

SC28-6478-1

Program Product

**IBM DOS/VS COBOL
Compiler and Library
Programmer's Guide**

IBM

SC28-6478-1

Program Product

**IBM DOS/VS COBOL
Compiler and Library
Programmer's Guide**

**Program Numbers: 5746-CB1 (Compiler and Library)
5746-LM4 (Library)**

IBM

Second Edition (March 1975)

This edition is a reprint of SC28-6478-0 incorporating changes released in Technical Newsletter SN28-1063 (dated March 22, 1974).

This edition corresponds to Release 1 of the IBM DOS/VS COBOL Compiler and Library.

Information in this publication is subject to significant change. Any such changes will be published in new editions or technical newsletters. Before using this publication, consult the latest *IBM System/360 and System/370 Bibliography*, GA22-6822, and the technical newsletters that amend the bibliography, to learn which editions and technical newsletters are applicable and current.

Requests for copies of IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for readers' comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, Department J04, 1501 California Avenue, Palo Alto, California 94304. Comments become the property of IBM.

Date of Publication: March 22, 1974

Form of Publication: TNL SN28-1063 to SC28-6478-0

New: Additional Compiler Capabilities

Lister feature
Execution Statistics and
Verb summary feature
SORT-OPTION

Maintenance: Documentation Only

Minor technical changes and corrections.

Editorial changes that have no technical significance are not noted here.

Specific changes to the text made as of this publishing date are indicated by a vertical bar to the left of the text. These bars will be deleted at any subsequent republication of the page affected.

PREFACE

This publication describes how to compile a COBOL program using the Program Product IBM DOS/VS COBOL Compiler. It also describes how to link edit the resulting object module, and execute the program. Included is a description of the output from each of these three steps: compile, link edit, and execute. This publication explains features of the DOS/VS Compiler and Library, and available options of the operating system.

This publication is logically and functionally divided into four parts. Part I contains information useful to programmers who are running COBOL programs compiled on the DOS/VS Compiler, under the control of the IBM Disk Operating System Virtual Storage. Part I covers such topics as job control language, library usage, and interpreting output.

Part II contains supplemental information on the use of the language as specified in the publication IBM DOS Full American National Standard COBOL, Order No. GC28-6394, and should be used in conjunction with this publication for coding COBOL programs. Part II covers in detail such topics as file organization, file label handling, and record formats. Part II is intended as reference material for language features that are primarily system-dependent.

Part III contains information on programming techniques useful to the programmer running COBOL programs compiled on the DOS/VS Compiler. Topics such as coding considerations, table handling considerations, and formatting data are covered in Part III.

Part IV contains error determination information. This part covers such topics as program debugging and program testing.

Diagnostic messages generated by the DOS/VS Compiler and Library and their accompanying documentation can be found in this publication.

Information on installing the DOS/VS Compiler and Library can be found in the following publication:

IBM DOS/VS COBOL Compiler and Library, Installation Reference Material, Order No. SC28-6479.

Wider ranging and more detailed discussions of the DOS/VS System are given in the following publications:

Introduction to DOS/VS, Order No. GC33-5370

DOS/VS System Generation, Order No. GC33-5377

DOS/VS System Management Guide, Order No. GC33-5371

DOS/VS Data Management Guide, Order No. GC33-5372

DOS/VS Supervisor and I/O Macro Reference, Order No. GC33-5373

DOS/VS Access Method Services, Order No. GC33-5382

DOS/VS System Control Statements, Order No. GC33-5376

DOS/VS System Utilities Reference, Order No. GC33-5381

DOS/VS Messages, Order No. GC33-5379

The following publications provide detailed information on the IBM 3886 Optical Character Reader:

IBM 3886 Optical Character Reader General Information Manual, Order No. GA21-9146

IBM 3886 Optical Character Reader Input Document Design and Specifications, Order No. GA21-9148

DOS/VS Planning Guide for the IBM 3886 Optical Character Reader, Model 1, Order No. GC21-5059

The following publications provide information on the IBM DOS/VS Sort/Merge Program Product, Program Number 5746-SM1, and the DOS Sort/Merge Program Product, Program Number 5743-SM1:

IBM DOS/VS Sort/Merge General Information, Order No. GC33-4030

IBM DOS/VS Sort/Merge Program Product Design Objectives, Order No. GC33-4027

IBM DOS/VS Sort/Merge Installation Reference Material, Order No. SC33-4026

IBM DOS Sort/Merge Programmer's Guide, Order No. SC33-4018

The titles and abstracts of related publications are listed in IBM System/360 and System/370 Bibliography, Order No. GA22-6822.

CONTENTS

FEATURES OF THE PROGRAM PRODUCT DOS/VS		
COMPILER	7	Mutually Exclusive Options 40
PART I	9	Changing the Installation Defaults . 40
INTRODUCTION	11	Significant Characters for Various Options 40
Control Program	11	Job Control Commands 40
Supervisor	11	Linkage Editor Control Statements . . . 40
Job Control Processor	11	Control Statement Placement 41
Initial Program Loader	11	PHASE Statement 41
Processing Programs	11	INCLUDE Statement 42
System Service Programs	11	ENTRY Statement 42
Application Programs	12	ACTION Statement 42
IBM-Supplied Processing Programs . . . 12		Autolink Feature 43
Data Management	12	Relocating Loader Feature 43
Multiprogramming	12	LIBRARIAN FUNCTIONS 45
Background vs. Foreground Programs . . 12		Planning the Libraries 45
JOB DEFINITION	13	Librarian 45
Job Steps	13	Core Image Library 45
Compilation Job Steps	13	Cataloging and Retrieving Program Phases -- Core Image Library 45
Multiphase Program Execution 13		Relocatable Library 46
Types of Jobs	14	Maintenance Functions 46
Job Definition Statements	15	Cataloging a Module -- Relocatable Library 46
Other Job Control Statements	16	Source Statement Library 47
JOB PROCESSING	17	Maintenance Functions 48
Compilation	17	Cataloging a Book -- Source Statement Library 48
Editing	17	Updating Books -- Source Statement Library 50
Phase Execution	18	UPDATE Function -- Invalid Operand Defaults 52
Multiphase Programs	18	The Procedure Library 53
PREPARING COBOL PROGRAMS FOR PROCESSING 19		MAINT, Procedure Library 53
Assignment of Input/Output Devices . . . 19		Catalog 53
Job Control	22	PSERV, Procedure Library 54
Job Control Statements	22	Calling Cataloged Procedures 54
Comments in Job Control Statements . 22		Private Libraries 55
Statement Formats	22	Determining the Location of the Libraries 55
Sequence of Job Control Statements . 23		Source Language Considerations 56
Description and Formats of Job Control Statements 23		Extended Source Program Library Facility 57
JOB Statement	23	Reformatted Source Deck 57
ASSGN Statement	24	INTERPRETING OUTPUT 59
CLOSE Statement	26	Compiler Output 59
DATE Statement	26	Object Module 69
TLBL Statement	27	Linkage Editor Output 69
DLBL Statement	28	Comments on the Phase Map 71
EXTENT Statement	28	Linkage Editor Messages 71
VOL, DLAB, TPLAB and XTENT Statements . 30		DOS ANS COBOL Unresolved External References 71
LBLTYP Statement	30	COBOL Execution Output 72
LISTIO Statement	31	Operator Messages 72
MTC Statement	31	STOP Statement 73
OPTION Statement	32	ACCEPT Statement 73
PAUSE Statement	34	System Output 73
RESET Statement	34	CALLING AND CALLED PROGRAMS 75
RSTRT Statement	34	Linkage 75
UPSI Statement	35	Linkage In A Calling Program 75
EXEC Statement	35	
CBL Statement -- COBOL Option		
Control Card	36	
LST Statement -- New Compiler		
Option Card	40	

Linkage In A Called Program	76	PROCESSING 3540 DISKETTE UNIT FILES	123
Entry Points	76	File Processing	123
Correspondence of Arguments and Parameters	77	DTFDDU	124
Link Editing Without Overlay	77	Job Control Requirements	124
Assembler Language Subprograms	78	DLBL Statement	125
Register Use	78	EXTENT Statement	125
Save Area	79	3540 File	125
Argument List	79	VIRTUAL STORAGE ACCESS METHOD (VSAM)	127
File-Name and Procedure-Name Arguments	79	File Organization	127
In-Line Parameter List	81	Key-Sequenced Files	127
Lowest Level Program	81	Entry-Sequenced Files	127
Overlays	81	Data Organization	128
Special Considerations When Using Overlay Structures	81	Data Access	128
Assembler Language Subroutine for Accomplishing Overlay	82	VSAM Catalog	128
Link Editing with Overlay	83	File and Volume Portability	130
Job Control for Accomplishing Overlay	84	Service Programs	130
USING THE SEGMENTATION FEATURE	89	Device Support	130
Operation	90	Security	130
Output From a Segmented Program	91	Error Processing	130
Compiler Output	91	VSAM Messages	130
Linkage Editor Output	92	Access Method Services	130
Cataloging a Segmented Program	92	Functional Commands	130
Determining the Priority of the Last Segment Loaded into the Transient Area	92	The DEFINE Command	130
Sort in a Segmented Program	93	Specification of the DEFINE Command	131
Using the PERFORM Statement in a Segmented Program	93	Defining a VSAM Master Catalog: DEFINE MASTERCATALOG	131
PART II	95	Defining a VSAM Data Space: DEFINE SPACE	132
PROCESSING COBOL FILES ON MASS STORAGE		Defining a VSAM File: DEFINE CLUSTER	133
DEVICES	97	File Processing Techniques	134
File Organization	97	Current Record Pointer	134
Sequential Organization	97	Error Handling	136
Direct Organization	97	Record Formats for VSAM Files	136
Indexed Organization	97	Initial Loading of Records into a File	137
Data Management Concepts	98	File Status Initialization	137
Sequential Organization (DTFSD)	99	Opening a VSAM File	137
Processing a Sequentially Organized File	99	Writing Records into a VSAM File	139
Direct Organization (DTFDA)	99	Entry-Sequenced File	
Accessing a Directly Organized File	100	Considerations for the WRITE Statement	139
ACTUAL KEY Clause	101	Key-Sequenced File Considerations for the WRITE Statement	139
Randomizing Techniques	102	Rewriting Records on a VSAM File	139
Actual Track Addressing		Entry-Sequenced File	
Considerations for Specific Devices	116	Considerations for the REWRITE Statement	139
Randomizing for the 2311 Disk Drive	116	Key-Sequenced File Considerations for the REWRITE Statement	140
Randomizing for the 2321 Data Cell	117	Reading Records on a VSAM File	140
Indexed Organization (DTFIS)	118	Entry-Sequenced File	
Prime Area	118	Considerations for the READ Statement	140
Indexes	119	Key-Sequenced File Considerations for the READ Statement	140
Track Index	119	READ NEXT Statement	140
Cylinder Index	119	READ Statement	141
Master Index	119	Using the START Verb	141
Overflow Area	119	DELETE Statement	141
Cylinder Overflow Area	119	COBOL Language Usage With VSAM	141
Independent Overflow Area	119	Creating a VSAM File	141
Adding Records to an Indexed File	120	Retrieving a VSAM File	143
Accessing an Indexed File (DTFIS)	121	Job Control Language for a VSAM File	144
Key Clauses	121		
Improving Efficiency	122		

Converting Non-VSAM Files to VSAM Files144	Data Division193
Using ISAM Programs to Process VSAM Files144	Overall Considerations193
DETAILED FILE PROCESSING CAPABILITIES145	FD Entries193
COBOL VSAM Control blocks145	Prefixes193
Control Blocks For VSAM146	Level Numbers193
VSAM File Information Block (FIB)146	File Section194
VSAM File Control Block148	RECORD CONTAINS Clause194
DTF Tables150	BLOCK CONTAINS Clause194
Pre-DTF Switch155	Working-Storage Section194
Error Recovery for Non-VSAM Files155	Separate Modules194
Volume and File Label Handling162	Locating the Working-Storage Section in Dumps194
Tape Labels162	REDEFINES Clause194
Volume Labels162	PICTURE Clause196
Standard File Labels162	USAGE Clause197
User Standard Labels162	SYNCHRONIZED Clause200
Nonstandard Labels162	Special Considerations for DISPLAY and COMPUTATIONAL Fields200
Label Processing Considerations165	Data Formats in the Computer200
Sample Programs166	Procedure Division202
Mass Storage File Labels174	Modularizing the Procedure Division202
Volume Labels174	Main-Line Routine203
Standard File Labels174	Processing Subroutines203
User Labels174	Input/Output Subroutines203
Label Processing Considerations174	Overall Considerations203
Files on Mass Storage Device174	OPTIMIZE Option203
Opened as Input174	Intermediate Results203
Files on Mass Storage Devices174	Intermediate Results and Binary Data Items203
Opened as Output175	Intermediate Results and COBOL Library Subroutines203
Unlabeled Files175	Intermediate Results Greater Than 30 Digits204
PROCESSING ASCII TAPE FILES177	Intermediate Results and Floating-point Data Items204
COBOL Language Considerations177	Intermediate Results and the ON SIZE ERROR Option204
File Handling177	Exponentiation204
Operational Considerations178	Optimization Based on Execution Frequency204
Obtaining an ASCII Collating Sequence on a Sort178	Procedure Division Statements204
RECORD FORMATS FOR NON-VSAM FILES179	COMPUTE Statement204
Fixed-length (Format F) Records179	IF Statement205
Undefined (Format U) Records180	MOVE Statement205
Variable-Length Records180	NOTE Statement205
APPLY WRITE-ONLY Clause183	PERFORM Statement205
Spanned (Format S) Records183	READ INTO and WRITE FROM Options205
S-Mode Capabilities184	TRANSFORM Statement206
Sequentially Organized S-Mode Files on Tape or Mass Storage Devices185	USING THE SORT/MERGE FEATURE207
Source Language Considerations185	Sort/Merge Job Control Requirements207
Processing Sequentially Organized S-Mode Files185	Sort Input and Output Control Statements207
Directly Organized S-Mode Files187	Sort Work File Control Statements208
Source Language Considerations187	Amount of Intermediate Storage Required208
Processing Directly Organized S-Mode Files188	Improving Performance208
PART III189	SORT-OPTION Clause208
PROGRAMMING TECHNIQUES191	PRINT Option208
Coding Considerations for DOS/V3191	LABEL Option208
General Considerations191	STORAGE Option208
COPY191	ALTWK Option208
Syntax Checking191	ERASE Option208
Formatting the Source Program191	ROUTE Option208
Listing191	SORTWK Option208
Environment Division192	SORT-OPTION Clause Examples208
RESERVE Clause192	Output File Statements208
APPLY WRITE-ONLY Clause192	Sort Diagnostic Messages209

Linkage With the Sort/Merge Feature	209	Debug Language	247
Completion Codes	209	Flow of Control	247
Cataloging a Sort Program	209	Displaying Data Values During	
Checkpoint/Restart During a Sort	210	Execution	248
Using Sort in a Multiphase Environment	210	Testing a Program Selectively	250
		Testing Changes and Additions to	
USING THE REPORT WRITER FEATURE	211	Programs	250
REPORT Clause in a File	211	Dumps	251
Description (FD) Entry	211	How to Use a Dump	251
Summing Techniques	211	Errors That Can Cause a Dump	252
Use of SUM	211	Locating a DTF	252
SUM Routines	212	Locating Data	253
Output Line Overlay	213		
Page Breaks	213	EXECUTION STATISTICS	260
Control Footings and Page Format	213	Obtaining Execution Statistics	260
WITH CODE Clause	214	Debugging and Testing	260
NEXT GROUP Clause	214	Optimization Methods	260
Floating First Detail	215	Resequencing the Program	260
Report Writer Routines	215	Insight into SYMDMP Output	260
		Common Expression Elimination	260
TABLE HANDLING CONSIDERATIONS	217	Backward Movement	260
Subscripts	217	Unrolling	260
Index-names	217	Jamming	260
Index Data Items	217	Unswitching	260
OCCURS Clause	217	Incorporating Procedures Inline	260
DEPENDING ON Option	217	Tabling	260
OCCURS Clause with the DEPENDING ON		Efficiency Guidelines	260
Option	218	Diagnostic Messages	260
SET Statement	221	Working with Diagnostic Messages	260
SEARCH Statement	224	Generation of Diagnostic Messages	260
SEARCH ALL Statement	224	Linkage Editor Output	261
Building Tables	225	Execution Time Messages	261
		Recording Program Status	262
PART IV	227	RERUN Clause	262
		Taking a Checkpoint	262
LISTER FEATURE	228	Restarting a Program	263
Overall Operation of the Lister	228		
The Listing	228	APPENDIX A: SAMPLE PROGRAM OUTPUT	265
The Output Deck	228	APPENDIX B: STANDARD TAPE FILE LABELS	279
Reformatting of Identification and		APPENDIX C: STANDARD MASS STORAGE	
Environment Divisions	228	DEVICE LABELS	281
Data Division Reformatting	228	APPENDIX D: TRACK FORMATS FOR	
Procedure Division Reformatting	228	DIRECT-ACCESS STORAGE DEVICES	287
Summary Listing	228	APPENDIX E: COBOL LIBRARY SUBROUTINES	289
The Source Listing	228	Input/Output Subroutines	289
General Appearance	228	Printer Spacing	289
Format Conventions	228	Tape and Sequential Disk Labels	289
Type Indicators	228	CLOSE WITH LOCK Subroutine	289
The Summary Listing	228	WRITE Statement Subroutines	289
General Appearance	228	READ Statement Subroutines	289
The Output Deck	228	REWRITE Statement Subroutines	290
Using the Lister	228	DISPLAY (EXHIBIT and TRACE)	
Options	228	Subroutines	290
Programming Considerations	228	ACCEPT and STOP (literal) Statement	
		Subroutines	290
SYMBOLIC DEBUGGING FEATURES	229	CLOSE Subroutine	290
Use of the Symbolic Debugging Features	229	Multiple File Tape Subroutine	290
Statement Number Option	229	Tape Pointer Subroutine	290
Flow Trace Option	229	Input/Output Error Subroutines	290
Symbolic Debug Option	229	Disk Extent Subroutines	290
Object-Time Control Cards	230	3886 OCR Subroutine	290
Overall Considerations	232	VSAM Subroutines	290
Sample Program -- Testrun	232	Auxiliary Subroutines	291
Debugging TESTRUN	233	ASCII Support Subroutines	291
PROGRAM CHECKOUT	247		
Syntax-Checking Compilation	247		
Identification of Program Versions	247		

Separately Signed Numeric Subroutine291	Retrieving and Updating a Direct File304
Conversion Subroutines291	Indexed Files305
Arithmetic Verb Subroutines293	Creating an Indexed File305
Sort/Merge Feature Interface Routine293	Retrieving and Updating an Indexed File306
Checkpoint (RERUN) Subroutine293	Files Used in a Sort Operation306
Segmentation Feature Subroutine293	Sorting an Unlabeled Tape File306
Other Verb Routines293	APPENDIX I: DIAGNOSTIC MESSAGES307
Compare Subroutines293	Compile-Time Messages307
MOVE Subroutines294	Operator Messages307
TRANSFORM Subroutine294	Object-Time Messages309
Class Test Subroutine294	COBOL Object Program Unnumbered Messages318
SEARCH Subroutine294	APPENDIX J: COBOL 3886 OPTICAL CHARACTER READER SUPPORT319
Main Program or Subprogram Subroutine294	3886 OCR Processing319
Object-Time Debugging Subroutines294	Implementing an OCR Operation319
Debug Control Subroutine294	Document Design319
Statement Number Subroutine294	Document Description320
Flow Trace Subroutine295	COBOL Support320
Symbolic Debug Subroutines295	File Description320
Object-Time Execution Statistics Subroutines295	Record Description320
COUNT Initialization Subroutine295	Procedural Code321
COUNT Frequency Subroutine295	JCL Considerations321
COUNT Termination Subroutine295	Subprogram Interface321
COUNT Print Subroutine295	Statements for Invoking 3886 I/O Functions324
Optimizer Subroutines295	OPEN Function (Equivalent to OPEN Macro)324
GO TO ... DEPENDING ON Subroutine295	CLOSE Function (Equivalent to DOS CLOSE Macro)324
Optimizer DISPLAY Subroutine296	READ Function (Equivalent to DOS READ and WAITF Macros)324
Transient Subroutines296	READO Function (Equivalent to DOS READ Macro)324
Symbolic Debug Subroutines296	WAIT Function (Equivalent to DOS WAITF Macro)324
SYMDMP Error Message Subroutine296	MARKL Function (Equivalent to DOS CNTRL Macro with LMK Option)324
Error Message Subroutine296	MARKD Function (Equivalent to DOS CNTRL Macro with DMK Option)325
Error Message Print Subroutine296	EJECT Function (Equivalent to DOS CNTRL Macro, with ESP Option)325
Reposition Tape Subroutine296	SETDV (Set Device by Loading a Format Record) Function (Equivalent to DOS SETDEV Macro)325
APPENDIX F: SYSTEM AND SIZE CONSIDERATIONS297	COBOL 3886 Library Routine325
Minimum Machine Requirements for the Compiler297	Processing Tapes from the OCR 3886, Model 2326
Source Program Size Considerations297	INDEX333
Compiler Capacity297		
Effective Storage Considerations298		
Execution Time Considerations299		
Multiprogramming Considerations300		
Sort Feature Considerations300		
APPENDIX G: COMMUNICATION REGION301		
Communication Region301		
APPENDIX H: SAMPLE JOB DECKS303		
Direct Files304		
Creating a Direct File304		

ILLUSTRATIONS

TABLES

Table 1. Job Control Statements . . .	16	Table 20. Fields Preceding DTFDA --	
Table 2. Symbolic Names, Functions,		ACCESS IS SEQUENTIAL -- Relative Track	
and Permissible Device Types	21	Addressing154
Table 3. Significant Characters for		Table 21. Fields Preceding DTFIS154
Various Options	40	Table 22. Fields Preceding DTFDU155
Table 4. Glossary Definition and		Table 23. Meaning of Pre-DTF Switch . .	.155
Usage	65	Table 24. Errors Causing an Invalid	
Table 5. Symbols Used in the Listing		Key Condition156
and Glossary to Define		Table 25. Meaning of Error Bytes for	
Compiler-Generated Information	66	GIVING Option of Error Declarative	
Table 6. System Message		(Part 1 of 2)157
Identification Codes	73	Table 26. Location and Meaning of	
Table 7. Conventional Use of Linkage		Error Bits for DTFMT159
Registers	78	Table 27. Location and Meaning of	
Table 8. Save Area Layout and Word		Error Bits for DTFSD159
Contents	79	Table 28. Location and Meaning of	
Table 9. Recording Capacities of Mass		Error Bits for DTFDA160
Storage Devices	97	Table 29. Location and Meaning of	
Table 10. Partial List of Prime		Error Bits for DTFIS160
Numbers105	Table 30. Location and Meaning of	
Table 11. File Status Values and		Error Bits for DTFDU161
Error Handling136	Table 31. Data Format Conversion	
Table 12. File Status Key Values at		(Part 1 of 2)198
OPEN138	Table 32. Relationship of PICTURE to	
Table 13. File Status at Action		Storage Allocation202
Request Time138	Table 33. Rules for the SET Statement .	.223
Table 14. COBOL Statements for		Table 34. Individual Type Codes Used	
Creating a VSAM File141	in SYMDMP Output234
Table 15. COBOL Statements for		Table 35. Functions of COBOL Library	
Retrieving a VSAM File143	Conversion Subroutines292
Table 16. Fields Preceding DTFMT and		Table 36. Functions of COBOL Library	
DTFSD151	Arithmetic Subroutines293
Table 17. Fields Preceding DTFDA --		Table 37. OCR Status Key Values and	
ACCESS IS RANDOM -- Actual Track		User Actions322
Addressing151	Table 38. Possible Status Key Values,	
Table 18. Fields Preceding DTFDA --		By Operation322
ACCESS IS RANDOM -- Relative Track		Table 39. User Responses to Status Key 323	
Addressing152	Table 40. CALL Statements for	
Table 19. Fields Preceding DTFDA --		Invoking 3886 I/O Functions326
ACCESS IS SEQUENTIAL -- Actual Track			
Addressing153		

FIGURES

Figure 1. Sample Structure of Job Deck for Compiling, Link Editing, and Executing a Main Program and Two Subprograms	13	Figure 30. Permissible Specifications for the First Eight Bytes of the Actual Key102
Figure 3. Possible Specifications for X'ss' in the ASSGN Control Statement	25	Figure 31. Creating a Direct File Using Method B (Part 1 of 4)107
Figure 4. Sample Label and File Extent Information for Mass Storage Files	30	Figure 32. Creating a Direct File with Relative Track Addressing Using Method B (Part 1 of 4)112
Figure 5. Job Definition -- Use of the Librarian	41	Figure 33. Formats of Blocked and Unblocked Records118
Figure 6. Options Available During Link-Editing	44	Figure 34. Adding a Record to a Prime Track120
Figure 7. The Relative Location of the Four System Libraries	56	Figure 35. VSAM Data Organization129
Figure 8. Sample Coding to Calculate FICA	58	Figure 36. Defining a VSAM Master Catalog131
Figure 9. Altering a Program from the Source Statement Library Using INSERT and DELETE Cards	58	Figure 37. Defining a VSAM Data Space	132
Figure 10. Effect of INSERT and DELETE Cards	58	Figure 38. Defining a Key-Sequenced Suballocated VSAM File133
Figure 11. Examples of Compiler Output (Part 1 of 4)	60	Figure 39. Standard Tape File Label and TLBL Card (Showing Maximum Specifications)163
Figure 12. A Program that Produces COBOL Compiler Diagnostics	69	Figure 40. Standard Tape File Label and TLBL Card (Showing Minimum Requirements)164
Figure 13. Linkage Editor Output	70	Figure 41. Standard, User Standard, and Volume Labels165
Figure 14. Output from Execution Job Step	72	Figure 42. Nonstandard Labels165
Figure 15. Calling and Called Programs	75	Figure 43. Processing an Unlabeled Multivolume Volume (Part 1 of 2)168
Figure 16. Example of Data Flow Logic in a Call Structure	78	Figure 44. Reading a Multivolume File with Standard Labels; Creating a Multivolume File with Standard Labels (Part 1 of 2)170
Figure 17. Sample Linkage Routines Used with a Calling Subprogram	80	Figure 45. Creating an Unlabeled Multivolume File (Part 1 of 2)172
Figure 18. Sample In-line Parameter List	81	Figure 46. Fixed-Length (Format F) Records179
Figure 19. Sample Linkage Routines Used with a Lowest Level Subprogram	81	Figure 47. Undefined (Format U) Records180
Figure 20. Example of an Assembler Language Subroutine for Accomplishing Overlay	82	Figure 48. Unblocked V-Mode Records180
Figure 21. Flow Diagram of Overlay Logic	84	Figure 49. Blocked V-Mode Records181
Figure 22. Job Control for Accomplishing Overlay	84	Figure 50. Fields in Unblocked V-Mode Records182
Figure 23. Calling Sequence to Obtain Overlay Between Three COBOL Subprograms (Part 1 of 3)	85	Figure 51. Fields in Blocked V-Mode Records182
Figure 24. Segmenting the Program SAVECORE	89	Figure 52. First Two Blocks of VARIABLE-FILE-2183
Figure 25. Storage Layout for SAVECORE	91	Figure 53. Control Fields of an S-Mode Record184
Figure 26. Compiler Output for SAVECORE	92	Figure 54. One Logical Record Spanning Physical Blocks185
Figure 27. Link Editing a Segmented Program	94	Figure 55. First Four Blocks of SPAN-FILE186
Figure 28. Location of Sort Program in a Segmentation Structure	94	Figure 56. Advantage of S-Mode Records Over V-Mode Records186
Figure 29. Structures of the Actual Key102	Figure 57. Direct and Sequential Spanned Files on a Mass Storage Device	187
		Figure 58. Treatment of Varying Values in a Data Item of PICTURE S9202
		Figure 58.1. OPTION Control Statement to SORT/MERGE208

Figure 58.2. File Name and Default Symbolic Unit Names208
Figure 58.3. SUMMARY OF SORT-OPTION Operands208
Figure 59. Sample of GROUP INDICATE Clause and Resultant Execution Output	.213
Figure 60. Format of a Report Record When the CODE Clause is Specified . .	.214
Figure 61. Activating the NEXT GROUP Clause215
Figure 62. Calculating Record Lengths When Using the OCCURS Clause with the DEPENDING ON Option220

Figure 63. Table Structure in Virtual Storage221
Figure 64. Using the Symbolic Debugging Features to Debug the Program TESTRUN (Part 1 of 12)235
Figure 65. Sample Output of EXHIBIT Statement With the CHANGED NAMED Option249
Figure 66. Sample Dump Resulting from Abnormal Termination (Part 1 of 6) . .	.254
Figure 67. Track Format288
Figure 68. Communication Region in the Supervisor302
Figure 69. Sample OCR Program (Part 1 of 5)327

FEATURES OF THE PROGRAM PRODUCT DOS/VVS COMPILER

The IBM DOS/VVS COBOL Compiler includes the following features:

- Object Code:

- (1) Optimized Object Code -- which results, when specified, in up to 30% space saving in object program generated code and global tables as compared with Version 2 of the IBM DOS Full American National Standard COBOL Compiler. The space saved depends on the number of referenced procedure-names and branches, and on 01-level data names.
- (2) Double-Buffered ISAM -- allows faster sequential processing of indexed files.
- (3) The MOVE Statement and Comparisons -- when a MOVE statement or a comparison involves a one-byte literal, generated code for the move and the comparison saves object program space and compilation time.
- (4) DISPLAY Routines -- the DISPLAY routine has been split into subsets for efficient object program code.

- Alphabetized Cross-Reference Listing (SXREF) -- for easier reference to user-specified names in a program. SXREF performs up to 25 times faster than the source-ordered cross-reference (XREF) feature of Version 2 of the IBM DOS Full American National Standard COBOL Compiler. The larger the source program, the more that performance is improved. Total compilation time is up to 2 times faster.

- Debugging Facilities:

- (1) Symbolic Debug Feature -- which provides a symbolic formatted dump at abnormal termination, or a dynamic dump during program execution.
- (2) Flow Trace Option -- a formatted trace can be requested for a variable number of procedures executed before abnormal termination.
- (3) Statement Number Option -- identifies the COBOL statement

being executed at abnormal termination.

- (4) Expanded CLIST and SYM -- for detailed information about the Data Division and Procedure Division.
- (5) Relocation Factor -- can be requested to be included in addresses on the object code listing, for easier debugging.
- (6) Working-Storage Location and Size -- When CLIST and SYM are in effect, the starting address and size of Working-Storage are printed.
- (7) Syntax-Check Feature -- optionally provides a quick scan of the source program without producing object code. Syntax checking can be conditional or unconditional.
- (8) WHEN-COMPILED Special Register -- makes the date-and-time-compiled constant carried in the object module available to the object program. This special register is a programmer aid that provides a means of associating a compilation listing with both the object program and the output produced at execution time.

- Device Support -- the following devices can be specified in addition to devices supported by the IBM DOS Full American National Standard COBOL compilers:

5203,3203 -- line printers

3211 -- 150-character printer

3330,3340 -- mass storage (direct access) facilities

3540 -- Diskette input/output unit

3410,3420 -- tape utility devices

2560,3504,3505,3525,3881,3886,5425 -- advanced unit-record devices

- ASCII Support -- allows creation and retrieval of tape files written in the American National Standard Code for Information Interchange (ASCII).

- VSAM (Virtual Storage Access Method) Support -- provides fast storage and

retrieval of records, password protection, centralized and simplified data and space management, advanced error recovery facilities, plus system catalog. COBOL supports indexed (key-sequenced) files and sequential (entry-sequenced) files. Records can be fixed or variable in length.

- FIPS (Federal Information Processing Standard) Flagger -- issues messages identifying nonstandard elements in a COBOL source program. The FIPS Flagger makes it possible to ensure that COBOL clauses and statements in a DOS/VS COBOL source program conform to the Federal Information Processing Standard.
- Lister -- provides a specially formatted source listing with embedded cross-references for increased intelligibility and ease of use. A reformatted source deck is available as an option.
- Generic Key Facility for ISAM Files -- sequential record retrieval can be requested using a search argument comprised of a user-specified number of high-order characters (generic portion) of the NOMINAL KEY. The user need not specify a full or exact search key. This feature is supported via the START verb.
- MERGE Support -- combines from two to eight identically sequenced files on a set of specified keys and makes records available, in merged order, to an output procedure or a sequential output file.
- Verb profiles -- facilitates identifying and locating verbs in the COBOL source program. Options provide a verb summary or a verb cross-reference listing which includes the verb summary.
- Execution-time statistics -- maintains a count of the number of times each verb in the COBOL source program is executed during an individual program execution.

PART I

INTRODUCTION → INTRO

JOB DEFINITION → JOB DEF

JOB PROCESSING → JOB PROC

PREPARING COBOL PROGRAMS FOR PROCESSING → PREP PGMS

LIBRARIAN FUNCTIONS → LIB FCNS

INTERPRETING OUTPUT → OUTPUT

CALLING AND CALLED PROGRAMS → CALL PGMS

USING THE SEGMENTATION FEATURE → SEG

COBOL has undergone considerable refinement and standardization since 1959. A standard COBOL has been approved by the American National Standards Institute, an industry-wide association of computer manufacturers and users. This standard is called American National Standard COBOL. IBM Full American National Standard COBOL is compatible with American National Standard COBOL and includes a number of extensions to it as well.

An IBM COBOL program may be processed by the IBM DOS/VS System. Under control of the operating system, a set of COBOL source statements is translated to form a module. In order to be executed, the module in turn must be processed to form a phase. The reasons for this will become clear later. For now it is sufficient to note that the flow of a COBOL program through the operating system is from source statements to module to phase.

The DOS/VS System consists essentially of a control program and a number of processing programs, and data management.

CONTROL PROGRAM

The components of the control program are: the Supervisor, Job Control Processor, and the Initial Program Loader.

SUPERVISOR

The main function of the Supervisor is to provide an orderly and efficient flow of jobs through the operating system. (A job is some specified unit of work, such as the processing of a COBOL program.) The Supervisor loads into the computer the phases that are to be executed. During execution of the program, control usually alternates between the Supervisor and the processing program. The Supervisor, for example, handles all requests for input/output operations.

JOB CONTROL PROCESSOR

The primary function of the Job Control Processor is the processing of job control

statements. Job control statements describe the jobs to be performed and specify the programmer's requirements for each job. Job control statements are written by the programmer using the job control language. The use of job control statements and the rules for specifying them are discussed later.

INITIAL PROGRAM LOADER

The Initial Program Loader (IPL) routine loads the Supervisor into storage when system operation is initiated. Detailed information about the Initial Program Loader need not concern the COBOL programmer. Anyone interested in this material, however, can find it in the publication DOS/VS System Management Guide.

PROCESSING PROGRAMS

The processing programs include the COBOL compiler, service programs, and application programs.

SYSTEM SERVICE PROGRAMS

The system service programs provide the functions of generating the system, creating and maintaining the library sections, and editing programs into disk residence before execution. The system service programs are:

1. Linkage Editor. The Linkage Editor processes modules and incorporates them into phases. A single module can be edited to form a single phase, or several modules can be edited or linked together to form one executable phase. Moreover, a module to be processed by the Linkage Editor may be one that was just created (during the same job) or one that was created in a previous job and saved.

The programmer instructs the Linkage Editor to perform these functions through job control statements. In addition, there are several linkage editor control statements. Information on their use is given later.

2. Librarian. The Librarian consists of a group of programs used for generating the system, maintaining and reorganizing the disk library areas, and providing printed and punched output from the libraries. The system libraries are: the core image library, the relocatable library, the source statement library, and the procedure library. In addition, the Librarian supports private core image, relocatable, and source statement libraries. Detailed information on the Librarian is given later.

APPLICATION PROGRAMS

Application programs are usually programs written in a higher-level programming language (e.g., COBOL). All application programs within the Disk Operating System/Virtual Storage are executed under the supervision of the control program.

IBM-SUPPLIED PROCESSING PROGRAMS

The following are examples of IBM-supplied processing programs:

1. Language translators, e.g., DOS/VS COBOL, which translate source programs written in various languages into machine (or object) language.
2. Sort/Merge
3. Utilities

DATA MANAGEMENT

A third important class of components is data management routines. These are available for inclusion in problem programs to relieve the programmer of the detailed programming associated with the transfer of data between programs and auxiliary storage.

MULTIPROGRAMMING

Multiprogramming refers to the ability of the system to control more than one

program concurrently by interleaving their execution. This support is referred to as fixed partitioned multiprogramming, since the virtual address space is divided into fixed number of partitions. Each program occupies a contiguous area of storage. The amount of virtual storage allocated to programs to be executed may be determined when the system is generated, or it may be determined by the operator when the program is loaded into storage for execution.

BACKGROUND VS. FOREGROUND PROGRAMS

There are two types of problem programs in multiprogramming: background and foreground. Background and foreground programs are initiated by the Job Control Processor from batched-job input streams.

Background and foreground programs initiate and terminate independently of one another. Neither is aware of the other's status or existence.

The system is capable of concurrently operating one background program and four foreground programs. Priority for CPU processing is controlled by the Supervisor with foreground programs normally having priority over background programs. Control is taken away from a high priority program when that program encounters a condition that prevents continuation of processing, until a specified event has occurred. Control is taken away from a lower priority program when an event for which a higher priority program was waiting has been completed. Interruptions are received and processed by the Supervisor.

In a multiprogramming environment, the DOS/VS COBOL compiler can execute either in the background or the foreground. In systems that support the batched-job foreground and private core image library options, the Linkage Editor can execute in any foreground partition as well as in the background partition. To execute the DOS/VS COBOL compiler for the linkage editor in any foreground partition, a private core-image library is required. Additional information on executing the compiler and Linkage Editor in the foreground is contained in "Appendix F: System and Size Considerations." COBOL program phases can be executed as either background or foreground programs.

A job is a specified unit of work to be performed under control of the operating system. A typical job might be the processing of a COBOL program -- compiling source statements, editing the module produced to form a phase, and then executing the phase. Job definition -- the process of specifying the work to be done during a single job -- allows the programmer considerable flexibility. A job can include as many or as few job steps as the programmer desires.

JOB STEPS

A job step is exactly what the name implies -- one step in the processing of a job. Thus, in the job mentioned above, one job step is the compilation of source statements; another is the link editing of a module; another is the execution of a phase. In contrast to a job definition, the definition of a job step is fixed. Each job step involves the execution of a program, whether it be a program that is part of the Disk Operating System/Virtual Storage or a program that is written by the programmer. A compilation requires the execution of the DOS/VS COBOL compiler. Similarly, an editing implies the execution of the Linkage Editor. Finally, the execution of a phase is the execution of the problem program itself.

Compilation Job Steps

The compilation of a COBOL program may necessitate more than one job step (more than one execution of the DOS/VS COBOL compiler). In some cases, a COBOL program consists of a main program and one or more subprograms. To compile such a program, a separate job step must be specified for the main program and for each of the subprograms. Thus, the DOS/VS COBOL compiler is executed once for the main program and once for each subprogram. Each execution of the compiler produces a module. The separate modules can then be combined into one phase by a single job step -- the execution of the Linkage Editor.

For a COBOL program that consists of a main program and two subprograms, compilation and execution require five

steps: (1) compile (main program), (2) compile (first subprogram), (3) compile (second subprogram), (4) link edit (three modules combined into one phase), and (5) execute (phase). Figure 1 shows a sample structure of the job deck for these five job steps. Compilation and execution in three job steps -- compile, link edit, and execute -- is applicable only when the COBOL source program is a single main program.

JOB DEF

```

|// JOB PROG1
|.
|.
|.
|// EXEC FCOBOL
| {source deck - main program}
|/*
|.
|.
|.
|// EXEC FCOBOL
| {source deck - first subprogram}
|/*
|.
|.
|.
|// EXEC FCOBOL
| {source deck - second subprogram}
|/*
|.
|.
|.
|// EXEC LNKEDT
|.
|.
|.
|// EXEC

```

Figure 1. Sample Structure of Job Deck for Compiling, Link Editing, and Executing a Main Program and Two Subprograms

Multiphase Program Execution

The execution of a COBOL program has thus far been referred to as the execution of a phase. It is possible, however, to organize a COBOL program so that it is executed as two or more phases. Such a program is known as a multiphase program.

By definition, a phase is that portion of a program that is loaded into virtual storage by a single operation of the Supervisor. A COBOL program can be

executed as a single phase only if there is an area of virtual storage available to accommodate all of it. A program that is too large to be executed as a single phase must be structured as a multiphase program. The technique that enables the programmer to use subprograms that do not fit into virtual storage (along with the main program) is called overlay.

The number of phases in a COBOL program has no effect on the number of job steps required to process that program. As will be seen, the Linkage Editor can produce one or more phases in a single job step. Similarly, both single-phase and multiphase programs require only one execution job step. Phase execution is the execution of all phases that constitute one COBOL program.

Detailed information on overlay structures, as well as information on using the facilities of the operating system to create multiple phases and to execute them, can be found in the chapter "Calling and Called Programs."

TYPES OF JOBS

A typical job falls into one of several categories. A brief description of these categories follows; a complete discussion is found in the chapter "Preparing COBOL Programs for Processing."

Compile-Only: This type of job involves only the execution of the COBOL compiler. It is useful when checking for errors in COBOL source statements. A compile-only job is also used to produce a module that is to be further processed in a subsequent job.

A compile-only job can consist of one job step or several successive job steps.

Edit-Only: This type of job involves only the execution of the Linkage Editor. It is used primarily to combine modules produced in previous compile-only jobs, and to check that all cross references between modules have been resolved. The programmer can specify that all modules be combined to form one phase; or he can specify that some modules form one phase and that others form additional phases. The phase output produced as the result of an edit-only job can be retained for execution in a subsequent job.

Compile and Edit: This type of job combines the functions of the compile-only and the edit-only jobs. It requires the execution of both the COBOL compiler and the Linkage Editor. The job can include one or more compilations, resulting in one or more modules. The programmer can specify that the Linkage Editor process an or all of the modules just produced; in addition, he can specify that one or more previously produced modules be included in the linkage editor processing.

Execute-Only: This type of job involves the execution of a phase (or multiple phases) produced in a previous job. Once COBOL program has been compiled and edited successfully, it can be retained as one or more phases and executed whenever needed. This eliminates the need for recompiling and re-editing every time a COBOL program is to be executed.

Edit and Execute: This type of job combines the functions of the edit-only and the execute-only jobs. It requires the execution of both the Linkage Editor and the resulting phase(s).

Compile, Edit, and Execute: This type of job combines the functions of the compile and edit and the execute-only jobs. It calls for the execution of the COBOL compiler, the Linkage Editor, and the problem program; that is, the COBOL program is to be completely processed.

When considering the definition of his job, the programmer should be aware of the following: if a job step is cancelled during execution, the entire job is terminated; any remaining job steps are skipped. Thus, in a compile-edit-and-execute job, a failure in compilation precludes the editing of the module(s) and phase execution. Similarly, a failure in editing precludes phase execution.

For this reason, a job usually should (but need not) consist of related job steps only. For example, if two independent single-phase executions are included in one job, the failure of the first phase execution precludes the execution of the second phase. Defining each phase execution as a separate job would prevent this from happening. If successful execution of both phases can be guaranteed before the job is run, however, the programmer may prefer to include both executions in a single job.

JOB DEFINITION STATEMENTS

Once the programmer has decided the work to be done within his job and how many job steps are required to perform the job, he can then define his job by writing job control statements. Since these statements are usually punched in cards, the set of job control statements is referred to as a job deck. In addition to job control statements, the job deck can include input data for a program that is executed during a job step. For example, input data for the COBOL compiler -- the COBOL program to be compiled -- can be placed in the job deck.

