SERIES 60 (LEVEL 68)

SOFTWARE

MULTICS COBOL USER'S GUIDE

SUBJECT

Information Concerning Efficient Techniques for Using Multics COBOL

SPECIAL INSTRUCTIONS

This manual supersedes AS43, Rev. 0, which has been extensively revised. Marginal change indicators have been omitted.

SOFTWARE SUPPORTED

Multics Software Release 6.0

Includes update pages issued as Addendum A in April 1978.

ORDER NUMBER

AS43, Rev. 1

December 1976

Honeywell

PREFACE

This manual is a supplement to the Multics COBOL Reference Manual, Order No. AS44.

Section I provides an introduction to the contents of this manual and lists terms the Multics COBOL user must understand before using this manual.

Section II explains how a COBOL source program is created on Multics.

Section III discusses in detail the COBOL compiler, its available options, and its output.

Section IV shows how COBOL source I/O statements interface and interact with the Multics I/O system.

Section V describes the run-time environment of a COBOL program.

Section VI discusses the use of the Multics symbolic debugging facilities with a COBOL program, explains the general procedure used for reporting run-time errors, and shows various ways of dealing with these errors.

Section VII contains suggestions for achieving most efficient use of Multics COBOL.

Section VIII describes the design of the runtime package that supports the ANSI COBOL-74 Communication Module.

Symbolic notations used to describe general source language formats within this manual are as follows. Brackets [ ] enclose portions of a general format that may be included or omitted at the user's discretion. Braces { } enclosing a portion of a general format indicate that an option contained within the braces must be selected. Choice indicators, {| |}, enclosing a portion of a general format mean that a selection of one or more of the options contained within the choice indicators must be made, but the same sequence of words cannot be chosen more than once in that entry or statement. The ellipsis ... represents an optional repetition of the preceding term. Components of a format description that refer to a set of possible values rather than to a value itself are given descriptive names enclosed in angle brackets < >.

Optional command arguments are enclosed in braces (e.g., {path}, {-control_args}). All other arguments are required. Control arguments are identified in the usage line with a leading hyphen (e.g., -control_arg) simply as a reminder that all control arguments must be preceded by a hyphen in the actual invocation of the command. To indicate that a command accepts more than one of a specific argument, an "s" is added to the argument name (e.g., paths, -control_args). See the MPM Commands, Section 3 introduction, for a complete description of the command description format.

© 1978, Honeywell Information Systems Inc. File No.: 1L23

The set of values constituting permissible values of path, and -control_args are defined (informally) at some point in this manual.

When a sequence of lines representing interactive communication between the user and Multics is given, the following conventions illustrate the nature of the interaction: all lines are indented, lines the user types are preceded by the exclamation mark (!), and those the system prints stand alone. Parenthesized information on the right side of a line constitutes additional commentary concerning the particular line. For example:

```
!    edm progname.cobol
     Edit.                  (segment already exists)
!    q                      (to return to command level)
     r 2314 1.007 1.567 18
```

COBOL key words may be written in uppercase or lowercase letters in the source program, but they always appear in uppercase in the text of this manual. Multics key words (i.e., command names and control arguments) must be given in lowercase.

In this manual many references are made to commands; some are described in detail, while others are mentioned only in passing. Any key word termed a command refers to a standard Multics command generally available to the Multics user at command level. Most commands have both a long and abbreviated form. Only the long, more descriptive name is given.

Some Multics subroutines are also referenced. These are not normally called from command level, as they often require noncharacter-string arguments. By convention, Multics subroutines have a name ending with the underscore character; e.g., print_cobol_error_.

A full description of standard commands is available in the Multics Programmers' Manual, Commands and Active Functions, Order No. AG92. Subroutines are described in the Multics Programmers' Manual, Subroutines, Order No. AG93. These references are given here to avoid repetitive references with the mention of each command and subroutine in the text. Further references to the Multics Programmers' Manual are abbreviated to MPM.

Primary reference for user and subsystem programming on the Multics system is contained in five manuals that are collectively referred to as the MPM. Throughout this manual, references are frequently made to the MPM. For convenience, these references will be as follows:

Document	Referred To In Text As
Reference Guide (Order No. AG91)	MPM Reference Guide
Commands and Active Functions (Order No. AG92)	MPM Commands
Subroutines (Order No. AG93)	MPM Subroutines
Subsystem Writers' Guide (Order No. AK92)	MPM Subsystem Writers' Guide
Peripheral Input/Output (Order No. AX49)	MPM I/O

ACKNOWLEDGMENT

This acknowledgment has been reproduced from the CODASYL COBOL Journal of Development 1976, as requested in that publication, prepared and published by the CODASYL Programming Language Committee.

"Any organization interested in reproducing the COBOL report and specifications in whole or in part, using ideas from this report as the basis for an instruction manual or for any other purpose, is free to do so. However, all such organizations are requested to reproduce the following acknowledgment paragraphs in their entirety as part of the preface to any such publication. Any organization using a short passage from this document, such as in a book review, is requested to mention "COBOL" in acknowledgment of the source, but need not quote the acknowledgment.

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 (R) I and II, Data Automation Systems copyrighted 1958, 1959, by Sperry Rand Corporation; IBM Commercial Translator Form No. F 28-8013, 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."

CONTENTS

		Page
Section I	Introduction	1-1
Section II	COBOL Source Program	2-1
	Source Segment	2-1
	Creating a COBOL Source Segment	2-2
	COBOL Reference Format	2-3
	Fixed Format on Multics	2-3
	Conversion Considerations	2-3
	Nonexistent Areas	2-4
	Alternate Terminal-Oriented Format	2-5
	Free-Form Format Definition	2-5
	Permanent Translation	2-6
	COBOL Source Code	2-6
	Use of Non-COBOL Characters	2-7
	Capitalization Considerations	2-8
	Special Characters in Nonnumeric Literals	2-9
	Escape Convention	2-9
	Control Division	2-11
	Sign Control	2-11
	Usage Control	2-11
	Precision Control	2-12
	Descriptor Control	2-12
	COBOL Library Facility	2-12.1
	Definition of a Library	2-12.1
	Dynamic Nature of the Library	2-12.2
	Format Restriction for Library Text	2-13
	Text Comparison and Replacement	2-13
	Auxiliary Commands	2-17
	expand_cobol_source, ecs	2-17
Section III	Compiling the COBOL Program	3-1
	Invoking the Compiler	3-1
	Source Errors	3-2
	Diagnostic Format	3-2
	Controlling Terminal Output	3-3
	Severity Control	3-3
	Suppression of Warnings and Fatal Errors	3-4
	Reporting Observations	3-4
	Unrecoverable Errors	3-4
	Repetition Control	3-5
	Leveling	3-5
	List File	3-5.1
	List Header	3-6
	Source Listing	3-6
	DATE-COMPILED Paragraph	3-7
	Cross-Reference Listing	3-8
	Usage	3-8
	Allocation	3-9
	Object Map	3-10
	Object Listing	3-10.1
	Additional Compiler Arguments	3-12
	Object Code Suppression	3-12
	Run-Time Error Checking	3-12
	Parameter Validation	3-12.1
	Subscript Range Checking	3-12.1

Page

String Range Checking 3-12.2
Index Integrity Verification 3-12.2
Source Transformations 3-13
Run-Time Performance Measurement 3-13
Source Level Debugging Requirements . . . 3-13.1
Allocation for Temporary Compile-Time Files 3-13.1
Compiler Development and Testing Facilities 3-14
Argument Summary 3-14
Object Segment 3-16
Segment Creation 3-16
Object Segment Format 3-16
Text Section 3-16
Definition Section 3-17
Linkage Section 3-17
Symbol Section 3-17
Compiler Characteristics 3-18
Reentrancy 3-18
Command Line Considerations 3-18
Ordering of Arguments 3-19
Multiple Compilations 3-19
Online Documentation 3-20

Section IV Input/Output Processing 4-1
Terminal I/O 4-1
Accepting Data 4-1
Displaying Data 4-2
Performing COBOL I/O on Multics 4-3
Opening a File 4-3
Transmitting Data 4-4
Closing a File 4-4
File Characteristics and Device Independence 4-5
File Sharing 4-5
Scope of Files 4-5
Attaching from Command Level 4-6
Implications for the OPEN and CLOSE Verbs 4-7
Defining a File 4-8
File Selection - SELECT Clause 4-8
File Structure 4-8
Organization 4-8
Access Mode 4-10
Record Format 4-11
Keys 4-11
Record Key 4-11
Relative Key 4-12
I/O Switch Assignment 4-12
EXTERNAL Attribute 4-12
Internal-file-name 4-12
Device Specification 4-13
File Status 4-16
Supplementary Options - APPLY Clause . . 4-18
Temporary Files 4-18
Attachment Control 4-20
Explicit Attach Specification 4-20
Tape Attachment Specialization 4-22
Record Description - FD Entry 4-22.1
Value of Catalog-Name is Clause . . . 4-23
Virtual Memory Files 4-23
Tape Files 4-23.1
Variable-Length Records 4-23.1
Declarative Procedures 4-24
Print Files 4-25
Page and Line Control 4-26

		Page
	File Opening Modes	4-27
	Implementation Specifics	4-27
	File State Block	4-29
	File Activity Recording	4-29
	File Organization and Structure	4-29
	Sequential Files	4-29
	Relative Files	4-30
	Sequential Mode	4-30
	Random Mode	4-31
	Dynamic Mode	4-31
	Indexed Files	4-32
	Sequential Mode	4-33
	Random Mode	4-33
	Dynamic Mode	4-34
	Example	4-35
Section V	Executing a COBOL Program	5-1
	Referencing an Object Segment	5-1
	Resolving External References	5-2
	Multics Environment	5-2
	Search Rules	5-3
	Dynamic Linking	5-4
	Binding	5-5
	Program Execution from Command Level	5-5
	COBOL Run-Unit	5-7
	Run-Unit Definition	5-8
	Run-Unit Related Statements	5-8
	STOP RUN Statement	5-8
	EXIT PROGRAM Statement	5-9
	CANCEL Statement	5-9
	Auxiliary Commands	5-10
	run_cobol Command	5-10
	stop_cobol_run Command	5-12
	cancel_cobol_program Command	5-12
	display_cobol_run_unit Command	5-13
	Aspects of the Run-Time Environment	5-16
	STOP <literal> Statement	5-16
	External Switches	5-16
	COBOL Segmentation	5-17
	Improper Program Termination	5-17
	COBOL Data	5-17
	Data Types	5-18
	Unsigned Display Data	5-19
	Separate Sign Display Data	5-20
	Nonseparate Sign Display Data	5-21
	Packed Decimal Data	5-22
	Binary Data	5-22
	Data Allocation	5-23
	Interprogram Communication	5-24
	Aggregate Data	5-25
	Implementation Specifics	5-26.1
	Run-Unit Control	5-26.1
	Data Addressability	5-27
Section VI	Error Processing and Debugging	6-1
	Symbolic Debugging	6-1
	Monitoring Program Execution	6-2
	Displaying and Modifying Data	6-3
	Character-String Data (display)	6-3
	Packed Decimal Data (COMP, COMP-5, COMP-8)	6-4
	Binary Data (COMP-6, COMP-7)	6-6
	Run-Time Errors	6-6
	Anticipated Errors	6-7

		Page
	I/O Errors	6-9
	print_cobol_error_ Subroutine	6-10
	SIZE ERROR Option	6-11
	Unanticipated Errors	6-11
Section VII	Efficiency Considerations	7-1
	Program Size	7-1
	Data Definition	7-2
	I/O Considerations	7-2
	Use of Numeric Data Types	7-3
	Use of the Inspect Statement	7-4
	Miscellaneous Considerations	7-5
	Measuring a Program's Performance	7-6
Section VIII	COBOL Message Control System	8-1
	References	8-1
	Terminology	8-1
	Design Concepts	8-2
	COBOL MCS Queue Organization	8-5
	Overview of CMCS Data Bases	8-5
	Administrative Functions	8-6
	CMCS Administrator	8-6
	Message Processing Operation	8-8
	Daemon Message Processor	8-8
	System Administrator Actions	8-8
	Project Administrator Actions	8-8
	Operator Actions	8-8
	User Commands	8-9
	cobol_mcs, cmcs	8-10
	cobol_mcs_admin, cmcsa	8-17
	cv_cmcs_station_ctl	8-22
	cv_cmcs_terminal_ctl	8-23
	cv_cmcs_tree_ctl	8-24
Section IX	File Ordering -- Sort and Merge	9-1
	Concepts	9-1
	Sorting	9-1
	Sort Statement	9-1
	Merging	9-2
	Merge Statement	9-2
	Ordering	9-3
	Program Organization	9-3
	Sort Statement	9-4
	Sort File	9-5
	Sort Key Declarations	9-5
	Variable-Length Records	9-5
	Dominant Record Length	9-6
	Sort Key Evaluation	9-6
	Sort Input Processing	9-6
	Using Option	9-6
	Input Procedure Option	9-6
	RELEASE Statement	9-7
	Giving Option	9-7
	Output Procedure Option	9-8
	RETURN Statement	9-8
	Sort Operational Considerations	9-9
	Flow of Control	9-9
	Sort Examples	9-11
	Merge Statement	9-12
	The Merge File	9-12
	Merge Key Declarations	9-12
	Variable-Length Records	9-12
	Merge Key Evaluation	9-13
	Merge Input Processing	9-13

Page

Merge Output Processing 9-13
Giving Option 9-13
Output Procedure Option 9-13
RETURN Statement 9-14
Merge Operational Considerations 9-14
Flow of Control 9-14
Merge Examples 9-16
Work Requirements 9-17
Sort Work Files 9-17
Process Directory Work Files 9-17
Running COBOL Programs with the SORT Statement 9-18

Section X — Debug Facility 10-1
Description of the Debug Facility 10-1
Example of the Debug Facility 10-1

Section XI — Report Writer 11-1
Description of the Report Writer 11-1
Report Format 11-1
Report Control in the Procedure Division 11-2
Skeletal Format for the Report Section . 11-4
RD Entries 11-5
Report Group Entries 11-5
Elements of a Report 11-6
Report Groups 11-6
Control Data Items 11-7
File Characteristics 11-8
Line Counter 11-8
Page Counter 11-9
SUM Counter Manipulation 11-9
Subtotalling 11-10
Rolling Forward 11-10
Crossfooting 11-11
Producing A Report 11-12
Report Command 11-12

Appendix A — Order of COBOL Source Program A-1

Index i-1

ILLUSTRATIONS

Figure 9-1. Sort Program Organization 9-4
Figure 9-2. Merge Program Organization 9-4
Figure 9-3. Sort Input Procedure Organization 9-9
Figure 9-4. Sort Output Procedure Organization 9-10
Figure 9-5. Merge Output Procedure Organization 9-15

TABLES

Table 2-1. Fixed Format Areas 2-3
Table 2-2. Escape Convention 2-10
Table 3-1. Diagnostic Severity Levels 3-2
Table 3-2. Summary of List Arguments 3-6
Table 3-3. Summary of Compiler Control Arguments 3-15
Table 4-1. Multics I/O Modules 4-4
Table 4-2. File Organization 4-8

CONTENTS (cont)

		Page
Table 4-3.	File Opening Modes	4-28
Table 5-1.	COBOL Data Types	5-18
Table 5-2.	Display Data Digit Encoding	5-19
Table 5-3.	Display Separate Sign Encoding	5-20
Table 5-4.	Display Nonseparate Sign Encoding	5-21

SECTION I

INTRODUCTION

This manual provides information that enables the COBOL programmer to create new programs on Multics and to convert existing programs to run efficiently. The user is assumed to possess a basic knowledge of the COBOL language and some familiarity with the essential ideas, terminology, and usage of the Multics system. Specifically, he should understand the following terms:

Directory structure:

- pathname
- directory
- segment
- link
- access mode
- access control list (ACL)

Interactive control:

- command
- command processor
- command level
- ready message
- quit and restart

Processing environment:

- process
- user
- stack
- process directory
- working directory

File system:

- file
- I/O switch
- I/O module
- attach

If necessary, review the appropriate sections of the MPM Reference Guide.

This manual explains in detail those aspects of the COBOL language that interface with or are dependent on the Multics system. Particular attention is given to the interpretation of the various paragraphs of the Environment Division, since this is the system-dependent component of the COBOL program.

This manual deals specifically with the meaning of device-oriented COBOL files in the Multics virtual memory system. It discusses the run-time environment of the COBOL run-unit and the effects of dynamic linking on the execution of a modular run-unit and describes certain COBOL-oriented commands that give the user supplemental information and control over the run-unit. Finally, error processing and recovery are discussed.

SECTION II

COBOL SOURCE PROGRAM

This section explains the manner in which a COBOL source program is created on Multics. Uppercase, lowercase, and character set usage are discussed. Also, the COBOL card-oriented reference format is described, and an alternate, terminal-oriented format is presented. Such matters as the effect of the newline character and trailing spaces on nonnumeric literals and a source-level escape convention for nonprinting characters are explained. Finally, the COBOL library facility is defined in terms of the Multics storage system.

SOURCE SEGMENT

The source program is defined as a syntactically correct set of COBOL statements and serves as input to the COBOL compiler. It is contained in a segment that must have a name of the form:

<program name>.cobol

where <program name> corresponds to the name specified in the PROGRAM-ID paragraph of the Identification Division. The source segment may be created and modified by using any of the text editors available on the Multics system. Complete descriptions of the edm and qedx text editors can be found in the MPM Commands manual.

CREATING A COBOL SOURCE SEGMENT

To create a COBOL program named time-and-date that displays the time and date on the user's terminal, the user could invoke the qedx text editor to create a source segment named time-and-date.cobol as follows:

```
qedx                          (invoke the editor)
a                             (enter append mode)
       IDENTIFICATION DIVISION.
        PROGRAM-ID. time-and-date.
       ENVIRONMENT DIVISION.
        CONFIGURATION SECTION.
        source-computer. Multics.
        object-computer. Multics.
       DATA DIVISION.
         WORKING-STORAGE SECTION.
        01 time-out pic 99B99b99pp.
        01 DATE-OUT PIC x(8).
        01 date-in.
          02 yy pic 99.
          02 mm pic 99.
          02 dd pic 99.
       PROCEDURE DIVISION.
       begin. accept date-in from date.
           string mm "/" dd "/" yy delimited by size into date-out.
           accept time-out from time.
           inspect time-out replacing all spaces by ":".
           display "Time: " time-out "  Date: " date-out.
           exit program.
\f
w time-and-date.cobol         (create the segment)
q                             (return control)
r 1445 2.110 11.020 208       (system ready for next command)
```

At this point, the source program exists in a permanent segment in the user's working directory and is ready to be compiled. Another invocation of a text editor may be made to further alter the program contents. For example, the following editing sequence could be employed if the user decided to separate the hours, minutes, and seconds by commas instead of colons:

```
qedx                          (invoke the editor)
r time-and-date.cobol         (segment to be edited)
/:/                           (locate line containing colon)
 inspect time-out replacing all spaces by ":".
s/:/,/ p                      (change the colon to a comma and print results)
 inspect time-out replacing all spaces by ",".
w                             (permanently change the segment)
q
r 1447 1.773 3.747 47         (system ready for next command)
```

The source segment time-and-date.cobol is again ready to be compiled.

COBOL REFERENCE FORMAT

The COBOL language is specified in terms of a reference format that divides input source lines into areas. These areas control the interpretation of certain words and characters and exert constraints on the placement of certain constructs. For example, a hyphen in the Indicator Area has a different meaning than a hyphen appearing elsewhere, whereas division and procedure names are required to begin in Area A (defined below).

Fixed Format on Multics

The source program consists of a stream of characters divided into lines. Each line is variable in length and is delimited by a newline character (012 octal). The newline character itself is never considered part of the line.

The reference format of a COBOL source program is described in terms of character positions on a line, traditionally corresponding to columns on an 80-column card. The COBOL language itself is sensitive to areas defined by such column positions. For Multics COBOL, these are defined in Table 2-1.

Table 2-1. Fixed Format Areas

Character Position	Area Name
1 - 6	Sequence Number Area
7	Indicator Area
8 - 11	Area A
12 - 255	Area B

CONVERSION CONSIDERATIONS

Because any line may be of length 0 to 255 inclusive, all of the areas are of variable length. Additionally, since the positioning of the newline character determines the length of Area B, no right-hand margin exists in which comments may appear. Often, with card-oriented reference formats, Area B has a fixed length, extending from positions 12 through 72, thus allowing an eight-character right-hand margin for comments. Modifications must be made to existing programs that assume such a format. Also, for space considerations, trailing blanks that are left over after a card deck is converted to a Multics segment should be eliminated except where they are meaningful (as in continued nonnumeric literals; see "Special Characters in Nonnumeric Literals" below).

Area B is always variable in length. If the total line length is less than twelve, Area B is considered nonexistent, i.e., as a zero-length (or null) string of characters.

If the line length is less than eleven, Area A is assumed to contain the characters starting at position eight and extending to the end of the line, followed by as many blanks as necessary to total four characters. In this case Area B is nonexistent.

The Indicator Area may contain one of the following characters: b̸ (blank), -, *, /, or d. Any other value in this area is ignored and treated as though it were a blank with a warning message issued during compilation. If the total line length is less than seven, the Indicator Area is assumed to contain a blank character.

The Sequence Number Area is normally used for card numbers. Since Multics text editors are character-stream oriented rather than card-oriented, this area is usually not of use to COBOL users and may be left blank. All compiler diagnostic messages as well as run-time error messages referring to line numbers use the external line number. This is the actual relative number of the line in the segment and corresponds to the line number addressed by the text editors. The external line number is shown together with the corresponding line in the output listing if one is produced. COBOL accepts any value in the Sequence Number Area. If the line length is less than six, the sequence number is taken as the characters specified (if any) followed by as many blanks as necessary to total six characters.

For example, if the input were as follows (with b̸ representing the blank character):

```
b̸b̸b̸b̸b̸b̸b̸b̸b̸b̸b̸DISPLAYb̸"XYb̸
b̸b̸b̸b̸b̸b̸-b̸b̸b̸b̸"b̸b̸b̸
b̸b̸b̸b̸b̸b̸-b̸b̸b̸b̸"
b̸b̸b̸b̸b̸b̸-b̸b̸b̸b̸"Z".
```

the literal "XYb̸b̸b̸b̸Z" would be displayed. It would be invalid to insert a totally blank (or zero length) line between the first and last lines above as it would break the continuation of the literal.

Alternate Terminal-Oriented Format

COBOL rules concerning positioning by column number are not well suited to terminal input and tend to cause errors and annoyance for the interactive user. A free-form COBOL specification is available to the user to facilitate creating COBOL programs at the terminal.

FREE-FORM FORMAT DEFINITION

The expand_cobol_source command (abbreviated ecs) and the -format control argument of the cobol command allow the user to enter a COBOL source input file in free-form, typically through a terminal, and reformats each source line into the standard COBOL reference format. (See Table 2-1, above.) In a COBOL source program, statements generally begin in Area B. However, certain entries, such as COBOL-defined division names, section and paragraph names, level indicators, and special level numbers, must begin in Area A. In addition, special characters, such as the asterisk, slash, hyphen, and the letter d, have significant meanings when they appear in the indicator area.

If the -format (abbreviated -fmt) option is specified, a temporary internal file is generated in such a manner that source lines containing COBOL-defined names required to begin in Area A are actually interpreted as beginning in Area A. For example, lines beginning with level-numbers 01, 66, 77, or 88 are assumed to begin in Area A. Lines beginning with level-numbers 02 through 49 are assumed to be indented seven spaces plus the numeric value of the level-number. For example, level-number 02 begins at column 9; level-number 05, begins at column 12.

Some special characters force specific interpretation when they begin a free-form source line. The slash denotes a comment line with page ejection; the asterisk, a comment line without page ejection. The hyphen denotes a continuation line. For continuation lines, the remainder of the line following the hyphen is assumed to begin in Area B, since COBOL prohibits use of Area A in this case.

Debugging lines are probably of little interest to Multics COBOL users, due to the powerful symbolic debugging facilities available on an interactive basis, but they may be specified in free-form source by beginning the line with d*.

Source lines not beginning with a special character and not containing entries required to begin in Area A are assumed to begin in Area B. Any indentation already existing in the free-form file is thereby maintained relative to column 12. If a list file is produced, source lines appear in reformatted form.

The compiler converts all horizontal tab characters (ASCII 011) that are not contained in nonnumeric literals to spaces. The number of spaces is determined by subtracting the position of the tab character on the source line modulo 10 from 11. In this way, the user can use the tab character, a nonstandard COBOL character, as a formatting tool to input his source program.

If a free-form formatted program is invoked for a compilation and the -format control argument is not specified the compiler treats that source in a free-form manner. However, the following message is produced at the output device.

COBOL: The -fmt option is assumed since the source file is apparently in free format.

When debugging a COBOL program with its source in free-form format the Multics Debugger cannot display the source line. However, all other debugging facilities are available. An easy resolution to this situation is to make a permanent translation of the source.

Users wishing to create a permanent file in reformatted form may achieve identical functionality to that described above by using the format_cobol_source command. (Refer to the MPM Commands manual.)

PERMANENT TRANSLATION

COBOL source files may be entered and maintained in terminal-oriented format. In this case, the control argument available with the cobol command, -format, indicates to the compiler that the source file is in such a format. Alternately, a program entered through the terminal in this format can be permanently transformed to fixed format by the expand_cobol_source command. The usage of this command is:

```
ecs path1 path2
```

where path1 is the pathname of the segment containing terminal-oriented source statements (input) and path2 is the pathname of the segment that is to contain the translated fixed format statements (output). Full details are available in this manual.

COBOL SOURCE CODE

The basic unit of the COBOL language is the character. The character set used to form COBOL character-strings and separators includes the 26 letters of the alphabet, the 10 numeric digits, and the following 15 special characters:

```
b̸  +  -  *  /  =  $  ,  ;  .  "  (  )  <  >
```

Additionally, as an extension to standard COBOL, Multics COBOL allows the use of the lowercase in forming character-strings wherever the rules for their formation allow the use of uppercase. The following paragraphs discuss conditions under which uppercase and lowercase letters are treated as distinct characters and the instances where characters not in the COBOL character set are allowed.

Use of Non-COBOL Characters

The COBOL source program must consist entirely of characters in the COBOL character set except in the following instances, where any character in the Multics character set (with the exception of the newline character) may be used:

1. In comment lines (denoted by an asterisk in the Indicator Area)

2. In comment-entries of the Identification Division

3. In nonnumeric literals

4. In the internal-file-name specified in the SELECT clause (with the additional exclusion of the hyphen and blank)

The name specified with the PROGRAM-ID clause of the Identification Division and the text name specified in the COPY statement must contain only characters of the COBOL character set used for forming character-strings (i.e., the special characters, used as separators, are not available).

Although the name of a COBOL program may not contain non-COBOL characters, the program may call another program whose name contains such characters, because the object of a CALL statement is either an alphanumeric variable or literal, both of which may contain any character. Of particular interest is the underscore character that is used extensively in PL/I programming throughout the Multics system. The period often used in the names of Multics segments must never appear in the name of a COBOL source file nor in the <text-name> portion of the name of library text file. Conversely, if a COBOL program is to be declared by a PL/I program, it must not have a name containing the hyphen, for this would violate the syntax of PL/I.

Capitalization Considerations

COBOL source code can generally be written in uppercase, lowercase, or a combination of the two. Uppercase and lowercase letters are interchangeable in key words, variable names, and picture clauses. For example, the following statements are all equivalent:

```
01 NAME PIC XXBXX.
01 name pic xxbxx.
01 Name PiC xXbXx.
```

An attempt to define two data items with names differing only in capitalization (e.g., data-item and DATA-ITEM) results in a fatal diagnostic due to duplicate definition. Uppercase and lowercase letters are treated as distinct characters only in the following cases:

1. The contents of a nonnumeric literal maintains its case. Thus, "ABC" is not equal to "abc".

2. The name specified with the PROGRAM-ID clause of the Identification Division maintains its case. For example, a source program with the statement:

   ```
   PROGRAM-ID. ABC.
   ```

   is not equivalent to a program with a PROGRAM-ID of abc.

3. The name specified as the internal-file-name in the SELECT clause maintains its case although the device suffix does not. For example:

   ```
   select file-name assign to sw_name.1-printer
   ```

   is equivalent to

   ```
   select file-name assign to sw_name.1-PRINTER
   ```

   but not equivalent to

   ```
   select file-name assign to SW_NAME.1-printer
   ```

4. The text name given with the COPY statement maintains its case. For example:

   ```
   COPY ABC.
   ```

   is not equivalent to

   ```
   COPY abc.
   ```

Special Characters in Nonnumeric Literals

Only nonnumeric literals longer than 243 characters need be continued from one line to the next by use of the continuation convention (i.e., a hyphen in the Indicator Area in the following line). When such literals are continued, the actual number of blanks between the rightmost nonblank character on the line and the newline character are considered part of the literal. Such trailing blanks should be transferred to the beginning of the continued line, because the blanks are not readily apparent when the line is printed on the terminal or on the printed listing and the standard Multics terminal I/O module normally discards trailing white space (spaces or horizontal tabs).

Escape Convention

The newline character is not considered part of the source line and is not part of a continued literal. To include the newline character (or any other character) in a nonnumeric literal, a source-level escape convention is available. Any character within a nonnumeric literal can be represented by a string of digit pairs enclosed in quotation marks. The values are shown in Table 2-2. The leftmost digit represents the left five bits of the character, and the rightmost digit represents the right four bits of the character. For example, in the literal:

```
"AX$"3BHC"YZ"
```

AX$ and YZ represent normal ASCII characters. 3BHC represents the following bit pattern:

```
00011 1011  10001 1100
```

In Table 2-2, the digits in the top half of the table may be used as either the left or right digit. The lower half may be used only as the left digit.

The newline character in the ASCII character set has the octal value 012 and the hexadecimal value (the escape code) 0A. To define a data item one character long having the value of the newline character, the convention used is:

```
01 newline pic x value ""0A"".
```

To define the characters "A" and "B" separated by the newline character, the convention used is:

```
01 A-nl-B pic xxx value "A"0A"B".
```

Table 2-2. Escape Convention

Digit	Binary	Hexadecimal
0	0000	00
1	0001	01
2	0010	02
3	0011	03
4	0100	04
5	0101	05
6	0110	06
7	0111	07
8	1000	08
9	1001	09
A	1010	0A
B	1011	0B
C	1100	0C
D	1101	0D
E	1110	0E
F	1111	0F
G	10000	10
H	10001	11
I	10010	12
J	10011	13
K	10100	14
L	10101	15
M	10110	16
N	10111	17
O	11000	18
P	11001	19
Q	11010	1A
R	11011	1B
S	11100	1C
T	11101	1D
U	11110	1E
V	11111	1F

This convention is geared to the hexadecimal representation of an 8-bit character. A Multics character is nine bits long, and the system escape convention expresses the character as three octal digits preceded by a backslash character (\). For example, the newline character is represented by the characters \012. It is necessary to use the COBOL escape convention only when dealing with the newline character, the only character in a nonnumeric literal to which the compiler is sensitive. Other nonprinting characters can be included in the literal by typing the character itself or by using the Multics escape convention, if compatibility is not a consideration. For example:

```
01 beep pic x value "\007".
```

Expressing the newline character in a similar way would result in a fatal compile-time diagnostic.

NOTE: Use this feature only when absolutely necessary. The hexadecimal escape convention may change in future releases of the compiler. Furthermore, changes in future releases may make current programs operate incorrectly.

CONTROL DIVISION

Multics COBOL provides an optional Control Division to allow centralized control over certain aspects of the program. The Control Division, when present, is specified immediately before the Identification Division.

Sign Control

Standard COBOL assumes that, unless otherwise specified, the operational sign of signed numeric data is associated with some digit positions, rather than occupying a separate character. This convention is not well suited to the Multics hardware and separate sign data should be used whenever possible for efficiency considerations. Sign positioning can be controlled by the SIGN clause in the Data Division entry defining the data in question as follows:

```
[ DISPLAY SIGN IS { LEADING  } [ SEPARATE CHARACTER ] ]
                  { TRAILING }
```

To avoid adding a SIGN clause for all signed numeric data not already having one, the user may simply specify in the Default Section of the Control Division:

```
DEFAULT SIGN IS LEADING SEPARATE CHARACTER
```

or

```
DEFAULT SIGN IS TRAILING SEPARATE CHARACTER
```

Usage Control

Another useful function provided by the Control Division is controlling the default data type attribute of numeric data defined with the USAGE IS COMPUTATIONAL clause. The default for Multics COBOL COMPUTATIONAL is COMP-5, packed decimal. In some cases, the user may wish to treat computational data as though it were of DISPLAY or some other COMPUTATIONAL usage. To avoid changing the USAGE clauses for all computational data, the user may merely specify in the Default Section of the Control Division:

```
DEFAULT FOR COMPUTATIONAL IS DISPLAY
```

This will accomplish the same results. The complete COMPUTATIONAL format is:

```
                                       { DISPLAY          }
                                       { COMP-5           }
                                       { COMPUTATIONAL-5  }
                   { COMP          }   { COMP-6           }
[ DEFAULT FOR ]    {               } IS{ COMPUTATIONAL-6  }
                   { COMPUTATIONAL }   { COMP-7           }
                                       { COMPUTATIONAL-7  }
                                       { COMP-8           }
                                       { COMPUTATIONAL-8  }
```

For a complete description of the COMPUTATIONAL clause, refer to the Multics COBOL Reference Manual.

Precision Control

In addition, the NUMERIC LIMIT clause allows the user to specify the degree of precision he desires for intermediate results of arithmetic operations. If this clause is omitted, 30 positions are assumed. If an arithmetic result exceeds the size limit, only the specified number of most significant digits are retained.

The NUMERIC LIMIT clause has the following format:

```
[ DEFAULT FOR ] NUMERIC LIMIT IS integer-1
```

Descriptor Control

The GENERATE DESCRIPTOR clause allows the user to specify that either NO descriptors, or one of two types of descriptors are to be generated for CALL statements specifying arguments (via the USING clause).

They are:

1. SCALAR indicates that structures and arrays (tables) are to be described to the called program as a character string redefinition of the area they actually occupy. This is the way the COBOL language deals with such data. This is the default.

2. AGGREGATE indicates that structures and arrays are to be described with information about each individual component. This is the way the PL/I language treats aggregate data.

3. NO specifies that no descriptors are generated at object time when calls are made.

The GENERATE DESCRIPTOR clause has the following format:

```
                             { NO        }
[ DEFAULT FOR ]  GENERATE    { SCALAR    }  DESCRIPTORS
                             { AGGREGATE }
```

The GENERATE AGGREGATE DESCRIPTORS clause is used for the following purposes:

1. To communicate with PL/I program defining parameters with variable array extents.

2. To interface properly with the Multics data base manager MIDS (Multics Integrated Data Store) and MRDS (Multics Relational Data Store) which expects PL/I type descriptors in order to interpret the data passed to them.

The GENERATE NO DESCRIPTORS clause is used to cause efficient execution of the programs in situations where descriptors are not used by the called programs. Keep in mind that a COBOL program never requires that description be passed to it; however in this case, the -rck option's function of checking parameter validity is disabled. See "Interprogram Communication" in Section 5 for more details.

For a complete description of the GENERATE DESCRIPTORS clause, refer to the MULTICS COBOL Reference Manual.

COBOL LIBRARY FACILITY

The COBOL source text manipulation facilities are the COPY and REPLACE statements. These statements are described in the Multics COBOL Reference Manual, Order No. AS44. Programs which contain COPY REPLACING or REPLACE statements must be compiled using the -expand option (see "Source Transformations" in Section III of this manual). Programs which contain only simple COPY statements (i.e., COPY statements without the REPLACING option) need not use the -expand option.

The library facility allows a set (or sets) of source program statements that reside in a source library to be incorporated into a COBOL program. Data descriptions and procedures common to several programs can thus be maintained in only one location (the library) and included in all programs that require them. Multics COBOL supports the COPY statement, which inserts library text into the source program where it is treated by the compiler as a part of the source program.

General Format:

```
                     [ { OF }               ]
COPY text-name       [ {    }  library-name ]
                     [ { IN }               ]

  [            [   { ==pseudo-text-1== }      { ==pseudo-text-2== } ]     ]
  [            [   { identifier-1      }      { identifier-2      } ]     ]
  [ REPLACING  { , { literal-1         }  BY  { literal-2         } }...  ] .
  [            [   { word-1            }      { word-2            } ]     ]
```

Each set of source program statements subject to reference by a COPY statement is called a text item and is referenced by a text-name. Text items may be organized into a single directory or may be distributed among many directories, at the discretion of the user. Each text-name within any given directory must be unique. The COBOL constructs text-name and library are Multics entry names.

Definition of a Library

On Multics, a COBOL library may be defined as a set of segments existing in one or more storage system directories, each of which additionally:

1. Contains COBOL source code

2. Has a name in the form <text name>.incl.cobol where <text name> must contain characters only of the COBOL character set

3. Is accessible to the process performing the compilation at the time the compilation is being performed

DYNAMIC NATURE OF THE LIBRARY

If the text-name in a COPY statement is not qualified by a library-name, then the library text is located by using the Multics search facility with the "translator" or "trans" search list. For more information on the Multics search facility see add_search_paths in the MPM Commands manual.

If the text-name in a COPY statement is qualified by a library-name, then library-name specifies a Multics directory which must contain the segment:

text-name.incl.cobol

An absolute pathname for the include file is obtained by applying the Multics subroutine

expand_pathname_$add_suffix

to the following relative pathname

library-name>text-name

Format Restriction for Library Text

The -format control argument affects only the text in the source program itself, not the library text segments that may be referenced in the source. Thus, library text segments must be in fixed format at the time of compilation. If such a segment has been created in terminal-oriented format, it must be converted to fixed format by the expand_cobol_source command before compilation.

Text Comparison and Replacement

Source statements may be incorporated into a program intact by using a simple COPY statement, or may be modified as they are incorporated by using the REPLACING phrase of the COPY and the REPLACE statement.

The logic for text comparison and replacement is the same for both COPY...REPLACING and for REPLACE statements.

The text of a COBOL source program or library text item consists of a sequence of text words where a text word may be:

a. A separator, except for: a pseudo-text delimiter, and opening and closing delimiters for nonnumeric literals. The right and left parentheses are always considered text words regardless of their context.

b. A literal including, in the case of nonnumeric literals, the opening quotation mark and the closing quotation mark which bound the literal.

c. Any other sequence of contiguous characters except comment lines, bounded by separators, which is neither a separator nor a literal.

Characters recognized as separators are the space (or a series of spaces), the punctuation characters: comma, semicolon, and period (when one is followed by a space), left and right parenthesis, quote, and the pseudo-text delimiter.

Each text-word and each separator, other than a space or a series of spaces, is recognized as a text-unit, and it is on the basis of text-units that comparisons are made for purposes of replacement.

Example:

```
Margins
A    B                                                              R
_____________________________________________________________________
     .
     .
     .
     05  alpha picture s9(5).
     05  beta picture 9(4)v99.
     05  gamma picture 999v999, value 123.456.
procedure division.
begin.
     open input input-file, output output-file.
     display "Start program execution...".
     .
     .
     .
```

The text-words contained in the previous format are shown below in brackets:

```
[05] [alpha] [picture] [s9] [(] [5] [)] [.]
[05] [beta] [picture] [9] [(] [4] [)] [v99] [.]
[05] [gamma] [picture] [999v999] [,] [value] [123.456] [.]
[procedure] [division] [.]
[begin] [.]
     [open] [input] [input-file] [,] [output] [output-file] [.]
     [display] ["Start program execution..."] [.]
```

For purposes of replacement, a table of search operands is formed, consisting of all words, literals, identifiers, or pseudo-texts that precede the word BY in the COPY...REPLACING or REPLACE statement. In the case of an identifier or pseudo-text, the search operand may consist of a series of

text-words. A corresponding table of replacement operands is built consisting of all words, literals, identifiers, or pseudo-texts that follow the word BY in the COPY or REPLACE statement.

The library text is scanned, text-word by text-word, until a text-word or series of text-words is found that exactly matches one of the search operands.

Except for the very special case where the search operand consists solely of a comma or semicolon, the occurrence of a text-word consisting of a comma or semicolon separator is treated as a single space. For example, the text 'a, b, c' is equal to 'a; b; c' and to 'a b c' and to 'a b c'.

The continuation of lines has no effect on comparison. If a line contains a continuation indicator, the partial character-string at the end of the preceding line is simply concatenated with the partial character-string at the beginning of the continuation line, thus forming a complete text-word. For example, the pseudo-text ==add a b giving c== is found to match the following:

```
    Margins

    A   B                                                          R
    ________________________________________________________________

                          add a b giv

    -   ing c.
```

Comment-lines do not affect comparison since a comment-line is treated as a space, which is a separator, but not a text-word. Therefore, the pseudo-text ==move a to b c== is found to match the following:

```
    Margins

    A   B                                                          R
    ________________________________________________________________

        move x to y.  move a

    *      This is a comment line.

        to b c.
```

Comparisons are always made on the basis of complete text-words. A partial text-word that happens to match a search operand is not a candidate for replacement. For example, the pseudo-text ==master== does not match either of the following:

```
    Margins

    A   B                                                          R
    ________________________________________________________________

    fd  master-file
        .
        .
        .
        display "Error on master record.".
```

When the search operand is either an identifier or pseudo-text in which subscripting of indexing is expressed, the presence or absence of spaces has no

effect on comparison. The construct 'array(m,n)' is logically equivalent to 'array (m,   n)'.

Once a match is found between one of the search operands and the library text, the corresponding replacement operand is substituted for the matching text in the library.

If the replacement operand is a single word, literal, an identifier, or pseudo-text wholly contained within Area B, the replacement operand is simply substituted into the library text at the same position occupied by the text being replaced. If the replacement operand is longer than the text it is replacing, extra lines may be added to the library text.

Sometimes, however, the replacement operand must occupy a specific position within a source line. This is often the case when one or more complete lines of text are being substituted and they contain a level indicator or a paragraph-name that must begin in Area A, or when comment-lines are being inserted. In such cases, the replacement operand (pseudo-text-2) must be positioned with respect to standard COBOL reference format exactly as it is to appear in the resulting source program.

expand_cobol_source,ecs

The expand_cobol_source command applies a transformation to a COBOL source program. The nature of the source transformation is defined by control arguments. If no control argument is given, then a segment containing text of a standard format COBOL source program which possibly contains COPY and REPLACE statements is translated into an equivalent source program not containing these statements.

Usage

```
expand_cobol_source oldpath {newpath} {-control_args}
```

where:

1. oldpath
   is the pathname of the input segment. If the path does not have a suffix of .cobol, one is assumed. However the suffix .cobol must be the last component of the name of the source segment.

2. newpath
   is the pathname of the output segment. If the path does not have a suffix of .cobol then one is assumed. If this argument is omitted then the translated segment is in the form of the first component with the suffix .ex.cobol.

3. control_args
   can be selected from the following:

   -format, -fmt
   a pseudo free-form COBOL source program is translated into a standard fixed format COBOL source program. All characters in the source program are left exactly as typed.

   -upper_case, -uc
   translation to standard fixed format occurs as described for the argument, -format. All characters except for those in alphanumeric literals are converted to uppercase.

   -lower_case, -lc
   translation to standard fixed format occurs as described for the argument, -format. All characters except for those in alphanumeric literals are converted to lowercase.

   -card
   meaningless trailing blanks are deleted from a standard fixed format COBOL source program in card-image format. Characters in the identification field (columns 73-80) are ignored.

   -expand, -exp
   a standard fixed format COBOL source program which possibly contains COPY and REPLACE statements is translated into an equivalent source program not containing these statements. This argument is the default.

   -no_expand, -no_exp
   COPY and REPLACE statements in a standard fixed format COBOL source program are not translated.

The control arguments -format, -upper_case and -lower_case cause a pseudo free-form COBOL source program to be translated into an equivalent standard

fixed format COBOL source program. They may be used in combination with the control argument -expand but are inconsistent with the control argument -card. They should not be used if the source program is already in standard fixed format.

The control argument -card causes a standard fixed format COBOL source program in card image format to be translated into an equivalent standard fixed format program. Meaningless trailing blanks are deleted from each line in the source program. If a line is exactly 80 characters long then the identification field (columns 73 - 80) is deleted before removing meaningless trailing blanks. This argument may be used in combination with the control argument -expand but is inconsistent with the control arguments -format, -upper_case and -lower_case.

The argument -expand causes a standard fixed format COBOL source program possibly containing COPY and REPLACE statements to be translated into an equivalent standard fixed format program not containing these statements. The argument may be used in combination with the control arguments -format, -upper_case, -lower_case and -card. Expansion of COPY and REPLACE statements will take place unless the argument -no_expand is present. This argument is the default for the command.

SECTION III

COMPILING THE COBOL PROGRAM

Once a source program is available in a properly named segment and all library text segments referenced by the program exist in one or more directories specified in the current translator search rules, the source program may be compiled. This section discusses in detail the COBOL compiler, its available control arguments, and its output.

INVOKING THE COMPILER

The COBOL compiler is invoked by using the following command:

```
cobol path {-control_args}
```

where path is the pathname of the source segment that is to be translated by the COBOL compiler. If path does not have a suffix of cobol, then one is assumed. However, the suffix cobol must be the last component of the name of the source segment. For example, typing:

```
cobol >udd>PROJ>Smith>time-and-date.cobol
```

causes compilation of the source program contained in the segment named time-and-date.cobol that is contained (or linked to) in the directory >udd>PROJ>Smith. Since the source segment is required to have a name in the form <program name>.cobol, it is not necessary to specify the language suffix explicitly. Typing the command:

```
cobol time-and-date
```

causes compilation of the source segment time-and-date.cobol which, in this case, must exist in the current working directory.

