           {{LEADING} }
                           {{FIRST } }
                           { CHARACTERS }

[{{BEFORE} INITIAL {identifier-7}]
                           {literal-5 }

{AFTER }

```

Identifier-1 must reference either a group item or any category of elementary item, described (either implicitly or explicitly) as usage is DISPLAY.

Identifier-3 ... identifier-n must reference either an elementary alphabetic, alphanumeric or numeric item described (either implicitly or explicitly) as usage is DISPLAY and a size of one character.

Each literal may be either a figurative constant (which is treated as a one-character data item) or a nonnumeric literal one character in length.

The general rules that apply to the INSPECT statement are:

1. Inspection (which includes the comparison cycle, the establishment of boundaries for the BEFORE or AFTER phrase, and the mechanism for tallying and/or replacing) begins at the leftmost character position of the data item referenced by identifier-1, regardless of its class, and proceeds from left to right to the rightmost character position as described in general rules 4 through 6.

2. For use in the INSPECT statement, the contents of the data item referenced by identifier-1, identifier-3, identifier-4, identifier-5, identifier-6 or identifier-7 will be treated as follows:
 - a. If any of identifier-1, identifier-3, identifier-4, identifier-5, identifier-6, or identifier-7 are described as alphanumeric, the INSPECT statement treats the contents of each such identifier as a character-string.
 - b. If any of identifier-1, identifier-3, identifier-4, identifier-5, identifier-6, or identifier-7 are described as alphanumeric edited, numeric edited or unsigned numeric, the data item is inspected as though it had been redefined as alphanumeric (see general rule 2a) and the INSPECT statement had been written to reference the redefined data item.
 - c. If any of the identifier-1, identifier-3, identifier-4, identifier-5, identifier-6, or identifier-7 are described as signed numeric, the data item is inspected as though it had been moved to an unsigned numeric data item of the same length and then the rules in general rule 2b had been applied. (See the MOVE statement.)
3. In general rules 4 through 10, all references to literal-1, literal-2, literal-3, literal-4, and literal-5 apply equally to the contents of the data item referenced by identifier-3, identifier-4, identifier-5, identifier-6, and identifier-7, respectively.
4. During inspection of the contents of the data item referenced by identifier-1, each properly matched occurrence of literal-1 is tallied (Formats 1 and 3) and/or each properly matched occurrence of literal-3 is replaced by literal-4 (Formats 2 and 3).

5. The comparison operation to determine the occurrences of literal-1 to be tallied and/or occurrences of literal-3 to be replaced, occurs as follows:
 - a. The character specified by literal-1, literal-3 is compared to successive characters, starting with the leftmost character position in the data item referenced by identifier-1. Literal-1, literal-3 and that portion of the contents of the data item referenced by identifier-1 match if, and only if, they are equal.
 - b. If no match occurs in the comparison of literal-1, literal-3, the comparison is repeated starting with the next character position of identifier-1.
 - c. Whenever a match occurs, tallying and/or replacing takes place as described in general rules 8 through 10. The character position in the data item referenced by identifier-1 immediately to the right of the character position that caused the match is now considered to be the leftmost character position of the data item referenced by identifier-1, and the comparison cycle starts again with literal-1, literal-3.
 - d. The comparison operation continues until the rightmost character position of the data item referenced by identifier-1 has participated in a match or has been considered as the leftmost character position. When this occurs, inspection is terminated.
 - e. If the CHARACTERS phrase is specified, an implied one-character operand participates in the cycle described in paragraphs 5a through 5d above, except that no comparison to the contents of the data item referenced by identifier-1 takes place. This implied character is considered always to match the leftmost character of the contents of the data item referenced by identifier-1 participating in the current comparison cycle.
6. The comparison operation defined in general rule 5 is affected by the BEFORE and AFTER phrases as follows:
 - a. If the BEFORE and AFTER phrase is not specified, literal-1, literal-3 or the implied operand of the CHARACTERS phrase participates in the comparison operation as described in general rule 5.

- b. If the BEFORE phrase is specified, the associated literal-1, literal-3 or the implied operand of the CHARACTERS phrase participates only in those comparison cycles which involve that portion of the contents of the data item referenced by identifier-1 from its leftmost character position up to, but not including the first occurrence of literal-2, literal-5 within the contents of the data item referenced by identifier-1. The position of this first occurrence is determined before the first cycle of the comparison operation described in general rule 5 is begun. If, on any comparison cycle, literal-1, literal-3 or the implied operand of the CHARACTERS phrase is not eligible to participate, it is considered not to match the contents of the data item referenced by identifier-1. If there is no occurrence of literal-2, literal-5 within the contents of the data item referenced by identifier-1, its associated literal-1, literal-3, or the implied operand of the CHARACTERS phrase participates in the comparison operation as though the BEFORE phrase had not been specified.
- c. If the AFTER phrase is specified, the associated literal-1, literal-3 or the implied operand of the CHARACTERS phrase may participate only in those comparison cycles which involve that portion of the contents of the data item referenced by identifier-1 from the character position immediately to the right of the rightmost character position of the first occurrence of literal-2, literal-5, within the contents of the data item referenced by identifier-1 and the rightmost character position of the data item referenced by identifier-1. The position of this first occurrence is determined before the first cycle of the comparison operation described in general rule 5 is begun. If, on any comparison cycle, literal-1, literal-3, or the implied operand of the CHARACTERS phrase is not eligible to participate, it is considered not to match the contents of the data item referenced by identifier-1. If there is no occurrence of literal-2, literal-5 within the contents of the data item referenced by identifier-1, its associated literal-1, literal-3, or the implied operand of the CHARACTERS phrase is never eligible to participate in the comparison operation.

Format 1

- 7. The contents of the data item referenced by identifier-2 is not initialized by the execution of the INSPECT statement.

8. The rules for tallying are as follows:

- a. If the ALL phrase is specified, the contents of the data item referenced by identifier-2 is incremented by one (1) for each occurrence of literal-1 matched within the contents of the data item referenced by identifier-1.
- b. If the LEADING phrase is specified, the contents of the data item referenced by identifier-2 is incremented by one (1) for each contiguous occurrence of literal-1 matched within the contents of the data item referenced by identifier-1, provided that the leftmost such occurrence is at the point where comparison began in the first comparison cycle in which literal-1 was eligible to participate.
- c. If the CHARACTERS phrase is specified, the contents of the data item referenced by identifier-2 is incremented by one (1) for each character matched, in the sense of general rule 5e, within the contents of the data item referenced by identifier-1.

Format 2

9. The rules for replacement are as follows:

- a. When the CHARACTERS phrase is specified, each character matched, in the sense of general rule 5e, in the contents of the data item referenced by identifier-1 is replaced by literal-4.
- b. When ALL is specified, each occurrence of literal-3 matched in the contents of the data item referenced by identifier-1 is replaced by literal-4.
- c. When LEADING is specified, each contiguous occurrence of literal-3 matched in the contents of the data item referenced by identifier-1 is replaced by literal-4, provided that the leftmost occurrence is at the point where comparison began in the first comparison cycle in which literal-3 was eligible to participate.
- d. When FIRST is specified, the leftmost occurrence of literal-3 matched within the contents of the data item referenced by identifier-1 is replaced by literal-4.

Format 3

10. A Format 3 INSPECT statement is interpreted and executed as though two successive INSPECT statements specifying the same identifier-1 had been written with one statement being a Format 1 statement with TALLYING phrases identical to those specified in the Format 3 statement, and the other statement being a Format 2 statement with REPLACING phrases identical to those specified in the Format 3 statement. The general rules given for matching and counting apply to the Format 1 statement and the general rules given for matching and replacing apply to the Format 2 statement.

INSPECT

INSPECT Examples:

INSPECT word TALLYING count FOR LEADING "L" BEFORE INITIAL "A",

Where word=LARGE, count=1.
Where word=ANALYST, count=0.

INSPECT word TALLYING count FOR LEADING "A" BEFORE INITIAL "L".

Where word=LARGE, count=0.
Where word=ANALYST, count=1.

INSPECT word TALLYING count FOR ALL "L", REPLACING LEADING "A" BY
"E" AFTER INITIAL "L".

Where word=CALLAR, count=2, word=CALLER.
Where word=SALAMI, count=1, word=SALEMI.
Where word=LATTER, count=1, word=LETTER.

INSPECT word REPLACING ALL "A" BY "G" BEFORE INITIAL "X".

Where word=ARXAX, word=GRXAX.
Where word=HANDAX, word=HGNDGX.

INSPECT word TALLYING count FOR CHARACTERS AFTER INITIAL "J"
REPLACING ALL "A" BY "B".

Where word=ADJECTIVE, count=6, word=BDJECTIVE.
Where word=JACK, count=3, word=JBCK.
Where word=JUJMAB, count=5, word=JUJMBB.

INSPECT word REPLACING ALL "W" BY "Q" AFTER
INITIAL "R".

Where word=RXXBQWY, word=RXXBQQY.
Where word=YZACDWBR, word=YZACDWBR.
Where word=RAWRXEB, word=RAQRXEB.

INSPECT word REPLACING CHARACTERS BY "B" BEFORE INITIAL "A".

word before: 12 XZABCD
word after: BBBBABC

The MOVE Statement

The MOVE statement transfers data, in accordance with the rules of editing, to one or more data areas.

FORMAT 1

```
MOVE {identifier-1} TO identifier-2 [,identifier-3]...
      {literal      }
```

FORMAT 2

```
MOVE {CORRESPONDING} identifier-1 TO identifier-2
      {CORR          }
```

Identifier-1 and literal-1 represent the sending area; identifier-2, identifier-3, ..., represent the receiving area(s).

An index data item cannot appear as an operand of a MOVE statement.

The data designated by literal-1 or identifier-1 is moved first to identifier-2, then to identifier-3, The rules referring to identifier-2 also apply to the other receiving areas. Any subscripting or indexing associated with identifier-2, ..., is evaluated immediately before the data is moved to the respective data item.

Any subscripting or indexing associated with identifier-1 is evaluated only once, immediately before data is moved to the first of the receiving operands. The result of the statement

```
MOVE a (b) TO b, c (b)
```

is equivalent to:

```
MOVE a (b) TO temp
MOVE temp TO b
MOVE temp TO c (b).
```

Any MOVE in which the sending and receiving items are both elementary items is an elementary move. Every elementary item belongs to one of the following categories: numeric, alphabetic, alphanumeric, numeric edited, alphanumeric edited. These categories are described in the PICTURE clause. Numeric literals belong to the category numeric, and nonnumeric literals belong to the category alphanumeric. The figurative constant ZERO belongs to the category numeric. The figurative constant SPACE belongs to the category alphabetic. All other figurative constants belong to the category alphanumeric.

The following rules apply to an elementary move between these categories:

1. The figurative constant SPACE, a numeric edited, alphanumeric edited, or alphabetic data item must not be moved to a numeric or numeric edited data item.
2. A numeric literal, the figurative constant ZERO, a numeric data item or a numeric edited data item must not be moved to an alphabetic data item.
3. A non integer numeric literal or a non integer numeric data item must not be moved to an alphanumeric or alphanumeric edited data item.
4. All other elementary moves are legal and are performed according to the rules given below.

Any necessary conversion of data from one form of internal representation to another takes place during legal elementary moves, along with any editing specified for the receiving data item:

1. When an alphanumeric edited or alphanumeric item is a receiving item, alignment and any necessary space-filling takes place as defined under Standard Alignment Rules. If the size of the sending item is greater than the size of the receiving item, the excess characters are truncated on the right after the receiving item is filled. If the sending item is described as being signed numeric, the operational sign will not be moved; if the operational sign occupies a separate character position (see the SIGN clause), that character will not be moved and the size of the sending item will be considered to be one less than its actual size (in terms of standard data format characters).

2. When a numeric or numeric edited item is the receiving item, alignment by decimal point and any necessary zero-filling takes place as defined under the Standard Alignment Rules except where zeroes are replaced because of editing requirements.

When a signed item is the receiving item, the sign of the sending item is placed in the receiving item. (See the SIGN clause). Conversion of the representation of the sign takes place as necessary. If the sending item is unsigned, a positive sign is generated for the receiving item.

When an unsigned numeric item is the receiving item, the absolute value of the sending item is moved and no operational sign is generated for the receiving item.

When a data item described as alphanumeric is the sending item, data is moved as if the sending item were described as an unsigned numeric integer.

3. When a receiving field is described as alphabetic, justification and any necessary space-filling takes place as defined under the Standard Alignment Rules. If the size of the sending item is greater than the size of the receiving item, the excess characters are truncated on the right after the receiving item is filled.

Any move that is not an elementary move is treated exactly as if it were an alphanumeric to alphanumeric elementary move, except that there is no conversion of data from one form of internal representation to another. In such a move, the receiving area will be filled without consideration for the individual elementary or group items contained within either the sending or receiving area, except as noted in the OCCURS clause.

When a sending and receiving item share a part of their storage areas, the result of the execution of such a statement is undefined.

The CORRESPONDING Phrase

When the CORRESPONDING phrase is specified, data items in identifier-1 are moved to corresponding data items in identifier-2 according to the following rules:

A data item in identifier-1 and a data item in identifier-2 are not designated by the key word FILLER and have the same qualifiers up to, but not including, identifier-1 and identifier-2.

At least one of the data items is an elementary data item.

MOVE

The description of identifier-1 and identifier-2 must not contain level-number 66, 77, or 88 or the USAGE IS INDEX clause.

A data item that is subordinate to identifier-1 or identifier-2 and contains a REDEFINES, RENAMES, OCCURS or USAGE IS INDEX clause is ignored, as well as those data items subordinate to the data item that contains the REDEFINES, OCCURS, or USAGE IS INDEX clause. However, identifier-1 and identifier-2 may have REDEFINES or OCCURS clauses or be subordinate to data items with REDEFINES or OCCURS clauses.

Data in the following chart summarizes the legality of the various types of MOVE statements.

CATEGORY OF RECEIVING DATA ITEM			
CATEGORY OF SENDING DATA ITEM	ALPHANUMERIC	NUMERIC INTEGER	
	EDITED	NUMERIC NON-INTEGER	
ALPHABETIC	YES	YES	NO
ALPHANUMERIC	YES	YES	YES
ALPHANUMERIC EDITED	YES	YES	NO
INTEGER	NO	YES	YES
NON-INTEGER	NO	NO	YES
NUMERIC EDITED	NO	YES	NO

MOVE Examples

MOVE INCOME TO TOTAL-INCOME.

MOVE 1 TO PAGE-COUNT, LINE-NUM

MOVE "MARMACK INDUSTRIES" TO TITLE-HEADER.

MOVE PERSON IN FILE-RECORD TO
PERSON OF ALABAMA (I-A OF ALABAMA),
PERSON OF CROSS-CENSUS.

MOVE NUM TO NUM-ED

MOVE TABLE-ELT (N, 1, M) TO NEXT-ENTRY
PREVIOUS-ENTRY

MOVE -36.7 TO DEFICIT.

MOVE QUOTES TO SECTION-DIVIDER.

MOVE ZERO TO COUN-TER

MOVE ZEROES TO COUN-TER.

The MULTIPLY Statement

The MULTIPLY statement causes numeric data items to be multiplied and stores the result.

FORMAT 1

```
MULTIPLY {identifier-1}
    {literal-1    }

    BY identifier-2 [ROUNDED]

[;ON SIZE ERROR imperative-statement]
```

FORMAT 2

```
MULTIPLY {identifier-1} BY {identifier-2}
    {literal-1    }    {literal-2    }

    GIVING identifier-3 [ROUNDED]

[;ON SIZE ERROR imperative-statement]
```

In Format 1, the value of identifier-1 or literal-1 is multiplied by the value of identifier-2. The value of the multiplier (identifier-2) is replaced by this product.

In Format 2, the value of identifier-1 or literal-1 is multiplied by identifier-2 or literal-2 and the result is stored in identifier-3.

Each identifier must refer to a numeric elementary item, except that in Format 2 the identifier following the word GIVING must refer to either an elementary numeric item or an elementary numeric edited item.

Each literal must be a numeric literal.

The ROUNDED Phrase

The MULTIPLY statement may optionally include the ROUNDED phrase.

If, after decimal point alignment, the number of places in the fraction of the result of an arithmetic operation is greater than the number of places provided for the fraction of the resultant-identifier, truncation is relative to the size provided for the resultant-identifier. When rounding is requested, the absolute value of the resultant-identifier is increased by one (1) whenever the most significant digit of the excess is greater than or equal to five (5).

When the low-order integer positions in a resultant-identifier are represented by the character 'P' in the picture for that resultant-identifier, rounding or truncation occurs relative to the rightmost integer position for which storage is allocated.

The SIZE ERROR Phrase

If, after appropriate decimal point alignment, the absolute value of the result exceeds the largest value that can be contained in the associated resultant-identifier, a size error condition exists. If the ROUNDED phrase is specified, rounding takes place before checking for size error.

If the resultant-identifier has COMPUTATIONAL-3 usage, size error is detected only for data items declared with an odd length picture clause. Therefore all COMP-3 data items should be declared with an odd number of character positions.

If the SIZE ERROR phrase is not specified and a size error condition exists, the value of the resultant-identifier is undefined.

If the SIZE ERROR phrase is specified and a size error condition exists, the value of the resultant-identifier is not altered and the imperative statement is the SIZE ERROR phrase is executed.

MULTIPLY Examples

MULTIPLY 10 BY INCOME.

MULTIPLY PRINCIPAL BY INTEREST-RATE
GIVING INTEREST ROUNDED.

MULTIPLY INFLATION-RATE BY EXPENSES
ON SIZE ERROR MOVE 0 TO ECONOMY-RATING.

OPEN (Sequential I-O)

The OPEN Statement (Sequential I-O)

The OPEN statement initiates the processing of sequential files.

FORMAT

```
| OPEN {{INPUT {file-name-1 [WITH LOCK] [WITH NO REWIND] }...}...  
|           {OUTPUT {file-name-2 [WITH NO REWIND] }...}...  
|           {I-O {file-name-3 [WITH LOCK] ... }... }...  
|           {EXTEND {file-name-4 }... }... }...}
```

The successful execution of an OPEN statement determines the availability of the file and results in the file being in an open mode.

The successful execution of an OPEN statement makes the associated record area available to the program.

The files referenced in the OPEN statement need not all have the same organization or access.

Prior to the successful execution of an OPEN statement for a given file, no statement can be executed that references that file, either explicitly or implicitly.

An OPEN statement must be successfully executed prior to the execution of any of the permissible input-output statements. In the Permissible Statements Table below, 'X' at an intersection indicates that the specified statement, used in the sequential access mode, may be used with the sequential file organization and open mode given at the top of the column.

Open Mode				
Statement	Input	Output	Input-Output	Extend
READ	X		X	
WRITE		X		X
REWRITE			X	

Permissible Statements Table

A file may be opened with the INPUT, OUTPUT, EXTEND, and I-O phrases in the same program. Following the initial execution of an OPEN statement for a file, each subsequent OPEN statement execution for that same file must be preceded by the execution of a CLOSE statement, without the LOCK phrase, for that file.

Execution of the OPEN statement does not obtain or release the first data record.

The file description entry for file-name-1, file-name-3 or file-name-4 must be equivalent to that used when this file was created.

The execution of an OPEN statement causes the value of the specified FILE STATUS data item, if any, associated with file-name-1 ... to be updated.

The INPUT Phrase

For files being opened with the INPUT phrase, the OPEN statement sets the current record pointer to the first record currently existing within the file. If no records exist in the file, the current record pointer is set such that the next executed READ statement for the file will result in an AT END condition.

The OUTPUT Phrase

Upon successful execution of an OPEN statement with the OUTPUT phrase specified, a file is created. At that time the associated file contains no data records.

The EXTEND Phrase

When the EXTEND phrase is specified, the OPEN statement positions the file immediately following the last logical record of that file. Subsequent WRITE statements referencing the file will add records to the file as though the file has been opened with the OUTPUT phrase.

The EXTEND phrase and NO REWIND phrase can be used only for sequential files. The EXTEND phrase must not be specified for a file whose device-type is INPUT.

OPEN (Sequential I-O)

When the EXTEND phrase is specified and the LABEL RECORDS clause indicates label records are present, the execution of the OPEN statement includes the following:

The beginning file labels are processed only in the case of a single reel/unit file.

Processing then proceeds as though the file has been opened with the OUTPUT phrase.

The I-O Phrase

The I-O phrase permits the opening of a mass storage file for both input and output operations. Since this phrase implies the existence of the file, it cannot be used if the mass storage file is being initially created.

The I-O phrase can be used only for mass storage files (files assigned to the RANDOM device-type).

When the I-O phrase is specified and the LABEL RECORDS clause indicates that label records are present, the execution of the OPEN includes the following:

The labels are checked.

New labels are written.

The OPEN statement sets the current record pointer to the first record currently existing in the file. If no records exist in the file, the current record pointer is set such that the next executed READ statement for that file will result in an AT END condition.

The NO REWIND Phrase

The NO REWIND phrases can only be used with sequential single reel/unit files. Both phrases will be ignored if they do not apply to the storage media on which the file resides.

If the storage medium for the file permits rewinding, the following rule applies:

When neither the EXTEND nor the NO REWIND phrase is specified, execution of the OPEN statement causes the file to be positioned at its beginning.

When the NO REWIND phrase is specified, execution of the OPEN statement does not cause the file to be repositioned; the file must be already positioned at its beginning prior to the execution of the OPEN statement.

The LOCK Phrase

The LOCK phrase permits the opening of a mass storage file for exclusive access of the file. This is the default mode for files OPEN for OUTPUT.

The OPEN Statement (Relative and Indexed I-O)

The OPEN statement initiates the processing of mass storage files.

FORMAT

```
|   OPEN {INPUT {file-name-1 [WITH LOCK] }...}...
|   {OUTPUT {file-name-2 [WITH LOCK] }...}...
|   {I-O    {file-name-3 [WITH LOCK] }...}...}
```

The successful execution of an OPEN statement determines the availability of the file and results in the file being in an open mode.

The successful execution of the OPEN statement makes the associated record area available to the program.

The files referenced in the OPEN statement need not all have the same organization or access.

Prior to the successful execution of an OPEN statement for a given file, no statement can be executed that references that file, either explicitly or implicitly.

A file may be opened with the INPUT, OUTPUT, and I-O phrases in the same program. Following the initial execution of an OPEN statement for a file, each subsequent OPEN statement execution for that same file must be preceded by the execution of a CLOSE statement, without the LOCK phrase, for that file.

Execution of the OPEN statement does not obtain or release the first data record.

If label records are specified for the file, the beginning labels are processed as follows:

When the INPUT phrase is specified, the execution of the OPEN statement causes the labels to be checked in accordance with the System conventions for input label checking.

When the OUTPUT phrase is specified, the execution of the OPEN statement causes the labels to be written in accordance with the System conventions for output label writing.

The behavior of the OPEN statement when label records are specified but not present, or when label records are not specified but are present, is undefined.

The file description entry for file-name-1 or file-name-3 must be equivalent to that used when this file was created.

The execution of the OPEN statement causes the value of the specified FILE STATUS data item, if any, associated with file-name-1 ... to be updated.

An OPEN statement must be successfully executed prior to the execution of any of the permissible input-output statements. In the Permissible Statements Table below, 'X' at an intersection indicates that the specified statement, used in the access mode given for that row, may be used with the open mode given at the top of the column.

		Open Mode		
File Access Mode	Statement	Input	Output	Input-Output
Sequential	READ	X		X
	WRITE		X	
	REWRITE			X
	START	X		X
	DELETE			X
Random	READ	X		X
	WRITE		X	X
	REWRITE			X
	START			
	DELETE			X
Dynamic	READ	X		X
	WRITE		X	X
	REWRITE			X
	START	X		X
	DELETE			X

Permissible Statements Table

The INPUT Phrase

For files being opened with the INPUT phrase, the OPEN statement sets the current record pointer to the first record currently existing within the file. If no records exist in the file, the current record pointer is set such that the next executed Format 1 READ statement for the file will result in an AT END condition.

The OUTPUT Phrase

Upon successful execution of an OPEN statement with the OUTPUT phrase specified, a file is created. At that time the associated file contains no data records.

The I-O Phrase

For files being opened with the I-O phrase, the OPEN statement sets the current record pointer to the first record currently existing within the file. If no records exist in the file, the current record pointer is set such that the next executed Format 1 READ statement for the file will result in an AT END condition.

The LOCK Phrase

The LOCK phrase permits the opening of a mass storage file for exclusive access of the file. This is the default mode for files OPEN for OUTPUT.

PERFORM

The PERFORM Statement

The PERFORM statement is used to transfer control explicitly to one or more procedures and to return control implicitly whenever execution of the specified procedure is complete.

FORMAT 1

PERFORM procedure-name-1 [**{THROUGH}** procedure-name-2]
 {THRU}

FORMAT 2

PERFORM procedure-name-1 [**{THROUGH}** procedure-name-2]
 {THRU}

 {identifier-1} TIMES
 {integer }

FORMAT 3

PERFORM procedure-name-1 [**{THROUGH}** procedure-name-2]
 {THRU}

 UNTIL condition-1

FORMAT 4

```

PERFORM procedure-name-1 [{THROUGH} procedure-name-2]
                     {THRU    }

VARYING {identifier-2} FROM {identifier-3}
           {index-name-1}      {index-name-2}
                           {literal-1    }

BY {identifier-4} UNTIL condition-1
      {literal-2    }

[AFTER {identifier-5} FROM {identifier-6}
   {index-name-3}      {index-name-4}
   {literal-3    }

BY {identifier-7} UNTIL condition-2
      {literal-4    }

[AFTER {identifier-8} FROM {identifier-9}
   {index-name-5}      {index-name-6}
   {literal-5    }