The inclusion of input data in the job deck depends upon the manner in which the installation has assigned input/output devices. Job control statements are read from the unit named SYSRDR (system reader), which can be either a card reader, a magnetic tape unit, or a disk extent. Input to the processing programs is read from the unit named SYSIPT (system input), which also can be either a card reader, a magnetic tape unit, or a disk extent. The installation has the option of assigning either two separate devices for these units (one device for SYSRDR, a second device for SYSIPT) or one device to serve as both SYSRDR and SYSIPT. If two devices have been assigned, the job deck must consist of only job control statements; input data must be kept separate. If only one device has been assigned, input data must be included within the job deck.

There are four job control statements that are used for job definition: the JOB statement, the EXEC statement, the end-of-data statement (/*), and the end-of-job statement (/&). In this chapter, the discussion of these job control statements is limited to the function and use of each statement. The rules for writing each statement are given in the chapter "Preparing COBOL Programs for Processing."

The JOB statement indicates the beginning of control information for a job. The specified job name is stored in the communications region of the corresponding partition and is used by job accounting and to identify listings produced during execution of the job.

The JOB statement may be omitted, in which case the job name NONAME is stored in the communications region. If the JOB statement is present, it must contain a job name; otherwise, an error condition occurs.

The JOB statement is always printed in positions 1 through 72 on SYSLST and SYSLOG. The time-of-day and date are also printed. The JOB statement causes a skip to a new page before printing is started on SYSLST.

When a JOB statement is encountered, the job control program stores the job name from the JOB statement into the communications region. If the /& statement was omitted, the next JOB statement will cause control to be transferred to the end-of-job routine to simulate the /& statement.

The EXEC statement requests the execution of a program. Therefore, one EXEC statement is required for each job step within a job. The EXEC statement indicates the program that is to be executed (for example, the COBOL compiler, the Linkage Editor). As soon as the EXEC statement has been processed, the program indicated by the statement begins execution.

The end-of-data statement, also referred to as the /* (slash asterisk) statement, defines the end of a program's input data. When the data is included within the job deck (that is, SYSIPT and SYSRDR are the same device), the /* statement immediately follows the input data. For example, COBOL source statements would be placed immediately after the EXEC statement for the COBOL compiler; a /* statement would follow the last COBOL source statement.

Note: For an input file on a 5425 MFCU, the /* card must be followed by a blank card.

When input data is kept separate (that is, SYSIPT and SYSRDR are separate devices), the /* statement immediately follows each set of input data on SYSIPT. For example, if a job consists of two compilation job steps, an editing job step, and an execution job step, SYSIPT would contain the source statements for the first compilation followed by a /* statement, the source statements for the second compilation followed by a /* statement, any input data for the Linkage Editor followed by a /* statement, and perhaps some input data for the problem program followed by a /* statement.

The end-of-job statement, also referred to as the /& (slash ampersand) statement, defines the end of the job. A /& statement must appear as the last statement in the job deck.

JOB
DEF

OTHER JOB CONTROL STATEMENTS

The four job definition statements form the framework of the job deck. There are a number of other job control statements in the job control language; however, not all of them must appear in the job deck. The job control statements are summarized briefly in Table 1.

The double slash preceding each statement name identifies the statement as a job control statement. Most of the statements are used for data management -- creating, manipulating, and keeping track of data files. (Data files are externally stored collections of data from which data is read and onto which data is written.)

Table 1. Job Control Statements

Statement	Function
// ASSGN	Input/output assignments.
// CLOSE	Closes a logical unit assigned to magnetic tape.
// DATE	Provides a date for the Communication Region.
// DLAB	Disk file label information.
// DLBL	Disk file label information and VSAM file processing.
// EXEC	Execute program.
// EXTENT	Disk file extent.
// JOB	Beginning of control information for a job.
// LBLTYP	Reserves storage for label information.
// LISTIO	Lists input/output assignments.
// MTC	Controls operations on magnetic tape.
// OPTION	Specifies one or more job control options.
// PAUSE	Creates a pause for operator intervention.
// RESET	Resets input/output assignments to standard assignments.
// RSTRT	Restarts a checkpointed program.
// TLBL	Tape label information.
// TPLAB	Tape label information.
// UPSI	Sets user-program switches.
// VOL	Disk/tape label information.
// XTENT	Disk file extent.
/*	End-of-data-file or end-of-job-step.
/&	End-of-job.
*	Comments.

This chapter describes in greater detail the three types of job steps involved in processing a COBOL program. Once the reader becomes familiar with the information presented here, he should be able to write control statements by referring only to the next chapter, "Preparing COBOL Programs for Processing."

COMPILATION

Compilation is the execution of the COBOL compiler. The programmer requests compilation by placing in the job deck an EXEC statement that contains the program name FCOBOL, the name of the DOS/VS COBOL compiler. This is the EXEC FCOBOL statement.

Input to the compiler is a set of COBOL source statements, consisting of either a main program or a subprogram. Source statements must be punched in Extended Binary-Coded-Decimal Interchange Code (EBCDIC). The COBOL source statements are read from SYSIPT. The job deck is read from SYSRDR. If SYSRDR and SYSIPT are assigned to the same unit, the COBOL source statements should be placed after the EXEC COBOL statement in the job deck.

Output from the COBOL compiler is dependent upon the options specified when the system is generated. This output may include a listing of source statements exactly as they appear in the input deck. The source listing is produced on SYSLSLST. In addition, the module produced by the compiler may be written on SYSLNK, the linkage editor input unit, and punched on SYSRDR. Separate Data and/or Procedure Division maps, a symbolic cross-reference list, and diagnostic messages can also be produced. The format of compiler output is discussed and illustrated in the chapter "Interpreting Output."

The programmer can override any of the compiler options specified when the system was generated, or include some not previously specified, by using the OPTION control statement in the compile job step. Compiler options are discussed in detail in the chapter "Preparing COBOL Programs for Processing."

EDITING

Editing is the execution of the Linkage Editor. The programmer requests editing by placing in the job deck an EXEC statement that contains the program name LNKEDT, the name of the Linkage Editor. This is the EXEC LNKEDT statement.

Input to the Linkage Editor consists of a set of linkage editor control statements and one or more modules to be edited. These modules include any of the following:

1. Modules that were compiled previously in the job and placed at that time on the linkage editor input unit, SYSLNK.
2. Modules that were compiled in a previous job and saved as module decks. The module decks must be placed on SYSIPT. Linkage editor control statements are read from SYSRDR.
3. Modules that were compiled in a previous job step and cataloged in the relocatable library. The relocatable library is a collection of frequently used routines in the form of modules, that can be included in a program phase via the INCLUDE control statement in the linkage editor job step.

Output from the Linkage Editor consists of one or more phases. A phase may be an entire program or it may be part of an overlay structure (multiple phases).

A phase produced by the Linkage Editor can be executed immediately after it is produced (that is, in the job step immediately following the linkage editor job step), or it can be executed later, either in a subsequent job step of the same job or in a subsequent job. In either of the latter cases, the phase to be executed must be cataloged in the core image library. Such a phase can be retrieved in the execute job step by specifying the phase name in the EXEC statement, where phase name is the name under which it was cataloged. Otherwise, the phase output is retained only for the duration of one job step following the linkage editor job step. That is, if the module that was just link edited is to be executed in the next job step, it need not have been cataloged. An EXEC statement will cause the phase to be brought in from the temporary part of the

JOB
PROC

core image library and will begin execution. However, the next time such a module is to be executed, the linkage editor job step is required since the phase was not cataloged in the core image library.

If a private core image library is assigned, output from the Linkage Editor is placed in the private core image library (either permanently or temporarily) rather than in the resident system core image library. When execution of a program is requested and a private core image library is assigned, this library is searched first for the requested phase name and then the system core image library is searched.

In addition to the phase, the Linkage Editor produces a phase map on SYSLSST. Linkage editor diagnostic messages are also printed on SYSLSST. If the NOMAP option of the linkage editor ACTION control statement is specified, no phase map is produced and linkage editor diagnostic messages are listed on SYSLSST, if assigned. Otherwise, the diagnostic messages are listed on SYSLOG. The contents of the phase map are discussed and illustrated in the chapter "Interpreting Output."

Linkage editor control statements direct the execution of the Linkage Editor. Together with any module decks to be processed, they form the linkage editor input deck, which is read by the Job Control Processor from SYSIPT and written on SYSLNK.

There are four linkage editor control statements: the ACTION statement, the PHASE statement, the ENTRY statement, and the INCLUDE statement. These statements are discussed in the next chapter.

PHASE EXECUTION

Phase execution is the execution of the problem program, for example, the program written by the COBOL programmer. If the program is an overlay structure (multiple phase), the execution job step actually involves the execution of all the phases in the program.

The phase(s) to be executed must be contained in the core image library. The core image library is a collection of executable phases from which programs are loaded by the Supervisor. A phase is written in the temporary part of the core image library by the Linkage Editor at the time the phase is produced. It is permanently retained (cataloged) in the core image library, if the programmer has so requested, via the CATAL option in the OPTION control statement.

The programmer requests the execution of a phase by placing in the job deck an EXEC statement that specifies the name of the phase. However, if the phase to be executed was produced in the immediately preceding job step, it is not necessary to specify its name in the EXEC statement.

MULTIPHASE PROGRAMS

A COBOL program can be executed as a single phase as long as there is an area of virtual storage available to accommodate it. This area, known as the problem program area, must be large enough to contain the main program and all called subprograms. When a program is too large to be executed as a single phase, it must be structured as a multiphase program.

The overlay structure available to the COBOL programmer for multiphase programs is known as root phase overlay, and is used primarily for programs of three or more phases. One phase of the program is designated as the root phase (main program) and, as such, remains in the problem program area throughout the execution of the entire program. The other phases in the program -- subordinate phases -- are loaded into the problem program area as they are needed. A subordinate phase may overlay any previously loaded subordinate phase, but no subordinate phase may overlay the root phase. One or more subordinate phases can reside simultaneously in storage with the root phase.

Use of the linkage editor control statements needed to effect overlay are discussed in the chapter "Calling and Called Programs."

This chapter provides information about preparing COBOL source programs for compilation, link editing, and execution.

ASSIGNMENT OF INPUT/OUTPUT DEVICES

Almost all COBOL programs include input/output statements calling for data to be read from or written into data files stored on external devices. COBOL programs do not reference input/output devices by their actual physical address, but rather by their symbolic names. Thus, a COBOL program is dependent on the device type but not on the actual device address. Using VSAM, it is not even dependent on the device type. The COBOL programmer need only select the symbolic name of a device from a fixed set of symbolic names. At execution time, as a job control function, the symbolic name is associated with an actual physical device. The standard assignment of physical addresses to symbolic names may be made at system generation time. However, job control statements and operator commands can alter the standard device assignment before program execution. This is discussed later in this chapter.

Using DOS/VS, a logical unit may also be assigned to another logical unit or a general device class or specific device type. For more information on this, see DOS/VS System Management Guide and DOS/VS System Control Statements.

The symbolic names are divided into two classes: system logical units and programmer logical units.

The system logical units are used by the control program and by IBM-supplied processing programs. SYSIPT, SYSLST, SYSPCH, and SYSLOG can be implicitly referenced by certain COBOL procedural statements. Two additional names, SYSIN and SYSOUT, are defined for background program assignments. The names are valid only to the Job Control Processor, and cannot be referenced in the COBOL program. SYSIN can be used when SYSRDR and SYSIPT are the same device; SYSOUT must be used when SYSLST and SYSPCH are assigned to the same magnetic tape unit. A complete discussion of the assignment of the logical unit SYSCLB can be found in the publication DOS/VS System Control Statements.

Programmer logical units are those in the range SYS000 through SYS240 (depending on the number of partitions in the system) and are referred to in the COBOL source language ASSIGN clause.

A COBOL programmer uses the source language ASSIGN clause to assign a file used by his program to the appropriate symbolic name. Although symbolic names may be assigned to physical devices at system generation time, the programmer may alter these assignments at execution time by means of the ASSGN control statement. However, if the programmer wishes to use the assignments made at system generation time for his own data files in the COBOL program, ASSGN control statements are unnecessary.

Table 2 is a complete list of symbolic names and their usage.

PREP
PGMS

Table 2. Symbolic Names, Functions, and Permissible Device Types

Symbolic Name	Function	Permissible Device Types
SYSRDR	Input unit for control statements or commands.	Card reader Magnetic Tape unit Disk extent 3540 diskette
SYSIPT	Input unit for programs.	Card reader Magnetic tape unit Disk extent 3540 diskette
SYSPCH	Main unit for punched output.	Card punch Magnetic tape unit Disk extent 3540 diskette
SYSLST	Main unit for printed output.	Printer Magnetic tape unit Disk extent 3540 diskette
SYSLOG	Receives operator messages and logs in job control statements.	Printer keyboard Printer Display operator console
SYSLNK	Input to the Linkage Editor.	Disk extent
SYSRES	Contains the operating system, the core image library, relocatable library, source statement library, and procedure library.	Disk extent (2314,3330,3340)
SYSCLB	A private core image library.	Disk extent
SYS SLB	A private source statement library.	Disk extent
SYSRLE	A private relocatable library.	Disk extent
SYSIN	Must be used when SYSRDR and SYSIPT are assigned to the same disk extent. May be used when they are same disk extent. May be used when they are assigned to the same card reader or magnetic tape.	Disk Magnetic tape unit Card reader 3540 Diskette
SYSOUT	This name must be used when SYSPCH and SYSLST are assigned to the same magnetic tape unit. It must be assigned by the operator ASSGN command.	Magnetic tape unit
SYSmax	These units are available to the programmer as work files or for storing data files. They are called <u>programmer logical units</u> as opposed to the above-mentioned names which are always referred to as <u>system logical units</u> . The largest number of programmer logical units available in the system is 240 (SYS000 through SYS240, depending on number of partitions). The value of SYSmax is determined by the distribution of the programmer logical units among the partitions.	Any unit
SYSVIS	Holds virtual storage page data set.	Disk extent
SYSCAT	Holds the VSAM catalog.	Disk extent
SYSREC	Logs error records.	Disk extent



JOB CONTROL

The Job Control Processor for the Disk Operating System/Virtual Storage prepares the system for execution of programs in a batched job environment. Input to the Job Control Processor is in the form of job control statements and job control commands.

JOB CONTROL STATEMENTS

Job control statements are designed for an 80-column punched card format. Although certain restrictions must be observed, the statements are essentially free form. Job control statements conform to these rules:

1. Name. Two slashes (//) identify the statement as a job control statement. They must be in columns 1 and 2. At least one blank immediately follows the second slash.

Exceptions: The end-of-job statement contains /& in columns 1 and 2; the end-of-data-file statement contains /* in columns 1 and 2; the comment statement contains * in column 1 and a blank in column 2.
2. Operation. This identifies the operation to be performed. It can be up to eight characters long. At least one blank follows its last character.
3. Operand. This may be blank or may contain one or more entries separated by commas. The last term must be followed by a blank, unless its last character is in column 71.
4. Comments. Optional programmer comments must be separated from the operand by at least one space.

Continuation cards are not recognized by the Job Control Processor. For the exception to this rule, see the descriptions of the DLAB and TPLAB statements.

All job control statements are read from the device identified by the symbolic name SYSRDR.

Comments in Job Control Statements

Comment statements (i.e., statements preceded by an asterisk in column 1 followed by a blank) may be placed anywhere

in the job deck. The remainder of the card may contain any character from the EBCDIC set. Comment statements are designed for communication with the operator; accordingly, they are written on the console output unit, SYSLOG, in addition to being written on SYSRDR. If followed by a PAUSE control statement, the comment statement can be used to request operator action.

Statement Formats

The following notation is used in the statement formats:

1. All upper-case letters represent specifications that are to appear in the actual statement exactly as shown in the statement format. For example, JOB in the operation field of the JOB statement should be punched exactly as shown.
2. All lower-case letters represent generic terms that are to be replaced in the actual statement. For example, jobname is a generic term that should be replaced by the name that the programmer is giving his job.
3. Hyphens are used to join two or more words in order to form a single generic term. For example, device-address is one generic term.
4. Brackets are used to indicate that a specification is optional and is not always required in the statement. For example, [type] indicates that the programmer's replacement for the generic term, type, may or may not appear in the statement, depending on the programmer's requirements.
5. Braces enclosing stacked items indicate that a choice of one item must be made by the programmer. For example:

```
SYS  
PROG  
ALL  
SYSxxx
```

indicates that either SYS, PROG, ALL, or SYSxxx must appear in the actual statement.

6. Brackets enclosing stacked items indicate that a choice of one item may, but need not, be made by the programmer. For example:

```
,X'ss'
,ALT
```

indicates that either ,X'ss' or ,ALT but not both, may appear in the actual statement, or the specification can be omitted entirely.

7. All punctuation marks shown in the statement formats other than hyphens, brackets, and braces must be punched as shown. This includes periods, commas, and parentheses. For example, ,[date] means that the specification, if present in the statement, should consist of the programmer's replacement for the generic term date preceded by the comma with no intervening space. Even if the date is omitted, the comma must be punched as shown.
8. The ellipsis (...) indicates where repetition may occur at the programmer's option. The portion of the format that may be repeated is determined as follows:
- Scanning right to left, determine the bracket or brace delimiter immediately to the left of the ellipsis.
 - Continue scanning right to left and determine the logically matching bracket or brace delimiter.
 - The ellipsis applies to the words and punctuation between the pair of delimiters.

Sequence of Job Control Statements

The job deck for a specific job always begins with a JOB statement and ends with a /& (end-of-job) statement. A specific job consists of one or more job steps. The beginning of a job step is indicated by the appearance of an EXEC statement. When an EXEC statement is encountered, it initiates the execution of the job step, which includes all preceding control statements up to, but not including, a previous EXEC statement.

The only limitation on the sequence of statements within a job step is that which is discussed here for the label information statements.

The label statements must be in the order:

```
DLBL
EXTENT (one for each area or file in
the volume)
```

or

```
TLBL
```

and must immediately precede the EXEC statement to which they apply.

DESCRIPTION AND FORMATS OF JOB CONTROL STATEMENTS

This section contains descriptions and formats of job control statements.

Job control statements, with the exception of /*, /&, and *, contain two slashes in columns 1 and 2 to identify them.

JOB Statement

The JOB control statement indicates the beginning of control information for a job. The JOB control statement is in the following format:

```
[/ / JOB jobname
```

jobname

is a programmer-defined name consisting of from one to eight alphanumeric characters. Any user comments can appear on the JOB control statement following the jobname (through column 72). The time of day and date appear in columns 73 to 80 when the JOB statement is printed on SYSLSL. The time of day and date are also printed in columns 1 through 8 on the next line of SYSLOG.

If a job is restarted, the jobname must be identical to that used when the checkpoint was taken.

Note: The JOB statement resets the effect of all previously issued OPTION and ASSGN control statements.

PREP
PGMS

ASSGN Statement

The ASSGN control statement assigns a logical input/output unit to a physical device. An ASSGN control statement must be present in the job deck for each data file assigned to an external storage device in the COBOL program where these assignments differ from those established at system generation time. Data files are assigned to programmer logical units in COBOL by means of the source language ASSIGN clause. An ASSGN statement or command can also be used

- to unassign a logical unit to free it for assignment to another partition
- to ignore the assignment of a logical unit, that is, program references to the logical unit are ignored (useful in testing and certain rerun situations)
- to specify an alternate tape unit to be used when the capacity of the original is reached.

The assignment routines check the operands of the ASSGN statement/command for the relationship between the physical device, the logical unit, the type of assignment (permanent or temporary), etc. The following list summarizes the most pertinent items to remember when making assignments:

1. Assignments are effective only for the partition in which they are issued.
2. No physical device except DASD can be assigned to more than one active partition at the same time.
3. All system input and output file assignments to disk or diskette must be permanent.
4. SYSIN must be assigned if both SYSRDR and SYSIPT are to be assigned to the same extent.
5. SYSOUT cannot be assigned to disk or diskette; it must be a permanent assignment if assigned to tape.
6. SYSLNK must be assigned before issuing the LINK or CATAL option in the OPTION statement; otherwise, the option is ignored and the message 'PLEASE ASSIGN SYSLNK' is issued to the operator.
7. If SYSRDR, SYSIPT, SYSLST, or SYSPCH is assigned to tape or diskette, or disk when the system is generated, it will be unassigned by IPL. Such assignments can be made effective only with the job control ASSGN statement

or command, because ASSGN also opens the file.

8. Before a tape unit is assigned to SYSLST, SYSPCH, or SYSOUT, all previous assignments to this tape unit must be permanently unassigned. This may be done by using a DVCDN command instead.
9. The assignment of SYSLOG cannot be changed while a foreground partition is active.
10. SYSRES, SYSCAT, and SYSVIS can never be assigned by an ASSGN statement or command. An IPL is required to change these assignments.

The ASSGN control statement may also be used to change a system standard assignment for the duration of the job.

The format of the ASSGN control statement is as follows:

```
// ASSGN SYSxxx,device-address [ ,X'SS' ] [ ,ALT ]
```

SYSxxx

is one of the logical devices listed in Table 2.

Exception: SYSOUT must be assigned using the ASSGN job control command. Job control commands are described in detail in the publication DOS/VS System Control Statements.

device-address

allows three different formats:

X'cuu'

where c is the channel number and uu the unit number in hexadecimal notation. The values of 'cuu' are determined by each installation.

c = 0 for multiplexor channel,
1 through 6 for selector
channels 1 through 6.

uu = 00 to FE (0 to 254) in
hexadecimal.

UA

indicates that the logical unit is to be unassigned. Any source language input/output operation attempted on this device causes cancellation of the job.

IGN

indicates that the logical unit is to be unassigned. Each time a READ statement for the file assigned to IGN is encountered, control will be transferred to the imperative-statement following the AT END option. The IGN option is not valid for SYSRDR, SYSIPT, and SYSIN. This option is useful in program debugging since source language references to input files residing on symbolic units for which IGN has been specified are ignored. Any file for which the IGN option is used must be a sequential input file. Output files assigned with the IGN option are not supported by DOS/VS COBOL object programs.

X'ss'

is the device specification. It is used for specifying mode settings for 7-track and dual density 9-track tapes. If X'ss' is not specified, the system assumes the value specified at system generation for 7-track tapes and X'C0' for 9-track tapes. The possible specifications for X'ss' are shown in Figure 3.

ALT

must be specified in the control statement that assigns an alternate magnetic tape unit which is used when the capacity of the original assignment is reached. The specifications for the alternate unit must be the same as those of the original unit, since X'ss' cannot be specified. The characteristics of the alternate unit must be the same as those of the original unit. Multiple alternates can be assigned to a symbolic unit.

H1

indicates input hopper one for 2560 or 5425.

H2

indicates input hopper two for 2560 or 5425. H2 may only be assigned to SYSRDR, SYSIPT or SYSPCH.

Device assignments made by the ASSGN control statement are considered temporary. They are in effect until another ASSGN control statement or a RESET statement for that logical unit, or the next /& or JOB statement is read, whichever occurs first. If a RESET, /&, or JOB statement is encountered, the assignment reverts to the standard assignment established at system generation time plus any modification by an ASSGN command.

The COBOL programmer may assign only the programmer logical units (SYS000 through SYS240, depending on the number of partitions) to data files used in his program. For example, if the following ASSIGN clause is used,

```
SELECT IN-FILE ASSIGN TO SYS004-DA-2314-S
```

an ASSGN control statement must appear in the job deck which assigns SYS004 to a physical device if the physical device differs from the permanent assignment. In this case, the physical device must be a 2314 direct access device. An example of such a control statement is:

```
// ASSGN SYS004,X'00C'
```

Physical unit X'00C' was permanently assigned to a 2314 direct access device at system generation time.

Note: The ASSGN control statement is necessary only when the symbolic unit assignment is being made to a physical device address which differs from that established at system generation time.

"Appendix H: Sample Job Decks" contains illustrations of ASSGN statement usage.

PREP
PGMS

ss	Bytes per Inch	7-Track Tape		
		Parity	Translate Feature	Convert Feature
10	200	odd	off	on
20	200	even	off	off
28	200	even	on	off
30	200	odd	off	off
38	200	odd	on	off
50	556	odd	off	on
60	556	even	off	off
68	556	even	on	off
70	556	odd	off	off
78	556	odd	on	off
90	800	odd	off	on
A0	800	even	off	off
A8	800	even	on	off
B0	800	odd	off	off
B8	800	odd	on	off
		9-Track Tape		
C0	800	single density 9-track		
C0	1600	single density 9-track		
C0	1600	dual density 9-track		
C8	800	dual density 9-track		
D0	6250	single density 9-track		
D0	6250	dual density 9-track		

Figure 3. Possible Specifications for X'ss' in the ASSGN Control Statement

CLOSE Statement

The CLOSE control statement is used to close either a system or programmer logical unit assigned to tape. As a result of the CLOSE control statement, a standard end-of-volume label set is written and the tape is rewound and unloaded. The CLOSE statement applies only to a temporarily assigned logical unit, that is, a logical unit for which an ASSGN control statement has been specified within the same job. The format of the CLOSE control statement is as follows:

```
-----  
// CLOSE SYSxxx [ ,X'cuu' [,X'ss']  
                  ,UA  
                  ,IGN  
                  ,ALT  
-----
```

The logical unit can optionally be reassigned to another device, unassigned, or switched to an alternate unit.

Note that when SYSxxx is a system logical unit, one of the optional parameters must be specified. When closing a programmer logical unit, no optional parameter need be specified.

SYSxxx

may only be used for magnetic tape and may be specified as SYSPCH, SYSLST, SYSOUT, or SYS000 through SYS240, depending on the number of partitions.

X'cuu'

specifies that after the logical unit is closed, it will be assigned to the channel and unit specified. (See "ASSGN Control Statement" for an explanation of 'cuu'.) When reassigning a system logical unit, the new unit will be opened if it is either a mass storage device or a magnetic tape at load point.

X'ss'

represents device specification for mode settings on 7-track and 9-track tape. (See "ASSGN Control Statement" for an explanation of 'ss'.) If X'ss' is not specified, the mode settings remain unchanged.

UA

specifies that the logical unit is to be closed and unassigned.

IGN

specifies that the logical unit is to

be closed and unassigned with the ignore option. This operand is invalid for SYSRDR, SYSIPT, or SYSIN.

ALT

specifies that the logical unit is to be closed and an alternate unit is to be opened and used. This operand is valid only for system logical output units (SYSPCH, SYSLST, or SYSOUT) currently assigned to a magnetic tape unit.

DATE Statement

The DATE control statement contains a date that is put in the Communication Region of the Supervisor. A complete description of the fields of the Communication Region is given in "Appendix G: Communication Region." The DATE statement is in one of the following formats:

```
-----  
// DATE mm/dd/yy  
-----  
// DATE dd/mm/yy  
-----
```

where:

mm = month (01 to 12)
dd = day (01 to 31)
yy = year (00 to 99)

The format to be used is the format selected when the system was generated.

When the DATE statement is used, it applies only to the current job being executed. The Job Control Processor does not check the operand except to ensure that its length is eight characters. If no DATE statement is specified in the current job, the Job Control Processor supplies the date given in the last SET command. The SET command is discussed in detail in the publication DOS/VS System Control Statements.

A DATE statement should be included in every job deck that has as one of its job steps the execution of a COBOL program that utilizes the special register CURRENT-DATE, if the date desired is other than that designated in the previous SET command.

The DATE statement should be used at compile time so that the DATE-COMPILED paragraph is accurate and the WHEN-COMPILED special register is effective.

TLBL Statement

The TLBL control statement replaces the VOL and TPLAB combination used in previous versions of the system. However, the current system will continue to support these statements. The TLBL control statement contains file label information for tape label checking and writing. Its format follows:

```

|// TLBL filename,
|  ['file-identifier'],[date],
|  [file-serial-number],
|  [volume-sequence-number],
|  [file-sequence-number],
|  [generation-number],
|  [version-number]
|
```

filename

identifies the file to the control program. It can be from three to seven characters in length. If the following SELECT sentence appears in a COBOL program:

```
SELECT NEWFILE ASSIGN TO
SYS003-UT-2400-S-OUTFILE
```

the filename operand on control statements for this file must be OUTFILE. If the SELECT clause were coded:

```
SELECT NEWFILE ASSIGN TO
SYS003-UT-2400-S
```

the filename operand on the control statement for the file must be SYS003.

'file-identifier'

consists of from 1 to 17 characters, contained within apostrophes, indicating the name associated with the file on the volume. This operand may contain embedded blanks. If this operand is omitted on output files, the filename will be used. If this operand is omitted on input files, no checking will be done.

date

consists of from one to six characters, in the format yy/ddd, indicating the expiration date of the file for output or the creation date for input. (The day of the year may consist of from one to three characters.) For output files, a one to four character retention period (d-dddd) may be specified. If this operand is omitted, a 0-day retention period will be assumed for output files. For input files, no checking will be done if this operand is

omitted or if a retention period is specified.

file-serial-number

consists of from one to six characters indicating the volume serial number of the first (or only) reel of the file. If fewer than six characters are specified, the field will be right-justified and padded with zeros. If this operand is omitted on output files, the volume serial number of the first (or only) reel of the file will be used. If the operand is omitted on input files, no checking will be done.

volume-sequence-number

consists of from one to four characters in ascending order for each volume of a multivolume file. This number is incremented automatically by OPEN and CLOSE routines as required. If this operand is omitted on output files, BCD 0001 will be used. If omitted on input files, no checking is done.

file-sequence-number

consists of from one to four characters in ascending order for each file of a multivolume file. This number is incremented automatically by OPEN and CLOSE routines as required. If this operand is omitted on output files, BCD 0001 will be used. If it is omitted on input files, no checking will be done.

generation-number

consists of from one to four numeric characters that modify the file-identifier. If this operand is omitted on output files, BCD 0001 is used. If it is omitted on input files, no checking will be done.

version-number

consists of from one to two numeric characters that modify the generation number. If this operand is omitted on output files, BCD 01 will be used. If it is omitted on input files, no checking will be done.

Note: If a tape file with standard labels is opened two different ways in the same COBOL program, and that file resides on a multivolume file, the programmer should use two separate TLBL cards with different filenames specified on each.

DLBL Statement

The DLBL control statement, in conjunction with the EXTENT statement, replaces the VOL, DLAB, and XTENT combination used in previous versions of the Disk Operating System. The DLBL statement has the following format:

```
||// DLBL filename  
| ,['file-identifier'],[date],[codes]  
|
```

filename

identifies the file to the control program. It can be from three to seven characters long. If the following SELECT sentence appears in a COBOL program:

```
SELECT INFILE ASSIGN TO  
SYS005-DA-2314-A-INPUTA
```

the filename operand on control statements for this file must be INPUTA. If the SELECT sentence is coded:

```
SELECT INFILE ASSIGN TO  
SYS005-DA-2314-A
```

the filename operand on control statements for the file must be SYS005.

'file-identifier'

is the name associated with the file on the volume. This can consist of from 1 to 44 alphanumeric characters contained within apostrophes, including the file-identifier and, if used, generation-number and version-number of generation. If fewer than 44 characters are used, the field is left-justified and padded with blanks. If this operand is omitted, filename will be used.

date

consists of from one to six characters indicating either the retention period of the file in the format d through dddd (0-9999), or the absolute expiration date of the file in the format yy/ddd. When the d through dddd format is used, the file is retained for the number of days specified as dddd. For example, if date is specified as 31, the file will be retained a month from the day of creation. When the yy/ddd format is used, the file is retained until the day (ddd) in the year (yy) specified. For example, if date is specified as 73/200, the file will be retained

through the 200th day of the year 1973.

If date is omitted when the file is created, a 7-day retention period is assumed. If this operand is present for a file opened as INPUT or I-O, it is ignored.

codes

is a 2 to 4 character field indicating the type of file label, as follows:

SD = Sequential Disk
DA = Direct Access
ISC = Indexed Sequential using Load Create
ISE = Indexed Sequential using Load Extension, Add, or Retrieve
DU = 3540 Diskette
VSAM = VSAM file

If code is omitted, SD is assumed.

"Appendix H: Sample Job Decks" contains illustrations of DLBL statement usage.

See the section "Processing 3540 Diskette Unit Files" for the use of DLBL Cards for 3540 and the section "Virtual Storage Access Method" for use of DLBL cards for VSAM.

EXTENT Statement

The EXTENT control statement defines each area (or extent) of a DASD file -- a file assigned to a mass storage device. One or more EXTENT control statements must follow each DLBL statement.

The EXTENT control statement replaces the XTENT statement used in previous versions of the Disk Operating System. For more information on the XTENT statement, see DOS/VS System Control Statements.

The format of the EXTENT control statement is:

```
||// EXTENT [symbolic-unit],[serial-number]  
| ,[type],[sequence-number]  
| ,[relative-track],[number-of-tracks]  
| ,[split-cylinder-track],[B=bins]  
|
```

symbolic-unit

is a 6-character field indicating the symbolic unit (SYSxxx) of the volume for which this extent is effective. If this operand is omitted, the symbolic unit of the preceding EXTENT statement will be used. When specified, symbolic-unit may be any SYSxxx assigned to the device type

indicated in the SELECT sentence for the file. For example, if the following coding appears in a COBOL program:

```
SELECT OUTFILE ASSIGN TO
SYS004-DA-2314-A
```

the symbolic unit in the EXTENT control statement can be any SYSxxx assigned to a 2314 disk pack. The symbolic unit operand is not required for an IJSYSxx filename, where xx is IN, PH, LS, RS, SL, or RL. If SYSRDR or SYSIPT is assigned, this operand must be included.

serial-number

consists of from one to six characters indicating the volume serial number of the volume for which this extent is effective. If fewer than six characters are used, the field will be right-justified and padded with zeros. If this operand is omitted, the volume serial number of the preceding EXTENT control statement will be used. If no serial number was provided in the EXTENT control statement, the serial number will not be checked and it will be the programmer's responsibility if files are destroyed as a result of mounting the incorrect volume.

type

consists of one character indicating the type of the extent, as follows:

- 1 -- Data area (no split cylinder)
- 2 -- Overflow area (for an indexed file)
- 4 -- Index area (for an indexed file)
- 8 -- Data area (split cylinder)

If this operand is omitted, 1 is assumed.

sequence-number

consists of from one to three characters containing a decimal number from 0 to 255 indicating the sequence number of this extent within a multi-extent file. Extent sequence 0 is used for the master index of an indexed file. If the master index is not used, the first extent of an indexed file has the sequence number 1. The extent sequence number for all other types of files begins with 0. If this operand is omitted for the first extent of ISAM files, the extent will not be accepted. For SD or DA files, this operand is not required. For DA files this operand should be specified when using more than one EXTENT for a file. Direct files can have up to five extents. Indexed files can have up to eleven data

extents (nine prime, one cylinder index, one separate overflow).

relative-track

consists of from one to five characters indicating the sequential number of the track, relative to zero, where the data extent is to begin. If this field is omitted on an ISAM file, the extent will not be accepted. This field is not required for DA input or for SD input files (the extents from the file labels will be used).

Formulas for converting actual to relative track addresses (RT) and relative track to actual for the DASD devices follow.

Actual to Relative:

2311 10 x cylinder number + track
number = RT

2314 20 x cylinder number + track
or number = RT
2319

2321 1000 x subcell number + 100 x
strip number + 20 x block
number + track number = RT

3330 19 x cylinder number + track
number = RT

3340 12 x cylinder number + track
number = RT

Relative to Actual:

2311 $\frac{RT}{10}$ = quotient is cylinder
remainder is track

2314 $\frac{RT}{20}$ = quotient is cylinder,
or remainder is track
2319

3330 $\frac{RT}{19}$ = quotient is cylinder,
remainder is track

2321 $\frac{RT}{1000}$ = quotient is subcell,
remainder1

$\frac{\text{remainder1}}{100}$ = quotient is strip,
remainder2

$\frac{\text{remainder2}}{20}$ = quotient is block,
remainder is track

3340 $\frac{RT}{12}$ = quotient is cylinder,
remainder is track

PREP
PGMS

number-of-tracks

consists of from one to five characters indicating the number of tracks to be allocated to the file. For SD input files, this field may be omitted. The number of tracks for a split cylinder file must be a multiple of the number of cylinders specified for the file and the number of tracks specified for each cylinder.

split-cylinder-track

consists of from one to two characters, with a value of 0 through 19, indicating the upper track number for the split cylinder in SD files.

bins

consists of from one to two characters identifying the 2321 bin that the extent was created for, or on which the extent is currently located. If the field is one character, the creating bin is assumed to be zero. There is no need to specify a creating bin for SD or ISAM files. If this operand is omitted, bin 0 is assumed for both bins. If the operand is included and positional operands are omitted, only one comma is required preceding the keyword operand. If any operands preceding the bin specification are omitted, one comma for each operand is acceptable, but unnecessary.

Figure 4 shows examples of using the DLBL statement in conjunction with the EXTENT statement. "Appendix H: Sample Job Decks" contains illustrations of EXTENT statement usage.

VOL, DLAB, TPLAB AND XTENT STATEMENTS

These statements have been replaced by the DLBL, TLBL, and EXTENT statements, and, although they are still supported by the Disk Operating System, they cannot be used for 3330 or 3340 disk files, or for VSAM files. Details as to their usage can be found in DOS/VS System Control Statements. For their use with respect to COBOL, see IBM DOS Full American National Standard COBOL Programmer's Guide. When new label information statements are prepared, DLBL, TLBL, and EXTENT should be used.

LBLTYP Statement

The LBLTYP control statement defines the amount of storage to be reserved at linkage edit time in the problem program area of storage in order to process tape and nonsequential DASD file labels. It applies to both background and foreground object programs, and is required if the file contains standard labels.

The LBLTYP control statement immediately precedes the // EXEC LNKEDT statement in the job deck, with the exception of self-relocating programs for which it is instead submitted immediately preceding the // EXEC statement for the program. The format of the LBLTYP control statement is:

```

// LBLTYP { TAPE[(nn)] }
           { NSD(nn)   }

```

Direct file:

The following DLBL and EXTENT statements describe a direct file occupying 840 tracks, beginning on relative track 10.

```

// DLBL MASTER,,75/001,DA
// EXTENT SYS015,111111,1,0,10,840

```

Indexed file:

The following DLBL and EXTENT statements describe an indexed file on a 2314 occupying 100 tracks, beginning on relative track 1100. The first EXTENT allocates a 20-track cylinder index. The second EXTENT allocates a 80-track data area.

```

// DLBL MASTER,,75/001,ISC
// EXTENT SYS015,111111,4,1,1100,20
// EXTENT SYS015,111111,1,2,1120,80

```

Figure 4. Sample Label and File Extent Information for Mass Storage Files

TAPE[(nn)]

is used only if tape files requiring label information are to be processed and if no nonsequential DASD files are to be processed. nn is optional and is present only for future expansion. It is ignored by the Job Control Processor.

NSD(nn)

is used if any nonsequential DASD files are to be processed, regardless of other type files that are used. nn specifies the largest number of extents to be used for a single file.

LISTIO Statement

The LISTIO control statement causes the system to print a list of input/output assignments on SYSLST. The format of the LISTIO control statement is:

```

// LISTIO {
           SYS
           PROG
           BG
           F1
           F2
           F3
           F4
           ALL
           SYSxxx
           UNITS
           DOWN
           UA
           X'cuu'
}

```

SYS

causes the physical units assigned to all system logical units to be listed.

PROG

causes the physical units assigned to all background programmer logical units to be listed.

BG

lists the physical units assigned to all logical units of the background partition.

F1

causes the physical units assigned to all foreground-one logical units to be listed.

F2

causes the physical units assigned to all foreground-two logical units to be listed.

F3

causes the physical units assigned to all foreground-three logical units to be listed.

F4

causes the physical units assigned to all foreground-four logical units to be listed.

ALL

causes the physical units assigned to all logical units to be listed.

SYSxxx

causes the physical units assigned to the logical unit specified to be listed.

UNITS

causes the logical units assigned to all physical units to be listed.

DOWN

causes all physical units specified as inoperative to be listed.

UA

causes all physical units not currently assigned to a logical unit to be listed.

X'cuu'

causes the logical units assigned to the physical unit specified to be listed.

MTC Statement

The MTC control statement controls 2400 and 3400 series magnetic tape operations. The format is as follows:

```

// MTC opcode, SYSxxx[,nn]

```

opcode

specifies the operation to be performed. opcode can be chosen from the following:

BSF -- Backspace to tapemark

BSR -- Backspace to interrecord gap

ERG -- Erase gap (write blank tape)

FSF -- Forward space to tapemark

FSR -- Forward space to interrecord gap

RUN -- Rewind and unload



REW -- Rewind

WTM -- Write tapemark

SYSxxx

represents any logical unit assigned to magnetic tape upon which the MTC control statement is to operate.

[,nn]

is the decimal number (01 through 99) which, if specified, represents the number of times the operation is to be performed. If nn is omitted, the operation is performed once.

OPTION Statement

The OPTION control statement is used to specify one or more of the options of the Job Control Processor. The format of the OPTION statement is:

```
-----  
// OPTION option1[,option2]...  
-----
```

The order in which the selected options appear in the operand field is arbitrary. Options are reset to the standard established at system generation time upon encountering the next JOB statement or the /& statement.

The options are:

LOG

causes the listing of columns 1 through 80 of all control statements on SYSLST. If LOG is not the standard established at system generation time, control statements are not listed until a LOG option is encountered. Once a LOG option statement is read, logging continues from job step to job step until a NOLOG option is encountered or until either the JOB or /& control statement is encountered.

NOLOG

suppresses the listing of all control statements on SYSLST until a LOG option is encountered, or until either the JOB or /& control statement is encountered.

DUMP

causes a dump of the registers and virtual storage to be printed on SYSLST in the case of an abnormal program termination (such as a program check). Using the compiler SYMDMP, FLOW, or STATE features, it may not be necessary to use this option.

NODUMP

suppresses the DUMP option.

LINK

indicates that the object module is to be link edited. When the LINK option is used, the output of the COBOL compiler is written on SYSLNK. The LINK option must always precede an EXEC LNKEDT statement in the job deck. (CATAL also causes the LINK option to be set.) LINK is not acceptable to the Job Control Processor operating in the foreground unless the private core image library option is supported and a private core image library is assigned.

NOLINK

suppresses the LINK option. The COBOL compiler can also suppress the LINK option if the program contains an error that would preclude the successful execution of the program, or if SYNTAX is in effect, or if CSYNTAX is in effect and an E-level error is encountered.

DECK

causes the COBOL compiler to punch an object module on SYSPCH. If both DECK and LINK are specified, the output of the compiler is written on both SYSPCH and SYSLNK.¹

NODECK

suppresses the DECK option. The DECK option is also suppressed if SYNTAX is in effect, or if CSYNTAX is in effect and E-level errors exist.

LIST

causes the compiler to write the COBOL source statements on SYSLST. If lister is in effect, the LIST option is overridden; LISTER causes a listing regardless of whether LIST or NOLIST is specified.

NOLIST

suppresses the LIST option.

LISTX

causes the COBOL compiler to write a Procedure Division map on SYSLST. In addition, global tables, literal pools, register assignments, and procedure block assignments will be provided. You may want to use the CBL

¹The //option card options pertaining to the compiler will be suppressed if the "LISTER ONLY" option of lister is in effect. Otherwise, when "LISTER AND COMPILER" is in effect, the options specified will be in effect for compilation.

option CLIST (condensed list) in place of this.¹

NOLISTX
suppresses the LISTX option, as do the same conditions as cause DECK to be suppressed.

XREF
causes the COBOL compiler to write a symbolic cross-reference list on SYSLST. You may want to use the CBL option SXREF in place of this, or the lister cross-reference information for large COBOL programs.

NOXREF
suppresses the XREF option. SXREF also suppresses XREF, as do the same conditions as cause DECK to be suppressed.