Once the compiler is invoked, it confirms the existence of the named source segment and the validity of any specified options. After doing so, it prints the response message:

```
COBOL
```

indicating that compilation has commenced.

The <phase-diag number> actually contains a breakdown of <phase number> and <diag number> where the phases of the compiler are abbreviated as follows:

Phase Number	Phase Abb.	Description
1	LEX	Separates the source program into COBOL elements
2	IDED DDA	Identification and Environment Division processing, and Data Division and Allocation
3	DD	Data Division processing
5	PD	Procedure Division processing
6	REPL	Matches user words in the Procedure Division with definitions from the Data Division
8	CORR	Corresponding processing
L	LEVEL	FIPS Leveling processing

SOURCE ERRORS

The compiler diagnoses source program errors of various levels of severity: observations, warnings, fatal errors, and unrecoverable errors. Each is described in Table 3-1.

Table 3-1. Diagnostic Severity Levels

Severity	Description
1 (*)	Observation. Compilation continues without ill effect.
2 (**)	Warning. A possible error has occurred. The compiler attempts to remedy the situation and continues, possibly without ill effect. The assumptions the compiler makes to remedy the situation, however, do not guarantee the right results.
3 (***)	Fatal Error. An uncorrectable error has occurred. The program is definitely in error and no meaningful object code can be produced, but the compiler can continue executing and diagnosing further errors.
4 (****)	Unrecoverable Error. The compiler cannot continue beyond this error. A message is transmitted through the error_output I/O switch and compilation terminates.

Diagnostic Format

Diagnostics are printed on the terminal immediately following the response message. Each diagnostic along with the associated source line is given in the following format:

```
    <external line number>   <source line image>
                                <error indicator>...
 <severity indicator> <error indicator> <phase-diag number> <diag text>
```

The <external line number> is the relative number of the line in the source segment, the content of which is <source line image>, including the Sequence Number Area. The <error indicator> is a digit used as a pointer. It is positioned under the word or character of the source line that caused the compiler to determine an error existed. Where more than one error occurs on one line, the digit 1 is positioned under the first incorrect word, 2 under the second, and so on. There would follow as many diagnostic message lines as <error indicator>'s under <source image>. In this way, no source line is ever printed more than once. The <severity indicator> is either *, **, or ***, corresponding to severity 1, 2, and 3, respectively . Both <diag number> and <diag text> are explained in examples below.

After all diagnostics are printed, the compiler returns to Multics command level. Note: No more than 300 diagnostics can be printed on the terminal.

In the program time-and-date shown in Section II, if the statement on the sixteenth line contained a misspelling of the variable name "date-in", the following sequence would occur:

```
!   cobol time-and-date
    COBOL

    cobol: 1 fatal error encountered in time-and-date
        16        begin. accept daye-in from date.
                                 1
    *** 1 6-2   Data-name not declared
    cobol: Translation failed
    r 2359 0.368 2.034 58
```

The user would again be at command level. If the spelling were corrected, but the period following the paragraph name "begin" on the same line was omitted (a nonfatal error), the following sequence would occur:

```
!   cobol time-and-date
    COBOL
        16        begin accept date-in from date.
                         1
    ** 1 5-7  PERIOD expected after the previous word
    r 2359 0.369 2.035 59
```

An object program would be produced in this case.

Controlling Terminal Output

The user can control the type of diagnostics reported and the amount of information printed on the terminal by specifying arguments when the compiler is invoked.

SEVERITY CONTROL

The -severity*i* control argument (abbreviated -sv*i*) controls the number of diagnostics printed by allowing only those having a specified severity level *i* or higher to be reported. If this control argument is not specified, -sv2 is assumed as a default, and warnings, fatal errors, and unrecoverable errors are reported. This argument has no effect on the list file, if one is produced, which includes all diagnostics. See "List File," below.

Suppression of Warnings and Fatal Errors

It is sometimes desirable to suppress warning messages. Some COBOL programs may contain such messages, yet still run correctly as programmed. To suppress these messages, the user types:

```
cobol warningful -sv3
```

where warningful.cobol is such a program.

Even fatal errors can be suppressed by specifying the control argument -sv4. A message is issued indicating that no object segment is produced if fatal errors are encountered. The user can refer to the list file to determine the errors that were detected.

Reporting Observations

Observations are not normally displayed on the terminal. They exist only in the list segment. Often, such observations involve the moving of data from one field to a shorter field (causing truncation) or the moving of a signed value to an unsigned numeric field (causing loss of sign).

To print observations on the terminal, the user must override the compiler's default action by specifying the -sv1 control argument.

Unrecoverable Errors

Unrecoverable errors occur when the compiler reaches an internally inconsistent or unexpected state during execution. The diagnostic issued in these cases has a special format with the phase or module discovering the condition usually identified, along with a descriptive message. This type of error should not normally occur. If it does, the error should be reported as a compiler bug. By eliminating some sequence of invalid source code, the user can circumvent such errors.

When an unrecoverable error occurs, compilation is immediately terminated. The maximum value of _i_ in the -sv_i_ control argument is 4; -sv4 indicates only unrecoverable errors are to be reported.

REPETITION CONTROL

The -brief control argument (abbreviated -bf) suppresses the text portion of the diagnostic after it is printed once. If the time-and-date program shown in Section II contained misspellings in lines 16 and 18, the following sequence would occur:

```
!   cobol time-and-date -bf
    COBOL

    cobol: 2 fatal errors encountered in time-and-date.cobol

        16       begin. accept daye-in from date.
                                  1
    *** 1 6-2   Data-name not declared

        18            accept tide-out from time.
                             1
    *** 1 6-2
    cobol: Translation failed
    r 1111 0.347 1.999 23
```

The -brief control argument, like the -severityi control argument, applies only to messages displayed on the terminal and has no effect on the list file.

LEVELING

Use of the -leveli control argument (abbreviated -levi where i equals a number 1 through 5) restricts the user to a subset of the Multics COBOL language. The five values for i are:

1 low level

2 low intermediate level

3 high intermediate level

4 high level

5 Multics COBOL extensions

If a program compiles without any L type diagnostics it means the program is an acceptable subset of Multics COBOL at the level requested. Reference the Federal Information Processing Standards Publication December 1, 1975 (FIPS PUB 21-1) and the Multics COBOL Reference Manual for complete details. L type diagnostics which appear when -lev4 is specified describe extensions to ANSI-COBOL which are provided by Multics COBOL. The following example shows the results of a program compiled with the -lev1 option.

```
    1     IDENTIFICATION DIVISION.
    2     ENVIRONMENT DIVISION.

     ** 1 2-4   The PROGRAM-ID paragraph is missing - entry name assumed
                identical to object segment name
    *** 1 L-161 FIPS level restriction [extension]: missing PROGRAM-ID
                paragraph supplied

    3     CONFIGURATION SECTION.
    4     SOURCE-COMPUTER. MULTICS.
    5     OBJECT-COMPUTER. MULTICS SEGMENT-LIMIT IS 10.
```

```
                                                      1
*** 1 L-46  FIPS level restriction [2SEG- high level]: SEGMENT-LIMIT clause
6       DATA DIVISION.
7       WORKING-STORAGE SECTION.
8       01  GROUP-ITEM.
9       02  ELEM-ITEM PIC xxxxxx.
10      PROCEDURE DIVISION.
11      SECT-0 SECTION.
12      PAR-0.
13          MOVE "abcdef" TO ELEM-ITEM.
14      PAR-1.
15          CALL "EXTERNAL-PROCEDURE".
                 1
*** 1 L-66  FIPS level restriction [1IPC- low intermediate level]:
            CALL statement

16          CALL ELEM-ITEM.
                 1
*** 1 L-147 FIPS level restriction [2IPC- high intermediate level]:
            CALL identifier statement
```

The severity of the L type diagnostics may be controlled by the -levelij control argument, where j = 1, 2 or 3 and specifies the severity. The default (as illustrated above) is j = 3. If the control argument -leveli is used with i = 1, 2, 3, or 4 then non-L type diagnostics having severity 1 or 2 are not printed. Many of these diagnostics are duplicated by L-type diagnostics.

LIST FILE

A list file can be produced as the result of compilation if one of the following control arguments is specified:

*

-map
-list (-ls)

If none of these is used, the compiler will not produce a list file. Otherwise, a segment or multisegment file named <program name>.list is produced in the user's working directory, where <program name>.cobol is the name of the source segment. If such a file already exists, it is replaced by the new one.

The list file is intended for printing on a line printer (i.e., printed using the dprint command). The form-feed character is used to separate pages, and since this has no effect on most terminals, printing a list file on a terminal is not recommended. In addition, this file generally is large. Normally the list file is queued for printing.

It is general practice to delete list files from the storage system after printing as they are space consuming, they serve no purpose not met by the hard copy listing, and they can always be reproduced through recompilation.

The list file may contain the source listing, the cross-reference listing, the object map, and the object listing. The effect of each of the arguments on the production of these components is shown in Table 3-2.

Table 3-2. Summary of List Arguments

Option Name	Source	X-Ref	Map	Object List
-map	X	X	X	
-list (-ls)	X	X		X

List Header

A list file always starts with a block of information called a header. The header contains the program name, date and time of compilation, the compiler version used, and arguments, if specified, in the following format:

```
COMPILATION LISTING OF SEGMENT <name>
Compiled by: Multics COBOL, <version>
Compiled on: <date> <time>
    Options: {-control_args}
```

This information is permanently associated with the object segment resulting from a successful compilation. An object segment can be examined at any time for this and additional structural information by using the print_link_info command. In this way, the correspondence between object segment and list file can be established or verified.

Source Listing

The source listing is a line-numbered, printable ASCII listing of the source program. The entire source line image for each line is presented, always in the fixed reference format (see "Fixed Format on Multics" in Section II). The line number shown with each line, referred to as the external line number, represents the relative position of that line in the source segment as determined by a count of newline characters. This is the line number permanently associated with the statement(s) on that line for communication with the source-level debuggers as well as the reporting of run-time errors (see Section V). It is also the line number referenced in the additional output provided for the -map and -list control arguments.

If the source segment time-and-date.cobol shown in Section II were compiled with the -map control argument, a segment named time-and-date.list would be created in the user's working directory with contents as follows:

```
COMPILATION LISTING OF SEGMENT time-and-date
Compiled by: Multics COBOL, Version 3.0 of September 23, 1977
Compiled on: 10/01/77  0909.8 mst Sat
    Options: list, map, table;

        1       IDENTIFICATION DIVISION.
        2       program-id. time-and-date.
        3       ENVIRONMENT DIVISION.
        4       CONFIGURATION SECTION.
        5       source-computer. Multics.
        6       object-computer. Multics.
        7       DATA DIVISION.
        8        WORKING-STORAGE SECTION.
        9        01 time-out pic 99B99b99pp.
       10        01 DATE-OUT PIC x(8).
       11        01 date-in.
       12            02 yy pic 99.
       13            02 mm pic 99.
       14            02 dd pic 99.
       15       PROCEDURE DIVISION.
       16       begin. accept date-in from date.
       17           string mm "/" dd "/" yy delimited by size into date-out.
       18           accept time-out from time.
       19           inspect time-out replacing all spaces by ":".
       20           display "Time: " time-out "  Date: " date-out.
       21           exit program.
```

If source errors existed in the program, the diagnostic messages would appear interspersed with the lines of the source program in the format used for terminal diagnostics (see "Diagnostic Format" in Section III). This includes all observations, warnings, and fatal errors. Up to 3000 diagnostic messages can be printed in the listing. If more than 3000 occur, the user is so notified by a message on the terminal.

DATE-COMPILED PARAGRAPH

The date and time of compilation in the header corresponds to that generated for the DATE-COMPILED paragraph of the Identification Division. The characters following the colon on the third line of the header are identical to those following the period after the DATE-COMPILED key word in the source listing. Had time-and-date used this paragraph, the list file would show:

```
    .
    .
    1       IDENTIFICATION DIVISION.
    2       program-id. time-and-date.
    3       DATE-COMPILED. 10/01/77  0909.8 mst Sat
    .
    .
```

This information appears only in the list file. The source file is never modified as the result of a compilation.

Cross-Reference Listing

The cross-reference listing is an alphabetized list of all data names declared in the Data Division, files selected in the Environment Division, and paragraph and section names defined in the Procedure Division, along with, for each:

1. Its attributes (usage, level number, and picture, if applicable)
2. Its allocation location
3. The line number on which it is defined
4. The numbers of all lines containing a reference to it

For example, if the source file time-and-date.cobol were compiled with the -map control argument, the segment time-and-date.list would be created exactly as shown in "Source Listing," with the addition of a page skip and the following information:

```
IDENTIFIER LN  TYPE OFFSET      USAGE/CLASS     PICTURE         DEF. REF. LINES

begin          TEXT 000065      paragraph-name                  def 16 NOREF
date-in    01  DATA 000006      GROUP alphanum  X(6)            def 11 ref 16
date-out   01  DATA 000004      DSPLY alphanum  X(8)            def 10 ref 17 20
dd         02  DATA 000007      DSPLY numeric   9(2)            def 14 ref 17
mm         02  DATA 000006(18)  DSPLY numeric   9(2)            def 13 ref 17
time-out   01  DATA 000002      DSPLY num-edit  9(6)P(2)EDITED  def  9 ref 18 19 20
yy         02  DATA 000006      DSPLY numeric   9(2)            def 12 ref 17
```

Item names are given in lowercase regardless of how they appear in the text in either their definition or references.

USAGE

A data item's usage is determined by the USAGE clause (if given) or by default. The possible values in this case reflect all supported data formats, specifically: DISPLAY, COMP, INDEX, COMP-5, COMP-6, COMP-7, and COMP-8. If no USAGE clause is specified, the default is DISPLAY.

The organization, access mode, and associated device for a file are established, explicitly or implicitly, in the SELECT clause of the Environment Division. These are abbreviated and given in the following format:

organization/access mode - device

e.g., SEQ/SEQ-TAPE and REL/DYN-VRTL.

Procedure names are identified as either PARAGRAPH NAMES or SECTION NAMES.

Two additional types of usage are allowed: alphabet-names and mnemonic-names. Alphabet-name indicates a user-defined alphabet-name, as defined in the SPECIAL-NAMES paragraph. A mnemonic-name indicates a user-defined mnemonic-name defined in the SPECIAL-NAMES paragraph.

ALLOCATION

Storage allocation information is given in terms of a TYPE and OFFSET. The possible values of TYPE are:

TEXT - data is allocated in the object segment.

DATA - data is allocated in a temporary data segment.

PARM - data allocation is established by the calling program.

SYS - data allocation is in system storage.

The OFFSET field for parameter data (PARM) contains the relative number of the parameter in the calling sequence as expressed in the USING phrase of the PROCEDURE DIVISION header. For the other data types, it is the word offset of the leftmost location occupied by the data in the applicable segment. If the item is not word aligned, a bit offset is also given, within parentheses.

Data Division data is allocated according to the section in which it is defined as follows:

Section	Type
WORKING-STORAGE	DATA
CONSTANT	TEXT
FILE	DATA
LINKAGE	PARM
COMMUNICATION	DATA

The allocation associated with a file is that assigned for its use as a record area when I/O operations are performed on it. This allocation is contained in the temporary data segment and is identified by the TYPE DATA.

The location of paragraph names and section names is associated with the first machine instruction generated for the first executable statement following the particular name. Therefore, the TYPE value given for them is always TEXT and the OFFSET is the word address in the object segment of that instruction, i.e., the offset relative to the beginning of the segment. No bit offset can appear in this case, as all machine instructions are word aligned.

The OFFSET value associated with TEXT allocations is meaningful only when an object segment is produced. Since code is not generated if fatal errors exist in the source program or the -check control argument has been specified (see "Compilation Arguments" below), the OFFSET field is left blank in these cases. All other information in the cross-reference listing is identical, whether or not object code results from the compilation.

For example:

```
STORAGE REQUIREMENTS FOR THIS PROGRAM.

          Object     Text     Defs     Link     Symb   Static
Start          0        0      143      202      306      212
Length       642      143       36      104      322       70

External procedure time-and-date uses 160 words of temporary storage
  FILE SECTION:               No files defined
  WORKING-STORAGE SECTION:    22 characters for general storage with 0 items
                              initialized    (DATA:2->6)
  COMMUNICATION SECTION:      No data defined
  LINKAGE SECTION:            No parameters defined
  CONSTANT SECTION:           No constants defined explicitly;
                              44 words for constant storage    (TEXT:0->53)
Total allocation required for COBOL data is 8 words

THE FOLLOWING EXTERNAL OPERATORS ARE CALLED BY THIS PROGRAM:

     dsply_user_output       accept_date           accept_time

THE FOLLOWING EXTERNAL ENTRY IS CALLED BY THIS PROGRAM:

     cobol_rts_
```

Object Listing

The -list control argument (abbreviated -ls) causes the compiler to produce a list file containing a source listing, a cross-reference listing, and an assembly-like listing of the generated object code. This shows the machine language actually produced for each statement of the source program in octal format, together with a pseudo assembly language instruction interpretation consisting of an operation code, base register, and modifier mnemonics. Additionally, if the address field of the instruction uses the IC (self-relative) modifier, the text offset and contents of the word corresponding to the resolved relative address are printed in the remarks field on the line. *

If time-and-date were compiled with the -list control argument specified, the segment time-and-date.list would be created exactly as indicated in "Object Map" above, except that in place of the object map, the object code listing below would appear.

Object Map

The -map control argument causes the compiler to produce a list file containing a source listing, a cross-reference listing, and an object map. The object map is a table giving the location of the object code generated for each statement in the Procedure Division.

Each statement is identified by the external line number (i.e., the relative line number in the segment, not the sequence number) of the line on which it starts. The location of the generated code is indicated by the word offset in the object segment of the first machine instruction generated for the statement. If more than one statement appears on a given line, an identical number of entries will exist in the object map with the same line number component but different object location components. In this case, the sequence of the statements on the line corresponds with the sequence of the entries in the map for that line.

As an example, if time-and-date.cobol were compiled with the -map control argument specified, the segment time-and-date.list would be created and consist of the source listing exactly as shown in "Source Listing" above, followed by the cross-reference listing shown in "Cross-Reference Listing" above, which in turn would be followed by this object map:

```
LINE   LOC LINE   LOC LINE   LOC LINE   LOC LINE   LOC LINE   LOC LINE   LOC
 15 000054  16 000065  17 000071  18 000102  19 000107  20 000113  21 000136
```

If no object code is produced, no mapping can be made between the source statement and the machine code, and the object map is omitted.

Additionally, the compiler produces a program summary which describes:

- Storage requirements for this program.
- External procedure information on:

  Program name (number of temporary words used)
  File section
  Working-storage section
  Communication section
  Linkage section
  Constant section
  Total allocation required for COBOL data

- A list of External operators called by this program
- A list of External entries called by this program

```
OBJECT CODE
                                                        STATEMENT  1 ON LINE  15
     PROCEDURE DIVISION.

    54     000000 0000 00        ....       0
    55     000027 2000 00        cnax0      27
    56     000240 6270 00        eax7       240
    57    7 00050 3521 20        epp2       pr7¦50,*
    60    2 00006 2721 20        tsp2       pr2¦6,*
    61     000000 0000 75        ....       0,*7
    62    0 00102 0001 24        ....       pr0¦102,ic*
    63    4 00016 0541 00        aos        pr4¦16
    64    0 00100 0111 00        nop        pr0¦100

begin
                                                        STATEMENT  1 ON LINE  16
        begin. accept date-in from date.

    65    0 00054 7001 00        tsx0       pr0¦54
    66    6 00130 2371 00        ldaq       pr6¦130
    67    3 40006 7551 00        sta        pr3¦-37772
    70    3 40007 5521 60        stbq       pr3¦-37771,60

                                                        STATEMENT  1 ON LINE  17
        string mm "/" dd "/" yy delimited by size into date-out.

    71    057 100 100 500        mlr        (pr),(pr),fill(057)
    72    3 40006 40 0002        desc9a     pr3¦-37772(2),2
    73    3 40004 00 0003        desc9a     pr3¦-37774,3
    74    057 100 100 500        mlr        (pr),(pr),fill(057)
    75    3 40007 00 0002        desc9a     pr3¦-37771,2
    76    3 40004 60 0003        desc9a     pr3¦-37774(3),3
    77    3 40006 2351 00        lda        pr3¦-37772
   100     000022 7310 00        ars        22
   101    3 40005 5511 14        stba       pr3¦-37773,14

                                                        STATEMENT  1 ON LINE  18
        accept time-out from time.

   102    0 00056 7001 00        tsx0       pr0¦56
   103    100 004 024 500        mvne       (pr),(ic),(pr)
   104    6 00130 03 0010        desc9ns    pr6¦130,10,0
   105     777746 00 0007        desc9a     -32,7              000011 = 070322201322
   106    3 40002 00 0010        desc9a     pr3¦-37776,10

                                                        STATEMENT  1 ON LINE  19
        inspect time-out replacing all spaces by ":".

   107    000 100 160 500        mvt        (pr),(pr),fill(000)
   110    3 40002 00 0010        desc9a     pr3¦-37776,10
   111    3 40002 00 0010        desc9a     pr3¦-37776,10
   112     777701 0000 04        arg        -77,ic             000010 = 000001002003

                                                        STATEMENT  1 ON LINE  20
        display "Time: " time-out "Date: " date-out.
   113    0000362350 07          lda        36,dl
    .
    .
    .
   END OBJECT CODE
```

The object map is not given when the -list control argument is used. Information contained in the map is given in the object listing, although expressed in a different form. That is, the entry following "STATEMENT 1 ON LINE 16" shows that the first instruction associated with line 16 is at text offset 25. Therefore, if both an object listing and object map are wanted, both -list and -map must be specified. In this case, the map precedes the list.

Creation of an object listing significantly increases compilation time. Thus, the -list control argument should be used only when detailed information is required. Normally, use of this argument can be avoided by using the Multics debug command to display interactively that portion of the object code actually needed.

Like the object map, the object listing is meaningful only if object code is produced. If this is not the case, no object listing is produced.

ADDITIONAL COMPILER ARGUMENTS

Additional control arguments supported by COBOL are discussed below.

Object Code Suppression

It is possible to compile a program only for the purpose of syntactic and semantic checking and avoid the time and cost of code generation. Also, it may be desirable to produce a cross-reference listing and/or source listing for reference purposes only, for example, to plan future modifications. Both these listings can be produced without code generation. For this purpose, the -check control argument (abbreviated -ck) is available. When this argument is specified, use of the -map and -table control arguments is meaningless.

Run-Time Error Checking

The -runtime_check control argument (abbreviated -rck) causes the compiler to generate code that checks certain conditions during execution of a program. Specifically, it allows the user:

1. To validate the number and types of parameters

2. To check bounds on all subscripted references

3. To check the string range of all variable-length string references

4. To verify the legality of every index-name modification

Checking is not normally done during program execution due to the additional overhead it requires and because such errors do not occur in a fully debugged program. However, when a program is in the process of being debugged, it is useful to use this option to cause the occurrence of such conditions to be reported and the program halted rather than letting the program continue executing only to fail elsewhere either by producing wrong results or aborting. Also, the user is afforded the opportunity to correct many such errors "on the fly" by using the Multics debugging facility and then continue execution (see "Run-Time Errors" in Section VI for further details).

The types of error situations checked for are discussed below. If any error occurs during execution of a program not compiled with the -rck option then results of further execution is undefined.

PARAMETER VALIDATION

When the -rck control argument is used, the object program checks to make sure that the number of parameters passed to it by its caller is the same as the number it expects (as indicated in the USING clause of the Procedure Division header). Also, if possible, it makes sure that the characteristics of each data item actually passed as a parameter (i.e., usage, picture, synchronization, etc.) match those of the corresponding Linkage Section data description entry. The only condition under which this parameter type validation is not performed is when the calling program is a non-COBOL program which does not pass descripters. For details on the relationship between COBOL data types and those in other languages (particularly PL/I), refer to "Interprogram Communication" in Section V.

When an error occurs, one of the following messages is issued:

<prog name>: Argument mismatch - incorrect number of arguments supplied.

\- or -

<prog name>: Argument mismatch - data type is not compatible with that expected.

Additional information is given (refer to run-time errors in Section VI) and program execution is suspended. The user can ignore the error and continue execution by invoking the start command at this point. However, results are undefined if the program executes a statement that references a parameter which has not been supplied. If a data type mismatch occurs and execution is continued, all references to parameters are made according to the Linkage Section definition of the executing program (the callee). Thus, portions of the callers data may be inadvertently destroyed causing undefined results.

SUBSCRIPT RANGE CHECKING

When the -rck control argument is used, before every subscripted reference in the object program is made, a check is performed to make sure that the address determined by subscript calculation lies within the range of the table in question. Effectively, each subscript whose value is non-constant is compared to the bound of the corresponding dimension of the table (as specified in the relevant OCCURS clause), and an error occurs if its value is found to be outside that range. (Similar checking of constant subscripts is done at compile time; references that are out of range result in severity 3 diagnostics.)

When a subscripted reference is outside the range of the associated table, the following message is issued:

<program name>: A subscript is outside the range of the referenced table.

Additional information is given concerning the location of the error and source line number on which the reference was made, and execution is suspended. The user may then change the value of the subscript in error via the Multics debugging facility and continue execution by invoking the start command. If an attempt is made to continue execution and one or more subscripts is still out of

range, the same error occurs again. Thus, there is no way execution can continue with reference actually made to data outside the range of the table.

STRING RANGE CHECKING

Group items which contain subordinate data items having the OCCURS...DEPENDING ON clause are treated as variable length character strings. When such a group item is referenced, only that portion of it (as determined by the value of the DEPENDING ON variable) is involved in the operation. For example, in the following program:

```
01 vstring-length pic 999.
01 vstring.
   02 vchar pic x occurs 10 to 120 depending
      on vstring-length.
.
.
.
display vstring.
```

the number of characters displayed depend on the contents of vstring-length. When the -rck control argument is used, every reference to a variable length string is checked to make sure that the value of the applicable DEPENDING ON value is within the specified range.

When the value is outside the allowable range, the following message is issued:

<program name>: Out of range depending on item.

and program execution is suspended.

The user may then change the value of the DEPENDING ON item via the Multics debugging facility and continue execution by invoking the start command. If an attempt is made to continue execution and the value of that data item is still out of range, the same error occurs again. Thus there is no way that execution can continue causing the program to make reference to a string that extends beyond the defined range of the associated group item.

INDEX INTEGRITY VERIFICATION

When the -rck control argument is used, the object program checks every SET statement in which the possibility exists that an index name may take on a value outside the range of its associated table dimension. (A compile time check is made when SET references a constant value and a severity 3 diagnostic is issued for illegal assignments.) If such a condition occurs, the following message is issued:

<program name>: Attempt to set an index outside the range of the associated table.

Additional information is given concerning the location of the error and the source line number of the SET statement and program execution is suspended. The user may then change the value of the variable or the index name (for SET...TO), or the value of the index name (for SET...UP or DOWN) or both and continue execution by invoking the start command. If an attempt is made to continue execution and the execution of the SET statement still yields an

invalid index-name, the same error occurs again. Thus, there is no way in which the program can set an index name to an invalid value. Notice that because of this, indexed table references, unlike subscripted table references, are never checked for range errors, even when the -rck option is specified.

Source Transformations

Certain transformations can be made on a COBOL source program before it is compiled. These transformations are specified by control arguments which are defined below. They are a subset of the transformations available by using the expand_cobol_source command. If a source transformation is requested then the source program (e.g., name.cobol) is pre-translated and placed in the user's process directory with the suffix ex.cobol (i.e., [pd]>name.ex.cobol). If the segment being compiled already has the suffix ex.cobol then these control arguments are ignored.

The following control arguments may be used:

-card
: allows the user to compile a standard fixed format source program in card image format. Meaningless trailing blanks are deleted from each line in the source program. If a line contains precisely 80 characters, the characters in the identification field (columns 73-80) are removed before trailing blanks are removed. This argument may be used in combination with the -expand control argument but is inconsistent with the -format control argument.

-expand, -exp allows the user to compile a source program containing COPY and REPLACE statements. An equivalent source program is produced which does not contain these statements. This argument may be used in combination with the -format or -card control arguments. If the source program contains COPY REPLACING or REPLACE statements then this control argument must be used.

-format, -fmt
: allows the user to compile a source program that exists in terminal-oriented format. This is essentially free format and is described under "Alternate Terminal-Oriented Format" in Section II. This argument may be used in combination with the -expand control argument but is inconsistent with the -card control argument.

Run-Time Performance Measurement

The -profile control argument (abbreviated -pf) causes the compiler to allocate an additional block of information in which execution statistics are kept and to generate additional code with each source statement in order to record those statistics. Use of the -profile control argument significantly degrades run-time performance, but does allow the user to discover in which position of his program the most time is being spent in order to possibly improve or optimize that code. After execution of a COBOL program which has been compiled with the -profile option, the user then uses the profile command to display the statistics which have been recorded (see "Measuring A Program's Performance" in Section VII for additional information).

Source Level Debugging Requirements

In order to utilize the Multics debugging facilities, debug and probe, most effectively, it is necessary to produce a run-time symbol table as part of the object segment. This symbol table is produced by default. Use -no_table to prevent generation of the symbol table.

The run-time symbol table is recommended for use with those programs for which source-level debugging is needed or useful. Use -no_table with completely debugged programs used in a production environment, because of the additional storage space required for an object segment containing a table. Depending upon the amount of data defined in the program, this can significantly increase its size. Programs that are destined to be bound together using the Multics bind command may be compiled and checked out with a symbol table. When it comes time to produce the finished product, the various symbol tables can be discarded by the binder, eliminating the need for recompilation.

Programs which contain COPY REPLACING and REPLACE statements, or are in free format, undergo a complicated pre-translation before being presented to the compiler. Therefore debuggers cannot be used to see source lines at run-time. The compiler issues a warning diagnostic if the -table option is used in combination with either the -expand or -table options in the same compilation process.

Allocation for Temporary Compile-Time Files

Normally, all work files used by COBOL during compilation of a program are allocated in the process directory. The -temp_dir control argument (abbreviated -td) creates work files used by the compiler in a specified directory rather than in the process directory. Sometimes, when very large programs are compiled, quota is exhausted in the process directory, causing a record quota overflow error. The quota limit in the process directory is system dependent and cannot be controlled by the user. To circumvent this problem, the user should make sure the directory named after the -td option has sufficient quota for the compilation. When the compilation is finished, these work files are truncated. Another use of this argument is in the event a compiler error causes a fatal process error, thereby terminating the process and destroying all files in the process directory. If the work files are created in a permanent directory, their states at the time the error occurred can be seen.

Compiler Development and Testing Facilities

Two other arguments are available, but are probably of little use to most users. Designed for the use of compiler developers, they may, in some cases, be useful to the user who has discovered or is reporting a compiler malfunction.

The -time control argument (abbreviated -tm) causes the printing of the time, number of page faults, and number of prepages taken by each phase of the compiler. This information is directed through the user_output I/O switch. Once the information is printed for a phase, that phase is complete. This process isolates looping or unexpected errors at least to the phase level. Also, unusual paging figures may indicate possible sources of trouble in some situations.

The -debug control argument (abbreviated -db) leaves the work files generated by the compiler intact after normal or abnormal termination of a compilation. This argument is used primarily for debugging the compiler. The cobol$clean_up command may be used to discard these files once they are no longer needed. This argument causes severity 4 (unrecoverable) errors to avoid unwinding the stack after aborting the compilation. Instead, a new instance of the command processor is invoked at the point of the error, enabling the user to trace the stack to determine the exact sequence of program calls resulting in the error.

Argument Summary

A summary of all control arguments available with the COBOL compiler is given in Table 3-3. This information can be displayed online by typing cobol with no source segment or arguments specified.

Table 3-3. Summary of Compiler Control Arguments

Control Argument		Description
-map		Produces source, cross-reference, and object map listings.
-list	-ls	Produces source, cross-reference, and generated object code listings.
-table	-tb	Produces symbol table for symbolic debugging. (This is the default.)
-no_table	-ntb	Prevents the production of a symbol table for symbolic debugging.
-severityi	-svi	Suppresses printing of errors under severityi (0>i>4, default: -sv2).
-brief	-bf	Suppresses printing of duplicate error messages.
-format	-fmt	Accepts terminal-oriented format (format acceptable to the expand_cobol_source command).
-expand	-exp	Accepts source segments acceptable to the expand_cobol_source command.
-card		Accepts source statements in card image format.
-check	-ck	Bypasses code generation.
-time	-tm	Prints time and paging information for each phase of compilation.
-debug	-db	Leaves work files intact after compilation.
-temp_dir	-td	Creates the compiler's internal work files in the specified directory rather than in the process directory.
-runtime_check	-rck	Produces an object program in which various data integrity checks are performed throughout execution.
-profile	-pf	Produces object program which generates profile information.
-levelik	-levik	Produces leveling diagnostic for level i, where i is the level from 1 to 5 (default -lev5). The leveling diagnostics which result will have severity k where $1 \le k \le 3$ (default -levi3).

OBJECT SEGMENT

A normal compilation produces an object segment in the user's working directory. The following paragraphs explain the way in which this segment is created and give information concerning its format.

Segment Creation

If the source segment being compiled is named "X.cobol", the object segment created is named X. If a segment named X already exists in the working directory, its access control list (ACL) is saved and given to the new segment that replaces it. Otherwise, the user is given read and execute access (re) to the object segment with ring brackets v,v,v where v is the validation level of the process active when the compilation is performed. If a directory exists in the working directory with the name X, an error occurs and the compilation is discontinued. The directory is neither destroyed nor altered.

The user's arguments control creation of a list file. If created for the compilation of X.cobol, it is named X.list and its ACL is set as described for the object segment except that the user is given read and write access (rw) when it is created. Previous copies of X and X.list (if an appropriate argument has been specified) are replaced by the new segments created by the compilation.

Object Segment Format

The object segment produced by a COBOL compilation is in the standard Multics format for object segments described in the MPM Subsystem Writers' Guide. The object segment contains four sections (text, definition, linkage, and symbol), a map showing the location of each section, and a debug map, optionally appended by the debugger after compilation. A detailed breakdown of these sections is provided by the print_link_info command.

TEXT SECTION

The text section consists of two parts. The first portion contains all constants used by the program, both those data items defined in the Constant Section of the Data Division and those literals and figurative constants contextually defined through reference. The second portion contains all executable machine instructions. With this organization, all references to constant data are resolved by self-relative addressing. Additionally, since the object segment is not normally writable, any attempt to modify constants during program execution or debugging results in an access violation.

DEFINITION SECTION

The definition section contains information about externally visible data defined in the program, data used in parameter validation, and additional information used in resolving external references. In a COBOL program, the following entry points are externally visible:

1. The main entry point, which is defined by the name specified in the PROGRAM-ID paragraph

2. Optionally, uniquely named entries of compare routines called only by the Multics sort and merge

The information contained in the definition section allows other programs to call and (optionally) pass parameters to these programs.

LINKAGE SECTION

The linkage section contains information about externally visible data referenced by this program and per-process static data used in run-unit control (see Section V). No user-defined data appears here. Like the rest of the object segment, this section is not writable; therefore, it is copied into a combined linkage segment in the process directory when the program is first called in the process (i.e., when the segment is initiated). The writable linkage section serves as the means by which external references are dynamically resolved. Information describing these references is kept in the form of (initially unsnapped) links (not to be confused with storage system links). When a reference is resolved during execution, its link is said to be snapped. The original symbolic reference is changed to a virtual memory address at this time.

A link is created for each literal named in a CALL statement, as these constitute external references. (Take note that when a variable is used with the CALL statement, its resolution involves invocation of system subroutines.) Other links may be included such as a reference to the COBOL run-time support facility and references to the file state blocks (FSB) associated with files defined in the program (see Section IV).

SYMBOL SECTION

The symbol section contains information describing conditions under which the object segment was created, such as the name and project of the user who compiled it, the directory containing the source file used, and the date and time of the compilation. All information given in the header of the list file is recorded here and can be displayed by the print_link_info command. Also the symbol table produced by the -table control argument is placed in the symbol section. Finally, relocation information necessary to bind COBOL object segments is produced and stored in this section.

COMPILER CHARACTERISTICS

In the following paragraphs, characteristics of the COBOL compiler that are visible to or affect the user are discussed.

Reentrancy

Like most other language translators on Multics, the COBOL compiler is not recursive. The user can interrupt compilation and, at some later time, restart the compilation process. The user however, may not begin a second compilation without executing a release command or a new_proc command.

The release command is normally used explicitly to unwind the stack. If execution of the compiler (or any program) is interrupted (e.g., by the quit button or the signaling of an error), and there is no intention of continuing execution, the user should release the stack, or it grows unnecessarily large. If compilation is interrupted and the compiler's stack frame is released, any subsequent calls are not recursive.

*

Command Line Considerations

The following information deals with matters of interest involving specification of the cobol command on the command line.

ORDERING OF ARGUMENTS

Arguments can be given in any order on the command line, before or after the source program name. They are distinguished from the program name by the fact they start with a hyphen. For example:

```
cobol path -table -map
```

is equivalent to:

```
cobol -table -map path
```

which is also equivalent to:

```
cobol -map path -table
```

This convention implies the restriction that a source segment name must not begin with a hyphen.

MULTIPLE COMPILATIONS

The format of the cobol command accepts one and only one source program name. That is, an attempt to compile the three programs a.cobol, b.cobol, and c.cobol by the command:

```
cobol a b c -map
```

results in a diagnostic, and the compilation is aborted. However, it is possible to make use of the extended features of the Multics command language to accomplish such multiple compilation. The command line:

```
cobol (a b c) -map
```

is synonymous to the line:

```
cobol a -map; cobol b -map; cobol c -map
```

Additionally, to compile every COBOL source program in the working directory, simply type:

```
cobol ([segs *.cobol]) -map
```

*

This capability is not a function of the cobol command, but rather a general property of the Multics command language. Command language conventions are described in the MPM Reference Guide; active functions (such as 'segs' in the above example) are described in the MPM Commands manual.

Online Documentation

If the compiler is invoked with no source program name and no arguments given, a concise summary of available arguments is printed on the terminal, and control is returned to command level. This summary contains information similar to that shown in Table 3-3.

More detailed information about how to use the COBOL compiler is available through use of the help command as follows:

1. help cobol - Lists instructions for obtaining release-specific COBOL information

2. help cobol_implementation - Lists permanent restrictions, limitations, and a summary of useful notes applying to the most recent release

3. help cobol_status - Lists currently outstanding known COBOL difficulties

4. help cobol_changes - Lists new functionalities and changes to the compiler and run-time support system since the last release

SECTION IV

INPUT/OUTPUT PROCESSING

This section shows how the COBOL source I/O statements interface and interact with the system. The interpretation of the SELECT clause of the Environment Division, the File Description (FD) entries of the Data Division, and the various I/O statements and associated options of the Procedure Division are discussed in relation to the Multics storage system, the generalized Multics I/O facility, and the COBOL run-unit. Additionally, language extensions defined for the SELECT clause and I-O-CONTROL paragraph are explained.

TERMINAL I/O

The ACCEPT and DISPLAY statements allow communication with the user's terminal. With these statements, data is transferred directly to or from user-defined data fields through system-defined I/O switches. No files or record areas at the COBOL source-level are involved in this type of I/O.

Accepting Data

ACCEPT can be used both for reading data and for the non-I/O function of setting up data involving the time and date. The latter will be omitted in the following discussion. The format of ACCEPT is:

```
ACCEPT <identifier> [ FROM <mnemonic-name> ]
```

The ACCEPT statement initiates a read to the user_input I/O switch. This system-defined switch is normally attached to the user's terminal, but may be attached to a storage system file in the case of an absentee process or by the use of the &attach control line of an exec_com or of similar facilities. No interpretation or conversion of data type is attempted; data is transferred character by character regardless of the data type of <identifier>. This is determined by the combination of the USAGE and PICTURE clauses specified in the data description entry. In effect, it is assumed, but not necessary, that the data type of <identifier> is alphanumeric display. By knowing the internal representation of other data types and the character-string relationship, however, the user can use ACCEPT on any data type. For a description of all supported data types, see "Data Types" in Section V.

When an ACCEPT statement is executed, data is transferred from the user_input switch up to, but not including the first newline character encountered. No prompting message is issued to instruct the user to enter data. If he desires this, the user should execute a DISPLAY statement immediately before the ACCEPT statement. If more characters than can be contained in <identifier> are entered, the rest of the line (up to the newline character) is discarded. If insufficient characters are entered to fill <identifier>, then <identifier> is space-filled to the right. The effect of entering nothing as a response (i.e., a newline character only) is to blank out the field. To be sure a nonnull string is entered, the user may code as follows:

```
ASK.  ACCEPT ID  IF ID = ALL SPACES GO TO ASK.
```

without regard to the previous contents of ID.

Displaying Data

The format of the DISPLAY statement is:

```
         { <identifier-1> } [ , <identifier-2> ]
DISPLAY  {                } [                 ] ... [ UPON <mnemonic-name> ]
         { <literal-1>    } [ , <literal-2>   ]
```

The DISPLAY...UPON SYSOUT statement and the DISPLAY statement without the UPON phrase cause data to be transferred through the user_output I/O switch. This system-defined switch is normally attached to the user's terminal, but may be attached to a storage system file by the file_output command. DISPLAY UPON CONSOLE causes data to be transferred through the error_output I/O switch, another system-defined switch that is always attached to the user's terminal by convention.

When a DISPLAY statement is executed, data is transferred character by character from each specified <identifier> and/or <literal> specified with no interpretation or conversion performed. As with ACCEPT, it is assumed but not required that the usage of all variables is DISPLAY. No separator characters are produced between identifiers and/or literals; however, a newline is transmitted after the last <identifier> or <literal>. To print data on successive lines, the user must execute multiple DISPLAY statements or include the newline character in a <literal> (see "Escape Convention" in Section II) or <identifier>.

The statements

```
DISPLAY "ID1:", ID1.
DISPLAY "ID2 IS" SPACE ID2 ""OA"ID3 IS """" ID3 QUOTE.
```

will produce output of the form:

```
ID1:<value-of-id1>
ID2 IS <value-of-id2>
ID3 IS "<value-of-id3>"
```

Data should be directed to error_output only in special cases, such as when a situation arises in which normal processing cannot continue. The effect of using this switch instead of user_output is to bypass any active file_output command specification.

PERFORMING COBOL I/O ON MULTICS

The COBOL I/O operations are performed by a group of run-time subroutines that use the Multics system I/O modules as described in the MPM Subroutines and MPM I/O manuals. In this discussion, the input-output operation is considered as a single process of placing a logical record on a file or obtaining a logical record from a file, although it is recognized that the operation actually consists of a series of separate steps.

All I/O is performed through one or more I/O switches. Thus, each file defined in a SELECT clause is associated with a particular I/O switch at run-time. The following paragraphs describe the actual activities that take place when a file is opened, when data is transmitted to or from that file, and when the file is closed.

Opening a File

Using the COBOL OPEN statement to open a file involves a number of discrete steps. First, an I/O Control Block (IOCB) must be established for the particular I/O switch associated with the file in question. An IOCB is a process-local data block containing various control information for a file. It is a temporary structure in that it exists only for the life of the process. Functionally, the IOCB is the embodiment of the conceptual I/O switch. That is, it contains, among other things, the data that supports the switching mechanism, which in this case is a sequence of entry variables (in PL/I terminology) serving as a transfer vector. Every uniquely named switch in a process has its own such control block.

The next step is to attach the switch. This involves specifying a source/target for subsequent I/O operations and an I/O module that will perform these operations. The nature of the source/target (e.g., tape, a virtual memory segment, or a terminal) depends on I/O module. Table 4-1 lists those I/O modules supported by Multics COBOL. The attachment resulting from the execution of the OPEN statement is based on information the user provides in the Environment Division, as well as in some cases the current state of the switch. (For a complete description of I/O modules, refer to the MPM Subroutines and the MPM I/O manuals.)

After the I/O switch is attached, it should be opened. This prepares the switch for the particular mode of processing (e.g., reading records sequentially) using the already established attachment. This mode is determined by the organization and access mode as specified in the SELECT clause as well as the opening mode (INPUT, OUTPUT, I-O, EXTEND) given in the OPEN statement itself. For a mapping of the COBOL source designation into the Multics opening mode, refer to "File Opening Modes" later in this section.

Finally, the file state block (FSB) is updated to record the opening, and the various internal and external control data it holds are reinitialized. The FSB is further discussed in "Implementation Specifics," below.

Table 4-1. Multics I/O Modules

Module Name	Description
discard_	Provides a sink for output operations
tape_nstd_	Supports I/O from/to tapes in nonstandard or unknown formats
rdisk_	Supports I/O from/to removable disk packs
record_stream_	Maps stream calls into record calls or vice versa
syn_	Attaches an I/O switch as a synonym for another switch
tape_mult_	Supports I/O to and from Multics standard tapes
tape_ansi_	Implements the processing of magnetic tape files according to American National Standard COBOL
tape_ibm_	Implements the processing of magnetic tape files according to IBM standards
tty_	Supports I/O from/to terminal devices
vfile_	Supports I/O from/to files in the virtual memory storage system

Transmitting Data

At this point, the OPEN statement is completed, and the file is considered open and active in the run-unit. The subsequent execution of other COBOL I/O statements referring to this file will cause an appropriate transfer of data (or a control operation) to be directed through the associated I/O switch. However, just as an I/O switch cannot be opened until it is attached, it cannot be detached until it is closed.