BY {identifier-10} UNTIL condition-3]]
      {literal-6    }

```

Format 1 is the basic PERFORM statement. A procedure referenced by this type of PERFORM statement is executed once and then control passes to the next executable statement following the PERFORM statement.

Format 2 is the PERFORM...TIMES. The procedures are performed the number of times specified by integer or by the initial value of the data item referenced by identifier-1 for that execution. If, at the time of execution of a PERFORM statement, the value of the data item referenced by identifier-1 is equal to zero or is negative, control passes to the next executable statement following the PERFORM statement. Following the execution of the procedures the specified number of times, control is transferred to the next executable statement following the PERFORM statement.

During execution of the PERFORM statement, references to identifier-1 cannot alter the number of times the procedures are to be executed from that which was indicated by the initial value of identifier-1.

PERFORM

Format 3 is the PERFORM...UNTIL. The specified procedures are performed until the condition specified by the UNTIL phrase is true. When the condition is true, control is transferred to the next executable statement after the PERFORM statement. If the condition is true when the PERFORM statement is entered, no transfer to procedure-name-1 takes place, and control is passed to the next executable statement following the PERFORM statement.

Format 4 is the PERFORM...VARYING. This variation of the PERFORM statement is used to augment the values referenced by one or more identifiers or index-names in an orderly fashion during the execution of a PERFORM statement. In the following discussion, every reference to identifier as the object of the VARYING, AFTER and FROM (current value) phrases also refers to index-names. When index-name appears in a VARYING and/or AFTER phrase, it is initialized and subsequently augmented (as described below) according to the rules of the SET statement. When index-name appears in the FROM phrase, identifier, when it appears in an associated VARYING or AFTER phrase, is initialized according to the rules of the SET statement; subsequent augmentation is as described below.

In Format 4, when one identifier is varied, identifier-2 is set to the value of literal-1 or the current value of identifier-3 at the point of initial execution of the PERFORM statement; then, if the condition of the UNTIL phrase is false, the sequence of procedures, procedure-name-1 through procedure-name-2, is executed once. The value of identifier-2 is augmented by the specified increment or decrement value (the value of identifier-4 or literal-2) and condition-1 is evaluated again. The cycle continues until this condition is true; at which point, control is transferred to the next executable statement following the PERFORM statement. If condition-1 is true at the beginning of execution of the PERFORM statement, control is transferred to the next executable statement following the PERFORM statement.

Each identifier represents a numeric elementary item described in the Data Division. In Format 2, identifier-1 must be described as a numeric integer.

Each literal represents a numeric literal.

The words THRU and THROUGH are equivalent.

If an index-name is specified in the VARYING or AFTER phrase, then:

The identifier in the associated FROM and BY phrases must be an integer data item.

The literal in the associated FROM phrase must be a positive integer.

The literal in the associated BY phrase must be a non zero integer.

If an index-name is specified in the FROM phrase, then:

The identifier in the associated VARYING or AFTER phrase must be an integer data item.

The identifier in the associated BY phrase must be an integer data item.

The literal in the associated BY phrase must be an integer.

Literal in the BY phrase must not be zero.

Condition-1, condition-2, condition-3 may be any conditional expression.

When procedure-name-1 and procedure-name-2 are both specified and either is the name of a procedure in the declarative section of the program then both must be procedure-names in the same declarative section.

The data items referenced by identifier-4, identifier-7, and identifier-10 must not have a zero value.

If an index-name is specified in the VARYING or AFTER phrase, and an identifier is specified in the associated FROM phrase, then the data item referenced by the identifier must have a positive value.

When the PERFORM statement is executed, control is transferred to the first statement of the procedure named procedure-name-1. This transfer of control occurs only once for each execution of a PERFORM statement. For those cases when a transfer of control to the named procedure does take place, an implicit transfer of control to the next executable statement following the PERFORM statement is established as follows:

PERFORM

If procedure-name-1 is a paragraph-name and procedure-name-2 is not specified, then the return is after the last statement of procedure-name-1.

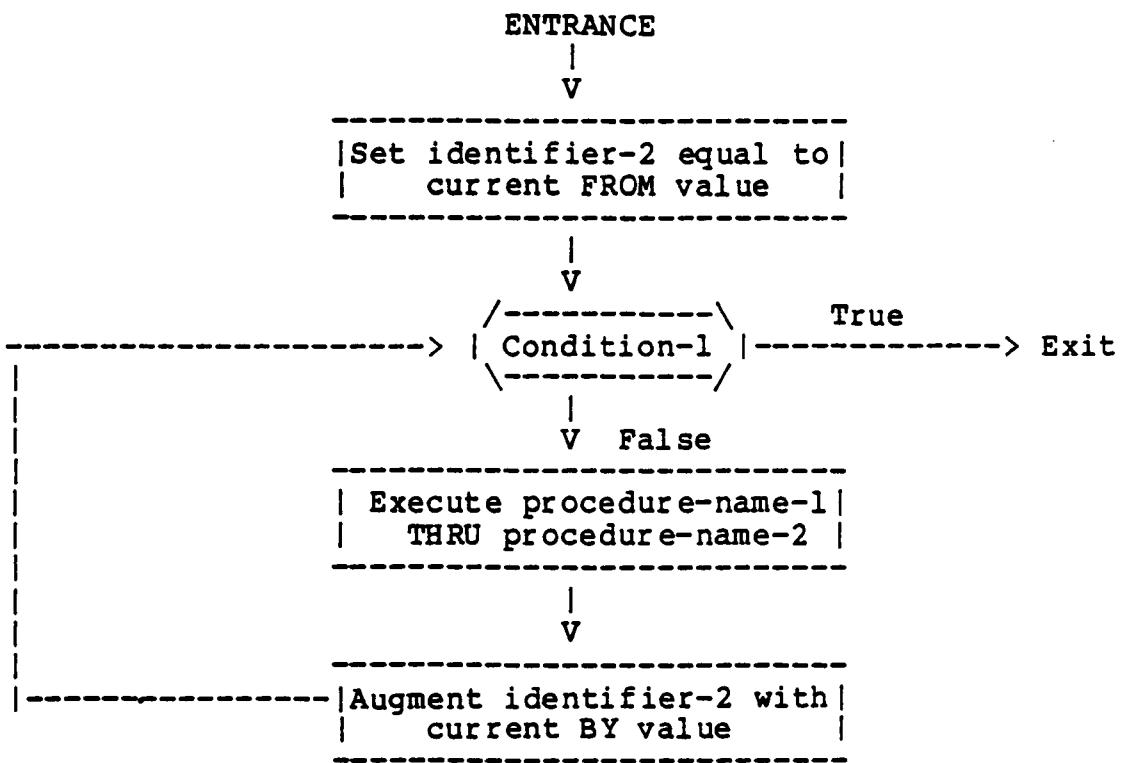
If procedure-name-1 is a section-name and procedure-name-2 is not specified, then the return is after the last statement of the last paragraph in procedure-name-1.

If procedure-name-2 is specified and it is a paragraph-name, then the return is after the last statement of the paragraph.

If procedure-name-2 is specified and it is a section-name, then the return is after the last statement of the last paragraph in the section.

There is no necessary relationship between procedure-name-1 and procedure-name-2 except that a consecutive sequence of operations is to be executed beginning at the procedure named procedure-name-1 and ending with the execution of the procedure named procedure-name-2. In particular, GO TO and PERFORM statements may occur between procedure-name-1 and the end of procedure-name-2. If there are two or more logical paths to the return point, then procedure-name-2 may be the name of a paragraph consisting of the EXIT statement, to which all of these paths must lead.

If control passes to these procedures by means other than a PERFORM statement, control will pass through the last statement of the procedure to the next executable statement as if no PERFORM statement mentioned these procedures.



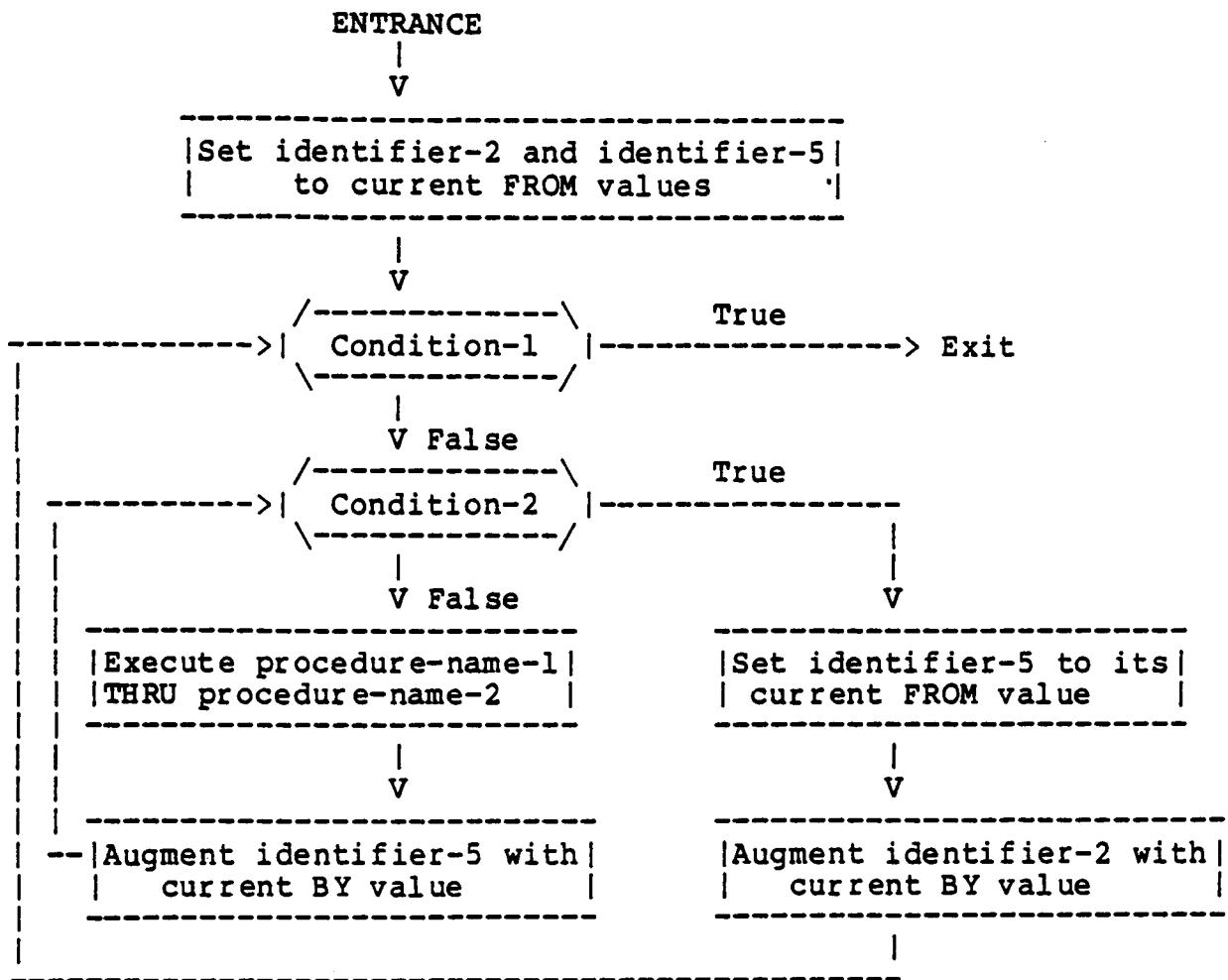
Flowchart for the VARYING Phrase of a PERFORM Statement Having One Condition.

PERFORM

In Format 4, when two identifiers are varied, identifier-2 and identifier-5 are set to the current value of identifier-3 and identifier-6, respectively.

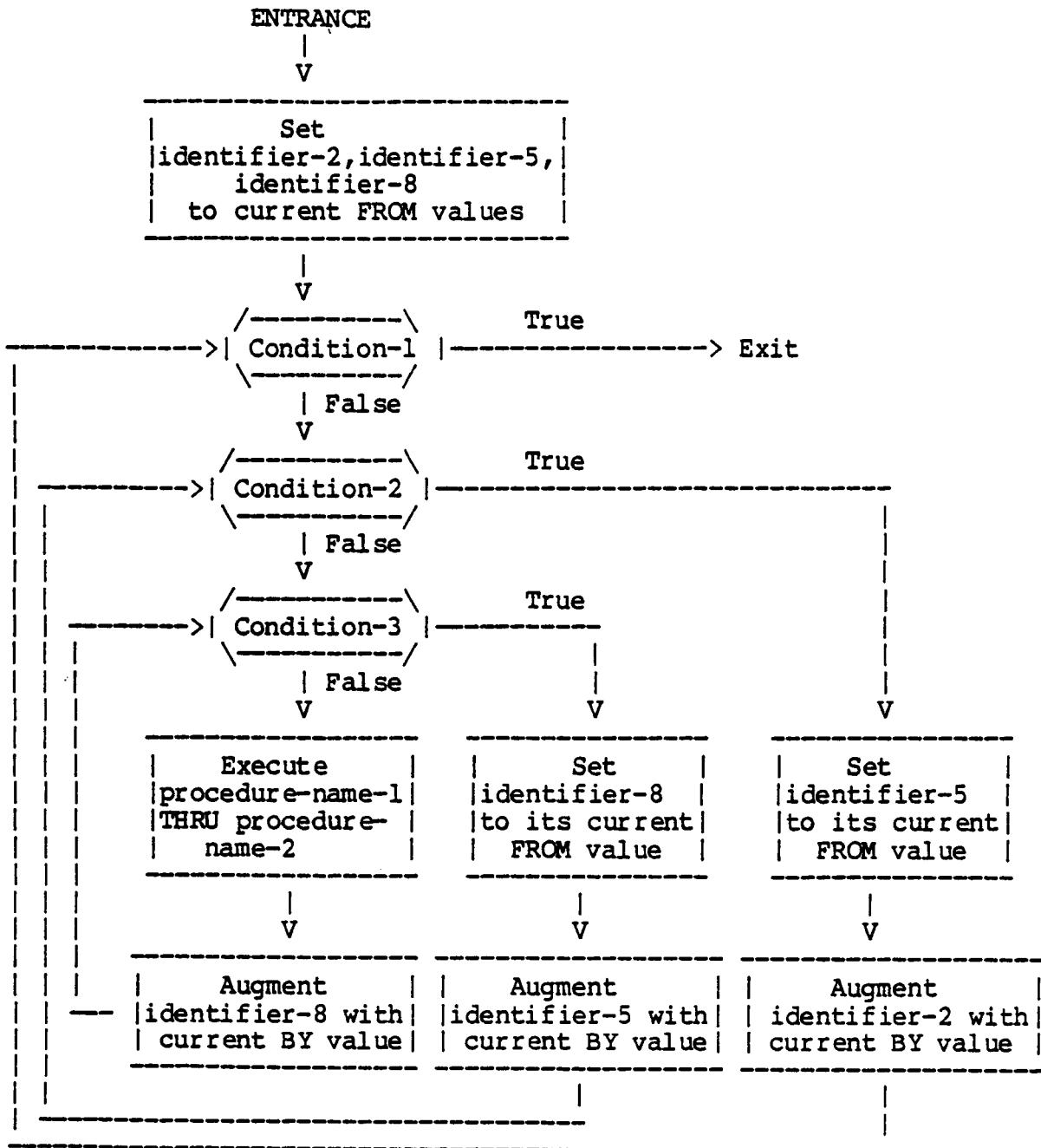
After the identifiers have been set, condition-1 is evaluated; if true, control is transferred to the next executable statement; if false, condition-2 is evaluated. If condition-2 is false, procedure-name-1 through procedure-name-2 is executed once, then identifier-5 is augmented by identifier-7 or literal-4 and condition-2 is evaluated again. This cycle of evaluation and augmentation continues until this condition is true. When condition-2 is true, identifier-5 is set to the value of literal-3 or the current value of identifier-6, identifier-2 is augmented by identifier-4 and condition-1 is re-evaluated. The PERFORM statement is completed if condition-1 is true; if not, the cycles continue until condition-1 is true.

During the execution of the procedures associated with the PERFORM statement, any change to the VARYING variable (identifier-2 and index-name-1), the BY variable (identifier-4), the AFTER variable (identifier-5 and index-name-3), or the FROM variable (identifier-3 and index-name-2) will be taken into consideration and will affect the operation of the PERFORM statement.



Flowchart for the VARYING Phrase of a PERFORM Statement Having Two Conditions.

PERFORM



Flowchart for the VARYING Phrase of a PERFORM Statement Having Three Conditions.

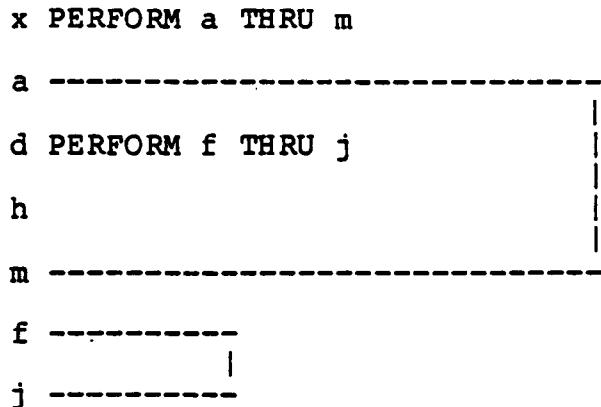
At the termination of the PERFORM statement identifier-5 contains the current value of identifier-6. Identifier-2 has a value that exceeds the last setting by an increment or decrement value, unless condition-1 was true when the PERFORM statement was entered, in which case identifier-2 contains the current value of identifier-3.

When two identifiers are varied, identifier-5 goes through a complete cycle (FROM, BY, UNTIL) each time identifier-2 is varied.

For three identifiers the mechanism is the same as for two identifiers except that identifier-8 goes through a complete cycle each time that identifier-5 is augmented by identifier-7 or literal-4, which in turn goes through a complete cycle each time identifier-2 is varied.

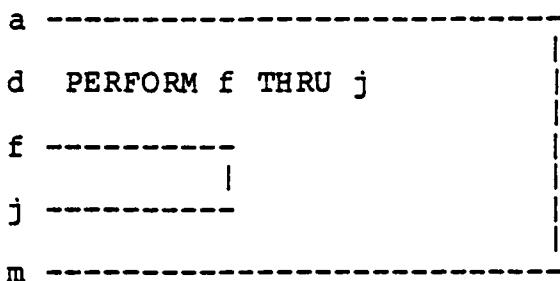
After the completion of a Format 4 PERFORM statement, identifier-5 and identifier-8 contain the current value of identifier-6 and identifier-9 respectively. Identifier-2 has a value that exceeds its last used setting by one increment or decrement value, unless condition-1 is true when the PERFORM statement is entered, in which case identifier-2 contains the current value of identifier-3.

If a sequence of statements referred to by a PERFORM statement includes another PERFORM statement, the sequence of procedures associated with the included PERFORM must itself either be totally included in, or totally excluded from, the logical sequence referred to by the first PERFORM. Thus an active PERFORM statement, whose execution point begins within the range of another active PERFORM statement, must not allow control to pass to the exit of the other active PERFORM statement; furthermore, two or more such active PERFORM statements may not have a common exit. See the valid illustrations below.

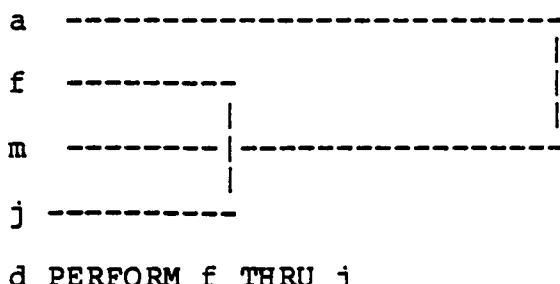


PERFORM

x PERFORM a THRU m



x PERFORM a THRU m



d PERFORM f THRU j

A PERFORM statement that appears in a section that is not in an independent segment can have within its range, in addition to any declarative sections whose execution is caused within that range, only one of the following:

Sections and/or paragraphs wholly contained in one or more non-independent segments.

Sections and/or paragraphs wholly contained in a single independent segment.

A PERFORM statement that appears in an independent segment can have within its range, in addition to any declarative sections whose execution is caused within that range, only one of the following:

Sections and/or paragraphs wholly contained in one or more non-independent segments.

Sections and/or paragraphs wholly contained in the same independent segment as the PERFORM statement.

The READ Statement (Sequential I-O)

The READ statement makes available the next logical record from a file.

FORMAT

```
READ file-name RECORD [INTO identifier]  
[;AT END imperative-statement]
```

The associated file must be open in the INPUT or I-O mode at the time this statement is executed.

The record to be made available by the READ statement is determined as follows:

If the current record pointer was positioned by the execution of the OPEN statement, the record pointed to by the current record pointer is made available.

If the current record pointer was positioned by the execution of a previous READ statement, the current record pointer is updated to point to the next existing record in the file and then that record is made available.

The execution of the READ statement causes the value of the FILE STATUS data item, if any, associated with file-name to be updated.

When the logical records of a file are described with more than one record description the contents of any data items which lie beyond the range of the current data record are undefined at the completion of the execution of the READ statement.

If, at the time of execution of a READ statement, the position of the current record pointer for that file is undefined, the execution of that READ statement is unsuccessful.

Following the unsuccessful execution of any READ statement, the contents of the associated record area and the position of the current record pointer are undefined.

The INTO Phrase

If the INTO phrase is specified, the record being read is moved from the record area to the area specified by identifier according to the rules specified for the MOVE statement. The implied MOVE does not occur if the execution of the READ statement was unsuccessful. Any subscripting or indexing associated with identifier is evaluated after the record has been read and immediately before it is moved to the data item.

When the INTO phrase is used, the record being read is available in both the input record area and the data area associated with identifier.

The INTO phrase must not be used when the input file contains logical records of various sizes as indicated by their record descriptions. The storage area associated with identifier and the record area associated with file-name must not be the same storage area.

The AT END Phrase

If, at the time of the execution of a READ statement, no next logical record exists in the file, the AT END condition occurs, and the execution of the READ statement is considered unsuccessful.

When the AT END condition is recognized the following actions are taken in the specified order.

A value is placed into the FILE STATUS data item, if specified for this file, to indicate an AT END condition.

If the AT END phrase is specified in the statement causing the condition, control is transferred to the AT END imperative-statement. Any USE procedure specified for this file is not executed.

If the AT END phrase is not specified, then a USE procedure must be specified, either explicitly or implicitly, for this file and that procedure is executed.

When the AT END condition has been recognized, a READ statement for that file must not be executed without first executing a successful CLOSE statement followed by the execution of a successful OPEN statement for that file.

The AT END phrase must be specified if no applicable USE procedure is specified for file-name.

The READ Statement (Relative and Indexed I-O)

The READ statement makes available a specified record from a mass storage file.

FORMAT 1

```
READ file-name [NEXT] RECORD [WITH NO LOCK] [INTO identifier]  
[;AT END imperative-statement]
```

FORMAT 2

```
READ file-name RECORD [WITH NO LOCK] [INTO identifier]  
[;KEY IS data-name]  
[;INVALID KEY imperative-statement]
```

Format 1 must be used for all files in sequential access mode.

The NEXT phrase must be specified for files in dynamic access mode, when records are to be retrieved sequentially.

Format 2 is used for files in random access mode or for files in dynamic access mode when records are to be retrieved randomly.

The INVALID KEY phrase or the AT END phrase must be specified if no applicable USE procedure is specified for file-name.

The associated files must be open in the INPUT or I-O mode at the time this statement is executed.

The KEY phrase may be specified only when the organization of file-name is index. When the KEY clause is present, data-name must be the name of one of the record keys associated with file-name. Data-name may be qualified.

READ (Relative and Indexed I-O)

The record to be made available by a Format 1 READ statement is determined as follows:

The record, pointed to by the current record pointer, is made available provided that the current record pointer was positioned by the START or OPEN statement and the record is still accessible through the path indicated by the current record pointer. If the record is no longer accessible, which may have been caused by the deletion of the record, the current record pointer is updated to point to the next existing record in the file and that record is then made available.

If the current record pointer was positioned by the execution of a previous READ statement, the current record pointer is updated to point to the next existing record in the file and then that record is made available.

The execution of the READ statement causes the value of the FILE STATUS data item, if any, associated with file-name to be updated.

When the logical records of a file are described with more than one record description, these records automatically share the same storage area; this is equivalent to an implicit redefinition of the area. The contents of any data items which lie beyond the range of the current data record are undefined at the completion of the execution of the READ statement.

If, at the time of execution of a Format 1 READ statement, the position of current record pointer for that file is undefined, the execution of that READ statement is unsuccessful.

The INTO Phrase

If the INTO phrase is specified, the record being read is moved from the record area to the area specified by identifier according to the rules specified for the MOVE statement. The implied MOVE does not occur if the execution of the READ statement was unsuccessful. Any subscripting or indexing associated with identifier is evaluated after the record has been read and immediately before it is moved to the data item.

When the INTO phrase is used, the record being read is available in both the input record area and the data area associated with identifier.

The INTO phrase must not be used when the input file contains logical records of various sizes as indicated by their record descriptions. The storage area associated with identifier and the record area associated with file-name must not be the same storage area.

Following the unsuccessful execution of any READ statement, the contents of the associated record area and the position of the current record pointer are undefined.

For relative files if the RELATIVE KEY phrase is specified, the execution of a Format 1 READ statement updates the contents of the RELATIVE KEY data item such that it contains the relative record number of the record made available.

For relative files the execution of a Format 2 READ statement sets the current record pointer to, and makes available, the record whose relative record number is contained in the data item named in the RELATIVE KEY phrase for the file. If the file does not contain such a record, the INVALID KEY condition exists and execution of the READ statement is unsuccessful.

For an indexed file being sequentially accessed, records having the same duplicate value in an alternate record key which is the key of reference are made available in the same order in which they are released by execution of WRITE statements, or by execution of REWRITE statements which create such duplicate values.

For an indexed file if the KEY phrase is specified in a Format 2 READ statement, data-name is established as the key of reference for this retrieval. If the dynamic access mode is specified, this key of reference is also used for retrievals by any subsequent executions of Format 1 READ statements for the file until a different key of reference is established for the file.

If the KEY phrase is not specified in a Format 2 READ statement, the prime record key is established as the key of reference for this retrieval.

If the dynamic access mode is specified, this key of reference is also used for retrievals by any subsequent executions of Format 1 READ statements for the file until a different key of reference is established for the file.

For indexed files the execution of a Format 2 READ statement causes the value of the key of reference to be compared with the value contained in the corresponding data item of the stored records in the file, until the first record having an equal value is found. The current record pointer is positioned to this record which is then made available. If no record can be so identified, the INVALID KEY condition exists and execution of the READ statement is unsuccessful.

The AT END Phrase

If, at the time of the execution of a Format 1 READ statement, no next logical record exists in the file, the AT END condition occurs, and the execution of the READ statement is considered unsuccessful.

When the AT END condition is recognized, the following actions are taken in the specified order:

A value is placed into the FILE STATUS data item, if specified for this file, to indicate an AT END condition.

If the AT END phrase is specified in the statement causing the condition, control is transferred to the AT END imperative-statement. Any USE procedure specified for this file is not executed.

If the AT END phrase is not specified, then a USE procedure must be specified, either explicitly or implicitly, for this file, and that procedure is executed.

When the AT END condition occurs, execution of the input-output statement which caused the condition is unsuccessful.

When the AT END condition has been recognized, a Format 1 READ statement for that file must not be executed without first executing one of the following:

A successful CLOSE statement followed by the execution of a successful OPEN statement for that file.

A successful START statement for that file.

A successful Format 2 READ statement for that file.

For a file for which dynamic access mode is specified, a Format 1 READ statement with the NEXT phrase specified causes the next logical record to be retrieved from the file.

The REWRITE Statement (Sequential I-O)

The REWRITE statement logically replaces a record existing in a mass storage file.

FORMAT

REWRITE record-name [FROM identifier]

Record-name and identifier must not refer to the same storage area.

Record-name is the name of a logical record in the File Section of the Data Division and may be qualified.

The file associated with record-name must be a mass storage file and must be open in the I-O mode at the time of execution of this statement.

The last input-output statement executed for the associated file prior to the execution of the REWRITE statement must have been a successfully executed READ statement.

The number of character positions in the record referenced by record-name must be equal to the number of character positions in the record being replaced.

The logical record released by successful execution of the REWRITE statement is no longer available in the record area.

The current record pointer is not affected by the execution of a REWRITE statement.

The execution of the REWRITE statement causes the value of the FILE STATUS data item, if any, associated with the file to be updated.

The FROM Phrase

The execution of a REWRITE statement with the FROM phrase is equivalent to the execution of:

MOVE identifier TO record-name

followed by the execution of the same REWRITE statement without the FROM phrase. The contents of the record area prior to the execution of the implicit MOVE statement have no effect on the execution of the REWRITE statement.

The REWRITE Statement (Relative and Indexed I-O)

The REWRITE statement logically replaces a record existing in a mass storage file.

FORMAT

```
REWRITE record-name [FROM identifier]  
[;INVALID KEY imperative-statement]
```

Record-name and identifier must not refer to the same storage area.

Record-name is the name of a logical record in the File Section of the Data Division and may be qualified.

For relative files the INVALID KEY phrase must not be specified for a REWRITE statement which references a file in sequential access mode.

The INVALID KEY phrase must be specified in the REWRITE statement for files in the random or dynamic access mode for which an appropriate USE procedure is not specified.

For indexed files the INVALID KEY phrase must be specified in the REWRITE statement for files for which an appropriate USE procedure is not specified.

The file associated with record-name must be open in the I-O mode at the time of execution of this statement.

For files in the sequential access mode, the last input-output statement executed for the associated file prior to the execution of the REWRITE statement must have been a successfully executed READ statement without the WITH NO LOCK phrase.

The number of character positions in the record referenced by record-name must be equal to the number of character positions in the record being replaced.

The logical record released by a successful execution of the REWRITE statement is no longer available in the record area.

The current record pointer is not affected by the execution of a REWRITE statement.

The execution of the REWRITE statement causes the value of the FILE STATUS data item, if any, associated with the file to be updated.

The INVALID KEY Phrase

For a relative file accessed in either random or dynamic access mode, the System logically replaces the record specified by the contents of the key data item associated with the file. If the file does not contain the record specified by the key, the INVALID KEY condition exists.

For indexed files the INVALID KEY condition exists when:

The access mode is sequential and the value contained in the prime record key data item of the record to be replaced is not equal to the value of the prime record read from the field, or

The value contained in the prime record key item does not equal that of any record stored in the file.

When the INVALID KEY condition exists the updating operation does not take place and the data in the record area is unaffected.

The FROM Phrase

The execution of a REWRITE statement with the FROM phrase is equivalent to the execution of:

MOVE identifier TO record-name

followed by the execution of the same REWRITE statement without the FROM phrase. The contents of the record area prior to the execution of the implicit MOVE statement have no effect on the execution of the REWRITE statement.

The SET statement

The SET statement establishes reference points for table handling operations by setting index-names associated with table elements.

FORMAT 1

```
SET {identifier-1} [,identifier-2] ... } TO {identifier-3}
                                         {index-name-3}
{index-name-1} [,index-name-2]           {integer-1  },
```

FORMAT 2