SYM
causes the COBOL compiler to write a Data Division map on SYSLST. In addition, global tables, literal pools, register assignments, and procedure block assignments will be provided.¹

NOSYM
suppresses the SYM option.

ERRS
causes the COBOL compiler to write the diagnostic messages related to the source program on SYSLST.¹

NOERRS
suppresses the ERRS option. It does not suppress FIPS messages.

CATAL
causes the cataloging of a phase or program in the core image library upon completion of a linkage editor job step. CATAL also causes the LINK option to be set. CATAL is not accepted by the Job Control Processor operating in a batched-job foreground environment unless the private core image library option is supported and a private core image library is assigned.

STDLABEL
causes the standard label track to be cleared and all DASD or tape labels submitted after this point to be

¹The //option card options pertaining to the compiler will be suppressed if the "LISTER ONLY" option of lister is in effect. Otherwise, when "LISTER AND COMPILE" is in effect, the options specified will be in effect for compilation.

written on the standard label track. This option is reset to the USRLABEL option at end-of-job or end-of-job step. All file definition statements submitted after the STDLABEL option are available to any program in any area until another set of standard file definition statements is submitted. STDLABEL is not accepted by the Job Control Processor operating in a batched-job foreground environment. All file definition statements following OPTION STDLABEL are included in the standard file definition set until one of the following occurs:

- End-of-job step
- End-of-job
- OPTION USRLABEL is specified
- OPTION PARSTD is specified

USRLABEL

causes all DASD or tape labels submitted after this point to be written at the beginning of the user label track.

PARSTD

causes all DASD or tape labels submitted after this point to be written at the beginning of the partition standard label track. The PARSTD option is reset to the USRLABEL option at end-of-job or end-of-job step. All file definition statements submitted after the PARSTD option will be available to any program in the current partition until another set of partition standard file definition statements is submitted. All file definition statements submitted after OPTION PARSTD will be included in the standard file definition set until one of the following occurs:

- End-of-job step
- End-of-job
- OPTION USRLABEL is specified
- OPTION STDLABEL is specified

For a given filename, the sequence of search for label information during an OPEN is the USRLABEL area, followed by the PARSTD area, followed by the STDLABEL area.

Note: If NOLINK and NODECK are requested on the OPTION control statement and either SYMDMP or OPT is specified on the CBL card, the SYMDMP or OPT specification is ignored.



The options specified in the OPTION statement remain in effect until a contradictory option is encountered or until a JOB control statement is read. In the latter case, the options are reset to the standard that was established at system generation time.

Any assignment for SYSLNK, after the occurrence of the OPTION statement, cancels the LINK and CATAL options. These two options are also canceled after each occurrence of an EXEC statement with a blank operand.

PAUSE Statement

The PAUSE control statement allows for operator intervention between job steps. The format of the PAUSE control statement is:

```

[// PAUSE [comments]]

```

The PAUSE control statement is effective just before the next input control statement in the job deck is read. The PAUSE control statement always prints on SYSLOG and SYSLST.

An example of this statement is:

```

// PAUSE SAVE SYS004, SYS005, MOUNT
NEW TAPES

```

This sample statement instructs the operator to save the output tapes and mount two new tapes.

When the PAUSE statement is encountered by the Job Control Processor, processing is stopped in the partition until a response is given. The end/enter key causes processing to continue.

RESET Statement

The RESET control statement resets input/output assignments to the standard assignments. The standard assignments are those specified at system generation time plus any modifications made by the operator by means of the ASSGN command without the TEMP option. The RESET command is discussed in detail in the publication DOS/Vs System Control Statements. The format of the RESET statement is:

```

[// RESET { SYS
           { PROG
           { ALL
           { SYSxxx } } } ]

```

SYS resets all system logical units to their standard assignments.

PROG resets all programmer logical units to their standard assignments.

ALL resets all system and programmer logical units to their standard assignments.

SYSxxx resets the logical unit specified to its standard assignment.

RSTRT Statement

A restart facility is available for checkpoint programs. A programmer can use the source language RERUN clause in his program to cause checkpoint records to be written. This allows sufficient information to be stored so that program execution can be restarted at a specified point. The checkpoint information includes the registers, tape positioning information, a dump of virtual storage, and a restart address.

The restart facility allows the programmer to continue execution of an interrupted job at a point other than the beginning. The procedure is to submit a group of job control statements including a RSTRT control statement. The format is as follows:

```

[// RSTRT SYSxxx,nnnn,filename]

```

SYSxxx is the symbolic unit name of the 2400, 3410, 3420, 2311, 2314, 2319, 3330, or 3340 checkpoint file used for restarting. This unit must have been assigned previously.

nnnn

is the identification of the checkpoint record to be used for restarting. This serial number consists of four characters. It corresponds to the checkpoint identification used when the checkpoint was taken. The serial number is supplied by the checkpoint routine.

filename

is the symbolic name of the 2311, 2314, 2319, 3330, or 3340 disk checkpoint file used for restarting. It must be identical to the SYSxxx of the system-name specified in the RERUN clause.

When a checkpoint is taken, the completed checkpoint is noted on SYSLOG. Restarting can be done from any checkpoint record, not just the last. The jobname specified in the JOB statement must be identical to the jobname used when the checkpoint was taken. The proper input/output device assignments must precede the RSTRT control statement.

Assignment of input/output devices to symbolic unit names may vary from the initial assignment. Assignments are made for restarting jobs in the same manner as assignments are made for normal jobs.

See the chapter "Program Checkout" for further details on taking checkpoints and restarting a program for which checkpoints have been taken.

UPSI Statement

The UPSI control statement allows the programmer to set program switches that can be tested by problem programs at execution time. The UPSI control statement has the following format:

```
-----
// UPSI nnnnnnnn
-----
```

nnnnnnnn

consists of from one to eight characters of 0, 1, or X. Positions containing 1 are set to 1; positions containing X are unchanged. Unspecified rightmost positions are assumed to be X.

The UPSI byte is the 24th byte in the Communication Region of the Supervisor. A complete description of the fields of the Communication Region is given in "Appendix G: Communication Region." The Job Control Processor clears the UPSI byte to binary zeros before reading control statements for each job. When the UPSI control statement is read, the Job Control Processor sets these bits to the programmer's specifications. Any combination of the eight bits can be tested in the COBOL source program at execution time by means of the source language switches UPSI-0 through UPSI-7.

EXEC Statement

The EXEC statement (Execute Program or Procedure) indicates the end of control information for a job step and the beginning of execution of a program, in which case it must be the last command or statement processed before a job step is executed.

```
// EXEC [[PGM=]programname][,REAL][,SIZE]
        [PROC=procedurename]
```

PGM=programname

represents the name of the program in the core image library to be executed. The program name corresponds to the first or only phase of the program in the library. The program name can be one to eight alphanumeric characters (0-9, A-Z, #, \$, @). The first character must not be numeric.

If the program to be executed has just been processed by the linkage editor, the program name is omitted and the PGM keyword cannot be used.

REAL

indicates that the job step started by EXEC will be executed in real mode. If REAL is not specified the job step is always executed in virtual mode. REAL cannot be specified for programs using VSAM, the 3886, for ISAM programs using the ISAM interface program or, for programs compiled with the CBL option count.

SIZE=size

Size can be nK, AUTO or (AUTO, nK).

- (a) If specified with REAL, it indicates the size of that part of the real partition that will be needed by the job step's associated EXEC. The remaining part of the real partition is given to the page pool.

PREP
PGMS

If SIZE is omitted and REAL is specified, the whole real partition is used by the job step.

- (b) If used without REAL, it specifies that the virtual partition to be used by the job step is divided into two parts: the lower part with a size of nK will contain the program initiated with EXEC; the upper part serves as additional storage pool for other modules (for example, VSAM) required by the program in that partition. The program reserves the upper storage part for its needs by issuing GETVIS macros with the required amount of storage as parameter; it releases the storage by issuing FREEVIS macros.

If SIZE is omitted, the whole virtual partition is used for the job initiated with EXEC.

SIZE (without REAL) must always be specified for VSAM programs or for ISAM programs using the ISAM Interface Program (IIP), as well as for 3886 processing, and for programs compiled with the CBL option count.

If you specify SIZE=AUTO, the system automatically uses the information in the core image directory to calculate the size of the program to be loaded. If you specify SIZE=(AUTO,nK). The system adds nK bytes to the calculated length.

The following restrictions apply to n:

- n must not be larger than the size of the partition it refers to.
- n must be greater than zero.
- if n is not a multiple of 2, n+1 is used

PROC=procedurename
 represents the name of the procedure to be retrieved from the procedure library. The procedure name can be from one to eight alphameric characters, the first of which must be alphabetic.

For more information on cataloged procedures, as well as the use of overwrite statements and the rules that apply to temporary procedure modification, refer to the DOS/VS System Management Guide and the chapter "Librarian Functions" in this book.

CBL STATEMENT -- COBOL OPTION CONTROL CARD

Although some options for compilation are specified either at system generation time or in the OPTION control statement, the COBOL compiler provides an additional statement, the CBL statement, for the specification of compile-time options unique to COBOL.

The CBL card must be placed between the EXEC FCOBOL statement and the first statement in the CCBOL program. The CBL card cannot be continued. However, if specification of options will continue past column 71, multiple CBL cards may be used.

The options shown in the following format may appear in any order. No embedded blanks may appear in the operand field, and no comments should appear in the operand field. Underscoring indicates the default case. To change the defaults for your installation, see "Changing the Installation Defaults".

```

CBL [BUF=nnnnn] [ ,SEQ ] [ ,FLAGW ]
                [ ,NOSEQ ] [ ,FLAGE ]

[ ,SUPMAP ] [ ,SPACEn ] [ ,CLIST ]
[ ,NOSUPMAP ] [ ,NOCLIST ]

[ ,STXIT ] [ ,QUOTE ] [ ,TRUNC ] [ ,ZWB ]
[ ,NOSTXIT ] [ ,APOST ] [ ,NOTRUNC ] [ ,NOZWB ]

[ ,SXREF ] [ ,PMAP=h ] [ ,OPTIMIZE ]
[ ,NOSXREF ] [ ,NCOPTIMIZE ]
                [ ,OPT ]
                [ ,NCOPT ]

[ ,FLOW[=nn] ] [ ,STATE ] [ ,SYNTAX ]
                [ ,NOSTATE ] [ ,CSYNTAX ]
                [ ,NOSYNTAX ]

[ ,SYMDMP[=filename] ] [ ,VERBSUM ]
                        [ ,NOVERBSUM ]

[ ,VERBREF ] [ ,COUNT ]
[ ,NOVERBREF ] [ ,NOCOUNT ]

[ ,CATALR ] [ ,LIB ] [ ,VERB ] [ ,LVL= { A }
[ ,NOCATALR ] [ ,NOLIB ] [ ,NOVERB ] [ ,LVL= { B }
                                                { C }
                                                { D }
[ ,NOLVL ]
  
```

CBL

must begin in column 2 (column 1 must be blank) and be followed by at least one blank.

BUF=nnnnn

the BUF option specifies the amount of storage to be assigned to each compiler work file buffer. nnnnn is a decimal number from 512 to 32,767. If this option is not specified, 512 is assumed. The BUF option should be used to specify an optional blocksize (which will depend on the device type) for the workfiles. Usually, a larger blocksize will enhance the performance of the compiler. However, for any given BUF specification the compiler space requirements (over 60K) are increased by a factor of 6x(nnnnn-512).

**SEQ
NOSEQ**

indicates whether or not the compiler is to check the sequence of source statements. If SEQ is specified and a statement is not in sequence, it is flagged. If the lister feature is invoked, the source statements are resequenced automatically before the sequence check is performed.

**FLAGW
FLAGE**

determines which diagnostics the compiler will list. FLAGW indicates that all diagnostics will be listed (severity levels W, C, E, and D). FLAGE indicates that only those diagnostics with severity levels C, E, and D will be listed. This has no effect on FIPS messages.

**SUPMAP
NOSUPMAP**

causes the CLIST and LISTX options to be suppressed if an E-level diagnostic message is produced by the compiler. SUPMAP also suppresses the DECK option and no object module is produced.

SPACEN

indicates the type of spacing to be used on the output listing. n can be specified as either 1 (single spacing), 2 (double spacing), or 3 (triple spacing). If the SPACEN option is omitted, single spacing is provided. Single spacing is always in effect if the lister feature is invoked.

**CLIST
NOCLIST**

indicates that a condensed listing is to be produced. The condensed listing will contain only the address of the

first generated instruction for each verb in the Procedure Division. In addition, global tables, literal pools, register assignments, and procedure block assignments will be provided. The CLIST option overrides the LISTX or NOLISTX options. The LISTX or NOLISTX options are either established at system generation time or specified in the OPTION control statement.

**STXIT
NOSTXIT**

enables a USE AFTER STANDARD ERROR declarative to receive control when an input/output error occurs on a unit record device. The use of STXIT precludes the use of SYMDMP, STATE, and FLOW in the compiled program and in any other program link-edited with the compiled program, and vice versa.

**QUOTE
APOST**

QUOTE indicates to the compiler that the double quotation marks (") should be accepted as the character to delineate literals; APOST indicates that the apostrophe (') should be accepted instead. The compiler will generate the specified character for the figurative constant QUOTE(S).

**TRUNC
NOTRUNC**

applies only to COMPUTATIONAL receiving fields in MOVE statements and arithmetic expressions. If TRUNC is specified, extra code is generated to truncate the final intermediate result of the arithmetic expression, or the sending field in the MOVE statement, to the number of digits specified in the PICTURE clause of the COMPUTATIONAL receiving field. If NOTRUNC is specified, the compiler assumes that the data being manipulated conforms to PICTURE and USAGE specifications. The compiler then generates code to manipulate the data based on the size of the field in storage (halfword, etc.). TRUNC conforms to the American National Standard, while NOTRUNC leads to more efficient processing. This will occasionally cause dissimilar results for various sending fields because of the different code generated to perform the operation.

**ZWB
NOZWB**

determines if the compiler will generate code to strip the sign when comparing a signed external decimal field to an alphanumeric field. If ZWB is in effect, the signed external



decimal field is moved to an intermediate field and has its sign stripped before being compared to the alphanumeric field. ZWB conforms to the ANS standard, while NOZWB allows the user to test input numeric fields for SPACES to prevent abnormal termination.

SXREF
NOSXREF

causes the compiler to write an alphabetically-ordered cross-reference list on SYSLST. You may want to use the lister cross-reference information in place of this option for large COBOL programs, to decrease run time.

PMAP=h

enables the programmer to request a relocation factor "h". If the PMAP option is specified, the relocation factor is included in the addresses of the object code listing. The relocation factor "h" is a hexadecimal number of from one to eight digits. If the PMAP option is not specified, the relocation factor is assumed to be zero. When PMAP is specified in a segmented program, the listing for segments of priority higher than the segment limit (49, if the SEGMENT-LIMIT clause is not specified), will not be relocated. The PMAP option is effective only if LISTX or CLIST is specified.

OPTIMIZE
NOOPTIMIZE
OPT
NOOPT

OPTIMIZE (OPT) causes optimized object code to be generated by the compiler. The more efficient code generated considerably reduces the amount of space required by the object program.

This option cannot be used if either the symbolic debug option (SYMDMP), the statement number option (STATE), or the flow trace option (FLOW[=nn]) is requested.

FLOW[=nn]

provides the programmer with a formatted trace (i.e., a list containing the program identification and statement numbers) corresponding to a variable number of procedures executed prior to an abnormal termination. The value "nn" may range from 0 through 99. If "nn" is not specified, a value of 99 is assumed.

FLOW and STXIT, and FLOW and OPT are mutually exclusive options, i.e., only one may be in effect during a given

compilation. In addition, FLOW and STXIT are mutually exclusive at execution time. Additional information on the flow trace option can be found in the chapter "Symbolic Debugging Features."

STATE
NOSTATE

STATE provides the programmer with information about the statement being executed at the time of an abnormal termination of a job. It identifies the program containing the statement and provides the number of the statement and of the verb being executed. STATE and STXIT, STATE and SYMDMP, and STATE and OPT are mutually exclusive options, i.e., no more than one may be in effect during a given compilation. (However, the facilities provided by STATE automatically exist with SYMDMP.) In addition, STATE and STXIT are mutually exclusive at execution time. Additional information on the statement number option can be found in the chapter "Symbolic Debugging Features."

SYNTAX
CSYNTAX
NOSYNTAX

indicates whether the source text is to be scanned for syntax errors only and appropriate error messages are to be generated. For conditional syntax checking (CSYNTAX), a full compilation is produced so long as no messages exceed the C level. If one or more E-level or higher severity messages are produced, the compiler generates the messages but does not generate object text.

Notes:

1. When the SYNTAX option is in effect, all of the following compile-time options are suppressed:

OPTION control statement: LINK, DECK, XREF

CBL statement: SXREF, CLIST, COUNT, VERBREF, VERBSUM
2. When conditional syntax-checking is requested, the preceding options are suppressed only if one or more E- or D-level messages are generated.
3. Unconditional syntax checking is assumed if all of the following compile-time options are specified:

OPTION control statement: NOLINK,
NOXREF, NODECK

CBL statement: SUPMAP (and CLIST,
SXREF, VERBSUM, and VERBREF are
not specified)

4. Some compiler diagnostics do not appear when SYNTAX or CSYNTAX is in effect. These are listed in "Program Checkout."

SYMDMP[=filename]

indicates to the compiler that execution-time dumps might be requested for the program currently being compiled. If dumps are desired, the programmer must provide the required control cards at execution time.

Use of the symbolic debug option necessitates the presence of an additional work file, SYS005, at compile time. The "filename" parameter enables the programmer to specify a name for the SYS005 file that he can retain. If no filename is specified, IJSYS05 will be used. When several COBOL programs are link edited together, the "filename" parameter enables each to have a unique SYMDMP name. For a tape file, only unlabeled tapes may be used, and the filename in the SYMDMP=filename parameter is ignored.

SYMDMP and STXIT, SYMDMP and STATE, and SYMDMP and OPT are mutually exclusive options, i.e., no more than one may be in effect during a given compilation. (However, the facilities provided by STATE are automatically included with SYMDMP.) In addition, SYMDMP and STXIT are mutually exclusive at execution-time. Additional information on the symbolic debug option and the required execution-time control cards can be found in the chapter "Symbolic Debugging Features."

Note: If NODECK and NOLINK are requested on the OPTION control statement and either SYMDMP or OPT is specified on the CBL card, the SYMDMP or OPT specification is ignored.

CATALR
NOCATALR

causes the compiler to generate CATALR card images on the SYSPCH file if OPTION DECK is in effect during compilation. This will allow cataloging of the compiler produced object modules into the relocatable library. The module names in the CATALR cards adhere to the same rules as the phase names in the compiler

produced PHASE cards according to the segmentation and sort phase naming conventions (see the sections on Sort and Segmentation Features).

LIB
NOLIB

indicates that BASIS and/or COPY statements are in the source program. If either COPY or BASIS is present, LIB must be in effect. If COPY and/or BASIS statements are not present, use of the NOLIB option yields more efficient compiler processing.

VERB
NOVERB

indicates whether procedure-names and verb-names are to be listed with the associated code on the object-program listing. VERB has meaning only if LISTX, CLIST, VERBSUM, VERBREF, COUNT or READY TRACE are in effect. NOVERB yields more efficient compilation.

A
LVL= B
C
D
NOLVL

indicates whether the compiler should identify COBOL clauses and statements in a DOS/VS COBOL source program that do not conform to the Federal Information Processing Standard. FIPS recognizes four language levels: low, low-intermediate, high-intermediate and full. The FIPS Flagger provides four levels of flagging from low (A) to high (D) to conform to the four levels of the FIPS.

VERBSUM
NOVERBSUM

provides a brief summary of verbs used in the program and a count of how often each verb was used. This option provides the user with a quick search for specific types of statements. VERBSUM implies VERB.

VERBREF
NOVERBREF

provides a cross reference of all verbs used in the program. This option provides the programmer with a quick index to any verb used in the program. VERBREF implies VERB and VERBSUM.

COUNT
NOCOUNT

generates code to produce verb execution summaries at the end of problem program execution. Each verb is identified by procedure-name and by

PREP
PGMS

statement number, and the number of times it was used is indicated. In addition, the percentage of verb execution for each verb with respect to the execution of all verbs is given. A summary of all executable verbs used in a program and the number of times they are executed is provided. COUNT implies VERB.

Note: If COUNT and STXIT are desired, then either STXIT must be requested in the program unit requesting COUNT, or the program unit requesting COUNT must be entered before the program unit requesting STXIT. See the chapter entitled "Execution Statistics" for additional information on the COUNT option.

LST Statement -- New Compiler Option Card

The LST statement is used to invoke the lister, a portion of the compiler that processes programs written in American National Standard COBOL to produce a reformatted source code listing containing embedded cross-reference information, and uniform indenting conventions.

The LST option card can be placed anywhere between the EXEC statement and the first statement of the COBOL program. It may be placed between any other compiler option cards. The options shown in the following format may appear in any order. Underscoring indicates the default case.

```

-----
[ DECK, ] [ COPYPCH, ] [ LSTCOMP, ] [ PROC=1col, ]
[ NODECK ] [ NOCOPYPCH ] [ LSTONLY ] [ 2col ]
-----

```

LST must begin in column 2 (column 1 must be blank) and be followed by at least one blank.

DECK
NODECK indicates whether an updated source deck is to be produced as a result of the lister reformatting and/or the update BASIS library.

COPYPCH
NOCOPYPCH will punch updated and reformatted copy libraries as a permanent part of the source when DECK is specified. When no updated source deck is

requested, an updated and reformatted COPY library will be punched out.

LSTONLY
LSTCOMP

when LSTONLY is specified, the program will not be compiled, but a reformatted listing will be produced along with a deck if DECK has been specified. LSTCOMP will provide a source listing and will compile the program as part of the job step. LSTCOMP does not suppress CLIST.

PROC=1col
2col will list the Procedure Division in either single- or double-column format. At least 132 print positions are required on the printer for the double-column format.

For more details on the lister program, see the chapter entitled "Using the Lister Feature".

Mutually Exclusive Options

In some of the preceding descriptions of the CBL card options, restrictions have been placed on the use of one option in conjunction with others. It should be noted that if these restrictions are violated, the compiler ignores all but the last of the conflicting options specified. For this reason, if after a CBL card is coded the programmer decides to use a new option that is mutually exclusive with an option on the original CBL card, a new CBL card can be added rather than changing the original card.

Changing the Installation Defaults

In order to change the compiler default options to suit your installation, a new member, C.CBLOPTNS, must be added to the source statement library. This module must contain CBL option cards specifying the desired defaults. Resultant defaults may be overridden at compilation time by supplying a CBL card in the compiler input stream.

Significant Characters for Various Options

The DOS/VS COBOL compiler selects the valid options for processing by looking for three significant characters of each key

option word. When the keyword is identified, it is checked for the presence or absence of the prefix NO, as appropriate. The programmer can make the most efficient use of the CBL card by using the significant characters instead of the entire option. Table 3 lists the significant characters for each option.

Table 3. Significant Characters for Various Options

Option	Significant Characters
SEQ	SEQ
FLAGE(W)	LAG,LAGW
BUF	BUF
SPACE	ACE
PMP	PMA
SUPMAP	SUP
CLIST	CLI
TRUNC	TRU
APOST	APO
QUOTE	QUO
SXREF	SXR
STATE	STA
FLOW	FLO
LIB	LIB
SYMDMP	SYM
OPTIMIZE	OPT
SYNTAX	SYN
CSYNTAX	CSY
VERB	VER
ZWB	ZWB
LVL	LVL
COUNT	COU
VERBSUM	VERBSUM
VEREREF	VERBREF
STXIT	STX
DECK	DEC
COPYPCH	COP
LSTCOMP	STC
LSTONLY	STO
PROC	PRO

Note: SYM on the CBL card should not be confused with SYM on the OPTION card.

JOB CONTROL COMMANDS

Job control commands are distinguished from job control statements by the absence of // blank in positions 1 through 3 of each command. They permit the operator to adjust the system according to day-to-day operating conditions. This is particularly true in the area of device assignment, where the operator may need to (1) communicate to the system that a device is unavailable, or (2) designate a different device as the standard for a given symbolic unit. Therefore, these commands normally are not a part of the

regular job deck for a job. Job control commands tend to be effective across jobs, whereas job control statements are confined within a job.

Job control commands are discussed in detail in the publication DOS/VIS System Control Statements.

LINKAGE EDITOR CONTROL STATEMENTS

Object modules used as input to the Linkage Editor must include linkage editor control statements. There are four linkage editor control statements: PHASE, INCLUDE, ENTRY, and ACTION.

Linkage editor control statements initially enter the system through the device assigned to SYSRDR as part of the input job stream. PHASE and INCLUDE statements may also be present on SYSIPT or in the relocatable library. All four statements are verified for operation (INCLUDE, ACTION, ENTRY, or PHASE) and are copied to SYSLNK to become input when the Linkage Editor is executed.

Linkage editor control statements must be blank in position 1 of the statement. The operand field is terminated by the first blank position. It cannot extend beyond column 72.

The Linkage Editor is executed as a distinct job step. Figure 5 shows how the linkage editor function is performed as a job step in three kinds of operations.

1. Catalog Programs in Core Image Library. The linkage editor function is performed immediately preceding the operation that catalogs programs into the core image library. When the CATAL option is specified, programs edited by the Linkage Editor are cataloged in the core image library by the Librarian after the editing function is performed. The sequence of this operation is shown in Part A of Figure 5. Note that the input for the LNKEDT function could contain modules from the relocatable library instead of, or in addition to, those modules from the card reader, tape unit, or mass storage unit extent assigned to SYSIPT. This is accomplished by naming the module(s) to be copied from the relocatable library in an INCLUDE statement.



2. Load-and-Execute. The sequence of this operation is shown in Part B of Figure 5. Specifying OPTION LINK causes the Job Control Processor to open SYSLNK, and allows the Job Control Processor to place the object module(s) and linkage editor control statements on SYSLNK. As with the catalog operation, the input can consist of object modules from the relocatable library instead of, or in addition to, those modules from the card reader, tape unit, or disk extent assigned to SYSIPT. This is accomplished by specifying the name of the module to be included in the operand of an INCLUDE statement. After the object modules have been edited and placed in the core image library, the program is executed. The blank operand in the EXEC control statement indicates that the program that has just been link edited and temporarily stored in the core image library is to be executed.
3. Compile-and-Execute. Source modules can be compiled and then executed in a

single sequence of job steps. In order to do this, the COBOL compiler is directed to write the object module directly on SYSLNK. This is done by using the LINK option in the OPTION control statement. Upon completion of this output operation, the linkage editor function is performed. The program is link edited and temporarily stored in the core image library. The sequence of this operation is shown in Part C of Figure 5.

In each of the operations described in Figure 5, if a private core image library is assigned, output from the Linkage Editor will be placed (either permanently or temporarily) in the private core image library rather than in the system core image library. If the Linkage Editor is executed in a batched-job foreground partition, a private core image library must be assigned. Private core image libraries are a system generation option.

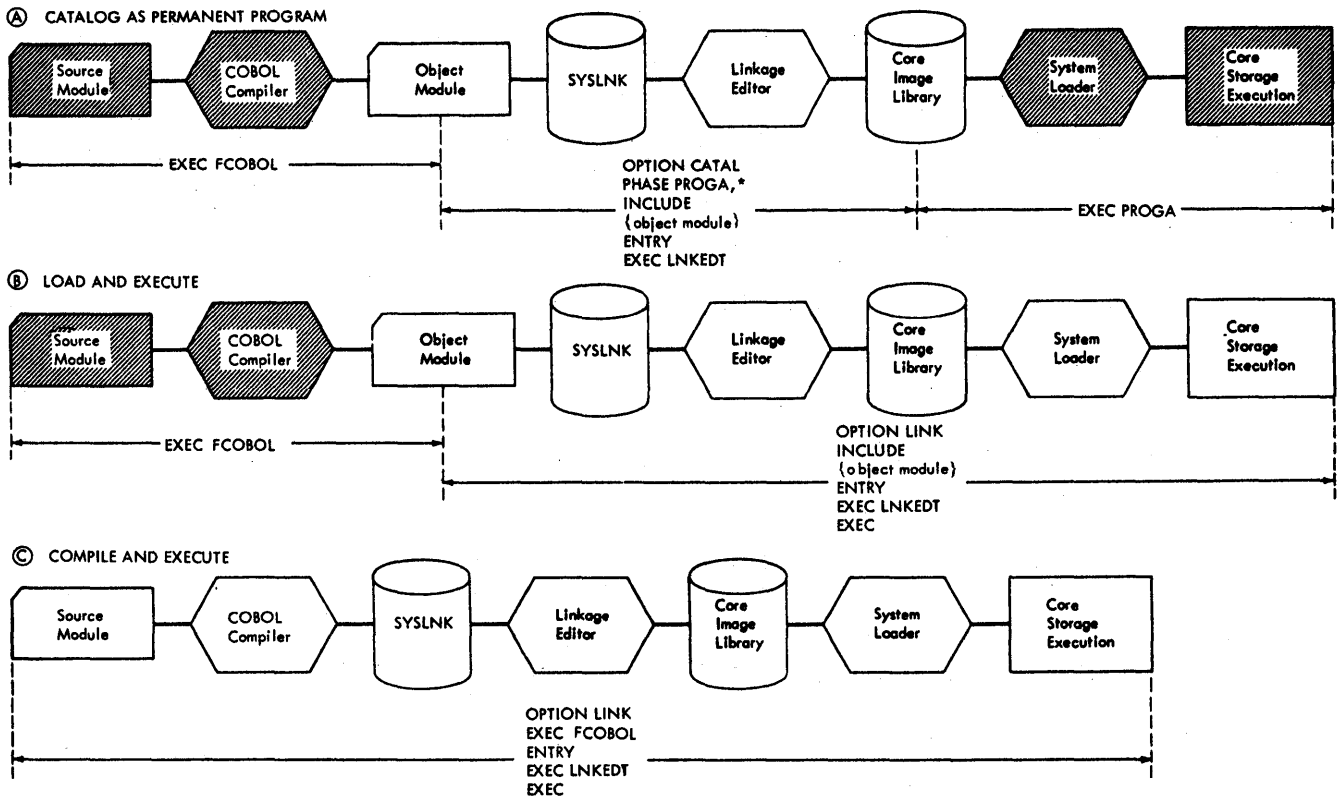


Figure 5. Job Definition -- Use of the Librarian

Control Statement Placement

The placement of linkage editor control statements is subject to the following rules:

1. The ACTION statement must be the first linkage editor control statement encountered in the input stream; otherwise, it is ignored.
2. The PHASE statement must precede each object module that is to begin a phase.
3. The INCLUDE statement must be specified for each object module that is to be included in a program phase.
4. A single ENTRY statement should follow the last object module when multiple object modules are processed in a single linkage editor run.

ACTION and ENTRY statements, when present, must be on SYSRDR. PHASE and INCLUDE statements may be present on SYSRDR, SYSIPT, or in the relocatable library.

PHASE Statement

The PHASE statement must be specified if the output of the Linkage Editor is to consist of more than one phase or if the program phase is to be cataloged in the core image library. Each object module that begins a phase must be preceded by a PHASE statement. Any object module not preceded by a PHASE statement will be included in the current phase.

The statement provides the Linkage Editor with a phase name and an origin point for the phase. The PHASE statement is in the following format:

```
PHASE name,origin[,NOAUTO]
```



name

is the symbolic name of the phase. It is the name under which the program phase is to be cataloged. This name does not have to be the name specified in the PROGRAM-ID paragraph in the Identification Division of the source program and, in the case of segmentation and/or sort, it should not be the same. It must consist of from one to eight alphanumeric characters. Phases that are to be executed in a segmentation and/or sort structure should have phase names of from five to eight alphanumeric characters, the first four of which should be the same. An asterisk cannot be used as the first character of a phase name. If no phase name is specified, a dummy phase name of PHASE*** is used and execution stops at end of compilation. The job is then cancelled.

origin

indicates to the Linkage Editor the starting address of this specific phase. An asterisk may be used as an origin specification to indicate that this phase is to follow the previous phase. This origin specification format of the PHASE statement covers all applications that do not include setting up overlay structures. See the chapter "Calling and Called Programs" for information on the PHASE statement for overlay applications.

NOAUTO

indicates that the Automatic Library Look-Up (AUTOLINK) feature is suppressed for both the private relocatable library and the system relocatable library. (The use of NOAUTO causes the AUTOLINK process to be suppressed for that phase only.) The AUTOLINK feature is discussed later in this chapter.

INCLUDE Statement

The INCLUDE statement must be specified for each object module deck or object module in the relocatable library that is to be included in a program phase. The format of the INCLUDE statement is as follows:

```
INCLUDE [module-name][,(namelist)]
```

The INCLUDE statement has two optional operands. When both operands are used, they must be in the prescribed order. When the first operand is omitted and the second

operand is used, a comma must precede the second operand.

module-name

must be specified when the object module is in the relocatable library. It is not specified when the module to be included is in the form of a card deck being entered from SYSIPT. module-name is the name under which the module was cataloged in the library, and must consist of from one to eight alphanumeric characters.

(namelist)

causes the Linkage Editor to construct a phase from the control sections specified in the list. Since control sections are of no interest to the COBOL programmer, users interested in this option should refer to the description of the INCLUDE statement in the publication DOS/V S System Control Statements.

ENTRY Statement

The ENTRY statement is required only if the programmer wishes to provide a specific entry point in the first phase produced by the Linkage Editor. When no ENTRY statement is provided, the Job Control Processor writes an ENTRY statement with a blank operand on SYSLNK to ensure that an ENTRY statement will be present to halt link editing. The transfer address will be the load address of the first phase. The ENTRY statement is described further in the publication DOS/V S System Control Statements.

ACTION Statement

The ACTION statement is used to indicate linkage editor options. When used, the statement must be the first linkage editor statement in the input stream. The format of the ACTION statement is as follows:

```
ACTION { CLEAR
        MAP
        NOMAP
        NOAUTO
        NOREL
        CANCEL
        BG
        F1
        F2
        F3
        F4 }
```

CLEAR
indicates that the entire temporary portion of the core image library will be set to binary zero before the beginning of the linkage editor function. CLEAR is a time-consuming function and should be used only when necessary.

MAP
indicates that SYSLST is available for diagnostic messages. In addition, a storage map is output on SYSLST.

NOMAP
indicates that SYSLST is unavailable when performing the link edit function. The mapping of storage is not performed, and all linkage editor diagnostic messages are listed on SYSLOG.

NOAUTO
suppresses the AUTOLINK function for both the private and system relocatable libraries during the link editing of the entire program. AUTOLINK is discussed later in this chapter.

CANCEL
causes an automatic cancellation of the job if any of the linkage editor errors 2100I through 2170I occur. These diagnostic messages can be found in the publication DOS/VS System Control Statements.

BG, F1, F2, F3, and F4
are options used to link edit a program for execution in a partition other than that in which the link edit function is taking place. See the publication DOS/VS System Control Statements.

NOREL
suppresses the relocating loader.

Link editing for a specific address is performed.

AUTOLINK FEATURE

If any references to external-names are still unresolved after all modules have been read from SYSLNK, SYSIPT, and/or the relocatable library, AUTOLINK collects each unresolved external reference from the phase. It then searches the private relocatable library (if SYSLPB has been assigned) and the system relocatable library for module names identical to the unresolved names and includes these modules in the program phase. This feature should not be suppressed (via PHASE or ACTION statements) in linkage editor job steps which include COBOL subroutines cataloged in the relocatable library. See the chapter "Calling and Called Programs" for additional details.

RELOCATING LOADER FEATURE

The relocating loader feature allows users to load single-phase and multi-phase programs at any valid problem program address in the system. Under this option, the linkage editor catalogs relocatable phases into the core image library, and the relocating loader in the supervisor assigns the absolute machine addresses that are necessary for program execution. This means the user need retain only one copy of the program in the core image library.

The relocating loader is an optional feature, and must be specified at system generation time.

Figure 6 illustrates options available during link-editing.



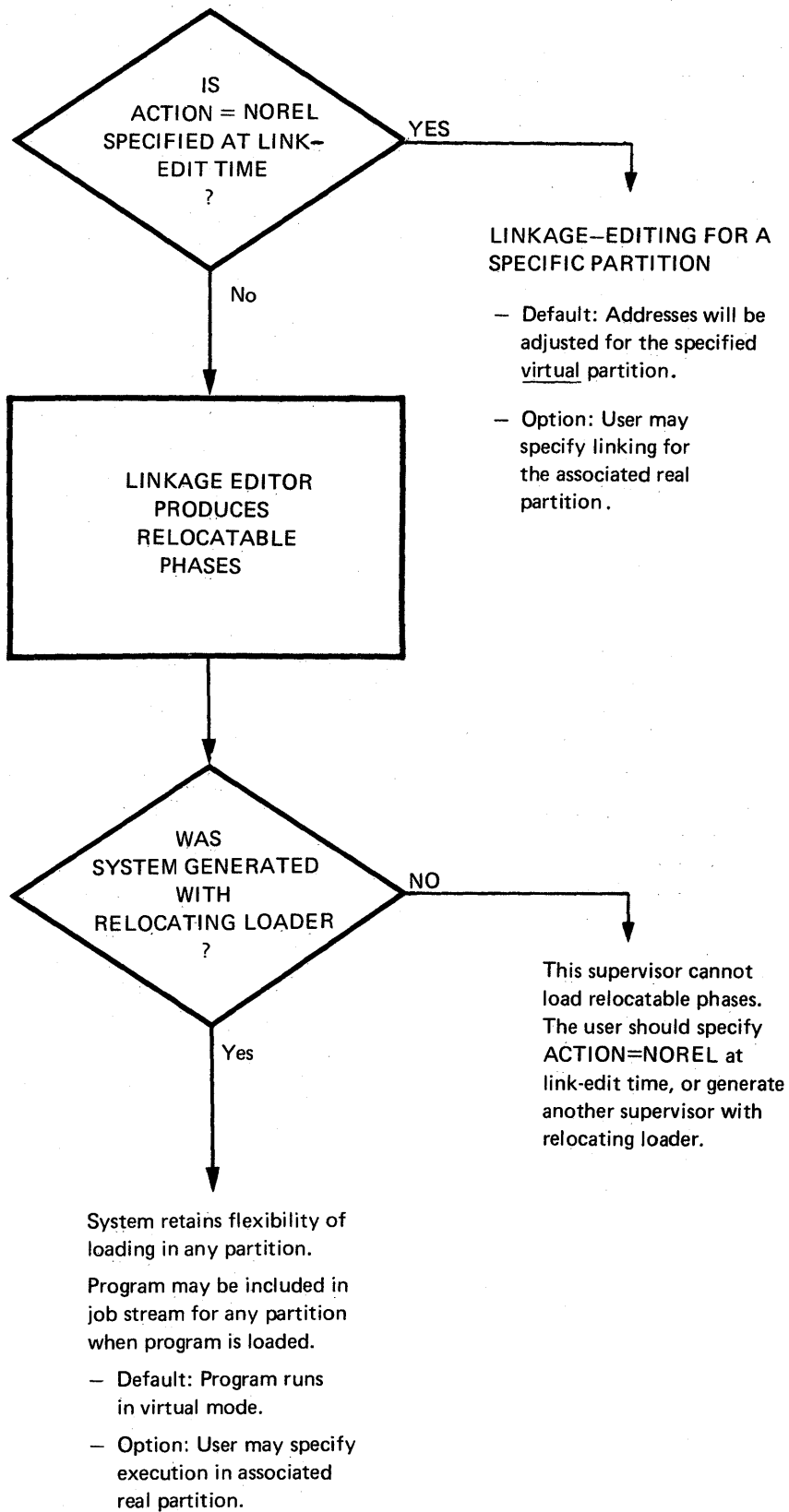


Figure 6. Options Available During Link-Editing

DOS/VS supports four libraries: the core image library, the relocatable library, the source statement library, and the procedure library. The core image, relocatable, and source statement libraries are classified as system libraries and private libraries. The procedure library exists only as a system library. The system residence device (SYSRES) contains the system libraries. Private libraries can be contained on separate disk packs. These libraries are discussed under "Private Libraries" in this chapter. Executable programs (core image format) are stored in the core image library; relocatable object modules are stored in the relocatable library; source language routines are stored in the source statement library; catalogued procedures are stored in the procedure library.

PLANNING THE LIBRARIES

The components of the DOS/VS system are shipped in three system libraries: the core image library, the relocatable library, and the source statement library. A fourth library -- the procedure library -- is available but it does not contain any information when the system is shipped. Most programs and procedures developed and used by your installation will also be stored in these libraries. In addition to the system libraries, DOS/VS supports private libraries which you can use to either substitute for or supplement the corresponding system libraries.

Planning the size, contents, and location of these libraries according to the needs of your installation is an essential part of the system generation procedure. Such detailed planning will ensure that:

- No disk space is wasted by components not required in your installation.
- The libraries are large enough to allow for future additions.
- The libraries are accessed by the system with maximum efficiency.

LIBRARIAN

The Librarian is a group of programs that perform three major functions:

1. Maintenance
2. Service
3. Copy

Maintenance functions are used to catalog (that is, add), delete, or rename components of the four libraries, condense libraries and directories, set a condense limit for an automatic condense function, reallocate directory and library extents, and update the source statement and procedure libraries.

The copy function is used either to completely or selectively copy the disk on which the system resides. Service functions are used to translate information from a particular library to printed (displayed) or punched output.

Only the catalog maintenance function of the Librarian is discussed in this publication for the four system libraries. In addition, the update function of the source statement library is discussed. A complete description of librarian functions can be found in the publication DOS/VS System Control Statements.

LIB
FCNS

CORE IMAGE LIBRARY

The core image library may contain any number of programs. Each program consists of one or more separate phases. Associated with the core image library is a core image directory which contains a unique descriptive entry for each phase in the core image library. These entries in the core image directory are used to locate and retrieve phases from the core image library.

Cataloging and Retrieving Program Phases -- Core Image Library

If a program is to be cataloged in the core image library, the job control statement // OPTION with the CATAL option

must be specified prior to the first linkage editor control card, and must precede the first PHASE card of the program to be cataloged. Upon successful completion of the linkage editor job step, output from the Linkage Editor is placed in the core image library as a permanent member. The program phase is cataloged under the name specified in the PHASE statement.

If a phase in the core image library is to be replaced by a new phase having the same name, only the catalog function need be used. The previously cataloged phase of the same name is implicitly deleted from the core image directory by the catalog function, and the space it occupies in the library can later be released by the condense function.

Note: The necessary ASSGN control statements must follow the // JOB control statement if the current assignments are not the following:

1. SYSRDR -- Card reader, tape unit, or disk extent
2. SYSIPT -- Card reader, tape unit, or disk extent
3. SYSLST -- Printer, tape unit, or disk extent
4. SYSLOG -- Printer keyboard
5. SYSLNK -- Disk extent

The following is an example of cataloging a single phase, FOURA, into the core image library. (The program phase FOURA can be executed in the next job step by specifying the // EXEC statement with a blank name field.)

```
// JOB CATALOG
// OPTION CATAL
  PHASE FOURA,*
  INCLUDE

      {object deck}
/*
// LBLTYP TAPE
// EXEC LNKEDT
// EXEC
/8
```

To compile, link edit, and catalog the phase FOURA into the core image library in the same job, the following job deck could be used:

```
// JOB CATALOG
// OPTION CATAL
  PHASE FOURA,*
// EXEC FCOBOL
```

{source deck}

```
/*
// EXEC LNKEDT
/*
/8
```

When the phase is executed in a subsequent job, the EXEC statement that calls for execution must specify FOURA, i.e., the name by which the phase has been cataloged.

```
// JOB EXJOB
// EXEC FOURA
/8
```

Phases can be in either non-relocatable or relocatable format. The non-relocatable phases are loaded at the address computed at link-edit time into a real or virtual partition. The load addresses and address constants of relocatable phases can be modified by the relocating loader. These phases can be loaded at a virtual address different from the one for which it was link-edited.