Closing a File

Using the CLOSE statement to close a file also involves multiple steps. First, the switch is closed and detached (provided that it was attached by the program), and secondly if no errors have occurred, the FSB is updated to indicate the file is no longer open. Any subsequent attempt to perform I/O on this file while it is closed will result in a run-time error.

FILE CHARACTERISTICS AND DEVICE INDEPENDENCE

The following information describes those characteristics of COBOL files in relation to the Multics I/O system concepts involving device independence, specifically, separation of the attaching and opening functions.

File Sharing

On Multics, files can be shared on two levels: on the source/target or device level and on the I/O switch level.

With device-level sharing, two or more programs, not necessarily in the same process, reference the same device or pseudodevice through different I/O switches. Whether or not files can be referenced simultaneously depends on the nature of the device. For example, two programs could not control the reading or writing of the same tape device at the same time. However, given a pseudodevice such as the virtual memory storage system, it is possible for any number of programs to access the contents of a segment simultaneously. (This can be a source of trouble if, for example, two programs attempt to read and write an unstructured file at the same time. Protection is provided for the user in the case of storage system structured files in the form of file locking. However, the user must provide his own interlock mechanism when dealing with unstructured files.) Regardless of the device, however, a program that opens, writes, and closes a file through one I/O switch can then call another program that opens and reads the same file through an entirely different switch, thereby sharing the file only on the device level.

Switch-level sharing, on the other hand, occurs when two or more programs in the same process reference a file through the same I/O switch. Changing the attachment of such a shared switch affects all programs referencing through it. It is possible, in this situation, for one program to write a record in a file and then call on another program that writes another record in the same file, regardless of the device involved. A third program could then be called to close the file. It is also possible to associate two different I/O switches so that they behave as one and become, in effect, synonymous with each other. This is done by the syn_ I/O module, which attaches one I/O switch to another instead of to a device. In this case, an I/O switch itself is treated as a pseudodevice. This is discussed and shown in examples later in this section.

Scope of Files

All Multics COBOL files have an attribute referred to as "scope." The scope of a file, as determined by an option in the SELECT clause, is either internal or external. The concept of scope, or visibility, is an extension to the American National Standard COBOL definition and, indeed, need not even be considered when dealing with existing programs. However, it is crucial to effective file sharing.

If a file has internal scope, all related I/O operations are performed through a uniquely named I/O switch. This ensures that no inadvertent matching of switches in the same process will occur. Files defined in two programs with identical SELECT clauses will be referenced by the I/O operations within each program through different switches. Files not explicitly given the external attribute (i.e., files defined in existing programs) are considered internal. This allows the execution of any number of COBOL programs containing only internal files in a process with complete independence. Internal files may be shared on the device level by using the CATALOG-NAME phrase, a Multics extension to the American National Standard; however, switch level sharing is impossible.

A file must be explicitly defined as having external scope, if it is to be shared on the switch level. This is also true if the file is to be referenced by the io_call command (e.g., attached). With external files, the user specifies the name of the I/O switch that is to be used. External files declared identically in the SELECT clause in two different programs thus will be referenced by the I/O operations in each program through the same switch. For details on establishing the switch name, refer to "I/O Switch Specification," below.

On Multics, commands perform the functions of job control language (JCL) on a batch-oriented system. Since commands are nothing more than external executable programs, it is necessary to assign a file external scope if it is to be referenced by command (i.e., controlled in any way outside the program itself). The io_call command, for example, keys on the I/O switch name for all its operations; for internal files this name is unique and not readily apparent to the user.

Attaching from Command Level

Attaching an I/O switch (as well as any other I/O operation) can be done at command level by using the io_call command. In order to specify an attachment, the I/O module interface must be known. Details concerning the I/O modules shown in Table 4-1 can be found in the MPM Subroutines manual. A simple example would be to attach an I/O switch by the name "sw_name" to a file contained in the segment "seg_name" in the working directory. This attachment could be made from command level as follows:

```
io_call attach sw_name vfile_ seg_name
```

Once a switch is attached, it attains an "attach description," which is kept in the IOCB. This is a string of characters that describe the current attachment of the switch, with the null string indicating the switch is unattached. For example, after the above invocation to io_call, the resulting attach description for sw_name is "vfile_ seg_name".

The attachment made as the result of the execution of the OPEN statement in a COBOL program is made through a subroutine interface with iox_$attach_iocb rather than through the io_call command interface. However, the effect of the resulting attachment is identical.

Implications for the OPEN and CLOSE Verbs

Since an internal file cannot be referenced (e.g., attached or detached) outside the program that defines it, the COBOL OPEN and CLOSE statements always function exactly as described above in "Performing COBOL I/O on Multics." That is, OPEN always involves:

1. Attaching the uniquely named I/O switch

2. Opening the file through that switch

CLOSE always involves:

1. Closing the file through the switch

2. Detaching the switch

If the switch is already attached, a logic error has occurred in the execution of the program (i.e., two OPEN statements have been executed with no intervening CLOSE statement). In this case, a run-time error occurs. Likewise, an erroneous situation exists if, upon attempting to detach the switch, the program detects that the switch is not attached (i.e., a CLOSE statement has been executed with no preceding OPEN statement). The code generated for the OPEN and CLOSE statements controls the attachment of the associated I/O switch because of the switch's inaccessibility. (Actually, such a switch can be referenced outside the program that defines the associated file if the unique name is known. All switches and their current attach descriptions can be displayed by using the print_attach_table command. However, any reference made to an internal file's switch outside the program is considered illegal and results of continued execution of that program are undefined.)

External files, on the other hand, may be referenced by any number of programs. Thus, the OPEN and CLOSE statements must be aware of the state of the switch. For external files, execution of the OPEN statement attaches the switch, if it is not already attached; opens the file, if it is not already opened. Execution of the CLOSE statement closes the file, if it is not already closed, but only if a COBOL program caused it to be opened.

The I/O switch of an external file is detached only if a COBOL program actually caused it to be attached. That is, I/O switches attached by command or by the system (e.g., user_output) are never detached. I/O switches attached by the execution of an OPEN statement in a COBOL program are detached when the file is closed, unless otherwise specified by the APPLY clause of the I-O-CONTROL paragraph. (Refer to "Supplementary Options - APPLY Clause" later in this section.)

DEFINING A FILE

A file is defined in the FILE-CONTROL paragraph in the INPUT-OUTPUT section of the Environment Division. Additional information concerning I/O techniques may be specified in the APPLY clause of the I-O-CONTROL paragraph. The File Description (FD) entry of the Data Division provides label and record size information. Additionally, procedures that are to be performed upon encountering I/O-related errors for specific files or classes of files may be included.

The reader should have a basic understanding of the functions provided by the various I/O related statements and the standard options available with the SELECT clause. Details concerning these subjects can be found in the *Multics COBOL Reference Manual*.

File Selection - SELECT Clause

A SELECT clause is specified for each file referenced in the program. These clauses are grouped together under the FILE-CONTROL paragraph in the INPUT-OUTPUT section of the Environment Division. The complete format of the SELECT clause, showing all available options, appears on the following page. Specific formats for sequential, relative, indexed, and stream files are presented in detail in Section IX of the *Multics COBOL Reference Manual*.

FILE STRUCTURE

The following paragraphs provide information concerning logical file structures supported on Multics and the relationship of these file structures to the logical file types defined in a COBOL source program.

Organization

The Multics I/O system provides for three types of logical file structure: sequential, indexed, and unstructured. In COBOL, structure is determined for the most part by the value given in the ORGANIZATION clause with the mapping as shown in Table 4-2.

Table 4-2. File Organization

Organization	Multics File Structure
SEQUENTIAL*	sequential
RELATIVE	indexed
INDEXED	indexed
STREAM	unstructured
* when DEVICE is not PRINTER, CARD-PUNCH, or CARD-READER	

```
SELECT [OPTIONAL] [EXTERNAL] file-name-1

    ASSIGN [TO] internal-file-name-1 [ -VIRTUAL      ]
                                     [ -PRINTER      ]
                                     [ -CARD-PUNCH   ]
                                     [ -CARD-READER  ]
                                     [ -PREATTACHED  ]
                                     [ -TAPE         ]

    [ ; ORGANIZATION IS { [ ANSI    ]  SEQUENTIAL                              } ]
    [                   { [ MULTICS ]                                          } ]
    [                   { [ IBM-DOS ]                                          } ]
    [                   { [ IBM-OS  ]                                          } ]
    [                   {                                                      } ]
    [                   { INDEXED RECORD KEY IS data-name-4                    } ]
    [                   {                                                      } ]
    [                   {   [ ALTERNATE RECORD KEY IS data-name-5              } ]
    [                   {                                                      } ]
    [                   {        [ WITH DUPLICATES ] ] ...                     } ]
    [                   {                                                      } ]
    [                   { RELATIVE [RELATIVE KEY is data-name-3]               } ]
    [                   { STREAM                                               } ]

    [ ; ACCESS MODE IS { RANDOM     } ]
    [                  { SEQUENTIAL } ]
    [                  { DYNAMIC    } ]

    [ ; FILE STATUS IS data-name-1 [, data-name-2] ]

    [ ; [WITH] { FLR     } ] .
    [          { VLR     } ]
    [          { SPANNED } ]
```

Both RELATIVE and INDEXED files map into the same Multics file structure. A RELATIVE file on Multics is a special type of indexed file for which the COBOL I/O run-time system maintains a 12-character key with a value always corresponding to the relative number of the record in the file (in decimal representation).

STREAM organization is a Multics COBOL extension to the American National Standard that allows the user to define and reference Multics unstructured files. An unstructured file consists of a stream of 9-bit bytes, normally ASCII characters. Examples of an unstructured file are a list file or a source file. Records in such a file vary in length from zero to 1,048,575 bytes. They are delimited by the newline character, which is not considered part of the record.

A file with STREAM organization can be accessed only sequentially and cannot be opened in I-O mode. When a READ is performed, data is transferred up to, but not including the newline character. The short record error (01) is not produced for records that are shorter than the defined record. When a WRITE is performed, a newline character is appended to the contents of the record by the COBOL I/O run-time system. Thus, the user never need deal with this character. If a record is written that contains the newline character (octal 012), it causes the same effect as if i+1 records were written, where i is the number of newline characters contained.

Often in dealing with such files, it is useful to know the number of characters actually transmitted as the result of a READ, or to control the number of characters that are to be written in some way other than alternate record descriptions. A general mechanism that applies to files of all organizations is available for this purpose: the RECORD CONTAINS clause of the FD entry. This is described under "Variable-Length Records" later in this section.

Access Mode

The value of <access> must be one of the following:

SEQUENTIAL
RANDOM
DYNAMIC

This value defines the manner in which the records in the file are accessed; it does not define the structure of the file. If a file is to be accessed only sequentially or only randomly in a particular program, it is worthwhile to so indicate, as more efficient record access will result.

Programs may use two different access modes to access the same file. However, an external file that is shared by two or more programs on the I/O switch level must be defined in all such programs with identical access as well as identical organization. This is because such programs attempt to share the same opening modes for that file (see "Open Modes" below). If this happens, an attempt to open the file will result in a run-time error, indicating the file is being referenced in an inconsistent mode. An attempt to reference such a file without opening it causes undefined results.

Record Format

For labelled tape files, the optional key words VLR, FLR, and SPANNED denote the record format of the file. (Refer to the tape_ansi_ and tape_ibm_ I/O modules in the MPM I/O manual.) SPANNED is meaningful only for labelled tape files.

For all other files, VLR and FLR indicate that variable-length records or fixed-length records are to be written. This overrides the determination that would normally be made based on factors expressed in the corresponding FD entry for this file. (Refer to "Variable-Length Records" later in this section.)

KEYS

The identifier specified with either the RECORD KEY or RELATIVE KEY clause associates an index or a relative record number with each record, providing an access path to a file's records.

Record Key

For INDEXED files the RECORD KEY clause must be specified. The Multics I/O system does not require that the identifier representing the prime record key be defined in the Record Description entry for that file. This American National Standard limitation is relaxed in Multics COBOL. That is, the prime record key may be defined in the WORKING-STORAGE section, in the LINKAGE section (i.e., passed by a calling program as an argument), or even in the Record Description entry of another file. In any of these cases, an observation diagnostic (severity level 1) is issued when the program is compiled, indicating that a nonstandard usage exists. If alternate record keys are used, all keys (prime and alternate) must be defined in the Record Description entry for that file. In the above case, if the prime or alternate keys are not contained inside the Record Description, a fatal diagnostic (severity level 3) is issued when the program is compiled. However, the actual allocation of the record key identifier in no way affects the functioning of I/O operations.

The record key is used to establish the index of records being written. For files with SEQUENTIAL access mode, the record key must have an ASCII value for the current record that collates higher than the value of the key of the previous record written or a run-time error occurs. For files being read sequentially, the record key is not necessarily updated after each READ statement to reflect the key of the record just read. Had the record key been defined outside the Record Description entry for the file (and thus been allocated outside the file's record area), then execution of a READ statement does not affect its value, as is the case with the relative key (see below). Otherwise, it is updated due to the data transfer caused by the read.

The identifier specified as the record key must not exceed 256 characters in length. This is the limit imposed by the Multics I/O system.

Relative Key

For RELATIVE files, The RELATIVE KEY clause may or may not be specified, depending on the access mode. If the file is to be accessed nonsequentially, an identifier must be specified that will contain the value of the relative record number. For sequential files, specification of a relative key is optional. If given, it is updated by the I/O system to reflect the relative number of the record just read (for sequential reads only) and to reflect the relative number of the record just written (for files having SEQUENTIAL access only).

The identifier specified for the relative key must represent an unsigned integer value. Its data type (usage) may be any of those available for numeric data (e.g., DISPLAY, COMP-5, COMP-6). However, as far as I/O operations are concerned, display numeric data is more efficient than any of the computational data types.

If the relative key is within the Record Description entry (an extension to the American National Standard) an observation diagnostic is issued. If the relative key is contained within a file's record area, in the cases where the I/O system maintains the value, the key is updated only _after_ the I/O operation is successfully completed. Thus, in the case of a WRITE, the record is written with the current value of the relative key in the record, not the relative number of the record, which will not be placed in the record until after the WRITE. In the case of a READ, the key in the record area would be updated after the transfer of data, overlaying the actual contents of that field in the record just read. These two cases compensate for each other and require no special concern.

I/O SWITCH ASSIGNMENT

Various language elements allow the user to establish and identify an I/O switch to be used in referencing a particular file and control its final attachment.

EXTERNAL Attribute

If the optional key word EXTERNAL is specified, the scope of the file is considered external; otherwise, the file is internal to the program. This option affects the interpretation of the internal-file-name (discussed below).

Internal-file-name

The name given for <internal-file-name> concerns only external files. Otherwise, it has no meaning, although a name must be specified. Often, in this case, <internal-file-name> is made identical to <filename>, the name used to refer to the file internally throughout the COBOL source program.

For external files and preattached internal files (see below), the internal-file-name identifies the name of the I/O switch through which the file is to be referenced at run-time. Two programs required to share a file on the I/O switch level need only to specify identical internal-file-names. The value of <filename> does not have to match.

The additional descriptive information for the file in both programs must be consistent and compatible to avoid run-time errors.

As an example, suppose program A contains the following code:

```
INPUT-OUTPUT SECTION.
FILE CONTROL.
     SELECT file-a ASSIGN TO file_a.
     SELECT EXTERNAL file-b ASSIGN TO file_b.
     SELECT EXTERNAL file-c ASSIGN TO sw_name.
```

and program B contains:

```
     SELECT file-a ASSIGN TO file_a-PREATTACHED.
     SELECT EXTERNAL file-b ASSIGN TO FILE_B.
     SELECT EXTERNAL cfile ASSIGN TO sw_name.
```

If programs A and B were active in the same run-unit, the only file shared on the switch level would be file-c and cfile, both referenced through the I/O switch sw_name. The internal-file-name can contain any character in the Multics character set except the space and hyphen. Uppercase and lowercase letters maintain their identity. Thus, no match occurs between the switches file_b and FILE_B. The internal-file-name must be no more than 16 characters long.

Since all files are referenced through an I/O switch, unique switches are created for internal files to avoid inadvertent matching. This is done by appending unique characters to the specified internal-file-name. Thus, in the above example, if all files were opened, the following I/O switches would exist:

```
file_a.!BBBJFDkwbzbmNn        (unpredictable)
file_a
file_b
FILE_B
sw_name
```

Device Specification

As shown in the SELECT clause format above, the internal-file-name may contain a device suffix to describe the nature of the file and to indicate how the I/O switch is to be attached when the file is opened. The possible values of <device> supported are:

```
VIRTUAL
PRINTER
CARD-PUNCH
CARD-READER
TAPE
PREATTACHED
```

The default is VIRTUAL if no device suffix is given. This causes the vfile_ I/O module to be used to reference a virtual memory storage system file. This may be either a segment or multisegment file.

PRINTER and CARD-PUNCH may be given only for SEQUENTIAL and STREAM files. PRINTER indicates that a printable ASCII file is to be created and allows the use of the ADVANCING clause in the WRITE statement. CARD-PUNCH indicates that a file of card-images is to be produced. In either case, the vfile_ I/O module is again used to create an unstructured file in the Multics storage system acceptable to the dprint and dpunch commands. An unstructured file is the type of file produced using the STREAM organization. It is not the same as a Multics sequential file, which is not printable. The device suffix in this case actually alters the resulting structure of the file. Although CARD-PUNCH and PRINTER indicate that the final destination of the output file is to a punch or printer, the user must use the dpunch or dprint command to queue these files.

CARD-READER indicates that a file of card-images is to be read. Whereas the physical reading of cards is a system function, the compiler treats all action against this file type as though it were defined as VIRTUAL.

TAPE may be specified only for SEQUENTIAL files to indicate that a file is contained on tape. If the <qualifier> in the ORGANIZATION clause is ANSI or omitted, attachment is made to the tape_ansi_ I/O module. If the <qualifier> is IBM-OS or IBM-DOS, attachment is made once again to the tape_ibm_ I/O module. For complete details on tape attachment and proper usage, see the APPLY statement in the COBOL Reference Manual and the tape_ansi_ and tape_ibm_ I/O modules in the MPM Subroutines manual.

PREATTACHED is used to indicate that no attachment is to take place when the file is opened. The associated I/O switch must have been attached prior to execution of the program in question; if not, a run-time error occurs. Thus, the user is able to ensure that the switch has been attached and to avoid generating code to make the attachment.

This page has been deleted with Addendum A.

The user may name one or two fields into which values are to be moved after the execution of every statement that references either explicitly or implicitly an associated file. The first of these is defined as a 2-digit variable in which the leftmost digit is referred to as Status Key 1 and the rightmost as Status Key 2. A detailed description of the usage of this field and possible values are given in Section IX of the Multics COBOL Reference Manual. The second field is a Multics language extension provided to give the user more control over error handling. This is referred to as Status Key 3 and gives more detailed information than Status Keys 1 and 2. It is in the form of a 4-digit numeric value in which each digit and grouping of digits have a specified meaning. If a data field longer than 4 digits is specified, zero-fill occurs on the left; if it is shorter, left truncation occurs (as though a 4-digit numeric were moved to the specified data field). Status Key 3 is designed so that the more critical information is in the rightmost digits so that left truncation can be used effectively to limit the scope of comparisons made to it. Its format is as follows:

wxyz

where: wx describes the operation being performed
yz describes the cause of the error

specifically:

w = the COBOL I/O statement in which the error occurred

```
1        OPEN
2        CLOSE
3        READ
4        WRITE
5        REWRITE
6        START
7        DELETE
8,9,0    undefined
```

x = the Multics I/O system subroutine that discovered the error

```
0        none
1        iox_$find_iocb
2        iox_$attach or iox_$open
3        iox_$detach or iox_$close
4        iox_$read_record, iox_$write_record,
         iox_$rewrite_record, or iox_$delete_record
5        iox_$get_line, iox_$get_chars, or
         iox_$put_chars
6        iox_$seek_key or iox_$position
7        iox_$control or iox_$modes
8        iox_$read_key or iox_$read_length
9        other
```

y = the general category of the cause of the error (similar to the COBOL-defined STATUS-KEY-1)

```
0        successful completion
1        at end of file
2        invalid key
3        permanent error
9        unable to make file available
```

z = specific cause of the error. This depends on the value of y. Legitimate combinations of yz are:

00	no error
01	short record
10	at end of file
21	invalid key - sequence error
22	invalid key - duplicate
23	invalid key - record not found
24	invalid key - new key ^= old key
30	unspecified error
31	file not open
32	invalid operation for current open mode
33	previous I/O operation was not READ
34	new record length ^= old record length
35	long record
36	file already open or already closed
90	cannot make file available
91	file is busy
92	format error in file
93	cannot attach or detach the I/O switch
94	attach and open modes are incompatible
95	file does not exist
97	label error

All possible values of Status Key 3 that are returned by a Multics COBOL object program along with the corresponding values of Status Keys 1 and 2 and a brief description of the specific error are shown in Section IX of the Multics COBOL Reference Manual.

Status keys should be included only if they are to be used by the program. The time taken for their setting can be costly in a program with heavy I/O usage. Use them only for programmable reactions to anticipated situations. Provisions exist for interactive decision making in regard to error corrections, program réstarting, and debugging on the system level. These are discussed in detail in Section VI.

Supplementary Options - APPLY Clause

The I-O-CONTROL paragraph of the Environment Division allows the user further control over the definition of his files by specifying one or more APPLY clauses. The format is:

APPLY <io-technique> ON <filename>...

The io-technique variable specified in the APPLY clause of the I-O-CONTROL paragraph can consist of the following options.

```
[ FILE IS { TEMPORARY } ]
          { PERMANENT }

[ NO DETACH AT CLOSE ]

[ ATTACH-OPTIONS ARE { literal-1     } ]
                     { data-name-1   }

[                     { OUTPUT-MODE IS { GENERATION                                } }      ]
[                     {                { MODIFICATION                              } }      ]
[                     {                { REPLACEMENT { literal-2     }             } }      ]
[                     {                {             { data-name-2   }             } }      ]
[                     {                                                              }      ]
[                     { DEVICE IS { integer-1   }                                    }      ]
[                     {           { data-name-3 }                                    }      ]
[                     {                                                              }      ]
[ TAPE-OPTIONS ARE    { DENSITY IS { 800  }                                          }...  ]
[                     {            { 1600 }                                          }      ]
[                     { RETAIN                                                       }      ]
[                     { FORCE                                                        }      ]
[                     { PROTECT                                                      }      ]
[                     { ADDITIONAL { CATALOG-NAME IS   } { literal-3   }...          }      ]
[                     {            { CATALOG-NAMES ARE } { data-name-4 }             }      ]
```

TEMPORARY FILES

The FILE IS TEMPORARY io-technique is normally used for work files that are not to be retained after completion of the run-unit. Use of this option overrides any previous specification for the VALUE of CATALOG-NAME IS entry in the file description. For virtual memory files, a catalog-name is instead developed at run-time as follows:

<process directory>><unique characters>.<progid>

where <process directory> is the name of the user's process directory at execution time (as returned by the [pd] active function) and <unique characters> is a 15-character string generated at compile-time and guaranteed unique.

In the following program:

```
PROGRAM-ID. progid.
.
SELECT fname ASSIGN TO sname.
.
APPLY TEMPORARY ON fname.
.
```

when the file fname is opened, the attach description of the I/O switch through which it is referenced would be something like:

```
"vfile_ >pdd>!BGMBGpwBBBBBBB>progid.sname"
```

All such files allocated in the process directory will be automatically deleted when the process terminates. Although the user does not "pay" for records used in this directory, its quota is nonetheless finite. The actual allowable quota is installation dependent, but care must be taken not to use an excessive number of records in these temporary files so as to avoid record quota overflow in the process directory. They can be cleaned up between run-units with the command:

```
delete [pd]>*.progid
```

where progid is the name of the program defining such files. See "Run-Unit Definition."

Temporary files defined with the PRINTER and CARD-PUNCH device suffix are an exception to the above rules and are allocated in the working directory. Such files must not reside in the process directory because it is possible the actual printing or punching of such files will not be completed until after the user's process terminates. If the user desires to make such printer and card punch files "temporary" in that they are deleted after being printed or punched, the user should invoke the dprint command with the -dl control argument. For example:

```
dprint -dl sname.progid
```

The FILE IS PERMANENT io-technique is provided only for documentation.

ATTACHMENT CONTROL

The activities performed by the CLOSE statement for an external file can be specialized by specification of the NO DETACH AT CLOSE io-technique in the APPLY clause.

Consider the program:

```
PROGRAM-ID. myprog.
.
SELECT EXTERNAL fname ASSIGN TO sname.
.
     OPEN OUTPUT fname.
.
     CLOSE fname.
```

The I/O switch sname is normally detached upon execution of the CLOSE statement. However, the attachment can be maintained until the run-unit terminates, if the following statement is included:

```
APPLY NO DETACH ON fname
```

If sname is already attached when the OPEN statement is executed, the implied attachment "vfile_ myprog.sname" is ignored, and the current attachment is used instead.

EXPLICIT ATTACH SPECIFICATION

The entire attach description can be defined explicitly by using the ATTACH-OPTIONS io-technique. When this is used, it overrides all other implicit attach information given in the SELECT clause. For example, in the program:

```
PROGRAM-ID. myprog.
.
SELECT fname ASSIGN TO sname.
.
APPLY ATTACH-OPTIONS "vfile_ <segname" ON fname.
```

the implied attachment "vfile_ myprog.sname" is overridden and becomes instead "vfile_ <segname". That is, the virtual file named segname in the directory containing the current working directory is attached through the vfile_ I/O module. The same result could have been achieved by including:

```
VALUE of CATALOG-NAME IS "<segname"
```

in the File Description entry.

The following excerpt from a COBOL program shows other possibilities.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. progid.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
   SELECT EXTERNAL file-1 ASSIGN TO switch_1-VIRTUAL.
   SELECT EXTERNAL file-2 ASSIGN TO switch_2.
   SELECT EXTERNAL file-3 ASSIGN TO switch_3
     ORGANIZATION IS STREAM.
I-O-CONTROL.
   APPLY ATTACH-OPTIONS "discard_" ON file-2
   APPLY TEMPORARY ON file-2
   APPLY ATTACH-OPTIONS "syn_ user_output" ON file-3
   APPLY NO DETACH ON file-3.
FD file-1 VALUE OF CATALOG-NAME IS "seg-1"
.
PROCEDURE DIVISION.
.
      OPEN EXTEND file-1.
      OPEN OUTPUT file-2, file-3.
```

When file-1 is opened, a check is made to see if switch_1 is attached. If so, that attachment is used, and the file is opened. Otherwise, attachment is made to the vfile_ I/O module with the attach description "vfile_ seg_1 -extend." Likewise, for file-2, the destination of the output directed via WRITE statements depends on the previous attachment of switch_2. If it had been attached by use of the io_call command or by an OPEN statement in another program, then the output is directed as specified in that attachment. Otherwise, no output is produced (via attachment to the discard_ I/O module). Applying the TEMPORARY io-technique to file-2 has no effect (other than for documentation), since it affects the catalog-name used, which is overridden by the explicit attach description. When file-3 is opened, switch-3 is attached (again, if not already attached) as a synonym for the system-defined user_output I/O switch, normally opened for stream-output and attached to the user's terminal. This is accomplished by the syn_ I/O module. This attachment has the effect of directing all output for file-3 to the terminal (or to an unstructured file, if the file_output command is in effect). This makes WRITE statements to file-3 equivalent to DISPLAY statements.

To check out a program that takes input from an unstructured file (call it external "file-in" with switch "switch_in") and produces output to another (external "file-out" with switch "switch_out"), attach the two switches before running as follows:

```
io_call attach switch_in syn_ user_input
io_call attach switch_out syn_ user_output
```

When the program executes, the input normally read from file-in is accepted from the terminal, allowing for spontaneous creation of test data. Likewise, the output normally written to file-out is displayed upon the terminal for immediate inspection. When the program is run again (in a different run-unit) without use of the io_call command beforehand, the attachments for switch_in and switch_out as specified or implied in the program are established.

TAPE ATTACHMENT SPECIALIZATION

Attachment to tape_ansi_ and tape_ibm_ I/O modules can be specialized by the TAPE-OPTIONS I/O techniques. Using these techniques, the user can specify values of various control arguments to the associated I/O modules. However, when the normal defaults are sufficient to do the job the following I/O attachments are generated.

If LABEL RECORDS within the FD clause are specified as OMITTED, the tape is assumed to be nonstandard; attachment is made to the tape_ibm_ I/O module with the following conditions set:

-fmt u	for undefined records. U format records are undefined in format. Each block is treated as a single record, and a block may contain a maximum of 8192 characters.
-rg	insert a ring only when file is opened for output or extend
-den 800	specifies that density be set to 800 bits per inch
-nb 1	specifies the file sequence number be set to 1
-bk	specifies that the block length be set to the maximum size of the current record area specified in the FD section
-nlb	specifies that unlabeled tapes are to be processed
-mode ASCII	specifies the encoding mode used to record the file data
-dv 1	only 1 tape drive is used during an attachment

These options can all be explicitly controlled by the TAPE-OPTIONS I/O technique.

If LABEL RECORDS within the FD clause are specified as STANDARD, the tape is assumed to be ANSI standard; attachment is made to the tape_ansi_ I/O module with the following conditions set:

-fmt fb	for fixed-length records, blocked. Used when every record has the same length, not to exceed 8192 characters
-rg	insert a ring only when file is opened for output or extend
-den 800	specifies that density be set to 800 bits per inch
-nb 1	specifies the file sequence number be set to 1
-bk	specifies that the block length be set to the maximum size of the current record area specified in the FD section
-nlb	specifies that unlabeled tapes are to be processed
-dv 1	only 1 tape device is used during an attachment

Record Description - FD Entry

The records of a file are described in the File Description (FD) entry for that file in the Data Division. The FD entry also provides information about a file's physical structure and labels, but such data is accepted only for documentation on Multics. For example, the BLOCK CONTAINS and VALUE OF FILE-ID clauses are significant only for tape files; they have no meaning for virtual files. The general format of the FD entry is shown below.

```
FD  filename

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

  [ ; RECORD CONTAINS [integer-3 TO] integer-4 CHARACTERS

        [DEPENDING ON data-name-1] ]

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

 [             [ | FILE-ID IS        { data-name-2 } | ] ]
 [             [ |                   { literal-1   } | ] ]
 [ VALUE OF  { [ | RETENTION IS      { data-name-3 } | ] } ]
 [             [ |                   { literal-2   } | ] ]
 [             [ | CATALOG-NAME IS   { data-name-4 } | ] ]
 [             [ |                   { literal-3   } | ] ]

  [ ; DATA { RECORD IS   } { data-name-5 } ... ]
           { RECORDS ARE }

  [ ; LINAGE IS { data-name-6 } LINES [ , WITH FOOTING AT { data-name-7 } ]
                { integer-5   }                           { integer-6   }

    [ , LINES AT TOP { data-name-8 } ] [ , LINES AT BOTTOM { data-name-9 } ] ]
                     { integer-7   }                       { integer-8   }
  [ ; CODE-SET IS alphabet-name ]
```

This format shows all syntactical elements, though not all are discussed below. A full description of the FD entry syntax is presented in Section IX of the <u>Multics COBOL Reference Manual</u>.

In the preceding format, <filename> is the name of a file defined previously in a SELECT clause (see "File Selection - SELECT Clause" above). For virtual files, the LABEL phrase is used only for documentation with STANDARD considered equivalent to OMITTED. However, it must be specified due to American National Standard requirements. For tape files, if labels are omitted, it indicates attachment to the tape_ibm_ I/O module. Standard indicates attachment to either tape_ansi_ or tape_ibm_ I/O modules, depending on the organization <qualifier> in the SELECT clause. The RECORD CONTAINS clause is used only for documentation unless the DEPENDING ON clause is used. In that case, the specified variable is set by the system each time a record is read and is used each time a record is written, overriding all other record length determining factors. No checking is done either after a READ or before a WRITE to ensure that this variable contains a value within the specified bounds.

VALUE OF CATALOG-NAME IS CLAUSE

Once the I/O module to be used in attaching the I/O switch is determined, additional information must be established for completing the attach description. The nature of this data depends on the particular module, but some form of external identification is necessary to attach to the specific file. This is the function of the data-name or alphanumeric literal specified within the VALUE OF CATALOG-NAME clause.

VIRTUAL MEMORY FILES

For virtual memory files, the vfile_ I/O module is used. This requires the pathname of a segment or multisegment file in the virtual memory storage system. In this case, the catalog-name represents the full or relative pathname of the file in question.

It is generally considered bad practice to include full pathnames in source programs. If a relative pathname is given, it is expanded at run-time. Thus, the statements:

```
SELECT fname ASSIGN TO sname-VIRTUAL.
.
.
.
FD fname VALUE OF CATALOG-NAME IS "xyz"
```

causes attachment to the file named "xyz" in whatever the working directory is when the program is being executed (or more precisely, when the OPEN statement for that file is executed). In this way, the same program is made to reference any number of different files, each with the entry name "xyz".

Also, the catalog-name can be given in a variable. In this case, the contents of that field at the time the file is opened establishes the name of the file in the attach description. For example:

```
SELECT fname ASSIGN TO sname-VIRTUAL.
FD fname VALUE OF CATALOG-NAME IS segname
.
WORKING-STORAGE SECTION.
01 segname pic x(32).
.
     DISPLAY "Enter segment name:".
     ACCEPT segname.
     OPEN fname.
```

In this way, the program can reference any file in the storage system that is accessible to the user. The user can wait until the program is executing to decide.

A run-time error occurs if the attachment cannot be made or the file cannot be opened by using that attachment. If the catalog-name literal or variable contains an invalid sequence of characters that cannot be interpreted as a pathname (for example, "><" or all blanks), the attachment is not made. The attachment is made if the catalog-name specifies a nonexistent file. In this case, an OPEN with OUTPUT mode causes creation of such a file (assuming, of course, correct access in the directory). An OPEN with INPUT mode causes an error, since the file is not found. The attachment has been made at the time of the error. Refer to Section VI, "Error Processing and Debugging."

If no VALUE OF CATALOG-NAME clause is specified in the File Description entry, a default name is established as follows: the PROGRAM-ID given in the Identification Division is concatenated with a period that is concatenated with the internal-file-name of the SELECT clause. It is as though:

```
VALUE OF CATALOG-NAME IS "<progid>.<internal_file_name>"
```

had been coded in the File Description entry. This ensures that no inadvertent references are made to the same segment for files whose catalog-name was established by default. For example, if the following program were run:

```
PROGRAM-ID. myprog.
.
SELECT print-file ASSIGN TO Report-PRINTER.
.
      OPEN OUTPUT print-file.
      WRITE print-file.
      CLOSE print-file.
```

a printable ASCII segment is created in the working directory with the name "myprog.Report".

TAPE FILES

For tape files, the catalog-name represents the physical tape identification (default is "work"), which is limited to six characters. For labelled tape files, if the catalog-name is less than six characters and is entirely numeric, the I/O module pads high-order positions with zeros. If it is less than six characters but is not entirely numeric, low-order positions are padded with blanks.

Variable-Length Records

Variable-length records are written if any of the following conditions hold:

1. The key word VLR is specified in the SELECT clause.
2. More than one <data description> is given and more than one size record is defined. (The definition of <data description> is given in Sections VI and IX of the Multics COBOL Reference Manual.)
3. The expression <data description> indicates a variable length record (contains an OCCURS ... DEPENDING ON clause).

4. The FD entry indicates a variable length record (contains a RECORD CONTAINS ... DEPENDING ON clause).

If none of these conditions is true, fixed-length records are written.

For files having fixed-length records, the maximum record size is used for each WRITE. Any padding required is the responsibility of the user. If any portion of the record area is unused, its previous contents are written rather than some special padding characters. This can only happen if FLR has been specified in the SELECT clause _and_ either condition 2 or 3 above is true.

For variable-length records, the following determination of record length is made upon each WRITE. First of all, if RECORD CONTAINS ... DEPENDING ON is specified, the number contained in the associated variable at the time of the WRITE is used. Otherwise, if the record named in the WRITE statement is variable (an OCCURS ... DEPENDING ON specified in its <data description>), its current length is determined by evaluating the variable associated with the OCCURS clause. In this case, it is ensured that this variable is not outside the range of the bounds given with the specified array. This variable is _not_ automatically set as the result of a READ as is the variable named in the RECORD CONTAINS clause. If neither RECORD CONTAINS nor OCCURS variables have been specified, the length of the fixed-length record named in the WRITE statement is used.

The user can take advantage of both the variable RECORD CONTAINS and OCCURS clauses to process the type of variable-length records that compose Multics unstructured files. For example, consider the following program:

```
DATA DIVISION.
FILE SECTION.
FD fname,
   RECORD CONTAINS 0 to 256 CHARACTERS DEPENDING ON reclen,
   LABEL RECORDS ARE OMITTED.
   01 rec.
    02 char PIC X OCCURS 0 TO 256 TIMES DEPENDING ON reclen.
.
WORKING-STORAGE SECTION.
01 reclen PIC 999.
```

If the 1-character array "char" is set up so that the number of occurrences corresponds to the variable set by the READ statement, any reference to "rec" after performing a READ references a string of characters with length equal to the number of characters just read. Any WRITE to "fname" causes only as many characters as indicated by the value of "reclen" to be transferred from "rec".

DECLARATIVE PROCEDURES

The declaratives of a COBOL program are a set of one or more special purpose sections written at the beginning of the Procedure Division. The USE statement serves to identify the particular use of a declarative section as follows:

```
                                                        { INPUT            }
                      { EXCEPTION }                     { OUTPUT           }
USE AFTER STANDARD    {           }  PROCEDURE ON       { I-O              }
                      { ERROR     }                     { EXTEND           }
                                                        { <filename> ...   }
```

The section that follows this statement defines the actions to be taken when an I/O error occurs on a file opened in the specified mode or on the specific file <filename>.

An I/O error can be caused by a logic error in the program (e.g., INVALID KEY), an unexpected condition (e.g., incorrect access on a file), or an expected condition (e.g., AT END). Hardware errors, as such, cannot occur in referencing the virtual memory storage system, at least at a level visible to the COBOL I/O. After the occurrence of an I/O error, the user can:

1. Check a status code for an expected condition and alter data to affect the continuation of the program

2. Ignore or record the error (by a message to the terminal, for example) and continue processing

3. Perform clean-up functions and terminate execution of the program

Errors may be easily and descriptively reported by including a CALL to the print_cobol_error_ system subroutine in the declarative procedure. This writes either to the terminal or to a specified I/O switch a message that specifies the cause of the error, the line number of the source statement causing it, and additional information. Refer to Section VI, "Error Processing and Debugging" for full details on the use of this subroutine.

Continuation of program execution is accomplished by the execution of an EXIT statement in the declarative procedure or by having control reach a point where the next section begins. In all cases, execution resumes at the statement immediately following the I/O statement that caused the error, even in the case of I/O operations that involve more than one step, such as OPEN and CLOSE. Regardless of which step caused the error (e.g., attach, open), no more actions involving the operation are performed once the associated declarative section is executed.

Situations may arise where more than one declarative applies to the same file. An example is if:

```
USE AFTER STANDARD ERROR PROCEDURE ON INPUT.
.
USE AFTER STANDARD ERROR PROCEDURE ON file-a.
.
```

is specified in a program and an error occurs when referencing file-a while it is open in the INPUT mode, only the more specific declarative procedure is executed; the more general procedure is never executed. In the above case, control is passed to the statement following the second USE statement shown.

Certain I/O conditions are considered warnings and do not cause execution of an associated declarative procedure. These are identified by a value of "0" in Status Key 1 (see "File Status" above). The user must check for such warnings by including a test of one of the status keys after the applicable I/O statement.

PRINT FILES

Print files are identified by the following characteristics:

1. ORGANIZATION is SEQUENTIAL.
2. ACCESS MODE is SEQUENTIAL.
3. The device suffix is PRINTER (e.g., SELECT a ASSIGN TO a-PRINTER).
4. All OPEN statements to that file must be with OUTPUT mode.

The use of channel-m (m= 1 to 16) mnemonic-name causes slewing to the particular channel. This results in the insertion of the characters 1 through 16. For example, if CHANNEL-12, then the following insertion occurs:

```
(ESC)           (ETX)
\033 \061\062   \003
```

or, if CHANNEL-6, then insertion occurs as:

```
(ESC)           (ETX)
\033 \060\066   \003
```

The interpretation of these characters is site dependent.

Print files are treated as Multics unstructured files with each record delimited by the newline character. Records are, by definition, of variable length. This file structure is accepted by the print command and the dprint command. The latter also interprets the contents of the unstructured file in producing the printed listing. Backspaces, newlines, and form-feeds are translated to provide meaningful output. If the position of the newline character indicates a record (i.e., line) is shorter than the standard print line for the printer being used, it is padded to the right with blanks. If the print line is longer than the record, that record is split over as many physical lines as necessary to contain all the information.

A print file is similar to a file defined with STREAM organization with one exception: if not otherwise specified (by the ADVANCING clause), the newline character is placed immediately before the contents of the record area, not after, as in STREAM files. This is due to the COBOL-defined rules concerning print files and the interpretation of the ADVANCING clause.

Page and Line Control

In print files, it is often desirable to control the page and line spacing. COBOL provides the user with the ADVANCING clause of the WRITE statement for this purpose. This can be used only when writing print files. The format is as follows:

```
WRITE <record name>  [FROM <identifier-1>]

 [                          { { <identifier-2> } [ LINE  ] } ]
 [ { BEFORE }               { { <integer>      } [ LINES ] } ]
 [ { AFTER  }  ADVANCING    {                                } ]
 [                          { { <mnemonic-name> }            } ]
 [                          { { PAGE            }            } ]
```

The print file is written as a stream of characters consisting of records separated by the newline characters (octal 012) and possibly carriage-return character (octal 015) and form-feed character (octal 014). The ADVANCING ... LINES option merely controls the number of newline characters (octal 012) placed before or after the contents of the record area. If the ADVANCING clause is not specified with a WRITE statement to a print file, then, according to COBOL rules, it is assumed that:

```
... AFTER ADVANCING 1 LINE
```

was requested. The ADVANCING ... PAGE and HOF (mnemonic-name) options cause the insertion of a form-feed character (octal 014) before or after the contents of the record area. This character has no effect on most terminals; however, it causes a page eject on a listing produced on a line printer by a dprint command.

If an identifier is used with the ADVANCING clause, the possibility of a run-time error exists. The value of the identifier can never be less than zero, because it must be defined as an unsigned integer. However, an upper limit of 120 is enforced. If, at the time of the WRITE, the identifier contains a value greater than this, an error occurs, and a message such as the following is issued:

```
progid: The value of a data-name used with the ADVANCING
   clause is inordinately large
Error occurred at 476¦1327 in >udd>PROJ>user>progid on line 259
system handler for error returns to command level
```

As with most run-time errors, corrective action may be taken (e.g., the identifier could be set to a reasonable value by the Multics debugging facility) and execution resumed with no ill effects. See Section VI for full details on handling run-time errors.

FILE OPENING MODES

In COBOL, a file can be opened in one of four modes: INPUT, OUTPUT, I-O, and EXTEND. (Relative and indexed files cannot be opened with EXTEND; stream files cannot be opened with I-O.) The combination of organization, access mode, and open mode map into one of the Multics I/O system opening modes is shown in Table 4-3.

Reference to the "extend bit" in Table 4-3 indicates an opening parameter currently supported by the vfile_ I/O module. In the future, this support will be eliminated. Since extension is also an attachment argument, a conflict in interpretation may occur. When COBOL files are opened in EXTEND mode, both the attachment and the opening indicate the file is to be extended. The attach description becomes "vfile_ filename -extend" and the extend bit passed as a parameter in the iox_$open calling sequence is set to "1"b. When support disappears for opening level extend specification and the value of this bit is ignored, the user must ensure that any attachments made outside the program for external files opened in EXTEND mode include an -extend control argument in the attach description. There will be no difference for a file opened in EXTEND mode than for a file opened in OUTPUT mode except on the attach level. An external file opened in both OUTPUT and EXTEND modes in the same program (or run-unit) cannot share the same attach description.

IMPLEMENTATION SPECIFICS

The following information is provided for users who require more details concerning the operation of the COBOL I/O run-time system and the structures it uses to reference and keep track of files in the run-unit. Such information is necessary to debug programs on the nonsymbolic object code level. It is assumed that such users will have a more intimate familiarity with the Multics system than is necessary for understanding most other parts of this guide. The information can be bypassed by the casual reader without loss of continuity.

Table 4-3. File Opening Modes

Access Mode	Organization	Input	Output	I-O	Extend
SEQUENTIAL	SEQUENTIAL	4	5	7	6*,**
SEQUENTIAL	RELATIVE	8	9	10	NA
SEQUENTIAL	INDEXED	8	9	10	NA
SEQUENTIAL	STREAM	1	2	NA	2*
RANDOM	SEQUENTIAL	NA	NA	NA	NA
RANDOM	RELATIVE	11	12	13	NA
RANDOM	INDEXED	11	12	13	NA
RANDOM	STREAM	NA	NA	NA	NA
DYNAMIC	SEQUENTIAL	NA	NA	NA	NA
DYNAMIC	RELATIVE	8	10**	10	NA
DYNAMIC	INDEXED	8	10**	10	NA
DYNAMIC	STREAM	NA	NA	NA	NA

where:

iox_ open mode	Description
1	stream input
2	stream outout
3	stream input-output
4	sequential input
5	sequential output
6	sequential input-output
7	sequential update
8	keyed sequential input
9	keyed sequential output
10	keyed sequential update
11	keyed nonsequential input
12	keyed nonsequential output
13	keyed nonsequential update

* With extend bit - "1"b

** Reads are prevented by run-time check

File State Block

Many of the mode mappings shown in Table 4-3 are not exact fits for COBOL requirements and, in many cases, additional checks and data collection must be kept in order to maintain American National Standard COBOL. For this purpose, each COBOL file has associated with it a data structure called the file state block (FSB). This contains various COBOL-related information not available in the IOCB, the data structure associated with each file by the Multics I/O system.

Each FSB is allocated in a portion of system free area via a *system link (with initialization). The entry portion of the name is formed from the name specified by the user as the internal-file-name, so that all external files having the same internal-file-name will reference the same FSB after the links have been resolved. A pointer to the file's IOCB, used in all iox_ calls, is stored in the first two words of the FSB. This pointer is available for reference by the object program at the location pr4|fsb_link_offset,*.

File Activity Recording

A record is kept of all active files for clean-up purposes. A file is active in the run-unit if a program contained in the run-unit has opened it at least one time. An active file may be in either an open or closed state. A file is active in a program if that program has opened that file at least once in the run-unit. (This terminology is used in the output provided by the display_cobol_run_unit command.) Pointers to the FSB's of all active files in a COBOL program are kept in a table, which in turn is pointed to by a pointer in a fixed location in the object program's internal static data area. This pointer is initialized null (indicating no active files) by a run-unit support routine invoked by the object program's prologue code sequence the first time the program is executed in the run-unit.

When the first file is opened in a program, space is allocated in the segment used for run-unit control in the process (cobol_control_seg_) for a table of pointers, each of which will eventually point to the FSB of an active file. When subsequent files are opened in the program this table is updated. In this way, all files active in any particular program or in the entire run-unit can be referenced at any time. Active files are closed, if they have been left open, when a program is cancelled, when the run-unit is stopped, or when the process is ended.

FILE ORGANIZATION AND STRUCTURE

Sequential Files

A sequential file is 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 in which the records are written 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. A file that is organized sequentially must be accessed sequentially.

Sequential files may be recorded in variable-length or fixed-length record form. The following operations may be performed on a sequential file:

WRITE
: accesses the space immediately following that area into which the previous logical record was written and places the contents of the specified record in that space. The file must be open for output.

READ
: accesses the next logical record on the file and makes the contents of that record available in the file record area. If no "next" record exists, an AT END condition exists. The file must be open for input or input-output.

REWRITE
: replaces the logical record accessed by the previous input-output operation (which must have been a successful READ) with the contents of the specified record. The logical record being replaced must be equal in size to the record specified in the REWRITE statement (the replacement record). REWRITE can be executed only on files that are open for input-output.

Relative Files

A relative file is organized such that each record location is uniquely identified by an integer value greater than zero which specifies ordinal position on the file. In the RELATIVE KEY phrase of the SELECT clause, the source program specifies a numeric integer data item as the relative key item. A relative file may be accessed in the sequential, random, or dynamic mode.

SEQUENTIAL MODE

The following operations may be performed on a relative file accessed in the sequential mode:

WRITE
: accesses the next record location on the file, places the contents of the specified record in that location, and places the relative record number of the record in the relative key data item. If there is no "next" record location, an INVALID KEY condition exists. The file must be open for output.

DELETE
: removes the logical record accessed by the previous input-output operation (which must have been a successful READ statement) from the file. The file must be open for input-output.

START
: positions the file such that the relative record number of the next logical record accessed is based on the comparison specified in the START statement, in relation to the contents of the relative key data item. If the comparison is not satisfied by any logical record on the file, an INVALID KEY condition exists. The file must be open for input or input-output.

READ
: accesses the next logical record on the file, makes the contents of that record available in the file record area, and places the relative record number of the record in the relative key data item. The number of record locations traversed to access the next logical record is immaterial. If there is no "next" logical record, an AT END condition exists. The file must be open for input or input-output.

REWRITE
: replaces the logical record accessed by the previous input-output operation (which must have been a successful READ statement) with the contents of the specified record. The file must be open for input-output.

RANDOM MODE

The following operations may be performed on a relative file accessed in the random mode:

WRITE
: places the contents of the specified record in the record location identified by the contents of the relative key data item. If the specified record location is outside the boundaries of the allocated file space, or if a logical record already occupies that location, an INVALID KEY condition exists. The file must be open for output or input-output.

DELETE
: removes the logical record identified by the contents of the relative key data item from the file. If the specified record location is outside the boundaries of the allocated file space, or if a logical record does not occupy that location, an INVALID KEY condition exists. The file must be open for input-output.

READ
: accesses the logical record identified by the contents of the relative key data item and makes the contents of that record available in the file record area. If the specified record location is outside the boundaries of the allocated file space, or if a logical record does not occupy that location, an INVALID KEY condition exists. The file must be open for input or input-output.

REWRITE
: replaces the logical record identified by the contents of the relative key data item with the contents of the specified record. If the specified record location is outside the boundaries of the allocated file space, or if a logical record does not occupy that record location, an INVALID KEY condition exists. The logical record being replaced must be equal in size to the record specified in the REWRITE statement (the replacement record). The file must be open for input-output.

DYNAMIC MODE

The following operations may be performed on a relative file accessed in the dynamic mode:

WRITE
: places the contents of the specified record in the record location identified by the contents of the relative key data item. If the specified record location is outside the boundaries of the allocated file space, or if a logical record already occupies that location, an INVALID KEY condition exists. The file must be open for output or input-output.

DELETE
: removes the logical record identified by the contents of the relative key data item from the file. If the specified record location is outside the boundaries of the allocated file space, or if a logical record does not occupy the specified record location,

an INVALID KEY condition exists. The file must be open for input-output.

START
: positions the file at a record location such that the next logical record accessed is based on the comparison specified in the START statement, in relation to the contents of the relative key data item. If the comparison is not satisfied by any logical record on the file, an INVALID KEY condition exists. The file must be open for input or input-output.

READ NEXT
: if the file was positioned by an OPEN or START statement, and the record location to which it was positioned is still occupied by a logical record (the logical record not having been deleted), the input-output control system makes the contents of that logical record available in the file record area and places the relative record number of the record in the relative key data item. Otherwise, the input-output control system accesses the next logical record on the file, makes the contents of that record available in the file record area, and places the relative record number of the record in the relative key data item. If there is no "next" logical record, an AT END condition exists. The file must be open for input or input-output.

READ
: accesses the logical record identified by the contents of the relative key data item and makes the contents of that record available in the file record area. If the specified record location is outside the boundaries of the allocated file space, or if a logical record does not occupy that location, an INVALID KEY condition exists. The file must be open for input or input-output.

REWRITE
: replaces the logical record identified by the contents of the relative key data item with the contents of the specified record. If the specified record location is outside the boundaries of the allocated file space, or if that record location does not contain a logical record, an INVALID KEY condition exists. The logical record being replaced must be equal in size to the record specified in the REWRITE statement (the replacement record). The file must be open for input-output.

Indexed Files

An indexed file is organized such that each record is uniquely identified by the value of a key within the record (see MPM Subroutines manual).

In the RECORD KEY phrase of the SELECT clause, the source program specifies one of the data items within one of the records associated with the file as the prime record key data item. Each attempt to access a record based on the record key item causes a search of the index file for a key that matches the current contents of the record key data item in the file record area. The matching index record in turn points to the location of the associated data record.

An indexed file may be accessed in the sequential, random, or dynamic mode.

SEQUENTIAL MODE

The following operations may be performed on an indexed file accessed in the sequential mode:

WRITE
: accesses the space immediately following that location into which the previous logical record was written and places the contents of the specified record in that space. If the contents of the record key data item are less than or equal to the key of the previously written logical record, an INVALID KEY condition exists. The file must be open for output.

DELETE
: removes the logical record accessed by the previous input-output operation (which must have been a successful READ statement) from the file. The file must be open for input-output.

START
: positions the file such that the key of the next logical record accessed is based on the comparison specified in the START statement, in relation to the contents of the record key data item. If the comparison is not satisfied by any logical record on the file, an INVALID KEY condition exists. The file must be open for input or input-output.

READ
: accesses the next logical record on the file and makes the contents of that record available in the file record area. If there is no "next" logical record, an AT END condition exists. The file must be open for input or input-output.

REWRITE
: replaces the logical record accessed by the previous input-output operation (which must have been a successful READ statement) with the contents of the specified record. If the contents of the record key data item are not equal to the key of the last record read, an INVALID KEY condition exists. The logical record being replaced must be equal in size to the record specified in the REWRITE statement (the replacement record). The file must be open for input-output.

RANDOM MODE

The following operations may be performed on an indexed file accessed in the random mode:

WRITE
: places the contents of the specified record into record space situated such that the content of the record key data item is greater than the key of the preceding logical record and less than the key of the succeeding logical record. If the contents of the record key data item are equal to the key of an existing logical record, or if writing the logical record would exceed the boundaries of the allocated file space, an INVALID KEY condition exists. The file must be open for output or input-output.

DELETE
: removes the logical record identified by the contents of the record key data item from the file. If no such logical record exists on the file, an INVALID KEY condition exists. The file must be open for input-output.

READ
: accesses the logical record identified by the contents of the record

key data item and makes the contents of that record available in the file record area. If no such logical record exists on the file, an INVALID KEY condition exists. The file must be open for input or input-output.

REWRITE
: replaces the logical record identified by the contents of the record key data item with the contents of the specified record. If no such record exists on the file, an INVALID KEY condition exists. The logical record being replaced must be equal in size to the record specified in the REWRITE statement (the replacement record). The file must be open for input-output.

DYNAMIC MODE

The following operations may be performed on an indexed file accessed in the dynamic mode:

WRITE
: places the contents of the specified record into record space situated such that the content of the record key data item is greater than the key of the preceding logical record and less than the key of the succeeding logical record. If the contents of the record key data item are equal to the key of an existing logical record, or if writing the logical record would exceed the boundaries of the allocated file space, an INVALID KEY condition exists. The file must be open for output or input-output.

DELETE
: removes the logical record identified by the contents of the record key data item from the file. If no such logical record exists on the file, an INVALID KEY condition exists. The file must be open for input-output.

START
: positions the file such that the next logical record accessed is based on the comparison specified in the START statement, in relation to the contents of the record key data item. If the comparison is not satisfied by any logical record on the file, an INVALID KEY condition exists. The file must be open for input or input-output.

READ NEXT
: if the file was positioned by an OPEN or START statement, and the logical record to which it was positioned is still accessible (not having been deleted), the input-output control system makes the contents of that logical record available in the file record area. Otherwise, the input-output control system accesses the next logical record on the file and makes the contents of that record available in the file record area. If no "next" logical record exists on the file, an AT END condition exists. The file must be open for input or input-output.

READ
: accesses the logical record identified by the contents of the record key data item and makes the contents of that logical record available in the file record area. If no such record exists on the file, an INVALID KEY condition exists. The file must be open for input or input-output.

REWRITE
: replaces the logical record identified by the contents of the record key data item with the contents of the specified record. If no such logical record exists on the file, an INVALID KEY condition exists. The logical record being replaced must be equal in size to the

record specified in the REWRITE statement (the replacement record). The file must be open for input-output.

Example

```
************* This program is a sample for using indexed I-O in Multics COBOL.
 identification division.
 program-id. testio.
 environment division.
 configuration section.
 source-computer. multics.
 object-computer. multics.
 input-output section.
 file-control.
     select print-file
     assign to printfile-printer.

     select indexed-file1
     assign to indexed1-virtual
     organization is multics indexed
     access mode is sequential
     record key is prime-key1
     alternate record key is alternate-key1 with duplicates.

     select indexed-file2
     assign to indexed2-virtual
     organization is multics indexed
     access mode is dynamic
     record key is prime-key2
     alternate record key is alternate-key2.
 data division.
 file section.
 fd indexed-file1
     label record is standard
     data record is file-record1.
 01 file-record1.
     05 description-of-item   pic x(20).
     05 item-info.
          10 filler          pic x(10).
          10 prime-key1.
               15 key-number 1     pic  9(10).
               15 filler        pic 9(10).
          10 filler          pic 9(10).
          10 alternate-key1.
               15 filler        pic x(10).
               15 alt-key-number1     pic 9(10).
          10 number-in-stock  pic 9(10).

 fd indexed-file2
     label record is standard
     data record is file-record2.
 01 file-record2.
     05 prime-key2.
          10 part-number        pic 9(10).
          10 filler          pic 9(10).
     05 description pic x(20).
     05 alternate-key2.
          10 color          pic x(10).
          10 filler           pic 9(10).
          10 parts-in-stock     pic 9(10).
          10 parts-on-order     pic 9(10).
          10 order-date       pic 9(10).

 fd print-file
     label record is standard
     data record is print-record.
```

```
 01 print-record.
     05 description         pic x(20).
     05 part-number      pic 9(10).
     05 number-on-hand         pic 9(10).
 working-storage section.
 procedure division.
 begin.
     open output print-file.
     open input indexed-file1.
     open i-o indexed-file2.

 read-file.

     read indexed-file1 at end go to close-files.

     move description-of-item to description of print-record.
     move key-number1 to part-number of print-record.
     move number-in-stock to number-on-hand.

     write print-record after advancing 1 line.

     move key-number1 to part-number of file-record2.
     read indexed-file2 key is prime-key2 invalid key go to bad-key.

     move number-in-stock to parts-in-stock.

     rewrite file-record2 invalid key go to bad-key.

     go to read-file.

 bad-key.

     display "Bad key", prime-key2.

 close-files.

     close print-file.
     close indexed-file1.
     close indexed-file2.
 .
 .
 .
```

Print-file is a stream file to be dprinted. Indexed-file1 is a sequential indexed file used as input and indexed-file2 is a dynamic indexed file used as a master record to be updated.

File print-file is opened for output mode, indexed-file1 is opened for input mode to be read sequentially, and indexed-file2 is opened for I/O mode so that it can be read from, and updated, while the file is open.

Indexed-file1 is read sequentially using the end of file imperative statement. When an end of file is signalled, transfer in the program is automatically made to the close-files paragraph. Data is moved to the print record and the record is written. The prime record key of indexed-file2 is initialized and the file is read. Data is updated in the record and the record is rewritten.

In both the read and rewrite statements for indexed-file2, the invalid key phrase is used. This causes transfer of control to the paragraph bad-key if an invalid key is used.

SECTION V

EXECUTING A COBOL PROGRAM

This section deals with the run-time environment of a COBOL program. The resolution of external references with regard to Multics dynamic linking is discussed. Also, the COBOL run-unit is defined in relation to the Multics environment, and the effect of this definition on the interpretation of such statements as STOP, STOP RUN, and EXIT PROGRAM is discussed. Auxiliary Multics commands that are available for controlling and displaying information about the run-unit are shown. Other aspects of the run-time environment such as the COBOL-defined external switches and source-level segmentation control are explained. Interprogram communication is discussed, and it is shown how a COBOL program can call and be called by programs compiled by other languages on Multics. Finally, all COBOL data types are defined in relation to COBOL source-level declarations. They and the storage allocated for them are explained on an implementation level, and the methods by which data are referenced in a COBOL object program are discussed.

REFERENCING AN OBJECT SEGMENT

On Multics, object programs are referenced by a two-part symbolic name as follows:

```
<reference name>[$<entry point name>]
```

where if the optional <entry point name> is omitted, it is assumed to be identical to <reference name>. The reference name is equivalent to the segment name established by the compiler upon creation of the object segment; the name established by the compiler is the source segment name without the language suffix. This equivalency can be altered (e.g., by the initiate command), but for the scope of this discussion it is assumed.

The <entry point name> is a permanent fixture within the object segment and cannot be modified once the segment has been created. It serves to identify a particular entry point in the segment. Normally, this is the offset of the location in the segment to which control may be passed and program execution initiated. COBOL programs have only one <entry point name>, which is defined by the PROGRAM-ID paragraph of the Identification Division. Consider the program whose source is contained in the segment progname.cobol and which contains the statement:

```
PROGRAM-ID. PROGNAME.
```

The address at which execution is to begin in the resulting object segment may be symbolically referenced as:

```
progname$PROGNAME
```

Specifying progname alone would cause reference to the nonexistent entryname progname within object segment progname. Specifying PROGNAME alone would cause reference to the <entry point name> PROGNAME in the nonexistent segment PROGNAME. If the source segment containing PROGRAM-ID X is named X.cobol, a reference via the one-part name X is always properly resolved (i.e., X$X). This convention is normally adopted when source programs are written.

RESOLVING EXTERNAL REFERENCES

When a CALL statement is executed, control is transferred to the program identified by the string of characters constituting the literal or contained in the specified identifier at the time of execution. This character-string is called a reference name. It is not a pathname, nor is it necessarily a program name or segment name. Resolution of a reference name into a meaningful address within a segment where executable code exists is the subject of the following paragraphs.

Multics Environment

The address space in which an object program executes is organized as a set of segments. A segment is a linear address space beginning at 0. Each process runs in its own address space, which is established independently of other address spaces. The set of segments that constitute a process is dynamically determined by the execution of programs existing in the process. When a segment is first referenced, it becomes part of the process and is "made known" to the process or "initiated."

References to any portion of the address space consist of a segment identification and an offset. The segment identification is initially symbolic. However, when the segment identification is first used (referenced) in a process, the segment it identifies is initiated and assigned a number unique within the process. Additionally, a unique name is associated with it. The segment number and reference name are retained throughout the life of the process and identify the segment in all further references, the former absolutely and the latter symbolically. Thus, a symbolic reference in the form:

<reference name>$<entry point name>

is transformed into a segmented address usually expressed in the form:

<segment number>|<offset>

For example, 301|32.

with the OR sign (|) used to separate the segment number from the offset.

Before a segment becomes known to a process, it may be referenced only by means of a symbolic pathname that permanently identifies the segment within the directory structure of the virtual memory storage system. Each directory, itself a segment belonging to another directory (except for the root directory ">" which has no predecessor), also contains a list of other segment attributes.

Since the segment number used to reference a particular segment is process dependent, segment numbers may not appear internally in pure procedure code. For this reason, a segment is identified within a procedure segment by a symbolic reference name. Before a procedure can complete an external segment reference, the reference name must be translated into a segmented address. If the reference name is already associated with an initiated segment, the segment number has already been established and is readily available. Otherwise, it must be first translated into a pathname by means of a directory-searching algorithm and the segment located made known to the process (initiated).

Search Rules

The algorithm used in the resolution of reference names is based on a set of search rules. However, default search rules, established for the user whenever a process is created, can be modified by using the Multics commands add_search_rules and set_search_rules. Search rules can be indirectly influenced by the initiate and change_wdir commands. Default search rules are:

initiated segments (i.e., already assigned a segment number)
referencing directory
working directory
system libraries

The set of segments that have already been initiated is loosely considered a directory in the rules. This convention makes the mechanism more easily defined. Initiated segments must be used first, in which case no actual directory searching is performed. The user can alter the search rules by the above-named commands, but must always leave initiated segments as the first rule.

Assume a COBOL object segment running with default search rules set executes the statement:

```
CALL "XYZ".
```

First, it is determined whether the name XYZ has been previously referenced in the process. If it has, it will have already been associated with a segment and a segment number. This address is used in resolving the reference (i.e., the link between caller and called). Otherwise, a new segment must be made known to the process; various directories are searched to determine which segment it will be. Assuming no segment has been initiated with reference name XYZ, the next step is to search the referencing directory for a segment named XYZ. This is the directory containing the segment making the reference. If such a segment is found, it is initiated; i.e., associated with a unique segment number and the reference name XYZ. This action establishes the link between the segment making the reference and this segment. Otherwise, the working directory and then the system library directories are searched in predefined order. If such a segment cannot be found after all directories in the search rules have been exhausted, a linkage error occurs.

Dynamic Linking

Resolution of external references (or linking) is done at run-time, while the program making the reference is actually running, not beforehand in a separate step. With dynamic linking, an unresolvable reference is not discovered until it is attempted. Thus, if the COBOL program prog in the working directory >udd>PROJ>Smith executes the CALL statement:

```
CALL "XYZ"
```

and it is discovered after a search of all directories that XYZ does not exist, a message such as the following would be issued:

```
Error: Linkage error by >udd>Proj>Smith>prog|71 (line 28)
referencing XYZ|XYZ
Segment not found.
```

Corrective action can be taken and execution of the program continued. If the user has merely forgotten to compile XYZ he can perform a compilation and then continue execution incorporating the newly created XYZ into the process. If, on the other hand, the segment XYZ exists in a directory not named in the current search rules, the user can alter the search rules to include that directory:

```
add_search_rules >udd>PROJ>Jones -after working_dir
```

This serves to include the directory >udd>PROJ>Jones immediately after the user's working directory in the current search rules. Or the user could have specifically initiated that segment:

```
initiate >udd>PROJ>Jones>XYZ
```

or established a storage system link to it in a directory that is in the search rules:

```
link >udd>PROJ>Jones>XYZ
```

The initiate command initiates the segment identified by the given pathname with the reference name XYZ. The link command adds a permanent entry to the working directory, in effect making the segment XYZ in >udd>PROJ>Jones permanently referencable by pathname >udd>PROJ>Smith>XYZ. If the user misnames the source segment as xyz.cobol, the object segment produced by the compiler is xyz rather than XYZ. The user initiates the segment xyz (in the working directory) with the reference name XYZ:

```
initiate xyz XYZ
```

After any of the above alterations, the user types:

```
start
```

The start command causes execution to continue at the point of the linkage error and complete the reference.

A CALL in a program to a nonexistent program does not affect its execution so long as the unresolvable CALL is never executed. The programs composing a COBOL run-unit cannot be known until the execution of the run-unit is complete.

Binding

Dynamic linking gives the user great flexibility in determining the components of a run-unit, but resolving first-time references requires significant overhead. Once symbolic references are resolved, however, programs run efficiently, referencing each other directly through the established links.

If the components of a run-unit are fixed and known beforehand, the user can enhance the efficiency of execution by using the Multics bind command. This produces a single object program from a group of separately compiled object segments, and in so doing, eliminates all external references among the programs involved. The run-time activity necessary to resolve external references: fielding a hardware fault, searching tables and directories, possibly initiating a segment, and resolving a link is thus avoided.

PROGRAM EXECUTION FROM COMMAND LEVEL

Once an object program has been produced by the compiler and all files it references and programs it calls are accessible, it is ready to be executed. On Multics, programs are executed from command level exactly as if they were commands. For example, to execute the object program resulting from the compilation of the time-and-date.cobol source program shown in Section II, the user simply types:

```
time-and-date              (or time-and-date$time-and-date)
```

If a segment named time-and-date is in a directory named in the current search rules and it is a legitimate object segment having an entryname time-and-date, it will be initiated, start executing, and the response will be:

```
Time: 23:14:59  Date: 27/09/76
```

Upon execution of the EXIT PROGRAM statement, control is returned to command level, and the ready message is printed:

```
r 2315 0.396 0.666 12
```

Now the system is ready to accept another command. If an object program had been produced in the working directory with a name identical to the name of a Multics command, that program would be executed when the command name is typed, provided of course, the command is not already initiated. The default search rules place the working directory ahead of the system libraries.

Programs are executable by the command processor as if they were commands. Conversely, commands can be executed from within programs by using the CALL statement. For example, the statement:

```
CALL "dcr" USING "-lg"
```

causes execution of the dcr (display_cobol_run_unit) with the -long control argument. The calling of commands is not generally suggested from within PL/I programs on Multics, as most functions provided by commands have more efficient subroutine interfaces. Additionally, these subroutines report erroneous conditions to the calling program via a status code argument rather than directly to the terminal as do commands. COBOL does not define the data types necessary to communicate with many of these subroutines; however, alphanumeric character-strings (PIC IS XXX) are supported. Since commands accept input from the terminal, it is obvious that command interfaces are restricted to this type of data. Although a COBOL program may not be able to call a subroutine, it can always call a command. For example:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 detach-arg PIC X(6) VALUE "detach".
01 switch-arg PIC X(9) VALUE "my_switch".
.
PROCEDURE DIVISION.
.
     CALL "io_call" USING detach-arg, switch-arg.
```

is equivalent to executing the command:

```
io_call detach my_switch
```

Blanks in arguments are significant because the command line is interpreted by the command processor and parsed into a number of separate character-string arguments, which are passed to the called program. For example:

```
01 segment-name PIC X(32).
.
     MOVE "XYZ.list" TO segment-name.
     CALL "delete" USING segment-name.
```

would cause the delete command to attempt deleting a segment in the working directory with the name:

```
XYZ.listbbbbbbbbbbbbbbbbbbbbbbbb
```

which probably would not exist. To avoid this problem, take advantage of COBOL variable-length strings, e.g.:

```
01 segment-name PIC X(32).
01 var-segment-name REDEFINES segment-name.
  02 char PIC X OCCURS 1 TO 32 TIMES DEPENDING ON name-len.
01 name-len PIC 99.
.
     MOVE "XYZ.list" TO segment-name.
     MOVE 8 TO name-len.
     CALL "delete" USING var-segment-name.
```

or, if the segment name was not predetermined:

```
     ACCEPT segment-name.
     INSPECT segment-name TALLYING name-len FOR ALL SPACES.
     SUBTRACT name-len FROM 32 GIVING name-len.
     CALL "delete" USING var-segment-name.
```

All commands may be considered callable programs, and all programs that accept nothing other than character-string arguments may be considered commands.

COBOL RUN-UNIT

COBOL defines a run-unit as "a set of one or more object programs that function, at object-time, as a unit to provide problem solutions." In a conventional system, the run-unit can be thought of as the output of the static linker, i.e., a fully linked and loadable set of object segments together with associated data. A run-unit starts execution after being loaded and given control by the system; when execution terminates, control returns to the system. The former is done by a JCL directive and the latter by the execution of a STOP RUN statement in any component of the run-unit. Attempting to fit this traditional view of the world into a dynamic linking environment creates certain problems and questions of interpretation.

Run-Unit Definition

In the Multics environment, the run-unit can be considered the set of COBOL object segments referenced in a process from the time a COBOL program is first executed until a STOP RUN is issued, or from the issuance of one STOP RUN to another. It is identified by the name of the first COBOL program executed in it.

A run-unit is started either explicitly by execution of the run_cobol command or implicitly by invocation from command level or from a call by another language program. A run-unit is stopped either by the execution of the STOP RUN statement in a COBOL object program or by the invocation of the stop_cobol_run command. For the duration of time after a run-unit is started and before it is stopped, it is said to be active. All COBOL object programs executed while a run-unit is active are considered part of that run-unit. A run-unit is a subset of a Multics process; it is stopped when the process is ended. Only one run-unit may be active at any given time in a process.

All data defined in the Working-Storage Section of a COBOL program is allocated and initialized, if necessary, the first time the program is executed in the run-unit.

Run-Unit Related Statements

The definition of the COBOL run-unit on Multics affects the interpretation of the following source language statements.

STOP RUN STATEMENT

The STOP RUN statement causes the current run-unit to be terminated. This affects all COBOL programs that have been executed during the life of this particular run-unit. Files referenced by these programs are cleaned up (i.e., buffers written and files closed, if necessary); action is taken to ensure that the next time any of these programs is called, its data is in initial state.

Any time the user is at command level, the Multics stop_cobol_run command can be issued to terminate an entire run-unit. This has the same effect as executing a STOP RUN statement in a COBOL program.

When a run-unit is stopped, the following message is issued through the user_output I/O switch:

<program name>: Run-unit <run-unit name> terminated at line <lineno>

where <run-unit name> is the name of the current run-unit, <program name> is equal to the <reference name> of the program executing the STOP RUN statement (or <reference name>$<entry point name> if the two are different), and <lineno> is the number of the source line containing the STOP RUN statement. If the run-unit is terminated by the stop_cobol_run command, no message is given unless an error is encountered.

EXIT PROGRAM STATEMENT

COBOL defines the function of the EXIT PROGRAM statement as follows: "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." That is to say, an EXIT PROGRAM statement in a main program causes no action and the next sequential statement is executed. In an implicitly started run-unit, every program is a called program (i.e., there is no main program); every EXIT PROGRAM statement encountered causes return of control to the caller. Since there is no distinction between invocation of a procedure as a command (by using the Multics command processor) and invocation by any other procedure (by using the CALL statement), one or more sequences of program execution can be initiated from command level, all within the same run-unit. Any COBOL program entered more than once in a given run-unit finds its data in last-used state. Data is in initial state only upon the program's first invocation within the run-unit.

If, on the other hand, a run-unit has been explicitly started as a result of a run_cobol command, the main program in which execution is to start has been named. Such a program is not considered a called program and cannot return (i.e., EXIT PROGRAM) to a caller. The only way to return to command level is by execution of a STOP RUN statement. However, if the command processor is recursively invoked (e.g., after a QUIT or an error condition), the run-unit is still in effect and any COBOL programs called from the new command level become part of the already existing run-unit.

CANCEL STATEMENT

The CANCEL statement eliminates a program from the run-unit. The actions involved in cancelling a program are a subset of the actions taken for a STOP RUN. Specifically, steps are taken to ensure that the data of the named program are in initial state upon its next invocation within the run-unit.

Any time the user is at command level, one or more programs in the current run-unit may be cancelled by issuing the Multics command cancel_cobol_program. The cancel_cobol_program command is a counterpart to a terminate command. NOTE: A COBOL program that is part of a currently active run-unit must not be terminated from the process unless it has first been cancelled. Otherwise, the run-unit will be left in an inconsistent state.

If a program is not part of the currently active run-unit, a request to cancel it with the cancel_cobol_program command results in an error message; a request to cancel it by execution of the CANCEL statement in a COBOL program causes no action, as required by the American National Standard COBOL definition of that statement.

Auxiliary Commands

Two Multics commands aid the user in controlling the state of the run-unit: the stop_cobol_run and cancel_cobol_program commands. A complete description of these commands is available in the MPM Commands manual.

run_cobol COMMAND

A run-unit can be explicitly defined and initiated by using the run_cobol command (abbreviated rc). This command starts execution in a specified main program from which control cannot be returned. It is stressed that the use of this command is in no way necessary to execute COBOL object programs on Multics. It is provided merely to simulate an environment in which more traditional COBOL concepts may be easily defined. The format of the command is as follows:

rc <program name> {-control_args}

where <program name> is the reference name or pathname of the "main program" in which execution is to be initiated, and -control_args may be one of the following:

-cobol_switch, -cs
: Sets one or more of the eight COBOL-defined external switches ON. The format is:

  -cs <switchno>...

  where <switchno> is an integer from 1 to 8 referring to a correspondingly numbered external switch to be set ON. For example, the command

  run cobol sk -cs 1 3 5

  sets external switches 1, 3, and 5

-no_stop_run, -nsr
: Avoids establishment of a handler for the stop_run condition.

-sort_dir
: Specifies the directory to be used during execution of this run-unit for temporary sort work files. If this argument is not specified, the process directory is assumed.

-sort_file_size, -sfs
: Specifies, in the floating point representation, the total amount of data to be sorted during execution of this run-unit in millions of bytes. This information is used to optimize sorting. If this argument is not specified, 1e6 is assumed (i.e., one million characters).

-debug, -db
: Sets the object time switch used by the debug facility to 'ON' for the run unit. The effect of this switch is discussed in Chapter XIII of the Multics COBOL Reference Manual, Order No. AS44.

-continue, -ctu
: Continue the execution of the COBOL program if an overflow occurs during an arithmetic statement not having an ON SIZE ERROR option.

The explicit creation of a run-unit via the run_cobol command provides for the following: (1) the establishment of a main program, (2) the setting of external switches, and (3) the ability to control the action taken for STOP RUN.

The program named by <program name> becomes the main program of the run-unit and therefore the run-unit's name. The execution of an EXIT PROGRAM statement in this program has no effect. (An implicitly started run-unit has no main program. Execution of the EXIT PROGRAM statement in all programs contained in such a run-unit always causes control to be returned to the caller, even when the caller is a system program, e.g., the command processor.)

COBOL external switches are externally visible binary switches that can be referenced by any program in the run-unit. These most closely correspond in concept to sense switches of some computers that can be set by the operator. These switches are initialized OFF at the outset of each run-unit unless otherwise specified by the -cobol_switch control argument.

The action normally taken for STOP RUN is to cancel all programs in the run-unit, closing any files left open. After this has been done, all data associated with any of the programs is no longer available. Thus, in a debugging environment, it may be useful to redefine the action taken for STOP RUN. When the run-unit is explicitly started via the run_cobol command, the STOP RUN statement causes the signaling of the stop_run condition for which a handler is established that performs the normal action described above. If the -no_stop_run control argument is specified, the handler is not established, thereby allowing the user to use his own means to handle the signal. If the user has not provided a handler for stop_run and specifies the -no_stop_run control argument, an unclaimed signal will result.

The program in which execution is to begin does not have to be a COBOL object program. It may be any program that can provide a meaningful interface with COBOL programs.

stop_cobol_run COMMAND

The stop_cobol_run command (abbreviated scr) causes termination of the current COBOL run-unit. The format is:

scr {-control_args}

where -control_args may be:

-retain_data, -rd — The data areas associated with the programs composing the run-unit are left intact for debugging purposes.

-retain_files, -rf — The files that have been left open are not closed.

This command causes the same result as executing the STOP RUN statement from within a COBOL program. Stopping the run-unit consists of: (1) cleaning up all files that have been opened by COBOL programs during execution of the current run-unit, and (2) ensuring that the next time a program that is a component of this run-unit is invoked, its data will be in initial state. Use of the -rd control argument avoids freeing the COBOL data area which has been allocated in user free storage, so that data in last used state can be displayed. However, it does not prevent the reinitialization of data when a program in the run-unit is again invoked.

cancel_cobol_program COMMAND

The cancel_cobol_program command (abbreviated ccp) causes one or more programs in the current run-unit to be cancelled. Cancelling consists of ensuring that the next time the program is invoked within the run-unit, its data is in initial state. Any files that have been opened by the program(s) and are still open are closed. Additionally, the data area associated with each program is freed. The format for this command is:

ccp <program name> {-control_args}

where <program name> represents the <reference name> of the program to be cancelled (or <reference name>$<entry point name> if the two are different) and -control_args may be:

-retain_data, -rd — The data area(s) associated with the program(s) is left intact for debugging purposes.

-retain_files, -rf — Files that have been opened by the program(s) and have not been closed are left open.

This command causes the same effect as the execution of the CANCEL statement from within a COBOL program. The only difference is that if a specified <program name> is not a component of the run-unit, an error message is issued and no action is taken; for the CANCEL statement, no warning is given in such a case.

To maintain the value of the program's data (e.g., for debugging purposes), use the -rd control argument. Data associated with the cancelled program is in its last used state; it is not restored to initial state until the next time the program is invoked in the run-unit.

display_cobol_run_unit COMMAND

The display_cobol_run_unit command (abbreviated dcr) displays the current state of a COBOL run-unit, i.e., the names of the programs that compose it. Optionally, more detailed information may be displayed concerning active files, data location, and other aspects of the run-unit. The format for this command is:

```
dcr {-control_args}
```

where -control_args may be one of the following:

-long, -lg
: Causes more detailed information concerning each COBOL program in the run-unit to be displayed.

-files
: Displays information about the current state of the files that have been referenced during execution of the current run-unit.

-all, -a
: Includes information about programs that have been cancelled during execution of this run-unit.

When no arguments are specified, the names of the programs currently active in the run-unit (i.e., not cancelled) are displayed along with the number of times each has been invoked beside the name within parentheses. For example:

```
!   display_cobol_run_unit

    Run-unit test$sk contains four COBOL programs

    test$sk        (2)
    sk             (19)
    buglog         (3)
    string-test    (1)

    All external-switches off
```

Use of the -long control argument causes such additional information to be displayed as the location of the object segment and associated data segment, the number of words of data being used, and the location of the file activity records (from which the location of the file state blocks can be found). The -all control argument includes information about programs that were once part of the run-unit, but have been cancelled. For example:

```
!   dcr -lg -a

    Run-unit test$sk contains 4 COBOL programs
    Control segment at 305|0
    1 inactive program

     1 Name: test$sk
      at 303|73
      invoked 2 times
      data at 310|0 for 52 words
      file_info at 305|100002

     2 Name: sk
      at 350|15
      invoked 19 times
      data at 351|0 for 4 words
      file_info at 77777|1

     3 Name: sk-file  (inactive)
      at 476|35
      data at 77777|1 for 52 words

     4 Name: buglog
      at 500|1343
      invoked 3 times
      data at 501|0 for 3240 words
      file_info at 305|100010

     5 Name: string-test
      at 523|561
      invoked 1 time
      data at 524|0 for 184 words
      file_info at 305|100022

      All external-switches off
```

The -files control argument gives additional information concerning active files in the programs. An active file is one that has been opened at least once by the program in question. Information is given on the location of the file state block, the current state of the file (i.e., open or closed), the name of the program that last opened or closed the file, and, if the file is open, the COBOL open mode, organization, and access mode. For example:

```
!   dcr -files

    Run-unit test$sk contains 4 COBOL programs

    test$sk         (2)
      1 file active, 2 files declared
         Internal file ifile at 311¦2
          opened by test$sk for output with indexed organization
                                       and dynamic access
    sk              (19)
      No active files

    buglog          (3)
      3 files active, 4 files declared
         External file print_iocb at 311¦726
          opened by buglog for extend with stream organization
                                       and indexed access
         External file bug_iocb at 311¦236
          closed by buglog
         External file desc_iocb at 311¦472
          closed by buglog

    string-test     (1)
      1 file active, 1 file declared
         External file testout at 311¦1162
          closed by string-test
```

Had external switches been set, the last printed line would indicate switch settings. For example, if switches 1 and 3 were set, a line would be printed as follows:

```
    External switch status:  1    2   3    4   5   6   7   8
                             ON  OFF  ON  OFF OFF OFF OFF OFF
```

ASPECTS OF THE RUN-TIME ENVIRONMENT

The following information pertains to those aspects of a COBOL program that interface with or are related to the operational environment.

STOP <literal> Statement

The action taken by the STOP statement when a literal is specified is to halt execution of the program temporarily after printing the literal on the operator's console. On Multics, the effects of this statement are to output the literal through the error_output I/O switch and recursively to reinvoke the command processor. This produces the same situation as if error condition or QUIT had occurred.

The name of the program issuing the STOP statement is displayed along with the literal in the following format:

<program name>: <literal>

After this is printed, the user may restart execution of the program at the next executable statement by invoking the Multics start command. He is also free, at this point, to invoke other commands or subroutines and then at some later time restart the stopped program. If he decides that the program issuing the STOP is not to be restarted in the future, the user should free the stack space being held by using the release command. The run-unit is still in effect at this time; that is, data is in last used state if any program is again executed. To terminate the run-unit (i.e., STOP RUN), the user must use the stop_cobol_run command.

The STOP <literal> statement is meant to be used only in an interactive environment. If a program issues a STOP literal statement in an absentee process, the absentee run will be aborted at that point.

External Switches

Inherently related to the run-unit is the implementation of the COBOL external switches. These are a set of eight binary switches that can be referenced within a COBOL program by the names SWITCH-n, where n = 1, 2, ..., 8. They are normally initialized OFF at the start of the run-unit. However, they can be set within the COBOL program and also via a control argument of the run_cobol command.

The external switches are visible to all COBOL programs in the run-unit. They are allocated in a special control segment built at run-time in the process directory. See "Implementation Specifics" later in this section.

COBOL Segmentation

COBOL source-level segmentation allows the user to divide a program into different portions based on each portion's memory requirements. With this information provided by the programmer, the compiler can then arrange to have a large program executed in a small amount of memory by proper communication with the memory management system. However, with the Multics virtual memory, as far as the user is concerned, a limitless amount of memory is always available for use. The actual memory management is performed via paging, which goes on behind the scenes and cannot be explicitly controlled at the program level.

The use of independent segments (i.e., those with specified segment numbers greater than 49) may cause unnecessary overhead when executing COBOL programs and should be avoided where possible. The effect of their use is that all alterable GO TO statements in such a segment must be reinitialized each time control enters it.

Improper Program Termination

At the end of a COBOL program, there is no automatic return of control generated as in some other languages. If control reaches the last statement of the program and that statement does not cause a transfer of control to another point in the program (GO TO) or a return of control (EXIT PROGRAM, STOP RUN), the program is considered in error. Multics COBOL provides a controlled situation for this eventuality. If a source logic error causes an attempt to execute beyond the code generated for the last statement, control is returned to the user with the message:

```
progname: Attempt to execute beyond logical end of program
Error occurred at 305|125 in >udd>PROJ>user>progname
system handler for error returns to command level
```

This is a specific example of the general run-time error handling mechanism described in Section VI of this guide.

COBOL DATA

The following information explains the meaning of the different data types as determined in a COBOL source program by the PICTURE and USAGE clauses. The allocation of the various data referenced within the COBOL program is also shown. More detailed information concerning establishing addressability is available under "Implementation Specifics" later in this section. Further information on how to display and change the different data types in an interactive debugging environment is discussed in Section VI.

Data Types

Multics COBOL supports the five categories of data defined by COBOL: alphabetic, alphanumeric, alphanumeric edited, numeric edited, and numeric. The category of a data item is determined entirely by the contents of the PICTURE clause associated with it as described in Section VI of the *Multics COBOL Reference Manual*.

Additionally, there are further sub-categories of numeric data referred to as *type*. Multics COBOL supports six data types as determined primarily by the USAGE clause. The SIGN clause and/or the presence of the operational sign character "S" in the picture string also serve as determining factors. The DEFAULT COMP and DEFAULT SIGN statements of the Control Division can be used to influence the USAGE and SIGN clauses and in this way also play a role in establishing data type.

The seven data types and their relation to the information given in the corresponding USAGE, PICTURE, and SIGN clauses are shown below in Table 5-1.

Table 5-1. COBOL Data Types

Data Type	USAGE	PICTURE	SIGN
Unsigned Display	DISPLAY	S not present	N.A.
Separate Sign Display	DISPLAY	S present	SEPARATE specified
Nonseparate Sign Display	DISPLAY	S present	SEPARATE not specified
Packed Decimal (aligned)	COMP or COMP-5	either	N.A.
Long Binary	COMP-6	N.A.	N.A.
Short Binary	COMP-7	N.A.	N.A.
Packed Decimal	COMP-8	either	N.A.
N.A. indicates that this clause is not applicable since it must not be used.			

All data is word aligned for 01 level items. Long Binary data is always word aligned, and Short Binary data is half-word aligned. All other types are byte aligned, except COMP-8, which is digit aligned.

Moving unsigned data into signed data causes a positive result. Moving signed data into unsigned data causes loss of sign; the compiler issues a warning diagnostic in this case.

UNSIGNED DISPLAY DATA

Unsigned Display data is numeric data characterized by: (1) the absence of a USAGE clause or the specification of "USAGE IS DISPLAY," and (2) the absence of the operational sign character "S" in the picture string. The number of digits it contains corresponds to the number of "9" characters in the associated picture string. Each digit occupies a 9-bit byte. Input digits are checked and carried internally as 4-bit data, the four least significant bits of the byte. They are interpreted as shown in Table 5-2.

Table 5-2. Display Data Digit Encoding

Character	Interpreted
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	causes IPR
1011	causes IPR
1100	causes IPR
1101	causes IPR
1110	causes IPR
1111	causes IPR

The contents of the five high-order bits are not checked for a sending field; in a target field they are set to 00011, thereby creating the ASCII value corresponding to the appropriate digit (e.g., 0010 -> 000110010 = "2").

If this data is used in a computation, and the least significant four bits of any digit contain a value interpreted according to Table 5-2 as "causes IPR," then an illegal procedure fault occurs and program execution is terminated.

Separate Sign Display data is numeric data characterized by: (1) the absence of the USAGE clause or the specification of "USAGE IS DISPLAY", (2) the presence of the operational sign character "S" in the picture string, and (3) the explicit specification of the optional key word SEPARATE in either the SIGN clause or the DEFAULT SIGN statement. The sign may be either leading or trailing; if not specified, it is assumed to be trailing. In either case, a separate byte is allocated for the sign.

Other than for the presence of an additional byte, this data type is identical to Unsigned Display described above. Thus, the number of bytes occupied by this data is equal to the number of "9" characters in the picture string plus one, and all bytes except that representing the sign are interpreted as shown in Table 5-1 The input sign character, either the leftmost or rightmost of the field, is checked and carried as 4-bit data, the four least significant bits. These are interpreted as shown in Table 5-3 below.

Table 5-3. Display Separate Sign Encoding

Character	Interpreted
0000	causes IPR
0001	causes IPR
0010	causes IPR
0011	causes IPR
0100	causes IPR
0101	causes IPR
0110	causes IPR
0111	causes IPR
1000	causes IPR
1001	causes IPR
1010	+
1011	+
1100	+
1101	-
1110	+
1111	+

The contents of the high-order five bits of the sign byte are not checked for a sending field. For a target field, the sign is set to 000101011 if the value is positive, 000101101 if negative. This corresponds to the ASCII value of the appropriate sign (e.g., octal 53 = "+"; octal 55 = "-").

If this data is used in a computation and the least significant four bits of a digit or of the sign contain a value interpreted according to Tables 5-2 and 5-3 as "causes IPR," an illegal procedure fault occurs and program execution is terminated.