```
SET index-name-4 [,index-name-5] ... {UP BY } {identifier-4}
                                         {DOWN BY } {integer-2  }
```

All references to index-name-1, identifier-1, and index-name-4 apply equally to index-name-2, identifier-2, and index-name-5, respectively.

Identifier-1 and identifier-3 must name either index data items, or elementary items described as an integer.

Identifier-4 must be declared as an elementary numeric integer.

Integer-1 and integer-2 may be signed. Integer-1 must be positive.

Index-names are considered related to a given table and are defined by being specified in the INDEXED BY clause.

If index-name-3 is specified, the value of the index before the execution of the SET statement must correspond to an occurrence number of an element in the associated table.

If index-name-4, index-name-5 is specified, the value of the index both before and after the execution of the SET statement must correspond to an occurrence number of an element in the associated table. If index-name-1, index-name-2 is specified, the value of the index after the execution of the SET statement must correspond to an occurrence number of an element in the associated table. The value of the index associated with an index-name after the execution of a PERFORM statement may be undefined.

In Format 1, the following action occurs:

Index-name-1 is set to a value causing it to refer to the table element that corresponds in occurrence number to the table element referenced by index-name-3, identifier-3, or integer-1. If identifier-3 is an index data item, or if index-name-3 is related to the same table as index-name-1, no conversion takes place.

If identifier-1 is an index data item, it may be set equal to either the contents of index-name-3 or identifier-3 where identifier-3 is also an index data item; no conversion takes place in either case.

If identifier-1 is not an index data item, it may be set only to an occurrence number that corresponds to the value of index-name-3. Neither identifier-3 nor integer-1 can be used in this case.

The process is repeated for index-name-2, identifier-2, etc., if specified. Each time the value of index-name-3 or identifier-3 is used as it was at the beginning of the execution of the statement. Any subscripting or indexing associated with identifier-1, etc., is evaluated immediately before the value of the respective data item is changed.

In Format 2, the contents of index-name-4 are incremented (UP BY) or decremented (DOWN BY) by a value that corresponds to the number of occurrences represented by the value of integer-2 or identifier-4; thereafter, the process is repeated for index-name-5, etc. Each time the value of identifier-4 is used as it was at the beginning of the execution of the statement.

Data in the following chart represents the validity of various operand combinations in the SET statement.

		Receiving Item		
Sending Item	Integer Data Item	Index Name	Index Data Item	
Integer Literal	No	Valid	No	
Integer Data Item	No	Valid	No	
Index-Name	Valid	Valid	Valid*	
Index Data Item	No	Valid*	Valid*	

*No conversion takes place

START

The START Statement (Relative and Indexed I-O)

The START statement provides a basis for logical positioning within a file, for subsequent sequential retrieval of records.

FORMAT

```
. START file-name [KEY {IS EQUAL TO } data-name]  
          {IS = }  
          {IS GREATER THAN }  
          {IS > }  
          {IS NOT LESS THAN}  
          {IS NOT < }  
  
[;INVALID KEY imperative-statement]
```

Note: The required relational characters '>', '<' and '=' are not underlined to avoid confusion with other symbols.

File-name must be the name of a file with sequential or dynamic access.

Data-name may be qualified.

The INVALID KEY phrase must be specified if no applicable USE procedure is specified for file-name.

If file-name is the name of a relative file then data-name, if specified, must be the data item specified in the RELATIVE KEY phrase of the associated file control entry.

If file-name is the name of an indexed file then data-name, if specified, may reference the data items specified as the record keys associated with file-name or it may reference any data item of category alphanumeric whose leftmost character position corresponds to the leftmost character position of a record key data item.

File-name must be open in the INPUT or I-O mode at the time that the START statement is executed.

If the KEY phrase is not specified the relational operator 'IS EQUAL TO' is implied.

The type of comparison specified by the relational operator in the KEY phrase occurs between a key associated with a record in the file referenced by file-name and a data item.

If file-name references a relative file, the data item used in the comparison is the relative key associated with file-name.

If file-name references an indexed file, the data item used in the comparison is either the prime record key associated with file-name or, if the KEY phrase is specified, the data item referenced in the KEY phrase. If the operands of the comparison are of unequal size, comparison proceeds as though the longer one were truncated on the right such that its length is equal to that of the shorter. All other nonnumeric comparison rules apply except that the presence of the PROGRAM COLLATING SEQUENCE clause will have no effect on the comparison.

The current record pointer is positioned to the first logical record currently existing in the file whose key satisfies the comparison.

If the comparison is not satisfied by any record in the file, an INVALID KEY condition exists, the execution of the START statement is unsuccessful, and the position of the current record pointer is undefined.

The execution of the START statement causes the value of the FILE STATUS data item, if any, associated with file-name to be updated.

STOP

The STOP Statement

The **STOP** statement causes a permanent or temporary suspension of the execution of the object program.

FORMAT

```
STOP { RUN [,identifier-1 ] }  
      literal-1  
  
      { literal-2 }
```

The value of identifier-1 or literal-1 in the RUN phrase specifies a numeric value to be returned to the operating system upon execution of the **STOP RUN** statement. If identifier-1 or literal 1 is not specified in the RUN phrase, a value of 0 is assumed.

Literal-2 may be numeric or nonnumeric or may be any figurative constant.

If a **STOP RUN** statement appears in a consecutive sequence of imperative statements within a sentence, it must appear as the last statement in that sequence.

If the RUN phrase is used, then a **STOP RUN** message is logged and the execution is terminated with the specified return code.

If **STOP** literal is specified, the literal is logged in a **STOP "literal-value"** message and the execution is suspended.

STOP Examples

```
STOP RUN.  
STOP RUN 1.  
- STOP "END OF PROCEDURE".
```

The SUBTRACT Statement

The SUBTRACT statement is used to subtract one, or the sum of two or more, numeric data items from a numeric data item and store the result.

FORMAT 1

```
SUBTRACT {identifier-1} [,identifier-2] ...
    {literal-1 } [,literal-2 ]
FROM identifier-m [ROUNDED]
[;ON SIZE ERROR imperative-statement]
```

FORMAT 2

```
SUBTRACT {identifier-1} [,identifier-2] ...
    {literal-1 } [,literal-2 ]
FROM {identifier-m} GIVING identifier-n [ROUNDED]
    {literal-m }
[;ON SIZE ERROR imperative-statement]
```

FORMAT 3

```
SUBTRACT {CORRESPONDING} identifier-1
    {CORR }
FROM identifier-2 [ROUNDED]
[;ON SIZE ERROR imperative-statement]
```

In Format 1, all literals or identifiers preceding the word FROM are added together and this total is subtracted from the current value of identifier-m storing the result immediately into identifier-m.

In Format 2, all literals or identifiers preceding the word FROM are added together, the sum is subtracted from literal-m or identifier-m and the result of the subtraction is stored as the new value of identifier-n.

SUBTRACT

If Format 3 is used, data items in identifier-1 are subtracted from and stored into corresponding data items in identifier-2.

Each identifier must refer to a numeric elementary item except that:

In Format 2, the identifier following the word GIVING must refer to either an elementary numeric item or an elementary numeric edited item.

In Format 3, the identifiers must refer to group items.

Each literal must be a numeric literal.

The ROUNDED Phrase

The SUBTRACT statement may optionally include the ROUNDED phrase.

If, after decimal point alignment, the number of places in the fraction of the result of an arithmetic operation is greater than the number of places provided for the fraction of the resultant-identifier, truncation is relative to the size provided for the resultant-identifier. When rounding is requested, the absolute value of the resultant-identifier is increased by one (1) whenever the most significant digit of the excess is greater than or equal to five (5).

When the low-order integer positions in a resultant-identifier are represented by the character 'P' in the picture for that resultant-identifier, rounding or truncation occurs relative to the rightmost integer position for which storage is allocated.

The SIZE ERROR Phrase

If, after appropriate decimal point alignment, the absolute value of the result exceeds the largest value that can be contained in the associated resultant-identifier, a size error condition exists. If the ROUNDED phrase is specified, rounding takes place before checking for size error.

If the resultant-identifier has COMPUTATIONAL-3 usage, size error is detected only for data items declared with an odd length picture clause. Therefore, all COMP-3 data items should be declared with an odd number of character positions.

If the SIZE ERROR phrase is not specified and a size error condition exists, the value of the resultant-identifier is undefined.

If the SIZE ERROR phrase is specified and a size error condition exists, the value of the resultant-identifier(s) affected by the size error is not altered.

If the CORRESPONDING phrase is specified, and any of the individual subtractions produce a size error condition, the imperative-statement is not executed until all of the individual subtractions are completed.

The CORRESPONDING Phrase

If the CORRESPONDING phrase is used, selected items within identifier-1 are SUBTRACTed from, and the result stored in, the corresponding items in identifier-2. Data items referenced by the CORRESPONDING phrase must adhere to the following rules:

A data item in identifier-1 and a data item in identifier-2 must not be designated by the key word FILLER and must not have the same data-name and the same qualifiers up to, but not including, identifier-1 and identifier-2.

Both of the data items must be elementary numeric data items.

The description of identifier-1 and identifier-2 must not contain level-numbers 66, 77 or 88 or the USAGE IS INDEX clause.

A data item that is subordinate to identifier-1 or identifier-2 and contains a REDEFINES, RENAMES, OCCURS or USAGE IS INDEX clause is ignored, as well as those data items subordinate to the data item that contains the REDEFINES, OCCURS, or USAGE IS INDEX clause. However, identifier-1 and identifier-2 may have REDEFINES or OCCURS clauses or be subordinate to data items with REDEFINES or OCCURS clauses.

CORR is an abbreviation for CORRESPONDING.

SUBTRACT

SUBTRACT EXAMPLES

SUBTRACT TAXES FROM INCOME.

SUBTRACT 1 FROM TALLY GIVING TALLY-1.

SUBTRACT 2.68, INTEREST, PENALTY
FROM PRINCIPAL ROUNDED
ON SIZE ERROR GO TO ERROR-HANDLER.

The UNLOCK Statement

The UNLOCK statement makes available to other programs the most recently accessed record in a file that was read and locked.

FORMAT

UNLOCK file-name RECORD.

Note: The UNLOCK statement is nonstandard, but provides for compatibility with existing programs written for environments that allow multiple programs to concurrently update a data file. For systems that do not provide this capability, the UNLOCK statement will not affect execution except as described below.

The file associated with the file-name must be open in the I-O mode.

If no record in the file is locked, execution of an UNLOCK statement causes no action to be taken. If a record in the file is locked (unavailable to other programs), the last record to be locked is then made available to any other program upon execution of the UNLOCK statement.

The current record pointer is not affected by the execution of the UNLOCK statement. The FILE STATUS data item associated with the file, if one exists, is updated.

The UNLOCK statement may not be used to unlock records locked by other programs.

Note: Records that are read and locked are automatically unlocked by any subsequent operation on that file from the same program.

The WRITE Statement (Sequential I-O)

The WRITE statement releases a logical record for an output file. It can also be used for vertical positioning of lines within a logical page.

FORMAT

```
WRITE record-name [FROM identifier-1]  
[{BEFORE} ADVANCING {{identifier-2} [LINE ]}]  
{AFTER } { {integer } [LINES]}  
{ PAGE }
```

Record-name and identifier-1 must not reference the same storage area.

The record-name is the name of a logical record in the File Section of the Data Division and may be qualified.

When identifier-2 is used in the ADVANCING phrase, it must be the name of an elementary integer data item.

Integer or the value of the data item referenced by identifier-2 may be zero.

The associated file must be open in the OUTPUT or EXTEND mode at the time of the execution of this statement.

The logical record released by the execution of the WRITE statement is no longer available in the record area.

Upon completion of a WRITE statement, the information in the area referenced by identifier-1 is available even though the information in the area referenced by record-name may not be available.

The current record pointer is unaffected by the execution of a WRITE statement.

The execution of the WRITE statement causes the value of the FILE STATUS data item, if any, associated with the file to be updated.

The maximum record size for a file is established at the time the file is created and must not subsequently be changed.

The number of character positions on a mass storage device required to store a logical record in a file may or may not be equal to the number of character positions defined by the logical description of that record in the program.

The execution of the WRITE statement releases a logical record to the operating system. The contents of the record area are not changed.

When an attempt is made to write beyond the externally defined boundaries of a sequential file, an exception condition exists. The following action takes place:

The value of the FILE STATUS data item, if any, of the associated file is set to a value indicating a boundary violation.

If a USE AFTER STANDARD EXCEPTION declarative is explicitly or implicitly specified for the file, that declarative procedure will then be executed.

If a USE AFTER STANDARD EXCEPTION declarative is not explicitly or implicitly specified for the file, the result is undefined.

The FROM Phrase

The results of the execution of the WRITE statement with the FROM phrase is equivalent to the execution of the statement

MOVE identifier-1 TO record-name

according to the rules specified for the MOVE statement, followed by the same WRITE statement without the FROM phrase.

The contents of the record area prior to the execution of the implicit MOVE statement have no effect on the execution of this WRITE statement.

The ADVANCING Phrase

The ADVANCING phrase allows control of the vertical positioning of each line on a representation of a printed page. If the ADVANCING phrase is not used, automatic advancing will be provided by the compiler to act as if the user had specified AFTER ADVANCING 1 LINE. If the ADVANCING phrase is used, advancing is provided as follows:

If identifier-2 is specified, the representation of the printed page is advanced the number of lines equal to the current value associated with identifier-2.

If integer is specified, the representation of the printed page is advanced the number of lines equal to the value of integer.

If the BEFORE phrase is used, the line is presented before the representation of the printed page is advanced.

If the AFTER phrase is used, the line is presented after the representation of the printed page is advanced.

If PAGE is specified, the record is presented on the logical page before or after (depending on the phrase used) the device is repositioned to the next logical page.

The ADVANCING phrase is valid only if the device-type assigned to the file is PRINT.

THE WRITE STATEMENT (Relative and Indexed I-O)

The WRITE statement releases a logical record for an output or input-output file.

FORMAT