RELOCATABLE LIBRARY

The relocatable library contains any number of modules. Each module is a complete object deck in relocatable format. The purpose of the relocatable library is to allow the programmer to maintain frequently used routines in residence and combine them with other modules without recompiling.

Associated with the relocatable library is the relocatable directory. The directory contains a unique, descriptive entry for each module in the relocatable library. The entries in the relocatable directory are used to locate and retrieve modules in the relocatable library.

MAINTENANCE FUNCTIONS

To request a maintenance function for the relocatable library, the following control statement is used:

```
// EXEC MAINT
```

Cataloging a Module -- Relocatable Library

The catalog function adds a module to the relocatable library. A module in the relocatable library is the output of a complete COBOL compilation.

The catalog function implies a delete function. Thus, if a module exists in the relocatable library with the same name as a module to be cataloged, the module in the library is deleted by deleting reference to it in the relocatable directory.

The CATALR control statement is required to add a module to the relocatable library. The format of the CATALR control statement is:

```
CATALR module-name [,v.m]
```

module-name is the name by which the module is known to the control program. The module-name consists of from one to eight characters, the first of which must not be an asterisk.

v.m specifies the change level at which the module is to be cataloged. v may be any decimal number from 0 through 127. m may be any decimal number from 0 through 255. If this operand is omitted, a change level of 0.0 is assumed. A change level can be assigned only when a module is cataloged.

All control statements required to catalog an object module must be read from SYSIPT.

Note: If SYSRDR and/or SYSIPT are assigned to a tape unit, the MAINT program assumes that the tape is positioned to the first input record. The tape is not rewind at the end of the job. If a tape mark is found, MAINT assumes end-of-job.

The following is an example of compiling a source program and cataloging the resultant module in the relocatable library. The job deck is read from SYSIPT.

```
// JOB NINE
// OPTION DECK
// EXEC FCOBOL

{source deck}
/*
// PAUSE PLACE DECK AFTER CATALR CARD
// EXEC MAINT
   CATALR MOD9

(punched deck goes here)
/*
/;&
```

In the above example, as a result of the compile step, the object module is written

on SYSPCH. The next job step catalogs the object module (MOD9) into the relocatable library. Since the object module must be cataloged from SYSIPT, a message to the operator instructs him to place the object module on SYSIPT behind the CATALR statement.

The following is an example of cataloging two previously created object modules in the relocatable library:

```
// JOB EIGHT
// EXEC MAINT
   CATALR MOD8A

           {object deck}
   CATALR MOD8B

           {object deck}
/*
/;&
```

An additional capability of the system permits a programmer to compile a program and to catalog it to the system relocatable, or private relocatable, library in one continuous run. The programmer inserts a CATALR statement in his job control input stream preceding the compiler execute statement. The CATALR statement will be written on the SYSPCH file (tape or mass storage device) ahead of the compiler output when OPTION DECK is in effect. The programmer then reassigns the SYSPCH file as SYSIPT and executes the MAINT program to perform the catalog function. The output of the compilation (on tape or mass storage device) may be cataloged immediately or it may be cataloged at some later time. It can also be held after cataloging as backup of the compilation.

The preceding method is recommended for single-module object decks. In programs for which the compiler produces multimodule object decks (when segmentation and/or SORT are being used), it is necessary to use the CBL card CATALR option. This option causes a CATALR card to precede each object module.

SOURCE STATEMENT LIBRARY

The source statement library contains any number of books. Each book in the source statement library is composed of a sequence of source language statements. The purpose of the source statement library is to allow the COBOL programmer to initiate the compilation of a book into the source program by using the COPY statement or BASIS card.



Each book in the source statement library is classified as belonging to a specific sublibrary. Sublibraries are defined for three programming languages: Assembler, PL/I, and COBOL. Individual books are classified by sublibrary names. Therefore, books written in each of these languages may have the same name.

Associated with the source statement library is a source statement directory. The directory contains a unique descriptive entry for each book in the source statement library. The entries in the source statement directory are used to locate and retrieve books in the source statement library.

MAINTENANCE FUNCTIONS

To request a maintenance function for the source statement library, the following control statement must be used:

```
// EXEC MAINT
```

Cataloging a Book -- Source Statement Library

The CATALS control statement is required to add a book to a sublibrary of the source statement library.

A book added to a sublibrary of the source statement library is removed by using the delete function. When a book exists in a sublibrary with the same name as a book to be cataloged in that sublibrary, the existing book in the sublibrary is deleted. The following is the format of the CATALS control statement:

```
CATALS sublib.library-name[,v.m[,C]]
```

The operation field contains CATALS.

sublib

represents the sublibrary to which a book is to be cataloged and can be:

Any alphanumeric character (0-9, A-Z, #, \$, and @) representing source statement libraries. The characters A, C, E, and P have special uses:

A and E are used for the Assembler sublibrary

C is used for the COBOL sublibrary

P is used for POWER in PL/I

The sublib qualifier is required. If omitted, the operand will be flagged as invalid and no processing will be done on the book.

library-name

represents the name of the book to be cataloged. The library-name consists of from one to eight alphanumeric characters, the first of which must be alphabetic. It is the name the programmer uses to retrieve the book when using the source language COPY statement or BASIS card.

v.m

specifies the change level at which the book is to be cataloged. v may be any decimal number from 0 through 127; m may be any decimal number from 0 through 255. If this operand is omitted, a change level of 0.0 is assumed. The v.m operand becomes part of the entry in the directory for the specified book. Its value is incremented each time an update is performed on the book.

C

indicates that change level verification is required before updates are accepted for this book.

See the UPDATE control statement, discussed later in this chapter, for its relationship to the v.m and C operands of the CATALS control statement.

In addition to the CATALS control statement, a control statement of the following form must precede and follow the book to be cataloged:

```
BKEND [sublib.library-name],[SEQNCE],
      [count],[CMPRSD]
```

All operand entries are optional. When used, the entries must be in the prescribed order and need appear only in the BKEND statement preceding the book to be cataloged.

The first entry in the operand field is identical to the operand of the CATALS control statement.

SEQNCE

specifies that columns 76 to 80 of the card images constituting the book are to be checked for ascending sequence numbers. If an error is detected in the sequence checking, an error message is printed. The error can be corrected, and the book can be recataloged.

count

specifies the number of card images in the book. When the count operand is used, the card input is counted, beginning with preceding BKEND statement and including the subsequent BKEND statement. If an error is detected in the card count, an error message is printed. The error can be corrected, and the book can be recataloged.

CMPRSD

indicates that the book to be cataloged in the library is in compressed format as a result of CMPRSD having been specified when performing a PUNCH or DSPCH service function. These functions are described in the publication DOS/VS System Control Statements.

Card input for the catalog function is from the device assigned to SYSIPT. The CATALS control statement is also read from the device assigned to SYSIPT.

Frequently used Environment Division, Data Division, and Procedure Division entries can be cataloged in the COBOL sublibrary of the source statement library. A book in the source statement library might consist, for example, of a file

description of the Data Division or a paragraph of the Procedure Division.

The following is an example of cataloging a file description in the COBOL sublibrary of the source statement library.

```
// JOB ANYNAME
// EXEC MAINT
CATALS C.FILEA
BKEND C.FILEA
      BLOCK CONTAINS 13 RECORDS
      RECORD CONTAINS 120 CHARACTERS
      LABEL RECORDS ARE STANDARD
      DATA RECORD IS RECA.

BKEND
/*
/8
```

Retrieving a Cataloged Book -- COBOL COPY Statement: The preceding file description can be included in a COBOL source program by writing the following statement:

```
FD FILEB COPY FILEA.
```

Note that the library entry does not include FD or the file-name. It begins with the first clause that is actually to follow the file-name. This is true for all options of the COPY statement. However, data entries in the library may have a level number (01 or 77) identical to the level number of the data-name that precedes the COPY statement. In this case, all information about the library data-name is copied from the library and all references to the library data-name are replaced by the data-name in the program if the REPLACING option is specified. The change is made only for this program. The entry as it appears in the library remains unchanged. For example, assume the following data entry is cataloged under the library-name DATAR,

```
01 PAYFILE USAGE IS DISPLAY.
02 CALC PICTURE 99.
02 GRADE PICTURE 9
   OCCURS 1 DEPENDING ON CALC OF
   PAYFILE.
```

and the following statement is written in a COBOL source module:

```
01 GROSS COPY DATAR REPLACING PAYFILE
   BY GROSS.
```

The compiler interprets this as:

```
01 GROSS USAGE IS DISPLAY.
02 CALC PICTURE 99.
02 GRADE PICTURE 9
   OCCURS 1 DEPENDING ON CALC OF
   GROSS.
```



Note also that the library-name is used to identify the book in the library. It has no other use in the COBOL program.

Text cataloged in the source statement library must conform to COBOL margin restrictions.

The COBOL COPY statement is discussed in detail in the section "Extended Source Program Library Facility."

Updating Books -- Source Statement Library

The update function is used to make changes to properly identified statements within a book in the source statement library. Statements are identified in the identification field, columns 73 through 80, which is fixed in format as follows:

Columns 73-76	Program identification which must be constant throughout the book.
Columns 77-80	Sequence number of the statement within the book.

One or more source statements may be added to, deleted from, or replaced in a book in the library without the necessity of replacing the entire book. The update function also provides these facilities:

1. Resequencing statements within a book in the source statement library
2. Changing the change level (v.m) of the book
3. Adding or removing the change level requirement
4. Copying a book with optional retention of the old book with a new name (for backup purposes)

The UPDATE control statement is used for the update function and has the following format:

```
UPDATE sublib.library-name,[s.book1],  
[v.m],[nn]
```

The operation field contains UPDATE.

sublib
represents the sublibrary that contains the book to be updated. It may be any of the characters 0 through 9, A through Z, #, \$, or @.

s.book1
provides a temporary update option. The old book is renamed s.book1 and the updated book is named sublib.library-name. s indicates the sublibrary that contains the old, renamed book. It may be one of the characters 0 through 9, A through Z, #, \$, or @. If this operand is not specified, the old book is deleted.

v.m
represents the change level of the book to be updated. v may be any decimal number from 0 through 127; m may be any decimal number from 0 through 255. This operand must be present if change level verification is to be performed. Use of the optional entry C in the CATALS control statement at the time the book is cataloged in the library determines whether change level verification is required before updating. If the directory entry specifies that change level verification is not required before updating, the change level operand in the UPDATE control statement is ignored.

If the change level is verified, the change level in the book's directory entry is increased by 1 by the system for verification of the next update. If m is at its maximum value and an update is processed, m is reset to 0 and the value of v is increased by 1. If both v and m are at their maximum values and an update is processed, both v and m are reset to 0.

nn
represents the resequencing status required for the update. nn may be a 1- or 2-character decimal number from 1 through 10, or it may be the word NO. If nn is a decimal number, it represents the increment that will be used in resequencing the statements in the book. If nn is NO, the statements will not be resequenced. If nn is not specified, the statements will be resequenced with an increment of 1. When a book is resequenced, the sequence number of the first statement is 0000. For example, if a book is cataloged in the source statement library with sequence numbers ranging from 0010 through 1000 with increments of 5 for each statement:

and nn is not specified when the update function is performed, the book is resequenced with numbers 0000, 0001, 0002, ... etc.

and NO is specified, insertions, deletions, and/or replacements are made with no effect on the original sequence numbers.

and nn is specified as 2, the book is resequenced with numbers 0000, 0002, 0004, ... etc., regardless of the original sequencing of the book in the library or the sequence numbers of the added or replacement cards.

The UPDATE control statement is followed by ADD, DEL (delete), and/or REP (replace) control statements as required, followed by the terminating END statement. The ADD, DEL, REP, and END statements are identified as update control statements by a right parenthesis in the first position (column 1 in card format). This is a variation from the general librarian control statement format; thus, it clearly identifies these control statements as part of the update function.

ADD Statement: The ADD statement is used for the addition of source statements to a book. The format is:

```
[ ) ADD seq-no
```

ADD indicates that source statements following this statement are to be added to the book.

seq-no
represents the sequence number of the statement in the book after which the new statements are to be added. It may be any decimal number consisting of from one to four characters.

DEL Statement: The DEL statement causes the deletion of source statements from the book. The format is:

```
[ ) DEL first-seq-no[,last-seq-no]
```

DEL indicates that statements are to be deleted from the book.

first-seq-no
last-seq-no
represent the sequence numbers of the first and last statements of a section to be deleted. Each number may be a decimal number consisting of from one to four characters. If last-seq-no is not specified, the statement represented by first-seq-no is the only statement deleted.

REP Statement: The REP statement is used when replacement of source statements is required in a book. The format is:

```
[ ) REP first-seq-no[,last-seq-no]
```

REP indicates that source statements following this statement are to replace existing statements in a book.

first-seq-no
last-seq-no
represent the sequence numbers of the first and last statements of a section to be replaced. Each number may be a decimal number consisting of from one to four characters. Any number of new statements can be added to a book when a section is replaced. (The number of statements added need not equal the number of statements being replaced.)

Sequence number 9999 is the highest number acceptable for a statement to be updated. If the book is so large that statement sequence numbers have "wrapped around" (progressed from 9998, 9999, to 0000,0001), it will not be possible to update statements 0000 and 0001.

END Statement: This statement indicates the end of updates for a given book. The format is:

```
[ ) END [v.m[,C]]
```

v.m
represents the change level to be assigned to the book after it is updated; v may be any decimal number from 0 through 127. m may be any decimal number from 0 through 255. This operand provides an additional means of specifying the change level of a book in the library. (The other method is through the use of the v.m operand in the CATALS statement.)

C
indicates that change level verification is required before any subsequent updates for a given book.

If v.m is specified and C is omitted, the book does not require change level verification before a subsequent update. This feature removes a previously specified verification requirement for a particular book.

If both optional operands are omitted, the change level in the book's directory entry is increased as a result of the



update, and the verification requirement remains unchanged.

Control Statement Placement: Control statement input for the update function, read from the device assigned to SYSIN, must be in the following order:

1. The JOB control statement.
2. The ASSGN control statements, if the current assignments are not those required. The ASSGN control statements that can be used are SYSIN, SYSLST, and SYSLOG.
3. The EXEC MAINT control statement.
4. The UPDATE control statement.
5.) ADD,) DEL, or) REP statements with appropriate source statements.
6.) END statement.
7. The /* control statement.
8. The /& control statement, which is the last control statement of the job.

The source statement library can also be updated by using the DELETE and INSERT cards. These are discussed in "Extended Source Program Library Facility" in this chapter, and in the publication IBM DOS Full American National Standard COBOL.

UPDATE Function -- Invalid Operand Defaults

UPDATE Statement:

1. If the first or second operand is invalid, the statement is flagged, the book is not updated, and the remaining control statements are checked to determine their validity.
2. If change level verification is required and the incorrect change level is specified, the statement is flagged, the book is not updated, and the remaining control statements are checked to determine their validity.
3. If the resequencing operand is invalid, resequencing is done in increments of 1.

ADD, DEL, or REP Statements:

1. If there is an invalid operation or operand in an ADD, DEL, or REP statement, the statement is flagged, the book is not updated, and the remaining control statements are checked to determine their validity. All options of the UPDATE and END statements are ignored.
2. The second operand must be greater than the first operand in a DEL or REP statement. If not, the statement is considered invalid and is flagged, the book is not updated, and the remaining control statements are checked to determine their validity. All options of the UPDATE and END statements are ignored.
3. All updates to a book between an UPDATE statement and an END statement must be in ascending sequential order of statement sequence numbers. The first operand of a DEL or REP statement must be greater than the last operand of the preceding control statement. The operand of an ADD statement must be equal to or greater than the last operand of the preceding control statement. Consecutive ADD statements must not have the same operand. If these conditions are not met, the default is the same as for items 1 and 2.

END Statement: If the first operand of the END statement is invalid, the statement is flagged, both operands are ignored, and the book is updated as though no operands were specified. If the second operand is invalid, the statement is flagged, the operand is ignored, and the book is updated as though the second operand were not specified.

Out-of-Sequence Updates: If the source statements to be added to a book are not in sequence or do not contain sequence numbers, the book is updated, and a message indicating the error appears following the END statement. If the resequencing option has been specified in the UPDATE statement, the book is sequenced by the specified value, and subsequent updating is possible. If the resequencing option is not specified, the book is resequenced in increments of 1, and subsequent updating will be possible. If the resequencing option NO is specified, the book will be out of sequence, and subsequent updating may not be possible.

The Procedure Library

The procedure library is a new system library that may be used to store -- in card image format --

- Frequently used sets, procedures, of job control and linkage editor statements (basic support).
- Procedures additionally containing inline SYSIPT data, especially control statements for system utility and service programs (extended support). The inline SYSIPT data must be processed under control of the device-independent sequential IOCS or by IBM-supplied service programs and language translators.

The procedure library is part of SYSRES, so the maintenance and service functions available for the other DOS/VS libraries will also support the procedure library.

Cataloged procedures may be included in the job control input stream by a job control statement and temporarily modified by overwrite statements. For more details on cataloged procedures, see DOS/VS System Control Statements.

MAINT, PROCEDURE LIBRARY

To request a maintenance function for the procedure library, use the following EXEC control statement:

```
// EXEC MAINT
```

One or more of the maintenance functions (catalog, delete, rename, condense, set condense limit, or reallocate) can be requested within a single run. Any number of procedures within the procedure library can be acted upon in this run. Further, one or more of the maintenance functions for either of the other three libraries (core image, source statement, or relocatable) can be requested within this run, for the same MAINT program maintains all four libraries.

Catalog

The control statement required to add a procedure to the procedure library is the CATALP statement. Any number of procedures may be cataloged in a single run. Each procedure must immediately follow the respective CATALP statement.

Statement Format:

```
CATALP procedurename[,VM=v.m][,EOP=yy]  
      NO  
      ,DATA=YES
```

Each control statement in the procedure library should have a unique identity. This identity is required to modify the job stream at execution time. Therefore, when cataloging, identify each control statement in columns 73-79 (blanks may be embedded).

procedurename
represents the name of the procedure to be cataloged. The procedurename consists of one to eight alphanumeric characters, the first of which must be alphabetic. It must not be ALL.

VM=v.m
specifies the change level at which the procedure is to be cataloged. v may be any decimal number from 0-127. m may be any decimal number from 0-255. If this operand is omitted, a change level of 0.0 is assumed.

A change level can be assigned only when a procedure is cataloged. The change level is displayed and punched by the service functions.

EOP=yy
specifies a two-character end-of-procedure delimiter. The EOP parameter can be any combination of characters except /*, /%, //; it must not contain a blank or a comma. The system assumes /* as default end-of-procedure delimiter. Otherwise you can omit the EOP parameter.

DATA=YES
specifies that a procedure contains SYSIPT inline data.

These procedures can only be executed in the extended procedure support.

A procedure to be cataloged into the procedure library may consist of Job Control and linkage editor statements and, if the supervisor was generated with the SYSFIL option, additional control statements for IBM-supplied control and service programs and data processed under control of the device-independent sequential IOCS. The end of a procedure is indicated by the /* end-of-procedure delimiter or by the end-of-procedure delimiter as specified in the EOP parameter.

If SYSIN is assigned to a tape unit, the MAINT program assumes that the tape is positioned to the first input record. The tape is not rewound at the end of job.



Control statement input for the catalog function, read from the properly assigned device (usually SYSIN), is:

1. the JOB control statement, followed by
2. the ASSGN control statements, if the current assignments are not those required. The ASSGN statements that can be used are SYSIN, SYSLST, and SYSLOG. The ASSGN statements are followed by
3. the EXEC MAINT control statement, followed by
4. the CATALP control statement(s), followed by
5. the module to be cataloged, followed by
6. the /* control statement if other job steps are to follow, or
7. the /& control statement, which is the last control statement of the job.

For example:

```
// JOB CATPROC
.
.
ASSGN control statements,
if required
.
.
// EXEC MAINT
CATALP PROCA,EOP=AA,DATA=YES
.
.
control statements
.
.
SYSIPT inline data
.
.
/* END OF SYSIPT DATA
.
.
control statements
.
.
AA END OF PROCEDURE
```

The following restrictions apply when you catalog procedures to the procedure library:

1. A cataloged procedure cannot contain control statements or SYSIPT data for more than one job.
2. If the cataloged control statements include the JOB statement, you must not have a JOB statement when you retrieve the procedure through the

EXEC statement. Conversely, if the JOB statement is not cataloged, a JOB statement must precede the EXEC statement that retrieves the procedure.

3. A cataloged procedure must not include any of the following control statements because they are not accepted when the procedure is processed:

```
// ASSGN SYSRDR,X'cuu'
// RESET SYS
// RESET ALL
// RESET SYSRDR
// CLOSE SYSRDR,X'cuu'
// ASSGN SYSIPT,X'cuu'
// RESET SYSIPT
// CLOSE SYSIPT,X'cuu'
```

only if SYSIPT data is included

4. Cataloged procedures cannot be nested, that is, a cataloged procedure cannot contain an EXEC statement that invokes another cataloged procedure.

Note: Maintenance cannot be performed in the background partition on the procedure library while a foreground partition is using the library.

PSERV, PROCEDURE LIBRARY

To request a service function for the procedure library, use the following EXEC control statement:

```
// EXEC PSERV
```

One or more of the three service functions can be requested within a single run. Any number of procedures within the procedure library can be acted upon in this run.

CALLING CATALOGED PROCEDURES

A cataloged procedure is called by a job that appears in the input stream or via an operator command. The job must consist of a JOB statement and an EXEC statement that specifies the cataloged procedure name. For example:

```
// EXEC PROC=VCOCBLG
```

The programmer can write cataloged procedures which incorporate job control he used frequently. For example, the programmer may wish to catalog a procedure

for compiling, link-editing, and executing a program. It is particularly useful for compiling in a low-priority test partition to which no card reader has been assigned. Using cataloged procedures, the operator can execute via the EXEC statement a cataloged procedure from the console.

PRIVATE LIBRARIES

Private libraries are desirable in the system to permit some libraries to be located on a disk pack other than the one used by SYSRES.

Private libraries are supported for the core image library, the relocatable library, and the source statement library, on the 2311, 2314, 2319, 3330, and 3340 mass storage devices. However, the following restrictions apply:

1. The private library must be on the same type of disk device as SYSRES; the private core-image library can be on a type of device other than the one SYSRES is on.
2. Reference may be made to a private core image library only if SYSCLB is assigned. If SYSCLB is assigned, the system core image library cannot be changed.
3. Reference may be made to a private relocatable library only if SYSRLB is assigned. If SYSRLB is assigned, the system relocatable library cannot be changed.

4. Reference may be made to a private source statement library only if SYSSLB is assigned. If SYSSLB is assigned, the system source statement library cannot be changed.
5. Private libraries cannot be reallocated.
6. The COPY function is not effective for private libraries except when they are being created.

An unlimited number of private libraries is possible. However, each must be distinguished by a unique file identification in the DLBL statement for the library. No more than one private relocatable library and one private source statement library may be assigned in a given job.

The creation and maintenance of private libraries is discussed in the publication DOS/VS System Control Statements.

Determining the Location of the Libraries

Having decided which libraries you want in your system, you must determine where on the available devices these libraries are to be placed. All system libraries must reside in the SYSRES extent of the system disk pack in a predefined sequence (Figure 7). Although it is theoretically possible to have private libraries on the system pack (outside the SYSRES extent), this is not recommended because it involves increased movement of the disk arm.

LIB
FCNS

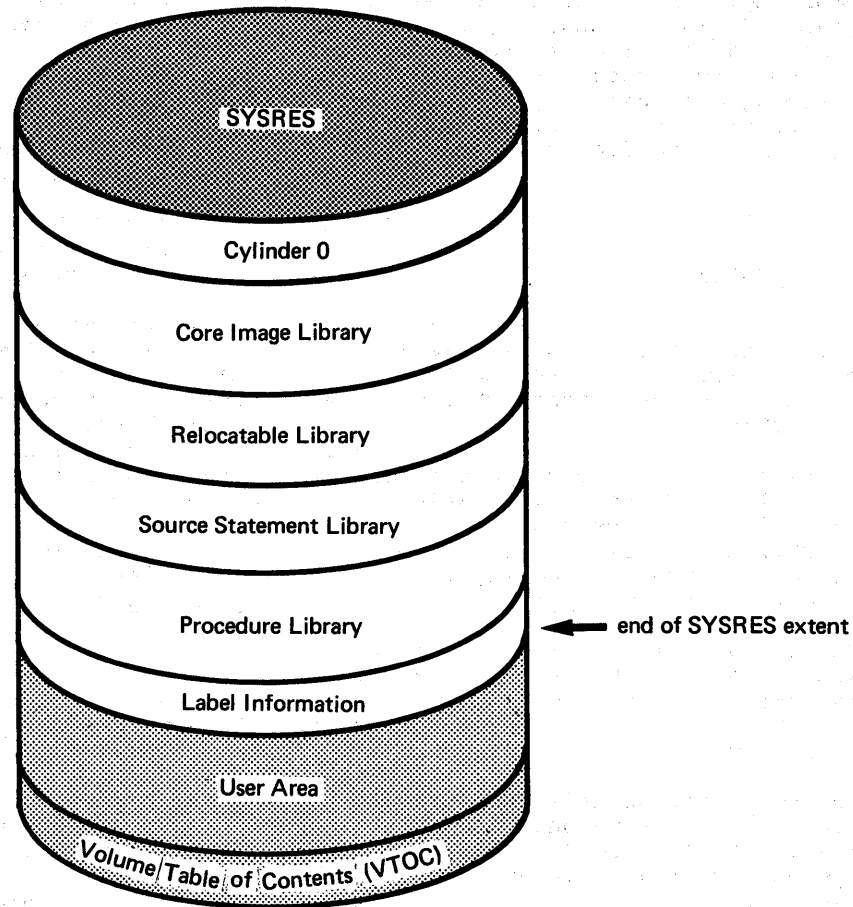


Figure 7. The Relative Location of the Four System Libraries

The directory area for each library is not shown in the Figure 7. By definition, all system libraries reside on the system residence file (SYSRES). If you have additional disk drives, you can define private core image, relocatable, and/or source statement libraries on the extra volumes. These volumes must be of the same type as the SYSRES pack. The system relocatable and system source statement libraries can be removed from SYSRES and established as private libraries; the system core image library, however, must always be present on SYSRES. It can be supplemented but not replaced by a private core image library. The procedure library is supported only as a system library; you cannot create a private procedure library.

SOURCE LANGUAGE CONSIDERATIONS

To use the private source statement library for COPY, BASIS, INSERT, and DELETE (see "Extended Source Program Library Facility" for further details), the ASSIGN, DLBL, and EXTENT control statements that define this private library must be present in the job deck for compilation (unless they are permanently set up by the installation). When present, a search for the book is made in the private library. If it is not there, the system library is searched. If the statements for the private library are not present, the system library is searched. A programmer may create several private libraries, but only one private library can be used in a given job.

EXTENDED SOURCE PROGRAM LIBRARY FACILITY

A complete program may be included as an entry in the source statement library by using the catalog function. This program can then be retrieved by a BASIS card and compiled in a subsequent job.

The following control statements would be used to catalog the program SAMPLE as a book in the COBOL sublibrary of the source statement library:

```
// JOB CATALOG
// EXEC MAINT
   CATALS C.SAMPLE
   BKEND C.SAMPLE

   {source program}

BKEND
/*
/£
```

When compiling a program that has been cataloged in the COBOL sublibrary of the source statement library, a BASIS card brings in an entire source program. The following control statements could be used to compile the cataloged program SAMPLE:

```
// JOB PGM1
// OPTION LOG,DECK,LIST,LISTX,ERRS
// EXEC FCOBOL
   CBL LIB
   BASIS SAMPLE
/*
/£
```

INSERT or DELETE cards may follow the BASIS card if the user wishes to modify the book SAMPLE before it is processed by the compiler. The original source program must have been coded with sequence numbers in columns 1 through 6 of each source card.

The INSERT statement will add new source statements after the specified sequence numbers. The DELETE statement will delete the statements indicated by the sequence numbers, or will delete more than one statement when the first and last sequence numbers to be deleted are specified, separated by a hyphen. Source program cards may follow a DELETE card for insertion before the card following the last one deleted. The sequence numbers in columns 1 through 6 are used to update COBOL source statements at compilation time, and are in effect for the one run only.

Assume that a company runs its payroll program each week as a source program taken

from the source statement library. The name of the program is PAYROLL. During a particular year, the old age insurance tax (FICA) is deducted at the rate of 4-2/5% each week for all personnel until earnings exceed \$7800. The coding to accomplish this is shown in Figure 8.

Now, however, due to a change in the old age tax laws, tax is to be taken out until earnings exceed \$10800 and a new percentage is to be placed. The programmer can code these changes as shown in Figure 9.

The altered program will contain the coding shown in Figure 10.

Reformatted Source Deck

By specifying the DECK option on the LST card, a new COBOL source deck can be produced that reflects the reformatted source listing. This deck may be saved in a BASIS library, used directly as input to the compiler, or punched onto cards. Because of reformatting, the new deck may contain more cards than the original, but the difference is not great enough to cause any appreciable increase in compilation time. The output deck differs from the listing as follows:

1. References, footnotes, and blank lines are omitted.
2. Literals will be repositioned, if needed, to assure proper continuation.
3. Statement numbers are converted to card numbers.
 - a. The statement number is multiplied by 10, and leading zeros are added as necessary to fill columns 1 through 6.
 - b. Comment and continuation cards are numbered one higher than the preceding card.
 - c. Statement-beginning cards are given the higher of the two numbers produced by the first two rules.

The use of this feature avoids having to resequence cards for permanent updating after they have been tested by temporary updating using the BASIS feature; it also avoids the errors incurred during that resequencing process.



```

000730          IF ANNUAL-PAY GREATER THAN 7800 GO TO PAY-WRITE.
000735          IF ANNUAL-PAY GREATER THAN 7800 - BASE-PAY GO TO LAST-FICA.
000740  FICA-PAYR.  COMPUTE FICA-PAY = BASE-PAY * .044
000745                MOVE TAX-PAY TO OUTPUT-TAX.
000750  PAY-WRITE.  MOVE BASE-PAY TO OUTPUT-BASE.
000755                ADD BASE-PAY TO ANNUAL-PAY.
.
.
.
000850                STOP RUN.

```

Figure 8. Sample Coding to Calculate FICA

```

// JOB PGM2
// OPTION LOG,DECK,LIST,LISTX,ERRS
// EXEC FCOBOL
CBL QUOTE, LIB
BASIS PAYROLL
DELETE 000730-000740
000730          IF ANNUAL-PAY GREATER THAN 10800 GO TO PAY-WRITE.
000735          IF ANNUAL-PAY GREATER THAN 10800 - BASE-PAY GO TO LAST-TAX.
000740  TAX-PAYR.  COMPUTE TAX-PAY = BASE-PAY * .0585
/*

```

Figure 9. Altering a Program from the Source Statement Library Using INSERT and DELETE Cards

```

000730          IF ANNUAL-PAY GREATER THAN 10800 GO TO PAY-WRITE.
000735          IF ANNUAL-PAY GREATER THAN 10800 - BASE-PAY GO TO LAST-TAX.
000740  TAX-PAYR.  COMPUTE TAX-PAY = BASE-PAY* .0585.
000750                MOVE TAX-PAY TO OUTPUT-TAX.
000760  PAY-WRITE.  MOVE BASE-PAY TO OUTPUT-BASE.
000770                ADD BASE-PAY TO ANNUAL-PAY.
.
.
.
000850                STOP RUN.

```

Figure 10. Effect of INSERT and DELETE Cards

The DOS/VS COBOL compiler, COBOL object module, Linkage Editor, and other system components can produce output in the form of printed listings, punched card decks, diagnostic or informative messages, and data files directed to tape or to mass storage devices. This chapter gives the format of and describes this output. The same COBOL program is used for each example. "Appendix A: Sample Program Output" shows the output formats in the context of a complete listing generated by the sample program.

COMPILER OUTPUT

The output of the compilation job step may include:

- A printed listing of the job control statements
- A printed listing of the statements contained in the source program
- A glossary of compiler-generated information about data
- Global tables, register assignments, and literal pools
- A printed listing of the object code
- A condensed listing containing only the relative address of the first generated instruction for each verb
- Compiler statistics
- Compiler diagnostic messages
- Cross-reference listings
- System messages
- An object module
- FIPS diagnostic messages

The presence or absence of the above-mentioned types of compiler output is determined by options specified at system generation time. These options can be overridden or additional options specified at compilation time by using the OPTION control statement and the CBL card.

The level of diagnostic message printed depends upon the FLAGW or FLAGE option of the CBL card.

All output to be listed is written on the device assigned to SYSLST. If SYSLST is assigned to a magnetic tape, COBOL will treat the file as an unlabelled tape. Line spacing of the source listing is controlled by the SPACEn option of the CBL card and by SKIP 1/2/3 and EJECT in the COBOL source program. (The lister feature ignores these commands.) The number of lines per page can be specified in the SET command. In addition, a listing of input/output assignments can be printed on SYSLST by using the LISTIO control statement.

On each page of the output, there is a header which contains the PROGRAM-ID, date and time of compilation, as well as an indication of the modification level of the compiler which produced this listing.

Figure 11 contains the compiler output listing shown in "Appendix A: Sample Program Output." Each type of output is numbered, and each format within each type is lettered. The text below and that following the figure is an explanation of the figure.

- ① The listing of the job control statements associated with this job step. These statements are listed because the LOG option was specified at system generation time.
- ② Compiler options. The CBL card, if specified, is printed on SYSLST unless the LIST option is suppressed.
- ③ The source module listing. The statements in the source program are listed exactly as submitted except that a compiler-generated card number is listed to the left of each line. This is the number referenced in diagnostic messages and in the object code listing. It is also the number printed on SYSLST as a result of the source language TRACE statement (if NOVERB is in effect). The source module is not listed when the NOLIST option is specified.



```
// JCE SAMPLE
// OPTION NODECK, LINK, LIST, LISTX, SYM, ERRS } ①
// EXEC FCOBOL
```

IBM DOS VS COBOL

REL 1.0

PP NO. 5746-CB1

07.43.04 03/03/74

```
CEL QUOTE, OPT, SXREF, LVL=A ②
C0001 000010 IDENTIFICATION DIVISION.
C0002 000020 PROGRAM-ID. TESTRUN.
C0003          AUTHOR. PROGRAMMER NAME.
C0004          INSTALLATION. NEW YORK DEVELOPMENT CENTER
C0005          DATE-WRITTEN. FEBRUARY 18, 1974
C0006          DATE-COMPILED. 03/03/74
C0007          REMARKS. THIS PROGRAM HAS BEEN WRITTEN AS A SAMPLE PROGRAM FOR
C0008          COBOL USERS. IT CREATES AN OUTPUT FILE AND READS IT BACK
C0009          AS INPUT.
C0010 000100
C0011 000110 ENVIRONMENT DIVISION.
C0012 000120 CONFIGURATION SECTION.
C0013 000130 SOURCE-COMPUTER. IBM-370.
C0014 000140 OBJECT-COMPUTER. IBM-370.
C0015 000150 INPUT-OUTPUT SECTION.
C0016 000160 FILE-CONTROL.
C0017 000170 SELECT FILE-1 ASSIGN TO SYS008-UT-2400-S.
C0018 000180 SELECT FILE-2 ASSIGN TO SYS008-UT-2400-S.
C0019 000190
.
.
.
C0056 000550 PROCEDURE DIVISION.
C0057          BEGIN.
C0058 000570     NOTE THAT THE FOLLOWING OPENS THE OUTPUT FILE TO BE CREATED
C0059 000580     AND INITIALIZES COUNTERS.
C0060 000590 STEP-1. OPEN OUTPUT FILE-1. MOVE ZERO TO COUNT, NUMBER.
.
.
.
C0073 000720 STEP-5. CLOSE FILE-1. OPEN INPUT FILE-2.
C0074 000730     NOTE THAT THE FOLLOWING READS BACK THE FILE AND SINGLES
C0075 000740     OUT EMPLOYEES WITH NO DEPENDENTS.
C0076 000750 STEP-6. READ FILE-2 RECORD INTO WORK-RECORD AT END GO TO STEP-8.
C0077 000760 STEP-7. IF NO-OF-DEPENDENTS IS EQUAL TO "0" MOVE "Z" TO
C0078 000770     NO-OF-DEPENDENTS. EXHIBIT NAME WORK-RECORD. GO TO STEP-6.
C0079 000780 STEP-8. CLOSE FILE-2.
C0080 000790 STOP RUN.
```

③

Figure 11. Examples of Compiler Output (Part 1 of 4)

(A)	(B)	(C)	(D)	(E)	(F)	(G)	(H)	(J)
INTRNL NAME	LVL	SOURCE NAME	BASE	DISPL	INTRNL NAME	DEFINITION	USAGE	R O Q M
DNM=1-148	FD	FILE-1	DTF=01		DNM=1-148		DTFMT	F
DNM=1-179	01	RECORD-1	BL=1	000	DNM=1-179	DS 0CL2C	GROUP	
DNM=1-200	02	FIELD-A	BL=1	000	DNM=1-200	DS 20C	DISP	
DNM=1-217	FD	FILE-2	DTF=02		DNM=1-217		DTFMT	F
DNM=1-248	01	RECORD-2	BL=2	000	DNM=1-248	DS 0CL20	GROUP	
DNM=1-269	02	FIELD-A	BL=2	000	DNM=1-269	DS 20C	DISP	
DNM=1-289	01	FILLER	BL=3	000	DNM=1-289	DS 0CL56	GROUP	
DNM=1-308	02	KOUNT	BL=3	000	DNM=1-308	DS 1H	COMP	
DNM=1-323	02	ALPHABET	BL=3	002	DNM=1-323	DS 26C	DISP	
DNM=1-341	02	ALPHA	BL=3	002	DNM=1-341	DS 1C	DISP	R O
DNM=1-359	02	NUMBR	BL=3	01C	DNM=1-359	DS 1H	COMP	
DNM=1-374	02	DEPENDENTS	BL=3	01E	DNM=1-374	DS 26C	DISP	
DNM=1-394	02	DEPEND	BL=3	01E	DNM=1-394	DS 1C	DISP	K O
DNM=1-410	01	WORK-RECORD	BL=3	038	DNM=1-410	DS 0CL20	GROUP	
DNM=1-434	02	NAME-FIELD	BL=3	038	DNM=1-434	DS 1C	DISP	
DNM=1-454	02	FILLER	BL=3	039	DNM=1-454	DS 1C	DISP	
DNM=1-473	02	RECORD-NO	BL=3	03A	DNM=1-473	DS 4C	DISP-NM	
DNM=1-492	02	FILLER	BL=3	03E	DNM=1-492	DS 1C	DISP	
DNM=2-000	02	LOCATION	BL=3	03F	DNM=2-000	DS 3C	DISP	
DNM=2-018	02	FILLER	BL=3	042	DNM=2-018	DS 1C	DISP	
DNM=2-037	02	NO-OF-DEPENDENTS	BL=3	043	DNM=2-037	DS 2C	DISP	
DNM=2-063	02	FILLER	BL=3	045	DNM=2-063	DS 7C	DISP	



MEMORY MAP

TGT	(A)	
		003F8
SAVE AREA		003F8
SWITCH		00440
TALLY		00444
SORT SAVE		00448
ENTRY-SAVE		0044C
SORT CORE SIZE		00450
NSTD-REELS		00454
SORT RET		00456
WORKING CELLS		00458
SORT FILE SIZE		00588
SORT MODE SIZE		0058C
PGT-VN TEL		00590
TGT-VN TEL		00594
SORTAB ADDRESS		00598
LENGTH OF VN TEL		0059C
LNQTH OF SORTAB		0059E
PGM ID		005A0
A(INIT1)		005A8
UPSI SWITCHES		005AC
DEBUG TABLE PTR		005B4
CURPENT PRIORITY		005B8
TA LENGTH		005B9
FRBL1 CELL PTR		005EC
UNUSED		005C0
RESERVED		005C4
VSAM SAVE AREA ADDRESS		005C8
UNUSED		005CC
RESERVED		005D4
OVERFLOW CELLS		005EC
BL CELLS		005FC
LTFADR CELLS		005F8
FIB CELLS		00600
TEMP STORAGE		00608
TEMP STORAGE-2		00610
TEMP STORAGE-3		00610
TEMP STORAGE-4		00610
BLL CELLS		00610
VLC CELLS		00614
SBL CELLS		00614
INDEX CELLS		00614
SUBADR CELLS		00614
ONCTL CELLS		0061C
PFMCTL CELLS		0061C
PFMSAV CELLS		0061C
VN CELLS		00620
SAVE AREA =2		00624
XSASW CELLS		00624



LITERAL POOL (HEX)

(B)						
00640 (LIT+0)	000000C1	001A5B5B	C2D6D7C5	D5405B5B	C2C3D3D6	E2C55B5B
00658 (LIT+24)	C2C6C3D4	E4D35B5B	C0000000			

DISPLAY LITERALS (BCD)

00664 (LITL+36) 'WORK-RECORD'

PGT	(C)	
		00628
DEBUG LINKAGE AREA		00628
OVERFLCW CELLS		00628
VIRTUAL CELLS		0062C
PROCEDURE NAME CELLS		00638
GENERATED NAME CELLS		00638
SUBDTF ADDRESS CELLS		0063C
VNI CELLS		0063C
LITERALS		00640
DISPLAY LITERALS		00664
PROCEDURE BLOCK CELLS		00670



Figure 11. Examples of Compiler Output (Part 2 of 4)

REGISTER ASSIGNMENT

REG 6 BL =3
 REG 7 BL =1
 REG 8 BL =2

7

WORKING-STORAGE STARTS AT LOCATION 00100 FOR A LENGTH OF 00050. 5

PROCEDURE BLOCK ASSIGNMENT

8

PEL = REG 11

PEL =1 STARTS AT LOCATION 000674 STATEMENT 60

A

B

C

E

F

```

57          000674          PN=02  EQU  *
60          000674          PN=03  EQU  *
60          000674          START  EQU  *
000674     58 B0 C 048      L      11,048(0,12)    PEL=1
000678     58 20 D 1F4      L      2,1F4(0,13)      BL =1
00067C     41 10 C 01E      LA     1,01E(0,12)      IIT+6
000680     58 00 D 200      L      0,200(0,13)    DTF=1
000684     18 40           LR      4,0
000686     05 F0           BALR   15,0
000688     50 00 F 008      ST     0,008(0,15)
00068C     45 00 F 00C      BAL   0,00C(0,15)
000690     00000000        DC     X'00000000'
000694     0A 02           SVC     2
000696     41 00 D 200      LA     0,200(0,13)    DTF=1
00069A     58 F0 C 008      L      15,008(0,12)  V(ILEE1M10)
00069E     05 EF           BALR   14,15
0006A0     58 10 D 200      L      1,200(0,13)    DTF=1
0006A4     96 10 1 020      OI     020(1),X'10'
0006A8     50 20 D 1F4      ST     2,1F4(0,13)    BI =1
0006AC     58 70 D 1F4      L      7,1F4(0,13)    BL =1

60          0006B0     D2 01 6 000 C 018    MVC  000(2,6),018(12)  DNM=1-308    IIT+0
0006B6     D2 01 6 01C C 018    MVC  01C(2,6),018(12)  DNM=1-359    IIT+0

64          0006BC          PN=04  EQU  *
64          0006BC     48 30 C 01A      LH     3,01A(0,12)    IIT+2
0006C0     4A 30 6 000      AH     3,000(0,6)    DNM=1-308
0006C4     4E 30 D 210      CVD   3,210(0,13)    TS=01
0006C8     D7 05 D 210 D 210      XC     210(6,13),210(13)  TS=01
0006CE     94 0F D 216      NI     216(13),X'0F'  TS=01+6
0006D2     4F 30 D 210      CVE   3,210(0,13)    TS=01
0006D6     40 30 6 000      STH   3,000(0,6)    DNM=1-308
0006DA     48 30 C 01A      LH     3,01A(0,12)    IIT+2
0006DE     4A 30 6 01C      AH     3,01C(0,6)    DNM=1-359
0006E2     4E 30 D 210      CVD   3,210(0,13)    TS=01
0006E6     D7 05 D 210 D 210      XC     210(6,13),210(13)  TS=01
0006EC     94 0F D 216      NI     216(13),X'0F'  TS=01+6
0006F0     4F 30 D 210      CVE   3,210(0,13)    TS=01
0006F4     40 30 6 01C      STH   3,01C(0,6)    DNM=1-359

64          0006F8     41 40 6 002      LA     4,002(0,6)    DNM=1-341
0006FC     48 20 6 000      LH     2,000(0,6)    DNM=1-308
000700     4C 20 C 01A      MH     2,01A(0,12)   IIT+2
000704     1A 42           AR     4,2
000706     5E 40 C 018      S      4,018(0,12)   IIT+0
00070A     50 40 D 21C      ST     4,21C(0,13)   SES=1
00070E     58 E0 D 21C      L      14,21C(0,13)  SES=1
66          000712     D2 00 6 038 E 000    MVC  038(1,6),000(14)  DNM=1-434
000718     41 40 6 01E      LA     4,01E(0,6)    DNM=1-394
00071C     48 20 6 000      LH     2,000(0,6)    DNM=1-308
000720     4C 20 C 01A      MH     2,01A(0,12)   IIT+2
000724     1A 42           AR     4,2
000726     5E 40 C 018      S      4,018(0,12)   IIT+0
00072A     50 40 D 220      ST     4,220(0,13)   SES=2
00072E     58 F0 D 220      L      15,220(0,13)  SES=2
000732     D2 00 6 043 F 000    MVC  043(1,6),000(15)  DNM=2-37    DNM=1-394
000738     92 40 6 044      MVI   044(6),X'40'   DNM=2-37+1
    
```

9

STATISTICS
 STATISTICS
 OPTIONS IN EFFECT
 OPTIONS IN EFFECT
 OPTIONS IN EFFECT
 OPTIONS IN EFFECT

SOURCE RECORDS = 80
 PARTITION SIZE = 655176
 PMAP RELOC ADR = NCNE
 LISTX QUOTE
 NOCLIST FLAGW
 NOSTATE TRUNC

DATA ITEMS = 22
 LINE COUNT = 56
 SPACING = 1
 SYM NOCATALR
 ZWB NOSUPMAP
 SEQ NOSYMDMF

NO OF VERBS = 28
 BUFFER SIZE = 512
 FLOW = NONE
 LIST LINK NCSTXIT
 XREF ERRS SXREF
 NCDECK NCVERE NCSYNTAX

NCLIB OPT
 LVL=A

10

Figure 11. Examples of Compiler Output (Part 3 of 4)

(A) CROSS-REFERENCE DICTIONARY

DATA NAMES	DEFN	REFERENCE
ALPHA	000042	000064
ALPHABET	000041	
DEPEND	000045	000066
DEPENDENTS	000044	
FIELD-A	000029	
FIELD-A	000037	
FILE-1	000017	000060 000068 000073
FILE-2	000018	000073 000076 000079
KCUNT	000040	000060 000064 000066 000070
LCATION	000051	
NAME-FIELD	000047	000064
NC-CF-DEPENDENTS	000053	000066 000077
NUMBR	000043	000060 000064 000067
RECORD-NO	000049	000067
RECORD-1	000028	000068
RECORD-2	000036	000076
WORK-RECORD	000046	000068 000076 000078

(B)

PROCEDURE NAMES	DEFN	REFERENCE
BEGIN	000057	
STEP-1	000060	
STEP-2	000064	000070
STEP-3	000068	000070
STEP-4	000070	
STEP-5	000073	
STEP-6	000076	000078
STEP-7	000077	
STEP-8	000079	000076

(A) CARD	(B) ERROR MESSAGE	(C)	(D)
00064	ILA5011I-W	HIGH ORDER TRUNCATION MIGHT OCCUR.	} (12)
00064	ILA5011I-W	HIGH ORDER TRUNCATION MIGHT OCCUR.	