NONSEPARATE SIGN DISPLAY DATA

Nonseparate Sign Display data is numeric data characterized by: (1) the absence of a USAGE clause or the specification of "USAGE IS DISPLAY," (2) the presence of the operational sign character "S" in the picture string, (3) the absence of a SIGN clause or, if present, the absence of the optional key word SEPARATE, and (4) the absence of the DEFAULT SIGN statement or, if present, the absence of the optional key word SEPARATE. The sign may be either leading or trailing as determined by the SIGN or DEFAULT SIGN clauses. If not explicitly specified, it is assumed to be trailing. In either case, the sign is combined with the value of the corresponding rightmost or leftmost digit to form a character interpreted as shown in Table 5-4.

Table 5-4. Display Nonseparate Sign Encoding

Digit	Sign	Octal	ASCII
0	+	173	{
1	+	101	A
2	+	102	B
3	+	103	C
4	+	104	D
5	+	105	E
6	+	106	F
7	+	107	G
8	+	110	H
9	+	111	I
0	-	175	}
1	-	112	J
2	-	113	K
3	-	114	L
4	-	115	M
5	-	116	N
6	-	117	O
7	-	120	P
8	-	121	Q
9	-	122	R

Other than for the alteration of the high-order or low-order byte to reflect the sign, this data type is identical to Unsigned Display described above. The number of bytes occupied by this data is equal to the number of "9" characters in the picture string, and all bytes except the one incorporating the sign are interpreted as shown in Table 5-2.

If this data is used in a computation and the least significant four bits of any digit not containing the sign contain a value interpreted according to Table 5-2 as "causes IPR" or the entire nine bits of the byte containing the sign do not correspond to a valid value as shown in Table 5-4, an illegal procedure fault occurs and program execution is terminated.

Nonseparate Sign Display data is not directly supported. Therefore, its use should be avoided whenever possible, because it causes a reduction in execution efficiency.

PACKED DECIMAL DATA

Multics COBOL supports two types of packed decimal data: byte aligned with trailing sign, and digit aligned with leading sign. Aligned Packed Decimal data is numeric data characterized by: (1) the specification of "USAGE IS COMP" or "USAGE IS COMPUTATIONAL" and the absence of the Control Division DEFAULT COMP statement, or (2) the specification of "USAGE IS COMP-5." It is packed four bits per digit, two digits per byte. The number of digits allocated for this type of data corresponds to the number of "9" characters in the picture string. The digit encoding is identical to that shown in Table 5-2. If the picture string contains the operational sign character "S", a trailing 4-bit sign is present. (The SIGN clause may not be given with any data whose USAGE is not DISPLAY; additionally, the DEFAULT SIGN clause does not apply to any such data.)

Aligned Packed Decimal data is byte aligned and occupies an integral number of bytes. If the total number of digits plus sign is odd, the data is right-justified in the field. The unoccupied five bits are ignored when the data is used as a sending field and left unchanged when it is used as a target field. The leftmost, unused bit of each byte is treated similarly. The input trailing sign is interpreted as shown in Table 5-3. For target fields, the sign is set to 1011 for positive values and 1101 for negative.

Unaligned Packed Decimal data is digit aligned and is characterized by "USAGE IS COMP-8." It is packed four bits per digit, two digits per byte. The number of digits allocated for this type of data corresponds to the number of "9" characters in the picture string. Digit encoding is identical to that shown in Table 5-2. If the picture string contains the operational sign character "S", a leading 4-bit sign is present. (The SIGN clause may not be given with any data whose USAGE is not DISPLAY; additionally, the DEFAULT SIGN clause does not apply to any such data.)

Unaligned Packed Decimal data is digit aligned and does not necessarily occupy an integral number of bytes. The leftmost, unused bit of each byte is ignored when the data is used as a sending field; it is left unchanged when used as a target field. The input leading sign is interpreted as shown in Table 5-3. For target fields, the sign is set to 1011 for positive values and to 1101 for negative values.

BINARY DATA

Two types of binary data are supported by Multics COBOL: Long Binary and Short Binary. They are numeric data that is characterized by the specification of "USAGE IS COMP-6" or "USAGE IS COMP-7," respectively. With this type of data, the SIGN clause must not be specified and the DEFAULT SIGN statement does not apply. Additionally, since a fixed format is dictated, the PICTURE clause must not be specified.

Long Binary data is always word aligned. It is 36 bits in length (i.e., one word) and is in the form of a two's complement binary integer. The assumed picture clause is S9(11) with the additional restriction that the value be within the range $-34,359,738,368 \leq x \leq 34,359,738,367$.

Short Binary data is always half-word aligned. It is 18 bits in length (i.e., two bytes) and is in the form of a two's complement binary integer. The assumed picture is S9(6) with the additional restriction that the value must be within the range $-131,072 \leq x \leq 131,071$.

Data Allocation

Various data is generated by the compiler and referenced by the object program. The allocation of this data is described below. More information on the establishment of addressability to the different storage locations is given later in this section under "Implementation Specifics."

Some temporary fields generated internally by the compiler are allocated in the stack frame of the running procedure. These are not normally user visible.

Literals and all data defined in the Constant Section are allocated in the text portion of the object segment, immediately before the executable code. In addition, some compiler-generated data such as a copy of the initial state of alterable GO TO, vectors of independent segments, and descriptors for expected arguments are kept here.

Some per-process data necessary for run-unit control is allocated in the linkage section of the object segment. It is referenced by the code generated for the entry sequence and is modified by the COBOL run-time support package.

Some external data that must be visible to all COBOL programs in the run-unit (e.g., External Switches) are allocated in a special control segment built at run-time in the process directory.

Some file-related data is contained in data structures called file state blocks (FSBs). Each file defined in the program has an associated FSB, which is allocated at run-time in the system free area via a *system link. Some of the data kept in the FSB is user visible. However, much serves to provide control over files and I/O operations and is primarily used to bridge the gap between the Multics I/O facility and COBOL I/O requirements.

All other data defined in a COBOL program as well as space for the record description areas are allocated in an area that is allocated at run-time in the user free area. Also allocated here are other run-unit related data such as indices, alterable GO TO vectors, and COBOL special registers. This type of data is allocated when the associated program is executed for the first time in the run-unit and is freed when the program is cancelled or when any component of the run-unit executes a STOP RUN statement. It is referred to as <u>COBOL data</u>; however, in the run-time symbol table it is described as though it were PL/I-based data for addressability by the debuggers.

A COBOL program is able to call and be called by any other program on Multics that uses the standard call/return sequences (e.g., PL/I, FORTRAN). COBOL programs may pass descriptors with arguments, but never require them for parameters. That is, a PL/I program that is to be called by a COBOL program could declare its character-string parameters with asterisk extent. For example:

```
PROGRAM-ID. cobol-prog.
.
01 arg pic xxxxx.
.
      CALL "pl1_prog" USING arg.

pl1_prog: proc(param);
dcl param char(*) parameter;
```

Any reference to param in pl1_prog references a five-character string when called by cobol-prog. Since there is no way for a COBOL program to define parameters with variable extent, calls to COBOL programs should not attempt to pass parameters with variable extent.

Data type correspondence between COBOL and PL/I is as follows. Any COBOL data of the categories alphanumeric, alphabetic, alphanumeric edited, or numeric edited corresponds to a non-varying PL/I character-string. The number of positions occupied by the COBOL data as indicated by the picture string corresponds to the number of characters in the PL/I character-string. Most simply:

```
PICTURE X(n)  ->  char(n).
```

Numeric COBOL data of Leading Separate Sign Display type is identical to fixed point decimal PL/I data as follows:

```
PICTURE S9(m)V9(n)  ->  fixed dec(m+n,n) aligned.
```

Numeric Leading Separate Sign Display COBOL data may also be described as PL/I pictured data with:

```
picture "s(m)9v(n)9".
```

Trailing Separate Sign and Unsigned Display COBOL data have no corresponding PL/I arithmetic data type.

The equivalent of the PL/I varying character string can be obtained in COBOL by use of COMP-6 (fixed binary) data together with a character string:

```
01.
 02 COMP-6.
 02 PIC X(n).
    --> char(n) varying.
```

*

COBOL signed numeric display data without separate sign has no corresponding PL/I data type. However, it may be dealt with in PL/I as character data as follows:

```
PICTURE S9(m)V9(n)  ->  char(m+n).
```

Long Binary data corresponds to PL/I word aligned fixed binary with precision 35. That is:

```
USAGE COMP-6  ->  fixed bin(35) aligned.
```

Short Binary data corresponds to PL/I fixed binary data with precision 17, which is half-word aligned. It is not generally useful to communicate with this data type between programs as it is difficult to ensure half-word alignment in the PL/I program. However, short binary can be forced to occupy the right half of the word with the left half unused by using SYNCHRONIZED; this is identical to PL/I fixed bin(17) aligned. Thus: USAGE COMP-7 SYNC --> fixed bin aligned.

COBOL packed decimal data has no corresponding PL/I data type. It may be dealt with in PL/I as bit-string data.

Aggregate Data

Structures and arrays are, by default, passed by a COBOL program as alphanumeric character-strings. This corresponds to the COBOL definition of aggregate references. For example, in the COBOL declaration:

```
01 lev1.
  02 lev2 PIC S99 SIGN LEADING SEPARATE OCCURS 5 TIMES.
```

all references to lev1 are treated as references to a field with the picture X(15). Thus, if lev1 were used in an argument list, it would be described as a 15-character alphanumeric field. Any reference to lev2 without index or subscript is invalid.

In PL/I, an aggregate reference is treated as a multiple reference to each element of the aggregate, rather than a single reference to the redefinition of the entire aggregate as a character-string. Therefore, a reference to a correspondingly defined lev1 in a PL/I program (or lev2) without index is considered a reference to all elements within the structure (or array).

If the COBOL program only calls other COBOL programs or other language programs that do not define their expected parameters with variable extents, no descriptors are required. Avoiding the generation of descriptors can increase program efficiency, especially when many calls are made with arguments passed. In this case, the user should specify:

```
GENERATE NO DESCRIPTORS
```

If a called program has been compiled with the -rck option (see "Run-Time Error Checking" in Section 3), then validation of the parameters as to data type and extent are not performed (the validation of number of parameters is still done however). If aggregate descriptors have been generated for the calling program in this case, the callee reports an argument mismatch for any aggregate data (i.e., table or structure). This is because the expected descriptors for such are assumed to be scalar i.e., character string redefinitions of the area occupied, as the COBOL language interprets them.

IMPLEMENTATION SPECIFICS

The following information is provided for users who require more details concerning the functioning of the object program and the control of the run-unit. Such information is necessary to debug programs effectively on the object code level. It is assumed that such users will have a more intimate familiarity with the Multics hardware and addressing mechanisms than is necessary for understanding most other parts of this guide. The following information can be bypassed by the casual reader without loss of continuity.

Run-Unit Control

Run-time support routines keep track of the programs that compose the run-unit and provide an interface with those commands that display or alter it: display_cobol_run_unit, stop_cobol_run, and cancel_cobol_program. This control is maintained by a data structure allocated as PL/I internal static in the linkage section of each COBOL object segment and a segment allocated in the process directory named cobol_control_seg.

The prologue code sequence of each COBOL program checks the setting of an internal static switch that has been initialized to zero at the outset of the process. If this switch is zero, it indicates that this is the first time this program has been invoked in the run-unit. In this case, the following actions are taken: (1) a transfer is made to the run-time support routine cobol_control_, (2) code is executed to initialize alterable GO TO vectors and data declared in the Working-Storage Section with the VALUE clause, and (3) the switch is incremented by one, indicating that the program is now part of the current run-unit and this initialization activity can be avoided next time the program is entered.

Because of the different interpretation of the two languages, the Multics COBOL compiler allows the user to specify how aggregate data being passed as parameters are to be described to the called program.

The user may do this via the GENERATE...DESCRIPTORS clause in the Default Section of the Control Division. (This is an extension to American National Standard for COBOL). In order to describe aggregate data to the called program as a character-string redefinition of the area occupied by the data (COBOL's interpretation), the user specifies:

```
GENERATE SCALAR DESCRIPTORS
```

or merely nothing at all since this is the default. A COBOL program can communicate with a PL/I program on this level by proper redefinition in the PL/I program. For example, if the COBOL program has the statement:

```
CALL "pl1_prog" USING lev1.
```

for proper matching the PL/I program should declare:

```
pl1_prog: procedure(arg);
declare arg character(10) parameter;
declare 1 lev1 defined(arg),
     2 lev2(5) fixed dec(2) unal;
```

If the user had specified:

```
GENERATE AGGREGATE DESCRIPTORS
```

then a more complex descriptor would be passed to the callee. A PL/I program called as above would describe the structure with each element corresponding to the COBOL definition. In the above example, this would be:

```
pl1_prog:  proc (arg);
dcl 1 arg,
    2  lev2 (5) fixed dec(2) unal;
```

Notice that although this seems more straightforward than in the preceding case, the descriptors built at run-time are substantially more complex. This may cause significant efficiency loss in passing very large, multi-dimensional arrays of structures. (This is true also in the PL/I to PL/I interface). It does, however allow additional functionality in certain cases. For example, a COBOL program could invoke a PL/I program which processed an array with variable extent only if aggregate descriptors are generated. For example, the PL/I program could have been written:

```
pl1_prog:  proc(arg);
dcl  1 arg ,
       2 lev2 (*) fixed dec(2) unal;
```

Thus any reference to lev2 in the PL/I program would be equivalent to 5 separate references to lev2(1), lev2(2), lev2(3), lev2(4), and lev2(5). The evaluation of the builtin function expressions lbound(lev2) would equal 1 and hbound(lev2) would equal 5.

In certain cases, called programs use the descriptor information to gather information about the nature of the data being passed. Such is the case with the Multics Data Base Manager's relational and procedural interfaces, the MRDS and MIDS commands respectively. For proper interaction with these subroutines, the COBOL program must generate aggregate descriptors.

The cobol_control_ run-time support program keeps a table of pointers to each of its caller's linkage sections (contained in the combined linkage segment) in the process directory segment cobol_control_seg_. This segment is also used to store COBOL External Switches and other run-unit related information. Additionally, this program allocates a data area in the user free area for each unique caller and stores a pointer to the base of that area in a known location in the caller's linkage section (internal static data area). Thus, cobol_control_ essentially provides addressability for each COBOL program to its data at the time it becomes part of the run-unit. It is clear then that cancelling a program (either by the COBOL CANCEL statement or via command) is simply a matter of resetting the switch to zero and freeing the data area (for space considerations). A STOP RUN simply cancels all programs in the run-unit and additionally truncates the control segment.

The cobol_control_ routine additionally sets up an on-unit that is executed by the standard handler for the "finish" condition that is always signaled when a process terminates. This serves to close all files that have been left open. Along with the internal static switch and data area pointer in the static data area of the linkage section, there is a file pointer that points to a list of pointers in a file state block (FSB) for each file actually opened by the program. This is used to close any open files, for the FSB contains a pointer to the I/O Control Block for the file. See Section IV for further information.

Data Addressability

Space in the stack is reserved for internally generated data of a temporary nature via the entry code sequence in the COBOL Operators segment that is executed each time the program is called. References to this type of data are made through pointer register 6, which is conventionally kept set to the base of the stack frame of the executing program.

Literals and Constant Section data are pooled and allocated before the executable code in the text section of the object segment. The actual allocation of literals and constants may be shared; that is, the literal "BCD" may occupy the same storage as "ABCDE". Since the object segment is created by the compiler with re access and this access is not normally changed thereafter, the user is afforded storage system level protection against inadvertent modification of constant data. All references to this type of data are self-relative (i.e., at a relative offset from the instruction word referencing it).

Some per-process data necessary for run-unit control is allocated in the linkage section of the object segment (see above). When the object segment is first made known to the process, the section is copied into a combined linkage segment in the process directory. Pointer register 4 is conventionally set to point to the base of this area when the program is executing. Thus, this data is referenced by the object program using a specific offset from pr4.

Each FSB is externally visible, and thus must be referenced through a link. A *system link with initialization is used. Data contained in the FSB is referenced through a pointer register set to point to the link with indirect modification.

Data allocated in the COBOL data area (i.e., Working-Storage Section data, Communication Section data, and file record areas) is referenced by an internal static pointer set during the prologue sequence (see above). Conventionally, pointer register 3 is set to the base address plus 16K words, allowing a 32K addressing range without the use of index registers. Thus, the block of virtual memory occupied by the first 32K words (131,071 bytes) of COBOL defined data is referenced through a fixed and preset pointer register thereby avoiding the need to maintain (i.e., reestablish) addressability to it during the execution of the program.

SECTION VI

ERROR PROCESSING AND DEBUGGING

This section discusses the use of the Multics symbolic debugging facilities with a COBOL program. Also, it explains the general procedure used for reporting run-time errors and shows the user alternate ways of dealing with such errors. Finally, some common causes of run-time errors are pointed out with suggested corrective and/or preventive measures.

SYMBOLIC DEBUGGING

The Multics debugger "probe" can be used to display and modify program data and to monitor execution of a COBOL program. The debugger "debug" is not recommended for use with COBOL programs, and is not discussed further here. (See MPM Commands manual.) If its use is anticipated, the program in question should be compiled with the -table control argument, so that the compiler creates a run-time symbol table and appends it to the resulting object segment. Refer to "Source Level Debugging Requirements" in Section III.

*

The user can set breakpoints in the object program at various points in the execution path at which he would like to gain control. This is useful in verifying that expected paths are taken. Also, by using debugging commands, the user can display data at these points or modify it to affect the ensuing execution of the program.

Breakpoints can be set in a COBOL program at section names, paragraph names, or line numbers. A breakpoint set at a line number causes execution to stop immediately before the first machine instruction generated for the first executable statement contained on that line of the source segment. A breakpoint set at a paragraph name is identical to one set at the following statement in the source segment. A breakpoint set at a section name is identical to one set at the first paragraph name defined in that section. In the COBOL program:

```
sec-1 SECTION.
  par-1.
      ADD 1 TO item-1. DISPLAY item-1.
      EXIT PROGRAM.
```

assume the external line number of the EXIT statement is 30 (i.e., it is contained on the thirtieth line of the source segment). The following probe commands all have the identical effect:

```
position "sec_1"; before
position "par_1"; before
before 29
position "ADD"; before
```

That is, a break is set on the first machine instruction generated for the ADD statement. The only way to set a break immediately before execution of the EXIT PROGRAM is by line number, i.e.:

```
before 30
```

The probe basic request goto (g) is not allowed from programs compiled by the COBOL compiler, nor is the probe break request after (a).

Once breaks have been established, probe is exited and the program is executed by the commands:

```
quit
.
.
.
progname
```

Displaying and Modifying Data

Data defined in the COBOL program can be displayed and modified using probe. This can be done any time after the program has started execution in the run-unit: at a breakpoint or after the program has finished executing, either due to a run-time error or execution of an EXIT PROGRAM statement. The execution of a STOP RUN statement (or the stop_cobol_run command without the -retain_data control argument) causes all program data to be no longer referable by the debuggers.

The following paragraphs describe how the various COBOL data types are displayed and modified by the debuggers. Refer to "Data Types" in Section V for a definition of these types. All data is displayed by the probe value request. While executing probe, type value followed by the data-name of interest, followed by a carriage return. Probe in turn prints the data-name, an equal sign, and the value of the data.

CHARACTER-STRING DATA (display)

All alphanumeric, alphabetic, alphanumeric edited, and numeric edited data are displayed as character-strings. This is true also of numeric data of the following type: Unsigned Display, Separate Sign Display, and Nonseparate Display. Consider the following program:

```
PROGRAM-ID. progname.
.
01 alphanum PIC XXXXX VALUE "abcde".
01 unsigned-dec PIC 999 VALUE 3.
01 struc.
  02 sepsign-dec PIC S999 SIGN IS LEADING SEPARATE VALUE -15.
  02 ovrpch-dec PIC S999 VALUE -15.
```

An aggregate data item is displayed on a component-by-component basis. For example,

```
! probe progname
! value alphanum
  alphanum = "abcde"
! value unsigned-dec
  unsigned-dec = 3
! value sepsign-dec
  sepsign-dec = -15
! value ovrpch-dec
  ovrpch-dec = -15
! value struc
  struc =
    sepsign-dec = -15
    ovrpch-dec = -15
! let sepsign-dec = 0
! value sepsign-dec
  sepsign-dec = 0
! let alphanum = "123"
! value alphanum
  alphanum = "123"
```

PACKED DECIMAL DATA (COMP, COMP-5, COMP-8)

Packed decimal data is displayed by probe as a decimal integer. If the data is positive, no sign is given. If it is negative, a minus sign appears as the leftmost or rightmost character. Consider the following program:

```
PROGRAM-ID. prog2.
.
01 struc2.
   02 a1 PIC S99 USAGE COMP-5  VALUE -2
   02 b1 PIC  99 USAGE COMP-5  VALUE  3
   02 c1 PIC 999 USAGE COMP-8  VALUE  4
   02 d1 PIC S99 USAGE COMP-8  VALUE -5
```

If these are examined with probe:

```
  !  probe prog2
  !  value a1
     -2
  !  value b1
     3
  !  value c1
     4
  !  value d1
     -5
```

Long and Short Binary data are displayed by probe as a decimal integer. If the data item is positive, no sign is given. If it is negative, a minus sign appears as the leftmost character. In the program:

```
PROGRAM-ID. prog3.
.
01 struc3.
  02 short-binary USAGE COMP-7 VALUE 3.
  02 long-binary USAGE IS COMP-6 VALUE -1.
```

the data items short-binary and long-binary represent Short Binary and Long Binary data, respectively, while struc3 represents a character-string. The following sequence may occur:

```
! probe prog3
! value short-binary
  3
! value long-binary
  -1
```

Two empty bytes exist between the allocation of sb and lb. This is because Long Binary is always word aligned.

The same convention is used to modify binary data except that an optional plus sign can immediately precede a positive value. For example:

```
! let short-binary = -3
! value short-binary
  -3
! let long-binary = 64
! value long-binary
  64
```

RUN-TIME ERRORS

Errors that occur at run-time fall into two general categories: first are those that are anticipated by the compiler and secondly those that the system reports directly. For the first type, the compiler generates code to test for an erroneous or inconsistent state of program execution or such a situation is communicated to the object program by the system via a status code. The setting of an index data item outside the range defined for the associated array or I/O errors on COBOL files are examples of such anticipated errors. Such errors as record quota overflow or an attempt by a machine instruction to reference an invalid address are termed unanticipated errors. The compiler generates no code to deal with such errors; instead they are reported directly to the user by the system.

Anticipated Errors

Most anticipated run-time errors involve some sort of I/O activity. However, the occurrence of the following situations also causes such errors.

1. The program attempts to execute beyond the end of the code generated for it. This occurs when the last statement of the source code does not cause either a return of control via a STOP RUN or EXIT PROGRAM statement or a transfer of control via a GO TO statement. Similarly, an error is caused by an attempt to execute beyond the end of a single declarative procedure.

2. An attempt is made to execute an uninitialized alterable GO TO statement that has not been set via an ALTER statement. If such a statement exists in a section defined as an independent segment, it must be altered every time control enters that section. Refer to "COBOL Segmentation" in Section V.

3. An invalid exponentiation operation is attempted in a COMPUTE statement. This includes: (1) a negative number exponentiated by a non-integer value, (2) zero exponentiated by zero, and (3) zero exponentiated by a negative number.

4. A WRITE statement is executed in which the variable specified with the ADVANCING clause contains a value greater than 120. This is the maximum number of lines that can be advanced before or after writing a line in a print file.

5. A CALL is made to a variable name and the contents of that variable cannot be interpreted as a resolvable reference name.

6. An error occurs during processing of either the SORT or MERGE statement.

7. An error occurs in referencing the I/O switches user_output, error_output, or user_input during the execution of the DISPLAY or ACCEPT statements.

8. A SEARCH statement is executed with the index data item specified in the VARYING clause uninitialized.

9. A PERFORM state has been executed with the variable specified in the BY clause equal to zero.

10. An attempt is made to set an index data item outside the allowable range as defined by the extent of the array it references when the -rck option has been specified for the compilation of the object program executing. This may occur as the result of execution of the SET or PERFORM statements.

11. A group data item that contains an item with the OCCURS ... DEPENDING ON clause is referenced when the value of the DEPENDING ON variable is outside the allowable range specified when the -rck option has been specified for the compilation of the object program executing. This may occur when the group item is referenced either as a sending or target field.

12. A program is called in which the -rck option was specified during its compilation in which the arguments passed by the calling program do not match in either number or data type the parameters expected as defined in the Linkage Section.

13. An attempt is made to reference an element of a table which does not exist when the -rck option has been specified for the compilation of the object program executing i.e., the evaluation of one or more subscripts or subscript expressions yield a value outside the range of the table.

After any anticipated error is encountered, a message is issued through the error_output I/O switch in the following format:

```
<program name>: <COBOL error message>
Error occurred at <segno>|<offset>
     [in <path> [on line <lineno>]]
```

where <program name> is the reference name of the program causing the error (or <reference name>$<entry point name> if the two are not the same) and <path> is its pathname in the storage system. The segmented address given by <segno> and <offset> is that of the instruction causing the error and <lineno> is the external line number of the corresponding source statement.

After this message is issued, the system-defined error condition is signaled. If no handler has been established for this condition, the default handler prints the message:

```
system handler for error returns to command level
```

and reinvokes the command processor. At this point, the user can take corrective action, possibly using a symbolic debugger to modify program data, and then restart the program by invoking the start command. The result of restarting program execution after any anticipated error is always a controlled attempt to retry the operation that encountered the error such that any modifications to user-defined data fields are taken into account. An attempt to restart immediately after an error occurs without any intervening corrective action always results in the recurrence of the error and again leaves the user at the above described state. If the user decides not to restart the program causing the error, he should free the stack space being held for this purpose by using the release command. The run-unit will still be in effect at this time.

For example, if the following situation occurs:

```
test: SET statement range error
Error occurred at 355|131 in >udd>P>u>test on line 256
system handler for error returns to command level
r 754 13.253 176.086 2153 level 2, 17
```

the user may proceed as follows:

```
!   edm test.cobol
!   n256
              SET index-name TO index-count.
!   t
!   l index-name
          02 xyz OCCURS 10 TIMES INDEXED BY index-name.
!   q
    r 755 1.866 80.202 931 level 2, 17

!   db
!   /test/index_count
    "11"
!   = "01"
    Changing 356|222
    "11" to "01"
!   .q
    r 756 0.988 3.658 69 level 2, 17

!   sr
```

The program then continues execution with index-name set to one, just as though the error had never occurred.

The type of error recovery shown above is strictly interactive. If such an error occurs in an absentee process, the run is aborted at that point; i.e., immediately after the "error" condition is signaled (assuming no handler has been established).

I/O ERRORS

Errors involving I/O activities performed on COBOL files are always anticipated errors. The user is able to define one or more status keys, which are set after each I/O operation, whether successful or not. These provide further information as to the nature of the error and can be tested by the program or displayed after the fact via a debugger. See "Status Keys" in Section IV for full details.

After the occurrence of an I/O error, a message in the format shown above is issued except that, in some cases, an additional message may precede. This occurs when the error has been discovered as the result of a status code returned by the I/O system. In this case the preliminary line of the message is:

<program name>: <Multics error message>

where <Multics error message> represents an interpretation of the status code identical to that obtained via the system subroutine com_err_. For more information on system status codes and error handling, refer to "Handling of Unusual Occurrences" in the MPM Reference Guide.

The user can cause a specified action to be taken on the occurrence of an I/O error either by including the INVALID KEY or AT END option with the DELETE, READ, REWRITE, START, or WRITE statements or by associating a declarative procedure with a particular file or class of files via the USE statement. The action performed is the group of statements immediately following the key word in the former case or the entire section headed by the USE statement in the latter. In either case, this action is performed in place of the normal action defined above. See "Declarative Procedures" in Section IV for further information.

When a declarative procedure is executed as the result of an I/O error, the user may wish to take different actions depending on the nature of the particular error. As mentioned above, data fields may be specified that will be set to certain values that correspond with specific errors. These can be used as determining factors in deciding what subsequent action is to be taken. However, the printing of the messages described above is preempted by the execution of the declarative procedure. These are usually more descriptive and precise than any information obtainable by evaluation of the status keys. In addition, they provide information concerning the line number and location of the statement causing the error. For this reason, the error messages that would have been issued to the error_output I/O switch are retained and may be printed via the print_cobol_error_ subroutine (refer to the print_cobol_error_ subroutine described in the MPM Subroutines manual). This subroutine may be called only from within a declarative procedure; it displays information applicable to the error causing execution of the declarative procedure that has invoked it. An example of how it may be used is:

```
SELECT file-a ASSIGN TO switch-a STATUS IS skey.
.
01 skey PIC XX.
.
PROCEDURE DIVISION.
DECLARATIVES.
  sec1 SECTION. USE AFTER ERROR PROCEDURE ON file-1.
    par1. IF skey = "10" GO TO par2.
          CALL "print_cobol_error_".
          STOP "File Error has occurred on file-a".
          GO TO par3.
    par2. DISPLAY "End of file on file-a".
          CLOSE file-a.
    par3. EXIT.
END DECLARATIVES.
```

In this way, the end-of-file condition can be checked without limiting the available information in case of an unexpected error. In addition, any restart after execution of the STOP causes execution to be continued after the I/O statement in error, rather than having it retried as would happen if a declarative procedure were not used.

The messages issued by print_cobol_error_ are directed to the error_output I/O switch. An additional entry point is available in this subroutine that allows the user to specify the switch to be used. Using print_cobol_error_$switch, the user can direct the output to error_output or to another switch, perhaps associated with an external file of stream organization defined in some program in the run-unit. For example:

```
SELECT EXTERNAL error-file ASSIGN TO error_report-PRINTER.
.
01 switch-name PIC X(12) VALUE "error_report".
.
PROCEDURE DIVISION.
DECLARATIVES.
report-error SECTION. USE AFTER STANDARD EXCEPTION PROCEDURE
                ON INPUT, OUTPUT, I-O, EXTEND.
par-1. OPEN EXTEND error-file.
       CALL "print_cobol_error_$switch" USING switch-name.
END DECLARATIVES.
```

In this way, all file errors occurring in the program are recorded in a printable segment and execution continues, bypassing the execution of any statements causing an error.

If print_cobol_error_ is improperly invoked with no pending error recorded, an error message is issued to error_output.

SIZE ERROR OPTION

If the result of a computation requires more significant digits than are available in the specified target field, a size error occurs. The user has the option of including a SIZE ERROR clause with any statement that involves an arithmetic computation: ADD, COMPUTE, DIVIDE, MULTIPLY, or SUBTRACT. With this argument he specifies a group of statements that are to be executed should such an error occur. If the argument is not used, the compiler makes no provisions for its occurrence. In this case, an unanticipated error occurs after which execution cannot normally be restarted.

Unanticipated Errors

Unanticipated errors are reported directly to the user through the error_output I/O switch as the result of the signaling of a system-defined condition. Condition handlers can be established to process any such condition, but not via a COBOL program. Each particular error is associated with a different condition; some are hardware detected such as numeric overflow errors, while others are software detected, such as record quota overflow. Whether corrective action can be taken following the error and the program successfully restarted, as with anticipated errors, depends on the nature of the particular error. A complete description of Multics error processing is given in "Handling of Unusual Occurrences" in the MPM Reference Guide.

The following example shows a correctable error situation:

```
    Error: record_quota_overflow condition by iox_$put_char|750
    (>system_library_standard>bound_vfile_)
    referencing >udd>PROJ>user>progname.switch_name|100000
    r 1016 7.480 106.128 1371 level 2, 18

!   delete *.list
    r 1016 0.064 0.210 13 level 2, 18

!   sr
```

In this case, the user has provided the additional records needed to perform the apparent I/O operation by deleting all the list files in the working directory and thus enabled the execution of the program to be completed.

Some common, and usually unrestartable, errors are described below.

1. A numeric data item is used as the sending field in a numeric MOVE or the operand in a computation, and it contains an invalid value in either the digit or sign position. This causes an IPR error; i.e., the signaling of the illegal_procedure condition, which in turn causes program termination, assuming that no handler has been provided. This is a common error in COBOL programs that attempt to use uninitialized signed numeric data under the assumption it has a zero value. This is due to the fact that uninitialized data normally consists of all zero bits (although this is not guaranteed) which happens to constitute an invalid sign value for Separate and Nonseparate Sign Display numeric data. Refer to "Data Types" in Section V and see Table 5-3. For example, in the program:

```
01 cnt PIC S999.
.
     ADD 1 TO cnt.
```

   such an error occurs upon execution of the ADD statement if cnt has not been previously set as the target of a MOVE or computation. The VALUE clause should always be used when it is necessary for data to have initial values.

2. A computation is performed in which the result requires more significant digits than the target field provides and the SIZE ERROR option has not been used. For example, in the program:

```
01 cnt pic 9 value 0.
.
     PERFORM par1 10 times.
par1. ADD 1 TO cnt.
```

   when cnt has the value 9, adding one does not set it to zero, but rather causes a fixed-point overflow error after which program execution is terminated.

3. A subscript is used to index an array and contains a value outside the range of the array. This type of error may go initially unrecognized causing the destruction of other data in the program. (Sometimes an attempt to address beyond the bounds of the segment occurs causing an out_of_bounds error.) This problem can be avoided if the -rck option is specified at compile-time. In this case, this situation results in an anticipated error.

4. A CALL statement is executed in which the reference name contained in the specified literal cannot be resolved by the linker. This results in a linkage error that may be recoverable by alteration of the execution environment. See "Dynamic Linking" in Section V for details on how to handle this type of error. A similar anticipated error occurs when a variable is specified with the CALL statement. In this case, recovery can be made as above or the contents of the variable can be altered, thereby changing the reference.

SECTION VII

EFFICIENCY CONSIDERATIONS

This section deals with efficient use of Multics COBOL. The user is warned against some particularly inefficient constructs and data types and also is given hints concerning program efficiency in the areas of program size, data definition, and I/O usage.

PROGRAM SIZE

On Multics, program size should be kept small. This allows the user to perform frequent interactive compilations and to take advantage of the flexibility provided by dynamic linking. Modularizing a complicated procedure into a number of programs with well-defined interfaces also serves to reduce the complexity of the resulting programs and to facilitate debugging activity.

In regard to code generated by Multics COBOL, programs that reference less than 32K words of data (i.e., 131,071 bytes) execute most efficiently. A pointer register is conventionally maintained so that the first 32K words defined can be referenced directly, without an index register and without the need to load a pointer register before each reference. Register pooling is employed for data beyond this nominal limit; i.e., registers to reference this data are shared for other purposes and may have to be reloaded before they can be used. For larger programs, the Working-Storage section should be set up so that the more frequently used data items are defined toward the beginning and less frequently used items toward the end. In this way, the more frequently used items are referenced via the dedicated pointer register. The record areas associated with files are usually allocated first and are thus usually referenced directly.

COBOL source-level segmentation should not be used. The user gains nothing from dividing a program into segments of varying priority, and the additional overhead necessary to simulate the activity of independent segments can even cause performance degradation. If segmentation must be used, for compatibility purposes, then the use of alterable GO TO statements in independent segments should be avoided if possible.

DATA DEFINITION

The VALUE clause should not be used to initialize Working-Storage data unless the data actually must be initialized. That is, it should not be used as a matter of course as it causes additional instructions to be executed when the program becomes part of the run-unit. It is however, the only way a COBOL program can cause data to be set upon first invocation only and thus make a distinction between the first and subsequent invocations.

Data not changed by the program should be defined in the Constant Section. Such data is not allocated in the data segment; i.e., it does not count in the 32K efficiency boundary mentioned above. Instead, it is allocated in the text section of the object segment immediately before the executable code. This not only allows it to be referenced efficiently by self-relative addressing, but also provides protection against accidental modification since the object segment is not normally writable. This data is pooled along with the literals used by the program so that many different data items may share the same storage space. For example, if a five-character data item with value "ABCDE" along with a one-character item with value "B" is defined in the constant section and the alphanumeric literals "BCD" and "A" are used in the program, all will refer to the same data area. That is, a total of only five characters need be allocated. Another advantage of Constant Section data is that the initialization is performed at compile-time rather than at run-time as with Working-Storage data. Thus, it takes no longer to execute a program for the first time in the run-unit with many constants than a similar program with none at all.

I/O CONSIDERATIONS

Relative and indexed files are treated as having identical file structure by the Multics I/O system. For relative files, the COBOL I/O run-time system keeps track of the relative record number for sequentially accessed files. Processing efficiency is approximately the same for relative and indexed files. Generally, sequential organization files are more efficient to process than relative or indexed, and stream files more efficient than sequential. Dynamic access mode, which should be specified only if the file must be referenced both randomly and sequentially, should be used sparingly because it reduces the efficiency of sequential reads when the program also contains DELETE and/or REWRITE statements.

Status keys should not be specified unless they are to be tested and used by the program, preferably in a declarative procedure. They add additional overhead on every I/O operation and provide no further information about an unexpected error than is provided by the normal error mechanism.

No significant extra activity is necessary to support the RECORD CONTAINS ... DEPENDING ON construct. It is advisable to use this construct when it is applicable, especially with files of stream organization. With print files, considerable space can be saved by writing variable-length records, especially if many trailing spaces are anticipated. Use of the ADVANCING option should be used, even with a variable, instead of writing blank lines.

USE OF NUMERIC DATA TYPES

Use display numeric data whenever the ratio of its use in computations to its use in a displayable format is fairly even. That is, the time saved in arithmetic operations by using more computational-oriented data may not outweigh the time lost in converting that data to an ASCII displayable format. Also, Display numeric data is often required for compatibility and portability when it is necessary that alphanumeric data share the same byte locations via the REDEFINES option. Nonseparate Sign Display data is not directly supported by Multics hardware and its manipulation requires significant software support. Thus, its use is strongly discouraged for it adversely affects program execution speed. It should be avoided and used only when required for compatibility purposes.

Packed Decimal data (designated by USAGE IS COMP, COMP-5, or COMP-8) is somewhat more efficient than Display data. Additionally, and most importantly, use of Packed Decimal reduces the space required by a factor of two to one. Conversion to displayable format is required, but involves minimal cost since the data is in decimal format, i.e., the 4-bit values represent decimal digits that are easily translatable to their ASCII counterparts.

Long Binary and Short Binary data (designated by USAGE IS COMP-6 and COMP-7, respectively) is highly efficient in arithmetic operations, especially under the conditions described below. However, conversion to a displayable format is significantly more expensive than for Packed Decimal. The choice between Long and Short Binary should be made on the basis of the precision required and/or compatibility requirements involving alignment. Word alignment is guaranteed only for Long Binary data; Short Binary is half-word aligned. Short Binary can hold values from -131,072 to 131,071, inclusive; Long Binary can hold values from -34,359,738,368 to 34,359,738,367, inclusive.

Code generated for computations involving Binary data is optimized when the computation can be performed in the hardware registers, with no possibility of the loss of significant digits. This code executes much faster than the code generated to deal with Display or Packed Decimal data. Code to perform computations using the hardware registers is generated for all addition, subtraction, multiplication and division operations (i.e., for the ADD, SUBTRACT, MULTIPLY, and DIVIDE statements and the COMPUTE statement not containing the exponentiation operator **) if the following conditions are true:

1. All target variables are either Long or Short Binary.

2. The number of operands in the expression is less than or equal to two.

3. All operands in the expression are either Binary data, the figurative constant zero, or literals having a value that is an integer and within the range specified above for Long Binary data.

4. None of the Short Binary targets or operands appearing in the statement are elements of an array.

5. If the operation is division, none of the target variables are to be rounded.

USE OF THE INSPECT STATEMENT

There are many variations of the INSPECT statement and for the general case a run-time support routine is invoked. However, optimization is performed for certain commonly used variations which result in efficient in-line code. It is often more efficient (and straightforward) to use more than one INSPECT statement rather than one with multiple operands.

A statement in the form:

```
INSPECT identifier-1 TALLYING identifier-2 FOR { { ALL     } { identifier-3 } }
                                               { { LEADING } { literal-1    } }
                                               {   CHARACTERS               }

    [ { BEFORE }  INITIAL  { identifier-4 } ]
    [ { AFTER  }           { literal-2    } ]
```

results in efficient code, provided identifier-3 or literal-1, and identifier-4 or literal-2 are non-numeric and have a length less than three. Note that if multiple tallying results (identifier-2), objects of inspection (identifier-3, literal-2), or inspection delimiters (identifier-4, literal-2) are specified, optimization cannot be performed and results may not be what is expected. For example:

```
01  str  pic  x(12) value "abababab cabа"
INSPECT str  TALLYING cnt1 FOR ALL "ab" AFTER "b"
                      cnt2 FOR ALL "ba" AFTER "a".
```

gives different results than:

```
INSPECT  str  TALLYING cnt1 FOR ALL "ab" AFTER "b"
INSPECT  str  TALLYING cnt2 FOR ALL "ba" AFTER "a".
```

Namely, for the former case (one INSPECT) cnt1 is increased by 1 and cnt2 is increased by 3, while for the latter (two INSPECT's) cnt1 is increased by 4 and cnt2 is increased by 4. The execution of the two INSPECT statements is faster by at least an order of magnitude than the one INSPECT statement and this difference increases drastically as the length of the inspected string increases and the number of hits decreases. For example, given a 1000 character string in which only a few matches are found, the difference in execution time would reach 4 orders of magnitude.

The same type of differences apply when using the REPLACING option. Statements in the form:

```
INSPECT identifier-1

    REPLACING ALL { { identifier-5 }  BY  { identifier-6 } } ...
                  { { literal-3    }      { literal-4    } }
```

and

```
INSPECT identifier-1 REPLACING

ALL { identifier-5 }
    { literal-3    }

    BY { identifier-6 }  [ { BEFORE }  INITIAL  { identifier-7 } ]
       { literal-4    }  [ { AFTER  }           { literal-5    } ]
```

result in efficient code if (1) all identifiers are non-numeric, (2) identifier-5 or literal-3 and identifier-6 or literal-4 are one character in length, and (3) identifier-7 or literal-5 (if used) are less than three characters in length. The code sequence generated is somewhat shorter if literals are specified for both the ALL value and BY value.

Like the TALLYING option, slight differences in format can result in enormous differences in efficiency and it is often more efficient (and straightforward) to use more than one INSPECT statement than multiple operands. For example:

```
move 'a' TO ida MOVE 'b' to idb
INSPECT str REPLACING ALL ida by 'A' AFTER 'b',
                      ALL idb BY 'B' BEFORE 'c',
```

yields the same results (i.e., "aBABABABcAbA") as:

```
INSPECT str REPLACING ALL ida BY 'A' AFTER 'b'
INSPECT str REPLACING ALL idb BY 'B' BEFORE 'c'
```

which is "aBABABABcAbA".

However, the latter is accomplished via a few machine instructions whereas the former requires a call to a run-time routine. Again, this is at least an order of magnitude difference with an exponential increase as the inspected string gets longer. It must be noted however, that the results of the above two variations are not necessarily always the same and depend on the values assigned to the identifiers. If "A" had been moved to idb, then the result of the single INSPECT would have been "abAbAbAbcAbA" while the result of the two INSPECT's would have been "abBbBbBbcBbB".

It should be noted that some very awkward and unwieldy forms of the INSPECT statement may be necessary to accomplish relatively simple and straight forward applications. For example, to convert all lower-case alphabetics in an alphanumeric data field to upper-case, the user would have to write the COBOL statement as:

```
INSPECT data-name REPLACING ALL "a" BY "A", "b" BY "B", "c" BY "C",
"d" BY "D", "e" BY "E", "f" BY "F", "g" BY "G", "h" BY "H", "i" BY
"I", "j" BY "J", "k" BY "K", "l" BY "L", "m" BY "M", "n" BY "N",
"o" BY "O", "p" BY "P", "q" BY "Q", "r" BY "R", "s" BY "S", "t"
BY "T", "u" BY "U", "v" BY "V", "w" BY "W", "x" BY "X", "y" BY "Y",
"z" BY "Z".
```

However this falls into the format shown above for which efficient code is produced. In fact, in this case, exactly one machine instruction is generated to accomplish the entire statement. On the other hand, the statement:

```
INSPECT data-name REPLACING ALL "ab" BY "AB", "bc" BY "BA".
```

results in the invocation of a run-time support routine and a character by character inspection (via software) of data-name which ANS rules demand.

MISCELLANEOUS CONSIDERATIONS

If possible, use indexing rather than subscripting. Using an index to reference an array element is considerably more efficient; additionally, this prevents you from referencing outside the range of the array.

It is more efficient to use the DIVIDE statement than to use COMPUTE with the division operator. The division is correct to the maximum number of digits of precision which the hardware can support in the latter case, whereas it is carried out only to the precision necessary for satisfying specified COBOL rules in the former.

If multiple operands are specified with the STRING, UNSTRING, and DISPLAY statements and some are variable length, it is more efficient if the fixed-length items are specified first. For the STRING and UNSTRING statements, nondelimited variables should be given first, if possible. Literal delimiters less than three characters in length are processed significantly more efficiently than longer literal delimiters which, in turn, are processed more efficiently than variable delimiters. It is far more efficient to execute one DISPLAY statement with multiple operands than to execute multiple DISPLAY statements. The same is not true of the OPEN statement.

Defining an item with the OCCURS ... DEPENDING ON clause causes references to a containing item to be significantly more time consuming. Thus, this construct should only be used when necessary. If necessary only for some references, it is worthwhile to REDEFINE the containing item as fixed length for the other times it is referenced.