```
WRITE record-name [FROM identifier]
[;INVALID KEY imperative-statement]
```

Record-name and identifier must not reference the same storage area.

The record-name is the name of a logical record in the File Section of the Data Division and may be qualified.

The INVALID KEY phrase must be specified if an applicable USE procedure is not specified for the associated file.

The associated file must be open in the OUTPUT or I-O mode at the time of the execution of this statement.

The logical record released by the execution of the WRITE statement is no longer available in the record area.

The current record pointer is unaffected by the execution of a WRITE statement.

The execution of the WRITE statement causes the value of the FILE STATUS data item, if any, associated with the file to be updated.

The maximum record size for a file is established at the time the file is created and must not subsequently be changed.

The number of character positions on a mass storage device required to store a logical record in a file may or may not be equal to the number of character positions defined by the logical description of that record in the program.

The execution of the WRITE statement releases a logical record to the operating system.

WRITE (Relative and Indexed I-O)

When a relative file is opened in the output mode, records may be placed into the file by one of the following:

If the access mode is sequential, the WRITE statement will cause a record to be released to the System. The first record will have a relative record number of one (1) and subsequent records released will have relative record numbers of 2, 3, 4, If the RELATIVE KEY data item has been specified in the file control entry for the associated file, the relative record number of the record just released will be placed into the RELATIVE KEY data item by the System during execution of the WRITE statement.

If the access mode is random or dynamic, prior to the execution of the WRITE statement the value of the RELATIVE KEY data item must be initialized in the program with the relative record number to be associated with the record in the record area. That record is then released to the System by execution of the WRITE statement.

When a relative file is opened in the I-O mode and the access mode is random or dynamic, records are to be inserted in the associated file. The value of the RELATIVE KEY data item must be initialized by the program with the relative record number to be associated with the record in the record area. Execution of a WRITE statement then causes the contents of the record area to be released to the System.

For an indexed file, the data item specified as the prime record key must be set by the program to the desired value prior to the execution of the WRITE statement. Records may be placed into the file by one of the following:

If the access mode is sequential, records must be released to the System in ascending order of prime record key values.

If the access mode is random or dynamic, records may be released to the System in any program-specified order.

The FROM Phrase

The results of the execution of the WRITE statement with the FROM phrase is equivalent to the execution of the statement:

MOVE identifier-1 TO record-name
according to the rules specified for the MOVE statement, followed by the same WRITE statement without the FROM phrase.

The contents of the record area prior to the execution of the implicit MOVE statement have no effect on the execution of this WRITE statement.

The INVALID KEY Phrase

The INVALID KEY condition exists under the following circumstances:

When the access mode is sequential for an indexed file opened in the output mode, and the value of the prime record key is not greater than the value of the prime record key of the previous record, or

When an indexed file is opened in the output or I-O mode, and the value of the prime record key is equal to the value of a prime record key of a record already existing in the file, or

When a relative file has random or dynamic access mode and the RELATIVE KEY data item specifies a record which already exists in the file, or

When an attempt is made to write beyond the externally defined boundaries of the file.

When the INVALID KEY condition is recognized the execution of the WRITE statement is unsuccessful, the contents of the record area are unaffected and the FILE STATUS data item, if any, associated with file-name of the associated file is set to a value indicating the cause of the condition.

APPENDIX A

ERROR MESSAGES

ERROR MESSAGES (Compile Time)

The text of the source program is checked for syntax and semantic errors as it is scanned. Errors may cause interruption in scanning. In this case, text is ignored until a recovery point is found and a resume message is printed. Recovery points are chosen to minimize the amount of unanalyzed text without producing irrelevant error messages. In any case the constructs at fault are undermarked and error messages listed when the source line is printed. The error message includes either E's or W's indicating error or warning. For example:

indicates a semantic number size error but

indicates a syntax error at the first undermark and a recovery to the second undermark.

The number preceding the error message is the undermark number, counting from left to right. More than one message may refer to the same undermark.

Global errors such as undefined paragraph names and illegal control transfers are listed with the program summary at the end of the source listing.

Compilation always proceeds to the end of the program, regardless of the number of errors found. Object code is produced such that an attempt to execute an erroneous statement will terminate execution with an appropriate error message.

COMPILER ERROR MESSAGES

ACCESS CLASH

Nonsequential access given for sequential file.

BLANK WHEN ZERO

BLANK WHEN ZERO clause given for nonnumeric or group item.

CLASS

The referenced identifier is not valid in a class condition.

COPY

COPY statement failed because of permanent error associated with the undermarked file-name.

CORRESPONDING

The CORRESPONDING phrase cannot be used with the referenced identifier.

DATA OVERFLOW

The data area (working-storage and literals) is larger than 65535 bytes in length.

DATA TYPE

Context does not allow data type of the referenced identifier.

DEVICE CLASH

Random characteristics given to nonrandom device.

DEVICE TYPE

OPEN or CLOSE mode inconsistent with device type.

DOUBLE DECLARATION

Multiple declaration of a file or identifier attribute.

DOUBLE DEFINITION

Multiple definition of an identifier.

DUPLICATE

Warning only. Multiple USE procedure declared for same function or file.

FILE DECL. ERROR

The referenced file-name is SELECTed and has an invalid or missing file description (FD).

FILE NAME ERROR

The referenced file-name has an invalid external file name declaration.

FILE NAME REQUIRED

File name not given as reference in I/O verb.

FILE RECORD KEY ERROR

The referenced file-name has a RECORD KEY which is incorrectly qualified or is not defined as a data item of the category alphanumeric within a record description entry associated with that file name.

FILE RECORD SIZE ERROR

The referenced file-name has a declared record size which conflicts with the actual data record descriptions or is a relative organization file with variable length records.

FILE RELATIVE KEY ERROR

The referenced file-name has a RELATIVE KEY which is incorrectly qualified, is defined in a record description associated with that file-name, or is not defined as an unsigned integer.

FILE STATUS ERROR

The referenced file-name has a status item which is incorrectly qualified, is not defined in the WORKING-STORAGE SECTION, or is not a two-character alphanumeric item.

FILE TYPE

Access or organization of file conflicts with undermarked statement.

FILLER LEVEL

A nonelementary FILLER item is declared.

GROUP CLASH

USAGE or VALUE clause of group member conflicts with same clause for group.

GROUP VALUE CLASH

Warning only. An item subordinate to a group with the VALUE IS clause is described with the SYNCHRONIZED, JUSTIFIED, or USAGE (other than USAGE IS DISPLAY) clause.

IDENTIFIER

Identifier reference is incorrectly constructed or the identifier has an invalid or double definition.

ILLEGAL ALTER

An ALTER statement references an unalterable paragraph or violates the rules of segmentation.

ILLEGAL PERFORM

A PERFORM statement references undefined or incorrectly qualified paragraph or the reference violates the rules of segmentation.

INVALID ID

The referenced identifier was not successfully defined.

INVALID PARAGRAPH

Context does not allow section name.

JUSTIFY

JUSTIFY clause given in conflict with other attributes.

KEY REQUIRED

Relative key not declared for random access relative file or record key not declared for indexed file.

LABEL

Presence or absence of label record conflicts with device standards.

LEVEL

Level-number given is invalid either intrinsically or because of position within a group.

LINKAGE

An identifier in the USING clause of the PROCEDURE title is not a linkage item or a statement references a linkage item not subordinate to an identifier in the USING clause of the PROCEDURE title.

LITERAL VALUE

Literal value given is incorrect in context.

MOVE

Operands of MOVE verb specify an invalid move.

MUST BE INTEGER

Context requires decimal integer.

MUST BE PROCEDURE

Context requires procedure name either as reference or definition, or the reference must be a nondeclarative procedure-name.

MUST BE SECTION

Context requires procedure-name to be section.

NESTING

Illegal nesting of condition that is not an IF condition.

NOT IN REDEFINE

VALUE IS clause given in REDEFINES item.

OCCURS

Occurs clause given at invalid level or after three have been given for the same item.

OCCURS DEPENDING ERROR

The referenced object of a DEPENDING phrase has not been defined correctly.

OCCURS-VALUE CLASH

VALUE IS and OCCURS in effect for the same item.

PICTURE

Invalid picture syntax.

PICTURE-BWZ CLASH

Zero suppression and BLANK WHEN ZERO cannot be in effect for the same item.

PICTURE-USAGE CLASH

USAGE clause or implied usage conflicts with usage implied by picture.

PROCEDURE INDEPENDENCE

PERFORM given for procedures in independent segments not in the current segment.

PROGRAM OVERFLOW

The instruction area is larger than 32767 bytes in length.

RECORD KEY

Record key declared for other than an indexed organization file or a START statement KEY phrase references a data item not aligned on the declared key's leftmost byte.

RECORD REQUIRED

Context requires record name.

REDEFINES

REDEFINES given within an OCCURS or not redefining the last allocated item.

REDEFINES ERROR

The referenced data-name redefines an item which does not have the same number of character positions and is not level 01.

REFERENCE INVALID

Reference given is not valid in context.

RELATION

Operands of relation test are incompatible.

RELATIVE KEY

Relative key declared for other than a relative organization file or a START statement KEY phrase references a data item other than the declared key.

RESERVED WORD CONFLICT

A COBOL reserved word or symbol is given where a user word is required. In the summary this is only a warning about an ANSI COBOL reserved word that is not an implemented COBOL reserved word.

SCAN RESUME

Warning only. Scanning was terminated at previous error message and resumes at undermarked character.

SECTION CLASH

A VALUE IS clause appears in the FILE or LINKAGE section.

SEGMENT

Warning only. Segment number given in an independent segment is not the same as the current segment or the number of a new independent segment.

SEPARATOR

Warning only. Redundant punctuation or a separator is not followed by the required space.

SIGN

SIGN clause given in conflict with usage and picture.

SIZE

Warning only. Size of data referenced not correct for context.

SIZE ERROR

Declared size of record conflicts with present reference.

SUBSCRIPT

Incorrect number of subscripts or indices for a reference.

SYNC

Synchronized clause given for a group item.

SYNTAX

Incorrect character or reserved word given for context.

UNDEFINED

File referenced in FD entry was not defined.

UNDEFINED DECLARATIVE PROCEDURE

A declarative statement references a procedure not defined within the DECLARATIVES.

UNDEFINED PROCEDURE

A GO TO statement references an undefined or incorrectly qualified paragraph.

USE REQUIRED

A DECLARATIVES section must begin with a USE statement.

USING COUNT

Warning only. The item count in the USING list of a CALL statement is different from that of the first reference to the same program name.

VALUE ERROR

Value given in VALUE IS required truncation of nonzero digits.

VALUE

VALUE IS clause given in conflict with other declared attributes.

VARIABLE RECORD

Warning only. The INTO phrase is not allowed with variable size records.

APPENDIX B

RESERVED WORDS

RESERVED WORD LIST

The following is a list of RM/COBOL reserved words where:

- * denotes reserved words not reserved in ANSI standard COBOL
- + denotes ANSI COBOL reserved words not reserved by the compiler. Their appearance will generate a warning at the end of the compilation listing.
- ** denotes system-name.

ACCEPT	ALPHABETIC	AREA
ACCESS	+ALSO	+AREAS
ADD	ALTER	+ASCENDING
ADVANCING	ALTERNATE	ASSIGN
AFTER	AND	AT
ALL	ARE	AUTHOR
*BEEP	*BLINK	BY
BEFORE	BLOCK	
BLANK	+BOTTOM	
CALL	+CODE-SET	COMPUTE
+CANCEL	COLLATING	CONFIGURATION
+CD	+COLUMN	CONTAINS
+CF	COMMA	+CONTROL
+CH	+COMMUNICATION	+CONTROLS
CHARACTER	COMP	*CONVERT
CHARACTERS	*COMP-1	COPY
+CLOCK-UNITS	*COMP-3	CORR
CLOSE	COMPUTATIONAL	CORRESPONDING
+COBOL	*COMPUTATIONAL-1	+COUNT
+CODE	*COMPUTATIONAL-3	CURRENCY
DATA	+DEBUG-SUB-1	+DESCENDING
DATE	+DEBUG-SUB-2	+DESTINATION
+DATE-COMPILED	+DEBUG-SUB-3	+DETAIL
DATE-WRITTEN	+DEBUGGING	+DISABLE
DAY	DECIMAL-POINT	DISPLAY
+DE	DECLARATIVES	DIVIDE
+DEBUG-CONTENTS	DELETE	DIVISION
+DEBUG-ITEM	+DELIMITED	DOWN
+DEBUG-LINE	+DELIMITER	DUPLICATES
+DEBUG-NAME	DEPENDING	DYNAMIC

Appendix B

*ECHO	+END-OF-PAGE	ERROR
+EG I	+ENTER	+ESI
ELSE	ENVIRONMENT	+EVERY
+EMI	+EOP	EXCEPTION
+ENABLE	EQUAL	EXIT
END	*ERASE	EXTEND
FD	FILLER	+FOOTING
FILE	+FINAL	FOR
FILE-CONTROL	FIRST	FROM
+GENERATE	GO	+GROUP
GIVING	GREATER	
+HEADING	HIGH-VALUE	
*HIGH	HIGH-VALUES	
I-O	INDEXED	INSPECT
I-O-CONTROL	+INDICATE	INSTALLATION
IDENTIFICATION	INITIAL	INTO
IF	+INITIATE	INVALID
IN	INPUT	IS
INDEX	INPUT-OUTPUT	
JUST	JUSTIFIED	
KEY		
LABEL	+LIMIT	LINES
+LAST	+LIMITS	LINKAGE
LEADING	+LINAGE	LOCK
LEFT	+LINAGE-COUNTER	LOW
+LENGTH	LINE	LOW-VALUE
LESS	+LINE-COUNTER	LOW-VALUES
MEMORY	MODE	+MULTIPLE
+MERGE	MODULES	MULTIPLY
+MESSAGE	MOVE	
NATIVE	NO	NUMERIC
+NEGATIVE	NOT	
NEXT	+NUMBER	

OBJECT-COMPUTER	OMITTED	OR
OCCURS	ON	ORGANIZATION
OF	OPEN	OUTPUT
OFF	+OPTIONAL	+OVERFLOW
PAGE	+PLUS	+PROCEDURES
+PAGE-COUNTER	+POINTER	PROCEED
PERFORM	POSITION	PROGRAM
+PF	+POSITIVE	PROGRAM-ID
+PH	*PRINT	*PROMPT
PIC	+PRINTING	
PICTURE	PROCEDURE	
+QUEUE	QUOTE	QUOTES
RANDOM	+REMAINDER	*REVERSE
+RD	+REMOVAL	+REVERSED
READ	RENAMES	REWIND
+RECEIVE	REPLACING	REWRITE
RECORD	+REPORT	+RF
RECORDS	+REPORTING	+RH
REDEFINES	+REPORTS	RIGHT
REEL	+RERUN	ROUNDED
+REFERENCES	+RESERVE	RUN
RELATIVE	+RESET	
+RELEASE	+RETURN	
SAME	SIZE	+SUB-QUEUE-2
+SD	+SORT	+SUB-QUEUE-3
+SEARCH	+SORT-MERGE	SUBTRACT
SECTION	+SOURCE	+SUM
SECURITY	SOURCE-COMPUTER	+SUPPRESS
+SEGMENT	SPACE	**SWITCH-1
+SEGMENT-LIMIT	SPACES	**SWITCH-2
SELECT	SPECIAL-NAMES	'
+SEND	STANDARD	'
SENTENCE	STANDARD-1	**SWITCH-8
SEPARATE	START	+SYMBOLIC
SEQUENCE	STATUS	SYNC
SEQUENTIAL	STOP	SYNCHRONIZED
SET	+STRING	
SIGN	+SUB-QUEUE-1	

Appendix B

*TAB	+TEXT	TO
+TABLE	THAN	+TOP
TALLYING	THROUGH	TRAILING
+TAPE	THRU	+TYPE
+TERMINAL	TIME	
+TERMINATE	TIMES	
UNIT	UNTIL	USAGE
*UNLOCK	UP	USE
+UNSTRING	+UPON	USING
VALUE	VALUES	VARYING
WHEN	WORDS	WRITE
WITH	WORKING-STORAGE	
ZERO	ZEROES	ZEROS
+	>	*
-	<	/
=	,	**

APPENDIX C

GLOSSARY

GLOSSARY

The terms in this appendix are defined in accordance with their meaning as used in this document describing COBOL and may not have the same meaning for other languages.

These definitions are also intended to be either reference material or introductory material to be reviewed prior to reading the detailed language specifications. For this reason, these definitions are, in most instances, brief and do not include detailed syntactical rules.

Access Mode:

The manner in which records are to be operated upon within a file.

Actual Decimal Point:

The physical representation, using either of the decimal point characters period (.) or comma (,), of the decimal point position in a data item.

Alphabet-Name:

A user-defined word, in the SPECIAL-NAMES paragraph of the Environment Division, that assigns a name to a specific character set and/or collating sequence.

Alphabetic Character:

A character that belongs to the following set of letters: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, and the space.

Alphanumeric Character:

Any character in the computer's character set.

Alternate Record Key:

A key, other than the prime record key, whose contents identify a record within an indexed file.

Arithmetic Expression:

An arithmetic expression can be an identifier or a numeric elementary item, a numeric literal, such identifiers and literals separated by arithmetic operators, two arithmetic expressions separated by an arithmetic operator, or an arithmetic expression enclosed in parentheses.

Arithmetic Operator:

A single character that belongs to the following set:

<u>Character</u>	<u>Meaning</u>
+	addition
-	subtraction
*	multiplication
/	division

Ascending Key:

A key upon the values of which data is ordered starting with the lowest value of key up to the highest value of key in accordance with the rules for comparing data items.

Assumed Decimal Point:

A decimal point position which does not involve the existence of an actual character in a data item. The assumed decimal point has logical meaning but no physical representation.

At End Condition:

A condition caused during the execution of a READ statement for a sequentially accessed file.

Block:

A physical unit of data that is normally composed of one or more logical records. For mass storage files, a block may contain a portion of a logical record. The size of a block has no direct relationship to the size of the file within which the block is contained or to the size of the logical record(s) that are either continued within the block or that overlap the block. The term is synonymous with physical record.

Called Program:

A program which is the object of a CALL statement combined at object time with the calling program to produce a run unit.

Calling Program:

A program which executes a CALL to another program.

Character:

The basic indivisible unit of the language.

Character Position:

A character position is the amount of physical storage required to store a single standard data format character described as USAGE is DISPLAY (one byte).

Character-String:

A sequence of contiguous characters which form a COBOL word, a literal, a PICTURE character-string, or a comment-entry.

Class Condition:

The proposition, for which a truth value can be determined, that the content of an item is wholly alphabetic or is wholly numeric.

Clause:

A clause is an ordered set of consecutive COBOL character-strings whose purpose is to specify an attribute of an entry.

COBOL Character Set:

The complete COBOL character set consists of the 51 characters listed below.

<u>Character</u>	<u>Meaning</u>
0,1,...,9	digit
A,B,...,Z	letter
	space (blank)
+	plus sign
-	minus sign (hyphen)
*	asterisk
/	stroke (virgule, slash)
=	equal sign
\$	currency sign
,	comma (decimal point)
;	semicolon
.	period (decimal point)
"	quotation mark
(left parenthesis
)	right parenthesis
>	greater than symbol
<	less than symbol

COBOL Word. (See Word)**Collating Sequence:**

The sequence in which the characters that are acceptable in a computer are ordered for purposes of comparing.

Column:

A character position within a print line. The columns are numbered from 1, by 1, starting at the leftmost character position of the print line and extending to the rightmost position of the print line.

Combined Condition:

A condition that is the result of connecting two or more conditions with the 'AND' or the 'OR' logical operator.

Comment-Entry:

An entry in the Identification Division that may be any combination of characters from the computer character set.

Comment Line:

A source program line represented by an asterisk in the indicator area of the line and any characters from the computer's character set in area A and area B of that line. The comment line serves only for documentation in a program. A special form of comment line represented by a stroke (/) in the indicator area of the line and any characters from the computer's character set in area A and area B of that line causes page ejection prior to printing the comment.

Compile-Time:

The time at which a COBOL source program is translated, by a COBOL compiler, to a COBOL object program.

Compiler Directing Statement:

A statement, beginning with a compiler directing verb, that causes the compiler to take a specific action during compilation.

Complex Condition:

A condition in which one or more logical operators act upon one or more conditions.

Computer-Name:

A system-name that identifies the computer upon which the program is to be compiled or run (commentary only).

Condition:

A status of a program at execution time for which a truth value can be determined. Where the term 'condition' (condition-1, condition-2, ...) appears in these language specifications in or in reference to 'condition' (condition-1, condition-2, ...) of a general format, it is a conditional expression consisting of a simple condition, optionally parenthesized, consisting of the syntactically correct combination of simple conditions, logical operators, and parentheses, for which a truth value can be determined.

Condition-Name:

A user-defined word assigned to a specific value, set of values, or range of values, within the complete set of values that a conditional variable may possess; or the user-defined word assigned to a status of a system software switch.

Condition-Name Condition:

The proposition, for which a truth value can be determined, that the value of a conditional variable is a member of the set of values attributed to a condition-name associated with the conditional variable.

Conditional Expression:

A simple condition or a complex condition specified in an IF or PERFORM statement.

Conditional Statement:

A conditional statement specifies that the truth value of a condition is to be determined and that the subsequent action of the object program is dependent on this truth value.

Conditional Variable:

A data item one or more values of which has a condition name assigned to it.

Configuration Section:

A section of the Environment Division that describes overall specifications of source and object computers.

Connective:

A reserved word that is used to:

Associate a data-name, paragraph-name or condition-name with its qualifier.

Link two or more operands written in a series.

Form conditions (logical connectives).

Contiguous Items:

Items that are described by consecutive entries in the Data Division, and that bear a definite hierachic relationship to each other.

Counter:

A data item used for storing numbers or number representations in a manner that permits these numbers to be increased or decreased by the value of another number, or to be changed or reset to zero or to an arbitrary positive or negative value.

Currency Sign:

The character '\$' of the COBOL character set.

Currency Symbol:

The character defined by the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph. If no CURRENCY SIGN clause is present in a COBOL source program, the currency symbol is identical to the currency sign.

Current Record:

The record which is available in the record area associated with the file.

Current Record Pointer:

A conceptual entity that is used in the selection of the next record.

Data Clause:

A clause that appears in a data description entry in the Data Division and provides information describing a particular attribute of a data item.

Data Description Entry:

An entry in the Data Description that is composed of a level-number followed by a data-name, if required, and then followed by a set of data clauses, as required.

Data Item:

A character or a set of contiguous characters (excluding in either case literals) defined as a unit of data by the COBOL program.

Data-Name:

A user-defined word that names a data item described in a data description entry in the Data Division. When used in the general formats, 'data-name' represents a word which can neither be subscripted, indexed, nor qualified unless specifically permitted by the rules for that format.

Debugging Line:

A debugging line is any line with 'D' in the indicator area of the line.

Declaratives:

A set of one or more special purpose sections, written at the beginning of the Procedure Division, the first of which is preceded by the key word DECLARATIVES and the last of which is followed by the key words END DECLARATIVES. A declarative is composed of a section header, followed by a USE compiler directing sentence, followed by a set of zero, one or more associated paragraphs.

Declarative-Sentence:

A compiler-directing sentence consisting of a single USE statement terminated by the separator period.

Delimiter:

A character or a sequence of contiguous characters that identify the end of a string of characters and separates that string of characters from the following string of characters. A delimiter is not part of the string of characters that it delimits.

Digit Position:

A digit position is the amount of physical storage required to store a single digit. This amount may vary depending on the usage of the data item describing the digit position.

Division:

A set of zero, one or more sections of paragraphs, called the division body, that are formed and combined in accordance with a specific set of rules. There are four (4) divisions in a COBOL program: Identification, Environment, Data, and Procedure.

Division Header:

A combination of words followed by a period and a space that indicates the beginning of a division. The division headers are:

IDENTIFICATION DIVISION.

ENVIRONMENT DIVISION.

DATA DIVISION.

PROCEDURE DIVISION [USING data-name-1 [data-name-2]...].

Dynamic Access:

An access mode in which specific logical records can be obtained from or placed into a mass storage file in a non sequential manner (see Random Access) and obtained from a file in a sequential manner (see Sequential Access), during the scope of the same OPEN statement.

Editing Character:

A single character or fixed two-character combination belonging to the following set:

<u>Character</u>	<u>Meaning</u>
B	space
0	zero
+	plus
-	minus
CR	credit
DB	debit
Z	zero suppress
*	check protect
\$	currency sign
,	comma (decimal point)
.	period (decimal point)
/	stroke (virgule, slash)

Elementary Item:

A data item that is described as not being further logically subdivided.

End of Procedure Division:

The physical position in a COBOL source program after which no further procedures appear.

Entry:

Any descriptive set of consecutive clauses terminated by a period and written in the Identification Division, Environment Division, or Data Division of a COBOL source program.

Environment Clause:

A clause that appears as part of an Environment Division entry.

Execution Time. (See Object Time)**Extend Mode:**

The state of a file after execution of an OPEN statement, with the EXTEND phrase specified, for that file and before the execution of a CLOSE statement for that file.

Figurative Constant:

A compiler generated value referenced through the use of certain reserved words.

File:

A collection of records.

File Clause:

A clause that appears as part of the file description (FD) entries in the Data Division.

FILE-CONTROL:

The name of an Environment Division paragraph in which the data files for a given source program are declared.

File Description Entry:

An entry in the File Section of the Data Division that is composed of the level indicator FD, followed by a file-name, and then followed by a set of file clauses as required.

Appendix C

File-Name:

A user-defined word that names a file described in a file description entry within the File Section of the Data Division.

File Organization:

The permanent logical file structure established at the time that a file is created.

File Section:

The section of the Data Division that contains file description entries together with their associated record descriptions.

Format:

A specific arrangement of a set of data.

Group Item:

A named contiguous set of elementary or group items.

I-O-CONTROL:

The name of an Environment Division paragraph in which sharing of same areas by several data files is specified.

I-O-Mode:

The state of a file after execution of an OPEN statement, with the I-O phrase specified, for that file and before the execution of a CLOSE statement for that file.

Identifier:

A data-name, followed as required, by the syntactically correct combination of qualifiers, subscripts, and indices necessary to make unique reference to a data item.

Imperative Statement:

A statement that begins with an imperative verb and specifies an unconditional action to be taken. An imperative statement may consist of a sequence of imperative statements.

Index:

A data item, the contents of which represent the identification of a particular element in a table.

Index Data Item:

A data item in which the value associated with an index-name can be stored.

Index-Name:

A user-defined word that names an index associated with a specific table.

Indexed Data-Name:

An identifier that is composed of a data-name, followed by one or more index-names enclosed in parentheses.

Indexed File:

A file with indexed organization.

Indexed Organization:

The permanent logical file structure in which each record is identified by the value of one fixed length key within that record.

Input File:

A file that is opened in the input mode.

Input Mode:

The state of a file after execution of an OPEN statement, with the INPUT phrase specified, for that file and before the execution of a CLOSE statement for that file.

Input-Output File:

A file that is opened in the I-O mode.

Input-Output Section:

The section of the Environment Division that names the files and the external media required by an object program and which provides information required for transmission and handling of data during execution of the object program.

Integer:

A numeric literal or a numeric data item that does not include any character positions to the right of the assumed decimal point. Where the term 'integer' appears in general formats, integer must not be a numeric data item, and must not be signed, nor zero, unless explicitly allowed by the rules of that format.

Invalid Key Condition:

A condition, at object time, caused when a specific value of the key associated with an indexed or relative file is determined to be invalid.

Key:

A data item which identifies the location of a record.

Key Word:

A reserved word whose presence is required when the format in which the word appears is used in a source program.

Level Indicator:

Two alphabetic characters that identify a specific type of file or a position in hierarchy.

Level-Number:

A user-defined word which indicates the position of a data item in the hierarchical structure of a logical record or which indicates special properties of a data description entry. A level-number is expressed as a one- or two-digit number. Level-numbers in the range 1 through 49 indicate the position of a data item in the hierarchical structure of a logical record. Level-numbers in the range 1 through 9 may be written either as a single digit or as a zero followed by a significant digit. Level-numbers 77 and 88 identify special properties of a data description entry.

Library-Name:

A user-defined word that names a COBOL library that is to be used by the compiler for a given source program compilation.

Linkage Section:

The section in the Data Division of the called program that describes the data items available from the calling program. These data items may be referred to by both the calling and called program.

Literal:

A character-string whose value is implied by the ordered set of characters comprising the string.

Logical Operator:

One of the reserved words AND, OR, or NOT. In the formation of a condition, both or neither of AND and OR can be used as logical connectives. NOT can be used for logical negation.

Mass Storage:

A storage medium on which data may be organized and maintained in both a sequential and nonsequential manner.

Mass Storage File:

A collection of records that is assigned to a mass storage medium.

Mnemonic-Name:

A user-defined word that is associated in the Environment Division with a specified system-name.

Native Character Set:

The character set associated with the COBOL Compiler (ASCII).

Native Collating Sequence:

The collating sequence associated with the native character set.

Negated Combined Condition:

The 'NOT' logical operator immediately followed by a parenthesized combined condition.

Negated Simple Condition:

The 'NOT' logical operator immediately followed by a simple condition.

Next Executable Sentence:

The next sentence to which control will be transferred after execution of the current statement is complete.

Next Record:

The record which logically follows the current record of a file.

Noncontiguous Items:

Elementary data items, in the Working-Storage and Linkage Sections, which bear no hierachic relationship to other data items.

Nonnumeric Item:

A data item whose description permits its contents to be composed of any combination of characters taken from the computer's character set. Certain categories of nonnumeric items may be formed from more restricted character sets.

Nonnumeric Literal:

A character-string bounded by quotation marks. The string of characters may include any character in the computer's character set. To represent a single quotation mark character within a nonnumeric literal, two contiguous quotation marks must be used.

Numeric Character:

A character that belongs to the following set of digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Numeric Item:

A data item whose description restricts its contents to a value represented by characters chosen from the digits '0' through '9'; if signed, the item may also contain a '+', '-', or other representation of an operational sign.

Numeric Literal:

A literal composed of one or more numeric characters that also may contain either a decimal point, or an algebraic sign, or both. The decimal point must not be the rightmost character. The algebraic sign, if present, must be the leftmost character.

OBJECT-COMPUTER:

The name of an Environment Division paragraph in which the computer environment, within which the object program is executed, is described.

Object of Entry:

A set of operands and reserved words, within a Data Division entry, that immediately follows the subject of the entry.

Object Program:

A set or group of executable instructions and other material designed to interact with data to provide problem solutions. In this context, an object program is generally the result of the operation of a COBOL compiler on a source program. Where there is no danger of ambiguity, the word 'program' alone may be used in place of the phrase 'object program'.

Object Time:

The time at which an object program is executed.

Open Mode:

The state of a file after execution of an OPEN statement for that file and before the execution of a CLOSE statement for that file. The particular open mode is specified in the OPEN statement as either INPUT, OUTPUT, I-O, or EXTEND.

Occurrence Number:

The relative data item number in a table.

Operand:

Whereas the general definition of operand is 'that component which is operated upon', for the purposes of this publication, any lowercase word (or words) that appears in a statement or entry format may be considered to be an operand and, as such, is an implied reference to the data indicated by the operand.

Operational Sign:

An algebraic sign, associated with a numeric data item or a numeric literal, to indicate whether its value is positive or negative.

Optional Word:

A reserved word that is included in a specific format only to improve the readability of the language and whose presence is optional to the user when the format in which the word appears is used in a source program.

Output File:

A file that is opened in either the output mode or extend mode.

Output Mode:

The state of a file after execution of an OPEN statement, with the OUTPUT or EXTEND phrase specified, for that file and before the execution of a CLOSE statement for that file.

Paragraph:

In the Procedure Division, a paragraph-name followed by a period and a space and by zero, one, or more sentences. In the Identification and Environment Divisions, a paragraph header followed by zero, one, or more entries.

Paragraph Header:

A reserved word, followed by a period and a space that indicates the beginning of a paragraph in the Identification and Environment Divisions. The permissible paragraph headers are:

In the Identification Division:

PROGRAM-ID.
AUTHOR.
INSTALLATION.
DATE-WRITTEN.
SECURITY.

In the Environment Division:

SOURCE-COMPUTER.
OBJECT-COMPUTER.
SPECIAL-NAMES.
FILE-CONTROL.
I-O-CONTROL.

Paragraph-Name:

A user-defined word that identifies and begins a paragraph in the Procedure Division.

Phrase:

A phrase is an ordered set of one or more consecutive COBOL character-strings that form a portion of a COBOL procedural statement or of a COBOL clause.

Physical Record. (See Block)**Prime Record Key:**

A key whose contents uniquely identify a record within an indexed file.

Procedure:

A paragraph or group of logically successive paragraphs, or a section or group of logically successive sections, within the Procedure Division.

Procedure-Name:

A user-defined word which is used to name a paragraph or section in the Procedure Division. It consists of a paragraph-name (which may be qualified), or a section-name.

Program-Name:

A user-defined word that identifies a COBOL source program.

Punctuation Character:

A character that belongs to the following set:

<u>Character</u>	<u>Meaning</u>
,	comma
;	semicolon
.	period
"	quotation mark
(left parenthesis
)	right parenthesis
	space
=	equal sign

Qualified Data-Name:

An identifier that is composed of a data-name followed by one or more sets of either of the connectives OF and IN followed by a data-name qualifier.

Qualifier:

A data-name which is used in a reference together with another data name at a lower level in the same hierarchy. A section-name which is used in a reference together with a paragraph-name specified in that section.

Random Access:

An access mode in which the program-specified value of a key data item identifies the logical record that is obtained from, deleted from, or placed into a relative or indexed file.

Record Area:

A storage area allocated for the purpose of processing the record described in a record description entry in the File Section.

Record Description. (See Record Description Entry)

Record Description Entry:

The total set of data description entries associated with a particular record.

Record Key:

The prime record key whose contents uniquely identify a record within an indexed file.

Record-Name:

A user-defined word that names a record described in a record description entry in the Data Division.

Reference Format:

A format that provides a standard method for describing COBOL source programs.

Relation. (See Relational Operator)

Relation Character:

A character that belongs to the following set:

<u>Character</u>	<u>Meaning</u>
>	greater than
<	less than
=	equal to

Relation Condition:

The proposition, for which a truth value can be determined, that the value of a data item has a specific relationship to the value of another data item. (See Relational Operator)

Relational Operator:

A reserved word, a relation character, a group of consecutive reserved words, or a group of consecutive reserved words and relation characters used in the construction of a relation condition. The permissible operators and their meanings are:

<u>Relational Operator</u>	<u>Meaning</u>
IS [NOT] GREATER THAN	Greater than or not greater than
IS [NOT] >	
IS [NOT] LESS THAN	Less than or not less than
IS [NOT] <	
IS [NOT] EQUAL TO	Equal to or not equal to
IS [NOT] =	

Relative File:

A file with relative organization.

Relative Key:

A key whose contents identifies a logical record in a relative file.

Relative Organization:

The permanent logical file structure in which each record is uniquely identified by an integer value greater than zero, which specifies the record's logical ordinal position in the file.

Reserved Word:

A COBOL word specified in the list of words which may be used in COBOL source programs, but which must not appear in the programs as user-defined words or system-names.

Run Unit:

A set of one or more object programs which function at object time, as a unit to provide problem solutions.

Section:

A set of zero, one, or more paragraphs or entries, called a section body, the first of which is preceded by a section header. Each section consists of the section header and the related section body.

Section Header:

A combination of words followed by a period and a space that indicates the beginning of a section in the Environment, Data and Procedure Division.

In the Environment and Data Divisions, a section header is composed of reserved words followed by a period and a space. The permissible section headers are:

In the Environment Division:

CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.

In the Data Division:

FILE SECTION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.

In the Procedure Division, a section header is composed of a section-name, followed by the reserved word SECTION, followed by a segment-number (optional), followed by a period and a space.

Section-Name:

A user-defined word which names a section in the Procedure Division.

Segment-Number:

A user-defined word which classifies sections in the Procedure Division for purposes of segmentation. Segment-numbers may contain only the characters '0', '1',..., '9'. A segment-number may be expressed either as a one- or two-digit number.

Sentence:

A sequence of one or more statements, the last of which is terminated by a period followed by a space.

Separator:

A punctuation character used to delimit character-strings.

Sequential Access:

An access mode in which logical records are obtained from or placed into a file in a consecutive predecessor-to-successor logical record sequence determined by the order of records in the file.

Sequential File:

A file with sequential organization.

Sequential Organization:

The permanent logical file structure in which a record is identified by a predecessor-successor relationship established when the record is placed into the file.

Simple Condition:

Any single condition chosen from the set:

- relation condition
- class condition
- condition-name condition
- switch-status condition
- (simple-condition)

SOURCE-COMPUTER:

The name of an Environment Division paragraph in which the computer environment, within which the source program is compiled, is described.

Source Program:

A syntactically correct set of COBOL statements beginning with an Identification Division and ending with the end of the Procedure Division. In contexts where there is no danger of ambiguity, the word 'program' alone may be used in place of the phrase 'source program.'

Special Character:

A character that belongs to the following set:

<u>Character</u>	<u>Meaning</u>
+	plus sign
-	minus sign
*	asterisk
/	stroke (virgule, slash)
=	equal sign
\$	currency sign
,	comma (decimal point)
;	semicolon
.	period (decimal point)
"	quotation mark
(left parenthesis
)	right parenthesis
>	greater than symbol
<	less than symbol

Special-Character Word:

A reserved word which is an arithmetic operator or a relation character.

SPECIAL-NAMES:

The name of an Environment Division paragraph in which switch-names are related to user-defined words.

Standard Data Format:

The concept used in describing the characteristics of data in a COBOL Data Division under the characteristics or properties of the data are expressed in a form oriented to the appearance of the data on a printed page of infinite length and breadth, rather than a form oriented to the manner in which the data is stored internally in the computer, or on a particular external medium.

Statement:

A syntactically valid combination of words and symbols written in the Procedure Division beginning with a verb.

Subject of Entry:

An operand or reserved word that appears immediately following the level indicator or the level-number in a Data Division entry.

Subprogram. (See Called Program)**Subscript:**

An integer whose value identifies a particular element in a table.

Subscripted Data-Name:

An identifier that is composed of a data-name followed by one or more subscripts enclosed in parentheses.

Switch-Status Condition:

The proposition, for which a truth value can be determined that a switch, capable of being set to an 'on' or 'off' status, has been set to a specific status.

System-Name:

A COBOL word which is used to communicate with the operating environment.

Table:

A set of logically consecutive items of data that are defined in the Data Division by means of the OCCURS clause.

Table Element:

A data item that belongs to the set of repeated items comprising a table.

Text-Name:

A file access name that identifies library text.

Appendix C

Truth Value:

The representation of the result of the evaluation of a condition in terms of one of two values:

true
false

Unary Operator:

A plus (+) or a minus (-) sign, which precedes a variable or a left parenthesis in an arithmetic expression and which has the effect of multiplying the expression by +1 or -1 respectively.

User-Defined Word:

A COBOL word that must be supplied by the user to satisfy the format of a clause or statement.

Variable:

A data item whose value may be changed by execution of the object program. A variable used in an arithmetic expression must be a numeric elementary item.

Verb:

A word that expresses an action to be taken by a COBOL compiler or object program.

Word:

A character-string of not more than 30 characters which forms a user-defined word, a system-name, or a reserved word.

Working-Storage Section:

The section of the Data Division that describes working storage data items, composed either of noncontiguous items or of working storage records or of both.

77-Level-Description-Entry:

A data description entry that describes a noncontiguous data item with the level-number 77.

APPENDIX D

COMPOSITE LANGUAGE SKELETON

COMPOSITE LANGUAGE SKELETON

This section contains the composite language skeleton of the American National Standard COBOL. It is intended to display complete and syntactically correct formats.

For the general formats of the four divisions the leftmost margin is equivalent to margin A in a COBOL source program. The first indentation after the leftmost margin is equivalent to margin B in a COBOL source program.

For the general formats of the verbs and conditions the leftmost margin indicates the beginning of the format for a new COBOL verb. The first indentation after the leftmost margin indicates continuation of the format of the COBOL verb.

The following is a summary of the formats shown on the following pages:

- Identification Division general format
- Environment Division general format
- The three formats of the file control entry
- Data Division general format
- The three formats for a data description entry
- The format for a field definition entry
- Procedure Division general format
- General format of verbs listed in alphabetical order
- General format for conditions
- Formats for qualification, subscripting, indexing, and an identifier
- General format for a COPY statement

RM/COBOL LANGUAGE SYNTAX

The RM/COBOL language is based upon the ANSI X3.23-1974 COBOL standard. Minor departures from that document are reflected in the syntax description which follows but are not separately noted. Semantic rules are not changed.

The description is in a condensed form of the standard COBOL syntax notation. In some cases separate formats are combined and general terms are employed for user names.

System-names and implementation restrictions are:

computer-name:	User-defined word
program-name:	8-character name
switch-names:	SWITCH-1,..., SWITCH-8
device-types:	PRINT INPUT OUTPUT INPUT-OUTPUT RANDOM
external-file-name:	One- to thirty-character name

IDENTIFICATION DIVISION GENERAL FORMAT

IDENTIFICATION DIVISION.

PROGRAM-ID. program-name.

[AUTHOR. [comment-entry] ...]

[INSTALLATION. [comment-entry] ...]

[DATE-WRITTEN. [comment-entry] ...]

[SECURITY. [comment-entry] ...]

ENVIRONMENT DIVISION GENERAL FORMAT

ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

SOURCE-COMPUTER. computer-name.

OBJECT-COMPUTER. computer-name.

[, MEMORY SIZE integer {WORDS }]
 {CHARACTERS}
 {MODULES }

[, PROGRAM COLLATING SEQUENCE IS alphabet-name].

[SPECIAL-NAMES. [, switch-name

{ON STATUS IS condition-name-1 [, OFF STATUS IS condition-name-2]}]

{OFF STATUS IS condition-name-2 [, ON STATUS IS condition-name-1]}]

[, alphabet-name IS {STANDARD-1}] ...
 {NATIVE }

[, CURRENCY SIGN IS literal-1]

[, DECIMAL-POINT IS COMMA].]

[INPUT-OUTPUT SECTION.

FILE-CONTROL.

{file-control-entry} ...

[I-O-CONTROL.

[; SAME AREA FOR file-name-1 [, file-name-2] ...]... .]]

FILE CONTROL ENTRY GENERAL FORMAT**FORMAT 1**

```
SELECT file-name  
ASSIGN TO device-type {"external-file-name"}  
                      {data-name-1 }  
[; ORGANIZATION IS SEQUENTIAL]  
[; ACCESS MODE IS SEQUENTIAL]  
[; FILE STATUS IS data-name-2].
```

FORMAT 2

```
SELECT file-name  
ASSIGN TO RANDOM, {"external-file-name"}  
                      {data-name-1 }  
; ORGANIZATION IS RELATIVE  
[; ACCESS MODE IS { SEQUENTIAL [, RELATIVE KEY IS data-name-2] } ]  
          {{RANDOM } , RELATIVE KEY IS data-name-2 }  
          {{DYNAMIC} }  
[; FILE STATUS IS data-name-3].
```

FORMAT 3

```
SELECT file-name  
ASSIGN TO RANDOM, {"external-file-name"}  
                  {data-name-1               }  
  
; ORGANIZATION IS INDEXED  
  
[; ACCESS MODE IS {SEQUENTIAL}]  
                  {RANDOM            }  
                  {DYNAMIC        }  
  
; RECORD KEY IS data-name-2  
  
[; ALTERNATE RECORD KEY IS data-name-3 [WITH DUPLICATES]]...  
  
[; FILE STATUS IS data-name-4].
```

DATA DIVISION GENERAL FORMATDATA DIVISION.[FILE SECTION.[FD file-name

[; BLOCK CONTAINS [integer-1 TO] integer-2 {RECORDS }]
{CHARACTERS}

[; RECORD CONTAINS [integer-3 TO] integer-4 CHARACTERS]

; LABEL {RECORD IS } {STANDARD}
{RECORDS ARE} {OMITTED }

[; VALUE OF LABEL IS nonnumeric-literal-1]

[; DATA {RECORD IS } data-name-1 [, data-name-2] ...]
{RECORDS ARE}

[record-description-entry] ...] ...

[WORKING-STORAGE SECTION.

[77-level-description-entry] ...]
[record-description-entry]

[LINKAGE SECTION.

[77-level-description-entry] ...]]
[record-description-entry]

DATA DESCRIPTION ENTRY GENERAL FORMAT

FORMAT 1

```

level-number {data-name-1}
              {FILLER      }

[; REDEFINES data-name-2]

[; {PICTURE} IS character-string]
  {PIC     }

[; [USAGE IS] {COMPUTATIONAL } ]
  {COMP          }
  {COMPUTATIONAL-1}
  {COMP-1        }
  {COMPUTATIONAL-3}
  {COMP-3        }
  {DISPLAY       }
  {INDEX         }

[; [SIGN IS] TRAILING [SEPARATE CHARACTER] ]

[; OCCURS {integer-1 TIMES
             {integer-1 TO integer-2 TIMES DEPENDING ON data-name-3}

  [INDEXED BY index-name-1 [, index-name-2] ...] ]

[; {SYNCHRONIZED} [LEFT ] ]
  {SYNC          } [RIGHT]

[; {JUSTIFIED} RIGHT]
  {JUST          }

[; BLANK WHEN ZERO]

[; VALUE IS literal] .

```

FORMAT 2

```
66 data-name-1; RENAMES data-name-2 [{THROUGH} data-name-3].  
                                {THRU }
```

FORMAT 3

```
88 condition-name; {VALUE IS }  
                    {VALUES ARE}  
  
literal-1 [{THROUGH} literal-2]  
          {THRU }  
  
, literal-3 [{THROUGH} literal-4] ... .  
          {THRU }
```

PROCEDURE DIVISION GENERAL FORMAT

FORMAT 1

```
PROCEDURE DIVISION [USING data-name-1 [,data-name-2] ... ] .  
[DECLARATIVES.  
 {section-name SECTION [segment-number]. declarative-sentence  
 [paragraph-name. [sentence] ... ] ... } ...  
END DECLARATIVES.]  
 {section-name SECTION [segment-number].  
 [paragraph-name. [sentence] ... ] ... } ...  
END PROGRAM.
```

FORMAT 2

```
PROCEDURE DIVISION [USING data-name-1 [,data-name-2] ... ] .  
 {paragraph-name. [sentence] ... } ...  
END PROGRAM.
```

GENERAL FORMAT FOR VERBS

```
ACCEPT {identifier-1 [, UNIT {identifier-2}]}
           {literal-1   }

           [, LINE {identifier-3}] [, POSITION {identifier-4}]
           {literal-2   }           {literal-3   }

           [, SIZE {identifier-5}] [, PROMPT [literal-5]]
           {literal-4   }

           [, ECHO] [, CONVERT] [, TAB] [, ERASE] [, NO BEEP]
           [, {OFF}] [, ON EXCEPTION identifier-6 imperative statement]...