FEDERAL INFORMATION PROCESSING STANDARDS (FIPS) DIAGNOSTIC MESSAGES

(A) LINE	(B) NUMBER	(C) MESSAGE	(D)
C0006	ILA8003I-W	DATE-COMPILED PARAGRAPH IS AN EXTENSION TO FIPS LEVEL A.	} (13)
C0025	ILA8002I-W	RECORDING MODE IS CLAUSE IS AN EXTENSION TO ALL FIPS LEVELS.	
C0034	ILA8002I-W	RECORDING MODE IS CLAUSE IS AN EXTENSION TO ALL FIPS LEVELS.	
C0054	ILA8003I-W	SPACES IS AN EXTENSION TO FIPS LEVEL A.	
C0060	ILA8003I-W	COMMA OR SEMICOLON AS PUNCTUATION IS AN EXTENSION TO FIPS LEVEL A.	
C0062	ILA8003I-W	COMMA OR SEMICOLON AS PUNCTUATION IS AN EXTENSION TO FIPS LEVEL A.	
C0062	ILA8003I-W	COMMA OR SEMICOLON AS PUNCTUATION IS AN EXTENSION TO FIPS LEVEL A.	
C0064	ILA8003I-W	COMMA OR SEMICOLON AS PUNCTUATION IS AN EXTENSION TO FIPS LEVEL A.	
C0064	ILA8003I-W	MULTIPLE RESULTS IN ADD STATEMENT IS AN EXTENSION TO FIPS LEVEL A.	
C0068	ILA8003I-W	UPON OPTION OF DISPLAY STATEMENT IS AN EXTENSION TO FIPS LEVEL A.	
C0068	ILA8002I-W	UPON CONSOLE OPTION OF DISPLAY STATEMENT IS AN EXTENSION TO ALL LEVELS.	
C0068	ILA8003I-W	FROM OPTION OF WRITE STATEMENT IS AN EXTENSION TO FIPS LEVEL A.	
C0070	ILA8003I-W	UNTIL OPTION OF PERFORM STATEMENT IS AN EXTENSION TO FIPS LEVEL A.	
C0076	ILA8003I-W	INTO OPTION OF READ STATEMENT IS AN EXTENSION TO FIPS LEVEL A.	
C0078	ILA8002I-W	EXHIBIT STATEMENT IS AN EXTENSION TO ALL FIPS LEVELS.	

END OF COMPILATION

OUTPUT

Figure 11. Examples of Compiler Output (Part 4 of 4)

The following notations may appear on the listing:

- C Denotes that the statement was inserted with a COPY statement.
- ** Denotes that the card is out of sequence. NOSEQ should be specified on the CBL card if the sequence check is to be suppressed.
- I Denotes that the card was inserted with an INSERT or BASIS card.

If DATE-COMPILED is specified in the Identification Division, any sentences in that paragraph are replaced in the listing by the date of compilation. It is printed in one of the following formats depending upon the format chosen at system generation time.

DATE-COMPILED. month/day/year or

DATE-COMPILED. day/month/year

- ④ Glossary. The glossary is listed when the SYM option is specified. The glossary contains information about names in the COBOL source program.

- ① and ② The internal-name generated by the compiler. This name is used in the compiler object code listing to represent the name used in the source program. It is repeated in column F for readability.

- ③ A normalized level number. This level number is determined by the compiler as follows: the first level number of any hierarchy is always 01, and increments for other levels are always by one. Only level numbers 03 through 49 are affected; level numbers 66, 77, and 88, and FD, SD, and RD indicators are not changed.

- ④ The data-name that is used in the source module.

Note: The following Report Writer internally-generated data-names can appear under the SOURCE NAME column:

- CTL.LVL Used to coordinate control break activities.
- GRP.IND Used by coding for GROUP INDICATE clause.

- TER.COD Used by coding for TERMINATE clause.
- FRS.GEN Used by coding for GENERATE clause.
- nnnn Generated report record associated with the file on which the report is to be printed.
- RPT.RCD Build area for print record.
- CTL.CHR First or second position of RPT.RCD. Used for carriage control character.
- RPT.LIN Beginning of actual information which will be displayed. Second or third position of RPT.RCD.
- CODE-CELL Used to hold code specified.
- E.nnnn Name generated from COLUMN clause in 02-level statement.
- S.nnnn Used for elementary level with SUM clause, but not with data-name.
- N.nnnn Used to save the total number of lines used by a report group when relative line numbering is specified.

- ⑤ and ⑥ For data-names, these columns contain information about the address in the form of a base and displacement. For file-names, the column contains information about the associated DTF or FIB (for VSAM). An indication is also given here if the FD is invalid.

- ⑦ This column defines storage for each data item. It is represented in assembler-like terminology. Table 4 refers to information in this column.

- ⑧ Usage of the data-name. For FD entries, either VSAM is specified, or the DTF type is identified (e.g., DTFDA). For group items containing a USAGE clause, the usage type is printed. For group items that do not contain a USAGE clause, GROUP is printed. For elementary items, the information in the USAGE clause is printed.

Table 4. Glossary Definition and Usage

Type	Definition	Usage
Group Fixed-Length	DS 0CLN	GROUP
Alphabetic	DS NC	DISP
Alphanumeric	DS NC	DISP
Alphanumeric Edited	DS NC	AN-EDIT
Numeric Edited	DS NC	NM-EDIT
Index-Name	DS 1H	INDEX-NM
Group Variable-Length	DS VLI=N	GROUP
Sterling Report	DS NC	RPT-ST
External Decimal	DS NC	DISP-NM
External Floating-Point	DS NC	DISP-FP
Internal Floating-Point	DS 1F	COMP-1
	DS 1D	COMP-2
Binary	DS 1H, 1F, OR 2F	COMP
Internal Decimal	DS NP	COMP-3
Sterling Non-Report	DS NC	DISP-ST
Index-Name	BLANK	INDEX-NAME
File (FD)	BLANK	DTF TYPE
Condition (88)	BLANK	BLANK
Report Definition (RD)	BLANK	BLANK
Sort Definition (SD)	BLANK	BLANK

Note: Under the definition column, N = size in bytes, except in group variable-length where it is a variable cell number.

- ⓐ A letter under column:
- R - Indicates that the data-name redefines another data-name.
 - O - Indicates that an OCCURS clause has been specified for that data-name.
 - Q - Indicates that the data-name is or contains the DEPENDING ON object of the OCCURS clause.
 - M - Indicates the record format. This field is not applicable to VSAM. The letters which may appear under column M are:
 - F - fixed-length records
 - U - undefined records
 - V - variable-length records
 - S - spanned records

addressable information needed by the object program for execution. For example, in the Procedure Division output coding (3), the address of the first instruction under STEP-1 (OPEN OUTPUT FILE-1) is found in the PROCEDURE NAME CELLS portion of the Program Global Table (PGT).

ⓐ The Task Global Table (TGT). This table is used to record and save information needed during the execution of the object program. This information includes switches, addresses, and work areas.

ⓑ The Literal Pool. This lists all literals used in the program, with duplications removed. These literals include those specified by the programmer (e.g., MOVE "ABC" TO DATA-NAME) and those generated by the compiler (e.g., to align decimal points in arithmetic computations). The literals are divided into two groups: those that are referenced by instructions (marked "LITERAL POOL") and those that are parameters to the display object time subroutine (marked "DISPLAY LITERALS").

- ⑤ The location and length of WORKING-STORAGE are noted here.
- ⑥ Global tables and literal pool: Global tables and the literal pool are listed when the CLIST, SYM, or LISTX option is specified, unless SUPMAP is specified and an E-level error is encountered, or CSYNTAX is specified and an E-level error is encountered. A global table contains easily



⑦ The Program Global Table (PGT). This table contains literals and the addresses of procedure-names, generated procedure-names, and procedure block locators referenced by Procedure Division instructions.

⑧ Compiler-generated information about the operands of the generated instruction. This includes names and relative locations of literals. Table 5 refers to information in this column.

⑦ Register assignment: This lists the permanent register assigned to each base locator in the object program. The remaining base locators are given temporary register assignments but are not listed. Register assignments are listed when CLIST, SYM, or LISTX is specified, and output is not overridden by the same conditions as above.

Table 5. Symbols Used in the Listing and Glossary to Define Compiler-Generated Information

Symbol	Meaning
DNM	SOURCE DATA NAME
SAV	SAVE AREA CELL
SWT	SWITCH CELL
TLY	TALLY CELL
WC	WORKING CELL
TS	TEMPORARY STORAGE CELL
VLC	VARIABLE LENGTH CELL
SBL	SECONDARY BASE LOCATOR
BL	BASE LOCATOR
BLL	BASE LOCATOR FOR LINKAGE SECTION
ON	ON COUNTER
PFM	PERFORM COUNTER
PSV	PERFORM SAVE
VN	VARIABLE PROCEDURE NAME
SBS	SUBSCRIPT ADDRESS
XSW	EXHIBIT SWITCH
XSA	EXHIBIT SAVE AREA
PRM	PARAMETER
PN	SOURCE PROCEDURE NAME
PBL	Procedure Block Locator
GN	GENERATED PROCEDURE NAME
DTF	DTF ADDRESS
FIB	File Information Block (for VSAM)
VNI	VARIABLE NAME INITIALIZATION
LIT	LITERAL
TS2	TEMPORARY STORAGE (NON-ARITHMETIC)
RSV	REPORT SAVE AREA
SDF	Secondary DTF Pointer
TS3	TEMPORARY STORAGE (SYNCHRONIZATION)
TS4	TEMPORARY STORAGE (SYNCHRONIZATION)
INX	INDEX CELL
V(BCDNAME)	VIRTUAL
VIR	VIRTUAL
OVF	Overflow Cell

⑧ Procedure block assignments: Procedure block assignments are printed when OPT is specified. The procedure block assignments give the location within the object program for each block of code addressed by register 11.

⑨ Object code listing. The object code listing is produced when the LISTX option is specified, unless SUPMAP is also specified and an E-level error is encountered, or unless CSYNTAX is specified and an E-level error is encountered. The actual object code listing contains:

① The compiler-generated card number. This number identifies the COBOL statement in the source deck which contains the verb that generates the object code found in column C. When VERB is specified, the actual verb or paragraph-name is listed with the generated card number.

② The relative location, in hexadecimal notation, of the object code instruction in the module.

③ The actual object code instruction in hexadecimal notation.

④ The procedure-name number. A number is assigned only to procedure-names referred to in other Procedure Division statements.

⑤ The object code instruction in the form that closely resembles assembler language. (Displacements are in hexadecimal notation.)

⑩ Statistics: The compiler statistics list the options in effect for this run, the number of Data Division statements specified, and the Procedure Division size. Each level number is counted as one statement in the Data Division. The Procedure Division size is approximately the number of verbs in the Procedure Division.

An indicator is also given here if dictionary spill occurred during compilation. If spill occurred, the amount of storage assigned to the compiler may be increased for better performance. Statistics are not listed if SYNTAX (or CSYNTAX and an E-level or higher error occurred) was in effect.

⑪ Cross-reference dictionary: The cross-reference dictionary is produced when the XREF or SXREF option is specified. It is suppressed if CSYNTAX is in effect and an E-level error is encountered. It consists of two parts:

(A) The cross-reference dictionary for data-names consists of data-names followed by the generated card number of the statement which defines each data-name, and the generated card number of statements where each data-name is referenced. Report Writer data-names, with the exception of data-names in the form "-nnn", are defined with the generated card number of their respective RD's.

(B) The cross-reference dictionary for procedure-names consists of the procedure-names followed by the generated card number of the statement where each procedure-name is used as a section-name or paragraph-name, and the generated card number of statements where each procedure-name is referenced.

A reference will appear to a procedure name if there is a reference to a logically equivalent procedure-name; a reference will also appear to a procedure name, if, in a segmented program, an implied branch to a segment entry is made.

If XREF is specified, the names are presented in the order in which they appear in the source program. If SXREF is specified, the names are presented alphabetically. The number of references appearing in the cross-reference dictionary for a given name is based upon the number of times the name is referenced in the code generated by the compiler.

⑫ Verb Cross-Reference: A verb cross-reference is produced when VERBSUM or VERBREF is specified. It is suppressed when CSYNTAX is in effect and an E-level error is encountered. The cross-reference consists of a listing of all Procedure

Division verbs used in the source program followed by the number of times the verb is actually used in the source program. In addition, if VERBREF is specified, the generated card numbers of each verb are printed. For VERBSUM and VERBREF, the COBOL verb OTHERWISE is treated as if the source program used the verb ELSE.

⑬ Diagnostic messages: The diagnostic messages associated with the compilation are always listed. The format of the diagnostic message is:

(A) Compiler-generated card number. This is the number of a line in the source program related to the error.

(B) Message identification. The message identification for the DOS/VS COBOL compiler always begins with the symbols ILA.

(C) The severity level. There are four severity levels as follows:

(W) Warning

This level indicates that an error was made in the source program. However, it is not serious enough to interfere with the execution of the program. These warning messages are listed only if the FLAGW option is specified in the CBL card or chosen at system generation time.

(C) Conditional

This level indicates that an error was made but the compiler usually makes a corrective assumption. The statement containing the error is retained. Execution can be attempted.

(E) Error

This level indicates that a serious error was made. Usually the compiler makes no corrective assumption. The statement or option containing the error is dropped. Compilation is completed, but execution of the program should not be attempted.

(D) Disaster

This error indicates that a serious error was made. Compilation is not completed. Results are unpredictable. If this is a compiler error, the job will terminate via the

OUTPUT

CANCEL macro and produce a dump.

- Ⓓ The message text. The text identifies the condition that caused the error and indicates the action taken by the compiler.

Since Report Writer generates a number of internal data items and procedural statements, some error messages may reflect internal names. In cases where the error occurs mainly in these generated routines, the error messages may indicate the card number of the RD entry for the report under consideration. In addition, there are errors that may indicate the number of the card upon which the statement containing the error ends rather than the card upon which the error occurs. Internal name formats for Report Writer are discussed under "Glossary" (heading 4, item C). Statement numbers are generated when a verb or procedure name is encountered.

The COBOL compile-time message that follows serves as an example of the format of COBOL compiler messages:

CARD ERROR MESSAGE

```
00105 ILA1002I-W ***** SECTION HEADER
                          MISSING.
                          ASSUMED PRESENT.
```

- The code "00105" at the left is the card number of the statement in which the error has occurred. (Some errors may not be discovered until information from various sections of the program is combined. For this reason, the source card number in the error message may not be exact.)
- ILA identifies this as a DOS/VS COBOL compiler message.
- The numeral "1002" represents the identifying number of the message; the first digit of this identifier indicates the phase in which the error was detected. In this case the message was generated by phase 1.
- The symbol "I" means that this is a message to the programmer for his action.
- "W" (warning) is a level of severity in the error codes described in item C.
- The five asterisks (*****) indicate words in a message that vary according to the program being compiled.

The message text is usually composed of two sentences. The first describes the error; the second describes what the compiler has done as a result of the error.

Note: By specifying a PROGRAM-ID of ERRMSG in any source program, the user can generate a complete listing of compiler diagnostics and problem determination aids. (See Figure 12.) In this case, a normal compilation never takes place. Only a list of all error messages and problem determination information is produced. The link option is reset if it was in effect.

Some messages are not given if CSYNTAX or SYNTAX is in effect. See "Program Checkout" for the list of these messages.

- Ⓙ FIPS Diagnostic Messages: The diagnostic messages associated with FIPS are listed separately from the compiler diagnostic messages, with a header identifying them as FIPS diagnostics. The format of the FIPS diagnostic messages is:

- Ⓐ Compiler-generated line number. This is the number of a line in the source program containing a nonstandard element.
- Ⓑ Message identification. The message identification for FIPS diagnostic messages always begins with the symbols ILA. The identifying numbers of the messages will always be 8001, 8002, 8003, or 8004, where:
 - 1 indicates an extension to a certain level of the FIPS
 - 2 indicates an extension to all levels of the FIPS
 - 3 indicates an extension to one or all levels of the FIPS, or an unusual condition;
 - 4 indicates that there are no FIPS diagnostic messages.
- Ⓒ The severity level. All FIPS diagnostic messages have a severity level of W (warning). This level indicates that something in the source program does not conform to the FIPS, but the compilation of the program will not be interrupted.
- Ⓓ The message text. The text identifies the condition or element that does not conform to the FIPS. The FIPS level is also designated.

```

|// JOB          ERRORMSG User information
|//            EXEC      FCOBOL
|
| IDENTIFICATION DIVISION.
| PROGRAM-ID.    ERRMSG.
| REMARKS.      COMPILATION OF THIS PROGRAM WILL RESULT IN ALL COMPILER
|               DIAGNOSTICS BEING PRODUCED. NO OBJECT MODULE AND NO COMPILE-
|               TIME STATISTICS ARE PRODUCED.
|
| ENVIRONMENT DIVISION.
| DATA DIVISION.
| PROCEDURE DIVISION.
| *             THE SAME RESULTS CAN BE ACHIEVED BY CHANGING THE PROGRAM-ID OF
| *             ANY PROGRAM TO 'ERRMSG'.
|               STOP RUN.

```

Figure 12. A Program that Produces COBOL Compiler Diagnostics

OBJECT MODULE

The object module contains the external symbol dictionary, the text of the program, and the relocation dictionary. It is followed by an END statement that marks the end of the module. For additional information about the external symbol dictionary and the relocation dictionary, see the publication DOS/VS System Control Statements.

An object deck is punched if the DECK option is specified, unless an E-level diagnostic message is generated. The object module is written on SYSLNK if the LINK option is specified, unless an E-level diagnostic message is generated. No deck is punched if CSYNTAX is in effect and E-level errors are encountered, or if SYNTAX is in effect.

LINKAGE EDITOR OUTPUT

The output of the link edit step may include:

- A printed listing of the job control statements

- A map of the phase after it has been processed by the Linkage Editor
- Diagnostic messages
- A listing of the linkage editor control statements
- A phase which may be assigned to the core image library

Any diagnostic messages associated with the Linkage Editor are automatically generated as output. The other forms of output may be requested by the OPTION control statement. All output to be listed is printed on the device assigned to SYSLST.

Figure 13 is an example of a linkage editor output listing. It shows the job control statements and the phase map. The different types of output are numbered and each type to be explained is lettered. The text following the figure is an explanation of the figure.

OUTPUT

// EXEC LNKEDT ①

JOB SAMPLE DOS LINKAGE EDITOR DIAGNOSTIC OF INPUT ②

ACTION TAKEN MAP REL
LIST AUTOLINK IJFFZZN
LIST AUTOLINK ILBDDSP0
LIST AUTOLINK IJJCPDV
LIST AUTOLINK ILBDDSS0
LIST INCLUDE IJJCPDV
LIST AUTOLINK ILBDIML0
LIST AUTOLINK ILBDMNS0
LIST AUTOLINK ILBDSAE0
LIST ENTRY

①	②	③	④	⑤	⑥	⑦	⑧	⑨	⑩	
PHASE	XFR-AD	LOCORE	HICORE	DSK-AD	ESD TYPE	LABEL	LOADED	REL-FR		
PHASE***	07D878	07D878	07F1FF	05F OF 4	CSECT	TESTRUN	07D878	07D878	RELOCATAELL	
					CSECT	IJFFZZN	07E1C8	07E1C8		
					* ENTRY	IJFFZZN	07E1C8			
					* ENTRY	IJFFZZZ	07E1C8			
					* ENTRY	IJFFZZZ	07E1C8			
					CSECT	ILBDSAE0	07F078	07F078		
					ENTRY	ILBDSAE1	07F0C0			
					CSECT	ILBDMNS0	07F070	07F070		
					CSECT	ILBDIML0	07F018	07F018		
					CSECT	ILBDDSP0	07E578	07E578		
					ENTRY	ILBDDSP1	07E978			
					CSECT	ILBDDSS0	07ECF0	07ECF0		
					ENTRY	ILBDDSS1	07EF50			
					ENTRY	ILBDDSS2	07EF48			
					ENTRY	ILBDDSS3	07F008			
					ENTRY	ILBDDSS4	07ED16			
					ENTRY	ILBDDSS5	07EDC2			
					ENTRY	ILBDDSS6	07EE22			
					ENTRY	ILBDDSS7	07EDEC			
					ENTRY	ILBDDSS8	07ED46			
					CSECT	IJJCPDV	07EAA8	07EAA8		
					ENTRY	IJJCPDV1	07EAA8			
					* ENTRY	IJJCPDV2	07EAA8			

* UNREFERENCED SYMBOLS

①

002 UNRESOLVED ADDRESS CONSTANTS

WXTRN STXITPSW
WXTRN ILBDBG2

③

Figure 13. Linkage Editor Output

- ① The job control statements. These statements are listed because the LOG option is specified.
- ② Disk linkage editor diagnostic message of input. The ACTION statement is not required. If the MAP option is specified, SYSLST must be assigned. If the statement is not used and SYSLST is assigned, MAP is assumed and a storage map and any error diagnostic messages are considered output on SYSLST.
- ③ Map of virtual storage. A phase map is printed when MAP is specified (or assumed) during linkage editor processing. The following information is contained in the storage map:
 - Ⓐ The name of each phase. This is the name specified in the phase statement.
 - Ⓑ The transfer address of each phase.
 - Ⓒ The lowest virtual storage location of each phase.
 - Ⓓ The highest virtual storage location of each phase.
 - Ⓔ The hexadecimal disk address where the phase begins in the core image library.
 - Ⓕ The names of all CSECT's belonging to a phase.
 - Ⓖ All defined entry points within a CSECT. If an entry point is not referenced, it is flagged with an asterisk (*).
 - Ⓗ The address where each CSECT is loaded.
 - Ⓙ The relocation factor of each CSECT.
 - Ⓚ The number of unresolved weak external references. This indication need not concern the programmer. An unresolved weak external reference does not cause the Linkage Editor to use the automatic library call mechanism. Instead, the reference is left unresolved, and the load module is marked as executable. The number of unresolved address constants will not necessarily be the same as the number of unreferenced symbols listed in the Linkage Editor output.

Comments on the Phase Map

The severity of linkage editor diagnostic messages may affect the production of the phase map. Since various processing options affect the structure of the phase, the text of the phase map will sometimes provide additional information. For example, the phase may contain an overlay structure. In this case, a map will be listed for each segment in the overlay structure.

Linkage Editor Messages

The Linkage Editor may generate informative or diagnostic messages. A complete list of these messages is included in the publication DOS/VS System Control Statements.

DOS ANS COBOL Unresolved External References

When the Linkage Editor encounters a weak external reference (WXTRN), autolinking is suppressed and the V-type address constant is either resolved from those modules included into the load module or it remains unresolved. Unresolved WXTRNs will not cause the Linkage Editor to cancel the link step if ACTION CANCEL is in effect.

The DOS/VS COBOL object time subroutine library utilizes WXTRNs not only as address constants but also as switches to determine at object time whether certain options are in effect. It is a very convenient feature which can lead to tight and efficient code.

Unresolved WXTRNs are normally intentional but unresolved EXTRNs are normally unintentional and an error.

Any of the following unresolved WXTRNs may appear when link editing an object module produced by an ANS COBOL compiler:

STXITPSW	ILBDFLW2	ILBDMRG0
ILBDDBG2	ILBDSRT0	ILBDFLW3
ILBDADR1	ILBDRELO	ILBDTC00
ILBDDBG0	ILBDTEF0	ILBDTC01
SORTEP	ILBDDSS1	ILBDDBG7
ILBDSTN0	ILBDDSS3	ILBDDBG8
ILBDFLW0	ILBDVOC1	ILBDTC30

OUTPUT

COBOL EXECUTION OUTPUT

The output generated by program execution (in addition to data written on output files) may include:

- Data displayed on the console or on the printer
- Diagnostic messages to the programmer
- Messages to the operator
- System informative messages
- SYMDMP, STATE, FLOW, and/or COUNT output
- System diagnostic messages
- A system dump

Appendix I contains the full list of execution time diagnostic messages.

A dump and system diagnostic messages are generated automatically during program execution only if the program contains errors that cause abnormal termination.

SYMDMP output is generated upon request, or upon abnormal termination. STATE and FLOW output are generated upon abnormal termination. The output of these features

is discussed in the chapter entitled "Symbolic Debugging Features".

COUNT output is generated upon normal or abnormal termination of the program. Output from this feature is described in the chapter "Execution Statistics".

Figure 14 is an example of output from the execution job step. The following text is an explanation of the illustration.

- ① Job control statements. These statements are listed because the LOG option is specified.
- ② Program output on printer. The results of execution of the EXHIBIT NAMED statement appear on the program listing.
- ③ Console output. Data is printed on the console output unit as a result of the execution of DISPLAY UPON CONSOLE.

OPERATOR MESSAGES

The COBOL phase may issue operator messages. In the message, XX denotes a system-generated 2-character numeric field that is used to identify the program issuing the message.

```
// ASSGN   SYS008,X'483' } ①
// EXEC

WORK-RECORD = A 0001 NYC Z
WCRK-RECORD = B 0002 NYC 1
WORK-RECORD = C 0003 NYC 2
WORK-RECORD = D 0004 NYC 3
WORK-RECORD = E 0005 NYC 4
WORK-RECORD = F 0006 NYC Z
WORK-RECORD = G 0007 NYC 1
WORK-RECORD = H 0008 NYC 2
WORK-RECORD = I 0009 NYC 3
WORK-RECORD = J 0010 NYC 4
WORK-RECORD = K 0011 NYC Z
WORK-RECORD = L 0012 NYC 1
WORK-RECORD = M 0013 NYC 2
WCRK-RECORD = N 0014 NYC 3
WORK-RECORD = O 0015 NYC 4
WORK-RECORD = P 0016 NYC Z
WORK-RECORD = Q 0017 NYC 1
WORK-RECORD = R 0018 NYC 2
WORK-RECORD = S 0019 NYC 3
WORK-RECORD = T 0020 NYC 4
WORK-RECORD = U 0021 NYC Z
WCRK-RECORD = V 0022 NYC 1
WORK-RECORD = W 0023 NYC 2
WORK-RECORD = X 0024 NYC 3
WORK-RECORD = Y 0025 NYC 4
WCRK-RECORD = Z 0026 NYC Z } ②

BG
BG A 0001 NYC 0
BG B 0002 NYC 1
BG C 0003 NYC 2
BG D 0004 NYC 3
BG E 0005 NYC 4
BG F 0006 NYC 0
BG G 0007 NYC 1
BG H 0008 NYC 2
BG I 0009 NYC 3
BG J 0010 NYC 4
BG K 0011 NYC 0
BG L 0012 NYC 1
BG M 0013 NYC 2
BG N 0014 NYC 3
BG O 0015 NYC 4
BG P 0016 NYC 0
BG Q 0017 NYC 1
BG R 0018 NYC 2
BG S 0019 NYC 3
BG T 0020 NYC 4
BG U 0021 NYC 0
BG V 0022 NYC 1
BG W 0023 NYC 2
BG X 0024 NYC 3
BG Y 0025 NYC 4
BG Z 0026 NYC 0
BG EOJ SAMPLE } ③
00.56.19,DURATION 00.03.42
```

Figure 14. Output from Execution Job Step

STOP Statement

The following message is generated by the STOP statement with the literal option:

XX C110A STOP 'literal'

Explanation: This message is issued at the programmer's discretion to indicate possible alternative action to be taken by the operator.

Operator Response: Follows the instructions given both by the message and on the job request form supplied by the programmer. If the job is to be resumed, hit the end/enter key.

ACCEPT Statement

The following message is generated by an ACCEPT statement with the FROM CONSOLE option:

XX C111A "AWAITING REPLY"

Explanation: This message is issued by the object program when operator intervention is required.

Operator Response: Enter the reply and hit the end/enter key. (The contents of the text field should be supplied by the programmer on the job request form.) Alphabetic characters may be entered lower case.

SYSTEM OUTPUT

Informative and diagnostic messages may appear in the listing during the execution of the object program.

Each of these messages contains an identification code in the first column of the message to indicate the portion of the operating system that generated the message. Table 6 lists these codes, together with identification for each.

Table 6. System Message Identification Codes

Code	Identification
0	An on-line console message from the Supervisor
1	A message from the Job Control Processor
2	A message from the Linkage Editor
3	A message from the Librarian
4	A message from LIOCS
7	A message from the Sort program
C	A message from COBOL object-time subroutines

OUTPUT

This chapter describes the accepted linkage conventions for calling and called programs and discusses linkage methods when using an assembler language program. In addition, this chapter contains a description of the overlay facility which enables different called programs to occupy the same area in virtual storage at different times. It also contains a suggested assembler language program to be used in conjunction with the overlay feature.

A COBOL source program that passes control to another program is a calling program. The program that receives control from the calling program is referred to as a called program. Both programs must be compiled (or assembled) in separate job steps, but the resulting object modules must be link edited together in the same phase.

A called program can also be a calling program; that is, a called program can, in turn, call another program. In Figure 15 for instance, program A calls program B; program B calls program C. Therefore:

1. A is considered a calling program by B
2. B is considered a called program by A
3. B is considered a calling program by C
4. C is considered a called program by B

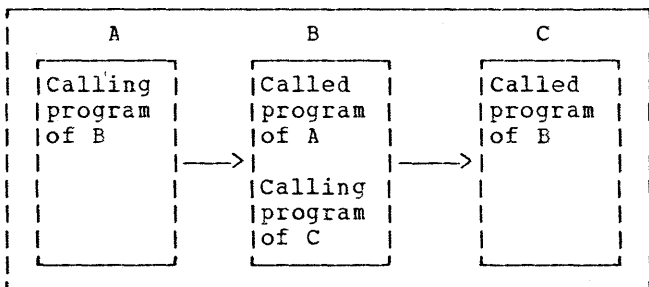


Figure 15. Calling and Called Programs

By convention, a called program may call to an entry point in any other program, except one on a higher level in the "path" of that program. That is, A may call to an entry point in B or C, and B may call C; however, C should not call A or B. Instead, C transfers control only to B by issuing the EXIT PROGRAM or GOBACK statements in COBOL (or its equivalent in another language). B then returns to A.

Compiler-generated switches, e.g., ON and ALTER, are not reinitialized upon each entrance to the called program, that is, the program is in its last executed state.

Note: It is necessary for an American National Standard COBOL program to know whether it is the main or the called program. For this reason, any non-American National Standard COBOL program calling an American National Standard program must first call the subroutine ILBDSET0. The function of this subroutine is to set a switch to X'FF' in subroutine ILBDMNS0, which is the indication to the COBOL program that it is a called program. Standard linkage conventions should be observed when calling ILBDSET0; there are no parameters to be passed.

LINKAGE

Whenever a program calls another program, linkage must be established between the two. The calling program must state the entry point of the called program and must specify any arguments to be passed. The called program must have an entry point and must be able to accept the arguments. Further, the called program must establish the linkage for the return of control to the calling program.

LINKAGE IN A CALLING PROGRAM

A calling COBOL program must contain the following statement at the point where another program is to be called:

```

CALL literal-1 [USING identifier-1
               [identifier-2]...]
  
```

literal-1

is the name specified as the program-name in the PROGRAM-ID paragraph of the called program, or the name of the entry point in the called program. When the called program is to be entered at the beginning of the Procedure Division, literal-1 is the name of the program being called. When the called program is to be entered at some point other than the beginning of the Procedure



Division, literal-1 should not be the same as the name specified in the PROGRAM-ID paragraph of the called program. Since the program-name in the PROGRAM-ID paragraph produces an external reference defining an entry point, this entry point name would not be uniquely defined as an external reference.

If the first character of PROGRAM-ID is numeric, the correspondence algorithm is as follows:

0 becomes J
1-9 become A-I

Since the system does not include the hyphen as an allowable character, the hyphen is converted to zero if it appears as the second through eighth character of the name.

identifier-1 [identifier-2]...

are the arguments being passed to the called program. Each identifier represents a data item defined in the File, Working-Storage, or Linkage Section of the calling program and should contain a level number 01 or 77. When passing identifiers from the File Section, the file should be open before the CALL statement is executed. If the called program is an assembler language program, the arguments may represent file-names and procedure-names in addition to data-names. If no arguments are to be passed, the USING option is omitted.

LINKAGE IN A CALLED PROGRAM

A called COBOL program must contain two sets of statements:

1. One of the following statements must appear at the point where the program is entered.

If the called program is entered at the first instruction in the Procedure Division and arguments are passed by the calling program:

```
PROCEDURE DIVISION [ USING  
  identifier-1 [identifier-2]...].
```

If the entry point of the called program is not the first statement of the Procedure Division:

```
ENTRY literal-1 [ USING identifier-1  
  [identifier-2]...]
```

literal-1

is the name of the entry point in the called program. It is the same name that appears in the CALL statement of the program that calls this program.

literal-1 must not be the name of any other entry point or program-name in the run unit.

identifier-1 [identifier-2]...

are the data items representing parameters. They correspond to the arguments of the CALL statement of the calling program. Each data item in this parameter list must be defined in the Linkage Section of the called program and must contain a level number of 01 or 77.

2. Either of the following statements must be inserted where control is to be returned to the calling program:

```
EXIT PROGRAM.
```

```
GOBACK.
```

Both the EXIT PROGRAM and GOBACK statements cause the restoration of the necessary registers, and return control to the point in the calling program immediately following the calling sequence.

ENTRY POINTS

Each time an entry point is specified in a called program, an external-name is defined. An external-name is a name that can be referenced by another program that has been separately compiled or assembled. Each time an entry name is specified in a calling program, an external reference is defined. An external reference is a symbol that is defined as an external-name in another separately compiled or assembled program. The Linkage Editor resolves external-names and external references, and combines calling and called programs into a format suitable for execution together, i.e., as a single phase.

Note: Several different entry points may be defined in one COBOL source module. Different CALL statements in any module of the phase may specify the same entry point, but each definition of an entry point must be unique in the same phase.

CORRESPONDENCE OF ARGUMENTS AND PARAMETERS

The number of identifiers in the argument list of the calling program should be the same as the number of identifiers in the parameter list of the called program. If the number of identifiers in the argument list of the calling program is greater than the number of identifiers in the parameter list of the called program, only those specified in the parameter list of the called program may be referred to by the called program. There is a one-for-one correspondence. The correspondence is positional and not by name. An identifier must not appear more than once in the same USING clause.

Only the address of an argument is passed. Consequently, both the identifier that is an argument and the identifier that is the corresponding parameter refer to the same location in storage. The pair of identifiers need not be identical, but the data descriptions must be equivalent. For example, if an argument is a level-77 data-name representing a 30-character string, its corresponding parameter could also be a level-77 data-name representing a character string of length 30, or the parameter could be a level-01 data item with subordinate items representing character strings whose combined length is 30.

Although all parameters in the ENTRY statement must be described with level numbers 01 or 77, there is no such restriction made for arguments in the CALL statement. An argument may be a qualified name or a subscripted name. When a group item with a level number other than 01 is specified as an argument, proper boundary word alignment is required if subordinate items are described as COMPUTATIONAL, COMPUTATIONAL-1, or COMPUTATIONAL-2. If the argument corresponds to an 01-level parameter, doubleword alignment is required.

LINK EDITING WITHOUT OVERLAY

Assume that a COBOL main program (COBMAIN), at one or more points in its logic executes CALL statements to COBOL programs SUEPRGA, SUBPRGB, SUBPRGC, and

SUBPRGD. Also assume that the module sizes for the main program and subprograms are:

<u>Program</u>	<u>Module Size</u> <u>(in bytes)</u>
COBMAIN	20,000
SUBPRGA	4,000
SUBPRGB	5,000
SUBPRGC	6,000
SUBPRGD	3,000

Through the linkage mechanism, all called programs plus COBMAIN must be link edited together to form one module of 38,000 bytes. Therefore, COBMAIN would require 38,000 bytes of storage in order to be executed. No overlay structure need be specified at link edit time if 38,000 bytes of virtual storage are available.

The following is an example of the job control statements needed to link edit these calling and called programs without specifying an overlay structure. The source decks for COBMAIN and SUBPRGA are included in the job deck, whereas SUBPRGB, SUBPRGC, and SUBPRGD are in the relocatable library.

```
// JOB NOVERLAY
// OPTION LINK,LIST,DUMP
ACTION MAP
PHASE EXAMP1,*
INCLUDE

{object module COBMAIN}
/*
INCLUDE SUBPRGB
INCLUDE SUBPRGC
INCLUDE SUBPRGD
INCLUDE

{object module SUBPRGA}
/*
ENTRY
// EXEC LNKEDT
// EXEC

{data for program}
/*
/8
```

CALL
PGMS

Figure 16 is an example of the data flow logic of this call structure where all the programs fit into virtual storage.

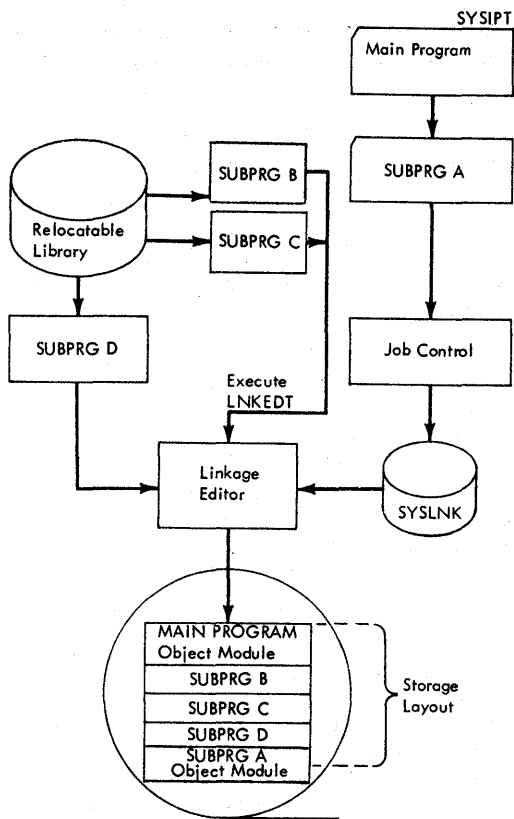


Figure 16. Example of Data Flow Logic in a Call Structure

Note: For the example given, it is assumed that SYSLNK is a standard assignment. The flow diagram illustrates how the various program segments are link edited into storage in a sequential arrangement.

ASSEMBLER LANGUAGE SUBPROGRAMS

A main program written in COBOL can call programs written in other languages that use the same linkage conventions. Whenever a COBOL program calls an assembler language program, certain conventions and techniques must be used.

There are three basic ways to use assembler-written called programs with a main program written in COBOL:

1. A COBOL main program or called program calling an assembler-written program.
2. An assembler-written program calling a COBOL program.
3. An assembler-written program calling another assembler-written program.

From these combinations, more complicated structures can be formed.

In a COBOL program, the expansions of the CALL and GOBACK or EXIT PROGRAM statements provide the save and return coding that is necessary to establish linkage between the calling and called programs in accordance with the linkage conventions of the system. Assembler language programs must be prepared in accordance with the same linkage conventions. These conventions include:

1. Using the proper registers to establish linkage.
2. Reserving, in the calling program, a storage area for items contained in the argument list. This storage area can be referenced by the called program.
3. Reserving, in the calling program, a save area in which the contents of the registers can be saved.

REGISTER USE

The Disk Operating System has assigned functions to certain registers used in linkages. Table 7 shows the conventions for using general registers as linkage registers. The calling program must load the address of the return point into register 14, and it must load the address of the entry point of the called program into register 15.

Table 7. Conventional Use of Linkage Registers

Reg. No.	Reg. Name	Function
1	Argument list register	Address of the argument list passed to the called program.
13	Save area register	Address of the area reserved by the calling program in which the contents of certain registers are stored by the called program.
14	Return register	Address of the location in the calling program to which control is returned after execution of the called program.
15	Entry point register	Address of the entry point in the called program.