MEASURING A PROGRAM'S PERFORMANCE

The cost of executing each statement of a program can be determined by specifying the compiler control argument profile. The information produced is of interest to both the beginning programmer and the expert. For the beginning programmer, it is a guide to the economics of programming and restores the view of hardware cost that a high-level language otherwise obscures. For the expert programmer, it is an indication of the points in a program that are unreasonably expensive and that require refinement.

To measure the performance of a program, the user specifies the -profile control argument in the cobol command that compiles the program. When the profile control argument is specified, additional code is generated to calculate statistics about the execution of each statement. After the program has been executed, the information involving the accumulated statistics can be displayed by invoking the following command:

profile sn

where sn is the segment name of an object segment. (Reference the profile command in the MPM Commands.)

For each statement in each line of the COBOL Program, a line is printed that gives the number of times the statement was executed, the number of instructions executed, and the support subroutines called as a result of the statement's execution. For example, consider the following lines from a profile listing:

LINE	STM	COUNT	COST	PROGRAM
8	1	1	7	
10	1	1	10 +	(open_int_file)
12	1	5	50 +	(write_stream)

The profile listing indicates that the statement on line 8 was executed once; this statement requires seven machine language instructions and does not require any support subroutines (i.e., operators). The statement on line 10 was executed once; this statement requires ten machine language instructions and one support subroutine, namely open_int_file. The statement on line 12 was executed five times; this statement requires ten machine language instructions and one support subroutine.

SECTION VIII

COBOL MESSAGE CONTROL SYSTEM

The design of the runtime package that supports the full Level-2 functional requirements of the ANSI COBOL-74 Communications Module is described in this section.

Multics COBOL processes ANSI COBOL-74 Message Control System syntax. Full Level-2 functions are provided for the SEND, RECEIVE, ENABLE, DISABLE, and ACCEPT (MESSAGE COUNT) verbs of the COBOL Message Control System, hereafter called CMCS or COBOL MCS. In addition, the PURGE verb from the CODASYL Journal of Development (JOD) is supported.

Messages can be any length up to 262,144 bytes (including a small amount of control information), and can be written and read in any number of pieces, thus allowing the possibility of intermixed messages in the queues. Delimiters for the pieces are specified, but cannot be imbedded in the data.

REFERENCES

It is recommended that the reader be familiar with the description of the Communications Module in the ANSI COBOL-74 Standard.

1. ANSI COBOL-74 Standard Definition, ANSI X3.23-1974

2. CODASYL JOD, 1976, for a description of the PURGE verb.

TERMINOLOGY

absolute tree path
: is a tree path that specifies all levels of a subtree necessary to identify a specific physical message queue.

command line
: is the command that is executed when a physical message queue contains a message. It is specified for the queue in the source for cmcs_tree_ctl.control. Rules for constructing command lines are given in the description of the cv_cmcs_tree_ctl command.

level name
: is the logical name, from 1-12 characters, associated with each level in a queue hierarchy definition.

level number
: is a number from 1-4 that is associated with each level in a queue hierarchy definition. It is not necessary to use all four levels.

mp line
: is similar to the command line and is used when a message processor is called to process a queue containing messages.

message delimiters
: egi end-of-group indicator
: emi end-of-message indicator
: esi end-of-segment indicator

physical queue name
: defines the name of the physical message queue. The actual entryname assigned always has a cmcs_queue suffix. In all subsequent discussions, the word "physical" is omitted from "physical queue name"; however, the meaning is the same.

queue hierarchy
: is a tree structure used by COBOL programs to access messages for (COBOL) communications processing. There can be up to four levels in any subtree and any number of subtrees. Each level is identified by a level number and a level name. In the Multics implementation, the level names are logical (i.e., the physical message queues are identified with a separate name associated with each terminating branch (tree path) in the hierarchy definition).

tree path
: is the concatenation of the level names in a particular branch of the subtree. It is the tree path by which COBOL application programs identify the particular physical message queue or queue hierarchy to be accessed. A tree path has one of two forms:

1. Internally, it is always a 48 character string, consisting of the concatenation of the four 12-character level names. The level names are blank filled to a maximum of 12 characters, and trailing unused level names must be supplied and must be blank.

2. Externally, the tree path can be a quoted string of the internal form or it can be a variable length string of characters with up to four period-delimited level names, similar to the components of an entryname in the storage system. In this form, the level names are not blank filled.

Examples of the two formats of tree paths are:

```
"orders      cloth       shirts      dress

orders.cloth.shirts.dress
```

Notice that it is possible to have a tree path (in this form) that is 51 characters in length if all four level names are given and they are each 12 characters long.

DESIGN CONCEPTS

The COBOL-74 Communications Module is the section of the ANSI COBOL standard that defines the COBOL MCS. COBOL MCS is a general facility used for writing and reading messages in message queues, invoking application routines to process messages, and controlling access to the terminals and queues. (Terminal access in Multics is controlled only as it relates to CMCS; outside the context of CMCS, no CMCS controls are imposed.)

- Full Level 2 COBOL MCS functions are provided with MR6.0. " Full Level 2" dictates that all functions adhere strictly to the ANSI COBOL-74 definition of CMCS. (The purge function as defined in the CODASYL JOD is also provided.)

- The Multics implementation of COBOL MCS is based on the concept of a "station." A station is a logical entity, having controls imposed by the system, that can be attached by a process. Its primary purpose is to provide a uniform mechanism for identifying sources and destinations of messages. Thus, the facility is independent from terminals, user-ids (including anonymous users), or constraints placed upon interactive or absentee users.

  In this usage, the term "attach" means only that an available resource becomes solely owned for current use by a specific process. The connotation of a Multics I/O attachment does not apply.

  A specific station can be attached by an individual process; by default, it is assigned to users dynamically on the basis of terminal subchannels.

- A process can attach only one station.

- CMCS uses the standard Multics interfaces for terminal I/O. The only control imposed by CMCS is that the runtime package checks to verify that a queue or terminal is enabled before attempting to do I/O with that target.

- Every process must attach its station before proceeding. The first attachment initializes the user's environment for CMCS processing. At this point, a user can perform any CMCS operation.

- All message queues and system control tables for a given set of users are contained in a single, user-specified directory. If desired, a different set of users can operate in a different directory.

- The COBOL language specification requires user programs to specify a password for the ENABLE and DISABLE verbs. The need to avoid putting literal passwords into program source is emphasized later in this section.

- All COBOL application programs are benign, i.e., they always access CMCS queues and control segments through the CMCS runtime interface. If this policy is violated, incorrect operation may occur.

- Most messages are short. Thus, a message sent to multiple destinations causes a copy of the message to be placed in each destination's queue.

- When a user sends only a portion of a message, it is likely that it is either a long report or a file data copy. Thus, a maximum size holding buffer must be used. For this reason, a temporary segment is assigned to each queue when the user sends a partial message to that queue.

- The meaning of message length (to a COBOL program) becomes ambiguous if the slew controls are imbedded in the data. For this reason, the slew control information is kept separate from the message text until the message is to be sent to an output device.

- The COBOL-74 Communications Module description is not specific as regards the number of passwords needed for CMCS. Until ANSI clarifies the rules for the use of passwords only one password, at the CMCS system level, is used. This password must be matched by all users wishing to perform enable/disable functions.

- The COBOL application program must use the absolute tree path of the target queue to receive subsequent pieces of a message. To eliminate ambiguities in the processing of receives with tree paths that are subsets of an absolute tree path for a receive in process (not all segments of the message have been read), the following rule is established:

A receive request with a tree path of a.b, that is answered with a message from a.b.c, must be completed before any other message request from a or a.b. can be processed. Any attempt to use just a.b is rejected.

Continuation receives for a.b.c are valid (and appropriate), as is a receive request addressed to a.b.d or a.b.f, where a.b.d and a.b.f are not necessarily absolute tree paths.

As an example,

Tree Path	In-process	Queue Name
a		
a.b		
a.b.c	yes	queue_1
a.b.d	yes	queue_2
a.b.f	yes	queue_3

given that requests for absolute tree paths a.b.d and a.b.f are in-process, a new request for tree path a.b is rejected with cmcs_error_table_$ambiguous_tree_path. This causes a status key of "20" to be returned to the requesting COBOL application program.

- A message queue is used to hold output messages for terminals until they are written to the output device, or to hold input messages until they are read by the COBOL application programs.

- In the Multics implementation, there is no physical distinction between queues accessed for receives, and queues used to hold messages for destinations.

  Because of this, the CMCS queue hierarchy definition must include the specification of all queues, both application queues and destination queues.

  In send operations, the destination is translated into a station name, and thus has a specific queue assigned to hold the messages for output to a terminal device.

- In an initial implementation, a user can attach (own) only one station (to receive output as a destination). The process does not own the queues it accesses for normal receives.

  Only one password is used to validate all enable and disable requests.

- The EMI and EGI message delimiters are processed identically by the runtime package. Differentiation in the meaning of these two (logical) delimiters is left totally to user software.

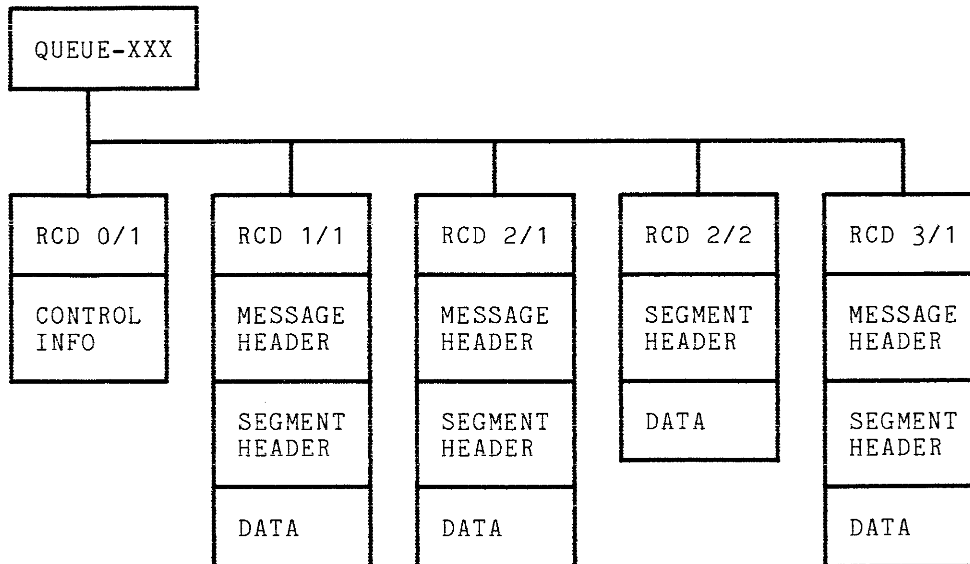
- Attempting to do a send and a receive on the same tree path in the same process is not allowed in the Multics implementation. Once either of the operations is completed, the other can be started without constraint.

  For multiple processes, locks are used for critical areas of queue manipulation. Locking is kept to a minimum to reduce inter-process interference.

- Only the particular message being received is locked on an extended basis. The entire queue is locked only long enough to accomplish the message lock and changing the status lists.

- The Multics implementation design requires a process to specify its CMCS directory explicitly. This is done with the cobol_mcs command.

COBOL MCS QUEUE ORGANIZATION



Notes:

Any given message can span multiple non-contiguous records of the file. Only the first record for a given message contains a message header. Subsequent pieces (segments) contain only the segment header and data.

Keys for the vfile records consist of two adjacent fields of fixed bin(35) values. The first field contains the ordinal number of the CMCS message. The second field holds the ordinal number for message segments within the given message. Message numbers and message segment numbers begin with the value one. When the physical queue is created, a record with key values of 0/1 is stored. This record contains global control information for that queue. It is called the queue control record.

Part of the header record for each message is a pair of forward and backward pointers. The message header of a message is linked into a list of pointer pairs, based upon the status of the record.

Current status codes are:

1 send in process (message being built)
2 send complete (available for processing)
3 receive in process
4 receive complete (ready for deletion)

OVERVIEW OF CMCS DATA BASES

For a given set of users, the data bases described below and the associated message queues reside in a single directory. A different set of users can have the control segments and queues in a different directory.

cmcs_terminal_ctl.control
: provides the default station_id for interactive users (based on user-device channel).

cmcs_tree_ctl.control
: contains the template definitions of all CMCS queue hierarchies for a given set of users. This segment is copied into the process_dir during the user's CMCS initialization and is then dynamically updated with user-specific information for each entry used.

cmcs_station_ctl.control
: defines all legitimate stations and contains per-station flags to indicate enable/disable conditions.

cmcs_wait_ctl.control
: is shared by all processes performing a receive with wait. Entries are searched by queue hierarchy on a first come, first served basis.

cmcs_system_ctl.control
: this segment initially contains the single CMCS password (up to 10 characters), used in granting permission to perform the enable and disable functions as given in the language. In addition it contains a field that specifies the number of seconds to wait in attempting to lock any of the CMCS control segment locks.

cmcs_queue_ctl.control
: contains the flags for enable/disable functions on a per-queue basis. Additionally, it holds the message status counters and linked-list pointers for each queue. Entries in this table are searched to find occurrences of available messages before the actual queues are accessed.

cmcs_user_ctl.control
: this per-process, external static data base contains all the per-process parameters used by the various CMCS subroutines.

ADMINISTRATIVE FUNCTIONS

CMCS Administrator

The CMCS administrator must define and generate the system data bases (and their containing directories). The following data bases require a source segment for compilation by CMCS compilers:

cmcs_station_ctl.control
cmcs_terminal_ctl.control
cmcs_tree_ctl.control

The station control segment must be generated before the terminal and tree control segments. After the tree control segment is generated, the cobol_mcs_admin command is used to create all queues and additional control segments:

cmcs_queue_ctl.control
cmcs_wait_ctl.control
cmcs_system_ctl.control

After the system control segment is created, the administrator must use the set_cmcs_psw request of the cobol_mcs_admin command to set the initial password for the CMCS system.

Additionally, the administrator must manually set the ACLs on all CMCS segments, as appropriate for the given set of users. All segments, with the exception of the cmcs_system_ctl.control and cmcs_terminal_ctl.control segments, must have read and write access for all users. Only the administrator need have write access on the cmcs_system_ctl.control segment (to change the password).

The cmcs_tree_ctl.control segment must have read and write access for all users. In addition, the administrator must set the copy switch "on". This segment is copied to the user's process directory, so that it can be updated with process-specific information.

DAEMON MESSAGE PROCESSOR

The message processors invoke the COBOL application programs that actually process the input messages and generate the output messages. They are similar to the system daemons in that they are initiated by the operations personnel from a system terminal, and thus are not subject to idle process timeouts or cpu time usage constraints. In addition, they can use the daemon message routing facility so that multiple processes do not require multiple terminals. However, in all other respects, they are like every other interactive process, subject to all access and privilege controls.

Each message processor process has a Person_id.Project_id chosen by the CMCS application administrator to be the application program User_id. The start_up.ec exec_com is used to condition the process environment and calls the cobol_mcs command, to start the actual processing of messages.

Once logged in, the message processor adds itself to a list of message processors waiting for a transaction. It remains in the list until told to log out by the application administrator (with the stop_mp request).

System Administrator Actions

The system administrator registers the project and users for a given CMCS application system. The project must be given the daemon attribute in the system administrator table (SAT). This permits the operator to log in the process from a system terminal.

In addition, the system administrator determines if and how the daemon terminal I/O is to be routed, and sets up the appropriate message routing segments in >sc1 and controls in the system_start_up.ec.

Project Administrator Actions

The project administrator adds the Person_id of each message processor to the project master file (PMF). The Person_id also must be given the daemon attribute.

The project administrator also creates a start_up.ec exec_com for the login home directory of the message processor User_id. It has the general format given below.

The project administrator coordinates with the system administrator on the names to be used by the operators when logging in the message processor daemons. (However, once the message processors are logged in, no further communications with that process are necessary unless some abnormal situation occurs.)

```
&command_line off
cwd <application directory path>
cobol_mcs -wd -message_processor <station_name>
logout
```

Operator Actions

The operator logs in the daemon message processors in the same manner as for logging in the IO daemons. It may be desirable to set up the login functions in the admin.ec exec_com.

USER COMMANDS

A summary of the commands provided as part of the COBOL MCS runtime support facility are provided below and are immediately followed by the detailed descriptions.

cobol_mcs, cmcs
: serves to establish the environment for further CMCS processing. Users specifying the terminal option can perform any of the functions of the COBOL send, receive, enable, disable, accept message count, and purge verbs. If the message processor option is used, the command performs the process initialization and then enters a wait-state. This is in preparation for the execution of a COBOL application program.

cobol_mcs_admin, cmcsa
: is an administrative command with three major functions. The first is to create necessary control segments and message queues for a given CMCS directory, using the cmcs_tree_ctl.control segment as input. The second is to set or change the system-wide password, used to validate enable or disable requests. The third is to administer the message processing interfaces (i.e., starting and stopping the message processor).

cv_cmcs_station_ctl
cv_cmcs_terminal_ctl
cv_cmcs_tree_ctl
: are three control segment compilers based on the reduction_compiler tool. They each read a source file called cmcs_XXX_ctl.src, where XXX is station, terminal, or tree, respectively, and generate a binary control segment of the name cmcs_XXX_ctl.control. Subroutines that access the binary control segments have a name of the form cmcs_XXX_ctl_.

A comprehensive example utilizing cmcs and cmcsa is provided at the end of this section rather than with each of the following command requests.

Name: cobol_mcs, cmcs

Provides a command interface to the CMCS and functions in a manner similar to that used inside a COBOL program. Refer to the Multics COBOL Reference Manual for a complete description of the Communications Module.

The first time this command is invoked in the user's process causes initialization for the execution of all subsequent CMCS operations. If the process is to operate as a CMCS terminal, the command reads subcommands from the user_input switch.

Usage

```
cobol_mcs cmcs_dir {-control_arg}
```

where:

1. cmcs_dir
   is the path of the directory containing the desired CMCS message queues and control segments.

2. control_arg
   must be one of the following:

   -message_processor, -mp
   followed by a valid CMCS station_name. The use of -station causes the process to be initialized for subsequent use by a COBOL application program.

   -terminal, -term
   optionally followed by a station_name. If the station_name is not given a default station_id is used. The use of -terminal causes the process to be initialized to act as a CMCS source/destination.

Notes

This command must be invoked to initialize the user's process before any other COBOL MCS functions are performed.

Once cobol_mcs is invoked for terminal operations, the command reads requests from the user_input switch. The requests supported are receive, send, enable, disable, accept, purge, and quit. They are identical in function, although slightly different in format, to the corresponding verbs described in the Multics COBOL Reference Manual under "Communications Module." (The purge request is described in the CODASYL JOD.) Full ANSI COBOL-74 Level 2 support is provided.

Request Summary

Requests and their abbreviated forms are listed below and are immediately followed by a detailed description.

. (who am I?)
prints the short name of the command and the attached station name.

accept_message_count, amc
prints a count of all messages available in the specified queue hierarchy.

disable_input, di
disable_input_terminal, dit
disable_output, do
disables a queue hierarchy, a station, or a set of stations, respectively, from further activity, after the currently active messages are processed. (This distinction is made because the COBOL definition requires that messages currently being processed must be completed before the queue or the terminal is disabled.)

enable_input, ei
enable_input_terminal, eit
enable_output, eo
enables a queue hierarchy, a station, or a set of stations, respectively, for further activity.

execute, e
passes the remainder of the line to the system command processor.

purge, p
causes all partially sent messages to be deleted, and all partially received messages to be marked again as available.

quit, q
causes the cobol_mcs command to purge any incomplete send and/or receive messages and then returns to command level.

receive, r
prints and deletes all messages available in the specified queue hierarchy.

send, s
sends input lines as messages (or message segments) to the destinations specified (partial messages are first accumulated until they are complete, before writing to the destinations).

Request

The "." request is a convenient means for the terminal user to verify that he is at request level in the cobol_mcs command. It causes the short name of

the command, cmcs, to be printed. In addition, the name of the attached station is printed.

Usage

Request

accept_message_count, amc

This request causes the command to search the appropriate queues for a count of all messages currently available for processing (send complete) and prints the sum on the user_output switch. Unless there are no other CMCS users on the system, the availability of messages may change between the accept_message_count and any subsequent receive requests.

Usage

accept_message_count tree_path

where tree_path is a character string of the form a.b.c.d. The components, a, b, c, and d represent the four levels of a CMCS queue hierarchy (maximum). The level names must be alphanumeric (including underscore), and can be from 1-12 characters in length; trailing blanks are appended internally when appropriate. At least the first component is required; trailing components are necessary only to define the desired level in the queue hierarchy.

Request

disable_input, di
disable_input_terminal, dit
disable_output, do

These requests require a password, using the non-print function of the terminal or a mask to avoid the printing of the password. The password response is encoded and compared to the CMCS system password. If equal, the command processes the arguments on the request line. The result is identical in function to that of the disable verb in the COBOL language.

Usage

```
disable_input tree_path
disable_input_terminal source
disable_output dest1 {dest2 ... destn}
```

where:

1. tree_path
   is as above.

2. source and desti
   both refer to station names. They can be the same name if it is desired to prevent a particular station from both entering messages into the system and getting output (messages) from the system. Up to ten destinations can be specified.

Request

```
enable_input, ei
enable_input_terminal, eit
enable_output, eo
```

These requests operate identically to that of the disable requests, except that they enable the specified queues or terminals.

Usage

```
enable_input tree_path
enable_input_terminal source
enable_output dest1 {dest2 ... destn}
```

where tree_path, source, and desti are the same as in the disable request above.

Request

```
execute, e
```

This request is used to execute other Multics commands while operating under the cobol_mcs command.

Usage

```
execute command_line
```

where command line is any command line to be passed to the Multics system. The format is the same as input at normal command level.

Example

To print summary information on how many users are on the system, type:

```
! execute how_many_users
  Multics 2.0, load 5.0/50.0; 6 users
```

Request

```
purge, p
```

This request causes all partially sent messages to be deleted, and all partially received messages to be marked again as available.

Usage

```
purge s {dest1 ... destn}
```

where desti is as above.

Notes

The purge s request causes all partially sent messages (for that process) being sent to the listed destinations to be deleted. If no destination list is given, all partially sent messages are deleted.

Request

quit, q

This request causes an implied purge all request and terminates the cobol_mcs session. The user is then returned to Multics command level.

Usage

quit

Request

receive, r

This request causes the appropriate queue or queues to be searched for the first available message. If none is found, a "No messages" comment is generated and the command returns to request level. (The cobol_mcs command interface does not "wait" for a message to become available.) If one is found, the message is printed on the user-output switch, along with any appropriate slew-control data.

Usage

receive delim tree_path

where:

1. delim
   is either esi or emi, to indicate that either a message segment, or an entire message is desired.

2. tree_path
   is the same as in the accept_message_count request above.

Notes

After the complete message is printed, it is deleted from the queue. When performing a receive segment request, it finds the first available message and prints the first segment of that message. Subsequent invocation of the request with either a null or the absolute tree_path causes the following segments of the message to be printed. The message is deleted after the last segment of the message is printed.

The receive request must be used to obtain output sent to the station (destination). For this purpose, tree_path is the user's station_name. (The "." request can be used to print the station_name of the process.)

Request

send, s

This request duplicates the function of the send verb with one of the three logical delimiters (esi, emi, or egi respectively).

Usage

send delim dest1 {dest2 ... destn}

where:

1. delim
   is either esi, emi, or egi, to indicate that the data is to be sent as a segment, message, or group, respectively.

2. desti
   is a list of one or more destinations to which the message is sent. Up to ten destinations can be specified.

Note

When any of the send requests are issued, the command enters an input mode, similar to that in the edm command. Lines of input are accumulated until a line is input with a single period. At this point, the accumulated data is sent to the appropriate destination(s).

Name: cobol_mcs_admin, cmcsa

This command is used to perform the software functions involved in COBOL MCS administration. When the command is invoked, it reads request lines from the user_input switch. Command usage is terminated when the user enters the quit request.

Usage

```
cobol_mcs_admin cmcs_dir
```

where cmcs_dir is the path of the desired CMCS directory.

Request Summary

Requests and their abbreviated forms are listed below and are immediately followed by a detailed description.

. (who am I?)
    prints the short name of the command.

change_cmcs_psw, ccpsw
    verifies and then overwrites a new password over an old password.

create_cmcs_queues, ccq
    creates the queues and other data bases using tree control as its input.

execute, e
    passes the remainder of the line to the system command processor.

quit
    causes the termination of the cobol_mcs_admin session. The user is then returned to Multics command level.

set_cmcs_psw, scpsw
    overwrites a new password over an old password.

start_mp
    begin processing of queues that contain messages.

stop_mp
    causes a logout after completion of the current message.

Request

.

This request prints the short name of the command, cmcsa.

Usage

.

Request

change_cmcs_psw, ccpsw

The user is asked for the old password, where password is a character string of 1-10 characters in length. If the response is correct, the command requests the new password. It requests it a second time to verify that the first typein was correct. The password given is encoded and written over the old password.

Usage

change_cmcs_psw

Request

create_cmcs_queues, ccq

This command reads the cmcs_tree_ctl.control segment in the cmcs_dir directory to obtain the defined queue names and the command_line control information associated with each queue. It creates these queues if they do not already exist; if a given queue already exists, the queue is truncated, a warning to that effect is printed, and the command continues. In addition, the command creates (or recreates) the cmcs_queue_ctl.control segment. This segment contains a list of the queues and, for each queue, the message status lists and flags for enabling/disabling input and output to and from the queues, respectively. It also creates the cmcs_wait_ctl.control and cmcs_system_ctl.control segments. These segments initially contain only the header information.

Usage

    create_cmcs_queues

Request

    execute, e

This request is used to execute other Multics commands while operating under the cobol_mcs command.

Usage

    execute command_line

where command line is any command line to be passed to the Multics system. The format is the same as input at normal command level.

Example

To print summary information on how many users are on the system, type:

```
! execute how_many_users
  Multics 2.0, load 5.0/50.0; 6 users
```

Request

    quit, q

This request causes the termination of the cobol_mcs_admin session. The user is then returned to Multics command level.

Usage

    quit

Request

set_cmcs_psw, scpsw

The user is asked twice for a new password, the second time to verify correctness of the first, where password is a character string of 1-10 characters in length. The password given is encoded and written over the old password without verification of the old password.

Usage

set_cmcs_psw

Request

start_mp

This request causes message processing processes to be awakened to begin processing of queues that contain messages.

Usage

start_mp

Note

The start_mp request should not be used as long as any message processors from a previous request are still active.

---

cobol_mcs_admin

Request

stop_mp

This request is used to cause the message processing processes to log out after completing the processing of the current message.

Usage

stop_mp

Name: cv_cmcs_station_ctl

This command compiles a source file named cmcs_station_ctl.src into a binary control file that is accessed by the CMCS runtime subroutines. In addition, it converts a source file that contains a list of all valid stations and becomes the master file for all station names.

Usage

```
cv_cmcs_station_ctl cmcs_station_ctl.src
```

where the source file has the following syntax:

```
<station_name>;
     .
     .
     .
end;
```

Notes

The binary control file is set up with the standard CMCS header, which gives information about the compilation and the table sizes.

This compilation must be done before either the cmcs_tree_ctl or cmcs_terminal_ctl compilations are done because the cmcs_station_ctl.control segment is used to validate the station names used in the other source files.

Name: cv_cmcs_terminal_ctl

This command compiles a source file named cmcs_terminal_ctl.src into a binary control file that is accessed by the CMCS runtime subroutines. In addition, it converts a source file giving pairs of terminal subchannels (tty_device_channels) and their default station names.

Usage

```
cv_cmcs_terminal_ctl cmcs_terminal_ctl.src
```

where the source file has the following syntax:

```
<terminal_subchannel>:        <default_station_name>;
     .
     .
     .
end;
```

Notes

The binary control file is set up with the standard CMCS header, which gives information about the compilation and the table sizes.

This compilation must be done after cmcs_station_ctl.control is generated as it uses information in that segment to validate the station names.

Name: cv_cmcs_tree_ctl

This command compiles a source file named cmcs_tree_ctl.src into a binary control file that is accessed by the CMCS runtime subroutines. In addition, it converts a source file that defines the COBOL MCS queue hierarchy, along with controls to be associated with each message queue. A discussion of the source segment is shown below.

Usage

```
cv_cmcs_tree_ctl cmcs_tree_ctl.src
```

where cmcs_tree_ctl.src contains the complete queue hierarchy definitions for application programs and terminal users. See below for further details.

Notes

The binary control file is set up with the standard CMCS header that gives information about the compilation and the table sizes.

This compilation must be done after cmcs_station_ctl.control is generated as it uses information in that segment to validate the station names.

Source Format for the Tree Definition

1. There can be any number of subtrees, and each subtree can have from 1-4 levels. The "root" for all subtrees is an implied one; this allows the CMCS administrator complete flexibility in the hierarchy definition.

2. The queue names specified in the queue hierarchy are entirely logical. For the terminating queue name in any branch, there is an associated name for the physical message queue. The physical queue name defined in the source is append with a cmcs_queue suffix when the queue is created.

3. The source file consists of one or more PL/I-like structure declaration statements. Each statement must begin with a "declare" or "dcl" and end with a semicolon (;). The structure can be up to four levels deep. All lines but the last line of a statement are terminated with a comma (,). The source file is terminated with a statement of "end;". PL/I-style comments may be included.

4. Immediately following the level indicator is the hierarchy level name; this is the only required argument on higher levels. An optional control for any line is mp_line <mp_line>. This control is in effect for the current level and all subordinate levels, unless overridden at a subordinate level. The <mp_line> argument is executed, if appropriate, when the associated queue goes non-empty.

5. When a particular level is the final level of a given tree path (terminal node), the "queue <queue_name>" control must be given. This identifies the desired name for the given physical message queue.

6. Since all physical queues must exist in the same directory as the compiled tree control, the associated queue names must be unique. Since they have an appended cmcs_queue suffix, the names must be fewer than 22 characters in length.

7. All stations (sources and destinations) must have a one level entry of the form:

   ```
   declare 1 <station_ name> queue <queue_name>;
   ```

8. All physical input queues must have a corresponding declaration as a "destination" queue. This destination is used by the terminal operator to direct a given input transaction to the proper queue for processing by the COBOL application programs.

Usage Example

```
/* COBOL MCS Hierarchy Queue Definition */


dcl 1 ln_a1,
    2 ln_a2,
      3 ln_a3,
        4 ln_a4 queue_name queue_1;

dcl 1 ln_b1 mp_line "ioa_ ""This is a command line (^a, ^a).""",
    2 ln_b2 cobol_program_id cobol_program_name,
      3 ln_b3a queue_name queue_2,
      3 ln_b3b queue_name queue_3;

/* Station Queue Definitions */

dcl 1 station_1 queue_name station_1;
dcl 1 station_2 queue_name station_2;
dcl 1 station_3 queue_name station_3;

/* Input Message Queue Definitions */

dcl 1 t1 queue_name queue_1;
dcl 1 t2 queue_name queue_2;
dcl 1 t3 queue_name queue_3;

end;
```

In this example, the command control given in the first level 2 line causes the <command_line> to be set for both of the level 3 queues immediately following. Effect of the command control is terminated by the following level 2 line.

Constructing Command Lines for the Message Processor

The CMCS message processing command interface is designed to provide considerable flexibility in the way COBOL application programs are invoked. Three requirements that must be satisfied for proper operation are:

1. The application program must have the FOR INITIAL INPUT clause for one communication description (CD) entry.

2. The program must be invoked with the 48 character absolute tree path of the message queue causing the invocation of the message processor.

3. The programs must use the NO DATA phrase on all receive statements and return (STOP RUN) when none are found. (This frees the message processor process to be available for processing other messages.)

The following procedure is used by the message processor to construct a command line that is to be given to the command processor:

1. Initialize the command string to null.

2. If there is an mp_line entry in the tree_ctl_entry for the given queue, append it to the command string and follow it with one blank character.

3. If there is a cobol_program_id entry in the tree_ctl_entry for the given queue, append it to the command string and follow it with one blank character.

4. Check the command string to see if it is still null; if it is, ignore the new message and queue up in the wait line.

5. Append the absolute tree path (as a quoted string) to the end of the command string and pass the string to the command processor.

COMMAND EXAMPLE

The following example takes into account most of the requests that are associated with the cobol_mcs and cobol_mcs_admin commands.

In the example, lines typed by the user are indicated with an exclamation mark (!) to the left of the line. This is for illustrative purposes only; the user does not actually type the exclamation mark. Likewise, comments that serve an explanatory purpose are included in the example enclosed within "/* ... */".

Refer to "Usage Example" shown under the cv_cmcs_tree_ctl command for the COBOL MCS hierarchy queue definition.

```
!  cmcsa -wd
!  ccq
!  .     /* who am I? */
   cmcsa
!  scpsw
   Input COBOL MCS password:
!  ABGDEFGH9JKB      /* new password */
   Please repeat for verification...
!  ABGDEFGH9JKB
!  quit
   r 815 0.178 0.110 10

!  cmcs -wd -term station_1
!  .
   cmcs, station_1
!  accept_message_count ln_a1
   Message count for "ln_a1" is 000000.
!  accept_message_count ln_b1
   Message count for "ln_b1" is 000000.
!  send emi t1 t2
!  1     /* message 1 */
!  .     /* message termination */
!  accept_message_count ln_a1
   Message count for "ln_a1" is 000001.
!  accept_message_count ln_b1
   Message count for "ln_b1" is 000001.
!  receive emi ln_a1

   1
!  receive esi ln_b1

   1
!  accept_message_count ln_a1
   Message count for "ln_a1" is 000000.
!  accept_message_count ln_b1
   Message count for "ln_b1" is 000000.
!  send esi t1 t2 t3
!  2     /* message 2 */
!  .
!  send esi t1 t2
!  3     /* message 3 */
!  .
!  send emi t1 t2 t3
!  4    /* message 4 */
!  .
!  accept_message_count ln_a1
   Message count for "ln_a1" is 000001.
!  accept_message_count ln_b1
   Message count for "ln_b1" is 000002.
```

```
!  accept_message_count ln_b1.ln_b2.ln_b3a
   Message count for "ln_b1.ln_b2.ln_b3a" is 000001.
!  accept_message_count ln_b1.ln_b2.ln_b3b
   Message count for "ln_b1.ln_b2.ln_b3b" is 000001.
!  receive esi ln_a1

   2
!  receive emi

   3

   4
!  receive emi ln_b1

   2

   3

   4
!  receive esi ln_b1

   2
!  receive esi

   4
!  receive esi
   cobol_mcs: Previous tree path is blank.  Please reenter request with new tree path.
!  accept_message_count ln_a1
   Message count for "ln_a1" is 000000.
!  accept_message_count ln_b1
   Message count for "ln_b1" is 000000.
!  disable_input ln_a1
   Input COBOL MCS password:
!  [illegible]
   Please repeat for verification...
!  [illegible]
!  send emi t1
!  rejected_input.  /* message 5 */
!  .
   cobol_mcs: A specified message queue is currently disabled. From send.
   IO Type: "Send    ", IO Subtype: "Message                    ", Status Key: "10"
   One or more destinations are disabled. Action completed.
   Station       Error Code
   t1            1

!  enable_input ln_a1.ln_a2.ln_a3.ln_a4
   Input COBOL MCS password:
!  [illegible]
   Please repeat for verification...
!  [illegible]
!  send emi t1
!  accept input. /* message 6 */
!  .
!  receive emi ln_a1

   accept input.
!  disable_input_terminal station_1
   Input COBOL MCS password:
!  [illegible]
   Please repeat for verification...
!  [illegible]
!  send emi t1 t2
!  reject input.  /* message 7 */
!  .
   cobol_mcs: A specified message source is currently disabled. From send.
   IO Type: "Send    ", IO Subtype: "Message                    ", Status Key: "20"
```

```
   One or more destinations unknown.  Action completed for known destinations.
   No  action  taken  for  unknown  destinations.   Data-name-4  (ERROR  KEY)
   indicates known or unknown.
   Station       Error Code
   t1            1
   t2            1

!  enable_input_terminal station_1
   Input COBOL MCS password:
!  [illegible]
   Please repeat for verification...
!  [illegible]
!  send emi t1
!  accept input.  /* message 8 */
!  .
!  receive emi ln_a1.ln_a2.ln_a3.ln_a4

   accept input.
!  disable_output station_1
   Input COBOL MCS password:
!  [illegible]
   Please repeat for verification...
!  [illegible]
!  send emi t2
!  accept input.  /* message 9 */
!  .
!  receive emi ln_b1
   cobol_mcs: A specified message destination is currently disabled. From receive.
   IO Type: "Receive ", IO Subtype: "Message, No Wait        ", Status Key: "00"
   No error detected. Action completed.
!  enable_output station_1
   Input COBOL MCS password:
!  [illegible]
   Please repeat for verification...
!  [illegible]
!  receive emi ln_b1

   accept input.
!  receive emi ln_a1
   cobol_mcs: No message exists in the specified queue hierarchy. From receive.
   IO Type: "Receive ", IO Subtype: "Message, No Wait        ", Status Key: "00".
   No error detected.  Action completed.
!  receive emi ln_b1
   cobol_mcs: No message exists in the specified queue hierarchy. From receive.
   IO Type: "Receive ", IO Subtype: "Message, No Wait        ", Status Key: "OC
   No error detected. Action completed.
!  send esi t1
!  partial input.  /* message 10 */
!  .
!  receive emi ln_a1
   cobol_mcs: No message exists in the specified queue hierarchy. From receive.
   IO Type: "Receive ", IO Subtype: "Message, No Wait        ", Status Key: "00"
   No error detected. Action completed.
!  send emi t1
!  complete input.  /* message 11, the balance of message 10 */
!  .
!  receive emi ln_a1

   partial input.

   complete input.
!  quit
   r 816 0.145 0.228 20
```

SECTION IX

FILE ORDERING -- SORT AND MERGE

CONCEPTS

Multics COBOL includes the SORT and MERGE statements to provide a generalized file sorting and merging capability.

Sorting

Much data processing depends upon the order in which records appear on the files being processed. Such processing often depends upon the order of the records being sequenced according to the values of one or more fields that appear in each of the records. The fields upon which the ordering depends are called the "keys" of the file.

Since data in its original form seldom occurs in well ordered sequences, a technique, sorting, is provided by which the user can impose the desired ordering upon the records in a file. A sorting procedure manipulates an input file whose records are in an indeterminate sequence and produces an output file containing the same set of data records rearranged into the desired sequence. The number of records in the input file is usually unknown at the inception of the sort procedure and is generally not relevant to the sorting procedure.

SORT STATEMENT

The SORT statement creates a sort file by executing input procedures or by transferring records from another file; sorts the records in the sort file on a set of specified keys; and, in the final phase of the sort operation, makes available each record from the sort file, in sorted order, to some output procedures or to an output file.

General Format:

```
SORT file-name-1 ON { ASCENDING  } KEY data-name-1 [ , data-name-2 ] ...
                    { DESCENDING }

           [ ON { ASCENDING  } KEY data-name-3 [ , data-name-4 ] ... ] ...
           [    { DESCENDING }                                       ]

     [ COLLATING SEQUENCE IS alphabet-name ]

     { INPUT PROCEDURE IS section-name-1 [ { THROUGH } section-name-2 ] }
     {                                   [ { THRU    }                ] }
     { USING file-name-2 [ , file-name-3 ] ...                          }

     { OUTPUT PROCEDURE IS section-name-3 [ { THROUGH } section-name-4 ] }
     {                                    [ { THRU    }                ] }
     { GIVING file-name-4                                                }
```

Merging

Some data processing operations are performed on a group of records that are distributed on several files. If all of those files are themselves ordered by the same rules, their contents may be merged into one composite file which is ordered. A merging procedure manipulates two or more ordered input files and produces an output file containing the total set of data records in one ordered sequence.

MERGE STATEMENT

The MERGE statement combines two or more identically sequence files on a set of specified keys, and during the process makes records available, in merged order, to an output procedure or to an output file.

General Format:

```
MERGE file-name-1 ON { ASCENDING  } KEY data-name-1 [ , data-name-2 ] ...
                     { DESCENDING }

           [ ON { ASCENDING  } KEY data-name-3 [ , data-name-4 ] ... ] ...
           [    { DESCENDING }                                       ]

     [ COLLATING SEQUENCE IS alphabet-name ]

       USING file-name-2 , file-name-3 [ , file-name-4 ] ...

     { OUTPUT PROCEDURE IS section-name-1 [ { THROUGH } section-name-2 ] }
     {                                    [ { THRU    }                ] }
     { GIVING file-name-5                                                }
```

Ordering

The sequencing of the output file in a sorting or merging procedure is governed by the values of one or more fields in each of the records being ordered. These fields (the keys) must appear in the same position, relative to the start of the record, in every record being sorted or merged.

The order in which the keys are specified to the sort or merge procedure determines their hierarchical relationship. The first key named (the major key) is the most significant field. Each successive key specified is of decreasing significance until the last key (the most minor key) is reached.

Each key may also be specified as determining an ascending ordering or a descending ordering. Ascending ordering means that those records with lower values of that key appears in the output file prior to the records with higher values for that key. Descending ordering implies the inverse result.

Ordering is accomplished by comparing (from major to most minor) the corresponding keys of two records until an inequality of value is found. The output order is then determined by the ascending or descending rule which applies to that particular key field. If there is no inequality of value in any corresponding pair of keys, the ordering is determined by the sort or merge procedure.

Program Organization

The COBOL language contains two verbs which initiate ordering procedures, SORT and MERGE. Several additional language features are associated with both of these verbs. The RELEASE verb may be used with the execution of a SORT verb and the RETURN verb may be used with the execution of SORT and MERGE verbs. A special form of the SELECT clause in the FILE-CONTROL paragraph of the Environment Division is associated with the intermediate working files of the sort procedure and the implicit working file of the merge procedure. In addition, those files are described by a special type of file description (SD rather than FD) in the Data Division.

The SORT verb invokes the execution of a set of sorting procedures. These procedures operate with the COBOL object program to perform the sort. During the execution of the sorting function, the object program and sorting procedures combine in the organization pictured in Figure 9-1. The user can include several SORT statements in the source program. If several SORT statements are present, they are completely independent of each other.

The MERGE verb invokes the execution of a set of merging procedures contained within the standard software library. These procedures operate with the COBOL object program to perform the merge. During the execution of the merging function, the object program and merging procedures combine in the organization pictured in Figure 9-2. The user can include several MERGE statements in the source program. If several MERGE statements are present, they are completely independent of each other.

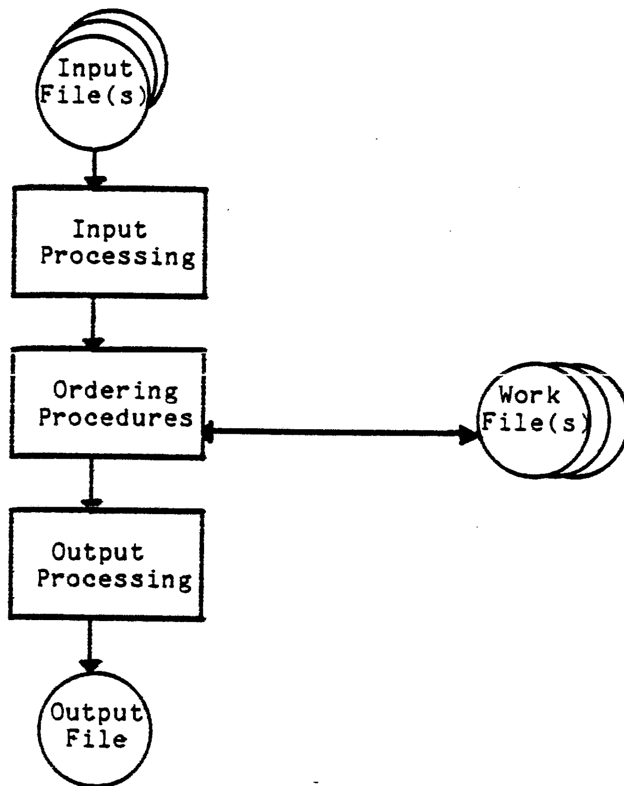


Figure 9-1. Sort Program Organization

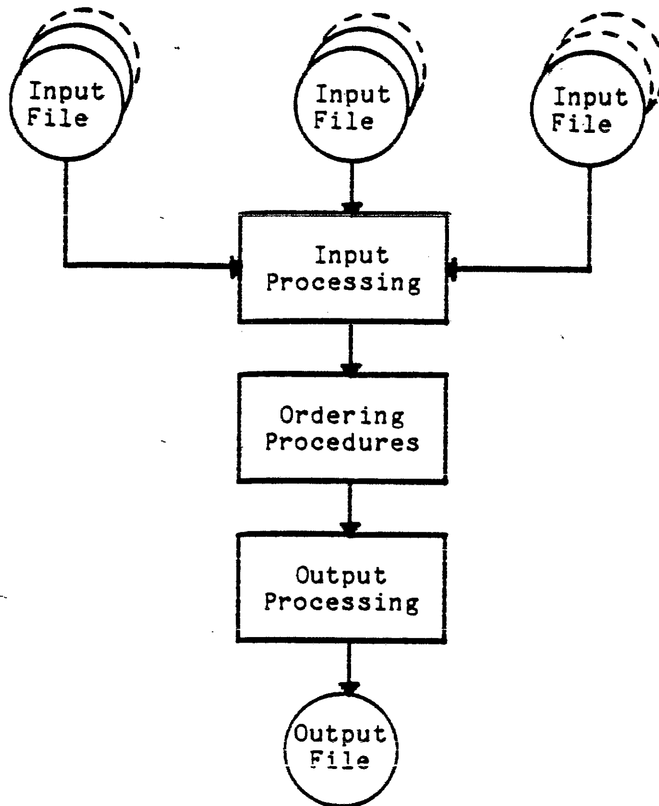


Figure 9-2. Merge Program Organization

SORT STATEMENT

The purpose of the SORT statement is to invoke the execution of a sorting procedure.