ACCEPT identifier FROM {DATE}
           {DAY }
           {TIME}

ADD {identifier-1} [, identifier-2] ... TO identifier-m [ROUNDED]
     {literal-1   } [, literal-2   ]
     [; ON SIZE ERROR imperative-statement]

ADD {identifier-1}, {identifier-2} [, identifier-3] ...
     {literal-1   } {literal-2   } [, literal-3   ]

     GIVING identifier-m [ROUNDED]
     [; ON SIZE ERROR imperative-statement]

ADD {CORRESPONDING} identifier-1 TO identifier-2
     {CORR       }
     [ROUNDED] [; ON SIZE ERROR imperative-statement]

ALTER procedure-name-1 TO [PROCEED TO] procedure-name-2
     [, procedure-name-3 TO [PROCEED TO] procedure-name-4] ...
     [; ON SIZE ERROR imperative-statement]

CALL {identifier-1} [USING data-name-1 [, data-name-2] ... ]
     {literal-1   }
```

```

CLOSE file-name-1 [{REEL} [WITH NO REWIND] ]
    {UNIT}
    WITH {NO REWIND}
    {LOCK      }

[, file-name-2 [{REEL} [WITH NO REWIND] ] ] ...
    {UNIT}
    WITH {NO REWIND}
    {LOCK      }

COMPUTE identifier-1 [ROUNDED] = arithmetic-expression
    [; ON SIZE ERROR imperative-statement]

DELETE file-name RECORD [; INVALID KEY imperative-statement]

DISPLAY {{identifier-1} [, UNIT {identifier-2} ]
    {literal-1 }           {literal-2  }
    [, LINE {identifier-3}][, POSITION {identifier-4}]
        {literal-3 }           {literal-4  }
    [, SIZE {identifier-5}][, BEEP][, ERASE]
        {literal-5  }

    [, {HIGH}][, BLINK][, REVERSE} ...  

        {LOW  }

DIVIDE {identifier-1} INTO identifier-2 [ROUNDED]
    {literal-1 }

    [; ON SIZE ERROR imperative-statement]

DIVIDE {identifier-1} INTO {identifier-2} GIVING identifier-3
    {literal-1 }           {literal-2  }

    [ROUNDED] [; ON SIZE ERROR imperative-statement]

DIVIDE {identifier-1} BY {identifier-2} GIVING identifier-3 [ROUNDED]
    {literal-1 }           {literal-2  }

    [; ON SIZE ERROR imperative-statement]

```

EXIT [PROGRAM].

GO TO procedure-name-1

GO TO procedure-name-1 [, procedure-name-2] ... , procedure-name-n

DEPENDING ON identifier

IF condition; {statement-1} {; ELSE statement-2}
{NEXT SENTENCE} {; ELSE NEXT SENTENCE}

INSPECT identifier-1

[TALLYING identifier-2 FOR {{ALL } {identifier-3}}}
{literal-1 }}
{ {LEADING}}
{ CHARACTERS }

[{BEFORE} INITIAL {identifier-4}]]
{literal-2 }

{AFTER }

[REPLACING {{ALL } {identifier-5}} BY {identifier-6}}
{literal-3 } {literal-4 }
{ {LEADING}}
{ {FIRST }}
{ CHARACTERS }

[{BEFORE} INITIAL {identifier-7}]]
{literal-5 }

{AFTER }

NOTE: The TALLYING option, the REPLACING option, or both options must be selected.

MOVE {identifier-1} TO identifier-2 [, identifier-3]...
 {literal }

MOVE {CORRESPONDING} identifier-1 TO identifier-2
 {CORR }

MULTIPLY {identifier-1} BY identifier-2 [ROUNDED]
 {literal-1 }
 [; ON SIZE ERROR imperative-statement]

MULTIPLY {identifier-1} BY {identifier-2} GIVING identifier-3
 {literal-1 } {literal-2 }
 [ROUNDED] [; ON SIZE ERROR imperative-statement]

OPEN {[INPUT file-name-1 [WITH NO REWIND]]}
 [, file-name-2 [WITH NO REWIND]]...

 {OUTPUT file-name-3 [WITH NO REWIND]]}
 [, file-name-4 [WITH NO REWIND]]...

 {I-O file-name-5}{[, file-name-6]}...

 {EXTEND file-name-7}{[, file-name-8]}...}...
}

```
PERFORM procedure-name-1 [{THROUGH} procedure-name-2]
                           {THRU   }

PERFORM procedure-name-1 [{THROUGH} procedure-name-2]
                           {THRU   }

                           {identifier-1} TIMES
                           {literal-1   }

PERFORM procedure-name-1 [{THROUGH} procedure-name-2]
                           {THRU   }

                           UNTIL condition-1

PERFORM procedure-name-1 [{THROUGH} procedure-name-2]
                           {THRU   }

                           VARYING {identifier-2} FROM {identifier-3}
                           {index-name-1}          {index-name-2}
                           {literal-1   }

                           BY {identifier-4} UNTIL condition-1
                           {literal-3   }

                           [AFTER {identifier-5} FROM {identifier-6}]
                           {index-name-3}          {index-name-4}
                           {literal-3   }

                           BY {identifier-7} UNTIL condition-2
                           {literal-4   }

                           [AFTER {identifier-8} FROM {identifier-9}]
                           {index-name-5}          {index-name-6}
                           {literal-5   }

                           BY {identifier-10} UNTIL condition-3 ]
                           {literal-6   }
```

```

READ file-name RECORD [INTO identifier]
    [; AT END imperative-statement]

READ file-name [NEXT] RECORD [WITH NO LOCK] [INTO identifier]
    [; AT END imperative-statement]

READ file-name RECORD [WITH NO LOCK] [INTO identifier]
    [; KEY IS data-name]
    [; INVALID KEY imperative-statement]

REWRITE record-name [FROM identifier]
    [; INVALID KEY imperative-statement]

SET {identifier-1 [,identifier-2] ...} TO {identifier-3}
    {index-name-1 [,index-name-2] ...} {index-name-3}
        {integer-1 }

SET index-name-4 [,index-name-5] ... {UP BY} {identifier-4}
    {integer-2 }
        {DOWN BY}

START file-name [KEY {IS EQUAL TO } data-name]
    {IS = }
    {IS GREATER THAN }
    {IS > }
    {IS NOT LESS THAN}
    {IS NOT < }

    [; INVALID KEY imperative-statement]

STOP {RUN }
    {literal }

```

```

SUBTRACT {identifier-1} [, identifier-2] ... FROM identifier-m
{literal-1 } [, literal-2 ]
[ROUNDED] [: ON SIZE ERROR imperative-statement]

SUBTRACT {identifier-1} [, identifier-2] ... FROM {identifier-m}
{literal-1 } [, literal-2 ] {literal-m }
GIVING identifier-n [ROUNDED]
[: ON SIZE ERROR imperative-statement]

SUBTRACT {CORRESPONDING} identifier-1 FROM identifier-2 [ROUNDED]
{CORR }
[: ON SIZE ERROR imperative-statement]

UNLOCK file-name-1 RECORD

USE AFTER STANDARD {EXCEPTION}
{ERROR }

PROCEDURE ON {file-name-1 [, file-name-2] ...} .
{INPUT }
{OUTPUT }
{I-O }
{EXTEND }

WRITE record-name [FROM identifier-1]
{BEFORE} ADVANCING {{identifier-2} {LINE }}
{{integer } {LINES}}
{AFTER } {PAGE }

WRITE record-name [FROM identifier]
[: INVALID KEY imperative-statement]

```

GENERAL FORMAT FOR CONDITIONS

RELATION CONDITION:

```
{identifier-1 } {IS [NOT] GREATER THAN} {identifier-2      }
{literal-1     }                                {literal-2      }
{index-name-1 } {IS [NOT] LESS THAN   } {index-name-2      }
              {IS [NOT] EQUAL TO    }
              {IS [NOT] >        }
              {IS [NOT] <        }
              {IS [NOT] =        }
```

CLASS CONDITION:

```
identifier IS [NOT] {NUMERIC    }
                  {ALPHABETIC}
```

CONDITION-NAME CONDITION:

condition-name

SWITCH-STATUS CONDITION:

condition-name

NEGATED SIMPLE CONDITION:

NOT simple-condition

COMBINED CONDITION:

```
condition {{AND} condition} ...
           {OR }
```

MISCELLANEOUS FORMATSQUALIFICATION:

```
{data-name-1      } [{QE} data-name-2] ...
{condition-name}
    {IN}

paragraph-name [{QE} section-name]
    {IN}
```

SUBSCRIPTING:

```
{data-name      } (subscript-1 [, subscript-2 [, subscript-3] ] )
{condition-name}
```

INDEXING:

```
{data-name      } ({index-name-1 [{+} literal-2]}
{condition-name}   {literal-1      {-}           }
                  [, {index-name-2[{+} literal-4]}
                   {literal-3      {-}           }]
                  [, {index-name-3 [{+} literal-6] } ] ] )
                  {literal-5      {-}           }
```

Appendix D

IDENTIFIER:

FORMAT 1

```
data-name-1 [{OF} data-name-2] ...
{IN}
[(subscript-1 [, subscript-2 [, subscript-3] ] ) ]
```

FORMAT 2

```
data-name-1 [{OF} data-name-2] ... [( {index-name-1 [{+} literal-2]}
{literal-1      {-}          }
{IN}
[, {index-name-2 [{+} literal-4]}
{literal-3      {-}          }
[, {index-name-3 [{+} literal-6]} ])]
{literal-5      {-}          }
```

GENERAL FORMAT FOR COPY STATEMENT

COPY text-name

COBOL LEVEL OF IMPLEMENTATION

<u>Function Module</u>	<u>Implementation</u>
Nucleus	Level 2.
Table Handling	Level 1+.
Sequential I/O	Level 2.
Relative I/O	Level 2.
Indexed I/O	Level 2.
Sort-Merge	Null.
Report Writer	Null.
Segmentation	Level 1.
Library	Level 1.
Debug	N/S. Conditional compile and execution time interactive debugger.
Inter-program Communication	Level 1.
Communication	Modified ACCEPT and DISPLAY for terminal communication.

ANSI COBOL X3.23 1974

MODULE	FEDERAL INFORMATION PROCESSING STANDARD (FIPS)					RM COBOL
	HIGH	HIGH INTERMEDIATE	LOW INTERMEDIATE	LOW		
NUCLEUS	2	2	1	1	2	
TABLE HANDLING	2	2	1	1	1+	
SEQUENTIAL I/O	2	2	1	1	2	
RELATIVE I/O	2	2	1	-	2	
INDEXED I/O	2	-	-	-	2	
SORT-MERGE	2	1	-	-	-	
REPORT WRITER	-	-	-	-	-	
SEGMENTATION	2	1	1	-	1	
LIBRARY	2	1	1	-	1	
DEBUG	2	2	1	-	N/S	
INTER-PROGRAM COMMUNICATION	2	2	1	-	1+	
COMMUNICATION	2	2	-	-	N/S	

N/S = Nonstandard

EXTENSIONS BEYOND STATED LEVELS

Level 2 Nucleus (2 NUC):

- Data description includes a USAGE type of COMPUTATIONAL-1 or COMP-1 for describing single word two's complement signed binary data (nonstandard).
- Data description includes a USAGE type of COMPUTATIONAL-3 or COMP-3 for describing packed decimal data (nonstandard).
- The ACCEPT statement allows multiple operands (nonstandard).
- The ACCEPT statement includes syntax for specifying CRT control information (nonstandard).
- The DISPLAY statement includes syntax for specifying CRT control information (nonstandard).

Level 1 Table Handling (1 TBL):

- Variable group size (OCCURS DEPENDING).

Level 2 Sequential I-O (2 SEQ):

- The file control SELECT clause allows specification of the external file name as a literal or data item (nonstandard).
- The READ statement includes the WITH NO LOCK option (nonstandard).
- The UNLOCK statement is included (nonstandard).

Level 2 Relative I-O (2 REL):

- The file control SELECT clause allows specification of the external file name as a literal or data item (nonstandard).
- The READ statement includes the WITH NO LOCK option (nonstandard).
- The UNLOCK statement is included. (nonstandard).

Level 2 Indexed I-O (2 INX):

- The file control SELECT clause allows specification of the external file name as a literal or data item (nonstandard).

- The READ statement includes the WITH NO LOCK option (nonstandard).
- The UNLOCK statement is included (nonstandard).

Level 1 Debug (1 DEB):

- An interactive execution time debug facility is provided (nonstandard).

Level 1 Inter-Program Communication (1 IPC):

- The CALL statement allows literals in USING phrase (nonstandard).
- The CALL statement allows identifiers in the USING phrase to be described with level number 01 through 49 and level number 77 (nonstandard).
- The CALL statement supports specification of a variable program name as identifier-1 (level 2 IPC).

Level 1 Communication (1 COM):

- ACCEPT and DISPLAY allow specification of complete screen format in the Procedure Division (nonstandard).

EXCEPTIONS TO STATED LEVELS

Level 2 Nucleus (2 NUC):

- DATE-COMPILED is not supported in the Identification Division.
- In data description the SIGN clause cannot specify LEADING for the operational sign; omission of the SEPARATE phrase has no effect; all operational signs are separate trailing characters.
- Alphabet-name IS literal or implementor-name may not be specified in SPECIAL-NAMES paragraph.
- Multiple results are not supported in arithmetic statements.
- REMAINDER is not supported in DIVIDE statement.
- A procedure-name is required in GO TO statements.
- INSPECT data items are restricted to single character.
- Compound TALLYING and REPLACING clauses in the INSPECT statement are not supported.
- When used in the Procedure Division, the numeric literal in the ALL form of a figurative constant may not contain more than one character.
- Arithmetic expressions may be used only in COMPUTE statements.
- Exponentiation to a noninteger power is not supported.
- Sign conditions are not supported.
- Abbreviated combined relation conditions are not supported.
- The STRING and UNSTRING statements are not supported.

Level 2 Sequential I-O (2 SEQ):

- OPTIONAL and RESERVE may not be specified in the SELECT clause.
- RERUN, SAME AREA or MULTIPLE FILE clauses are not supported in I-O-CONTROL.

- CODE-SET and LINAGE clauses may not be specified in a file description entry.
- The mnemonic-name and EOP options of the WRITE statement are not supported.
- The REVERSED option of the OPEN statement is not supported.
- The FOR REMOVAL option of the CLOSE statement is not supported.

Level 2 Relative I-O (2 REL):

- The RESERVE clause of the SELECT entry is not supported.
- RERUN, SAME AREA or MULTIPLE FILE clauses are not supported in I-O-CONTROL.
- The VALUE OF clause in an FD entry must not specify a data name.

Level 2 Indexed I-O (2 INX):

- The RESERVE clause of the SELECT entry is not supported.
- RERUN, SAME AREA or MULTIPLE FILE clauses are not supported in I-O-CONTROL.

Level 1 Segmentation (1 SEQ):

- All independent segments must physically follow the fixed permanent segments in the source program.

Level 1 Library (1 LIB):

- A copy sentence must be the last entry in area B of a source record.

Level 1 Inter-Program Communication (1 IPC):

- A CALLED program is automatically cancelled upon execution of the EXIT PROGRAM statement.

APPENDIX E

SAMPLE PROGRAMS

Ryan-McFarland COBOL (RM/COBOL ver 1.3B for CP/M 2.0)
 SOURCE FILE: CALCXMPL

OPTION LIST: L O=N X

PAGE 1

LINE DEBUG PG/LN A...B.....

```

1      IDENTIFICATION DIVISION.
2      PROGRAM-ID.
3          CALCULATOR.
4
5      ENVIRONMENT DIVISION.
6      CONFIGURATION SECTION.
7          SOURCE-COMPUTER.      RMC.
8          OBJECT-COMPUTER.    RMC.
9
10     DATA DIVISION.
11     WORKING-STORAGE SECTION.
12     77 RESULT           PICTURE S9(9)V9(9) VALUE ZERO.
13     77 OPERAND-1        PICTURE S9(9)V9(9).
14     77 OPERAND-2        PICTURE S9(9)V9(9).
15     77 WAIT-CHAR        PICTURE X.
16     01 GREETING.
17         02 FILLER        PICTURE X(18)
18             VALUE "CALCULATOR PROGRAM".
19     01 OPERATION-MESSAGE.
20         02 FILLER        PICTURE X(37)
21             VALUE "CHOOSE YOUR OPERATION (+,-,*,/) = ".
22     01 OPERATOR          PICTURE X(2).
23     01 RESULT-MESSAGE.
24         02 FILLER        PICTURE X(12)
25             VALUE "RESULT IS = ".
26         02 RESULT-EDITED   PICTURE -(9)9.9(9).
27         02 FILLER          PIC X(4) VALUE SPACES.
28         02 OVERFLOW-FIELD  PIC X(8) VALUE SPACES.
29     01 WAIT-MESSAGE.
30         02 FILLER        PICTURE X(36)
31             VALUE "HIT NEWLINE TO CONTINUE (Q TO QUIT) ".
32     01 OPERAND-1-MESSAGE.
33         02 FILLER        PICTURE X(12)
34             VALUE "OPERAND-1 = ".
35     01 OPERAND-2-MESSAGE.
36         02 FILLER        PICTURE X(12)
37             VALUE "OPERAND-2 = ".

```

Ryan-McFarland COBOL (RM/COBOL ver 1.3B for CP/M 2.0) PAGE 2
 SOURCE FILE: CALCXMPL OPTION LIST: L O=N X

LINE DEBUG PG/LN A...B.....

```

38      / EJECT
39      PROCEDURE DIVISION.
40 >0000  RESIDENT SECTION 1.
41 >0000  NOT-START.
42 >0000  GO TO DISPLAY-GREETING.
43 >0004  RE-TRY.
44 >0004  DISPLAY OPERATION-MESSAGE, LINE 2, ERASE.
45 >000C  ACCEPT OPERATOR, POSITION 0, PROMPT, ECHO.
46 >0014  IF OPERATOR EQUAL "+" GO TO ADDITION.
47 >001C  IF OPERATOR EQUAL "-" GO TO SUBTRACTION.
48 >0024  IF OPERATOR EQUAL "*" GO TO MULTIPLICATION.
49 >002C  IF OPERATOR EQUAL "/" GO TO DIVISION.
50 >0034  IF OPERATOR EQUAL "Q" GO TO END-RUN.
51 >003C  GO TO RE-TRY.
52 >003E  DISPLAY-RESULT.
53 >003E  MOVE RESULT TO RESULT-EDITED.
54 >0042  DISPLAY RESULT-MESSAGE.
55 >0046  MOVE ZERO TO RESULT.
56 >004A  MOVE SPACES TO OVERFLOW-FIELD.
57 >0050  WAIT-ENTRY.
58 >0050  DISPLAY WAIT-MESSAGE.
59 >0054  ACCEPT WAIT-CHAR, POSITION 0, PROMPT, ECHO.
60 >005C  IF WAIT-CHAR EQUAL "Q" GO TO END-RUN.
61 >0064  GO TO RE-TRY.
62 >0066  GET-OPERANDS.
63 >0066  DISPLAY OPERAND-1-MESSAGE, LINE 4.
64 >006C  ACCEPT OPERAND-1, LINE 4, POSITION 13, SIZE 10,
65           PROMPT, CONVERT.
66 >0078  MOVE OPERAND-1 TO RESULT-EDITED.
67 >007C  DISPLAY RESULT-EDITED, LINE 4, POSITION 13.
68 >0084  DISPLAY OPERAND-2-MESSAGE.
69 >0088  ACCEPT OPERAND-2, LINE 5, POSITION 13, SIZE 10,
70           PROMPT, CONVERT.
71 >0094  MOVE OPERAND-2 TO RESULT-EDITED.
72 >0098  DISPLAY RESULT-EDITED, LINE 5, POSITION 13.
73 >00A2  END-RUN.
74 >00A2  EXIT PROGRAM.
75 >00A6  STOP-RUN.
76 >00A6  STOP RUN.
```

Ryan-McFarland COBOL (RM/COBOL ver 1.3B for CP/M 2.0) PAGE 3
 SOURCE FILE: CALCXMPL OPTION LIST: L O=N X

LINE DEBUG PG/LN A...B.....

```

77                / EJECT
78>0100A8        OVERLAY-ADDITION SECTION 51.
79>0100A8        ADDITION.
80>0100A8        PERFORM GET-OPERANDS.
81>0100AA        ADD OPERAND-1 OPERAND-2 GIVING RESULT
82                ON SIZE ERROR MOVE "OVERFLOW" TO OVERFLOW-FIELD.
83>0100B8        GO TO DISPLAY-RESULT.
84
85>0200A8        OVERLAY-SUBTRACTION SECTION 52.
86>0200A8        SUBTRACTION.
87>0200A8        PERFORM GET-OPERANDS.
88>0200AA        SUBTRACT OPERAND-2 FROM OPERAND-1 GIVING RESULT
89                ON SIZE ERROR MOVE "OVERFLOW" TO OVERFLOW-FIELD.
90>0200B8        GO TO DISPLAY-RESULT.
91
92>0300A8        OVERLAY-MULTIPLICATION SECTION 53.
93>0300A8        MULTIPLICATION.
94>0300A8        PERFORM GET-OPERANDS.
95>0300AA        MULTIPLY OPERAND-1 BY OPERAND-2 GIVING RESULT
96                ON SIZE ERROR MOVE "OVERFLOW" TO OVERFLOW-FIELD.
97>0300B8        GO TO DISPLAY-RESULT.
98
99>0400A8        OVERLAY-DIVISION SECTION 54.
100>0400A8       DIVISION.
101>0400A8       PERFORM GET-OPERANDS.
102>0400AA       DIVIDE OPERAND-1 BY OPERAND-2 GIVING RESULT ROUNDED
103                ON SIZE ERROR MOVE "OVERFLOW" TO OVERFLOW-FIELD.
104>0400BA       GO TO DISPLAY-RESULT.
105
106>0500A8       OVERLAY-DISPLAY-GREETING SECTION 98.
107>0500A8       DISPLAY-GREETING.
108>0500A8       DISPLAY GREETING.
109>0500AC       GO TO WAIT-ENTRY.
110
111               END PROGRAM.
```

ADDRESS	SIZE	DEBUG	ORDER	TYPE	NAME
>0004	19	NSS	0	NUMERIC SIGNED	RESULT
>0018	19	NSS	0	NUMERIC SIGNED	OPERAND-1
>002C	19	NSS	0	NUMERIC SIGNED	OPERAND-2
>0040	1	ANS	0	ALPHANUMERIC	WAIT-CHAR
>0042	18	GRP	0	GROUP	GREETING
>0054	37	GRP	0	GROUP	OPERATION-MESSAGE
>007A	2	ANS	0	ALPHANUMERIC	OPERATOR
>007C	44	GRP	0	GROUP	RESULT-MESSAGE
>0088	20	NSE	0	NUMERIC EDITED	RESULT-EDITED
>00A0	8	ANS	0	ALPHANUMERIC	OVERFLOW-FIELD
>00A8	36	GRP	0	GROUP	. WAIT-MESSAGE
>00CC	12	GRP	0	GROUP	OPERAND-1-MESSAGE
>00D8	12	GRP	0	GROUP	OPERAND-2-MESSAGE

READ ONLY BYTE SIZE = >01BE

READ/WRITE BYTE SIZE = >00EC

OVERLAY SEGMENT BYTE SIZE = >002E

TOTAL BYTE SIZE = >02D8

0 ERRORS

0 WARNINGS

Editor

**COBOL Development System
TRS-XENIX™ Operating System**

Ryan-McFarland COBOL (RM/COBOL ver 1.3B for CP/M 2.0)
 SOURCE FILE: CALCXMPL

OPTION LIST: L O=N X

PAGE 5

CROSS REFERENCE	/DECL/ *DEST*
ADDITION	0046 /0079/
DISPLAY-GREETING	0042 /0107/
DISPLAY-RESULT	/0052/ 0083 0090 0097 0104
DIVISION	0049 /0100/
END-RUN	0050 0060 /0073/
GET-OPERANDS	/0062/ 0080 0087 0094 0101
GREETING	/0016/ 0108
MULTIPLICATION	0048 /0093/
NOT-START	/0041/
OPERAND-1	/0013/ *0064* 0066 0081 *0088* 0095 0102
OPERAND-1-MESSAGE	/0032/ 0063
OPERAND-2	/0014/ *0069* 0071 0081 0088 *0095* 0102
OPERAND-2-MESSAGE	/0035/ 0068
OPERATION-MESSAGE	/0019/ 0044
OPERATOR	/0022/ *0045* 0046 0047 0048 0049 0050
OVERFLOW-FIELD	/0028/ *0056* *0082* *0089* *0096* *0103*
OVERLAY-ADDITION	/0078/
OVERLAY-DISPLAY-GREETING	/0106/
OVERLAY-DIVISION	/0099/
OVERLAY-MULTIPLICATION	/0092/
OVERLAY-SUBTRACTION	/0085/
RESIDENT	/0040/
RESULT	/0012/ 0053 *0055* *0081* *0088* *0095* *0102*
RESULT-EDITED	/0026/ *0053* *0066* 0067 *0071* 0072
RESULT-MESSAGE	/0023/ 0054
RE-TRY	/0043/ 0051 0061
STOP-RUN	/0075/
SUBTRACTION	0047 /0086/
WAIT-CHAR	/0015/ *0059* 0060
WAIT-ENTRY	/0057/ 0109
WAIT-MESSAGE	/0029/ 0058

A note about the vi documentation...

The vi editor is part of the TRS-Xenix Development System.
To use vi with COBOL, keep in mind the following:

- . Do not use <TAB> in your COBOL source.
- . Line numbers (Columns 1 through 6) are not required. Instead, you may use six spaces or a comment.

This section has been duplicated from the TRS-Xenix Fundamentals Manual.

CHAPTER 6

VI

CONTENTS

6.1	Introduction.....	6-1
6.2	Demonstration Run.....	6-2
6.3	Basic Concepts.....	6-12
6.3.1	The Editing Buffer.....	6-12
6.3.2	Modes of Operation.....	6-13
6.3.3	Special Keys.....	6-15
6.3.4	Text Objects.....	6-16
6.4	Invoking and Exiting Vi.....	6-16
6.5	Vi Commands.....	6-18
6.5.1	Cursor Movement.....	6-19
6.5.2	The Screen Commands.....	6-24
6.5.3	Text Insertion.....	6-26
6.5.4	Text Deletion.....	6-27
6.5.5	Text Modification.....	6-28
6.5.6	Text Movement.....	6-31
6.5.7	Searching.....	6-32
6.5.8	Exit and Escape Commands.....	6-34
6.6	Ex Commands.....	6-35
6.6.1	Command Structure.....	6-35
6.6.2	Command Addressing.....	6-36
6.6.3	Command Format.....	6-38
6.6.4	Argument List Commands.....	6-38
6.6.5	Edit Commands.....	6-39
6.6.6	Write Commands.....	6-40
6.6.7	Read Commands.....	6-41
6.6.8	Quit Commands.....	6-41
6.6.9	Global and Substitute Commands.....	6-42
6.6.10	Text Movement Commands.....	6-44
6.6.11	Shell Escape Commands.....	6-44
6.6.12	Other Commands.....	6-45
6.7	Start-Up Files and Options.....	6-47
6.7.1	Setting the Terminal Type.....	6-47

6.7.2	The .exrc File.....	6-48
6.7.3	Options.....	6-48
6.8	Regular Expressions.....	6-53
6.9	Speeding Things Up.....	6-55
6.9.1	When To Use Ex.....	6-56
6.10	Limitations.....	6-56
6.11	Troubleshooting.....	6-57
6.12	Character Functions.....	6-59

6.1 Introduction

Vi is a screen-oriented text editor that can be used for most any text editing purpose. It is well integrated into the XENIX environment and should be thought of, along with the shell, as one of the central XENIX tools. Vi offers a powerful set of text editing operations based on a mnemonic command set. Most commands are single keystrokes that perform simple editing functions. So that the terminal screen is used to the utmost, Vi displays a "window" into the file you are editing. This window can be changed quickly and easily within Vi, and provides visual feedback while editing (the name Vi itself is short for "visual").

As you use Vi, it is important to realize that it and the line editor Ex are one and the same editor: the names Vi and Ex identify a particular user interface rather than any underlying functional difference. The differences in user interface, however, are quite striking. Ex is a powerful line-oriented editor. However, visual updating of the terminal screen is limited, and commands are entered on a command line. Vi, on the other hand, is a screen-oriented editor designed so that what you see on the screen corresponds exactly and immediately with the contents of the file you are editing. Most Vi commands are single keystroke mnemonics that do not require a command line.

For most editing purposes, you will want to use Vi rather than either of the editors Ed or Ex because of the superior way in which Vi displays file contents on the screen. It is important, however, to realize that many of the commands available to you in Ex also work in Vi. Those commands that work in an identical fashion are those that you will type on the bottom status line in Vi. Keep this Vi/Ex split in mind as you use the editor -- it will help to eliminate confusion that can arise when learning the editor.

6.2 Demonstration Run

The following demonstration run gives you hands on experience using Vi. It should give you some initial satisfaction that you can use it for most any editing purpose. Remember that the best way to learn Vi is to actually use it, so don't be afraid to experiment.

NOTE: Most of the Vi commands in this demonstration run are single key strokes that do not require a terminating <RETURN>. Therefore, do not assume that a <RETURN> is needed after the entry of each command. Commands that do require a terminating <RETURN> are called Ex commands and are entered as command lines on the Vi status line.

To begin, you must first make sure that your terminal has been properly set up. Most terminals are supported, so there should be no problem. However, this demonstration run presumes that Vi knows about the terminal you are using. See Section 6.8 "Start-Up Files and Options," for more information about setting up your terminal for use with Vi.

To enter the editor type:

```
vi temp
```

This invokes the editor and places you in Vi command mode, where the keys you press are interpreted as editing commands. The editor then clears your screen and prints out a row of tildes (~). You are initially editing a temporary file called the editing buffer. The top line of your display is the only line in the editing buffer and is marked by the cursor. The line containing the cursor is called the "current line." The lines containing tildes are not part of the editing buffer: they indicate lines on the screen only, not real lines in the editing buffer. When you write out the editing buffer, you will write to the file named temp, which is the same as the file you named when you invoked the editor. In our examples, the cursor will be indicated by an underscore, as shown below:

```
=  
~  
~  
~  
~  
~  
~  
~  
~  
~
```

Note that we show a shrunken ten line screen to save space. In reality, however, Vi takes advantage of whatever size screen you have.

To begin, create some text by using the **i** (insert) command. To do this, type:

```
i
```

Next, type the following three lines of text to give you something to play around with (note that <RETURN> and <ESC> are single keys):

```
-----  
Files contain text.<RETURN>  
Text contains lines.<RETURN>  
Lines contain characters.<RETURN>  
<ESC>  
~  
~  
~  
~  
~  
~
```

Like most commands, the **i** command is mnemonic (for insert) and is not echoed on your screen. The command itself switches you from command mode to insert mode. Once in insert mode, the characters you type are inserted into the editing buffer; they are not interpreted as Vi commands. To exit insert mode and reenter command mode you will always need to type <ESC>. This switching between modes occurs often in Vi, so get used to it now.

Next comes a command that you'll use frequently in Vi: the repeat command. The repeat command repeats the most recent insert or delete command. Since we have just executed an

insert command, the repeat command will repeat the insertion, duplicating the inserted text. The repeat command is executed by typing a period (.) or "dot". So, just type

to insert another three lines of text. The command is repeated relative to the location of the cursor and inserts text below the current line. (The current line is always the line containing the cursor.) After you type dot, your screen will look like this:

```
-----  
Files contain text.  
Text contains lines.  
Lines contain characters.  
Files contain text.  
Text contains lines.  
Lines contain characters.
```

```
=  
~  
~
```

Another command which is very useful (and which you'll need often in the beginning) is the undo command, u. Type

u

and notice that the three lines you just finished inserting are deleted or "undone":

```
-----  
Files contain text.  
Text contains lines.  
Lines contain characters.
```

```
=  
~  
~  
~  
~  
~
```

Now type

u

again, and the three lines are reinserted! This undo feature can be remarkably useful in recovering from inadvertent deletions or insertions. Notice that in contrast to the repeat command (.), the undo command inverts the last insert or delete command. And because the undo command is an insert or delete command itself, two consecutive undo commands cancel each other out.

Now lets learn how to move the cursor around in Vi. Typing the keys "h", "j", "k", and "l", will move the cursor left, down, up, and right, respectively. Note that these keys are chosen because of their relative positions on the keyboard, not for any mnemonic reason. On most terminals, you can also use the arrow keys to move in the same way. Remember that the "h", "j", "k", and "l" keys and the arrow keys only work when in command mode.

Try moving the cursor using these keys. When you are done, type the H command to home the cursor in the upper left corner of the screen. Then type the L command to move to the lowest line on the screen. (Note that case is significant in our example: L moves to the lowest line on the screen; while l will move the cursor forward one character.) Next, try moving the cursor to the last line in the file with the "goto" command, G. If you type "2G", the cursor will move to the beginning of the second line in the file; if you have a 10,000 line file, and type "8888G", the cursor will go to the beginning of line 8888.

The above cursor movement commands should allow you to move around well enough for this demonstration run. Other cursor movement commands you might want to try out are: w, to move forward a word; b, to back up a word; 0 to move to the beginning of a line; and \$ to move to the end of a line.

Several screen-oriented scrolling commands also exist. These are all mnemonically named control characters:

- <CONTROL-U> Scroll up
- <CONTROL-D> Scroll down
- <CONTROL-F> Page forward one screenful
- <CONTROL-B> Page backwards one screenful

You can also search forward for a string of characters by typing a slash (/) followed by the string of characters you are searching for, terminated by a <RETURN>. For example, type

H

to move the cursor to the top of the screen, then type

/char<RETURN>

as shown below:

```
-----  
Files contain text.  
Text contains lines.  
Lines contain characters.  
Files contain text.  
Text contains lines.  
Lines contain characters.  
~  
~  
~  
~  
/char<RETURN>  
-----
```

Your cursor moves to the beginning of the word "characters" on line three. To search for the next occurrence of the string "char", simply type the character "n". This will take you to the beginning of the word "characters" on line six. If you type "n" again, Vi will search past the end of the file, wrap around to the beginning, and again find the occurrence of the string "char" on line three.

Note that the slash character and the string that you were searching for appear at the bottom of the screen before you type. This bottom line is the Vi status line. It is used to display several different kinds of information:

1. Ex commands
2. Strings that you are searching for
3. Buffer status information
4. Error messages
5. Any other information that needs to be distinguished from the text that is part of your file

For example, to get status information about the editing buffer, type <CONTROL-G>. This tells you the name of the file you are editing, whether it has been modified, the current line number, the number of lines in the file, and the percentage of the file (in lines) that precedes the

cursor (i.e., where you are in the file relative to the beginning). All this information is given on the status line.

Now that we know how to insert and create text, and how to move around within the editing buffer, we're ready to delete text. The three most common delete commands are:

- dd Delete the current line (i.e., the line in which the cursor resides).
- dw Delete the word in which the cursor resides.
- x Delete the character beneath the cursor.
- D Delete from the cursor to the end of the line.
- d0 Delete from the cursor to the start of the line.
- . Repeat the last change. (Use this only if your last command was a deletion and not an insertion, otherwise you'll insert text instead of delete it.)

To learn how all these commands work, we'll step you through the deletion of various parts of the editing buffer. To begin, move to the first line of your editing buffer by typing:

1G

At first, your editing buffer should look like this:

```
-----  
Files contain text.  
Text contains lines.  
Lines contain characters.  
Files contain text.  
Text contains lines.  
Lines contain characters.  
=
```

```
~  
~
```

Now type

dd

to delete the first line. Your editing buffer now looks

like this:

```
-----  
Text contains lines.  
Lines contain characters.  
Files contain text.  
Text contains lines.  
Lines contain characters.  
~  
~  
~  
~  
~  
~-----
```

Next type

dw

to delete the word in which the cursor resides. Your file now looks like this:

```
-----  
contains lines  
Lines contain characters.  
Files contain text.  
Text contains lines.  
Lines contain characters.  
~  
~  
~  
~  
~-----
```

You can quickly delete the character beneath the cursor by typing:

x

This leaves:

```
-----  
contains lines.  
Lines contain characters.  
Files contain text.  
Text contains lines.  
Lines contain characters.  
~  
~  
~  
~  
~
```

Now type a "w" command to move your cursor to the beginning of the word "lines" on line one. Then type an uppercase "D" to delete to the end of the line:

D

This leaves your editing buffer looking like this:

```
-----  
contains  
Text contains lines.  
Lines contain characters.  
Files contain text.  
Text contains lines.  
Lines contain characters.  
~  
~  
~  
~  
~
```

Now type

d0

to delete all characters on the line before the cursor.
This leaves a single space on the line:

```
-----  
Lines contain characters.  
Files contain text.  
Text contains lines.  
Lines contain characters.  
~  
~  
~  
~  
~
```

All of the editing you have been doing has affected the editing buffer, and not the file named temp that you specified when you invoked Vi. To write the editing buffer out to temp, use the Ex write command:

```
-----  
Lines contain characters.  
Files contain text.  
Text contains lines.  
Lines contain characters.  
~  
~  
~  
~  
~  
:w<RETURN>
```

All Ex commands are preceded by a colon which acts as a prompt on the status line. Ex commands themselves are entered on this line and terminated with a <RETURN>. If you want to write the editing buffer out to a file other than temp, you can give the w command a filename argument that specifies the name of the file you want to write to.

In general, Ex commands allow you to interface with the operating system. For instance, you can read in the contents of a file below the current line by typing

```
:r filename<RETURN>
```

where filename is the name of the file you want to read in. You can also execute arbitrary XENIX commands such as date, by typing:

```
::!date<RETURN>
```

This will output the date and then prompt you to press

<RETURN> to reenter Vi command mode. Go ahead and try it.

Note that when you execute Ex commands, you are really executing commands available in the line-oriented editor called Ex. Ex and Vi are really one and the same editor, the only difference between the two is the user interface.

Besides the set of editing commands described above, there are a number of options that can be set either when you invoke Vi, or later when editing. These options allow you to control a large number of editing parameters. For example, you can specify automatic line numbering, automatic word wrap, and whether or not case is significant in string searches. You can get a complete list of the available options by typing:

```
:set all<RETURN>
```

How to set these options is described in Section 6.7, "Start-Up Files and Options," but it is important now that you be aware of their existence. Depending on what you are doing, and your own personal preferences, you will want to alter the default settings for many of these options.

Finally, to exit Vi and save the editing buffer to the file named temp, type:

```
:x<RETURN>
```

If you have made any changes to the editing buffer, this writes out the editing buffer to the last named file and then exits the editor. If you don't want to save the editing buffer, you can abort the editing session by typing:

```
:q!<RETURN>
```

This completes the demonstration run. There are still many commands that have not been shown you, but nevertheless, the fundamentals of using Vi have been covered. You should now know how to get into and out of Vi, how to insert and delete text, how to move your cursor around, how to execute Ex commands, and how to write out the editing buffer to a file. Following sections give you more detailed information about the commands covered above and about Vi's other commands and features.

6.3 Basic Concepts

To use Vi effectively, you will need to understand the basic concepts that are essential in understanding how Vi works. The topics discussed here include:

- The Editing Buffer
- Modes of Operation
- Special Keys
- Text Objects

6.3.1 The Editing Buffer

Vi performs no editing operations on the file that you name during invocation. Instead, it works on a copy of the file in an editing buffer. The editor remembers the name of the file specified at invocation, so that it can later copy the editing buffer back to the named file. This means that you do not affect the contents of the named file unless and until you explicitly copy the changes you have made back to the original file. This setup allows you to edit the buffer without immediately destroying the contents of the original file. See Figure 6-1 for an illustration of how this all works.

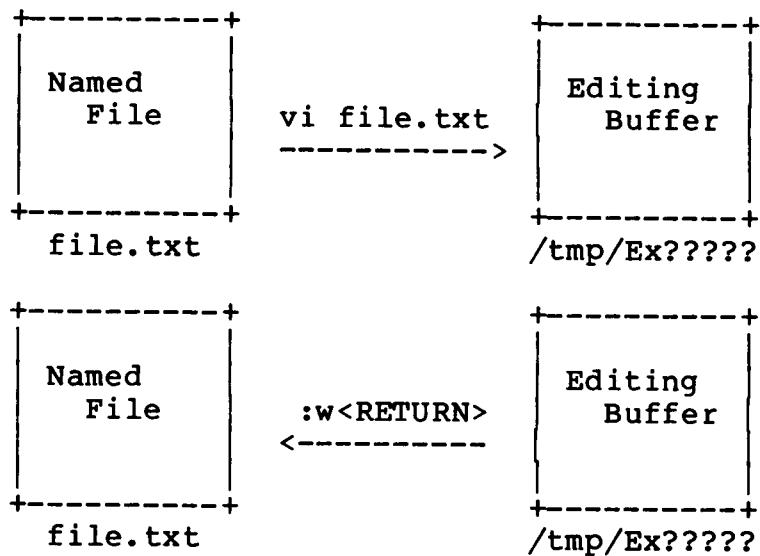


Figure 6-1. The Editing Buffer

When you invoke Vi with a single filename argument, the named file is copied to a temporary editing buffer. When the file is written out, the temporary file is written back to the named file.

6.3.2 Modes of Operation

Before using Vi extensively you need to clearly understand the concept of mode. Within Vi there are three distinct modes of operations:

Command Mode

After invoking Vi, you are automatically placed in Vi command mode. Within command mode, the keys that you press are interpreted as editing commands. In most cases, the keys that you press are not echoed on the screen. Because they are not echoed, you get no visual feedback on the command you are typing. This isn't bad for single keystroke commands, but it sometimes can be confusing for commands that require several keystrokes. However, if you ever get confused when typing a command, you can abort command entry by pressing <INTERRUPT>.

Insert Mode

Insert mode can be entered by typing any of the Vi insert, append, open, substitute, change, or replace commands. A few other commands also put

you into insert mode. Once in insert mode, the keys you type are inserted into your editing buffer as you type them. Some characters are special in insert mode; these are listed and described below:

<BKSP>

This backs up the cursor one character on the current line. The last character typed before the <BKSP> is removed from the input buffer, but remains displayed on the screen.

<CONTROL-U>

Moves the cursor back to the first character of the insertion, and restarts insertion.

<CONTROL-V>

Removes the special significance of the next typed character. Use <CONTROL-V> to insert control characters. Note that line feed <LF> and <CONTROL-J> cannot be inserted in the text except as newline characters. Also, <CONTROL-Q> and <CONTROL-S> are trapped by the operating system before they are interpreted by Vi, so they too cannot be inserted.

<CONTROL-W>

Moves the cursor back to the first character of the last inserted word.

<CONTROL-D>

Backtabs over whitespace at the beginning of a line. Otherwise, if the autoindent option is set this whitespace cannot be backspaced over.

<CONTROL-T>

During an insertion, with the autoindent option set and at the beginning of the current line, typing this character will insert shiftwidth white space.

<CONTROL-@>

If typed as the first character of an insertion it is replaced with the last text inserted, and the insertion terminates. Only 128 characters are saved from the last insertion. If more than 128 characters were inserted, then this command inserts no characters. A <CONTROL-@> cannot be part of

a file, even if quoted.

Ex Escape Mode

The Vi and Ex editors are one and the same editor differing mainly in their user interface. In Vi, as we have seen, commands are usually single keystrokes. In Ex, commands are lines of text terminated by a <RETURN>. Vi has a special "escape" command that gives you access to many of these line-oriented Ex commands. To escape to Ex escape mode, you need only type a colon (:). The colon is echoed on the bottom status line as a prompt for the Ex command that you want to execute. You then may type in the command, followed by a <RETURN> or an <ESC>. An executing command can also be aborted by typing <INTERRUPT>. Most file manipulation commands are executed in Ex escape mode; for example, the commands to read in a file, and to write out the editing buffer to a file. Other Ex commands that you need to know are the commands to perform global substitutions and to quit the editor. For more information, see Section 6.6 "Ex Commands."

6.3.3 Special Keys

There are several keys that you'll use over and over when editing in Vi. These keys are used to edit, delimit, or abort commands and command lines. The first key that we'll look at is the <ESC> key. It should be near the upper left corner of your terminal. Try pressing it a few times -- it rings the bell to indicate that Vi is in its normal command state. On some terminals, the editor quietly flashes the screen rather than ringing the bell. In general, the <ESC> key always returns you to Vi command mode. You'll use it most often in exiting from insert mode. It can also be used to terminate Ex command lines. In addition, partially formed commands are canceled by typing an <ESC>. This key is a fairly harmless one to press, so you should press it when you don't know exactly what is going on.

The <RETURN> key is used to terminate Ex commands when in Ex escape mode. You also type <RETURN> whenever you want to start a new line when in insert mode.

Another useful key is the <INTERRUPT> key, which is often the same as the or <ERUBOUT> key on many terminals. The <INTERRUPT> key, as its name implies, generates an interrupt, telling the editor to stop what it is doing. You can use <INTERRUPT> to abort any command that is executing.

It is a forceful way of making the editor listen to you, or to return to Vi command mode, if you don't know or don't like what is going on.

The next key of interest is the slash (/) key. This key is used when you want to specify a string to be searched for. When you type it, the slash appears on the bottom status line as a prompt for a search string. You can then enter the search string, followed by a <RETURN> or an <ESC>. You can get the cursor back to the current position by pressing the <INTERRUPT> key. Note that the question mark (?) works exactly like the slash key, except that it is used to search backwards in a file instead of forwards.

The last key that we'll discuss is the colon (:). When you type a colon, it is echoed on the status line as a prompt for an Ex command. You can then type in any Ex command, followed by an <ESC> or <RETURN> and the given Ex command will be executed.

6.3.4 Text Objects

The editing operations of the Vi editor are in most cases based on the notion of a text object. A text object is any sequence of consecutive characters in a line or any sequence of consecutive lines in a file. In general, text objects are delimited by the cursor and a cursor movement command. The character on which the cursor sits delimits one end of an intra-line object such as a word; the current line delimits one end of a multi-line object such as a line number range. Because the location of the cursor is always known by Vi, text objects are normally specified by simply typing the appropriate cursor movement command. Thus, naked cursor movement commands move the cursor to the opposite end of a given text object. Text objects can be moved, changed, or deleted by combining the appropriate command operator with a cursor movement command.

6.4 Invoking and Exiting Vi

The Vi invocation syntax is as follows:

```
vi [-option ...] [+command] [filename ...]
```

The simplest form of this syntax, and the easiest way to enter Vi is to type:

```
vi
```

This allows you to work on an empty editing buffer. The most common way to enter Vi, however, is to specify one or more filenames as shown below:

vi filename ...

Filename arguments indicate files to be edited. You can also enter the editor and then move to a particular place in a file by giving Vi a single line number or search string argument, preceded by a plus (+).

Examples:

vi	<u>Edit empty editing buffer</u>
vi file	<u>Edit named file</u>
vi +123 file	<u>Goto line 123</u>
vi +45 file	<u>Goto line 45</u>
vi +/dog file	<u>Find first occurrence of "dog"</u>
vi +/tty file	<u>Find first occurrence of "tty"</u>

Vi may be invoked with any of the following options:

- t This option is equivalent to an initial tag command, editing the file containing the tag and positioning the editor at its definition.
- r This option is used in recovering after an editor or system crash, retrieving the last saved version of the named file or, if no file is specified, printing a list of saved files.
- x This option causes Vi to prompt for a key used to encrypt and decrypt the contents of the named files.
- R This sets a readonly option so that files can be viewed but not edited.

There are several ways to exit the editor. One way is to type:

:wq<RETURN>

This command writes the editing buffer to the file you are editing, quits the editor, and then returns you to the XENIX shell. Similarly, if you type

zz

the same thing happens, except that the editing buffer is written to the file you are editing only if you have made

any changes. The "zz" command is equivalent to the command:

```
:x<RETURN>
```

To abort an editing session, type

```
:q!<RETURN>
```

The exclamation mark (!) tells Vi to quit unconditionally. In this case, the editing buffer is not written out. If you type just

```
:q<RETURN>
```

the editor will not let you quit unless you have written out your file and will print the message:

```
No write since last change (":q!" overrides)
```

This tells you to use ":q!" if you really want to quit without writing out your file.

6.5 Vi Commands

Vi is a visual editor with a window on the file. What you see on the screen is Vi's notion of what the file contains. Most commands do not cause any change to the screen until the complete command is typed. Also, most commands may take a preceding count that specifies repetition of the command. This count parameter is not given in the syntax, but is implied unless overriden by some other prefix argument.

Should you get confused while typing a command, you can abort the command by typing an <INTERRUPT> character. Usually typing an <ESC> will produce the same result. When Vi gets an improperly formatted command it also rings a bell.

Following subsections describe Vi commands. These are not all of the commands available in Vi. By typing a colon, you can enter Ex command mode and enter Ex commands. For more information, see Section 6.6, "Ex Commands"

6.5.1 Cursor Movement

The cursor movement keys allow you to move your cursor around in a file. Note in particular the arrow keys (if available on your terminal), the "hjkl" cursor keys, and <SPACE>, <BKSP>, <CONTROL-N>, and <CONTROL-P>. These three sets of keys perform identical functions; the choice of which to use is entirely up to you.

Forward Space - l, <SPACE>, or -->

Syntax: l
<SPACE>
-->

Function: Move cursor forward one character. If a count is given, move forward count characters. Note that you cannot move past the end of the line.

Backspace - h, <BKSP>, or <-->

Syntax: h
<BKSP>
<-->

Function: Move cursor backward one character. If a count is given, move backwards count characters. Note that you cannot move past the beginning of the current line.

Next Line - +, <RETURN>, j, <CONTROL-N>, and <LF>

Syntax: +
<RETURN>

Function: Move cursor down to the beginning of the next line.

Syntax: j
<CONTROL-N>
<LF>
(down arrow)

Function: Move cursor down one line, remaining in the same column. Note the difference between these commands and the preceding set of next line commands which move to the beginning of the next line.

Previous Line - k, <CONTROL-P>, and -

Syntax: k
 <CONTROL-P>
 (up arrow)

Function: Move cursor up one line, remaining in the same column. If a count is given then the cursor is moved up a number of lines equal to the count.

Syntax: -

Function: Move cursor up to the beginning of the previous line. If a count is given then the cursor is moved up a number of lines equal to the count.

Beginning of Line - 0 and ^

Syntax: ^
 0

Function: Move cursor to the beginning of the current line. Note that 0 always moves the cursor to the first character of the current line. The circumflex (^) works somewhat differently: it moves to the first character on a line that is not a tab or a space. This is useful when editing files that have a great deal of indentation, such as program texts.

End of Line - \$

Syntax: \$

Function: Move cursor to the end of the current line. Note that the cursor resides on top of the last character on the line. If a count is given, then the cursor is moved forward count-1 lines to the end of the line.

Goto Line - G

Syntax: [linenumber]G

Function: Go to the beginning of the line specified by linenumber. If no linenumber is given, the cursor moves to the beginning of the last line in the file. To find the line number of the

current line, use <CONTROL-G>. To turn on automatic line numbering on your screen, type:

```
:set linenumber<RETURN>
```

Column - |

Syntax: [column] |

Function: Move cursor to the column in the current line given by column. If no column is given then the cursor is moved to the first column in the current line.

Word Forward - w and W

Syntax: w
W

Function: Move cursor forward to the beginning of the next word. The lowercase w command searches for a word defined as a string of alphanumeric characters separated by punctuation or whitespace (i.e., tab, newline, or space characters). The uppercase W command searches for a word defined as a string of non-whitespace characters.

Back Word - b and B

Syntax: b
B

Function: Move cursor backward to the beginning of a word. The lowercase b command searches backwards for a word defined as a string of alphanumeric characters separated by punctuation or whitespace (i.e., tab, newline, or space characters). The uppercase B command searches for a word defined as a string of non-whitespace characters. If the cursor is already within a word, then it moves backwards to the beginning of that word.

End - e and E

Syntax: e
E

Function: Move cursor to the end of a word. The lowercase e command moves the cursor to the last character of a word, where a word is defined as a string of alphanumeric characters separated by punctuation or whitespace (i.e., tab, newline, or space characters). The uppercase E moves the cursor to the last character of a word where a word is defined as a string of non-whitespace characters. If the cursor is already within a word, then it moves to the end of that word.

Sentence - (and)

Syntax: ()

Function: Move cursor to the beginning (left parenthesis) or end of a sentence (right parenthesis). A sentence is defined as a sequence of characters ending with a period (.), question mark (?), or exclamation mark (!), followed by either two spaces or a newline. A sentence begins on the first non-whitespace character following a preceding sentence. Sentences are also delimited by paragraph and section delimiters. See below.

Paragraph - { and }

Syntax: { }

Function: Move cursor to the beginning ({}) or end (}) of a paragraph. A paragraph is defined with the paragraphs option. By default, paragraphs are delimited by the nroff macros ".IP", ".LP", ".PP", ".QP", and ".bp". Paragraphs also begin after empty lines.

Section - [[and]]

Syntax:]]
 [[

Function: Move cursor to the beginning ([[or end ()]) of a section. A section is defined with the sections option. By default, sections are delimited by the nroff macros ".NH" and ".SH". Sections also start at formfeeds (<CONTROL-L>) and at lines beginning with a brace ({}).

Match Delimiter - %

Syntax: %

Function: Move cursor to a matching delimiter, where a delimiter is a parenthesis, a bracket, or a brace. This is useful when matching pairs of nested parentheses, brackets, and braces.

Home - H

Syntax: offsetH

Function: Home cursor to upper left corner of screen. Use this command to quickly move to the top of the screen. If an offset is given, then the cursor is homed offset-1 number of lines from the top of the screen. Note that the command "dH" deletes all lines from the current line to the top line shown on the screen.

Middle Screen - M

Syntax: M

Function: Move cursor to the beginning of the screen's middle line. Use this command to quickly move to the middle of the screen from either the top or the bottom. Note that the command "dM" deletes from the current line to the line specified by the M command.

Lower Screen - L

Syntax: [offset]L

Function: Move cursor to the lowest line on the screen. Use this command to quickly move to the bottom of the screen. If an offset is given, then the cursor is homed offset-1 number of lines from the bottom of the screen. Note that the command "dL" deletes all lines from the current line to the bottom line shown on the screen.

Previous Context - `` and ''

Syntax: ''
'character
``
`character

Function: Move cursor to previous context or to context marked with the m command. If the single quote or back quote is doubled, then the cursor is moved to previous context. If a single character is given after either quote, then the cursor is moved to the location of the specified mark as defined by the m command. Previous context is the location in the file of the last "non-relative" cursor movement. The single quote ('') syntax is used to move to the beginning of the line representing the previous context. The back quote (`) syntax is used to move to the previous context within a line.

6.5.2 The Screen Commands

The screen commands are not cursor movement commands and cannot be used in delete commands as the delimiters of text objects. However, the screen commands do move the cursor and are useful in paging or scrolling through a file. These commands are described below:

Page - <CONTROL-U> and <CONTROL-D>

Syntax: [size]<CONTROL-U>
[size]<CONTROL-D>

Function: Scroll screen up a half window (<CONTROL-U>) or down a half window (<CONTROL-D>). If size is

given, then the scroll is size number of lines. This value is remembered for all later scrolling commands.

Scroll - <CONTROL-F> and <CONTROL-B>

Syntax: <CONTROL-F>
 <CONTROL-B>

Function: Page screen forward and backwards. Two lines of continuity are kept between pages if possible. A preceding count gives the number of pages to move forward or backwards.

Status - <CONTROL-G>

Syntax: <CONTROL-G>
 <CONTROL-G>

Function: Print Vi status on status line. This gives you the name of the file you are editing, whether it has been modified, the current line number, the number of lines in the file, and the percentage of the file (in lines) that precedes the cursor.

Zero Screen - z

Syntax: [linenumber]z[size]<RETURN>
 [linenumber]z[size].
 [linenumber]z[size]-

Function: Redraw the display with the current line zeroed at the top, bottom, or middle of the screen. If you give a size, then the number of lines displayed is equal to size. If a preceding linenumber is given, then the given line is zeroed at the top of the screen. If the last argument is a <RETURN>, then the current line is zeroed at the top of the screen. If the last argument is a period (.), then the current line is zeroed in the middle of the screen. If the last argument is a minus sign (-), then the current line is zeroed at the bottom of the screen.

Redraw - <CONTROL-R> or <CONTROL-L>

Syntax: <CONTROL-R>
 <CONTROL-L>

Function: Redraw the screen. Use this command to erase any system messages that may scramble your screen. These messages do not affect the file you are editing.

6.5.3 Text Insertion

The text insertion commands always place you in insert mode. Exit from insert mode is always done by pressing <ESC>. The following insertion commands are "pure" insertion commands, no text is deleted when you use them. This differs from the text modification commands -- change, replace, and substitute -- which delete and then insert text in one operation.

Insert - i and I

Syntax: i[text]<ESC>
 I[text]<ESC>

Function: Insert text in editing buffer. The lowercase i command places you in insert mode. Text is inserted before the character beneath the cursor. To insert a newline, just press a <RETURN>. Exit insert mode by typing the <ESC> key. The uppercase I command places you in insert mode, but begins text insertion at the beginning of the current line, rather than before the cursor.

Append - a and A

Syntax: a[text]<ESC>
 A[text]<ESC>

Function: Append text to editing buffer. The lowercase a command works exactly like the lowercase i command, except that text insertion begins after the cursor and not before. This is the only way to add text to the end of a line. The uppercase A command begins appending text at the end of the current line rather than after the cursor.

Open New Line - o and O

Syntax: o[text]<ESC>
 O[text]<ESC>

Function: Open a new line and insert text. The lowercase o command opens a new line below the current line; uppercase O opens a new line above the current line. After the new line has been opened, both these commands work like the I command.

6.5.4 Text Deletion

Many of the text deletion commands use the letter "d" as an operator. This operator deletes text objects delimited by the cursor and a cursor movement command. Deleted text is always saved away in a buffer. The delete commands are described below:

Delete Character - x and X

Syntax: x
 X

Function: Delete a character. The lowercase x command deletes the character beneath the cursor. With a preceding count, cnt characters are deleted to the left beginning with the character beneath the cursor. This is the quick and easy way to delete a few characters. The uppercase X command deletes the character just before the cursor. With a preceding count, cnt characters are deleted backwards, beginning with the character just before the cursor.

Delete - d and D

Syntax: dcursor-movement
 dd
 D

Function: Delete text object. The lowercase d command takes a cursor-movement as an argument. If the cursor-movement is an intra-line command, then deletion takes place from the cursor to the end of the text object delimited by the cursor-movement. Deletion forward deletes the

character beneath the cursor; deletion backwards does not. If the cursor-movement is a multi-line command, then deletion takes place from and including the current line to the text object delimited by the cursor-movement.

The dd command deletes whole lines. The uppercase D command deletes from and including the cursor to the end of the current line.

Deleted text is automatically pushed on a stack of buffers numbered 1 through 9. The most recently deleted text is placed in a special delete buffer that is logically buffer 0. This special buffer is the default buffer for all delete, put, and yank commands. The buffers 1 through 9 can be accessed with the p and P ("put") commands using the double quotation mark ("") to specify the number of the buffer. For example

"4p

puts the contents of delete buffer number 4 in your editing buffer just below the current line. Note that the last deleted text is "put" by default and does not need a preceding buffer number.

6.5.5 Text Modification

The text modification commands all involve the replacement of text with other text. This means that some text will necessarily be deleted. All text modification commands can be "undone" with the u command, discussed below:

Undo - u and U

Syntax: u
U

Function: Undo the last insert or delete command. The lowercase u command undoes the last insert or delete command. This means that after an insert, u deletes text; and after a delete, u inserts text. For the purposes of undo, all text modification commands are considered insertions.

The uppercase U command restores the current line to its state before it was edited, no matter how many times the current line has been edited since you moved to it.

Repeat - .

Syntax: .

Function: Repeat the last insert or delete command. A special case exists for repeating the p and P "put" commands. When these commands are preceded by the name of a delete buffer, then successive u commands print out the contents of the delete buffers.

Change - c and C

Syntax: ccursor-movement text<ESC>
Ctext<ESC>
cctext<ESC>

Function: Change a text object and replace it with text. Text is inserted as with the i command. A dollar sign (\$) marks the extent of the change. The c command changes arbitrary text objects delimited by the cursor and a cursor-movement. The C and cc commands affect whole lines and are identical in function.

Replace - r and R

Syntax: rchar
Rtext<ESC>

Function: Overstrike character or line with char or text, respectively. Use r to overstrike single characters and R to overstrike whole lines.

Substitute - s and S

Syntax: stext<ESC>
Stext<ESC>

Function: Substitute current character or current line with text. Use s to replace a single character with new text. Use S to replace the current

line with new text. If a preceding count is given, then text substitutes for count number of characters or lines depending on whether the command is s or S, respectively.

Filter - !

Syntax: !cursor-movement cmd<RETURN>

Function: Filter the text object delimited by the cursor and the cursor-movement through the XENIX command, cmd. For example, the following command sorts all lines between the cursor and the bottom of the screen, substituting the designated lines with the sorted lines:

!Lsort

Join Lines - J

Syntax: J

Function: Join the current line with the following line. If a count is given, then count lines are joined.

Shift - < and >

Syntax: >[cursor-movement]
<[cursor-movement]
>>
<<

Function: Shift text left (>) or right (<). Text is shifted by the value of the option shiftwidth, which is normally set to eight spaces. Both the > and < commands shift all lines in the text object delimited by the current line and cursor-movement. The >> and << commands affect whole lines. All versions of the command can take a preceding count that acts to multiply the number of objects affected.

6.5.6 Text Movement

The text movement commands move text in and out of the named buffers "a"--"z" and out of the delete buffers 1-9. These commands either "yank" text out of the editing buffer and into a named buffer or "put" text into the editing buffer from a named buffer or a delete buffer. By default, text is put and yanked from the "unnamed buffer", which is also where the most recently deleted text is placed. Thus it is quite reasonable to delete text, move your cursor to the location where you want the deleted text placed, and then put the text back into the editing buffer at this new location with the p or P command.

The named buffers are most useful for keeping track of several chunks of text that you want to keep on hand for later access, movement, or rearrangement. These buffers are named with the letters "a" through "z". To refer to one of these buffers (or one of the numbered delete buffers) in a command such as a put, yank, or delete command, use a quotation mark. For example, to yank a line into the buffer named a, type:

```
"ayy
```

To put this text back into the file, type:

```
"ap
```

If you delete text into the buffer named A rather than a, then text is appended to the buffer.

Note that the contents of the named buffers are not destroyed when you switch files. Therefore, you can delete or yank text into a buffer, switch files, and then do a put. Beware, buffer contents are destroyed when you exit the editor, so be careful.

Put - p and P

Syntax: [alphanumeric]p
[alphanumeric]P

Function: Put text from a buffer into the editing buffer. If no buffer name is specified, then text is put from the unnamed buffer. The lowercase p command puts text either below the current line or after the cursor, depending on whether the buffer contains a partial line or not. The uppercase P command puts text either above the

current line or before the cursor, again depending on whether the buffer contains a partial line or not.

Yank - y and Y

Syntax: ["letter]cursor-movement
["letter]yy
["letter]Y

Function: Copy text in the editing buffer to a named buffer. If no buffer name is specified, then text is yanked into the unnamed buffer. If an uppercase letter is used, then text is appended to the buffer and does not overwrite and destroy the previous contents.

The **Y** and **yy** commands yank lines.

6.5.7 Searching

The search commands search either forward or backwards in the editing buffer for a regular expression. For more information about regular expressions, see Section 6.8 "Regular Expressions".

Search - / and ?

Syntax: /[string]/<RETURN>
?[string]?<RETURN>

Function: Search forward (/) or backward (?) for string. A string is actually a regular expression as defined in Section 6.8 "Regular Expressions." The trailing delimiter is not required. See also the ignorecase and magic options.

Next String - n and N

Syntax: n
N

Function: Repeat the last search command. The **n** command repeats the search in the same direction as the last search command. The **N** command repeats the search in the opposite direction of the last search command.

Find Character - f and F

Syntax: fchar
 Fchar
 ;
 '

Function: Find character on current line. The lowercase f searches forward on the line; the uppercase F searches backwards. The semi-colon (;) repeats the last character search. The comma, (,), reverses the direction of the search.

To Character - t and T

Syntax: tchar
 Tchar
 ;
 '

Function: Move cursor up to but not on char. The semi-colon (;) repeats the last character search. The comma, (,) reverses the direction of the search.

Mark - m

Syntax: mletter

Function: Mark place in file with a lowercase letter. You can move to a mark using the "to mark" commands described below. It is often useful to create a mark, move the cursor, and then delete from the cursor to the mark with the following command: "d'letter".

To Mark - ' and '

Syntax: 'letter
 'letter

Function: Move to letter. These commands let you move to the location of a mark. Marks are denoted by single lowercase alphabetic characters. Before you can move to a mark, it must first have been created with the **m** command. The back quote (`) moves you to the exact location of the mark within a line; the forward quote (`) moves you

to the beginning of the line containing the mark. Note that these commands are also legal cursor movement commands.

6.5.8 Exit and Escape Commands

There are several commands that are used to escape from Vi command mode and to exit the editor. These are described below:

Ex Escape - :

Syntax: :

Function: Enter Ex escape mode to execute an Ex command. The colon appears on the status line as a prompt for an Ex command. You then can enter an Ex command line terminated by either a <RETURN> or an <ESC> and the Ex command will execute. You then will be prompted to press <RETURN> to return to Vi command mode. During the input of the Ex command line or during execution of the Ex command you may press <INTERRUPT> to abort what you are doing and return to Vi command mode.

Exit Editor - ZZ

Syntax: ZZ

Function: Exit Vi and write out the file if any changes have been made. This returns you to the shell from which you invoked Vi.

Quit to Ex - Q

Syntax: Q

Function: Enter the Ex editor. When you do this, you will still be editing the same file. You can return to Vi by typing the "vi" command from Ex.

6.6 Ex Commands

Typing the colon (:) escape command when in command mode, produces a colon prompt on the status line. This prompt is for a command available in the line-oriented editor, Ex. In general, Ex commands let you write out or read in files, escape to the shell, or switch editing files.

Many of these commands perform actions that affect the "current" file by default. The current file is normally the file that you named when you invoked Vi, although the current file can be changed with the "file" command, f, or with the "next" command, n. In most respects, these commands are identical to similar commands for the editor, ed. All such Ex commands are terminated by either a <RETURN> or an <ESC>. We shall use a <RETURN> in our examples. Command entry is terminated by typing an <INTERRUPT>.

6.6.1 Command Structure

Most Ex command names are English words, and initial prefixes of the words are acceptable abbreviations. In descriptions, only the abbreviation is discussed, since this is the most frequently used form of the command. The ambiguity of abbreviations is resolved in favor of the more commonly used commands. As an example, the command substitute can be abbreviated s while the shortest available abbreviation for the set command is se.

Most commands accept prefix addresses specifying the lines in the file that they are to effect. A number of commands also may take a trailing count specifying the number of lines to be involved in the command. Counts are rounded down if necessary. Thus, the command "10p" will print the tenth line in the buffer while "move 5" will move the current line after line 5.

Some commands take other information or parameters, stated after the command name. Examples might be option names in a set command, such as "set number", a filename in an edit command, a regular expression in a substitute command, or a target address for a copy command, such as

1,5 copy 25

Also, a number of commands have two distinct variants. The variant form of the command is invoked by placing an exclamation mark (!) immediately after the command name. Some of the default variants may be controlled by options;

in this case, the exclamation mark turns off the meaning of the default.

In addition, many commands take flags, including the characters "#", "p" and "l". A "p" or "l" must be preceded by a blank or tab. In this case, the command abbreviated by these characters is executed after the command completes. Since Ex normally prints the new current line after each change, p is rarely necessary. Any number of plus (+) or minus (-) characters may also be given with these flags. If they appear, the specified offset is applied to the current line value before the printing command is executed.

Most commands that change the contents of the editor buffer give feedback if the scope of the change exceeds a threshold given by the report option. This feedback helps to detect undesirably large changes so that they may be quickly and easily reversed with the undo command. After commands with global effect, you will be informed if the net change in the number of lines in the buffer during this command exceeds this threshold.

6.6.2 Command Addressing

The following specifies the line addressing syntax for Ex commands:

.	The current line. Most commands leave the current line as the last line which they affect. The default address for most commands is the current line, thus "." is rarely used alone as an address.
<u>n</u>	The <u>n</u> th line in the editor's buffer, lines being numbered sequentially from 1.
\$	The last line in the buffer.
%	An abbreviation for "1,\$", the entire buffer.
+ <u>n</u> or - <u>n</u>	An offset, <u>n</u> relative to the current buffer line. The forms ".+3" "+3" and "+++" are all equivalent. If the current line is line 100 they all address line 103.
/ <u>pat</u> / or ? <u>pat</u> ?	Scan forward and backward respectively for a line containing <u>pat</u> , a regular

expression. The scans normally wrap around the end of the buffer. If all that is desired is to print the next line containing pat, then the trailing slash (/) or question mark (?) may be omitted. If pat is omitted or explicitly empty, then the string matching the last specified regular expression is located. The forms "/<RETURN>" and "?<RETURN>" scan using the last named regular expression. After a substitute, "//<RETURN>" and "??<RETURN>" would scan using that substitute's regular expression.

'' or 'x

Before each non-relative motion of the current line dot (.), the previous current line is marked with a label, subsequently referred to with two single quotes (''). This makes it easy to refer or return to this previous context. Marks are established with the Vi **m** command, using a single lowercase letter as the name of the mark. Marked lines are later referred to with the notation

'x.

where x is the name of a mark.

Addresses to commands consist of a series of addressing primitives, separated by a colon (:) or a semicolon (;). Such address lists are evaluated left to right. When addresses are separated by a semicolon (;) the current line (.) is set to the value of the previous addressing expression before the next address is interpreted. If more addresses are given than the command requires, then all but the last one or two are ignored. If the command takes two addresses, the first addressed line must precede the second in the buffer. Null address specifications are permitted in a list of addresses, the default in this case is the current line ".;"; thus ",100" is equivalent to "..,100". It is an error to give a prefix address to a command which expects none.

6.6.3 Command Format

The following is the format for all Ex commands:

[address] [command] [!] [parameters] [count] [flags]

All parts are optional depending on the particular command and its options. Command descriptions follow.

6.6.4 Argument List Commands

The argument list commands allow you to easily work on a set of files. This is done by remembering the list of filenames that are specified when you invoke Vi. The args command lets you examine this list of filenames. The file command gives you information about the current file. The n (next) command lets you edit the next file in the argument list or change the list. And the rewind command lets you restart editing the files in the list. All of these commands are described below:

args The members of the argument list are printed, with the current argument delimited by brackets. For example, a list might look like this:

file1 file2 [file3] file4 file5

Here, the current file is file3.

f Prints the current filename, whether it has been modified since the last write command, whether it is readonly, the current line, the number of lines in the buffer, and the percentage of the way through the buffer of the current line. In the rare case that the current file is "[Not edited]" this is noted also; in this case you have to use the form "w!" to write to the file, since the editor is not sure that a w command will not destroy a file unrelated to the current contents of the buffer.

f file The current filename is changed to file which is considered "[Not edited]".

n The next file from the command line argument list is edited.

n! The variant suppresses warnings about the modifications to the buffer not having been written out, discarding irretrievably any changes

which may have been made.

n [+command] filelist

The specified filelist is expanded and the resulting list replaces the current argument list; the first file in the new list is then edited. If command is given (it must contain no spaces), then it is executed after editing the first such file.

rew The argument list is rewound, and the first file in the list is edited.

rew! Rewinds the argument list discarding any changes made to the current buffer.

6.6.5 Edit Commands

To edit a file other than the one you are currently editing, you will often use one of the variations of the e command.

In the following discussions, note that the name of the current file is always remembered by Vi and is specified by a percent sign (%). The name of the previous file in the editing buffer is specified by a number sign (#). Thus, to edit the last file in the editing buffer, you could type:

:e #

The edit commands are described below:

e file

Used to begin an editing session on a new file. The editor first checks to see if the buffer has been modified since the last w command was issued. If it has been, a warning is issued and the command is aborted. The command otherwise deletes the entire contents of the editor buffer, makes the named file the current file and prints the new filename. After ensuring that this file is sensible, (i.e., that it is not a binary file, directory, or a device), the editor reads the file into its buffer. If the read of the file completes without error, the number of lines and characters read is printed on the status line. If there were any non-ASCII characters in the file they are stripped of their non-ASCII high bits, and any null characters in the file are discarded. If none of these errors occurred, the file is considered edited. If the last line of the input file is missing the trailing newline character, it will be supplied and a complaint will be issued.

The current line is initially the first line of the file.

e! file The variant form suppresses the complaint about modifications having been made and not written from the editor buffer, thus discarding all changes which have been made before editing the new file.

e +n file Causes the editor to begin editing at line n rather than at the first line. The argument n may also be an editor command containing no spaces; for example, "+/pat".

<CONTROL-^> This is a short-hand equivalent for ":e #<RETURN>", which returns to the previous position in the last edited file. If you do not want to write the file you should use ":e! #<RETURN>" instead.

6.6.6 Write Commands

The write commands let you write out all or part of your editing buffer to either the current file or to some other file. These commands are described below:

w file Writes changes made back to file, printing the number of lines and characters written. Normally, file is omitted and the text goes back where it came from. If a file is specified, then text will be written to that file. The editor writes to a file only if it is the current file and is edited, or if the file does not exist. Otherwise, you must give the variant form w! to force the write. If the file does not exist it is created. The current filename is changed only if there is no current filename; the current line is never changed.

If an error occurs while writing the current and edited file, the editor considers that there has been "No write since last change" even if the buffer had not previously been modified.

w>> file Appends the buffer contents at the end of an existing file. Previous file contents are not destroyed.

w! name Overrides the checking of the normal write command, and writes to any file which the system permits.

w !command Writes the specified lines into command. Note the difference between w! which overrides checks and w ! which writes to a command. The output of this command is displayed on the screen and not inserted in the editing buffer.

6.6.7 Read Commands

The read commands let you read text into your editing buffer at any location you specify. The text you read in must be at least one line long, and can be either a file or the output from a command.

r file Places a copy of the text of the given file in the editing buffer after the specified line. If no file is given then the current filename is used. The current filename is not changed unless there is none, in which case the file becomes the current name. If the file buffer is empty and there is no current name then this is treated as an e command.

Address 0 is legal for this command and causes the file to be read at the beginning of the buffer. Statistics are given as for the e command when the r successfully terminates. After an r the current line is the last line read.

r !command Reads the output of command into the buffer after the specified line. A blank or tab before the exclamation mark (!) is mandatory.

6.6.8 Quit Commands

There are several ways to exit Vi. Some abort the editing session, some write out the editing buffer before exiting, and some warn you if you decide to exit without writing out the buffer. All of these ways of exiting are described below:

q Causes Vi to terminate. No automatic write of the editor buffer to a file is performed. However, Vi issues a warning message if the file has changed

since the last **w** command was issued, and does not quit. Vi will also issue a diagnostic if there are more files in the argument list left to edit. Normally, you will wish to save your changes, and you should give a **w** command. If you wish to discard them, use the "q!" command variant.

- q!** Quits from the editor, discarding changes to the buffer without complaint.
- wq name** Like a **w** and then a **q** command.
- wq! name** This variant overrides checking on the sensibility of the **w** command, as does "**w!**"
- x name** If any changes have been made and not written, writes the buffer out and then quits. Otherwise, it just quits.

6.6.9 Global and Substitute Commands

The global and substitute commands allow you to perform complex changes to a file in a single command. Learning how to use these commands is a must for the serious user of Vi. See also Section 6.8, "Regular Expressions."

g/pat/cmds

The **g** command has two distinct phases. In the first phase, each line matching pat in the editing buffer is marked. Next, the given command list is executed with dot (.) initially set to each marked line.

The command list consists of the remaining commands on the current input line and may continue to multiple lines by ending all but the last such line with a backslash (\). This multiple-line option will not work from within Vi, you must switch to Ex to do it. If cmds (or the trailing slash (/) delimiter) is omitted, then each line matching pat is printed.

The **g** command itself may not appear in cmds. The options autoprint and autoindent are inhibited during a global command and the value of the report option is temporarily infinite, in deference to a report for the entire global. Finally, the context mark (' or `) is set to the value of dot (.) before the global command begins and is not changed during a global command.

The following list of global commands, most of them substitutions, covers the most frequent uses of the global command. These examples are well worth studying.

g/s1/p This command simply prints all lines that contain the string "s1".

g/s1/s//s2/ This command substitutes the first occurrence of "s1" on all lines that contain it with the string "s2".

g/s1/s//s2/g This command substitutes all occurrences of "s1" with the string "s2". This includes multiple occurrences of "s1" on a line.

g/s1/s//s2/gp This command works the same as the preceding example, except that in addition, all changed lines are printed on the screen.

g/s1/s//s2/gc This command asks you to confirm that you want to make each substitution of the string "s1" with the string "s2". If you type a "y" then the substitution is made, otherwise it is not.

g/s0/s/s1/s2/g This command marks all those lines that contain the string "s0", and then for those lines only, it substitutes all occurrences of the string "s1" with "s2".

g!/pat/cmds This variant form of g runs cmds at each line not matching pat.

s/pat/repl/options On each specified line, the first instance of pattern pat is replaced by replacement pattern repl. If the global indicator option character "g" appears, then all instances are substituted. If the confirm indication character "c" appears, then before each substitution the line to be substituted is printed on the screen with the

string to be substituted marked with circumflex (^) characters. By typing a "y", you cause the substitution to be performed; any other input causes no change to take place. After an s command the current line is the last line substituted.

v/pat/cmds

A synonym for the global command variant g!, running the specified cmds on each line which does not match pat.

6.6.10 Text Movement Commands

The text movement commands are largely superseded by commands available in Vi command mode. However, the following two commands are still quite useful.

co addr flags

A copy of the specified lines is placed after addr, which may be "0". The current line "." addresses the last line of the copy.

[range]maddr

The m command moves the specified lines specified by range to be after addr. For example, "m+" swaps the current line and the following line, since the default range is just the current line. The first of the moved lines becomes the current line (dot).

6.6.11 Shell Escape Commands

You will often want to escape from the editor to execute normal XENIX commands. You may also want to change your working directory so that your editing can be done with respect to a different working directory. These operations are described below:

cd directory

The specified directory becomes the current directory. If no directory is specified, the current value of the home option is used as the target directory. After a cd the current file is not considered to have been edited so that write restrictions on pre-existing files still apply.

sh

A new shell is created. You may invoke as many commands as you like in this shell. To return to

Vi, you must type a <CONTROL-D> to terminate the shell.

!command

The remainder of the line after the "!" character is sent to a shell to be executed. Within the text of command the characters "%" and "#" are expanded as the filenames of the current file and the last edited file and the character "!" is replaced with the text of the previous command. Thus, in particular, "!!" repeats the last such shell escape. If any such expansion is performed, the expanded line is echoed. The current line is unchanged by this command.

If there has been "[No write]" of the buffer contents since the last change to the editing buffer, then a diagnostic is printed before the command is executed as a warning. A single exclamation (!) is printed when the command completes.

6.6.12 Other Commands

The following command descriptions explain how to use miscellaneous Ex commands that do not fit into the above categories:

nu

Prints each specified line preceded by its buffer line number. The current line is left at the last line printed. To get automatic line numbering of lines in the buffer, set the number option.

preserve

The current editor buffer is saved as though the system had just crashed. This command is for use only in emergencies when a w command has resulted in an error and you don't know how to save your work.

=

Prints the line number of the addressed line. The current line is unchanged.

recover file

Recovers file from the system save area. The system saves a copy of the editing buffer only if you have made changes to the file, the system crashes, or you execute a **preserve** command. Except when you use **preserve** you will be notified by mail when a file is saved.

set argument

With no arguments, **set** prints those options whose values have been changed from their defaults; With the argument all it prints all of the option values.

Giving an option name followed by a question mark (?) causes the current value of that option to be printed. The "?" is unnecessary unless the option is Boolean valued. Switch options are given values either by the form "set option" to turn them on or "set option" to turn them off; string and numeric options are assigned via the form "set option=value".

More than one parameter may be given to **set**; they are interpreted left to right. for more information, see Section 6.7, "Start-Up Files and Options."

tag label

The focus of editing switches to the location of label. If Vi has to, it will switch to a different file in the current directory to find label. If you have modified the current file before giving a tag command, you must first write it out. If you give another tag command with no argument, then the previous label is used.

Similarly, if you type only a <CONTROL-]>, Vi searches for the word immediately after the cursor as a tag. This is equivalent to typing ":ta", this word, and then a <RETURN>.

The tags file is normally created by a program such as **ctags**, and consists of a number of lines with three fields separated by blanks or tabs. The first field gives the name of the tag, the second the name of the file where the tag resides, and the third gives an addressing form which can be used by the editor to find the tag. This field is usually a contextual scan using

/pat/

to be immune to minor changes in the file. Such scans are always performed as if the nomagic option was set. The tag names in the tags file must be sorted alphabetically.

6.7 Start-Up Files and Options

There are a number of options that can be set to affect the Vi environment. These can be set with the Ex set command either while editing or immediately after Vi is invoked in the Vi start-up file, .exrc.

The first thing that must be done, however, before you can use Vi, is to set the terminal type so that Vi understands how to talk to the particular terminal you are using. This is the subject of the next subsection.

6.7.1 Setting the Terminal Type

To run Vi, the shell variable TERM must be defined and exported to your shell environment. How you do this depends on which shell you are using. You can usually determine which shell you are using by examining the prompt character. The normal shell prompts with a dollar sign (\$); the C shell prompts with a percent sign (%).

Once you have determined which shell you are using, you need to find the name of your terminal type. For these examples, we will suppose that you are using an HP 2621 terminal. In the file /etc/termcap is a description of the capabilities of this terminal. Each terminal capability description has a unique name that corresponds to the type of a terminal supported by Vi. For the HP 2621, this "termcap" name is "2621".

6.7.1.1 The Normal Shell

To set your terminal type to 2621 you would place the following commands in the file .profile:

```
TERM=2621  
export TERM
```

There are various ways of having this dynamically or semi-automatically done when you log in. Suppose you usually dial in on a 2621. You want to tell this to the machine, but still have it work when you use a another hardwired terminal. One way is to place the following sequence of commands

```
tset -s -d 2621 > tset$$  
. tset$$  
rm tset$$
```

in your .profile file. The above line says that if you are

dialing in you are on a 2621, but if you are on a hardwired terminal it figures out your terminal type from an on-line list. For the above sequence of commands to work, be sure that in your .profile file you have first set your PATH variable so that it includes the current directory

6.7.1.2 The C Shell

To set your terminal type to 2621 for the C shell, you would place the following commands in the file .login:

```
setenv TERM 2621
```

To specify your terminal type dynamically when you log in, you could use a procedure parallel to that discussed above for the normal shell:

```
tset -s -d 2621 > tset$$
source tset$$
rm tset$$
```

Place these commands in the file .login.

6.7.2 The .exrc File

Each time Vi is invoked, it reads commands from the file named .exrc in your home directory. This file normally sets the user's preferred options so that this need not be done each time Vi is invoked. A sample .exrc file follows:

```
set number
set wrapmargin=20
set errorbells
set ignorecase
set autoindent
```

Each of the above options is described in more detail below.

6.7.3 Options

There are only two kinds of options: switch options and string options. A switch option is either on or off. A switch is turned off by prefixing the word no to the name of the switch within a set command. String options are strings of characters that are assigned values with the syntax option=string. Multiple options may be specified on a line. Vi options are listed below:

autoindent, ai default: noai

Can be used to ease the preparation of structured program text. For each line created by an append, change, insert, open, or substitute operation, Vi looks at the preceding line to determine and insert an appropriate amount of indentation. To back the cursor up to the preceding tab stop one can press <CONTROL-D>. The tab stops going backwards are defined at multiples of the shiftwidth option. You cannot backspace over the indent, except by typing a <CONTROL-D>.

Specially processed in this mode is a line with no characters added to it, which turns into a completely blank line (the white space provided for the autoindent is discarded.) Also specially processed in this mode are lines beginning with an up-arrow (^) and immediately followed by a <CONTROL-D>. This causes the input to be repositioned at the beginning of the line, but retains the previous indent for the next line. Similarly, a "0" followed by a <CONTROL-D> repositions the cursor at the beginning but without retaining the previous indent. Autoindent doesn't happen in global commands.

autoprint ap default: ap

Causes the current line to be printed after each Ex copy, move, or substitute command. This has the same effect as supplying a trailing "p" to each such command. Autoprint is suppressed in globals, and only applies to the last of many commands on a line.

autowrite, aw default: noaw

Causes the contents of the buffer to be written to the current file if you have modified it and give an **next**, **rewind**, **tag**, or **!** command, or a <CONTROL-^> (switch files) or <CONTROL-]> (tag go to) command.

beautify, bf default: nobeautify

Causes all control characters except tab, new line and form-feed to be discarded from the input. A complaint is registered the first time a backspace character is discarded. Beautify does not apply to command input.

directory, dir default: dir=/tmp

Specifies the directory in which Vi places the editing buffer file. If this directory is not

writable, then the editor will exit abruptly when it fails to write to the buffer file.

edcompatible default: noedcompatible

Causes the presence or absence of g and c suffixes on substitute commands to be remembered, and to be toggled by repeating the suffixes. The suffix r makes the substitution be as in the command, instead of like &.

hardtabs, ht default: ht=8

Gives the boundaries on which terminal hardware tabs are set or on which the system expands tabs.

ignorecase, ic default: noic

All uppercase characters in the text are mapped to lowercase in regular expression matching. In addition, all uppercase characters in regular expressions are mapped to lowercase except in character class specifications enclosed in brackets.

lisp default: nolisp

Autoindent indents appropriately for LISP code, and the () { } [[and]] commands are modified to have meaning for LISP.

list default: nolist

All printed lines will be displayed unambiguously, showing tabs and end-of-lines.

magic default: magic

If nomagic is set, the number of regular expression metacharacters is greatly reduced, with only up-arrow (^) and dollar sign (\$) having special effects. In addition the metacharacters "~~" and "&" in replacement patterns are treated as normal characters. All the normal metacharacters may be made magic when nomagic is set by preceding them with a backslash (\).

mesg default: nomesg

Causes write permission to be turned off to the terminal while you are in visual mode, if nomesg is set. This prevents people writing to your screen with the XENIX write command and scrambling your screen as you edit.

number, n default: nonumber

Causes all output lines to be printed with their line numbers.

optimize, opt default: optimize
Throughput of text is expedited by setting the terminal to not do automatic carriage returns when printing more than one line of output, thus greatly speeding output on terminals without addressable cursors when text with leading whitespace is printed.

paragraphs, para default: para=IPLPPPQPPP LIBp
Specifies the paragraphs for the { and } operations. The pairs of characters in the option's value are the names of the nroff macros which start paragraphs.

redraw default: noredraw
The editor simulates (using great amounts of output), an intelligent terminal on a dumb terminal. Useful only at very high speed.

report default: report=5
Specifies a threshold for feedback from commands. Any command that modifies more than the specified number of lines will provide feedback as to the scope of its changes. For global commands and the undo command which have potentially far reaching scope, the net change in the number of lines in the buffer is presented at the end of the command, subject to this same threshold. Thus notification is suppressed during a g command on the individual commands performed.

scroll default: scroll=1/2 window
Determines the number of logical lines scrolled when an end-of-file is received from a terminal input in command mode, and the number of lines printed by a command mode z command (double the value of scroll).

sections default: sections=SHNHH HU
Specifies the section macros for the [[and]] operations. The pairs of characters in the option's value are the names of the nroff macros that start paragraphs.

shell, sh default: sh=/bin/sh
Gives the pathname of the shell forked for the shell escape command "!", and by the shell command. The default is taken from SHELL in the environment, if present.

shiftwidth, sw default: sw=8
Gives the width of a software tab stop, used in reverse tabbing with <CONTROL-D> when using autoindent to append text, and by the shift commands.

showmatch, sm default: nosm
When a) or } is typed, move the cursor to the matching (or { for one second if this matching character is on the screen. Useful with LISP.

tabstop, ts default: ts=8
The editor expands tabs in the input file to be on tabstop boundaries for the purposes of display.

taglength, tl default: tl=0
Tags are not significant beyond this many characters. A value of zero (the default) means that all characters are significant.

tags default: tags=/usr/lib/tags
A path of files to be used as tag files for the tag command. A requested tag is searched for in the specified files, sequentially. By default files named tag are searched for in the current directory and in /usr/lib.

term default=value of shell TERM variable
The terminal type of the output device.

terse default: noterse
Shorter error diagnostics are produced for the experienced user.

warn default: warn
Warn if there has been "[No write since last change]" before a shell escape command (!).

window default: window=speed dependent
This specifies the number of lines in a text window. The default is 8 at slow speeds (600 baud or less), 16 at medium speed (1200 baud), and the full screen (minus one line) at higher speeds.

w300, w1200, w9600
These are not true options but set window (above) only if the speed is slow (300), medium (1200), or high (9600), respectively.

wrapscan, ws default: ws
Searches using the regular expressions in

addressing will wrap around past the end of the file.

wrapmargin, *wm* default: *wm*=0

Defines a margin for automatic wrap over of text during input.

writeany, *wa* default: *nowa*

Inhibits the checks normally made before **write** commands, allowing a write to any file that the system protection mechanism will allow.

6.8 Regular Expressions

A regular expression specifies a set of strings of characters. A member of this set of strings is said to be matched by the regular expression. Vi remembers two previous regular expressions: the previous regular expression used in a substitute command and the previous regular expression used elsewhere, referred to as the previous scanning regular expression. The previous regular expression can always be referred to by a null regular expression: e.g., "://" or "??".

The regular expressions allowed by Vi are constructed in one of two ways depending on the setting of the magic option. The Ex and Vi default setting of magic gives quick access to a powerful set of regular expression metacharacters. The disadvantage of magic is that the user must remember that these metacharacters are magic and precede them with the character "\\" to use them as "ordinary" characters. With nomagic set, regular expressions are much simpler, there being only two metacharacters. The power of the other metacharacters is still available by preceding the now ordinary character with a "\\". Note that "\\" is thus always a metacharacter. In this discussion the magic option is assumed. With nomagic the only special characters are "^" at the beginning of a regular expression, dollar sign (\$) at the end of a regular expression, and backslash (\). The tilde (~) and the ampersand (&) also lose their special meanings related to the replacement pattern of a substitute.

The following basic constructs are used to construct magic mode regular expressions.

char

An ordinary character matches itself. Ordinary characters are any characters except a circumflex (^) at the beginning of a line, a dollar sign (\$) at the end of line, an asterisk (*) as any character other than the

first, and any of the following characters:

. \ [^

These characters must be escaped (i.e., preceded) by a backslash (\) if they are to be treated as ordinary characters.

^ At the beginning of a pattern this forces the match to succeed only at the beginning of a line.

\$ At the end of a regular expression this forces the match to succeed only at the end of the line.

. Matches any single character except the new-line character.

\< Forces the match to occur only at the beginning of a "word;" that is, either at the beginning of a line, or just before a letter, digit, or underline and after a character not one of these.

\> Similar to "\<", but matching the end of a "word," i.e. either the end of the line or before a character which is not a letter, a digit, or the underline character.

[string] Matches any single character in the class defined by string. Most characters in string define themselves. A pair of characters separated by a dash (-) in string defines the set of characters between the specified lower and upper bounds, thus "[a-z]" as a regular expression matches any single lowercase letter. If the first character of string is a circumflex (^) then the construct matches those characters which it otherwise would not. Thus "[^a-z]" matches anything but a lowercase letter or a newline. To place any of the characters caret, left bracket, or dash in string they must be escapes with a preceding backslash (\).

The concatenation of two regular expressions first matches the leftmost regular expression and then the longest string which can be recognized as a regular expression. The first part of this new regular expression matches the first regular expression and the second part matches the second.

Any of the single character matching regular expressions mentioned above may be followed by an asterisk or "star" (*) to form a regular expression which matches zero or more adjacent occurrences of the characters matched by the prefixing regular expression. The tilde (~) may be used in a regular expression, and matches the text which defined the replacement part of the last s command. A regular expression may be enclosed between the sequences "\(" and "\)" with side effects in substitute replacement patterns.

The basic metacharacters for the replacement pattern are the ampersand (&) and the asterisk (*) these are given as "\&" and "\~" when nomagic is set. Each instance of the ampersand is replaced by the characters matched by the regular expression. The metacharacter "~" stands, in the replacement pattern, for the defining text of the previous replacement pattern.

Other metasequences possible in the replacement pattern are always introduced by a backslash (\). The sequence "\n" is replaced by the text matched by the n-th regular subexpression enclosed between "\(" and "\)". When nested, parenthesized subexpressions are present, n is determined by counting occurrences of "\(" starting from the left. The sequences "\u" and "\l" cause the immediately following character in the replacement to be converted to upper or lowercase, respectively, if this character is a letter. The sequences "\U" and "\L" turn such conversion on, either until "\E" or "\e" is encountered, or until the end of the replacement pattern.

6.9 Speeding Things Up

At times, getting in and out of Vi can take much more time than you'd like. To keep the speed of invocation down, and to keep the number of invocations to a minimum, the following hints are offered:

- Keep files around 20,000 characters in size, or smaller. Files larger than 100,000 characters are intractable.
- Invoke Vi with a list of filename arguments. You can then use the next (:n) command to cycle through the named files. This is faster and more efficient than using a series of edit (:e) commands or invoking the editor for each file,
- When transferring text between files, yank or delete text into a buffer; switch files, and then "put" the

text into the new file.

- If you want to execute XENIX commands while in Vi, use the shell escape ":!", or the ":sh" command to create a new shell. With either method, you can quickly return to Vi without reinvocation.

6.9.1 When To Use Ex

Since Ex and Vi are one and the same program, you can switch back and forth between the two quite easily. To enter Ex from Vi, use the Q command. To enter Vi from Ex, use the vi command.

One major difference between the two is that Ex is line-oriented and uses XENIX standard input and output. Vi is screen-oriented and uses its own special input routines for handling keyboard input.

Because of this difference, Ex is in some cases better than Vi. For example, you can write scripts only for Ex: scripts won't work in Vi. Similarly, if you know how to use the line editor, Ed, the transition to Ex will be easy, since the Ex command set is largely a superset of Ed. Also use Ex whenever you need to perform a substitution command that requires more than one line of input. Newlines can be escaped with a backslash (\) in Ex, but not in Vi. Therefore, the following command is legal in Ex, but not in Vi:

```
s/one line/one line\
two lines\
three lines\
four lines/
```

6.10 Limitations

In using Vi, you should note the following limits:

- 250,000 lines in a file
- 1024 characters per line
- 256 characters per global command list
- 128 characters per filename
- 128 characters in the previous inserted and deleted text

- 100 characters in a shell escape command
- 63 characters in a string valued option
- 30 characters in a tag name

6.11 Troubleshooting

The following is a list of common problems that you may encounter when using Vi, along with the probable solution for each.

Don't know which operation mode you're in
You often want to abort a command or exit insert mode. To do this, type <ESC> until the bell rings to assure your return to Vi command mode.

Can't get out of sub-shell
Type <CONTROL-D> to exit any shell.

Accidentally entered Ex command mode
If you pressed colon (:) accidentally, you can return to where you were in your file by typing an <ESC>.

Inadvertent deletion or insertion
Type "u" to undo the last delete or insert command.

Scrambled Screen
Type <CONTROL-L> to redraw the screen.

Line too long
Go to the middle of the line and type

r<RETURN>

to break the line into two separate lines.

Can't see control characters
Set the list option by typing:

:set list

This will substitute a circumflex (^) and the name of the control letter in place of each control character. This is most useful for detecting tab characters in files. For example, "^I" would appear instead of the expansion of the tab into one or more spaces.

Not in screen mode

Type ":q!" to abort and then restart. Check to see if the TERM variable in the file .profile in your home directory contains the proper terminal type definition.

Mistakenly in open mode

See above.

Full file not read in

Type ":e! %" to re-edit the current file.

Keyboard locked up

Vi has crashed and you are now in the shell with your terminal characteristics set incorrectly. To reset the keyboard, type:

```
stty -nl echo -cbreak <LF>
```

Typing a linefeed, i.e., a <LF> or <CONTROL-J> is important here, since it is quite possible that the <RETURN> key will not work as a newline character.

System crashed

Normally, Vi will inform you that your editing buffer has been saved before a crash. The buffer can be recovered by typing

```
vi -r filename
```

If Vi was unable to save the buffer before the crash, then the editing buffer is irretrievably lost.

6.12 Character Functions

This section describes the use of each key on your keyboard in Vi command mode and in insert mode. It does not describe the use of these keys in Ex command mode. If a character's only meaning in insert mode is to insert the character in the text as-is, then only its meaning as a command is discussed. Characters are presented alphabetically with a lowercase, an uppercase, and a control character description for each letter of the alphabet.

- a Appends text after the current cursor position.
- A Appends text at the end of the current line. This is a synonym for "\$a".
- <CONTROL-A> Not used as a command.
- <BKSP> Moves the cursor one character to the left. This has the same function as the left arrow key, the "h" key, and <CONTROL-H>. Note that arrow keys on certain kinds of terminals cannot be used, and that some terminals have no arrow keys at all. A preceding count repeats the effect.

In insert mode, <BKSP> eliminates the last input character, backing over it and erasing it from the insertion, but not from the screen. The character remains so that you can see what you have typed if you wish to type something slightly different.
- b Backs up the cursor to the beginning of the current word. A word is a sequence of alphanumeric characters, or a sequence of special characters. A preceding count repeats the effect.
- B Backs up a word and places the cursor at the beginning of the current word, where words are composed of non-blank sequences. A preceding count repeats the effect.
- <CONTROL-B> Move the cursor back one screen window. A preceding count repeats the effect. Two lines from the preceding window are kept for continuity, if possible.

- c Changes the following text object, replacing it with input text up to an <ESC>. If more than part of a single line is affected, the text which is deleted is saved in the numeric named buffers. If only part of the current line is affected, then the last character in the text to be changed is marked with a dollar sign (\$). A count specifies the number of the given objects to be affected, thus both "3c)" and "c3)" change three sentences.
- C Changes the rest of the text on the current line, replacing it with input text up to an <ESC>. This is a synonym for "c\$".
- <CONTROL-C> Not used as a command.
- d Deletes the following text object. If more than part of a line is affected, the text is saved in the numeric buffers. A count specifies the number of the given objects to be affected, thus "3dw" is the same as "d3w".
- D Deletes text from the cursor to the end of the line. The character beneath the cursor is deleted as part of this command. This is a synonym for "d\$".
- <CONTROL-D> Scrolls down a half-window of text. A preceding count gives the number of (logical) lines to scroll, and is remembered for future <CONTROL-D> and <CONTROL-U> commands. When in insert mode, <CONTROL-D> backtabs over autoindent white space at the beginning of a line: this white space cannot be backspaced over.
- e Advances to the end of the next word, where a word is defined as for the b and w commands. A preceding count repeats the effect.
- E Moves forward to the end of a word, where a word is defined as for the B and W commands. A preceding count repeats the effect.
- <CONTROL-E> Not used as a command.
- <ESC> Exits insert mode and cancels a partially formed command, such as a z when no following character has yet been given. Also

- terminates command line input on the status line for commands such as colon (:), slash (/), and question mark (?). If an <ESC> is given when in command mode, the editor rings the bell or flashes the screen.
- f** Finds the first instance of a character following the cursor. The next typed character is the character given as an argument. The search for the character begins following the cursor on the current line. The search goes no farther than the end of line. A preceding count repeats the effect.
- F** Finds a single following character, preceding the cursor. The next typed character is the character given as an argument. The search for the character begins preceding the cursor on the current line. The search goes no farther than the beginning of the line. A preceding count repeats the effect.
- <CONTROL-F>** Move the cursor forward one screen window. A preceding count repeats the effect. Two lines from the preceding window are kept for continuity, if possible.
- g** Not used as a command.
- G** Moves the cursor to the line specified by a preceding line number argument (e.g., "12G" moves the cursor to the beginning of line 12. If no line number is given, then the cursor moves to the end of the file. The screen is redrawn with the new current line in the center of the screen.
- <CONTROL-G>** Equivalent to ":f<RETURN>", Prints the name of the file currently being edited, whether the file has been modified during the current editing session, the line number of the current line, the number of lines in the file, and the percentage of the file viewed in lines, relative to the current line.
- h** See <BKSP>.
- H** Homes the cursor to the beginning of the top line on the screen. If a count is given, then the cursor is moved to the beginning of

the line that is the "count'th" line from the top of the screen. In any case, the cursor is moved to the first non-whitespace character on the line.

- <CONTROL-H> See h.
- i Inserts text before the cursor, otherwise like a.
- I Inserts text at the beginning of the current line. This is a synonym for "^i".
- <CONTROL-I> See <TAB>.
- j Moves the cursor one line down in the same column. If the column position does not exist, vi comes as close as possible to the same column. Synonyms include <CONTROL-J>, <LF>, and <CONTROL-N>.
- J Joins together the current line with the next line, supplying appropriate white space: one space between words, two spaces after a period, and no spaces at all if the first character of the joined line is a right parenthesis. A count causes the given number of lines to be joined.
- <CONTROL-J> See j.
- k Moves the cursor up one line in the same column. <CONTROL-P> is a synonym.
- K Not used as a command.
- <CONTROL-K> Not used as a command.
- l Moves the cursor one character to the right. The <SPACE> and right arrow keys are synonyms.
- L Moves the cursor to the first non-white character of the last line on the screen. With a preceding count, it moves to the first non-white of the "count'th" line from the bottom.
- <CONTROL-L> Causes the screen to be cleared and redrawn. This is useful after a transmission error, if characters typed by a program scramble the

- screen, or after output is stopped by an interrupt. The <FF> or form feed key is a synonym for this key.
- m** Marks the current position of the cursor with an alphabetic "mark". The mark is specified by the character given as an argument to the command. This character is the next key typed, and must be in the range "a"- "z". Move to this marked position using ' or `.
- M** Moves the cursor to the middle line on the screen, at the first non-white position on the line.
- <CONTROL-M>** Same as <RETURN>.
- n** Repeats the last search command (/ or ?).
- N** Scans for the next match of the last pattern given to / or ?, but in the reverse direction; this is the reverse of n.
- <CONTROL-N>** See j.
- o** Opens a new line below the current line so that input text can be inserted up to an <ESC>.
- O** Opens a new line above the current line and inputs text there up to an <ESC>. A count can be used on dumb terminals to specify a number of lines to be opened; this is generally obsolete, as the slowopen option works better.
- <CONTROL-O>** Not used as a command.
- p** Puts last deleted text at either the line below the cursor or inserts that text before the cursor on the same line. See P below for details.
- P** Puts the last deleted text back above or before the cursor. The text goes back as whole lines above the cursor if it was deleted as whole lines. Otherwise, the text is inserted between the characters before and at the cursor. P may be preceded by a named buffer specification of the form

"x

to retrieve the contents of the named buffer. Buffers 1-9 contain deleted material; buffers "a"--"z" are available for general use.

<CONTROL-P> See **k**.

q Not used as a command.

Q Switches from Vi to Ex command mode. In Ex command mode, whole lines form commands, ending with a <RETURN>. You can give most Ex commands in Ex command mode without typing a preceding colon; the editor will supply the colon as a prompt. To return to Vi command mode, use the "vi" command.

<CONTROL-Q> Not a command character.

<RETURN> Advances to the next line, at the first non-white position in the line. If given a count, it advances that many lines.

When in insert mode, a <RETURN> causes the insertion to continue on to a new line. The <RETURN> key is the same as the <CONTROL-M> key.

r Replaces the single character at the cursor with the next typed character. The new character may be a <RETURN>; this is the easiest way to split lines. A count replaces each of the following count characters with the single character given.

R Replaces characters on the screen with characters you type (this is sometimes called overstrike mode). Replacement terminates with an <ESC>.

<CONTROL-R> Redraws the current screen, eliminating logical lines not corresponding to physical lines (lines containing only a single at-sign (@)).

s Substitutes the single character under the cursor to the following inserted text. Text insertion is terminated with an <ESC>. If given a count, s then substitutes the given number of characters rather than just one.

The last character to be changed is marked with a dollar sign (\$), as with the c command.

- s Changes whole lines, a synonym for the cc command. A count substitutes the given number of lines instead of just one. Substituted lines are saved in the numeric buffers and erased on the screen before the substitution begins.
- <CONTROL-S> Not used as a command.
- <SPACE> See l above.
- t Advances the cursor forward, up to, but not on a given character. The given character is the next character typed. Most useful with operators such as d and c to delete the characters up to a character. Use dot (.) to delete more if this doesn't delete enough the first time.
- T Takes a single following character, locates the character before the cursor in the current line, and places the cursor just after that character. A count repeats the effect.
- <CONTROL-T> Not a command character. During an insertion, with autoindent set and at the beginning of the line, inserts shiftwidth white space.
- <TAB> Not a command character. When in insert mode, a <TAB> prints as a number of spaces appropriate to vi's tab settings. When the cursor is at a tab character it rests at the last of the spaces which represent the tab. The spacing of tab stops is controlled by the tabstop option.
- u Undoes the last change made to the current buffer. If repeated, will alternate between these two states, thus this command is its own inverse. When used after an insertion that inserted text on more than one line, then the inserted lines are saved in the numeric named buffers.

- U** Restores the current line to its state before you started changing it. This works only for the duration of the particular line edit; not for the duration of the editing session.
- <CONTROL-U>** Scrolls the screen up. Counts work as they do for <CONTROL-D>. The previous scroll amount is common to both.
- v** Not used as a command.
- V** Not used as a command.
- <CONTROL-V>** Not a command character. In input mode, <CONTROL-V> quotes the next character so that it is possible to insert nonprinting and special characters into the file.
- w** Advances the cursor to the beginning of the next word, where a word is as defined in the b command, above.
- W** Moves the cursor forward to the beginning of a word in the current line, where words are defined as sequences of non-blank characters. A count repeats the effect.
- <CONTROL-W>** Not a command character. During an insertion, <CONTROL-W> backs up as b would in command mode.
- x** Deletes the single character under the cursor. With a count, x deletes the given number of characters forward from the cursor position, but only on the current line.
- X** Deletes the character before the cursor. A count repeats the effect, but only characters on the current line are deleted.
- <CONTROL-X>** Not used as a command.
- y** "Yanks" the following object into the unnamed temporary buffer. If preceded by a named buffer specification, then the text is placed in that buffer also. Text can be recovered with a later p or P command.
- Y** "Yanks" a copy of the current line into the unnamed buffer, to be put back by a later p or P command. This is a useful synonym for

"yy". A count yanks that many lines. May be preceded by a buffer name to put lines in that buffer.

<CONTROL-Y> Not used as a command.

z Adjusts and redraws the screen with the current line placed as specified by an alignment character. The alignment character may be either a <RETURN>, a period (.), or a minus (-). A <RETURN> specifies the top of the screen, a dot the center of the screen, and a minus the bottom of the screen. A count may be given after the z, but before the alignment character, to specify the new screen size for the redraw. A count before the z gives the number of the line to place in the center of the screen instead of the default current line.

zz Exits the editor. (Same as ":x<RETURN>".) If any changes have been made, the buffer is written out to the current file. Then the editor quits.

<CONTROL-Z> Not used as a command.

0 Moves the cursor to the first character on the current line. Also used in forming numbers, after an initial 1-9. Note that no command can take an initial zero as part of a count argument.

1-9 Used to form numeric arguments to commands.

(Retreats to the beginning of a sentence, or a LISP s-expression, if the lisp option is set. A sentence ends at a period (.), exclamation (!), or question mark (?) which is followed by either the end of a line or by two spaces. Any number of closing parentheses, brackets, quotation marks, or single quotes may appear after the period, exclamation, or question mark, and before the spaces or end of line. Sentences also begin at paragraph and section boundaries (see { and [[below). If given a count, the command is repeated. A count advances the given number of sentences.

) Advances to the beginning of a sentence. A count repeats the effect. See (above for

the definition of a sentence.

- [[Backs up to the previous section boundary. A section begins at each macro in the sections option, normally a ".NH" or ".SH" and also at lines that start with a formfeed, <CONTROL-L>. Lines beginning with { also stop [[. This makes it useful for looking backwards, a function at a time, in C programs. If the lisp option is set, the [[stops at each left parenthesis [() at the beginning of a line.
-]] Move forward to a section boundary. See [[for the definition of a section boundary.
- < An operator which shifts lines left one shiftwidth, normally eight spaces. Affects lines when repeated, as in "<<". Counts are passed through to the basic object, thus "3<<" shifts three lines.
- > Shifts lines right one shiftwidth, normally eight spaces. Counts repeat the basic object. This operator can be made to affect a given number of lines when the character is doubled and preceded by the number of lines to be affected, as in "23>>".
- { Retreats to the beginning of the preceding paragraph. A paragraph begins at each macro in the paragraphs option, normally at a ".IP", ".LP", ".PP", ".QP" and ".bp". A paragraph also begins after a completely empty line, and at each section boundary (see [[above).
- } Advances to the beginning of the next paragraph. See { for the definition of paragraph.
- ' When followed by another back quote, this returns to the previous context. The previous context is set whenever the current line is moved in a non-relative way. When followed by a letter in the range "a"- "z", returns to the position marked with the given letter by an m command. When used with an operator such as d, the operation takes place from the exact marked place to the current position within the line; If a single quote ('') is used, the operation takes place over

whole lines instead.

' When followed by a single quote (') returns to the previous context at the beginning of a line. The previous context is set whenever the current line is moved in a non-relative way. When followed by a letter in the range "a"- "z", it returns to the line marked with the given letter by an **m** command. When used with an operator such as **d**, the operation takes place over complete lines; if you use a back quote (`), the operation takes place from the exact marked place to the current cursor position within the line.

- + Same as <RETURN> when used as a command.
- Retreats to the previous line at the first non-white character. This is the inverse of plus (+) and <RETURN>. If the line moved to is not on the screen, the screen is scrolled, or, if this is not possible, cleared and redrawn. If a large amount of scrolling would be required the screen is cleared and redrawn, with the current line at the center.
- ^ Moves to the first non-white position on the current line.
- _ Not used as a command.
- ! Processes a text object in the editing buffer with a XENIX command given as the second argument. The first argument is a cursor movement command that delimits the text object. The XENIX command is processed by the shell, so normal XENIX quoting conventions apply. The command line itself is terminated with a <RETURN>. Doubling ! and preceding it by a count causes count lines to be filtered; otherwise the count is passed on to the object after the !. Thus "2!}sort<RETURN>" sorts the next two paragraphs by running them through the program sort. To read a file or the output of a command into the buffer use ":r". To simply execute a command use ":!".
- " Precedes a named buffer specification. There are named buffers 1-9 used for saving deleted text and named buffers "a"- "z" into which you

can place text.

- \$ Moves the cursor to the end of the current line. If you use ":set list<RETURN>", then the end of each line will be shown by printing a dollar sign (\$) after the end of the displayed text in the line. Given a count, advances to the count'th following end of line; thus "2\$" advances to the end of the following line.
- % Moves the cursor to the parenthesis or brace that balances the parenthesis or brace at the current cursor position.
- & A synonym for ":&<RETURN>", by analogy with the Ex & command.
- ,
 Reverse the last f, F, t, or T command, searching in the opposite direction on the current line. This is especially useful after typing too many semicolon (;) characters. A count repeats the search.- ;
 Repeats the last single character find command (either f, F, t, or T). A count repeats the scan count number of times.- .
 Repeats the last command that changed the buffer. This is especially useful when deleting words or lines; you can delete some words or lines and then press dot (.) to delete more words or lines. If given a count, "dot" passes it on to the command being repeated. Thus after a "2dw", "3." deletes three words.- /
 Reads a string from the last line on the screen, and scans forward for the next occurrence of this string. The normal input editing sequences may be used during input on the status line.

The search begins when you press <RETURN>. The cursor then moves to the beginning of the last line to indicate that the search is in progress. The search can be terminated by typing an <INTERRUPT>. Searches normally wrap around the end of the file to the beginning to find a string anywhere in the buffer.

When used with an operator the enclosed region is normally affected. By giving an offset from the line matched by the pattern you can force whole lines to be affected. To do this, give a pattern with a closing slash (/>) and then an offset +n or -n.

- ? Searches backwards for the string given as an argument. This is the opposite of slash (/). See the description of slash above for details on searching.
- :
- The Ex command escape character. Ex command input is entered on the status line. If you press colon (:) accidentally, you can return to where you were by typing an <INTERRUPT>.
- | Places the cursor on the character in the column specified by a preceding count.
- = Not used as a command.
- @ Not used as a command.
- ~ Not used as a command.
- *
- Not used as a command.
- # Not used as a command.
- \ Not used as a command.
- <CONTROL-_> Not used as a command. Reserved as the command character for some terminals.
- <CONTROL-?> Interrupts the editor, returning it to command mode.
- <CONTROL-@> Not a command character. If typed as the first character of an insertion it is replaced with the last text inserted, and the insertion terminates. Only 128 characters are saved from the last insertion. If more characters were inserted than 128, then this command inserts no characters. A <CONTROL-@> cannot be part of a file, even if quoted.
- <CONTROL-[> Same as <ESC>.
- <CONTROL-]> Searches for the word which is after the cursor as a tag. Equivalent to typing ":ta",

this word, and then a <RETURN>. Mnemonically, this command is "go right to".

<CONTROL-^> Equivalent to ":e #<RETURN>", Returns to the previous position in the last edited file. If you do not want to write the file you should use ":e! #<RETURN>" instead.

RADIO SHACK, A DIVISION OF TANDY CORPORATION

**U.S.A.: FORT WORTH, TEXAS 76102
CANADA: BARRIE, ONTARIO L4M 4W5**

TANDY CORPORATION

AUSTRALIA

91 KURRAJONG ROAD
MOUNT DRAUITT, N.S.W. 2770

BELGIUM

PARC INDUSTRIEL DE NANINNE
5140 NANINNE

U. K.

BILSTON ROAD WEDNESBURY
WEST MIDLANDS WS10 7JN