SAVE AREA

A calling assembler language program must reserve a save area of 18 words, beginning on a fullword boundary, to be used by the called program for saving registers; it must load the address of this area into register 13. Table 8 shows the layout of the save area and the contents of each word.

A called COBOL program does not save floating-point registers. The programmer is responsible for saving and restoring the contents of these registers in the calling program.

Table 8. Save Area Layout and Word Contents

AREA (word 1)	This word is a part of the standard linkage convention established under the DOS/VS System. The word must be reserved for proper addressing of the subsequent entries. However, an assembler subprogram may use the word for any desired purpose.
AREA+4 (word 2)	The address of the previous save area, that is, the save area of the subprogram that called this one.
AREA+8 (word 3)	The address of the next save area, that is, the save area of the subprogram to which this subprogram refers.
AREA+12 (word 4)	The contents of register 14, that is, the return address.
AREA+16 (word 15)	The contents of register 15, that is, the entry address.
AREA+20 (word 6)	The contents of register 0.
AREA+24 (word 7)	The contents of register 1.
.	.
.	.
.	.
AREA+68 (word 18)	The contents of register 12.

ARGUMENT LIST

The argument list is a group of contiguous fullwords, beginning on a fullword boundary, each of which is an address of a data item to be passed to the called program. If the program is to pass arguments, an argument list must be prepared and its address loaded into register 1. The high-order bit of the last argument, by convention, is set to 1 to indicate the end of the list.

Any assembler-written program must be coded with a detailed knowledge of the data formats of the arguments being passed. Most coding errors occur because of the data format discrepancies of the arguments.

If one programmer writes both the calling program and the called program, the data format of the arguments should not present a problem when passed as parameters. However, when the programs are written by different programmers, the data format specifications for the arguments must be clearly defined for the programmer.

The linkage conventions used by an assembler program that calls another program are illustrated in Figure 16. The linkage should include:

1. The calling sequence.
2. The save and return routines.
3. The out-of-line parameter list. (An in-line parameter list may be used.)
4. A save area on a fullword boundary.

FILE-NAME AND PROCEDURE-NAME ARGUMENTS

A calling COBOL program that calls an assembler-language program can pass file-names and procedure-names, in addition to data-names, as identifiers. In the actual identifier-list that the compiler generates, the procedure-name is passed as the address of the procedure. For a file, the address of the DTF is passed, and the user must ensure that the file is already open. A VSAM file-name may not be passed.

Care must be taken when using these options. The user must be thoroughly familiar with the generated coding for each option and statement, as well as the structure of the object program.

```

deckname START 0 INITIATES PROGRAM ASSEMBLAGE AT FIRST
* AVAILABLE LOCATION. ENTRY POINT TO THE
* PROGRAM.
ENTRY name1
EXTRN name2
USING name1, 15
* SAVE ROUTINE
name1 STM 14,r1,12(13) THE CONTENTS OF REGISTERS 14, 15, AND
* 0 THROUGH r1 ARE STORED IN THE SAVE
* AREA OF THE CALLING PROGRAM (PREVIOUS
* SAVE AREA). r1 IS ANY NUMBER FROM 0 THROUGH 12.
LR r3,15
DROP 15
USING name1,r3 WHERE r3 AND r2 HAVE BEEN SAVED
LR r2,13 LOADS REGISTER 13, WHICH POINTS TO THE
* SAVE AREA OF THE CALLING PROGRAM, INTO
* ANY GENERAL REGISTER, r2, EXCEPT 0 AND 13.
LA 13,AREA LOADS THE ADDRESS OF THIS PROGRAM'S
* SAVE AREA INTO REGISTER 13.
ST 13,8(r2) STORES THE ADDRESS OF THIS PROGRAM'S SAVE
* AREA INTO WORD 3 OF THE SAVE AREA OF THE
* CALLING PROGRAM.
ST r2,4(13) STORES THE ADDRESS OF THE PREVIOUS SAVE
* AREA (I.E., THE SAME AREA OF THE CALLING
* PROGRAM) INTO WORD 2 OF THIS PROGRAM'S
* SAVE AREA.
BC 15,prob1
AREA DS 18F RESERVES 18 WORDS FOR THE SAVE AREA
* THIS IS LAST STATEMENT OF SAVE ROUTINE.
prob1 {User-written program statements}
L 15,VCON INDICATE COBOL PROGRAM IS
BALR 14,15 A SUBPROGRAM
* CALLING SEQUENCE
LA 1,ARGLST
L 15,ADCON
BALR 14,15
{Remainder of user-written program statements}
* RETURN ROUTINE
L 13,4(13) LOADS THE ADDRESS OF THE PREVIOUS SAVE
* AREA BACK INTO REGISTER 13.
LM 2,r1,28(13) THE CONTENTS OF REGISTER 2 THROUGH r1 ARE
* RESTORED FROM THE PREVIOUS SAVE AREA.
L 14,12(13) LOADS THE RETURN ADDRESS, WHICH IS IN
* WORD 4 OF THE CALLING PROGRAM'S SAVE AREA,
* INTO REGISTER 14.
MVI 12(13),X'FF' SETS FLAG FF IN THE SAVE AREA OF THE
* CALLING PROGRAM TO INDICATE THAT CONTROL
* HAS RETURNED TO THE CALLING PROGRAM.
BCR 15,14 LAST STATEMENT IN RETURN ROUTINE
VCON DC V(ILBDSET0)
ADCON DC A(name2) CONTAINS THE ADDRESS OF SUBPROGRAM name2.
* PARAMETER LIST
ARGLST DC AL4(arg1) FIRST STATEMENT IN PARAMETER AREA SETUP
DC AL4(arg2)
.
.
DC X'80' FIRST BYTE OF LAST ARGUMENT SETS BIT 0 TO 1
DC AL3(argn) LAST STATEMENT IN PARAMETER AREA SETUP

```

Figure 17. Sample Linkage Routines Used with a Calling Subprogram

ADCON	DC	A (prob ₁)
	.	
	LA	14, RETURN
	L	15, ADCON
	CNOP	2, 4
	BALR	1, 15
	DC	AL4 (arg ₁)
	DC	AL4 (arg ₂)
	.	
	DC	X'80'
	DC	AL3 (arg _n)
RETURN	EQU	*

Figure 18. Sample In-line Parameter List

In-Line Parameter List

The assembler programmer may establish an in-line parameter list instead of an out-of-line list. In this case, he may substitute the calling sequence and parameter list illustrated in Figure 18 for that shown in Figure 17.

LOWEST LEVEL PROGRAM

If an assembler called program does not call any other program (i.e., if it is at the lowest level), the programmer should omit the save routine, calling sequence, and parameter list shown in Figure 17. If the assembler called program uses any registers, it must save them. Figure 19 illustrates the appropriate linkage conventions used by an assembler program at the lowest level.

deckname	START	0
	ENTRY	name
	USING	*, 15
name	STM	14, r ₁ , 12 (13)
	.	
User-written program statements		
	.	
	LM	2, r ₁ , 28 (13)
	MVI	12 (13), X'PF'
	BCR	15, 14

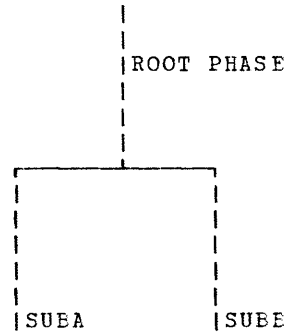
Note: If registers 13 and/or 14 are used in the called subprogram, their contents should be saved and restored by the called subprogram.

Figure 19. Sample Linkage Routines Used with a Lowest Level Subprogram

OVERLAYS

If a program is too large to be contained in the number of bytes available in virtual storage, it can still be executed by means of an overlay structure. An overlay structure permits the re-use of storage locations previously occupied by another program. In order to use an overlay structure, the programmer must plan his program so that one or more called programs need not be in storage at the same time as the rest of the program phase. The programmer should reassess, when going to VS, whether programs which used to require an overlay structure still do.

The following is a diagram of the basic form of a program to be overlaid:



The root phase consists of the COBOL main program and an assembler language subroutine which handles the overlay structures. SUBEA and SUBE are the called programs that are to be overlaid in storage.

In using the overlay technique, the programmer specifies to the Linkage Editor which programs are to overlay each other. These programs are processed by the Linkage Editor so they can be placed automatically in storage for execution when called by the main program. The resulting output of the Linkage Editor is called an overlay structure.

SPECIAL CONSIDERATIONS WHEN USING OVERLAY STRUCTURES

There are three areas of special concern to the programmer who decides to use the overlay feature. These problems concern the use of the assembler language subroutine, proper link editing, and job control statements.



ASSEMBLER LANGUAGE SUBROUTINE FOR
ACCOMPLISHING OVERLAY

The CALL statement is used for "direct" linkage; that is, the assistance of the Supervisor is not required (as it is when loading or fetching a phase). There are no COBOL statements that will generate the equivalent of the LOAD or FETCH assembler macro instructions. For this reason, one must call an assembler program to effect an overlay of a COBOL program. This routine must be link edited as part of either a root phase or permanently resident phase.

The sample overlay subroutine shown in Figure 20 is governed by the following restrictions:

1. The example is a suggested technique, and is not the only technique.

2. It can be used for assembler overlays if the programmer has a desired entry point in his END card and the first statement at that entry point is 'STM 14,12,12(13)'. .
3. This subroutine can be used for a COBOL program which contains an ENTRY statement immediately following the Procedure Division header. It will not work with a COBOL subprogram compiled with a Procedure Division USING statement or for entry points in a COBOL subprogram which appear anywhere other than as the first instruction of the Procedure Division. A suggested technique for diverse entry points is a table look-up using V-type constants.

STMNT	SOURCE STATEMENT	
0001	OVERLAY	START 0
0002		ENTRY OVRLAY
0003	* AT ENTRY TIME	
0004	* R1=POINTER TO ADCON LIST OF USING ARGUMENTS	
0005	* FIRST ARGUMENT IS PHASE OR SUBROUTINE NAME	
0006	* MUST BE 8 BYTES	
0007	* R13=ADDRESS OF SAVE AREA	
0008	* R14=RETURN POINT OF CALLING PROGRAM	
0009	* R15=ENTRY POINT OF OVERLAY PROGRAM	
0010	* AT EXIT	
0011	* R1=POINTER TO SECOND ARGUMENT OF ADCON LIST	
0012	* OF USING ARGUMENTS	
0013	* R14=RETURN POINT OF CALLING PROGRAM--NOT THIS PROG	
0014	* R15=ENTRY POINT OF PHASE OR SUBPROGRAM	
0015	*	
0016		USING *,15
0017	OVRLAY	STM 0,1,SAVE SAVE WORK REGS
0018		L 1,0(1) POINT R1 TO PHASE NAME
0019		CLC CORSUB,0(1) IN CORE?
0020		BE SUBIN YES,BR
0021		MVC CORSUB(8),0(1) SET CURRENT PHASE
0022		SR 0,0
0023		SVC 4 LOAD PHASE
0024	SEARCH1	LA 1,4(1) STEP SEARCH POINT
0025		CLC 0(3,1),=C'COB' END OF INIT1?
0026		BNE SEARCH1 NO, LOOP
0027		S 1,=F'8' POINT TO "START" ADCON
0028		L 1,0(1) LOAD "START"
0029		LA 1,8(1) INCREMENT TO "ENTRY"
0030		ST 1,ASUB SAVE ENTRY ADDRESS
0031	SUBIN	LM 0,1,SAVE RELOAD WORK REGS
0032		LA 1,4(1) POINT TO PARAMETERS
0033		L 15,ASUB
0034		BR 15 BRANCH TO ENTRY POINT
0035	CORSUB	DS 0CL8
0036		DC 8X'FF'
0037	ASUB	DS F
0038	SAVE	DS 2F
0039		END

Figure 20. Example of an Assembler Language Subroutine for Accomplishing Overlay

Note: Care should be taken with the techniques used in statements 0019 and 0020. Only when the COBOL program is loaded are altered GO TO statements reinitialized. A better technique would be to load the called programs each time they are required.

LINK EDITING WITH OVERLAY

In a linkage editor job step, the programmer specifies the overlay points in a program by using PHASE statements. In the Working-Storage Section, a level-01 or level-77 constant must be created for each phase to be called at execution time. These constants have a PICTURE of X(8) and a VALUE clause containing the same name as that appearing on the PHASE card for that segment in the link edit run.

In addition, each argument to be passed to the called program must have an entry in the Linkage Section. Remember, also, that the ENTRY statement should not refer to the program-name. (Use of the program-name will result in incorrect execution.)

When more than one subprogram in the overlay structure requires the same COBOL subroutine, the // EXEC LNKEDT statement must be preceded by INCLUDE cards for each of these subroutines. The names of these subroutines can be determined by requesting LISTX at compile time.

When preparing the control cards for the Linkage Editor, the programmer should be certain to include the assembler language subroutine with the main (root) phase. Also, to achieve maximum overlay, the phase names for the called programs should be different from the names of the called programs specified in the PROGRAM-ID paragraphs.

Figure 21 is a flow diagram of the overlay logic. The PHASE cards indicate the beginning address of each phase. The phases OVERLAYC and OVERLAYD will have the same beginning address as OVERLAYB. The sequence of events is:

1. The main program calls the overlay routine.
2. The overlay routine fetches the particular COBOL subprogram and places it in the overlay area.
3. The overlay routine transfers control to the first instruction of the called program.
4. The called program returns to the COBOL calling program (not to the assembler language overlay routine).

If OVERLAYB were known to be in storage, the CALL statement would be:

```
CALL "OVERLAYB" USING PARAM-1, PARAM-2.
```

But when using the assembler language overlay routine (OVLAY), it becomes:

```
CALL "OVLAY" USING PROCESS-LABEL,  
      PARM-1, PARM-2.
```

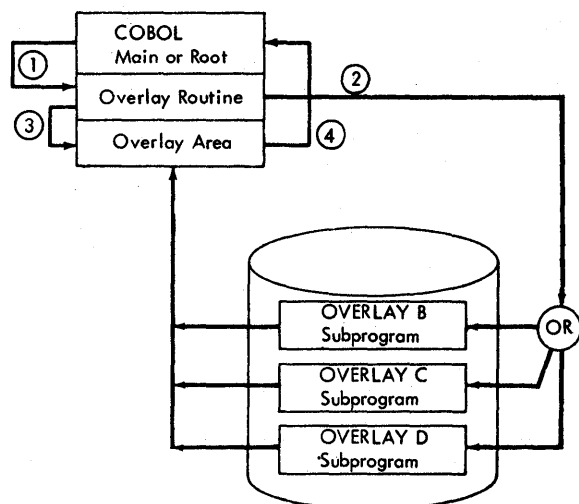
where PROCESS-LABEL contains the external-name OVERLAYB of the called program.

However, the ENTRY statement of the called program is the same for both cases, i.e., ENTRY "OVERLAYB" USING PARAM-1, PARAM-2, whether it is called indirectly by the main program through the overlay program or called directly by the main program.

Note: An ENTRY which is to be called by OVLAY must precede the first executable statement in the called program.

CALL
PGMS

JOB CONTROL FOR ACCOMPLISHING OVERLAY



The job control statements required to accomplish the overlay illustrated in Figure 21 are shown in Figure 22. The PHASE statements specify to the Linkage Editor that the overlay structure to be established is one in which the called programs OVERLAYB, OVERLAYC, and OVERLAYD overlay each other when called during execution.

Note: The phase name specified in the PHASE card must be the same as the value contained in the first argument for CALL "OVRLAY", i.e., PROCESS-LABEL, COMPUTE-TAX, etc., contain OVERLAYB, OVERLAYC, respectively, which are the names given in the PHASE card.

It is the programmer's responsibility to write the entire overlay, i.e., the COBOL main (or calling) program and an assembler language subroutine (for which a sample program is given in this chapter) that fetches and overlays the called programs. A calling sequence to obtain an overlay structure between three COBOL subprograms is illustrated in Figure 23.

Figure 21. Flow Diagram of Overlay Logic

```

// JOB OVERLAYS
// OPTION LINK
PHASE OVERLAY,ROOT
// EXEC FCOBOL
{COBOL Source for Main Program MAINLINE}
/*
// EXEC ASSEMBLY
[Source deck for Assembler Language Routine OVERLAY]
/*
PHASE OVERLAYB,*
// EXEC FCOBOL
{COBOL Source for Called Program OVERLAYB}
/*
PHASE OVERLAYC,OVERLAYB
// EXEC FCOBOL
{COBOL Source for Called Program OVERLAYC}
/*
PHASE OVERLAYD,OVERLAYC
// EXEC FCOBOL
{COBOL Source for Called Program OVERLAYD}
/*
// EXEC LNKEDT
// EXEC
/*
/£

```

Figure 22. Job Control for Accomplishing Overlay

```

COBOL Program Main (Root or Main Program)
IDENTIFICATION DIVISION.
PROGRAM-ID. MAINLINE.
.
.
.
ENVIRONMENT DIVISION.
.
.
.
DATA DIVISION.
.
.
.
WORKING-STORAGE SECTION.
77 PROCESS-LABEL PICTURE IS X(8) VALUE IS "OVERLAYB".
77 PARAM-1 PICTURE IS X.
77 PARAM-2 PICTURE IS XX.
77 COMPUTE-TAX PICTURE IS X(8) VALUE IS "OVERLAYC".

01 NAMET.
02 EMPLOY-NUMB PICTURE IS 9(5) .
02 SALARY PICTURE IS 9(4)V99.
02 RATE PICTURE IS 9(3)V99.
02 HOURS-REG PICTURE IS 9(3)V99.
02 HOURS-OT PICTURE IS 9(2)V99.
01 COMPUTE-SALARY PICTURE IS X(8) VALUE IS "OVERLAYD".
01 NAMES.
02 RATES PICTURE IS 9(6) .
02 HOURS PICTURE IS 9(3)V99.
02 SALARYX PICTURE IS 9(2)V99.
.
.
.
PROCEDURE DIVISION.
.
.
.
CALL "OVLAY" USING PROCESS-LABEL, PARAM-1, PARAM-2.
.
.
.
CALL "OVLAY" USING COMPUTE-TAX, NAMET.
.
.
.
CALL "OVLAY" USING COMPUTE-SALARY, NAMES.
.
.
.

```

Figure 23. Calling Sequence to Obtain Overlay Between Three COBOL Subprograms (Part 1 of 3)



COBOL Subprogram B

IDENTIFICATION DIVISION.
PROGRAM-ID. OVERLAY1.

.
.

ENVIRONMENT DIVISION.

.
.

DATA DIVISION.

.
.

LINKAGE SECTION.

01 PARAM-10 PICTURE X.
01 PARAM-20 PICTURE XX.

.
.

PROCEDURE DIVISION.

PARA-NAME. ENTRY "OVERLAYB" USING PARAM-10, PARAM-20.

.
.

GOBACK.

COBOL Subprogram C

IDENTIFICATION DIVISION.
PROGRAM-ID. OVERLAY2.

.
.

ENVIRONMENT DIVISION.

.
.

DATA DIVISION.

.
.

LINKAGE SECTION.

01 NAMEX.
02 EMPLY-NUMBX PICTURE IS 9(5).
02 SALARYX PICTURE IS 9(4) V99.
02 RATEX PICTURE IS 9(3) V99.
02 HOURS-REGX PICTURE IS 9(3) V99.
02 HOURS-OTX PICTURE IS 9(2) V99.

PROCEDURE DIVISION.

PARA-NAME. ENTRY "OVERLAYC" USING NAMEX.

.
.

GOBACK.

Figure 23. Calling Sequence to Obtain Overlay Between Three COBOL Subprograms
(Part 2 of 3)

```

COBOL Subprogram D
IDENTIFICATION DIVISION.
PROGRAM-ID. OVERLAY3.
.
.
.
ENVIRONMENT DIVISION.
.
.
.
DATA DIVISION.
.
.
LINKAGE SECTION
01 NAMES.
   02 RATES PICTURE IS 9(6) .
   02 HOURS PICTURE IS 9(3)V99.
   02 SALARYX PICTURE IS 9(2)V99.
.
PROCEDURE DIVISION.
PARA-NAME. ENTRY "OVERLAYD" USING NAMES.
.
.
.
      GOBACK.

```

Figure 23. Calling Sequence to Obtain Overlay Between Three COBOL Subprograms
(Part 3 of 3)



COBOL segmentation permits the user to subdivide logically and physically the Procedure Division of a COBOL object program. All source sections which contain the same segment-number in their section headers will be considered at object time to be one segment. Since segment-numbers can range from 00 through 99, it is possible to subdivide any object program into a maximum of 100 segments.

Program segments may be of three types: fixed permanent, fixed overlayable, and independent as determined by the programmer's assignment of segment numbers.

Segmentation of a program would be used when virtual storage is limited. In a real storage system, the following would apply:

1. Fixed segments are always in real storage during the execution of the entire program, that is, they cannot be overlaid except when the system itself is executing another program, in which case fixed segments may be "rolled out."
2. Fixed overlayable segments may be overlaid during program execution, but any such overlaying is transparent to the user, that is, they are logically identical to fixed segments, but physically different from them.
3. Independent segments may be overlaid, but such overlaying will result in the initialization of that segment. Therefore, independent segments are logically different from fixed permanent/fixed overlayable segments, and physically different from fixed segments.

In a virtual storage system, all logically "fixed" segments, that is, fixed permanent and fixed overlayable, are treated the same. They are both "paged in and out" as required for execution.

In the same manner, independent segments are paged in and out; when they are paged in, however, they are brought back in the initial state.

In DOS/VS COBOL, segments that are overlaid are not actually "paged out". All the variable data items associated with the segment are contained in one segment, which is considered the root segment. When a segment is "paged in", all the fields which must be reinitialized are contained

in the root segment. Thus no fields in other than the root segment are modified.

The program SAVECORE could be segmented as illustrated in Figure 24.

```

|IDENTIFICATION DIVISION.
|PROGRAM-ID. SAVECORE.
|.
|.
|ENVIRONMENT DIVISION.
|OBJECT-COMPUTER. IBM-370.
|    SEGMENT-LIMIT IS 15.
|.
|.
|DATA DIVISION.
|.
|.
|PROCEDURE DIVISION.
|SECTION-1 SECTION 8.
|.
|.
|SECTION-2 SECTION 8.
|.
|.
|SECTION-3 SECTION 16.
|.
|.
|SECTION-4 SECTION 8.
|.
|.
|SECTION-5 SECTION 50.
|.
|.
|SECTION-6 SECTION 16.
|.
|.
|SECTION-7 SECTION 50.
|.
|.

```

Figure 24. Segmenting the Program SAVECORE

Assuming that 12K of virtual storage is available for the program SAVECORE, Figure 25 shows the manner in which storage would be utilized. It is apparent from the illustration that SECTION-3, SECTION-6, and SECTION-7 cannot be in storage at the same time, nor can SECTION-3, SECTION-5 and SECTION-7 be in storage simultaneously.

Sections in the permanent segment (SECTION-1, SECTION-2, and SECTION-4) are those which must be available for reference at all times, or which are referenced frequently. They are distinguished here by the fact that they have been assigned



priority numbers less than the segment limit.

Sections in the overlayable fixed segment are sections which are less frequently used. They are always made available in the state they were in when last used. They are distinguishable here by the fact that they have been assigned priority numbers greater than the segment limit but less than 49.

Sections in the independent segment can overlay, and be overlaid by, either an overlayable fixed segment or another independent segment. Independent segments are those assigned priority numbers greater than 49 and less than 100, and they are always given control in their initial state.

OPERATION

Execution of the object program begins in the root segment. The first segment in the permanent segment is considered the root segment. If the program does not contain a permanent segment, the compiler generates a dummy segment which will initiate the execution of the first overlayable or independent segment. All global tables, literals, and data areas are part of the root segment. Called object time subroutines are also part of the root segment. When CALL statements appear in a segmented program, subprograms are loaded with the fixed portion of the main program as if they had a priority of zero.

Segmented programs must not be called by another program (segmented or not segmented). If a segmented program calls a subprogram, the CALL statement may appear in any segment. However, the object module associated with the subprogram must be included in the root segment prior to the execution of the main program. This can be accomplished in either of two ways as follows:

1. Produce object decks for both programs and place the one for the subprogram in the root segment:

```
PHASE,ROOT  
ESD card for the root segment
```

```
{object deck for the main program}
```

```
{object deck for the subprogram}
```

```
followed by a // EXEC LNKEDT and a //  
EXEC.
```

2. Catalog the object module for the subprogram in the relocatable library prior to link editing the main program. Insert an INCLUDE card for the subprogram and an ENTRY card for the root phase into the linkage editor control cards for the root phase of the main program. The ENTRY card will cause the linkage editor to pass control to the main program at execution time. The Linkage Editor will search the relocatable library for the subprogram and include it with the root phase.

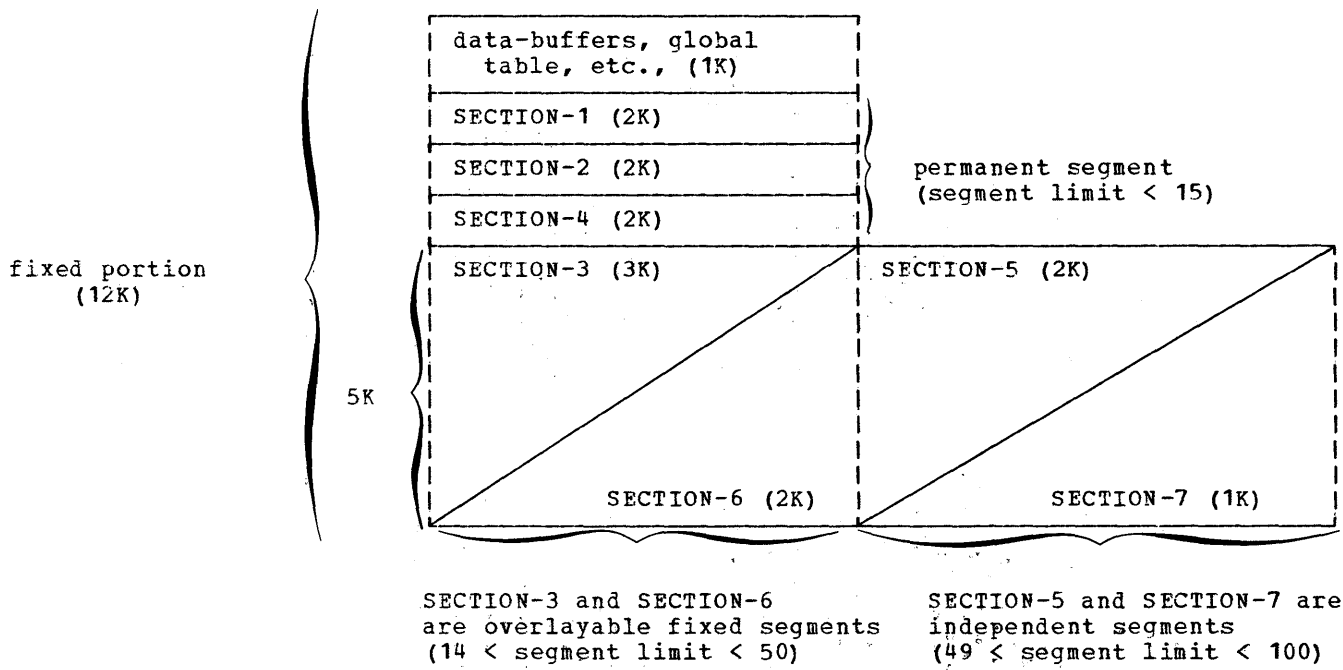


Figure 25. Storage Layout for SAVECORE

OUTPUT FROM A SEGMENTED PROGRAM

COMPILER OUTPUT

The output produced by the compiler is an overlay structure consisting of multiple object modules preceded by linkage editor control statements. Segments whose priority is greater than the segment limit (or 49, if no SEGMENT-LIMIT clause is specified) consist of executable instructions only.

The compiler generates each segment as a separate object module preceded by a PHASE card. The names appearing on these PHASE cards (segment-names) conform to the following naming conventions:

1. The name of the root segment is the same as the program-name specified in the PROGRAM-ID clause.
2. The name of each overlayable and independent segment is a combination of the program-name and the priority number of the segment. These names are formed according to the following rules:

- a. If the program-name is 6, 7, or 8 characters in length, the segment-name consists of the first 6 characters of program-name plus the 2-character priority number.
- b. If the program-name is less than 6 characters in length, the priority number is appended after the program-name.
- c. Since the system expects the first character of PROGRAM-ID to be alphabetic, the first character, if numeric, is converted as follows:

0 -> J
1-9 -> A-I

The hyphen is converted to zero if it appears as the second through eighth character.

- d. When DECK is specified, the punched object deck is sequenced according to segments. Columns 73-74 contain the first two characters of the program-id, columns 75-76 contain the priority number of the segment, and columns 77-80 contain the sequence number



of the card. The priority of the root segment is punched as 00.

- e. When the compiler option CATALR is in effect, the PHASE card for each segment is preceded by a CATALR card with the same name. This will enable direct cataloging of the compiler-produced object module into the relocatable library from which a load module may be link edited into the core-image library.

Note: Single-digit priority numbers are preceded by a zero.

Warning: In order to avoid duplicate names, the programmer must be aware of the above naming conventions. If the last two characters of an 8-character PROGRAM-ID are numeric, these same two characters may not appear in the source program as a segment number.

Figure 26 is an illustration of the compiler output for the skeleton program shown in Figure 24.

```

PHASE SAVECORE,ROOT
{object module for the root segment
 (sections with priority-numbers less
 than the segment limit) including any
 programs called by SAVECORE}

PHASE SAVECO16,*
{object module for segments with a
 priority of 16 (two sections)}

PHASE SAVECO50,SAVECO16
{object module for segments with a
 priority of 50 (two sections)}

```

Figure 26. Compiler Output for SAVECORE

LINKAGE EDITOR OUTPUT

Figure 27 is an illustration of the input to the Linkage Editor and the phase map produced by the Linkage Editor resulting from the compilation and editing of the segmented program BIGJOB. The following text is an explanation of the figure.

- ① PHASE card generated by the compiler for the root segment BIGJOB.

- ② AUTOLINK card for the Segmentation subroutine.
- ③ PHASE cards generated by the compiler for segments of priority 10, 47-50, 60, 62, and 63.
- ④ Control card generated for the Sort Feature. This card is explained in "Sort in a Segmented Program."
- ⑤ Location of the entry point CURSEGM. Item 5 is explained in "Determining the Priority of the Last Segment Loaded into the Transient Area."
- ⑥ Load address of phase BIGJOB00. Item 6 is explained in "Sort in a Segmented Program."

Note: If the CATALR option of the CBL card is specified, the compiler generates CATALR cards in front of PHASE cards.

Cataloging a Segmented Program

When the CATAL option is used to catalog a segmented program, the following points should be observed:

1. To avoid duplicate names, the programmer must be aware of the naming conventions used by the compiler (see "Compiler Output") because a segment-name may be the same as a phase-name already existing in the core image library.
2. Since the PHASE card is generated by the compiler, the programmer must not specify a PHASE card for the program.

To invoke a previously cataloged segmented program, the programmer must use the following control statement:

```
// EXEC name
```

where name is the program-name specified in the PROGRAM-ID clause.

Determining the Priority of the Last Segment Loaded into the Transient Area

If a segmented program is abnormally terminated during execution, and the SYMDMP option has been specified, the CURRENT PRIORITY cell in the Task Global Table contains the priority of the last segment loaded into the transient area. If SYMDMP has not been specified, the priority of this segment can be determined as follows:

1. In the map of virtual storage generated by the Linkage Editor, under the column LABEL, look for the name 'CURSEGM' (see item 5 in Figure 27).
2. Associated with this label, in the column LOADED, is an address.
3. At this location is stored the priority (one byte) of the segment current in the transient area. If this byte is X'00', no segment has been loaded into the transient area. This indicates that the error causing the dump occurred in the root segment.

SORT IN A SEGMENTED PROGRAM

If a segmented program contains a SORT statement, the sort program will be loaded above the largest overlayable or independent segment as shown in Figure 28.

The compiler accomplishes this by providing the following control statement at the end of the overlay structure:

```
PHASE BIGJOB00,transient area + L
```

This card is illustrated in Figure 27, item 4. The value of "L" in the figure is X'002F2' which is the length of the longest segment, BIGJOB47, rounded to the next halfword boundary. Note that Linkage Editor relocates the phase BIGJOB00 to the next doubleword boundary (see Figure 27, item 6).

Using the PERFORM Statement in a Segmented Program

When the PERFORM statement is used in a segmented program, the programmer should be aware of the following:

- A PERFORM statement that appears in a section whose priority-number is less than the segment limit can have within its range only (a) sections with priority-numbers less than 50, and (b) sections wholly contained in a single segment whose priority-number is greater than 49.

Note: As an extension to American National Standard COBOL, DOS/VS COBOL allows sections with any priority-number to fall within the range of a PERFORM statement.

- A PERFORM statement that appears in a section whose priority-number is equal to or greater than the segment limit can have within its range only (a) sections with the same priority-number as the section containing the PERFORM statement, and (b) sections with priority-numbers that are less than the segment limit.

Note: As an extension to American National Standard COBOL, DOS/VS COBOL allows sections with any priority-number to fall within the range of a PERFORM statement.

- When a procedure-name in a segment with a priority-number less than the segment limit is referred to by a PERFORM statement in a segment with a priority-number greater than the segment limit, the independent segment will be reinitialized upon exit from the PERFORM.

JOB BIGJ DISK LINKAGE EDITOR DIAGNOSTIC OF INPUT

ACTION TAKEN MAP

```

LIST PHASE BIGJOB,ROOT ← ①
.
.
LIST AUTOLINK ILBDSEMO ← ②
LIST AUTOLINK ILBDSRTO
.
LIST PHASE BIGJOB10,*
LIST PHASE BIGJOB47,BIGJOB10
LIST PHASE BIGJOB48,BIGJOB47
LIST PHASE BIGJOB49,BIGJOB48
LIST PHASE BIGJOB50,BIGJOB49 } ← ③
LIST PHASE BIGJOB60,BIGJOB50
LIST PHASE BIGJOB62,BIGJOB60
LIST PHASE BIGJOB63,BIGJOB62
LIST PHASE BIGJOB00,BIGJOB63+X'002F2' ← ④
    
```

PHASE	XFR-AD	LQCORE	HICORE	DSK-AD	ESD TYPE	LABEL	LOADED	REL-FR
ROOT	BIGJOB	003000	003000	0075A3	64 04 1	CSECT	BIGJOB	003000 003000
						CSECT	ILBDSEMO	006268 006268
						* ENTRY	CURSEGM	00637D ← ⑤
						CSECT	ILBDSRTO	006B38 006B38
BIGJOB10	0075A8	0075A8	0075E9	64 09 2	CSECT	BIGJOB10	0075A8	0075A8
BIGJOB47	0075A8	0075A8	007899	65 00 1	CSECT	BIGJOB47	0075A8	0075A8
BIGJOB48	0075A8	0075A8	0075DB	65 00 2	CSECT	BIGJOB48	0075A8	0075A8
BIGJOB49	0075A8	0075A8	0075D3	65 01 1	CSECT	BIGJOB49	0075A8	0075A8
BIGJOB50	0075A8	0075A8	0075F1	65 01 2	CSECT	BIGJOB50	0075A8	0075A8
BIGJOB60	0075A8	0075A8	0076ED	65 02 1	CSECT	BIGJOB60	0075A8	0075A8
BIGJOB62	0075A8	0075A8	0075D1	65 02 2	CSECT	BIGJOB62	0075A8	0075A8
BIGJOB63	0075A8	0075A8	007621	65 03 1	CSECT	BIGJOB63	0075A8	0075A8
BIGJOB00	0078A0	0078A0	0078A1	65 03 2	CSECT	ILBDDUM0	0078A0	0078A0 ← ⑥

Figure 27. Link Editing a Segmented Program

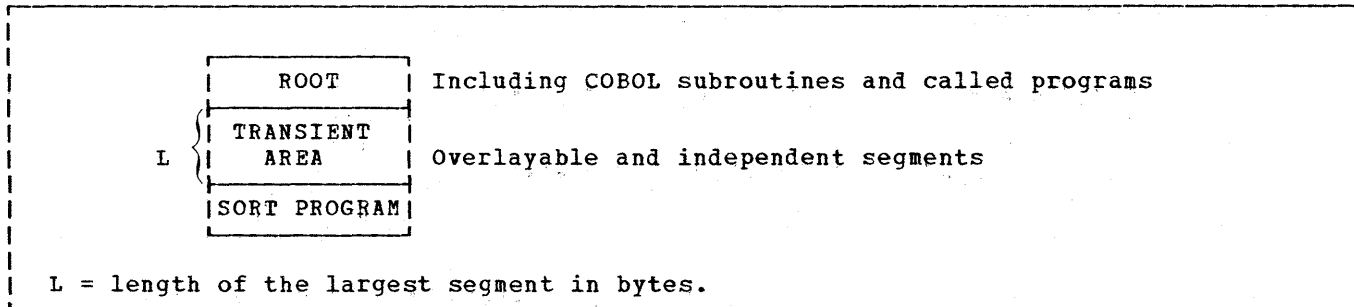


Figure 28. Location of Sort Program in a Segmentation Structure

PART II

PROCESSING COBOL FILES ON MASS STORAGE DEVICES →

MASS
STG

PROCESSING 3540 DISKETTE FILES →

3540

VSAM →

VSAM

DETAILED FILE PROCESSING CAPABILITIES →

FILE
PROC

PROCESSING ASCII TAPE FILES →

ASCII

RECORD FORMATS →

REC
FMTS





A mass storage device is one on which records can be stored in such a way that the location of any one record can be determined without extensive searching. Records can be accessed directly rather than serially.

The recording surface of a mass storage device is divided into many tracks. A track is defined as a circumference of the recording surface. The number of tracks per recording surface and the capacity of a track for each device are shown in Table 9.

Table 9. Recording Capacities of Mass Storage Devices

Device	Capacity
2311	200 tracks per surface; 3625 bytes per track.
2314, 2319	200 tracks per surface; 7294 bytes per track.
2321	100 tracks per strip; 2000 bytes per track.
3330	400 tracks per surface; 13030 bytes per track.
3340 Model 35	348 tracks per surface; 8368 bytes per track.
3340 Model 70	696 tracks per surface; 8368 bytes per track.

Each device has some type of access mechanism through which data is transferred to and from the device. The mechanisms are different for each device, but each mechanism contains a number of read/write heads that transfer data as the recording surfaces rotate past them. Only one head can transfer data (either reading or writing) at a time.

FILE ORGANIZATION

Records in a file must be logically organized so that they can be retrieved efficiently for processing. Four methods of organization for mass storage devices are supported by the DOS/VS COBOL compiler: sequential, direct, indexed, and VSAM. VSAM is discussed in the chapter entitled "Virtual Storage Access Method (VSAM)".

SEQUENTIAL ORGANIZATION

In a sequential file, records are organized solely on the basis of their successive physical location in the file. The records are read or updated in the same order in which they appear.

Individual records cannot be located quickly. Records usually cannot be deleted or added unless the entire file is rewritten. This organization is used when most of the records in the file are processed each time the file is used.

DIRECT ORGANIZATION

A file with direct organization is characterized by some predictable relationship between the key of a record and the address of that record on a mass storage device. This relationship is established by the programmer.

Direct organization is generally used for files where the time required to locate individual records must be kept to an absolute minimum, or for files whose characteristics do not permit the use of sequential or indexed organization.

This organization method has considerable flexibility. The accompanying disadvantage is that although the Disk Operating System Virtual Storage provides the routines to read or write a file of this type, the programmer is largely responsible for the logic and programming required to locate the key of a record and its address on a mass storage device.

INDEXED ORGANIZATION

An indexed file is similar to a sequential file in that rapid sequential processing is possible. The indexes associated with an indexed file also allow quick retrieval of individual records through random access. Moreover, a separate area of the file is set aside for additions; this eliminates the need to rewrite the entire file when adding records, a process that would usually be necessary with a sequentially organized file. Although the added records are not

physically in key sequence, the indexes are constructed in such a way that the added records can be quickly retrieved in key sequence, thus making rapid sequential access possible.

In this method of organization, the system has control over the location of the individual records. Since the characteristics of the file are known, most of the mechanics of locating a particular record are handled by the system.

DATA MANAGEMENT CONCEPTS

The data management facilities of the Disk Operating System Virtual Storage are provided by a group of routines that are collectively referred to as the Input/Output Control System (IOCS). A distinction is made between two types of routines:

1. Physical IOCS (PIOCS) -- the physical input/output routines included in the Supervisor. PIOCS is used by all programs run within the system. It includes facilities for scheduling input/output operations, checking for and handling error conditions related to input/output devices, and handling input/output interruptions to maintain maximum input/output speeds without burdening the programmer's problem program.
2. Logical IOCS (LIOCS) -- the logical input/output routines linked with the programmer's problem program. These routines provide an interface between the programmer's file processing routines and the PIOCS routines.

LIOCS performs those functions that a programmer needs to locate and access a logical record for processing. A logical record is one unit of information in a file of similar units, for example, one employee's record in a master payroll file, one part-number record in an inventory file, or one customer account record in an account file. One or more logical records may be included in one physical record. LIOCS refers to the routines that perform the following functions:

- a. Blocking and deblocking records
- b. Switching between input/output areas when two areas are specified for a file

- c. Handling end-of-file and end-of-volume conditions
- d. Checking and writing labels

A brief description of functions performed by LIOCS and their relationship to a COBOL program follows.

Whenever COBOL imperative-statements (READ, WRITE, REWRITE, etc.) are used in a program to control the input/output of records in a file, that file must be defined by a DTF (Define The File) or, for VSAM, an ACB (Access Method Control Block). A DTF or ACB is created for each file opened in a COBOL program from information specified in the Environment Division, FD entry, and input/output statements in the source program. The DTF for each file is part of the object module that is generated by the compiler. The ACB is generated at object time. They describe the characteristics of the logical file, indicate the type of processing to be used for the file, and specify the storage areas and routines used for the file. Further and more detailed information in VSAM is to be found in the chapter "VSAM."

One of the constants in the DTF table is the address of a logic module that is to be used at execution time to process that file. A logic module contains the coding necessary to perform data management functions required by the file such as blocking and deblocking, initiating label checking, etc.

Generally, these logic modules are assembled separately and cataloged in the relocatable library under a standard name. At link edit time, the Linkage Editor searches the relocatable library using the virtual reference to locate the logic module. The logic module is then included as part of the program phase. Note that since the Autolink feature of the Linkage Editor is responsible for including the logic modules, the COBOL programmer need not specify any INCLUDE statements.

The type of DTF table prepared by the compiler depends on the organization of the file and the device to which it is assigned. The DTF's used for processing files assigned to mass storage devices are as follows:

- DTFSD -- Sequential organization, sequential access
- DTFDA -- Direct organization, sequential or random access
- DTFIS -- Indexed organization,

For a 3540 diskette unit, the DTF is DTFDU. More detail on this is given in the chapter "Processing 3540 Diskette Unit Files."

The remainder of this chapter provides information about preparing programs which process files assigned to mass storage devices. Included are general descriptions of the organization, the COBOL statements that must be specified in order to build the correct DTF tables, and coding examples.

SEQUENTIAL ORGANIZATION (DTFSD)

In a sequential file on a mass storage device, records are written one after another -- track by track, cylinder by cylinder -- at successively higher addresses.

Records may be fixed-length, spanned, or variable-length, blocked or unblocked, or undefined. Since the file is always accessed sequentially, it is not formatted with keys.

Processing a sequentially organized file for selected records is inefficient. If it is done infrequently, the time spent in locating the records is not significant. The slowest way is to read the records sequentially until the desired one is located. On the average, half of the file must be read to locate one record.