Sort File

The definition of the sort file serves two purposes:

1. It is the vehicle for defining the working files associated with the sorting procedure

2. It is the file description within which all of the keys for the ordering are described

The former role is required by both the syntax rules of the SORT statement and the sorting procedures. The size of the data file being sorted cannot be determined from within the sorting procedure prior to the inception of the procedure. The file space required by this procedure is allocated by the Multics SORT/MERGE module.

Sort file allocation occurs on two levels: within the source program, and at the system level. At the source program level, assignment is made in the FILE-CONTROL paragraph with the ASSIGN clause. One or more external-file-names are specified, each of them denoting one working file.

Sort Key Declarations

The second purpose of the sort file is to provide a vehicle for the definition of all the sort keys which determine the record ordering. This definition is accomplished in a special type of file description in the File Section of the Data Division.

The definition is prefaced with the level indicator SD. Each KEY data-name associated with a SORT statement must be defined within the sort file-name description referred to by the SORT statement. The keys are listed in the SORT statement in order, from most significant (major key) to least significant (most minor key), with the word ASCENDING or DESCENDING preceding KEY data-names as appropriate. Key comparison coding is generated by the COBOL compiler, rather than by the sorting procedure, on the basis of the key declarations in the SORT statement.

When more than one record description entry appears in a sort file description, the key data items need be described in only one of the record description entries. Each key data item must occur in every data record of the sort file. It must have the same relative position and actual format in all records. The PICTURE and USAGE of a given key data item must be the same in all records in the sort file. If a key item is synchronized or justified, it must be identically synchronized or justified in all records in the sort file. The key data item descriptions must not contain an OCCURS clause or be subordinate to entries containing an OCCURS clause. Keys must be data items that do not require subscripting or indexing.

VARIABLE-LENGTH RECORDS

Although key items themselves may not be of variable length, the records within the sort file may be of variable length. Each record must be large enough to contain the entire set of keys described in the SORT statement.

DOMINANT RECORD LENGTH

If the data being sorted is in variable-length record format, the sort procedure must also be given a dominant record size. The dominant record size is set equal to the size of the first record described in the sort file description entry (SD) in the Data Division. Whenever the efficiency of any variable-length record sort process needs to be increased, the dominant record length should be evaluated.

Sort Key Evaluation

When the values of a key in a pair of sort file records are compared, one value is found to be greater than, equal to, or less than the other according to the rules given under the "Comparison of Operands" heading in the Multics COBOL Reference Manual, Order No. AS44. The key comparison determines the order of the records in the sort output.

All comparisons are made on the basis of the ASCII character set unless the COLLATING SEQUENCE phrase of the SORT statement is used to specify a different sequence.

Sort Input Processing

A choice must be made between having the sorting process handle the input processing of the file being sorted or having the user's program specify the input processing procedures. In most cases the former technique is more efficient but the latter may be necessary in order to accomplish selective editing of the input records. If such editing would result in the deletion of a significant portion of the input file, then the input procedure technique is more appropriate.

USING OPTION

If the USING option is specified, the sorting process handles the input file processing. The file specified by file-name-2 must not be in an open state when the SORT statement is executed. File-name-2 must have a file description (FD) entry in the Data Division. The sort file and the file referenced in the USING phrase are, in a sense, alternative descriptions of the same set of data.

INPUT PROCEDURE OPTION

When the INPUT PROCEDURE option is specified, the user is responsible for all the input processing for the sorting process. An input procedure must consist of one or more sections. Since the input procedure is invoked by the sorting procedure via the same techniques used in the PERFORM statement execution, the structure of the input procedure must follow the same basic rules

as a set of sections which are the object of a PERFORM statement. Control must not be passed to the input procedure except through the execution of a SORT statement. The input procedure may contain any procedures needed to select, create, or modify records for input to the sorting process. Three general restrictions apply to the procedural statements within the input procedure:

1. An input procedure must not contain a SORT, MERGE, or RETURN statement.

2. The input procedure must not contain ALTER statements or transfers of control to points outside the input procedure. This means that the GO TO and PERFORM statements in the input procedure must not refer to procedure-names outside the input procedure. COBOL statements that cause an implied transfer of control to USE procedures are allowed.

3. The remainder of the Procedure Division must not contain ALTER statements or transfers of control to points inside the input procedure. This means that GO TO and PERFORM statements in the remainder of the Procedure Division must not refer to procedure-names within the input procedure.

RELEASE Statement

The RELEASE statement is used to transfer logical input records from an input procedure to the initial phase of a sorting operation. The RELEASE statement may appear only in an input procedure and every input procedure must contain at least one RELEASE statement.

The record-name referenced by the RELEASE statement must be the name of a record defined within a sort file; that is, a file described with an SD level indicator. If the sort file description contains more than one record description, and if the record descriptions define records of different sizes, a separate RELEASE statement must be specified for each record size. If the FROM phrase is used, the contents of 'identifier' must be the name of a data item in working-storage or of an input record area. If the format of identifier is different from that of the record-name, moving takes place according to the rules specified for the MOVE statement without the CORRESPONDING option. The information in the sort record area is no longer available, but the information in the identifier area is available. It is illegal to use the same name for both the record-name and the identifier or for the two names to reference the same memory area.

After the RELEASE statement is executed, the contents of record-name are no longer available to the COBOL procedure. The execution of a RELEASE statement causes the contents of record-name (after the contents of identifier have been moved to it in the FROM phrase), to be made available to the initial phase of the sort process. When control passes from the input procedure, the sort file consists of all those records which were placed in it by the execution of RELEASE statements. No OPEN, READ, WRITE, or CLOSE statements may be given for the sort file.

GIVING OPTION

If the GIVING option is specified, the sorting process handles the output file processing. The file specified by file-name-4 must not be in an open state when the SORT statement is executed. File-name-4 must have a file description (FD) entry in the Data Division. If both the USING and GIVING options are specified, file-name-2 and file-name-4 are, in a sense, alternative descriptions of the same set of data. Therefore, all file level properties must be identical for both files. File-name-2 and file-name-4 may refer to the same file-name or

to different file-names. In the special case in which USING and GIVING refer to the same file-name, the sort procedure does not use the GIVING file as a collation file.

OUTPUT PROCEDURE OPTION

When the OUTPUT PROCEDURE option is specified, the final output file processing for the sorting process is the responsibility of the object program. An output procedure must consist of one or more sections. Since the output procedure is invoked by the sorting procedure via the same techniques used in the execution of a PERFORM statement, the structure of the output procedure must follow the same basic rules as a set of sections which are the object of a PERFORM statement. Control must not be passed to the output procedure except through the execution of a SORT statement. The output procedure may contain any procedures needed to select, modify, or copy the records which are being returned from the sorting process. Three general restrictions apply to the procedural statements within the output procedure:

1. An output procedure must not contain a SORT, MERGE, or RELEASE statement.

2. The output procedure must not contain ALTER statements or transfers of control to points outside the output procedure; i.e., GO TO and PERFORM statements in the output procedure are not permitted to refer to procedure-names outside the output procedure. COBOL statements that cause an implied transfer of control to USE procedures are allowed.

3. The remainder of the Procedure Division must not contain ALTER statements or transfers of control to points inside the output procedure; i.e., GO TO and PERFORM statements in the remainder of the Procedure Division are not permitted to refer to procedure-names within the output procedure.

RETURN Statement

The RETURN statement is used to obtain logical output records from a sort operation and to transfer them to an output procedure. The RETURN statement may appear only in an output procedure, and every output procedure must contain at least one RETURN statement.

The file-name referenced in the RETURN statement must be described with an SD level indicator and must be the same file-name that was referenced in the SORT statement currently being executed.

The INTO phrase may be used only when the file referred to by file-name contains just one type of record. The identifier must be the name of a data item in working-storage or of an output record area. If the format of the identifier differs from that of the input record, moving is performed according to the rules specified for the MOVE statement without the CORRESPONDING option. When the INTO phrase is used, the logical record is still available in the sort file's record area.

The execution of the RETURN statement causes the next record in sorted order (according to the keys listed in the SORT statement) to be made available for processing in the record area associated with the sort file.

After the contents of the sort file are exhausted, the next execution of the RETURN statement results in the execution of the imperative statement in the AT END phrase.

After execution of the AT END imperative-statement, no RETURN statement may be executed within the current output procedure.

No OPEN, READ, WRITE, or CLOSE statements may be given for the sort file.

SORT OPERATIONAL CONSIDERATIONS

Flow of Control

Any sequence of procedural statements may be executed before or after the SORT statement is executed. When the SORT statement is executed, the sorting process receives control.

If an input procedure has been specified, the sort transfers control to the input procedure to start processing records. The input procedure is responsible for opening the input file and reading the logical input records. Each time that a record is ready for the sort file, the input procedure executes a RELEASE statement, causing the sort to place the record in the sort file. Control then passes to the statement following the RELEASE statement. The input procedure continues reading and releasing until all the input records have been given to the sort, at which time the input procedure is responsible for closing the input file. Control must pass to the exit point of the input procedure, thereby returning control to the sorting process. A simple input procedure might be organized as in Figure 9-3.

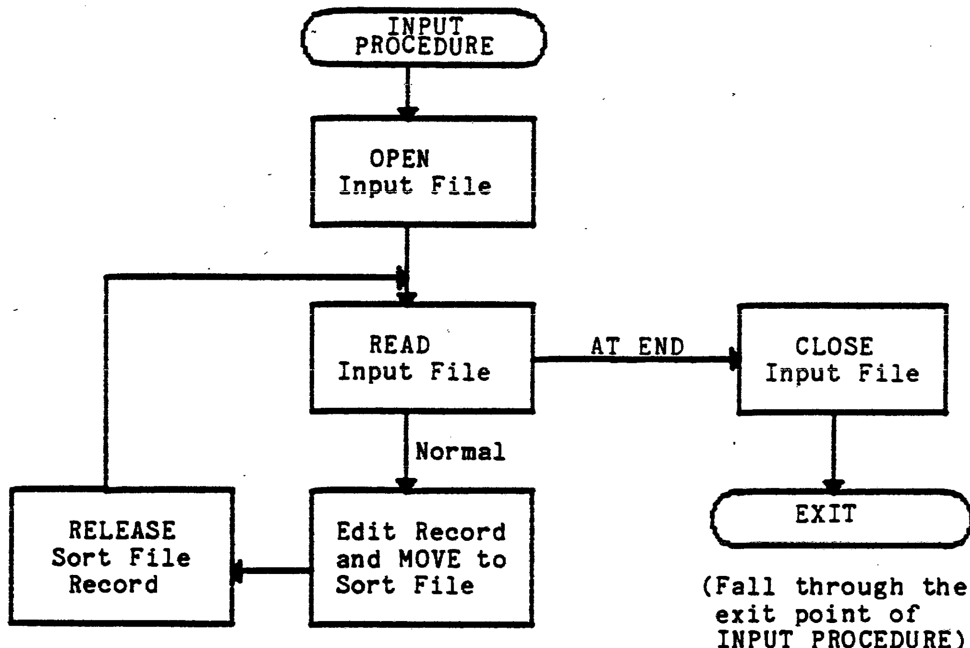


Figure 9-3. Sort Input Procedure Organization

The sorting process orders the records, up to the point of determining which record goes first in the final output sequence. If an output procedure has been specified, the sorting process at this point transfers control to the output procedure. The output procedure is responsible for opening its output file, if any, and obtaining records in the final sequence by means of the RETURN statement. When the output procedure has disposed of each record, it returns the next, and thus continues returning records and processing them. After the last record has been returned, the sorting process causes control to pass to the AT END phrase the next time a RETURN statement is executed. The output procedure is then responsible for closing its output file, if any, and allowing control to pass to its exit point. The sorting process then terminates its own procedures. Control then passes to the statement following the SORT statement. A simple output procedure might be organized as in Figure 9-4.

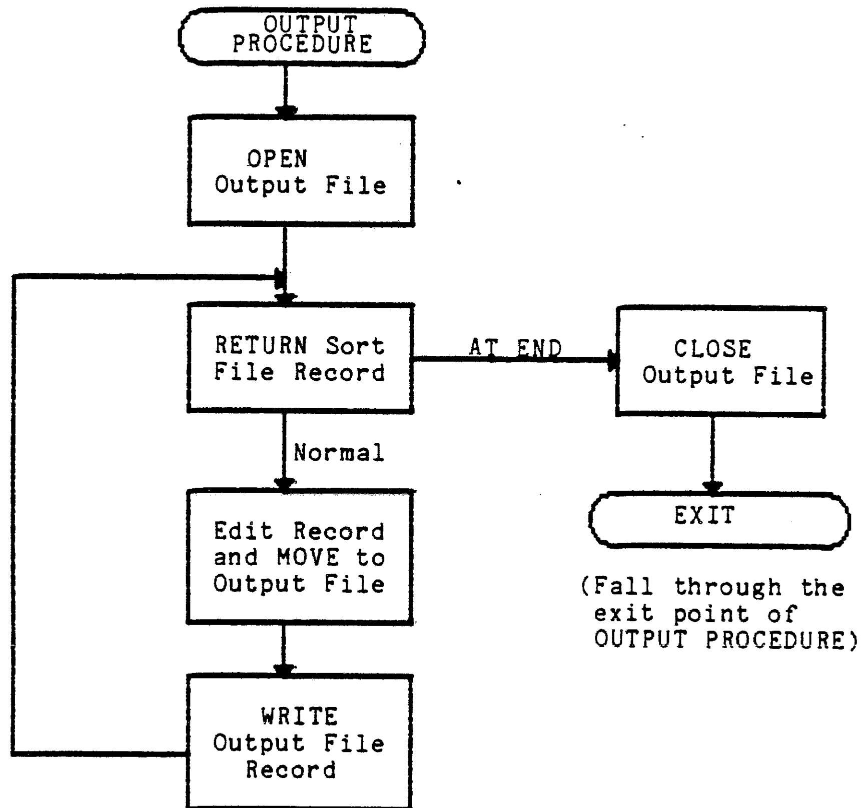


Figure 9-4. Sort Output Procedure Organization

In effect, the sequence of events just described applies also when the USING or GIVING option is used, except that the input or output procedure becomes implicit, rather than specified in detail by the user.

Sort Examples

The following example illustrates a basic SORT program:

```
        .
        .
        .
    environment division.
    file-control.
        select  input-file  assign  to  input-virtual.
        select  sort-file  assign  to  collate-virtual.
        select  output-file  assign  to  output-printer.
        .
        .
        .
    data division.
    file section.
    fd  input-file...
    01...
        .
        .
    sd  sort-file...
    01...
        .
        .
    fd  output-file...
    01...
        .
        .
    procedure  division.
    sort-call.  sort  sort-file  on  ... using
        input-file  giving  output-file.
        stop  run.
```

Note that these files may have multiple record types and sizes, provided the sort file and USING file have the same records, the sort file and GIVING file have the same records, and key descriptions and positions are equivalent for all record types.

Another SORT feature entails the use of an output procedure to deliver a report (on any suitable device) rather than an output tape as such:

```
        .
        .
        .
    environment  division.
    file-control.
        select input-file assign to inputfile-virtual.
        select sort-file assign to sortfile-virtual.
        select report-output assign to report-printer.
    data division.
    file section.
    fd  input-file...
    01...
        .
        .
    sd  sort-file...
    01...
        .
        .
    fd  report-output;  report  is  xyz...
        .
        .
        .
    working-storage  section.
        .
        .
        .
    report  section.
    rd  xyz...
```

```
01  detail-line; type de...
    .
    .
    .
procedure division.
sort-call section.
driver. sort sort-file on ... using input-file
    output procedure is edit. stop run.
edit section.
startup. open report-output; initiate xyz.
loop. return sort-file record; at end go to quit.
    .
    .
    .
    generate detail-line; go to loop.
quit. terminate xyz; close report-output.
```

MERGE STATEMENT

The purpose of the MERGE statement is to invoke the execution of a merging procedure.

The Merge File

The definition of the merge file serves only to provide a location where all the keys for the ordering are described. There is no application of the merge file which corresponds to the working file status of the sort file. However, COBOL syntax rules require that an appropriate SELECT clause be specified for the merge file. A file assignment control card should not be included for the file code assigned to the merge file.

Merge Key Declarations

The definition of the merge key fields is accomplished in a special type of file description in the File Section of the Data Division. The definition is prefaced with the level indicator SD. Each KEY data-name associated with a MERGE statement must be defined within the merge file-name description referred to by the MERGE statement. The keys are listed in the MERGE statement in order, from most significant (major key) to least significant (most minor key), with the word ASCENDING or DESCENDING preceding KEY data-names as appropriate. Key comparison coding is generated by the COBOL compiler, rather than by the merging procedure, on the basis of the key declarations in the MERGE statement.

Each key item must occur in every data record of the merge file. It must have the same relative position and actual format in all records. The PICTURE and USAGE of a given key item must be the same in all records in the merge file. If a key item is synchronized or justified, it must be identically synchronized or justified in all records in the merge file. Keys must be data items which do not require subscripting or indexing.

VARIABLE-LENGTH RECORDS

Although key items themselves may not be of variable length, the records within the merge file may be of variable length. Each record must be large enough to contain the entire set of keys described in the MERGE statement.

Merge Key Evaluation

When the values of a key in a pair of merge file records are compared, one value is found to be greater than, equal to, or less than the other according to the rules given under the "Comparison of Operands" heading in the Multics COBOL Reference Manual, Order No. AS44. The key comparison determines the order of the records in the merge output.

All comparisons are made on the basis of the ASCII character set unless the COLLATING SEQUENCE phrase of the MERGE statement is used to specify a different sequence.

Merge Input Processing

When a merging procedure is used, the merging process handles all of the processing of input files. The USING phrase must be specified in the MERGE statement and must identify at least two file-names. The referenced input files must not be in an open state when the MERGE statement is executed. All of the specified file-names must have file description (FD) entries in the Data Division. The merge file and the files referenced in the USING phrase are, in a sense, alternative descriptions of the same set of data.

Merge Output Processing

A choice must be made between having the merging process handle the output processing of the newly merged file or having the user's program specify the output processing procedures.

GIVING OPTION

If the GIVING option is specified, the merging process handles the output file processing. The file specified by file-name-5 must not be in an open state when the MERGE statement is executed. File-name-5 must have a file description (FD) entry in the Data Division. File-name-5 and the files specified in the USING phrase are, in a sense, alternative descriptions of the same set of data. Therefore, all file level properties must be identical for both files.

OUTPUT PROCEDURE OPTION

When the OUTPUT PROCEDURE option is specified, the final output file processing for the merging process is the responsibility of the object program. An output procedure must consist of one or more sections. Since the output procedure is invoked by the merging process via the same techniques used in the execution of a PERFORM statement, the structure of the output procedure must follow the same basic rules as a set of sections which are the object of a PERFORM statement. Control must not be passed to the output procedure except through the execution of a MERGE statement. The output procedure may contain any procedures needed to select, modify, or copy the records which are being returned from the merging process. Three general restrictions apply to the procedural statements within the output procedure:

1. An output procedure must not contain a MERGE, SORT, or RELEASE statement.

2. The output procedure must not contain ALTER statements or transfers of control to points outside the output procedure; i.e., GO TO and PERFORM statements in the output procedure are not permitted to refer to procedure-names outside the output procedure. COBOL statements that cause an implied transfer of control to USE procedures are allowed.

3. The remainder of the Procedure Division must not contain ALTER statements or transfers of control to points inside the output procedure; i.e., GO TO and PERFORM statements in the remainder of the Procedure Division are not permitted to refer to procedure-names within the output procedure.

RETURN Statement

The RETURN statement is used to obtain logical output records from a merge operation and to transfer them to an output procedure. The RETURN statement may appear only in an output procedure, and every output procedure must contain at least one RETURN statement.

The file-name referenced in the RETURN statement must be described with an SD level indicator and must be the same file-name that was referenced in the MERGE statement currently being executed.

The INTO phrase may be used only when the file referred to by file-name contains just one type of record. The identifier must be the name of a data item in working-storage or of an output record area. If the format of the identifier differs from that of the input record, moving is performed according to the rules specified for the MOVE statement without the CORRESPONDING option. When the INTO phrase is used, the logical record is still available in the merge file's record area.

The execution of the RETURN statement causes the next record in merged order (according to the keys listed in the MERGE statement) to be made available for processing in the record area associated with the merge file.

After the contents of the merge file are exhausted, the next execution of the RETURN statement results in the execution of the imperative statement in the AT END phrase.

After execution of the AT END imperative-statement, no RETURN statement may be executed within the current output procedure.

No OPEN, READ, WRITE, or CLOSE statements may be given for the merge file.

MERGE OPERATIONAL CONSIDERATIONS

Flow of Control

Any sequence of procedural statements may be executed before or after the MERGE statement is executed. When the MERGE statement is executed, the merging process receives control.

The merging process performs initial housekeeping, up to the point of determining which record is placed first in the final output sequence. If an output procedure has been specified, the merging process at this point transfers control to the output procedure. The output procedure is responsible for opening its output file, if any, and obtaining records in the final sequence by means of the RETURN statement. When the output procedure has disposed of each record, it returns the next, and thus continues returning records and processing them. After the last record has been returned, the merging process causes control to pass to the AT END phrase the next time a RETURN statement is executed. The output procedure is then responsible for closing its output file and allowing control to pass to its exit point. The merging process then terminates its own procedures. Control then passes to the statement following the MERGE statement. A simple output procedure might be organized as in Figure 9-5.

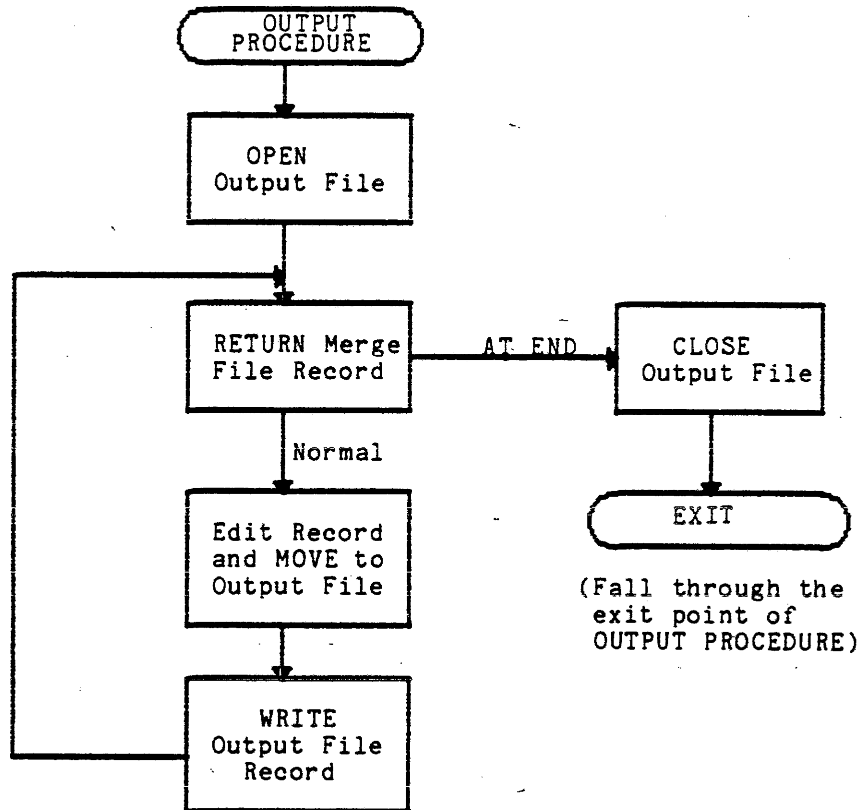


Figure 9-5. Merge Output Procedure Organization

In effect, the sequence of events just described applies also when the GIVING option is used, except that the output procedure becomes implicit, rather than specified in detail by the user.

Merge Examples

The following example illustrates a basic MERGE program:

```
     .
     .
     .
environment division.
file-control.
    select  input-file-1  assign  to  input1-virtual.
    select  input-file-2  assign  to  input2-virtual.
    select  merge-file  assign  to  mergefile-virtual.
    select  output-file  assign  to  outputfile-printer.
i-o-control.
     .
     .
     .
data  division.
file  section.
fd  input-file-1...    ┐
01...                  │
     .                 │
     .                 │
fd  input-file-2...    │
01...                  │  Note that these files may have multiple
     .                 │  record  types  and  sizes, provided the
     .                 }  merge file and USING file have the same
sd  merge-file...      │  records, the merge file and GIVING file
01...                  │  have the same records, and key descrip-
     .                 │  tions and positions are equivalent  for
     .                 │  all record types.
fd  output-file...     │
01...                  │
     .                 │
     .                 ┘
procedure  division.
merge-call.  merge  merge-file  on  ...  using  input-file-1,  input-file-2,
    giving  output-file.
```

Another MERGE feature entails the use of an output procedure to deliver a report (on any suitable device) rather than an output tape as such:

```
     .
     .
     .
environment  division.
file-control.
    select  input-file-1  assign  to  input1-virtual.
    select  input-file-2  assign  to  input2-virtual.
    select  merge-file  assign  to mergefile-virtual.
    select  report-output  assign  to  report-printer.
data  division.
file  section.
fd  input-file-1...
01...
     .
     .
fd  input-file-2...
01...
     .
     .
sd  merge-file...
01...
     .
     .
```

```
fd  report-output,  report  is  xyz...
        .
        .
        .
working-storage  section.
        .
        .
        .
report  section.
rd  xyz...
01  detail-line,  type  de...
        .
        .
        .
procedure  division.
merge-call  section.
driver.  merge  merge-file  on  ...  using
        input-file-1,  input-file-2,  output
        procedure  is  edit.
        stop  run.
edit  section.
startup.  open  report-output,  initiate  xyz.
loop.  return  merge-file  record,  at  end
      go  to  quit.
      .
      .
      .
      generate  detail-line,  go  to  loop.
quit.  terminate  xyz,  close  report-output.
```

WORK REQUIREMENTS

The SORT and MERGE statements require work files to be allocated in the Multics storage system. Thus the user must have sufficient quota for the work files in addition to that required for the output file, if the output file is to be in the storage system.

Sort Work Files

The SORT statement requires a number of large segments which are allocated in the directory specified by the user. As a first approximation, the space required by these segments is between 1.05 and 1.15 times the total size of all the input files.

A closer approximation to the size of the SORT work files is:

F + 64*sqrt(F)

where F is the total amount of data input to the SORT, in bytes. The MERGE function does not require these work files.

Process Directory Work Files

Both the SORT and the MERGE statements require a number of segments which are always allocated in the user's process directory. As a first approximation, the space required by these segments is from three to six storage system records (1024 words each). For additional detail, refer to the Multics SORT/MERGE manual, Order No. AW32.

Running COBOL Programs with the SORT Statement

When executing COBOL programs containing SORT or MERGE statements, it is recommended that the run_cobol command be used with the -sort_file_size argument. If work files for the SORT statement are expected to exceed the process directories quota, the argument -sort_dir should also be used. For additional detail, refer to the run_cobol command in the MPM Commands manual.

SECTION X

DEBUG FACILITY

DESCRIPTION OF THE DEBUG FACILITY

The debugging facility described in this section is defined by the ANS COBOL Standard. Changes must be made to a source program if the facility is to be used. Multics also offers powerful symbolic debugging facilities (probe and debug) which can be applied to a program without making source changes. These facilities are described in Section VI of this manual and in the Multics Programmers Manual - Commands and Active Functions, Order Number AG92.

The debug facility assists in error detection by:

1. Monitoring transfers of control to user-selected procedures during program execution.

2. Monitoring values of user-selected data items during program execution.

The user-supplied statements required to accomplish such monitoring are included in the source program and can be compiled or not depending on the presence or absence of the DEBUGGING MODE clause in the source program. After the user statements are compiled into the program, they can be executed or ignored at object program execution according to the setting of a run-time switch. The run-time switch is set 'ON' when the program is executed by using a run_cobol command with the -debug option. (The run_cobol command is described in Section V.) The decisions concerning what to monitor and what information to display on the output device are at the discretion of the user. The main purpose of the COBOL debug facility is to provide convenient access to such information.

EXAMPLE OF THE DEBUG FACILITY

The following example contains brief program segments that illustrate the usage of the debug facility.

Example:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  DEBUG-EXAMPLE.
ENVIRONMENT  DIVISION.
CONFIGURATION  SECTION.
SOURCE-COMPUTER.
    MULTICS  WITH  DEBUGGING  MODE.
.
.
```

Example (cont.):

```
     01  ITEM-1.
         02  KEY-1  PIC  99.
         02  LINE-1  PIC  X(6).
         02  NAME-1  PIC  X(30).
         02  UNQUAL-NAME-1  PIC  X(30).
         02  SUB-1-1  PIC  X(5).
         02  SUB-2-1  PIC  X(5).
         02  SUB-3-1  PIC  X(5).
         02  CONTENTS-1  PIC  X(30).
     01  ITEM-2.
         02  KEY-2  PIC  99.
         02  LINE-2  PIC  X(6).
         02  NAME-2  PIC  X(30).
         02  UNQUAL-NAME-2  PIC  X(30).
         02  CONTENTS-2  PIC  X(30).
      .
      .
     PROCEDURE  DIVISION.
     DECLARATIVES.
     GO-TO-DEPENDING-PROC-1  SECTION.
         USE  FOR  DEBUGGING  ON  ALL  REFERENCES  OF  GO-TO-DEP-KEY
                                  ALL                 ID-2.
     GO-TO-DEPEND-1.
         ADD  1  TO  KEY-1.
     DB-COMMON-1.
         MOVE  DEBUG-LINE  TO  LINE-1.
         MOVE  DEBUG-NAME  TO  NAME-1  UNQUAL-NAME-1.
         MOVE  DEBUG-CONTENTS  TO  CONTENTS-1.
     DB-CLEAR-QUALIFIER-1.
         INSPECT  UNQUAL-NAME-1  REPLACING  CHARACTERS  BY  SPACES
             AFTER  INITIAL  SPACE.
     GO-TO-DEPENDING-PROC-2  SECTION.
         USE  FOR  DEBUGGING  ON  G-T-D-2.
     GO-TO-DEPEND-2.
         IF  KEY-1  IS  EQUAL  TO  1
             MOVE  2  TO  KEY-2
             ELSE  MOVE  1  TO  KEY-2.
     DB-COMMON-2.
         MOVE  DEBUG-LINE  TO  LINE-2.
         MOVE  DEBUG-NAME  TO NAME-2  UNQUAL-NAME-2.
         MOVE  DEBUG-CONTENTS  TO  CONTENTS-2.
     DB-CLEAR-QUALIFIER-2.
         INSPECT  UNQUAL-NAME-2  REPLACING  CHARACTERS  BY  SPACES
             AFTER  INITIAL  SPACE.
     GO-TO-DEPENDING-PROC-3  SECTION.
         USE  FOR  DEBUGGING  ON  GO-TO-DEP-KEY-1.
     GO-TO-DEPEND-3.
         MOVE  1  TO  KEY-1.
     PERFORM-PROC-1  SECTION.
         USE  FOR  DEBUGGING  ON  ID-1.
     PERFORM-1.
         ADD  1  TO  KEY-1.
         PERFORM  DB-COMMON-1.
         PERFORM  DB-CLEAR-QUALIFIER-1.
     TABLE-PROC-1  SECTION.
         USE  FOR  DEBUGGING  ON  B-LEVEL-1  B-LEVEL-2  B-LEVEL-3.
     TABLE-1.
         MOVE  1  TO  KEY-1.
         PERFORM  DB-COMMON-1.
         PERFORM  DB-CLEAR-QUALIFIER-1.
```

Example (cont.):

```
     DB-MOVE-SUBSC-1.
         MOVE  DEBUG-SUB-1  TO  SUB-1-1.
         MOVE  DEBUG-SUB-2  TO  SUB-2-1.
         MOVE  DEBUG-SUB-3  TO  SUB-3-1.
     QUAL-PROC-1  SECTION.
         USE  FOR  DEBUGGING  ON  ALL  REFERENCES  OF  ABC1  OF  AB2  OF  A1
                                  ALL                             AB2  OF  A2
                                  ALL                             AB1  OF  A1.
     QUAL-1.
         MOVE  1  TO  KEY-1.
         PERFORM  DB-COMMON-1.
         PERFORM  DB-MOVE-SUBSC-1.
     END  DECLARATIVES.
     .
     .
```

SECTION XI

REPORT WRITER

DESCRIPTION OF THE REPORT WRITER

The Report Writer feature emphasizes the organization, format, and contents of an output report. Although a report can be produced using the standard COBOL language, the Report Writer language characteristics provide a more concise method for report structuring and report production. Much of the Procedure Division coding which would normally be supplied by the user is instead provided automatically by the Report Writer Control System. Thus, the user is relieved of writing procedures for moving data, constructing print lines, counting lines on a page, numbering pages, producing heading and footing lines, recognizing the end of logical data subdivisions, updating SUM counters, etc. All of these operations are accomplished by the Report Writer Control System from source language statements that appear primarily in the Report Section of the Data Division of the source program.

A hierarchy of levels is used to define the logical organization of a report. Each report is divided into report groups, which in turn are divided into sequences of items. This hierarchical structure permits explicit reference to a report group with implicit reference to other levels in the hierarchy. A report group contains one or more items to be presented on one or more lines.

Data movement to a report description is directed by the Report Section clauses SOURCE, SUM, and VALUE. Fields of data are positioned on a print line by means of the COLUMN NUMBER clause. The PAGE clause specifies the length of the page, the size of the heading and footing areas, and the size of the area in which the detail lines will appear. Data items may be specified to form a control hierarchy. During the execution of a GENERATE statement, the Report Writer Control System uses the control hierarchy to check automatically for control breaks. When a control break occurs, summary information (SUM counters) can be presented.

Report Format

A report may consist of any meaningful combination of the following syntax selections:

- REPORT HEADING (one for each report)
- PAGE HEADING (one format for each report)

- CONTROL HEADING (one format for each control level)
- DETAIL (no limit for each report)
- CONTROL FOOTING (one format for each control level)
- PAGE FOOTING (one format for each report)
- REPORT FOOTING (one for each report)

In COBOL, each report is described in the Report Section of the Data Division. The user specifies the intended format for each of the headings, footings, and detail lines in the report, as well as all sources of data. A report may utilize data described in the File Section, Working-Storage Section, and Linkage Section. In addition, the user specifies the overall organization and intended page layout of the report.

The compiler provides the following functions in the object program:

1. Vertical format control, including line counting, page counting, and production of page headings and footings.
2. Detection of control breaks.
3. Production of control headings and footings.
4. Accumulation of SUM counters to any number of control levels.
5. Execution of user-defined procedures before presentation of nondetail report groups.

Report Control in the Procedure Division

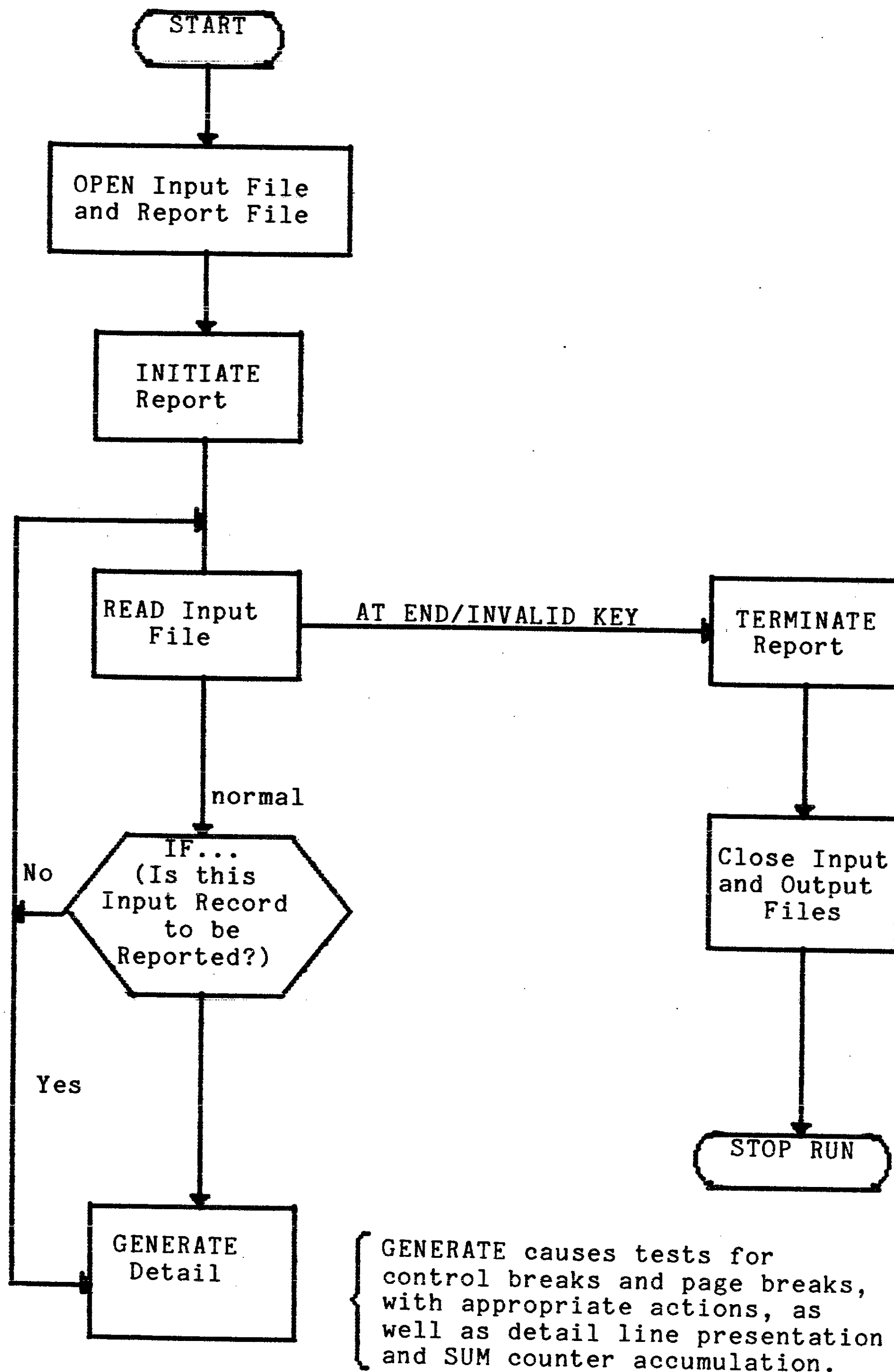
The production of a report is controlled in the Procedure Division with four report writing statements:

- INITIATE
- GENERATE
- SUPPRESS
- TERMINATE

In addition, the BEFORE REPORTING phrase of the USE statement may also be used to control the production of a report.

The SUPPRESS statement inhibits the presentation of a report group and may be specified only in a USE BEFORE REPORTING procedure.

A possible relationship of the above statements to other Procedure Division statements is illustrated by the following flow chart of a simple reporting program:



Before a GENERATE statement is executed, the report must be initiated. The INITIATE statement causes initial housekeeping values to be established.

The GENERATE statement provides for all aspects of report editing, writing, and housekeeping, but GENERATE in itself makes no provision for reading input data or deciding when detail lines should be produced. Instead, the user explicitly obtains each input record via COBOL statements such as the READ statement.

When the last GENERATE statement has been executed, the report must be terminated. The TERMINATE statement causes final control footings and report footings to be presented.

The immediate destination of a report is always a file specified in the File Section of the Data Division. The file must be explicitly opened prior to execution of the report's INITIATE statement, and the file must be explicitly closed after the TERMINATE. The report writing statements implicitly perform whatever writing is required for the report.

Skeletal Format for the Report Section

The definition of each report includes two types of entries:

1. The RD entry specifies the basic page layout and the overall organization of the report.

2. Report group description entries give the detailed formats of all elements of the report and the sources of all information for the report.

An RD entry in the Report Section is analogous to an FD entry in the File Section; it is the highest level of hierarchical organization for the report. The report-name specified in each RD entry must be unique.

A level 01 report group description entry is analogous to a level 01 data record description entry in the File Section. A level 01 report item is called a report group. Normally, the hierarchical definition of the report group is completed with a series of subordinate entries with levels 02-49.

An item with no subordinate items is an elementary item. Any report item whose entry is followed by subordinate entries is a group item.

Since several reports may be defined in the Report Section, the skeletal format of the Report Section is as follows:

```
REPORT  SECTION.
RD  report-name-1...      }
01  report-group-name...  }  Complete
    02...                 }  description of
    .                     }  first report
    .                     }
01...
.
.
RD  report-name-2...
01...
.
.
RD  report-name-n...
.
.
```

For detailed reporting, a GENERATE statement refers to the data-name of a level 01 detail report group. For summary reporting, GENERATE refers to the report-name of an RD entry instead of a detail report group data-name. The order in which level 01 report groups are specified for a given report is not significant.

Within each report group, items to be printed must be described from left to right. If the report group contains multiple lines, they must be described in order from top to bottom.

The length of each line is determined by the user. In the formatting of a print line, spaces are assumed except where a specific item is to be printed. (In a data record, on the other hand, every character position must be described.)

RD Entries

The description of each report begins with an RD entry. Except for the level indicator (RD) and the report-name, all clauses in an RD entry are optional.

The optional clauses in an RD entry are:

Clause	Function
CODE	To assign a two character nonnumeric label to each line of this report on intermediate storage. (The code character does not appear in the printed report.)
CONTROL(S)	To specify data-names of control items, in the order from most significant to least significant.
PAGE LIMIT(S)	To specify the maximum number of lines per page.
HEADING	To specify the line number at which page headings may begin.
FIRST DETAIL	To specify the line number at which detail and control lines may begin.
LAST DETAIL	To specify the line number beyond which detail and control heading lines must not be printed.
FOOTING	To specify the line number beyond which control footing lines must not be printed. Also, to specify the line number on or before which a page footing must not be printed.

Report Group Entries

Typically, the description of a report includes two or more level 01 report group entries, each followed by a hierarchy of subordinate entries. Depending upon a number of factors, most clauses (except the level-number clause) are optional. In most entries, the data-name is optional and is normally omitted. A data-name is specified in level 01 detail report group entries and in nondetail report groups that are referenced by the BEFORE REPORTING phrase of the USE statement.

At the 01 report group level, the following clause is required:

Clause	Function
TYPE	To specify the purpose of this report group (detail, page or control heading, etc.).

The optional clauses in a report group entry are:

Clause	Function
LINE NUMBER	To specify vertical spacing (slewing) that is to precede production of this report group (pre-slew).
NEXT GROUP	To specify vertical spacing that is to follow production of this report group (post-slew).
COLUMN NUMBER	To indicate that this item is to be printed, and to specify its horizontal position on the line.
BLANK WHEN ZERO	To cause this item's value to be 'spaces' when the SOURCE or SUM with which it is associated has the value zero.
GROUP INDICATE	To cause a repetitive item to be printed only at the top of the page and just after each control break.
JUSTIFIED RIGHT	To override normal left justification when this item is edited for output.
PICTURE	To specify the desired output format for this item.
RESET	To specify the control break where a SUM counter is to be reset to zero.
SOURCE, SUM, or VALUE	To specify the source of data for this item: 1. SOURCE - a data item. 2. SUM - a SUM counter. 3. VALUE - a literal.
USAGE	To declare the usage of printable items.

ELEMENTS OF A REPORT

Report Groups

Each integral unit of data presented in a report, such as a page heading or footing, control heading or footing, or detail line is called a report group. A report group may consist of one or several actual lines in the printed report. In the Report Section, the first entry of each report group must be level 01.

The TYPE clause is a required part of each level 01 report group description entry. The TYPE clause identifies the report group as detail or as report, page, or control heading or footing. Each report must contain at least one TYPE DETAIL report group. All other types are optional. A given heading type may be used with or without the corresponding footing, and vice versa. A report may have several distinct detail report groups or control heading or footing report groups, but no more than one of each of the other types.

Control Data Items

Each control heading or footing is associated with a specific control data item. A control item may be any item described in the File Section, Working-Storage Section, or Linkage Section.

Control items are related to the report by a list of control data-names specified in the CONTROL(S) clause of the RD entry. When control items are specified, the reporting procedures in the object program automatically monitor all control items for changes in value.

The most significant possible control level is associated with the reserved word FINAL, which may optionally be specified in the RD entry's CONTROL(S) clause and in a control heading and/or control footing report group description entry.

Any control item may be associated with a specific control heading and/or a specific control footing report group. (Control report groups may be specified for each control item, for none, or for any subset of control items.) Control footings may call for automatic accumulation of SUM counters.

Control heading report groups are presented in the following hierarchical order:

```
FINAL CONTROL HEADING
MAJOR CONTROL HEADING
          .
          .
MINOR CONTROL HEADING
```

Control footing report groups are presented in the following hierarchical order:

```
MINOR CONTROL FOOTING
          .
          .
MAJOR CONTROL FOOTING
FINAL CONTROL FOOTING
```

A control break is recognized whenever a control item has changed in value between the execution of the previous GENERATE statement and the current GENERATE statement.

If the item producing a control break is not the least significant (rightmost) item in the list of all control items, then a control break has occurred at all less significant levels as well.

A control break causes the following automatic actions:

1. Rolling forward and crossfooting SUM counters.

2. Presentation of control footings up through the control break level.

3. Resetting SUM counters to zero, up through the control break level, unless inhibited by a RESET phrase.

4. Presentation of control headings from the control break level down through the least significant control level.

When it is specified, a final control heading is presented only once for the report, upon the first execution of a GENERATE statement. Similarly, a final control footing is presented only once, upon execution of the TERMINATE statement.

When the TERMINATE statement is executed, a final control break is understood to have occurred, so all control footing report groups are then presented, in order from least significant through final.

When a control break occurs, SOURCE clauses in control footings must reflect the old values of the control data items, while SOURCE clauses in control headings reflect the new values of the control data items. A special provision causes old control item values to be retained for control footings. No such provision exists for items which are not control items.

File Characteristics

Each report is produced on an output file by the object program. The output file must be described by at least an FD entry in the File Section of the Data Division plus associated Environment Division paragraphs and phrases.

A given output file may receive one or more reports. The REPORT(S) clause of the FD entry lists the report or reports belonging to the file. This is the only explicit relationship between a report and the file to which it belongs. (Although a GENERATE statement does not mention the output file, any necessary 'writes' to the file are implied by each execution of a GENERATE statement.)

For a file receiving multiple reports, it is necessary to label each report line uniquely so that the lines belonging to the respective reports can be distinguished for printing. The CODE clause of the RD entry is used for that purpose. With the CODE clause, the user can associate a unique two-character nonnumeric literal with each report; the compiler then causes every line of the report to be labeled in a standard manner with the unique characters assigned by the user. The code characters appear in the intermediate file only. The command, process_cobol_report, is used to obtain the printed report from the intermediate file.

Line Counter

A line counter is implicitly provided for each report. It is used by the generated reporting procedures to recognize page breaks, and to control vertical page format.

The line counter is automatically set to zero initially, and it is reset to zero whenever a page break occurs. It is automatically set, reset, and incremented on the basis of values specified in the LINE NUMBER and NEXT GROUP clauses in the respective report groups. It is automatically tested on the basis of values specified in the PAGE clause of the RD entry.

A page break occurs whenever a relative LINE NUMBER or relative NEXT GROUP value causes the line counter to exceed the value specified in the PAGE clause.

The reserved word LINE-COUNTER may be referred to if it is necessary to access the line counter contents. The report-name may be used as a qualifier for LINE-COUNTER; such qualification is necessary whenever the Report Section includes more than one report.

If the last line produced has no relevant NEXT GROUP clause, the line counter value is the number of the last line printed. Otherwise, the line counter value is the number of the last line skipped.

Page Counter

A page counter is implicitly provided for each report. It is primarily used as a SOURCE data item within page heading or page footing report groups, to provide consecutive page numbers for the report.

The initial value of the page counter is one. Its value is automatically incremented by one each time a page break occurs. (The increment follows production of any page footing, but precedes production of any page heading.)

The reserved word PAGE-COUNTER may be referenced in a SOURCE clause or in the Procedure Division to access the page counter value. The report-name may be used as a qualifier for PAGE-COUNTER; such qualification is necessary whenever the Report Section includes more than one report.

Normally, Procedure Division statements should not change the value of a page counter. However, a Procedure Division statement may change the starting value of a page counter if an initial page number other than one (1) is desired.

SUM Counter Manipulation

A function of the Report Writer that must be clarified to avoid producing inefficient object code is the manipulation of SUM counters. There are three distinct types of SUM counter manipulation; subtotalling, rolling forward, and crossfooting. A definition and illustration of each type of manipulation is presented below.

SUBTOTALLING

Subtotalling is the most basic type of SUM counter manipulation. In this method, a SUM counter is augmented by the value of the SUM operand for each execution of a GENERATE statement of the TYPE DETAIL report group which contains the SOURCE counterpart of the SUM operand.

Example:

```
01  DETAIL-1  TYPE  DETAIL  LINE  PLUS  1.
    02  SOURCE  IS  COST.
    .
    .
01  MINOR  TYPE  CF  MINR  LINE  PLUS  1.
    02  SCTR-1  COLUMN  50  PIC Z(6).99  SUM  COST.
    .
    .
01  INTERMEDIATE  TYPE  CF  INTRM  LINE  PLUS  1.
    02  SCTR-2  COLUMN  50  PIC Z(6).99  SUM  COST.
    .
    .
01  MAJOR  TYPE  CF  MAJR  LINE  PLUS  1.
    02  SCTR-3  COLUMN  50  PIC Z(6).99  SUM  COST.
    .
    .
01  FIN-TOT  TYPE  CF  FINAL  LINE  PLUS  1  NEXT  GROUP  NEXT  PAGE.
    02  SCTR-4  COLUMN  50  PIC Z(6).99  SUM  COST.
    .
    .
```

At each execution of a GENERATE DETAIL-1, the value of COST will be added into SUM counters SCTR-1, SCTR-2, SCTR-3, and SCTR-4. When a control break occurs, no 'rolling forward' of counters is necessary since all counters are effectively 'subtotalled'. The only remaining actions to be performed are:

1. Presenting the control footing report groups from the least inclusive (MINOR) up through the control footing representing the control break level.

2. Resetting the corresponding SUM counters to zero after each control footing is presented.

ROLLING FORWARD

Rolling forward is a type of SUM counter manipulation in which SUM counters defined in control footing report groups of lower control levels are added to SUM counters defined in control footing report groups of higher control levels during control break processing.

In the previous example, for instance, the identical results may be obtained more efficiently by 'rolling forward' the SUM counters.

Example:

```
01  DETAIL-1  TYPE  DETAIL  LINE  PLUS  1.
    02  SOURCE  IS  COST.
    .
    .
01  MINOR  TYPE  CF  MINR  LINE  PLUS  1.
    02  SCTR-1  COLUMN  50  PIC Z(6).99  SUM  COST.
    .
    .
01  INTERMEDIATE  TYPE  CF  INTRM  LINE  PLUS  1.
    02  SCTR-2  COLUMN  50  PIC Z(6).99  SUM SCTR-1.
    .
    .
01  MAJOR  TYPE  CF  MAJR  LINE  PLUS  1.
    02  SCTR-3  COLUMN  50  PIC Z(6).99  SUM  SCTR-2.
    .
    .
01  FIN-TOT  TYPE  CF  FINAL  LINE  PLUS  1  NEXT  GROUP  NEXT  PAGE.
    02  SCTR-4  COLUMN  50  PIC Z(6).99  SUM  SCTR-3.
    .
    .
```

The following sequence of events occurs in the above example:

1. At each execution of a GENERATE DETAIL-1 statement, the value of COST is added into SUM counter SCTR-1 (subtotalling).

2. When a control break occurs on control data-name MINR, the control footing report group called MINOR is presented; then SUM counter SCTR-1 is added (rolled forward) to SUM counter SCTR-2.

3. When a control break occurs at a higher control break level, the control footing report groups are presented in sequence from the inclusive (MINOR) up to and including the control footing at which the control break occurred. After each control footing is presented, the SUM counters for that report group are rolled forward to corresponding SUM counters in higher level control footing report groups.

   Thus, the subtotalling operation occurs only at the least inclusive (MINOR) control break level. The remaining SUM counters are augmented only when control break processing takes place.

CROSSFOOTING

Crossfooting is a type of SUM counter manipulation in which SUM counters defined in a given control footing report group are added to other SUM counters in the same report group during control break processing.

Example:

```
     01  DETAIL-1  TYPE  DETAIL  LINE  PLUS  1.
         02  SOURCE  IS  COST-1.
         02  SOURCE  IS  COST-2.
         .
         .
     01  MINOR  TYPE  CF  MINR  LINE  PLUS  1.
         02  SCTR-1  COLUMN  50  PIC Z(6).99  SUM  COST-1.
         02  SCTR-2  COLUMN  60  PIC Z(6).99  SUM  COST-2.
         02  SCTR-3  COLUMN  70  PIC Z(9).99  SUM  SCTR-1,  SCTR-2.
         .
         .
     01  INTERMEDIATE  TYPE  CF  INTRM  LINE  PLUS  1.
         02  SCTR-4  COLUMN  50  PIC Z(6).99  SUM  SCTR-1.
         02  SCTR-5  COLUMN  60  PIC Z(6).99  SUM  SCTR-2.
         02  SCTR-6  COLUMN  70  PIC Z(9).99  SUM  SCTR-4,  SCTR-5.
         .
         .
```

The following sequence of events occurs in the above example:

1. At each execution of a GENERATE DETAIL-1 statement, SUM counters SCTR-1 and SCTR-2 are augmented by the corresponding values of COST-1 and COST-2 (subtotalling).

2. When a control break occurs for the control footing report group called MINOR, SUM counters SCTR-1 and SCTR-2 are added into SUM counter SCTR-3 before the report group is presented (crossfooting).

3. After the report group called MINOR is presented, SUM counters SCTR-1 and SCTR-2 are added into SUM counters SCTR-4 and SCTR-5, respectively (rolled forward).

4. SUM counters SCTR-1, SCTR-2, and SCTR-3 are reset to zero.

5. When a control break occurs for the control footing report group called INTERMEDIATE, SUM counters SCTR-4 and SCTR-5 are added into SUM counter SCTR-6 before the report group is presented (crossfooting).

PRODUCING A REPORT

Once a cobol program using the report writer feature is compiled, there are two or three steps involved in getting the report printed. First, execute the program. This produces a stream file with the name <progid>.<filename> . This file contains an intermediate representation of all reports associated with that file in the program. To extract the desired report(s) from the file, use the process_cobol_report command. This command formats the requested report(s) and sends the output to the users terminal or a file. If the report is sent to a file the dprint command (refer to the MPM Commands) can be used to have the file printed on a line printer.

Report Command

process_cobol_report COMMAND

The process_cobol_report command (abbreviated pcr) extracts reports from a file created by a cobol program using the report writer feature. The format is:

pcr path {-control_args}

where:

1. path

   is the pathname of the input file.

2. control_args

   may be chosen from the following:

   -all, -a
      specifies that all reports in the report file are to be processed. This control argument is incompatible with the -report_code control argument.

   -no_newpage, -nnp
      specifies that newpage characters are not to be emitted when -output_file is used. The default is for each page to end with a newpage character.

   -output_file path, -of path
      specifies that the output is to be directed to the file indicated by path. The default is for the output to be directed to user_output (terminal).

   -report_code STR, -rcd STR
      specifies which report(s) are to be extracted from the report file. STR consists of a string of report codes (the two character designators supplied in the code phrase in the cobol program) with values separated by commas with no spaces. This control argument is incompatible with the -all control argument.

   -stop, -sp
      waits for a carriage return from the user before beginning typing and after each page of output to the terminal.

Notes

When outputting to the terminal, pcr assumes the terminal is positioned to the top of a page for the first line of the report.

When neither -all nor -report_code control arguments are specified, then the report designated by the default code is processed.

APPENDIX A

ORDER OF COBOL SOURCE PROGRAM

An outline of the five divisions of a COBOL source program is given below. The grouping within each division is indicated by indention as follows:

Division / Section / Paragraph		Required By ANSI(X)	Required By MCOBOL(X)
CONTROL DIVISION.			
DEFAULT SECTION.			
IDENTIFICATION DIVISION.		X	
PROGRAM-ID.		X	External name used less [.cobol]
AUTHOR.	any order		
INSTALLATION.	any order		
DATE-WRITTEN.	any order		
DATE-COMPILED.	any order		
SECURITY.			
ENVIRONMENT DIVISION.		X	
CONFIGURATION SECTION.		X	
SOURCE-COMPUTER		X	MULTICS used
OBJECT-COMPUTER		X	MULTICS used
SPECIAL-NAMES.			
INPUT-OUTPUT SECTION.			
FILE-CONTROL.			
I-O-CONTROL.			
DATA DIVISION.		X	
FILE SECTION.			
input files.	any order		
output files.	any order		
sort files.	any order		
merge files.	any order		
WORKING-STORAGE SECTION.			
CONSTANT SECTION.			
LINKAGE SECTION.			
COMMUNICATION SECTION.			
PROCEDURE DIVISION.		X	X
DECLARATIVES.[1]			
(USE Sections)			
END DECLARATIVES.[1]			
(all other procedures)			

[1]If used, both DECLARATIVES and END DECLARATIVES must be present.

SERIES 60 (LEVEL 68)

MULTICS COBOL USERS' GUIDE
ADDENDUM B

SUBJECT

Additions and Changes to the *Multics COBOL Users' Guide*

SPECIAL INSTRUCTIONS

This is the second addendum to AS43, Revision 1, dated December 1976.

The addendum includes description of COPY. . .REPLACING, ALTERNATE RECORD KEY, file organization and structure, use of probe, and a new section on file ordering — SORT and MERGE. Throughout the addendum (except for new Section 9 which is not marked), change bars in the margin indicate technical additions and changes; asterisks denote deletions. These changes will be incorporated into the next revision of the manual.

Insert the attached pages into the manual according to the collating instructions on the back of this cover.

Note:

Insert this cover sheet after the manual cover to indicate the updating of the document with Addendum B.

SOFTWARE SUPPORTED

Multics Software Release 7.0

ORDER NUMBER

AS43B, Rev. 1

February 1979

23229
1379
Printed in U.S.A.

Honeywell

COLLATING INSTRUCTIONS

To update the manual, remove old pages and insert new pages as follows:

Remove	Insert
iii through vii	iii through viii
2-12.1, blank	2-12.1, 2-12.2
2-13, blank	2-13 through 2-16
3-5.1, 3-6	3-5.1, 3-6
3-13, blank	3-13, blank
3-15, 3-16	3-15, 3-16
4-3, 4-4	4-3, 4-4
4-9 through 4-14	4-9 through 4-14
4-29, blank	4-29 through 4-36
5-9, 5-10	5-9, 5-10
5-21, 5-22	5-21, 5-22
6-1 through 6-6	6-1 through 6-6
7-3, 7-4	7-3, 7-4
	9-1 through 9-18
i-1 through i-5	i-1 through i-4
	i-5, blank

The information and specifications in this document are subject to change without notice. This document contains information about Honeywell products or services that may not be available outside the United States. Consult your Honeywell Marketing Representative.

© Honeywell Information Systems Inc., 1979

File No.: 1L23

02/79

AS43B

SERIES 60 (LEVEL 68)

MULTICS COBOL USERS' GUIDE
ADDENDUM C

SUBJECT

Additions and Changes to the Multics COBOL Users' Guide

SPECIAL INSTRUCTIONS

This is the third addendum to AS43, Revision 1, dated December 1976.

The addendum is mainly related to COBOL Standards conformance and includes a new section dealing with the COBOL debug facility. Change bars in the margin indicate technical additions and changes; asterisks denote deletions. Section 10 (Debug Facility) is new and does not have change bars. These changes will be incorporated into the next revision of the manual.

Insert the attached pages into the manual according to the collating instructions on the back of this cover.

Note:

Insert this cover sheet after the manual cover to indicate the updating of the document with Addendum C.

SOFTWARE SUPPORTED

Multics Software Release 7.0b

ORDER NUMBER

AS43C, Rev. 1

September 1979

24495
5C979
Printed in U.S.A.

Honeywell

COLLATING INSTRUCTIONS

To update the manual, remove old pages and insert new pages as follows:

Remove

iii through viii
12-12.1, 2-12.2
2-13, 2-14
3-5, blank
3-5.1, 3-6
3-7, 3-8
3-10.1, blank
3-11, 3-12
3-15 through 3-20
4-3, 4-4
4-11 through 4-14
4-17 through 4-20
5-9, 5-10
5-23, 5-24
6-9, 6-10

i-1 through i-5

Insert

iii through viii
2-12.1, 2-12.2
2-13, 2-14
3-5, blank
3-5.1, 3-6
3-7, 3-8
3-10.1, blank
3-11, 3-12
3-15 through 3-20
4-3, 4-4
4-11 through 4-14
4-17 through 4-20
5-9, 5-10
5-23, 5-24
6-9, 6-10
10-1, 10-2
i-1 through i-6

© Honeywell Information Systems Inc., 1979

SERIES 60 (LEVEL 68)
MULTICS COBOL USERS' GUIDE
ADDENDUM D

SUBJECT

Additions and Changes to the Multics COBOL Users' Guide

SPECIAL INSTRUCTIONS

This is the fourth addendum to AS43, Revision 1, dated December 1976.

Change bars in the margin indicate technical additions and changes; asterisks denote deletions. These changes will be incorporated into the next revision of the manual.

Insert the attached pages into the manual according to the collating instructions on the back of this cover.

Note:

Insert this cover sheet after the manual cover to indicate the updating of the document with Addendum D.

SOFTWARE SUPPORTED

Multics Software Release 8.0

ORDER NUMBER

AS43D, Rev. 1

December 1979

26351
7.5C1279
Printed in U.S.A.

Honeywell

COLLATING INSTRUCTIONS

To update the manual, remove old pages and insert new pages as follows:

Remove	Insert
iii through viii	iii through viii
	ix, blank
2-1, 2-2	2-1, 2-2
2-12.1, 2-12.2	2-12.1, 2-12.2
3-13, blank	3-13, blank
3-13.1, 3-14	3-13.1, 3-14
3-15, 3-16	3-15, 3-16
3-19, 3-20	3-19, 3-20
4-3, 4-4	4-3, 4-4
4-7, 4-8	4-7, 4-8
5-19 through 5-22	5-19 through 5-22
6-1 through 6-6	6-1 through 6-6
i-1 through i-6	i-1 through i-8

© Honeywell Information Systems Inc., 1979

File No.: 1L23

12/79

AS43D

LEVEL 68

MULTICS COBOL USERS' GUIDE ADDENDUM E

SUBJECT

Additions and Changes to the Multics COBOL Users' Guide

SPECIAL INSTRUCTIONS

This is the fifth addendum to AS43, Revision 1, dated December 1976.

Change bars in the margin indicate technical additions and changes; asterisks denote deletions. Section XI (Report Writer) is a new section and does not contain change bars. These changes will be incorporated into the next revision of the manual.

Insert the attached pages into the manual according to the collating instructions on the back of this cover.

Note:

Insert this cover sheet after the manual cover to indicate the updating of the document with Addendum E.

SOFTWARE SUPPORTED

Multics Software Release 9.0

ORDER NUMBER

AS43-01E

July 1981

32035
5C781
Printed in U.S.A.

Honeywell

COLLATING INSTRUCTIONS

To update the manual, remove old pages and insert new pages as follows:

Remove	Insert
v through ix	v through x
2-5, 2-6	2-5, 2-6
2-12.1 through 2-14	2-12.1 through 2-14
	2-17, 2-18
3-13, blank	3-13, blank
3-13.1 through 3-16	3-13.1 through 3-16
4-17 through 4-22	4-17 through 4-22
4-23, 4-23.1	4-23, 4-23.1
5-23, 5-24	5-23, 5-24
10-1, 10-2	10-1, 10-2
	10-3, blank
	11-1 through 11-12
	11-13, blank
i-1 through i-8	i-1 through i-10
Remarks Form (AS43D)	Remarks Form (AS43E)

The information and specifications in this document are subject to change without notice. This document contains information about Honeywell products or services that may not be available outside the United States. Consult your Honeywell Marketing Representative.

© Honeywell Information Systems Inc., 1981

INDEX

ACCEPT statement 4-1, 4-2, 6-7

access
  object segment access 3-16

access mode
  file 4-10
  object program 3-16

ACCESS MODE clause 4-10

ADD statement 6-11, 6-12

add_search_rules command 5-3, 5-4

ADVANCING clause 4-14, 4-26, 4-27, 6-7

ALTER statement 6-7

APPLY clause 4-18

areas
  reference format areas 2-3

AT END clause 6-9

attach
  attach description 4-3
  attaching from command level 4-6
  see I/O module
  see I/O switch

BEFORE
  BEFORE REPORTING phrase 11-2

binary data
  description 5-18, 5-22
  digit encoding 5-19
  displaying and modifying 6-3
  efficient use of 7-3
  invalid values 6-12
  PL/I equivalency 5-25
  sign encoding 5-20, 5-21

binding 5-5

block
  I/O control block 4-3, 4-29

BLOCK CONTAINS clause 4-22.1

BREAK
  control break 11-7

breakpoint
  see debugging

CALL statement 5-2, 5-6, 6-7

CANCEL statement 5-9, 5-12

cancel_cobol_program 5-12

cancel_cobol_program command 5-9, 5-10

catalog-name 4-18, 4-23

change_wdir command 5-3

character
  capitalization considerations 2-8
  case considerations 2-6, 2-8
  COBOL special characters 2-6
  in nonnumeric literals 2-9
  non-COBOL characters 2-7
  source-level escape convention 2-9
  use of
    asterisk 2-5
    backslash 2-10
    backspace 4-26
    carriage return 4-26
    form-feed 3-5.1, 4-26
    hyphen 2-3, 2-5, 2-7, 3-19
    newline 2-3, 2-9, 4-2, 4-26
    period 2-7
    quotation marks 2-9
    slash 2-5
    tab 2-6
    underscore 2-7

character set 2-6

clause
  ACCESS MODE clause 4-10
  ADVANCING clause 4-14, 4-26, 4-27, 6-7
  APPLY clause 4-18
  AT END clause 6-9
  BLOCK CONTAINS clause 4-22.1
  CODE clause 11-8
  DEBUGGING MODE clause 10-1
  DEFAULT COMP clause 5-18
  DEFAULT SIGN clause 5-22

clause (cont)
  DEPENDING ON clause 3-12.2, 4-23, 4-24, 6-7
  GENERATE AGGREGATE DESCRIPTORS clause 2-12
  GENERATE DESCRIPTOR
    SCALAR, AGGREGATE, NO clause 5-26, 5-26.1
  GENERATE DESCRIPTOR clause 2-12
  GENERATE NO DESCRIPTORS clause 2-12
  NUMERIC LIMIT clause 2-12
  OCCURS clause 3-12.1, 4-24
  ORGANIZATION clause 4-8
  PICTURE clause 5-17, 5-18
  RECORD CONTAINS clause 4-23, 4-23.2
  RECORD KEY clause 4-11
  RELATIVE KEY clause 4-11, 4-12
  REPORT(S) clause 11-8
  SELECT clause 4-13
  SELECT clauses 4-6
  SIGN clause 2-11, 5-18, 5-22
  SIZE ERROR clause 6-11, 6-12
  SOURCE clause 11-9
  TYPE clause 11-7
  USAGE clause 2-11, 3-8, 5-17, 5-18, 5-25
  USING clause 2-12, 3-12.1
  VALUE clause 4-20, 4-23, 4-22.1, 7-2

CLOSE statement 4-4, 4-7, 4-20

CMCS
  see COBOL Message Control System

cobol command 3-1, 3-18

COBOL data segment 5-23

COBOL MCS
  see COBOL Message Control System

COBOL Message Control System
  CMCS Administrator 8-5
  command
    cobol_mcs command 8-10
    cobol_mcs_admin command 8-17
    command example 8-27
    cv_cmcs_station_ctl command 8-22
    cv_cmcs_terminal_ctl command 8-23
    cv_cmcs_tree_ctl command 8-24
  data bases 8-5
  design concepts 8-2
  message processors 8-8
  terminology 8-1

cobol_mcs command 8-10

cobol_mcs_admin command 8-17

CODE
  CODE clause 11-8

command
  add_search_rules command 5-3, 5-4
  cancel_cobol_program command 5-9, 5-10, 5-12
  change_wdir command 5-3

command (cont)
  cobol command 3-1, 3-18
  display_cobol_run_unit command 4-29, 5-13
  file_output command 4-3, 4-21
  format_cobol_source command 2-5, 2-6, 2-13
  help command 3-20
  initiate command 5-4
  io_call command 4-6, 4-21, 5-6
  link command 5-4
  print_attach_table command 4-7
  print_link_info command 3-6, 3-16, 3-17
  probe command 6-2
  process_cobol_report command 11-12
  profile command 3-13
  release command 3-18, 5-16
  run_cobol command 5-10, 5-16
  set_search_rules command 5-3
  start command 3-12.1, 5-5
  stop_cobol_run command 5-10, 5-12

command level
  attaching a file from 4-6
  program execution from 5-5

communication
  interprogram communication 5-24

compatibility
  PL/I data type compatibility 5-24

compiler
  abort
    see unrecoverable errors
  characteristics 3-18
  debugging and testing 3-14
  development and testing 3-14
  documentation 3-20
  invoking the compiler 3-1
  reentrancy 3-18
  see control argument

compiling
  interrupted compilations 3-18
  invoking the compiler 3-1
  multiple compilations 3-19
  ordering control arguments 3-19

computational
  data
    see binary data
    summary 5-18

COMPUTE statement 6-7, 6-11

com_err_ subroutine 6-9

control
  automatic return of control 5-17
  control break 11-7
  Control Data Items 11-7
  control division 2-11, 5-18
  descriptor control 2-12
  I/O control block 4-3, 4-29
  Report Control 11-2

control (cont)
  transferring
    see statement
  transfers of control 10-1

control argument
  cancel_cobol_program
    -retain_data 5-12
    -retain_files 5-12
  cobol
    -brief 3-5
    -check 3-12
    -debug 3-14
    -format 2-5, 2-6, 2-13
    -leveli 3-5
    -list 3-12, 3-5.1, 3-10.1
    -map 3-10, 3-5.1
    -profile 3-13
    -runtime_check 3-12
    -severityi 3-3
    -source 3-6, 3-5.1
    -table 3-13.1
    -temp_dir 3-13.1
    -time 3-14
    summary 3-15
  display_cobol_run_unit
    -all 5-13, 5-14
    -files 5-13, 5-15
    -long 5-13, 5-14
  ordering 3-19
  run_cobol
    -cobol_switch 5-10
    -no_stop_run 5-10
    -sort_dir 5-10
    -sort_file_size 5-10
  stop_cobol_run
    -retain_data 5-12
    -retain_files 5-12

conventions
  naming of source program 5-1
  source level escape 2-9

COPY statement 2-12.1

COUNTER
  Line Counter 11-8
  Page Counter 11-9
  SUM Counter Manipulation 11-9

cross-reference listing 3-8, 3-9

CROSSFOOTING 11-11

cv_cmcs_station_ctl command 8-22

cv_cmcs_terminal_ctl command 8-23

cv_cmcs_tree_ctl command 8-24

data
  addressability 5-27
  aggregate 5-25, 5-26
  allocation 3-9, 5-23
  COBOL data segment 5-23
  Control Data Items 11-7
  cross-reference 3-8
  digit encoding 5-19

data (cont)
  displaying data 6-3
  efficiency considerations 7-2
  modifying data 6-3
  per-process 5-23
  PL/I data type compatibility 5-24
  sign encoding 5-20, 5-21
  types
    binary 5-18, 5-22, 5-25, 6-6, 7-3
    index 6-7
    nonseparate sign display 5-18, 5-21, 6-12, 7-3
    packed decimal 5-18, 5-22, 6-4, 7-3
    separate sign display 5-18, 5-20, 5-24, 6-12
    summary 5-18
    unsigned display 5-18, 5-19
  use of numeric data 7-3

DATE-COMPILED paragraph 3-7

DEBUG
  debug Facility 10-1
  DESCRIPTION OF THE DEBUG FACILITY 10-1
  EXAMPLE OF THE DEBUG FACILITY 10-1

debugging 2-6
  compiler debugging and testing 3-14
  displaying and modifying data 6-3
  monitoring program execution 6-2
  source level debugging requirements 3-13.1

DEBUGGING MODE clause 10-1

declarative error procedures 4-24

default
  default section 2-11
  search rules 5-3
  see clause

DEFAULT COMP clause 5-18

default section 2-11

DEFAULT SIGN clause 5-22

definition
  file definition 4-8
  run-unit definition 5-8

DELETE statement 6-9

DEPENDING ON clause 3-12.2, 4-23, 4-24, 6-7

DESCRIPTION
  DESCRIPTION OF THE DEBUG FACILITY 10-1
  DESCRIPTION OF THE REPORT WRITER 11-1

descriptor control 2-12
  SCALAR, AGGREGATE, NO 5-26, 5-26.1

DETAILED
  detailed reporting 11-4

diagnostic messages
  fatal errors 3-2, 3-4
  format 3-2
  observations 3-2, 3-4
  repetition control 3-5
  see control argument (-brief, -severity)
  severity control 3-2
  unrecoverable error 3-2, 3-4
  warnings 3-2, 3-4

digit
  digit encoding
    see data

display
  see binary data

DISPLAY statement 4-1, 6-7, 7-6

display_cobol_run_unit command 4-29, 5-13

DIVIDE statement 6-11, 7-5

documentation
  online information 3-20
  see command (help)

dynamic linking 5-4

ELEMENTARY
  elementary item 11-4

encoding
  see data

ENTRIES
  RD Entries 11-5
  Report Group Entries 11-5

entry
  <io-technique> entry 4-18
  entry point name 5-1, 5-2
  FD entry 4-8
  RD entry 11-4

entry point name 5-1, 5-2

equivalency
  PL/I data type equivalency 5-24, 5-25

error
  exponentiation error 6-7
  fixed-point overflow error 6-12
  illegal procedure condition error 6-11
  input/output error 6-9
  linkage error 6-12
  see subroutine (print_cobol_error)
  source error 3-2
  subscripting error 6-12
  unanticipated error 6-11
  unrecoverable error 3-2, 3-4

escape
  source level escape convention 2-9

EXAMPLE
  command example 8-27
  EXAMPLE OF THE DEBUG FACILITY 10-1

EXIT PROGRAM statement 5-9, 5-17, 6-7

exponentiation error 6-7

external
  files 4-5, 4-12
  switches 5-16, 5-23

FACILITY
  debug Facility 10-1
  DESCRIPTION OF THE DEBUG FACILITY 10-1
  EXAMPLE OF THE DEBUG FACILITY 10-1

FD entry 4-8

field (key) ordering 9-3

file
  activity recording 4-29
  attaching from command level 4-6
  closing 4-4
  defining a file 4-8
  device specification 4-13
  external 4-5, 4-12
  file state block 3-17, 4-3
  fixed length records 4-11
  FLR/VLR/SPANNED 4-11
  indexed 4-32
  internal-file-name 4-12
  key
    see key
  opening 4-3
  opening modes 4-27
  print file 4-25, 6-7
  relative 4-30
  scope 4-5, 4-12
  sequential 4-29
  sharing 4-5
  stream file 4-10
  structure 4-8
  tape files 4-23.1
  temporary files 4-18
  variable length records 4-11
  virtual memory files 4-14

file state block 4-3

FILE-CONTROL paragraph 4-8

file_output command 4-3, 4-21

FINAL 11-7

fixed
  fixed length records 4-11
  fixed reference format 2-3

fixed-point overflow error 6-12

format
  command line 3-1, 3-19
  data
    see data
  diagnostic messages 3-2
  free-form format 2-5, 2-6
  object segment 3-16
  reference format areas 2-3
  Report Format 11-1
  restriction for library files 2-13
  source program reference format 2-3
  terminal oriented format 2-5
  vertical page format 11-8

format_cobol_source command 2-5, 2-6, 2-13

FORWARD
  ROLLING FORWARD 11-10

free format
  see format

FSB
  file state block
    see file

GENERATE
  GENERATE statement 11-4

GENERATE AGGREGATE DESCRIPTORS 2-12

GENERATE DESCRIPTOR
  SCALAR, AGGREGATE, NO clause 5-26, 5-26.1

GENERATE DESCRIPTOR clause 2-12

GENERATE NO DESCRIPTORS clause 2-12

GO TO statement 5-17, 5-23, 6-7

GROUP
  NEXT GROUP 11-9
  Report Group Entries 11-5

GROUPS
  Report Groups 11-6

help command 3-20

I-O-CONTROL paragraph 4-8, 4-18

I/O
  control block 4-3, 4-29
  module
    discard 4-21
    summary 4-4
    syn_ 4-5, 4-21
    tape_ansi 4-14, 4-22
    tape_ibm 4-14, 4-22
  see input/output
  switch
    assignment 4-12
    attachment 4-3
    error_output 5-16
    switch-level sharing 4-5
    user_output 5-8

illegal procedure condition error 6-11

independent segments 5-17, 6-7

index integrity verification 3-12.2

INITIATE
  INITIATE statement 11-3

initiate command 5-4

initiation
  program execution initiation 5-5, 5-11
  run-unit initiation 5-5, 5-10
  segment 5-2

initiation segment 5-2

INPUT-OUTPUT section 4-8

input/output
  COBOL statements
    see statement (ACCEPT, CLOSE, DISPLAY, OPEN, READ, WRITE.)
  efficiency considerations 7-2
  errors 4-24, 6-9
  Multics input/output system
    see attach
  section
    see section (input/output)

input/output errors 6-9

INSPECT statement 7-4

internal files
  see scope of files

interprogram communication 5-24

invalid values 6-12

invoking the compiler
  see compiler

IOCB 4-3

io_call command 4-6, 4-21, 5-6

ITEM
  elementary item 11-4

ITEMS
  Control Data Items 11-7

key
  record key 4-11
  relative key 4-12
  status key 4-17

leveling 3-5

library
  definition 2-12.1
  see command
    (set_translator_search_rules)

library (cont)
  source format restrictions 2-13

LINE
  Line Counter 11-8
  LINE NUMBER 11-9

link
  linkage error 6-12
  linkage section 4-11, 5-23
  linking
    see dynamic linking

link command 5-4

linkage error 6-12

linkage section 4-11, 5-23

list file 3-5.1
  cross-reference listing 3-8, 3-9
  header 3-6
  object map 3-10, 3-16
  see diagnostic messages
  see object segment
  source listing 3-6, 3-10

literals 2-4, 2-7, 2-8, 2-9, 5-23

long binary data
  see binary data

lowercase
  use of lowercase characters 2-6, 2-8

MANIPULATION
  SUM Counter Manipulation 11-9

merge key fields 9-12

MERGE statement 9-2, 9-12

MERGE verb 9-3

messages
  see diagnostic messages

mnemonic-name 4-26.1

mode
  access mode 4-10
    file 4-10
    object program 3-16
  DEBUGGING MODE clause 10-1
  file opening mode 4-27

MOVE statement 6-12

MULTIPLY statement 6-11

names
  catalog-name 4-18, 4-23
  entry point name 5-1, 5-2
  mnemonic-name 4-26.1
  reference names 5-1, 5-2, 6-8, 6-12
  source program naming 5-1

NEXT
  NEXT GROUP 11-9

non-COBOL characters 2-7

nonnumeric literals
  see literals

nonseparate sign display data
  see binary data

NUMBER
  LINE NUMBER 11-9

NUMERIC LIMIT clause 2-12

object
  code suppression 3-12
  environment
    see run-time
  listing 3-16, 3-10.1
  map 3-10, 3-12, 3-16
  segment creation 3-16

object segment 3-16
  creation 3-16
  format 3-16
  see command (print_link_info)
  suppression 3-12

observation diagnostics 3-2, 3-4

OCCURS clause 3-12.1, 4-24

OF
  DESCRIPTION OF THE DEBUG FACILITY 10-1
  DESCRIPTION OF THE REPORT WRITER 11-1
  EXAMPLE OF THE DEBUG FACILITY 10-1
  transfers of control 10-1

OPEN statement 4-7, 4-20, 4-21, 4-23, 4-28, 7-6

opening modes 4-27

ORGANIZATION clause 4-8

overpunched data
  see binary data

packed decimal data
  see binary data

PAGE
  Page Counter 11-9
  vertical page format 11-8

paragraph
  DATE-COMPILED paragraph 3-7
  FILE-CONTROL paragraph 4-8
  I-O-CONTROL paragraph 4-8, 4-18
  PROGRAM-ID paragraph 3-17, 4-23.1, 5-1

parameter
  checking
    see control argument
        (-runtime_check)
  definition
    see section (linkage)
  passing 5-24
  validation 3-12.1

PERFORM statement 6-7

PHRASE
  BEFORE REPORTING phrase 11-2
  RESET phrase 11-8

PICTURE clause 5-17, 5-18

PL/I data type equivalency 5-24, 5-25

print files 4-25, 6-7

print_attach_table command 4-7

print_cobol_error_ subroutine 6-10

print_link_info command 3-6, 3-16,
    3-17

probe command 6-2

process_cobol_report command 11-12

profile command 3-13

program
  measuring performance 7-6
  monitoring program execution 6-2
  object program
    creation 3-16
    format 3-16
  size considerations 7-1
  source program considerations 2-2
  summary 3-10
  termination 5-8, 5-9, 5-11, 5-17

program execution
  continuation of
    see commmand (start)
  from command level 5-5
  initiation of 5-4
  termination of 5-8, 5-9, 5-16, 5-21
  via CALL statement
    see interprogram communication

PROGRAM-ID paragraph 3-17, 4-23.1,
    5-1

RD
  RD Entries 11-5
  RD entry 11-4

READ statement 4-10, 6-9

RECORD CONTAINS clause 4-23, 4-23.2

record key 4-11

RECORD KEY clause 4-11

reentrancy 3-18

reference format
  areas 2-3
  fixed 2-3
  terminal-oriented 2-5

reference names 5-1, 5-2, 6-8, 6-12
  see names

relative key 4-12

RELATIVE KEY clause 4-11, 4-12

release command 3-18, 5-16

RELEASE statement 9-7

REPORT
  DESCRIPTION OF THE REPORT WRITER
      11-1
  Report Control 11-2
  Report Format 11-1
  Report Group Entries 11-5
  Report Groups 11-6

REPORT WRITER
  producing a report 11-12
    process_cobol_report command
        11-12

REPORT(S
  REPORT(S) clause 11-8

report-name 11-9

REPORTING
  BEFORE REPORTING phrase 11-2
  detailed reporting 11-4
  summary reporting 11-4

RESET
  RESET phrase 11-8

RETURN statement 9-8

REWRITE statement 6-9

ROLLING
  ROLLING FORWARD 11-10

run-time
  environment
    binding 5-5
    dynamic linking 5-4
    search rules 5-3
  error checking 3-12
  errors
    input/output 6-9
    message format 6-7.1
    see control argument
        (-runtime_check)
    see subroutine
        (print_cobol_error_)

run-time error
  anticipated 6-6
  input/output 4-23.1

run-unit
  definition 5-8
  external switches 5-16, 5-23
  implementation specifics 5-26.1
  initiation of 5-5, 5-10
  related commands
    see command (cancel_cobol_program, display_cobol_run_unit, run_unit, stop_cobol_run)
  related statements
    see statement (EXIT program, STOP<literal>, STOP RUN)
  segmentation 5-17
  termination 5-8, 5-11

run_cobol command 5-10, 5-16

scope of files 4-5, 4-12

search
  search rules 5-3

SEARCH statement 6-7

section
  default section 2-11
  INPUT-OUTPUT section 4-8
  linkage section 4-11, 5-23
  WORKING-STORAGE section 4-11, 5-8

segment
  COBOL data segment 5-23
  independent 5-17
  independent segment 6-7
  initiation segment 5-2
  list segment
  object segment 3-16
  source segment
    see source program

segmentation 5-17

SELECT clause 4-6, 4-13

separate sign display data
  see binary data

separators 2-14

SET statement 3-12.2, 6-7

set_search_rules command 5-3

sharing
  file sharing 4-5

short binary data
  see binary data

sign
  control 2-11
  encoding 5-20, 5-21
  nonseparate sign display data 5-21, 6-3, 6-12, 7-3
  separate sign display data 5-20, 6-3, 6-12, 7-3
  unsigned display data 5-18, 5-19, 6-3

SIGN clause 2-11, 5-18, 5-22

SIZE ERROR clause 6-11, 6-12

slewing 4-26.1

sort key 9-5

SORT statement 6-7, 9-1

SORT verb 9-3

SOURCE
  SOURCE clause 11-9

source error 3-2

source listing 3-6, 3-10

source program
  area definition 2-3, 2-4
  conversion considerations 2-3
  creation 3-16
  division of A-1
  errors
    see diagnostic messages
  escape convention 2-9
  naming convention 5-1
  reference format
    fixed 2-3
    see control argument (-format)
    terminal oriented 2-5
  see character set
  see library
  translation 2-6, 3-13.1

source program naming 5-1

source transformation 3-13

SPANNED records 4-11

start command 3-12.1, 5-5

START statement 6-9

statement
  ACCEPT statement 4-1, 4-2, 6-7
  ADD statement 6-11, 6-12
  ALTER statement 6-7
  CALL statement 5-2, 5-6, 6-7
  CANCEL statement 5-9, 5-12
  CLOSE statement 4-4, 4-7, 4-20
  COMPUTE statement 6-7, 6-11
  COPY statement 2-12.1
  DELETE statement 6-9
  DISPLAY statement 4-1, 6-7, 7-6
  DIVIDE statement 6-11, 7-5
  EXIT PROGRAM statement 5-9, 5-17, 6-7
  GENERATE statement 11-4
  GO TO statement 5-17, 5-23, 6-7
  INITIATE statement 11-3
  INSPECT statement 7-4
  MERGE statement 9-2, 9-12
  MOVE statement 6-12
  MULTIPLY statement 6-11

statement (cont)
  OPEN statement 4-7, 4-20, 4-21, 4-23, 4-28, 7-6
  PERFORM statement 6-7
  READ statement 4-10, 6-9
  RELEASE statement 9-7
  RETURN 9-14
  RETURN statement 9-8
  REWRITE statement 6-9
  SEARCH statement 6-7
  SET statement 3-12.2, 6-7
  SORT statement 6-7, 9-1
  START statement 6-9
  STOP RUN statement 5-8, 5-9, 5-11, 5-16, 5-23, 6-7
  STOP statement 5-16
  STRING statement 7-6
  SUBTRACT statement 6-11
  SUPPRESS statement 11-2
  TERMINATE statement 11-4
  UNSTRING statement 7-6
  USE statement 4-25, 6-9
  WRITE statement 4-12, 4-14, 4-21, 6-7, 6-9

status
  file status 4-16
  status keys 4-17

status key 4-17

STOP RUN statement 5-8, 5-9, 5-11, 5-16, 5-23, 6-7

STOP statement 5-16

stop_cobol_run command 5-10, 5-12

stream files 4-10

string range checking 3-12.2

STRING statement 7-6

subroutine
  com_err_ 6-9
  print_cobol_error_ 6-10

subscript range checking 3-12.1

subscripting
  checking bounds 3-12, 3-12.1
  errors 6-12

subscripting error 6-12

SUBTOTALLING 11-10

SUBTRACT statement 6-11

SUM
  SUM Counter Manipulation 11-9

SUMMARY
  summary reporting 11-4

SUPPRESS
  SUPPRESS statement 11-2

switches
  external 5-16, 5-23

SYSIN
  ACCEPT...FROM<sysin> 4-1

SYSOUT
  DISPLAY...UPON<sysout> 4-2

tape files 4-23.1

temporary files 4-18

TERMINATE
  TERMINATE statement 11-4

termination
  program 4-19, 5-8, 5-11, 5-17
  run-unit 5-8, 5-11

text word 2-13

THE
  DESCRIPTION OF THE DEBUG FACILITY 10-1
  DESCRIPTION OF THE REPORT WRITER 11-1
  EXAMPLE OF THE DEBUG FACILITY 10-1

TRANSFERS
  transfers of control 10-1

transformation
  source 3-13

translation
  source program translation 2-6, 2-9, 3-13.1

TYPE
  TYPE clause 11-7

unanticipated error 6-11

unrecoverable error 3-2, 3-4

unsigned display data
  see binary data

UNSTRING statement 7-6

uppercase
  use of uppercase characters 2-6, 2-8

USAGE clause 2-11, 3-8, 5-17, 5-18, 5-25

USE statement 4-25, 6-9

USING clause 2-12, 3-12.1

VALUE clause 4-20, 4-23, 4-22.1, 7-2

variable length records 4-11

verb
  MERGE verb 9-3

verb (cont)
  SORT verb 9-3

VERTICAL
  vertical page format 11-8

virtual memory files 4-14, 4-23

WORKING-STORAGE section 4-11, 5-8

WRITE statement 4-12, 4-14, 4-21, 6-7, 6-9

WRITER
  DESCRIPTION OF THE REPORT WRITER 11-1

CUT ALONG LINE

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE	LEVEL 68 MULTICS COBOL USERS' GUIDE ADDENDUM E	ORDER NO.	AS43-01E
		DATED	JULY 1981

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. ☐

FROM: NAME ______________________ DATE ____________

TITLE ______________________

COMPANY ______________________

ADDRESS ______________________

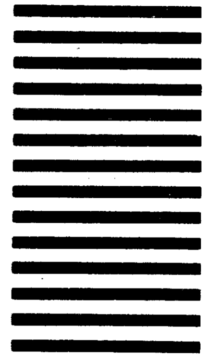
____

CUT ALONG LINE

FOLD ALONG LINE

PLEASE FOLD AND TAPE–
NOTE: U. S. Postal Service will not deliver stapled forms

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 39531 WALTHAM, MA02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

FOLD ALONG LINE

Honeywell

Honeywell

Honeywell Information Systems
In the U.S.A.: 200 Smith Street, MS 486, Waltham, Massachusetts 02154
In Canada: 2025 Sheppard Avenue East, Willowdale, Ontario M2J 1W5
In Mexico: Avenida Nuevo Leon 250, Mexico 11, D.F.

21334, 1778, Printed in U.S.A.

AS43, Rev. 1