Additions and deletions require a complete rewrite of a sequentially organized file on a mass storage device. Sequential organization is used on mass storage devices primarily for tables and intermediate storage rather than for master files.

Sequentially organized files formatted with keys cannot be created using DTFSD. DTFDA may be used to create and access (sequentially or randomly) such files.

PROCESSING A SEQUENTIALLY ORGANIZED FILE

To create, retrieve, or update a DTFSD file, the following specifications should be made in the source program:

ENVIRONMENT DIVISION

Required clauses:

SELECT [OPTIONAL] file-name

ASSIGN TO SYSnnn- { UT } - { 2311 }
 { DA } - { 2314 } -S
 { 2321 }
 { 2319 }
 { 3330 }
 { 3340 }

Optional clauses:

- RESERVE Clause
- FILE-LIMIT Clause
- ACCESS MODE IS SEQUENTIAL
- PROCESSING MODE IS SEQUENTIAL
- RERUN Clause
- SAME Clause
- APPLY WRITE-ONLY Clause (create only)
- APPLY WRITE-VERIFY Clause (create or update only)

Invalid clauses:

- ACCESS MODE IS RANDOM
- ACTUAL KEY Clause
- NOMINAL KEY Clause
- RECORD KEY Clause
- TRACK-AREA Clause
- MULTIPLE FILE TAPE Clause
- APPLY EXTENDED-SEARCH Clause
- APPLY CYL-OVERFLOW Clause
- APPLY { MASTER-INDEX } Clause
 { CYL-INDEX }
- APPLY CORE-INDEX Clause

DTFSD files may be opened as INPUT, OUTPUT, or I-O. When creating such a file, an INVALID KEY condition occurs when the file limit has been reached and an attempt is made to place another record on the mass storage device. The file limit is determined from the EXTENT control statements.

When a DTFSD file is opened as OUTPUT, each WRITE statement signifies the creation of a new record. When opened as I-O, each WRITE statement signifies that the record just read is to be rewritten.

DIRECT ORGANIZATION (DTFDA)

With direct organization, there is a definite relationship between the key of a record and its address. This relationship permits rapid access to any record if the file is carefully organized. The programmer develops a record address that ranges from zero to some maximum by converting a particular field in each record to a track address. Each byte in the address is a binary number. To



reference a particular record, the programmer must supply both the track address and the identifier that makes each record unique on its track. Both the track address and the identifier are supplied by the programmer in the ACTUAL KEY clause. This will be discussed in detail later in this chapter.

With direct organization, records may be fixed length, spanned or undefined. The records must be unblocked. R0 (record zero) of each track is used as a capacity record. It contains the address of the last record written on the track, and is used by the system to determine whether a new record will fit on the track. The capacity records are updated by the system as records are added to the file. The capacity records do not account for deletions: as far as the system is concerned, once a track is full it remains full (even if the programmer deletes records) until the file is reorganized.

Often, more records are converted to a given track address than will actually fit on the track. These surplus records are known as overflow records and are usually written into a separate area known as an overflow area.

As already noted, the programmer has an unlimited choice in deciding where records are to be located in a directly organized file. The logic and programming are his responsibility.

When creating or making additions to the file, the programmer must specify the location for a record (track address) and the identifier that makes each record on the track unique. If there is space on the track, the system writes the record and updates the capacity record. If the specified track is full, a standard error condition occurs, and the programmer may specify another track address in his USE AFTER STANDARD ERROR declarative routine.

In the case of one maximum size record per track (when spanned records are not specified), the data length plus the length of the symbolic key cannot exceed the following values:

- 2311 -- 3605 bytes
- 2314, 2319 -- 7249 bytes
- 2321 -- 1984 bytes
- 3330 -- 12974 bytes
- 3340 -- 8293 bytes

When reading or updating the file, the programmer must supply the track address and the unique identifier on the track for the specific record being sought. The system locates the track and searches that track for the record with the specified

identifier. If the record is not found, COBOL indicates this to the programmer by raising an INVALID KEY condition. Only the track specified by the programmer is searched. If, however, the APPLY EXTENDED-SEARCH clause has been specified for the file, the entire cylinder is searched for the desired record. In this case, the INVALID KEY condition arises only if the record cannot be found on the cylinder. To ensure file integrity, the upper limit of each extent of a file using EXTENDED-SEARCH must be the last track of a cylinder.

Error recovery from a DTFDA file is described in detail in the chapter "Advanced Processing Capabilities."

ACCESSING A DIRECTLY ORGANIZED FILE

A directly organized file (DTFDA) may be accessed either sequentially or randomly.

ACCESSING A DIRECTLY ORGANIZED FILE

SEQUENTIALLY: When reading a direct file sequentially, records are retrieved in logical sequence; this logical sequence corresponds exactly to the physical sequence of the records. To retrieve a DTFDA file sequentially, the following specifications are made in the source program:

ENVIRONMENT DIVISION

Required clauses:

```
SELECT [OPTIONAL] file-name
ASSIGN TO SYSnnn-DA- {
    (2311)
    2321 } (A)
    2314 } (D)
    2319 }
    3330 }
    3340 }
```

Optional clauses:

```
FILE-LIMIT Clause
ACCESS MODE IS SEQUENTIAL
PROCESSING MODE IS SEQUENTIAL
ACTUAL KEY Clause
RERUN Clause
SAME Clause
```

Invalid clauses:

```
RESERVE Clause
ACCESS MODE IS RANDOM
NOMINAL KEY Clause
RECORD KEY Clause
TRACK-AREA Clause
MULTIPLE FILE TAPE Clause
APPLY WRITE-ONLY Clause
```

APPLY CYL-OVERFLOW Clause
 APPLY EXTENDED- SEARCH Clause
 APPLY WRITE-VERIFY Clause

APPLY { MASTER-INDEX } Clause
 { CYL-INDEX }

APPLY CORE-INDEX Clause

When DTFDA records are retrieved sequentially, the file may be opened only as INPUT. The AT END condition occurs when the last record has been read and execution of another READ is attempted.

Note that in the ASSIGN clause, an A must be specified for files with actual track addressing, and a D must be specified for files with relative track addressing.

ACCESSING A DIRECTLY ORGANIZED FILE

RANDOMLY: To create a directly organized file randomly, the following specifications are made in the source program:

ENVIRONMENT DIVISION

Required clauses:

SELECT file-name

ASSIGN TO SYSnnn-DA- { 2311 } (A)
 { 2321 }
 { 2314 } (D)
 { 2319 }
 { 3330 }
 { 3340 }

ACCESS MODE IS RANDOM
 ACTUAL KEY Clause

Optional clauses:

FILE-LIMIT Clause
 PROCESSING MODE IS SEQUENTIAL
 RERUN Clause
 SAME Clause
 APPLY WRITE-VERIFY Clause

Invalid clauses:

RESERVE Clause
 ACCESS MODE IS SEQUENTIAL
 NOMINAL KEY Clause
 RECORD KEY Clause
 TRACK-AREA Clause
 MULTIPLE FILE TAPE Clause
 APPLY WRITE-ONLY Clause
 APPLY EXTENDED-SEARCH Clause
 APPLY WRITE-VERIFY Clause
 APPLY CYL-OVERFLOW Clause

APPLY { MASTER-INDEX } Clause
 { CYL-INDEX }

APPLY CORE-INDEX Clause

Note that in the ASSIGN clause, an A must be specified for files with actual track addressing, and a D must be specified for files with relative track addressing.

To retrieve or update a directly organized file randomly, the following specifications must be made in the source program.

ENVIRONMENT DIVISION

Required clauses:

SELECT file-name

ASSIGN TO SYSnnn-DA- { 2311 } (A)
 { 2314 }
 { 2321 } (D)
 { 2319 } (U)
 { 3330 } (W)
 { 3340 }

ACCESS MODE IS RANDOM
 ACTUAL KEY Clause

Note that in the ASSIGN clause an A must be specified for files with actual track addressing, a D must be specified for files with relative track addressing, a U must be specified for files with actual track addressing when the REWRITE statement is used, and W must be specified for files with relative track addressing when the REWRITE statement is used.

The optional and invalid clauses are the same as those specified previously for creating a directly organized file.

Exception: APPLY EXTENDED-SEARCH is optional when retrieving or updating a directly organized file randomly.

ACTUAL KEY CLAUSE

Note that the ACTUAL KEY clause is required for DTFDA files when ACCESS IS RANDOM, is optional for DTFDA files when ACCESS IS SEQUENTIAL, and is not used for DTFSD files.

The actual key consists of two components. One component expresses the track address at which the record is to be placed for an output operation, or at which the search is to begin for an input operation. The track address can be expressed either as an actual address or as a relative address, depending upon the addressing scheme chosen when the file was created. The other component is associated with the record itself and serves as its unique identifier. The structures of both actual keys are shown in Figure 29.



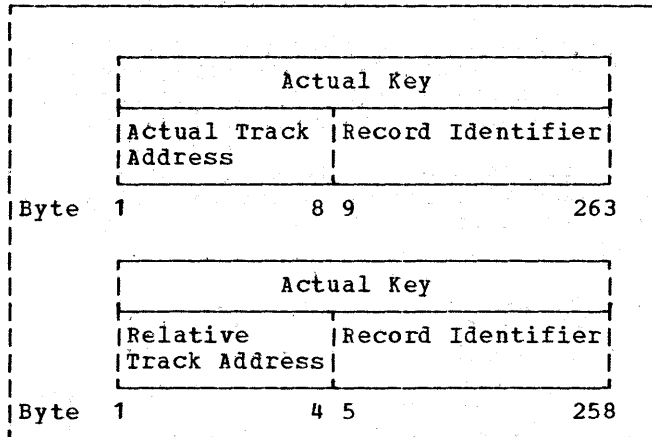


Figure 29. Structures of the Actual Key

The format of the ACTUAL KEY clause is:

ACTUAL KEY IS data-name

When actual track addressing is used, data-name may be any fixed item from 9 through 263 bytes in length. It must be defined in the Working-Storage, File, or Linkage Section. The first eight bytes are used to specify the actual track address. The structure of these bytes and permissible specifications for the mass storage devices are shown in Figure 30. The programmer may select from 1 to 255 bytes for the record identifier portion of the actual key field.

Note: If a SEEK statement is used when retrieving a direct file randomly, actual track addressing is required.

When relative track addressing is used, data-name may be any fixed item from 5

through 258 bytes in length. It must be defined in the File Section, the Working-Storage Section, or the Linkage Section. The first four bytes of data-name are the track identifier. The identifier is used to specify the relative track address for the record and must be defined as an 8-integer binary data item whose maximum value does not exceed 16,777,215. The remainder of data-name, which is 1 through 254 bytes in length, is the record identifier. It represents the symbolic portion of the key field used to identify a particular record on a track.

For a complete discussion of the ACTUAL KEY clause, see the publication IBM DOS Full American National Standard COBOL.

Randomizing Techniques

One method of determining the value of the track address portion of the field defined in the ACTUAL KEY clause is referred to as indirect addressing. Indirect addressing generally is used when the range of keys for a file includes a high percentage of unused values. For example, employee numbers may range from 000001 to 009999, but only 3000 of the possible 9999 numbers are currently assigned. Indirect addressing is also used for nonnumeric keys. Key, in this discussion, refers to that field of the record being written that will be converted to the track address portion.

Indirect addressing signifies that the key is converted to a value for the actual track address by using some algorithm intended to limit the range of addresses.

Byte	Pack	Cell			Cylinder		Head		Record
	M	B	B	C	C	H	E	R	
Device	0	1	2	3	4	5	6	7	
2311	0-221	0	0	0	0-199	0	0-9	0-255	
2314	0-221	0	0	0	0-199	0	0-19	0-255	
2321	0-221	0	0-9	0-19	0-9	0-4	0-19	0-255	
3330	0-221	0	0	0-403		0	0-18	0-255	
3340 Model 35	0-221	0	0	0-347		0	0-11	0-255	
3340 Model 70	0-221	0	0	0-695		0	0-11	0-255	

Figure 30. Permissible Specifications for the First Eight Bytes of the Actual Key

Such an algorithm is called a randomizing technique. Randomizing techniques need not produce a unique address for every record and, in fact, such techniques usually produce synonyms. Synonyms are records whose keys randomize to the same address.

Two objectives must be considered in selecting a randomizing technique:

1. Every possible key in the file must randomize to an address within the designated range.
2. The addresses should be distributed evenly across the range so that there are as few synonyms as possible.

Note that one way to minimize synonyms is to allocate more space for the file than is actually required to contain all the records. For example, the percentage of locations that are actually used might be 80% to 85% of the allocated space.

When actual track addressing is used, the first eight bytes of the ACTUAL KEY field can be thought of as a "discontinuous binary address." This is significant to the programmer because he must keep two considerations in mind. First, the cylinder and head number must be in binary notation, so the results of the randomizing formula must be in binary format. Second, the address is "discontinuous" since a mathematical overflow from one element (e.g., head number) does not increment the adjacent element (e.g., cylinder number).

DIVISION/REMAINDER METHOD: One of the simplest ways to indirectly address a directly organized file is by using the division/remainder method. (For a discussion of other randomizing techniques, see the publication Introduction to IBM Direct Access Storage Devices and Organization Methods, Order No. GC20-1649.)

1. Determine the amount of locations required to contain the data file. Include a packing factor for additional space to eliminate synonyms. The packing factor should be approximately 20% of the total space allocated to contain the data file.
2. Select, from the prime number table, the nearest prime number that is less than the total of step 1. A prime number is a number divisible only by itself and the integer 1. Table 10 is a partial list of prime numbers.
3. Clear any zones from the first eight bytes of the actual key field. This

can be accomplished by moving the key to a field described as COMPUTATIONAL.

4. Divide the key by the prime number selected.
5. Ignore the quotient; utilize the remainder as the relative location within the data file.
6. (For actual track addressing only) Locate the beginning of the space available and manipulate the relative address, to the actual device address if necessary.

For example, assume that a company is planning to create an inventory file on a 2311 disk storage device. There are 8000 different inventory parts, each identified by an 8-character part number. Using a 20% packing factor, 10,000 record positions are allocated to store the data file.

Method A: The closest prime number to 10,000, but under 10,000, is 9973. Using one inventory part number as an example, in this case #25DF3514, and clearing the zones we have 25463514. Dividing by 9973 we get a quotient of 2553 and a remainder of 2445. 2445 is the relative location of the record within the data file corresponding to part number 25DF3514. The record address can be determined from the relative location as follows:

1. (For actual track addressing only) Determine the beginning point for the data file (e.g., cylinder 100, track 0).
2. Determine the number of records that can be stored on a track (e.g., twelve per track on a 2314 disk pack, assuming each inventory record is 200 bytes long).

Because each data record contains non-data components, such as a count area and interrecord gaps, track capacity for data storage will vary with record length. As the number of separate records on a track increases, interrecord gaps occupy additional byte positions so that data capacity is reduced. Track capacity formulas provide the means to determine total byte requirements for records of various sizes on a track. These formulas can be found in the publications IBM Component Descriptions, Order Nos. GA26-5988 and GA26-3599.

3. Divide the relative number (2445) by the number of records to be stored on each track.

4. (For actual track addressing only)
The result, quotient = 203, is now divided into cylinder and head designation. Since the 2311 disk pack has ten heads, the quotient of 203 is divided by 10 to show:

Cylinder or CC = 20
Head or HH = 03 (high-order zero added)

- 4B. (For relative track addressing only)
The result, quotient = 203, now becomes the track identifier of the actual key.

Method B: Utilizing the same example, another approach will also provide the relative track address:

1. The number of records that may be contained on one track is twelve. Therefore, if 10,000 record locations are to be provided, 834 tracks must be reserved.
2. The prime number nearest, but less than 834, is 829.
3. Divide the zone-stripped key by the prime value. (In the example, 25463514 divided by 829 provides a quotient of 30715 and a remainder of 779. The remainder is the relative address.)

Table 10. Partial List of Prime Numbers
(Part 1 of 2)

A (Number)	B (Nearest Prime Number Less Than A)
500	499
600	599
700	691
800	797
900	887
1000	997
1100	1097
1200	1193
1300	1297
1400	1399
1500	1499
1600	1597
1700	1699
1800	1789
1900	1889
2000	1999
2100	2099
2200	2179
2300	2297
2400	2399
2500	2477
2600	2593
2700	2699
2800	2797
2900	2897
3000	2999
3100	3089
3200	3191
3300	3299
3400	3391
3500	3499
3600	3593
3700	3697
3800	3797
3900	3889
4000	3989
4100	4099
4200	4177
4300	4297
4400	4397
4500	4493
4600	4597
4700	4691
4800	4799
4900	4889
5000	4999
5100	5099
5200	4197
5300	5297
5400	4399
5500	5483

Table 10. Partial List of Prime Numbers
(Part 2 of 2)

A (Number)	B (Nearest Prime Number Less Than A)
5600	5591
5700	5693
5800	5791
5900	5897
6000	5987
6100	6091
6200	6199
6300	6299
6400	6397
6500	6491
6600	6599
6700	6691
6800	6793
6900	6899
7000	6997
7100	7079
7200	7193
7300	7297
7400	7393
7500	7499
7600	7591
7700	7699
7800	7793
7900	7883
8000	7993
8100	8093
8200	8191
8300	8297
8400	8389
8500	8467
8600	8599
8700	8699
8800	8793
8900	8899
9000	8899
9100	9091
9200	9199
9300	9293
9400	9397
9500	9497
9600	9587
9700	9697
9800	9791
9900	9887
10,000	9973
10,100	10,099
10,200	10,193
10,300	10,289
10,400	10,399
10,500	10,499
10,600	10,597

MASS
STG

4. (For actual track addressing only) To convert the relative address to an actual device address, divide the relative address by the number of tracks in a cylinder. The quotient will provide the cylinder number and the remainder will be the track number. For example, the 2311 disk pack would utilize 779 as:

Cylinder or CC = 77
Track or HH = 9

Figure 31 is a sample COBOL program which creates a direct file with actual track addressing using Method B and provides for the possibility of synonym overflow. Synonym overflow will occur if a record randomizes to a track that is already full. The following description highlights the features of the example. Circled numbers on the program listing correspond to the numbers in the text.

- ① The value 10 is added to TRACK-1 to ensure that the program does not write on cylinder 0. Cylinder 0 must be reserved for the Volume Table of Contents.
- Since the prime number used as a divisor is 829, the largest possible remainder will be 828. Adding 10 to TRACK-1 adjusts the largest possible remainder to 838.
- ② If synonym overflow occurs, control is given to the error procedure declarative specified in the first section of the Procedure Division. The declarative provides that:
- Any record which cannot fit on a track (i.e., tracks 0 through 8 of any cylinder) will be written in the first available position on the following track(s).
 - Any record which cannot fit within a single cylinder will be written on cylinder 84 (i.e., the cylinder overflow area).

- If a record cannot fit on either cylinders 1 through 83, or on cylinder 84, the job is terminated.

- ③ The standard error condition "no room found" is tested before control is given to the synonym routine. Other standard error conditions as well as invalid key conditions result in job termination.

ERROR-COND is the identifier which specifies the error condition that caused control to be given to the error declarative. ERROR-COND is printed on SYSLSST whenever the error declarative section is entered. TRACK-ID and C-REC are also printed on SYSLSST. They are printed before the execution of each WRITE statement. This output has been provided in order to facilitate an understanding of the logic involved in the creation of D-FILE.

- ④ The first twelve records which randomize to cylinder 002 track 8 are actually written on track 8.
- ⑤ The next twelve records which randomize to cylinder 002 track 8 are adjusted by the SYNONYM-ROUTINE and written on cylinder 002 track 9.
- ⑥ The next twelve records which randomize to cylinder 002 track 8 are adjusted by the SYNONYM-ROUTINE and written on cylinder 84 track 0 (i.e., the overflow cylinder).
- ⑦ The last two records which randomize to cylinder 002 track 8 are adjusted by the SYNONYM-ROUTINE and written on cylinder 84 track 1 (i.e., the overflow cylinder).

```
// JOB METHODBA
// OPTION NODECK, LINK, LIST, LISTX, SYM, ERRS
// EXEC FCOBOL
```

MASS
STG

1 IBM DOS VS COBOL

REL 1.0

PP NO. 5746-CB1

08.47.44 10/04/73

```
IDENTIFICATION DIVISION.
PROGRAM-ID. METHOD-B.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-370.
OBJECT-COMPUTER. IBM-370.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT D-FILE ASSIGN SYS015-DA-2314-A-MASTER
    ACCESS IS RANDOM
    ACTUAL KEY IS ACT-KEY.
    SELECT C-FILE ASSIGN TO SYS007-UR-2540R-S.
DATA DIVISION.
FILE SECTION.
FD D-FILE
    LABEL RECORDS ARE STANDARD.
01 D-REC.
    02 PART-NUM PIC X(8).
    02 NUM-ON-HAND PIC 9(4).
    02 PRICE PIC 9(5)V99.
    02 FILLER PIC X(181).
FD C-FILE
    LABEL RECORDS ARE OMITTED.
01 C-REC.
    02 PART-NUM PIC X(8).
    02 NUM-ON-HAND PIC 9(4)9.
    02 PRICE PIC 9(5)V99.
WORKING-STORAGE SECTION.
77 HD PIC 9 VALUE ZERO.
77 SAVE PIC S9(8) COMP SYNC.
77 QUOTIENT PIC S9(5) COMP SYNC.
01 ERROR-COND.
    02 FILLER PIC 99 VALUE ZERO.
    02 ERR PIC 9 VALUE ZERO.
    02 FILLER PIC 9(5) VALUE ZERO.
01 TRACK-1 PIC 9999.
01 TRACK-ID REDEFINES TRACK-1.
    02 CYL PIC 999.
    02 HEAD PIC 9.
01 KEY-1.
    02 M PIC S999 COMP SYNC VALUE ZEROES.
    02 BE PIC S9 COMP SYNC VALUE ZERO.
    02 CC PIC S999 COMP SYNC.
    02 HH PIC S999 COMP SYNC.
    02 R PIC X VALUE LOW-VALUE.
    02 REC-ID PIC X(8).
01 KEY-2 REDEFINES KEY-1.
    02 FILLER PIC X.
    02 ACT-KEY PIC X(16).
```

Figure 31. Creating a Direct File Using Method B (Part 1 of 4)

```

PROCEDURE DIVISION.
DECLARATIVES.
ERRCR-PROCEDURE SECTION. USE AFTER STANDARD ERROR PROCEDURE
    ON D-FILE GIVING ERROR-COND.

```

```

ERROR-ROUTINE.
  EXHIBIT NAMED ERROR-COND.
  IF ERR = 1 GO TO SYNONYM-ROUTINE ELSE } ③
    DISPLAY 'OTHER STANDARD ERROR' REC-ID
  GO TO EOJ.
SYNONYM-ROUTINE.
  IF CC = 84 AND HD = 9 DISPLAY 'OVERFLOW AREA FULL'
  GO TO EOJ.
  IF CC = 84 ADD 1 TO HD GO TO ADJUST-HD.
  IF HH = 9 GO TO END-CYLINDER.
  ADD 1 TO HH.
  GO TO WRITLS.
END-CYLINDER.
  MOVE 84 TO CC.
ADJUST-HD.
  MOVE HD TO HH.
  GO TO WRITES.
END DECLARATIVES.
FILE-CREATION SECTION.
  OPEN INPUT C-FILE
  OUTPUT D-FILE.
READS.
  READ C-FILE AT END GO TO EOJ.
  MOVE CORRESPONDING C-REC TO D-REC.
  MOVE PART-NUM OF C-REC TO REC-ID SAVE.
  DIVIDE SAVE BY 829 GIVING QUOTIENT REMAINDER TRACK-1. } ①
  ADD 10 TO TRACK-1.
  MOVE CYL TO CC.
  MOVE HEAD TO HH.
WRITES.
  EXHIBIT NAMED TRACK-ID C-REC CC HH.
  WRITE D-REC INVALID KEY GO TO INVALID-KEY.
  GO TO READS.
INVALID-KEY.
  DISPLAY 'INVALID KEY' REC-ID.
EOJ.
  CLOSE C-FILE D-FILE.
  STOP RUN.

```

```

// LBLTYP NSD(01)
// EXEC LNKEDT

```

Figure 31. Creating a Direct File Using Method B (Part 2 of 4)

```
// ASSGN SYS007,X'00C'
// ASSGN SYS015,X'231'
// DIEB MASTER,,99/365,DA
// EXTENT SYS015,111111,1,0,20,840
// EXEC
```

TRACK-ID = 0010	C-REC = 82900000	CC = 001	HH = 000
TRACK-ID = 0011	C-REC = 82900001	CC = 001	HH = 001
TRACK-ID = 0028	C-REC = 8290001801	CC = 002	HH = 002
TRACK-ID = 0028	C-REC = 8290001802	CC = 002	HH = 008
TRACK-ID = 0028	C-REC = 8290001803	CC = 002	HH = 008
TRACK-ID = 0028	C-REC = 8290001804	CC = 002	HH = 008
TRACK-ID = 0028	C-REC = 8290001805	CC = 002	HH = 008
TRACK-ID = 0028	C-REC = 8290001806	CC = 002	HH = 008
TRACK-ID = 0028	C-REC = 8290001807	CC = 002	HH = 008
TRACK-ID = 0028	C-REC = 8290001808	CC = 002	HH = 008
TRACK-ID = 0028	C-REC = 8290001809	CC = 002	HH = 008
TRACK-ID = 0028	C-REC = 8290001810	CC = 002	HH = 008
TRACK-ID = 0028	C-REC = 8290001811	CC = 002	HH = 008
TRACK-ID = 0028	C-REC = 8290001812	CC = 002	HH = 008
TRACK-ID = 0028	C-REC = 8290001813	CC = 002	HH = 008
TRACK-ID = 0028	C-REC = 8290001814	CC = 002	HH = 008
TRACK-ID = 0186	C-REC = 290001815	CC = 018	HH = 006
TRACK-ID = 0186	C-REC = 290001816	CC = 018	HH = 006
TRACK-ID = 0028	C-REC = 8290001817	CC = 002	HH = 008
TRACK-ID = 0028	C-REC = 8290001818	CC = 002	HH = 008
TRACK-ID = 0028	C-REC = 8290001819	CC = 002	HH = 008
TRACK-ID = 0028	C-REC = 8290001820	CC = 002	HH = 008
TRACK-ID = 0028	C-REC = 8290001821	CC = 002	HH = 008
TRACK-ID = 0028	C-REC = 8290001822	CC = 002	HH = 008
TRACK-ID = 0028	C-REC = 8290001823	CC = 002	HH = 008
ERROR-COND = 00100000			
TRACK-ID = 0028	C-REC = 8290001823	CC = 002	HH = 009
TRACK-ID = 0028	C-REC = 8290001824	CC = 002	HH = 008
ERROR-COND = 00100000			
TRACK-ID = 0028	C-REC = 8290001824	CC = 002	HH = 009



Figure 31. Creating a Direct File Using Method B (Part 3 of 4)

TRACK-ID = 0028 C-REC = 8290001825	CC = 002 HH = 008
ERROR-COND = 00100000	
TRACK-ID = 0028 C-REC = 8290001825	CC = 002 HH = 009
TRACK-ID = 0028 C-REC = 8290001826	CC = 002 HH = 008
ERROR-COND = 00100000	
TRACK-ID = 0028 C-REC = 8290001826	CC = 002 HH = 009
TRACK-ID = 0011 C-REC = 8290001827	CC = 001 HH = 001
TRACK-ID = 0011 C-REC = 8290001828	CC = 001 HH = 001
TRACK-ID = 0011 C-REC = 8290001829	CC = 001 HH = 001
TRACK-ID = 0028 C-REC = 8290001830	CC = 002 HH = 008
ERROR-COND = 00100000	
TRACK-ID = 0028 C-REC = 8290001830	CC = 002 HH = 009
TRACK-ID = 0028 C-REC = 8290001831	CC = 002 HH = 008
ERROR-COND = 00100000	
TRACK-ID = 0028 C-REC = 8290001831	CC = 002 HH = 009
TRACK-ID = 0028 C-REC = 8290001832	CC = 002 HH = 008
ERROR-COND = 00100000	
TRACK-ID = 0028 C-REC = 8290001832	CC = 002 HH = 009
TRACK-ID = 0028 C-REC = 8290001833	CC = 002 HH = 008
ERROR-COND = 00100000	
TRACK-ID = 0028 C-REC = 8290001833	CC = 002 HH = 009
TRACK-ID = 0028 C-REC = 8290001834	CC = 002 HH = 008
ERROR-COND = 00100000	
TRACK-ID = 0028 C-REC = 8290001834	CC = 002 HH = 009
TRACK-ID = 0028 C-REC = 8290001835	CC = 002 HH = 008
ERROR-COND = 00100000	
TRACK-ID = 0028 C-REC = 8290001835	CC = 002 HH = 009
TRACK-ID = 0028 C-REC = 8290001836	CC = 002 HH = 008
ERROR-COND = 00100000	
TRACK-ID = 0028 C-REC = 8290001836	CC = 002 HH = 009
TRACK-ID = 0028 C-REC = 8290001837	CC = 002 HH = 008
ERROR-COND = 00100000	
TRACK-ID = 0028 C-REC = 8290001837	CC = 002 HH = 009
TRACK-ID = 0028 C-REC = 8290001838	CC = 002 HH = 008
ERROR-COND = 00100000	
TRACK-ID = 0028 C-REC = 8290001838	CC = 002 HH = 009



Figure 31. Creating a Direct File Using Method B (Part 4 of 4)

Figure 32 is a sample COBOL program which creates a direct file with relative track addressing using Method B. The sample program provides for the possibility of synonym overflow. Synonym overflow will occur if a record randomizes to a track which is already full. The following discussion highlights some basic features. Circled numbers on the program listing correspond to numbers in the text.

caused control to be given to the error declarative. ERROR-COND is printed on SYSLST whenever the error declarative section is entered. TRACK-ID and C-REC are also printed on SYSLST before execution of each WRITE statement. This output has been provided in order to facilitate an understanding of the logic involved in the creation of D-FILE.



- ① Since the prime number used as a divisor is 829, the largest possible remainder will be 828.
- ② If synonym overflow occurs, control is given to the USE AFTER STANDARD ERROR declarative specified in the first section of the Procedure Division. The declarative provides that any record that cannot fit on the track to which it randomizes will be written on the first subsequent track available.
- ③ The standard error condition "no room found" is tested before control is given to the SYNONYM-ROUTINE. Other standard error conditions as well as invalid key conditions result in job termination (EOJ).

ERROR-COND is the identifier which specifies the error condition that
- ④ The first twelve records which randomize to relative track 18 are actually written on relative track 18.
- ⑤ The next twelve records which randomize to relative track 18 are adjusted by the SYNONYM-ROUTINE and are actually written on relative track 19.
- ⑥ The next twelve records which randomize to relative track 18 are adjusted by the SYNONYM-ROUTINE and are actually written on relative track 20.
- ⑦ The last two records which randomize to relative track 18 are adjusted by the SYNONYM-ROUTINE and are actually written on relative track 21.

```
// JOB METHODBR
// OPTION NODECK, LINK, LIST, LISTX, SYM, ERRS
// EXEC FCOBOL
```

1 IBM DOS VS COBOL

REL 1.0

PP NO. 5746-CB1

08.40.53 10/04.

CEL QUOTL

```
IDENTIFICATION DIVISION.
PROGRAM-ID. METHODE.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-370.
OBJECT-COMPUTER. IBM-370.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT D-FILE ASSIGN TO SYS015-DA-2314-D-MASTER
    ACCESS IS RANDOM
    ACTUAL KEY IS ACT-KEY.
    SELECT C-FILE ASSIGN TO SYS007-UR-2540R-S.
DATA DIVISION.
FILE SECTION.
FD D-FILE
    LABEL RECORDS ARE STANDARD.
01 D-REC.
    05 PART-NUM PIC X(8).
    05 NUM-ON-HAND PIC 9(4).
    05 PRICE PIC 9(5)V99.
    05 FILLER PIC X(181).
FD C-FILE
    LABEL RECORDS ARE OMITTED.
01 C-REC.
    05 PART-NUM PIC X(8).
    05 NUM-ON-HAND PIC 9(4).
    05 PRICE PIC 9(5)V99.
    05 FILLER PIC X(61).
WORKING-STORAGE SECTION.
77 SAVE PIC S9(8) COMP SYNC.
77 QUOTIENT PIC S9(8) COMP SYNC.
01 ACT-KEY.
    02 TRACK-ID PIC S9(8) COMP SYNC.
    02 REC-ID PIC X(8).
01 ERROR-COND.
    02 FILLER PIC 99 VALUE ZERO.
    02 ERR PIC 9 VALUE ZERO.
    02 FILLER PIC 9(5) VALUE ZERO.
```

Figure 32. Creating a Direct File with Relative Track Addressing Using Method B (Part 1 of 4)

```

PROCEDURE DIVISION.
DECLARATIVES.
ERROR-PROCEDURE SECTION. USE AFTER STANDARD ERROR PROCEDURE
ON D-FILE GIVING ERROR-COND.
ERROR-ROUTINE.
    EXHIBIT NAMED ERROR-COND.
    IF ERR = 1 GO TO SYNONYM-ROUTINE ELSE } ③
        DISPLAY "OTHER STANDARD ERROR " REC-ID
        GO TO EOJ.
SYNONYM-ROUTINE.
    IF TRACK-ID IS LESS THAN 834, ADD 1 TO TRACK-ID. GO TO
    WRITES.
END DECLARATIVES.
    OPEN INPUT C-FILE
        OUTPUT D-FILE.
READS.
    READ C-FILE AT END GO TO EOJ.
    MOVE CORRESPONDING C-REC TO D-REC.
    MOVE PART-NUM OF C-REC TO REC-ID, SAVE.
    DIVIDE SAVE BY 829 GIVING QUOTIENT REMAINDER TRACK-ID. } ①
WRITES.
    EXHIBIT NAMED TRACK-ID C-REC.
    WRITE D-REC INVALID KEY GO TO INVALID-KEY.
    GO TO READS.
INVALID-KEY.
    DISPLAY "INVALID KEY " REC-ID.
EOJ.
    CLOSE C-FILE D-FILE.
    STOP RUN.
    
```

MASS
STG

```

// LBLTYP NSD(01)
// EXEC LNKEDT
    
```

Figure 32. Creating a Direct File with Relative Track Addressing Using Method B (Part 2 of 4)

```
// ASSGN SYS007,X'00C'
// ASSGN SYS015,X'231'
// DLBL MASTER,,99/365,DA
// EXTENT SYS015,111111,1,0,20,840
// EXEC
```

```
TRACK-ID = 00000000 C-REC = 82900000
TRACK-ID = 00000001 C-REC = 82900001
TRACK-ID = 00000018 C-REC = 8290001801
TRACK-ID = 00000018 C-REC = 8290001802
TRACK-ID = 00000018 C-REC = 8290001803
TRACK-ID = 00000018 C-REC = 8290001804
TRACK-ID = 00000018 C-REC = 8290001805
TRACK-ID = 00000018 C-REC = 8290001806
TRACK-ID = 00000018 C-REC = 8290001807
TRACK-ID = 00000018 C-REC = 8290001808
TRACK-ID = 00000018 C-REC = 8290001809
TRACK-ID = 00000018 C-REC = 8290001810
TRACK-ID = 00000018 C-REC = 8290001811
TRACK-ID = 00000018 C-REC = 8290001812
TRACK-ID = 00000018 C-REC = 8290001813
TRACK-ID = 00000018 C-REC = 8290001814
TRACK-ID = 00000018 C-REC = 8290001815
TRACK-ID = 00000018 C-REC = 8290001816
TRACK-ID = 00000018 C-REC = 8290001817
TRACK-ID = 00000018 C-REC = 8290001818
TRACK-ID = 00000018 C-REC = 8290001819
TRACK-ID = 00000018 C-REC = 8290001820
TRACK-ID = 00000018 C-REC = 8290001821
ERROR-COND = 00100000
TRACK-ID = 00000019 C-REC = 8290001821
TRACK-ID = 00000018 C-REC = 8290001822
ERROR-COND = 00100000
TRACK-ID = 00000019 C-REC = 8290001822
TRACK-ID = 00000018 C-REC = 8290001823
ERROR-COND = 00100000
TRACK-ID = 00000019 C-REC = 8290001823
TRACK-ID = 00000018 C-REC = 8290001824
ERROR-COND = 00100000
TRACK-ID = 00000019 C-REC = 8290001824
```



Figure 32. Creating a Direct File with Relative Track Addressing Using Method B (Part 3 of 4)



```

TRACK-ID = 00000018 C-REC = 8290001825
ERROR-COND = 00100000
TRACK-ID = 00000019 C-REC = 8290001825
TRACK-ID = 00000018 C-REC = 8290001826
ERROR-COND = 00100000
TRACK-ID = 00000019 C-REC = 8290001826
TRACK-ID = 00000018 C-REC = 8290001827
ERROR-COND = 00100000
TRACK-ID = 00000019 C-REC = 8290001827
TRACK-ID = 00000018 C-REC = 8290001828
ERROR-COND = 00100000
TRACK-ID = 00000019 C-REC = 8290001828
TRACK-ID = 00000018 C-REC = 8290001829
ERROR-COND = 00100000
TRACK-ID = 00000019 C-REC = 8290001829
TRACK-ID = 00000018 C-REC = 8290001830
ERROR-COND = 00100000
TRACK-ID = 00000019 C-REC = 8290001830
TRACK-ID = 00000018 C-REC = 8290001831
ERROR-COND = 00100000
TRACK-ID = 00000019 C-REC = 8290001831
TRACK-ID = 00000018 C-REC = 8290001832
ERROR-COND = 00100000
TRACK-ID = 00000019 C-REC = 8290001832
TRACK-ID = 00000018 C-REC = 8290001833
ERROR-COND = 00100000
TRACK-ID = 00000019 C-REC = 8290001833
TRACK-ID = 00000018 C-REC = 8290001834
ERROR-COND = 00100000
TRACK-ID = 00000019 C-REC = 8290001834
TRACK-ID = 00000018 C-REC = 8290001835
ERROR-COND = 00100000
TRACK-ID = 00000019 C-REC = 8290001835
TRACK-ID = 00000018 C-REC = 8290001836
ERROR-COND = 00100000
TRACK-ID = 00000019 C-REC = 8290001836
TRACK-ID = 00000018 C-REC = 8290001837
ERROR-COND = 00100000
TRACK-ID = 00000019 C-REC = 8290001837
TRACK-ID = 00000018 C-REC = 8290001838
ERROR-COND = 00100000
TRACK-ID = 00000019 C-REC = 8290001838

```



Figure 32. Creating a Direct File with Relative Track Addressing Using Method B (Part 4 of 4)

Randomizing for the 2321 Data Cell

The track reference field for the 2321 Data Cell is composed of the following discontinuous binary address:

sub									
cell		cell		strip		cyl. head		record	
M	B	B	C	C	H	H	R		
0-9		0-19		0-9		0-4		0-19	

At first glance, this presents an almost impossible randomizing task; but since each strip includes 100 tracks that are accessible through cylinder and head number, the 2321 Data Cell can be considered to consist of consecutively numbered tracks.

Tracks	Strip
0 → 99	0
100 ← 199	1
900 ← 999	9
1000 ← 1099	10
1900 ← 1999	19
10000 ← 10099	100
19900 ← 19999	199
199900 ← 199999	1999

It can be seen that relative track 20 is located on cylinder 1, head 0 of some particular strip. Its address can be calculated by dividing by 20.

1 = cylinder number

$$20 \overline{) 20}$$

0 = head number

Thus, relative track number 120 will be located on strip 1, cylinder 1, head 0 of some subcell. Note that the strip number is given by the hundreds digit, and the cylinder and head number are derived by dividing the two low-order digits by 20.

The same relationship holds true for relative track number 900. It is located on strip 9, cylinder 0, track 0. Again, the hundreds digit gives the strip number, and dividing the two low-order digits by 20

results in a quotient and remainder of zero.

This relationship holds true through a relative track number of 19999, which is the number of tracks that can be contained on one cell of a data cell array. By applying the foregoing rules, an address of subcell 19, strip 9, cylinder 4, head 19 is derived.

Thus, by randomizing to a 5-digit decimal track number, the programmer will be able to access the 20,000 tracks (40,000,000 characters) contained in a cell.

The thousands digits would represent the subcell number, the hundreds digit the strip number, and the quotient and remainder of the two low-order digits divided by 20 would represent the cylinder and head number. Each one of these resulting decimal digits would then be converted to binary and placed in the appropriate location in the track reference field.

There is a total of 200,000 tracks per data cell array. To derive valid addresses that cross cell boundaries, the programmer should randomize to a 6-digit decimal track address. The highest address possible should be 199,999. To convert this to a data cell address, similar rules apply. In this case, the programmer must divide the three high-order digits by 20:

9 = cell

$$20 \overline{) 199}$$

180

19 = subcell

The quotient becomes the cell number and the remainder becomes the subcell number. The hundreds digit is still the strip number, and the cylinder and head number can be derived as previously illustrated. The resulting address is 0091994190 and would appear in the first eight bytes of the actual key field as follows:

sub							
cell		cell		strip		cyl. head	
M	B	B	C	C	H	H	R
0	0	9	19	9	4	19	0

Randomizing to the data cell can be accomplished by developing an algorithm to generate decimal track addresses. The use of the foregoing rules makes it possible to



convert these generated track addresses to the appropriate discontinuous binary address.

PRIME AREA

When the file is first created, or when it is subsequently reorganized, records are written in the prime area. Until the prime area is full, additions to the file may also be written there. The prime area may span multiple volumes. Note that the last track of the prime area may not be used by the COBOL programmer.

INDEXED ORGANIZATION (DTFIS)

An indexed file is a sequential file with indexes that permit rapid access to individual records as well as rapid sequential processing. Error recovery from a DTFIS file is described in detail in the chapter "Advanced Processing Capabilities." An indexed file has three distinct areas: a prime area, indexes, and an overflow area. Each area is described in detail below.

The records in the prime area must be formatted with keys, and must be positioned in key sequence. The records may be blocked or unblocked. If records are blocked, each logical record within the block contains its key, and the key area for the block contains the key of the highest record in the block. The Disk Operating System Virtual Storage permits fixed-length records only. Figure 33 shows the formats of blocked and unblocked records on a track.

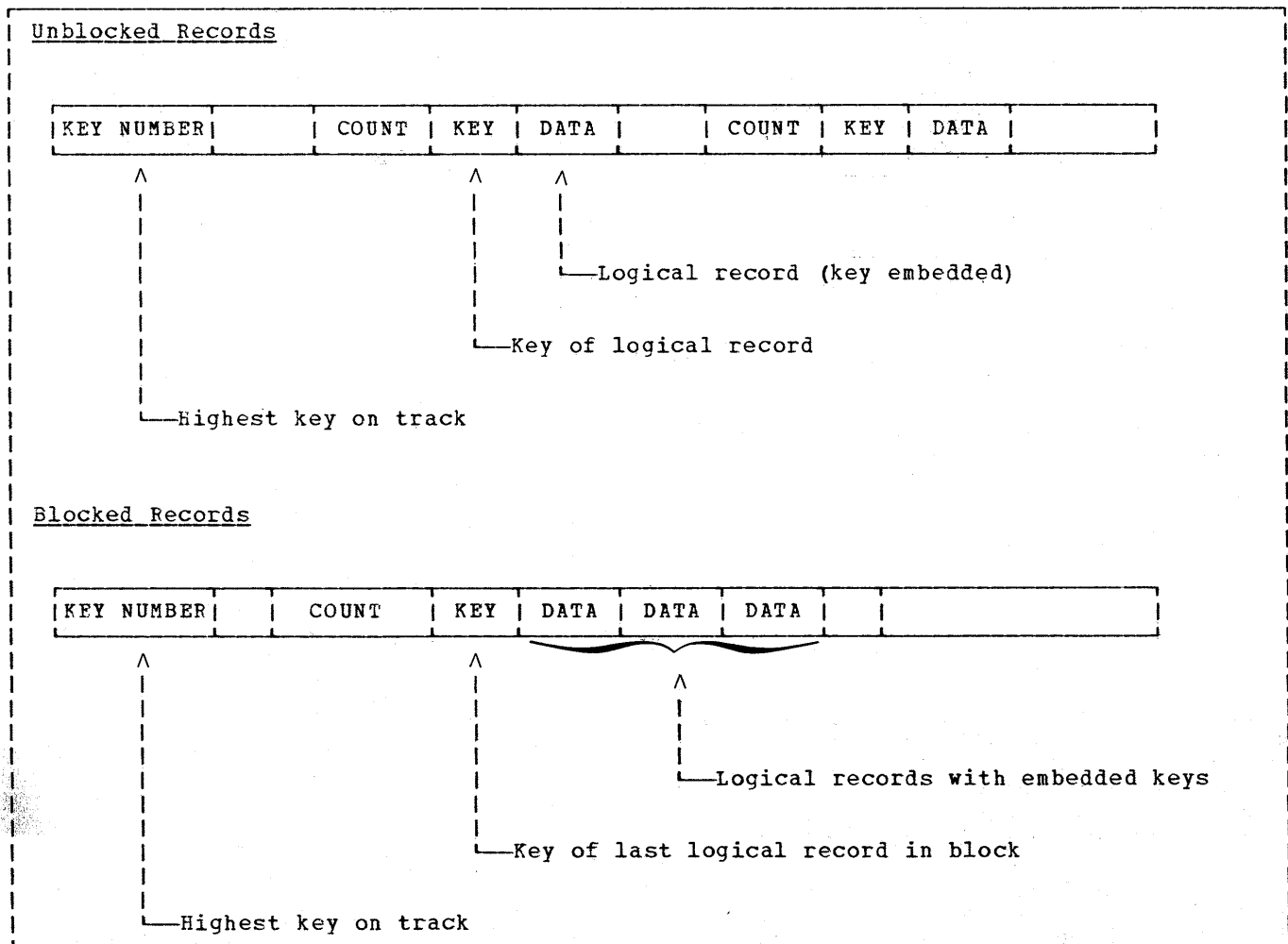


Figure 33. Formats of Blocked and Unblocked Records

INDEXES

There are three possible levels of indexes for a file with indexed organization: a track index, a cylinder index, and a master index. They are created and written by the system when the file is created or reorganized.

Track Index

This is the lowest level of index and is always present. There is one track index for each cylinder in the prime area. It is always written on the first track of the cylinder that it indexes.

The track index contains a pair of entries for each prime data track in the cylinder: a normal entry and an overflow entry. The normal entry contains the home address of the prime track and the key of the highest record on the track. The overflow entry contains the highest key associated with that track and the address of the lowest record in the overflow area. If no overflow entry has yet been made, the address of the lowest record in the overflow area is the dummy entry X'FF'.

Cylinder Index

The cylinder index is a higher level of index and is always present. Its entries point to track indexes. There is one cylinder index for the file. It is written on the device specified in the APPLY CYL-INDEX clause. If this clause is not specified, the cylinder index is written on the same device as the prime area.

Master Index

The master index is the highest level index and is optional. It is used when the cylinder index is so long that searching it is very time consuming. It is suggested that a master index be requested when the cylinder index occupies more than four tracks. (A master index consists of one entry for each track of the cylinder index.)

The DOS/VS System permits one level of master index for the file and requires that it be written immediately before the cylinder index. If a master index is desired, the APPLY MASTER-INDEX clause must

be specified in the source program. When this clause is specified, the cylinder index is placed on the same device as the master index.

Note: The indexes are terminated by a dummy entry containing a key composed of all ones (bits). To avoid any possibility of errors, the user should not specify a key of all ones (HIGH VALUES) for any of his records.

OVERFLOW AREA

There are two types of overflow areas: a cylinder overflow area and an independent overflow area. Either or both may be specified for an indexed file. Records are written in the overflow area(s) as additions are made to the file.

Cylinder Overflow Area

A certain number of whole tracks are reserved in each cylinder for overflow records from the prime tracks in that cylinder. The programmer may specify the number of tracks to be reserved by means of the APPLY CYL-OVERFLOW clause. If he specifies 0 as the number of tracks in this clause, no cylinder overflow area is reserved. If the clause is omitted, 20% of each cylinder is reserved for overflow. For the 3330, three tracks of each cylinder will be reserved for overflow. For the 3340, two tracks of each cylinder will be reserved for overflow.

Independent Overflow Area

Overflow records from anywhere in the prime area are placed in a certain number of cylinders reserved solely for this purpose. The size and location of the independent overflow area can be specified if the programmer includes the proper job control EXTENT cards. The area must, however, be on the same mass storage device type as the prime area.

A suggested approach is to have cylinder overflow areas large enough to contain the average number of overflow records caused by additions and an independent overflow area to be used as the cylinder overflow areas are filled.

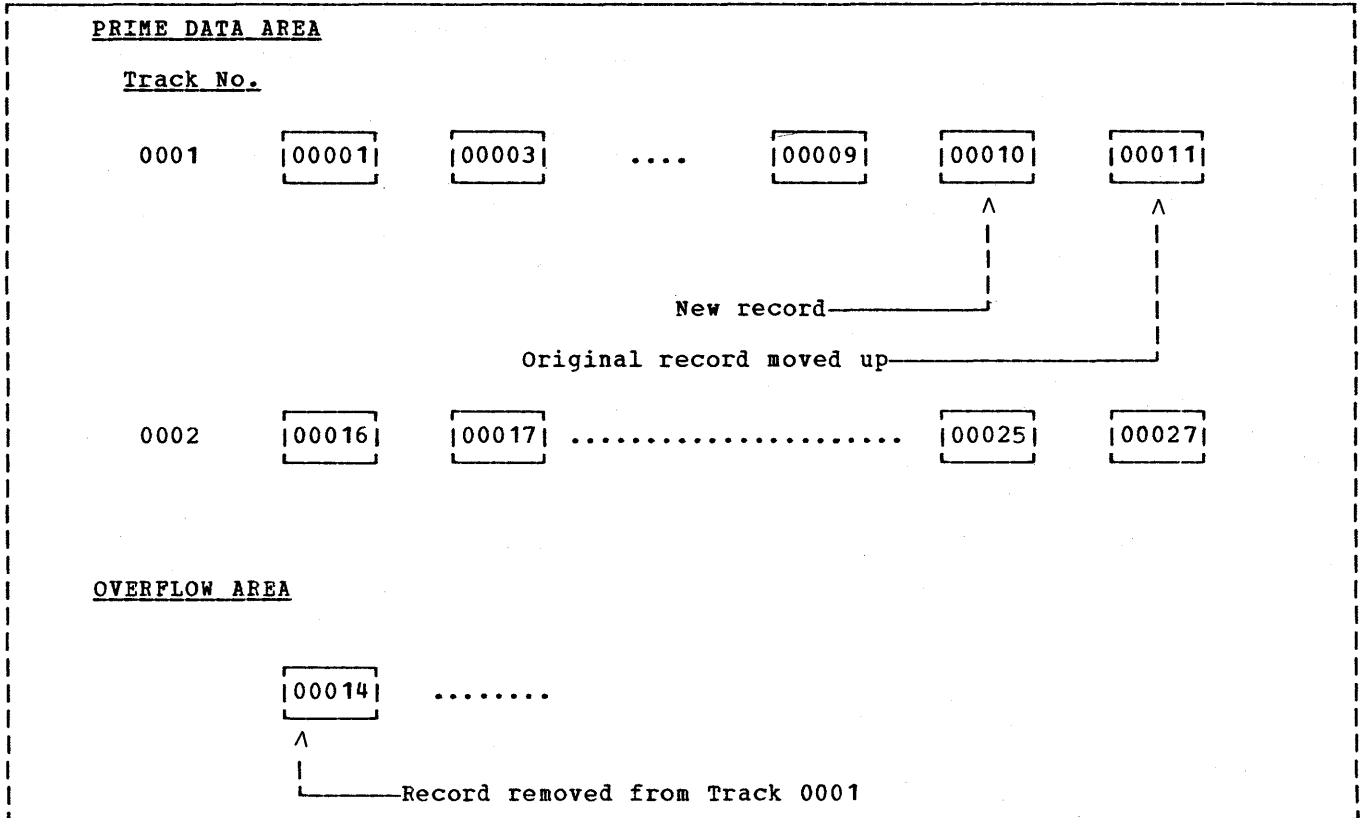


Figure 34. Adding a Record to a Prime Track

Adding Records to an Indexed File

A new record added to an indexed file is placed into a location on a track in the prime area determined by the value of its key field. If records in the file were placed in precise physical sequence, the addition of a new record would require the shifting of all records with keys higher than that of the one inserted. However, indexed organization allows a record to be inserted into its proper position on a track, with the shifting of only the records on that track. Any records for which there is no space on that track are then placed in an overflow area, and become overflow records. Overflow records are always fixed-length, unblocked records, formatted with keys.

As records are added to the overflow area, they are no longer in key sequence. The system ensures, however, that they are always in logical sequence.

Figure 34 illustrates the addition of a record to a prime track.

The new record (00010) is written in its proper sequential location on the prime track. The rest of its prime records are

moved up one location. The bumped record (00014) is written in the first available location in the overflow area. The record is placed in the cylinder overflow area for that cylinder, if a cylinder overflow area exists and if there is space in it; otherwise, the record is placed in the independent overflow area. The first addition to a track is always handled in this manner. Any record that is higher than the original highest record on the preceding track, but lower than the original highest record on this track, is written on the prime track. Record 00015, for example, would be written as the first record on track 0002, and record 00027 would be bumped into the overflow area.

Subsequent additions are written either on the prime track where they belong or as part of the overflow chain from that track. If the addition belongs between the last prime record on a track and a previous overflow from that track (as is the case with record 00013), it is written in the first available location in the overflow area on an empty track, or on a track whose first record has a numerically lower key.



If the addition belongs on a prime track (as would be the case with record 00005), it is written in its proper sequential location on the prime track. The bumped record (record 00011) is written in the overflow area.

A record with a key higher than the current highest key in the file is placed on the last prime track containing data records. If that track is full, the record is placed in the overflow area.

ACCESSING AN INDEXED FILE (DTFIS)

An indexed file may be accessed both sequentially and randomly.

ACCESSING AN INDEXED FILE SEQUENTIALLY: An indexed file may only be created sequentially. It can also be read and updated in the sequential access mode. The following specifications may be made in the source program.

ENVIRONMENT DIVISION

Required clauses:

SELECT [OPTIONAL] file-name

ASSIGN TO SYSnnn-DA- $\left. \begin{matrix} (2311) \\ 2314 \\ 2321 \\ 2319 \\ (3330) \\ (3340) \end{matrix} \right\} - I$

RECORD KEY Clause
NOMINAL KEY Clause (when reading, if the START statement is used)

Optional clauses:

FILE-LIMIT Clause
ACCESS MODE IS SEQUENTIAL
PROCESSING MODE IS SEQUENTIAL
RERUN Clause
SAME Clause
APPLY WRITE-VERIFY Clause (create and update)
APPLY CYL-OVERFLOW Clause (create)

APPLY $\left\{ \begin{matrix} \text{MASTER-INDEX} \\ \text{CYL-INDEX} \end{matrix} \right\}$ Clause

RESERVE Clause

Invalid clauses:

ACCESS MODE IS RANDOM
ACTUAL KEY Clause
TRACK-AREA Clause
MULTIPLE FILE TAPE Clause
APPLY WRITE-ONLY Clause
APPLY EXTENDED-SEARCH Clause
APPLY CORE-INDEX Clause

ACCESSING AN INDEXED FILE RANDOMLY: A randomly-accessed indexed file may be read, updated, or added to. The following specifications may be made in the source program:

ENVIRONMENT DIVISION

Required clauses:

SELECT [OPTIONAL] file-name

ASSIGN TO SYSnnn-DA- $\left. \begin{matrix} (2311) \\ 2314 \\ 2321 \\ 2319 \\ (3330) \\ (3340) \end{matrix} \right\} - I$

ACCESS IS RANDOM
NOMINAL KEY Clause
RECORD KEY Clause

Optional clauses:

FILE LIMIT Clause
PROCESSING MODE IS SEQUENTIAL
TRACK-AREA Clause
RERUN Clause
SAME Clause
APPLY WRITE VERIFY Clause
APPLY CYL-OVERFLOW Clause
APPLY CORE-INDEX Clause

APPLY $\left\{ \begin{matrix} \text{MASTER-INDEX} \\ \text{CYL-INDEX} \end{matrix} \right\}$ Clause

Invalid clauses:

RESERVE Clause
ACCESS MODE IS SEQUENTIAL
ACTUAL KEY Clause
MULTIPLE FILE TAPE Clause
APPLY EXTENDED-SEARCH Clause

Key Clauses

When creating an indexed file, the only key clause required is the RECORD KEY clause. The data-name specified in this clause is the name of the field within the record that contains the key. Keys must be in ascending numerical order when creating an indexed file.

If a START statement is used when retrieving an indexed file sequentially, the NOMINAL KEY clause is required.

When accessing an indexed file randomly, both the NOMINAL KEY and RECORD KEY clauses are required. When reading the file, the data-name specified in the NOMINAL KEY clause is the key of the record which is being retrieved. The data-name specified in the RECORD KEY clause is the name of the field within the record that contains this key.

When adding records to an indexed file, the data-name specified in the NOMINAL KEY clause is the key for the record being written and is used to determine its physical location. The data-name specified in the RECORD KEY clause specifies the field in the record that contains the key.

Note: If an INVALID KEY exit is taken on a START statement, the key value in the NOMINAL KEY data-name should be corrected

and another START statement issued to ensure correct retrieval of blocked records.

Improving Efficiency

When processing an indexed file, the following source language Environment Division clauses may be used to improve efficiency:

TRACK-AREA Clause
APPLY CORE-INDEX Clause

For additional details, see the publication IBM DOS Full American National Standard COFOL.

The DOS/VS Compiler supports 3540 Diskette unit file management. This device is quite different from standard direct access devices as it does not access data randomly. The medium used for reading and writing is a diskette which can be easily mailed from one location to another.

Data can be recorded on the 3540 diskette in two ways:

1. Key punching on the diskette via the 3740 processing device.
2. Writing sequential data sets on the diskette via the 3540 Diskette unit attached to a System/370.

DOS/VS COBOL processing applies only to the processing of data on the diskette by the 3540 Diskette unit.

For the use of system files on diskette, see DOS/VS System Management Guide.

FILE PROCESSING

File processing for the 3540 is sequential only. Only fixed-length physical records can reside on the diskette. Logical blocking of records is an available function and will be discussed in the section entitled "Cobol Language Considerations."

The system interfaces with the COBOL object module through DTFDU, (generated as part of the object module), and DUMOD logic modules (used to perform actual I-O processing). The generated DTFDU will correspond to a DTFDU generated by the DTFDU macro (described in DOS/VS Supervisor and I-O Macros) with the exceptions specified later in this section.

The physical considerations of the 3540 diskette include:

- The diskette is divided into character sectors with each sector containing 128 characters.
- Each record may occupy no more than one sector, and may be from 1 to 128 characters long.
- Each record in a file must be the same size.

- Blocking factors can be only 1, 2, 13, or 26 records.

Files may be extended to additional diskettes if one diskette is too small. This is done automatically by LIOCS if DLBL and EXTENT cards are provided for additional processing. There is no user program control to force end of volume for this device.

File labels exist on the 3540 Diskette for each file, but no user control or processing of these labels is provided by the DOS/VS system. Label management will be handled strictly by LIOCS. The user will only have to provide the name for the file in the DLBL control card.

COBOL LANGUAGE CONSIDERATIONS

ENVIRONMENT DIVISION

The following format of the SFLECT statement applies to the 3540:

Required clauses:

```
SELECT [OPTIONAL] file-name
ASSIGN TO SYSnnn-UT3540-S[-name]
                (DA)
```

Sort work files may not be assigned to the 3540. A 3540 may not be a checkpoint device.

Optional clauses:

```
RESERVE clause
ACCESS MODE IS SEQUENTIAL clause
PROCESSING MODE IS SFQUENTIAL clause
RERUN ON system-name EVERY integer
RECORDS OF file-name
(System-name cannot specify 3540;
file-name can refer to 3540 file;
checkpoint records cannot be taken on
a diskette, but a diskette can be used
to control when checkpoints are
taken.)
SAME clause
FILE LIMIT clause
```

Invalid Clauses:

```
APPLY WRITE-ONLY clause (only
fixed-length records allowed)
APPLY WRITE-VERIFY clause (function not
supported)
ACCESS MODE IS RANDOM clause
ACTUAL KEY clause
```



NOMINAL KEY clause
 RECORD KEY clause
 TRACK-AREA clause
 MULTIPLE FILE TAPE clause
 RERUN clause (see restrictions above)
 APPLY EXTENDED-SEARCH clause
 APPLY CYL-OVERFLOW Clause

APPLY { MASTER-INDEX } clause
 { CYL-INDEX }

APPLY CORE-INDEX clause

DATA DIVISION

The following restrictions apply to the FD and record description for a 3540 file:

- Recording mode must be F.
- Label records must be standard.
- RECORD CONTAINS clause cannot specify more than 128 characters, or "integer-1 to integer-2" CHARACTERS.
- The BLOCK CONTAINS clause must specify the RECORDS option only. Blocking is permitted for the most efficient usage of the 3540. If this clause is specified, only 1, 2, 13, or 26, will be accepted as the blocking factor. Any other number will cause a diagnostic.
- In the record description, a maximum of 128 characters will be allowed for a 3540 file.
- The record description for a 3540 file must not include any items with the OCCURS DEPENDING ON clause, as variable records are not allowed.

Procedure Division -- Special Considerations

- OPEN Statement. 3540 files may be opened for input or output only. Since updating is not permitted for a 3540 file, OPEN I-O is not allowed.
- Only one 3540 file per diskette may be open simultaneously.
- The REVERSED and NO REWIND options of the OPEN statement are not valid for a 3540 file.
- WRITE Statement. The INVALID KEY option may not be used for a 3540 file. If the end of the diskette is reached and additional diskette information has not been supplied via additional EXTENT control cards, the operator will be

queried to either supply an EXTENT through the console or cancel the job.

- Standard errors can be handled in a USE AFTER STANDARD ERROR Declarative. Two types of errors will cause control to return to an error declarative for 3540 files:

1. Data check
2. Equipment check

If the GIVING option is specified, byte 1 will indicate a data check, and byte 2 will indicate an equipment check.

In either case, the error procedure is used to continue processing or to close the file. If processing continues and the file is blocked, the remaining records in the block after the record causing the error may be lost when the next READ or WRITE statement is executed.

If no error declarative is specified, a message will be issued describing the type of error, and the job will be canceled.

CLOSE Statement. When a CLOSE statement is executed for a 3540 file, the present diskette will be fed out into the output hopper. CLOSE UNIT may not be used as no forced end-of-volume support is included for the 3540 Diskette unit. CLOSE NO REWIND may not be used. The LOCK option will be supported for 3540 files.

DTFDU

The compiler will generate DTFDU with the following defaults:

1. No write protection
2. Feed = yes
3. Volume sequencing will be checked.
4. No read/write security.

Job Control Requirements

Normal job control DLBL and EXTENT statements for the 3540 are shown below.

DLBL Statement

The format of the DLBL statement is:

```
// DLBL filename,['file-ID'],[date],[code]
```

filename -- is a unique filename of 3 to 7 characters identical to the symbolic name of the DTF that identifies the file. Supported in the same way as for current devices. This corresponds to the "name" field of system-name in the SELECT statement if specified, or to SYSnnn in the system-name.

'file-ID' -- only the first 8 characters will be used. Supported in the same manner as for current devices.

date -- provides the expiration date for the file. Supported in the same way as for current devices.

code -- is a field indicating the type of file label. DU for diskette unit is supported. It is supported in the same way as for current devices.

EXTENT Statement

The format of the EXTENT statement is:

```
// EXTENT [symbolic-unit],  
         [serial-number],[1]
```

symbolic unit -- indicates the symbolic unit (SYSxxx) of the volume for which the extent is effective. It is supported in the same way as for current devices.

serial number -- indicates the volume serial number of the volume for which this extent is effective. It is supported in the same way as for other devices. The serial number is optional. If omitted, the volume that is mounted is assumed to be the correct volume.

type -- indicates the type of extent. A '1' indicates 'data area.' No other types are supported.

3540 File

The following DLBL and EXTENT statements describe a file that resides on a 3540 diskette.

```
// DLBL MASTER,,75/001,DU  
// EXTENT SYS015,111111,1
```

In the following example, the program CREATES creates a diskette (DU) file named SALES that is to be retained until the end of 1975. The file comprises up to three diskettes. The diskettes have the volume serial numbers 111111, 111112, and 111113, and are mounted on the drive assigned to the symbolic device name SYS005.

```
// JOB EXAMPLE  
// ASSGN SYS005,X'060'  
// DLBL SALES,'ANNUAL',75/365,DU  
// EXTENT SYS005,111111,1  
// EXTENT SYS005,111112,1  
// EXTENT SYS005,111113,1  
// EXEC CREATE  
/8
```

The COBOL statements which correspond to this are:

```
SELECT SALES-FILE ASSIGN  
      TO SYS005-DA-3540-S-SALES.  
.  
.  
PD SALES-FILE  
RECORDING MODE IS F  
LABEL RECORDS ARE STANDARD  
RECORD CONTAINS 80 CHARACTERS.  
01 DISKETTE-RECORD.  
02  
.  
.  
.
```




VSAM is a new access method for direct or sequential processing of fixed and variable length records on direct-access devices. It has more functions, generally better performance, better data integrity and security, improved data organization, and is easier to use and control than the DOS/VS DAM and ISAM access methods.

VSAM files can be processed only by the VSAM file processing technique. The programmer can convert SAM and ISAM files to VSAM files by using the method described in the section entitled "Converting Non-VSAM Files to VSAM Files." The following topics related to VSAM are discussed in this chapter:

- VSAM File processing
- Access Method Services
- Error Handling

File Organization

The records in a VSAM file can be organized either in logical sequence by a key field (key-sequence) or in the physical sequence in which they are written on the file (entry-sequence). The user can read, add, delete, and modify records in a VSAM file. He can access the records sequentially or directly by key.

A key-sequenced file has an index, like ISAM; the records in a key-sequenced file can be accessed by key. An entry-sequenced file does not have an index, and records can be accessed like a sequential file.

Key-Sequenced Files

Like indexed sequential files, key-sequenced files are ordered according to a user-defined key field in each record. That is, they are ordered according to the collating sequence of the key field in each record. Each record has a unique value in the key field, such as employee number or invoice number. VSAM uses the key associated with each record to insert a new record in the file or to retrieve a record from the file. The order of access can be random or sequential. Key-sequenced files, however, can generally be processed faster than ISAM files because VSAM has a more efficient index and does not use chained record overflow.

When a key-sequenced file is created, certain portions can be left empty, that is, free space can be distributed throughout the file. When a record is inserted or when an existing record is lengthened, the free space at or near the existing records closest in key sequence is used. This eliminates the need for overflow chains and overflow areas; it also minimizes data movement. Thus performance does not degrade substantially as records are added and the file does not have to be reorganized as often as an ISAM file. VSAM reclaims space when a record is deleted or shortened, and the space released becomes free space.

The index of a key-sequenced VSAM file is more efficient than an ISAM index because it generally requires less direct-access space and less updating of index entries. Space is saved by eliminating redundant key information in the index entries (key compression) and by blocking index records. A shorter index requires less time to search and update. Updating is infrequent, however, because index entries are not usually modified when records are added to the file.

A key-sequenced file is defined in COBOL by specifying:

```
SELECT file-name ASSIGN TO
  SYSnnn[-class][[-device][[-name]
  ORGANIZATION IS INDEXED....
  RECORD KEY IS...
```

Several index options are available to speed VSAM processing:

- The index and the data can be on different devices.
- The lowest level of index records can be written adjacent to the data to reduce disk-arm movement, and several copies of an index record can be written on a track to reduce rotational delay.

Entry-Sequenced Files

Records are stored in entry-sequenced files in the order they are presented for inclusion on the file (that is, their entry-sequence), and without respect to the contents of the records. No keys are recognized and, consequently, no indexes



are built. The order of records is fixed; they are not moved. Thus, free space is not distributed throughout the file and new records are placed at the end. Records cannot be shortened, deleted, or lengthened. Since there is no index, the user must access the file sequentially (in the order the records were written).

An entry-sequenced file is defined in COBOL by specifying:

```
SELECT file-name ASSIGN TO
  SYSnnn[-class][-device]-AS[-name]
  ORGANIZATION IS SEQUENTIAL....
```

Data Organization

The data organization of ISAM is based on the physical units of disk cylinder and disk track, while the data organization of VSAM is based on logical units called control intervals and control areas. A control interval is the unit of direct-access storage that is transferred to and from virtual storage. It can contain one or more records in one or more blocks. Each entry in the lowest index level of a key-sequenced VSAM file points to a control interval. Free space in a key-sequenced file is distributed in terms of control intervals. A percentage of each control interval can be free space and some control intervals can be entirely free space. Indexes are also organized in control intervals. Each contains a single index record which can have many index entries. A control area is a group of control intervals. The number of control intervals of data in a control area equals the number of index entries in each index record. VSAM data organization provides for device independence by reducing the programmer's concern about the physical characteristics of the data and the index. Figure 35 illustrates VSAM data organization.

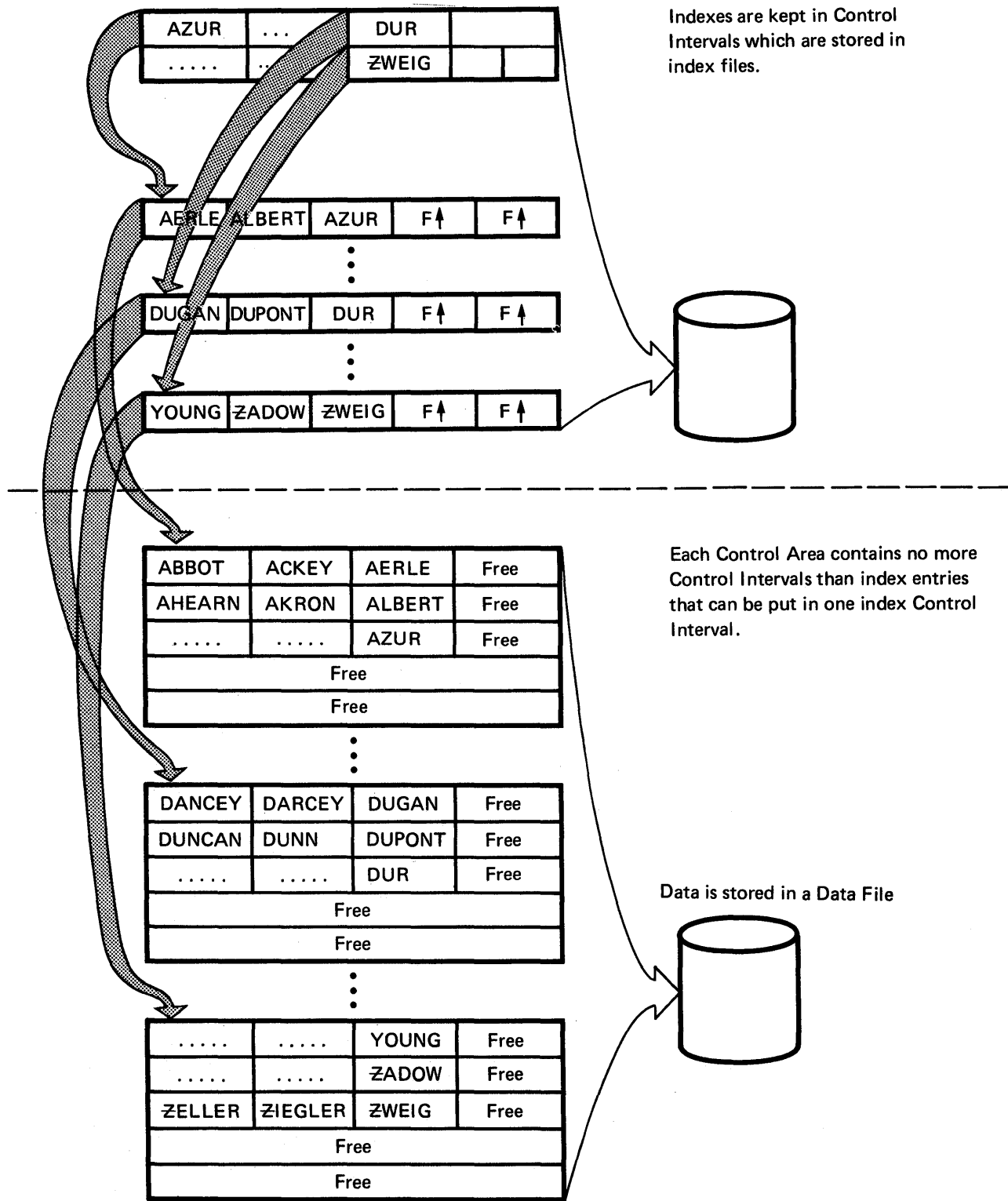
Data Access

As with ISAM, VSAM records can be accessed either sequentially or directly. Key-sequenced files can be accessed by key, entry-sequenced files can be accessed sequentially only. The key can be that of an individual record or it can be a generic key (the front part of a key field) which specifies the keys of a group of individual records.

VSAM allows retrieval, storage, update, and deletion of records in COBOL, via the READ, WRITE, REWRITE, and DELETE verbs. Also, the user can access several records in key-sequence and then have VSAM skip to another portion of the file and access more records in sequence without having to search the entire file to find the new group of records through the COBOL START verb.

VSAM Catalog

VSAM keeps central control over the creation, access, and deletion of files and over the management of direct-access storage space allocated to those files. This is done by keeping information on file and space characteristics in one place, the VSAM catalog. The catalog, which is unique to VSAM, makes it easier to (1) keep track of both files and available direct-access space, (2) write job control statements to create and process VSAM files, and (3) move VSAM files to other DOS/VS systems or to OS/VS systems. Once the user has allocated a volume or portion of a volume to VSAM, each file he creates is automatically sub-allocated space on that volume by VSAM. There can be more than one VSAM catalog. However, only one catalog at a time can be connected to the system. Each catalog can keep track of VSAM files on many volumes; it is not necessary to mount a volume to determine whether or not it has space available for a VSAM file.



Note: The length of the key and the percentage of free space to be distributed is determined by the user. The control area and the physical mapping are automatically determined by the system.

Figure 35. VSAM Data Organization

File and Volume Portability

A significant feature of VSAM files is that files can be moved from one DOS/VS system to another or to an OS/VS system. This is possible because VSAM data format and accessing techniques are identical under both DOS/VS and OS/VS.

Service Programs

VSAM has an extensive service program package, called Access Method Services, which can be used to:

- Define (create), print, copy, or reorganize VSAM files.
- Add, alter, delete, or print catalog entries.
- Convert ISAM and SAM files to VSAM files.
- Copy a VSAM file and catalog entries in a format required for transporting to an OS/VS system.

Device Support

VSAM files can be written on the 2314, 3330, and 3340 devices.

Security

Through COBOL, access to the file can be restricted by use of the PASSWORD clause in the SELECT statement.

Error Processing

VSAM provides exits to a user-supplied routine to handle I/O and/or logical errors or exception conditions. This is done in COBOL via the USE AFTER STANDARD ERROR declarative and the INVALID KEY and AT END clauses. A STATUS KEY may be specified, and the details of the condition determined.

VSAM Messages

Like other access methods, VSAM issues messages to the operator, if for example, the incorrect volume is mounted, etc. These messages are described in DOS/VS Messages. VSAM Access Method Services also issues messages to the programmer which are documented in DOS/VS Access Method Services. COBOL issues VSAM messages to the operator and/or programmer. These are listed in "Appendix I: Diagnostic Messages."

For more detail on VSAM, refer to DOS/VS Access Method Services.

ACCESS METHOD SERVICES

Access Method Services, a utility program, must be used before any COBOL program can process a VSAM file. A number of user-entered commands, either modal or functional, initiate the Access Method Services programs. The functional commands invoke the desired Access Method Services function while the modal commands control the sequence of execution of the functional commands. In this chapter, only certain commands and parameters are discussed. For complete details on the use of commands see DOS/VS Utilities Access Method Services.

Functional Commands

There are nine functional commands: DEFINE, ALTER, DELETE, LISTCAT, REPRO, PRINT, IMPORT, EXPORT, and VERIFY. The commands DEFINE, ALTER, DELETE, and LISTCAT are used to create, modify, remove, and list entries in the VSAM catalog. The REPRO and PRINT commands reproduce files either as new files or as printed output. The IMPORT and EXPORT commands provide for transfers of files from one system to another. The VERIFY command provides a file recovery service for VSAM files by ensuring that the end of the file indicated in the catalog is the same as the actual file end.

The DEFINE Command

The DEFINE command must be used to define the master catalog, data space, and VSAM files respectively. In VSAM terminology, an object is either a VSAM catalog, a VSAM data space, or a VSAM file.

Files must be introduced to the system and defined as entries in the master catalog. Non-VSAM files may also be cataloged in a VSAM catalog. All VSAM files are introduced to the system through the DEFINE command.

There are two steps in the creation of objects: defining the object to the system, and generating its contents. The DEFINE command generates entries into the master catalog rather than the contents of an object. The process of defining a VSAM object includes allocation of storage space, initialization of catalog-entry contents, and, in the case of defining catalogs, possible initialization of the object's storage space.

Specification of the DEFINE Command

Format
DEFINE object parameters

The definable objects are:

- MASTERCATALOG -- specifies that the VSAM master catalog is to be defined.
- SPACE -- specifies that a VSAM data space is to be defined.
- CLUSTER -- specifies that a file is to be defined.

For each file there is an associated valid parameter list.

Defining a VSAM Master Catalog: DEFINE MASTERCATALOG

The DEFINE MASTERCATALOG command must be used to set up the master catalog. It is the first Access Method Services command ever used since without a master catalog other objects cannot be defined. Defining a master catalog automatically creates the data space necessary to contain the catalog. Entries for both the master catalog itself and the volume containing the data space automatically created are placed in the master catalog.

The following is an example of defining a VSAM master catalog and its data space.

```

// JOB      DEFINE A VSAM CATALOG
// DLBL     IJSYSCT,'VSAMCAT',,VSAM
// EXTENT   SYSCAT,321940,1,,100,250
// EXEC     IDCAMS,SIZE=26K
//         DEFINE MASTERCATALOG(NAME(VSAMCAT) +
//         VOLUME (321942) TRACKS (250) +
//         FILE(IJSYSCT) UPDATEPW (SECRET) +
//         READPW(NOSFRET))
/*
/8

```

Figure 36. Defining a VSAM Master Catalog

The DLBL statement must be used to specify the filename and the code which identifies VSAM. The filename must be specified as IJSYSCT.

The logical unit in the EXTENT statement must be SYSCAT. The user must decide which volumes and which extents will contain the catalog. Note that the VOLUMES parameter and the space allocation parameter (CYLINDERS, TRACKS, or RECORDS) must be included in the DEFINE command, and must agree with the information in the EXTENT statement. If the CYLINDERS parameter is used, each extent must begin on a cylinder boundary.



The following parameters were used in the above example:

NAME (object name)
This is a required parameter. The object name is cataloged and is the name that must be used in all future references to the master catalog. The object name must follow the naming convention rules discussed in DOS/VS Utilities Access Method Services.

VOLUME (volser ...)
This is a required parameter that specifies the volume to contain the master catalog. The volume serial number containing the master catalog is substituted for volser.

TRACKS (primary[secondary])
This required parameter specifies the quantity of space required by the object being defined. The capacity of the allocated space is device-dependent. The size of the primary extent is specified by primary. Once the primary extent is filled, the data space can expand to include a maximum of 15 secondary extents if the size of secondary extents is specified through secondary.

Note that every key-sequenced file requires three catalog entries: one each for the cluster, data component, and index component. Every entry-sequenced file requires two

catalog entries: one for the cluster and one for the data component.

FILE (dname)

This is a required parameter identifying the filename of a DLBL statement that specifies the device and volume for allocation. The filename must be specified as IJSYSCAT.

UPDATEPW (password)

This is an optional parameter. However, if any file which is catalogued in the VSAM catalog is to be password protected, the catalog itself must also be password protected.

This password must be specified when changing the contents of the catalog (DEFINE, ALTER, DELETE).

READPW (password)

This is an optional parameter. If specified, all access to the catalog requires this password.

Defining a VSAM Data Space: DEFINE SPACE

The DEFINE SPACE command defines VSAM data spaces or reserves volumes for future use by VSAM. A VSAM data space is space on a direct access volume that is owned and managed by VSAM. Clusters cannot be defined without the UNIQUE attribute unless a VSAM data space is defined to contain them. A VSAM data space may include several extents on the same volume but it cannot span volumes. The volume containing the data space is owned by the catalog containing the entry for the space. Data spaces on several volumes can be defined at one time. When a data space is defined, DLBL and EXTENT statements are required.

Figure 37 is an example of defining a VSAM data space:

```
||// JOB      DEFINE A VSAM DATA SPACE  ||
||// ASSGN   SYS001,X'130'              ||
||// DLBL    VFILENM,,,VSAM             ||
||// EXTENT  SYS001,321942,1,,800,400    ||
||// EXEC    IDCAMS,SIZE=26K            ||
||  DEFINE   SPACE (FILE(VFILENM)      +  ||
||           TRACKS (400)              +  ||
||           VOLUMES (321942))         +  ||
||           CATALOG (VSAMCAT/SECRET)  ||
||/*                                           ||
||/&
```

Figure 37. Defining a VSAM Data Space

The DLBL statement must be used to specify the filename and the code which identifies VSAM files. The filename (VFILENM) is the same as the FILE parameter and connects the job control statements to the DEFINE command. The EXTENT statement must be used to specify the symbolic unit name, the volume serial number, and the space parameters. The VOLUMES parameter and the space allocation parameter (CYLINDERS, TRACKS, or RECORDS) must be included in the DEFINE command, and must agree with the information in the EXTENT statements. If the CYLINDERS parameter is used, each extent must begin on a cylinder boundary.

The following parameters were used in Figure 37.

FILE

This required parameter identifies the filename of a Job Control statement that specifies the devices and volumes to be used for space allocation.

TRACKS (primary[secondary])

This parameter specifies the amount of space to be allocated in terms of tracks. The capacity of the allocated space is device-dependent. Either this parameter or the CYLINDERS or RECORD parameter must be specified. The size of the primary extent is specified by primary. Once the primary extent is filled, the data space can expand to include a maximum of 15 secondary extents if the size of secondary extents is specified through secondary. The number used to specify the tracks to be allocated to the data space must agree with the information in the EXTENT statements.

VOLUMES (volser ...)

This required parameter specifies the volumes to contain the data spaces. If more than one volume is specified, each volume will contain a data space of the same size. Note that the VOLUMES parameter must agree with the information in the EXTENT statements. The volume serial number of the volume(s) containing the data space(s) is substituted for volser.

CATALOG (catalogname/updatepw)

This is a required parameter. It specifies the name of the catalog which is to own the space, and the update password for that catalog.

Defining a VSAM File: DEFINE CLUSTER

DEFINE CLUSTER is used to define all attributes of all VSAM files and to catalog the files in a VSAM catalog.

Note: This command cannot be used to add records to the VSAM file.

VSAM files can be sub-allocated or unique. A sub-allocated file is one which is defined using space from one or more existing data spaces. For such a file, DLBL and EXTENT statements are not required. Label processing is not performed since information needed to set up the file is in the DEFINE command, and information about the data spaces to be used for the file is in the VSAM catalog.

A unique VSAM file is one which is defined at the same time as the data spaces which will contain it. The data and the index of a key-sequenced unique file must occupy separate data spaces; each requires DLBL and EXTENT statements. An entry-sequenced file occupies only one data space.

Figure 38 is an example of defining a suballocated key-sequenced file.

```

|// JOB   DEFINE
|// EXEC  IDCAMS,SIZE=26K
|  DEFINE CLUSTER (NAME (MSTRFILE)  +
|          RECORDS (100,10)  +
|          VOLUME (231942)
|          RECORDSIZE (40 55)  +
|          FREESPACE (10 5)
|          SUBALLOCATION  +
|          INDEXED  +
|          KEYS (8 2) UPDATEPW (WRITEFL)  +
|          ATTEMPTS (0)  +
|          CATALOG (VSAMCAT/SECRET)

```

Figure 38. Defining a Key-Sequenced Suballocated VSAM File

The following parameters are used in Figure 38.

- NAME (objectname) -- This parameter is required and specifies the name to be given to the file being defined.
- VOLUME (volser ...) -- This required parameter is used to specify the volume on which the defined object is to be placed. More than one volume can be specified; the volumes actually allocated are listed.
- RECORDS (primary [secondary]) -- This parameter specifies the amount of space to be allocated in terms of the number of records the space is to hold.

- RECORDSIZE (size1 size2) -- This required parameter specifies the length attributes of the logical records in the file. The size specified can be from 1 to 32,761. size1 is the average length of all logical records. size2 is the maximum length of any logical record.

- FREESPACE (percent1 [percent2]) -- This parameter specifies the percentage of space allocation that is to be reserved during initial and subsequent allocations. percent1 specifies the amount of unused space to be left in each control interval. percent2 specifies the amount of unused space to be left in each control area.

Note: This parameter is valid for key-sequenced files only.

- UNIOUE/SUBALLOCATION -- This parameter specifies whether the object is allocated a space of its own, or whether a portion of an already defined VSAM data space is suballocated to the object.

UNIOUE

specifies that the object being defined is allocated a space of its own. An object with the UNIOUE attribute appears in the VTOC of its volume under its own name.

SUBALLOCATION

specifies that a portion of an already defined VSAM data space is allocated to the object. Objects with the SUBALLOCATION attribute do not appear in the VTOC. Only the name of the data space that contains the object appears there. If the object has the SUBALLOCATION attribute, there must be a VSAM data space defined on the volume in which the object is being defined.

- INDEXED/NONINDEXED -- This parameter specifies the type of cluster being defined.

INDEXED

specifies that the cluster being defined is for a key-sequenced file. This is the default.

NONINDEXED

specifies that the cluster being defined is for an entry-sequenced file.

- KEYS (length position) -- This parameter specifies the length and the starting position of the key field within each logical record. (Position 0 is the first byte in the logical record.) The key field with this specified length, and starting in the specified position, is in all logical records in a key-sequenced file. The sum of length and position must be equal to or less than the length of the logical record.
- UPDATEPW (password) -- This parameter specifies an update level password for the file being defined. The update level password permits input and update operations (READ, START, DELETE, WRITE, REWRITE) against the logical records of the file.

Passwords must conform to these rules:

1. Passwords must contain 1 to 8 characters. All EBCDIC characters are allowed.
2. Passwords must be enclosed in single quotation marks if the passwords contain commas, semicolons, blanks, parentheses, slashes, or slash-asterisk combinations.
3. Single quotation marks within passwords must be coded as two single quotation marks if the password is enclosed in single quotation marks.

If you supply the read level password but do not supply the update password, the read password becomes the update password.

ATTEMPTS (count)

specifies the maximum number of times the operator can try to enter the password in response to a prompting message. Count can be any number from 0 through 7. The value 0 prevents any password prompting.

CATALOG (catalog name/password)

specifies the catalog and its update level password that is to contain the entries for the cluster.

File Processing Techniques

The COBOL user has three different file processing techniques available to him; sequential, random, and a combination of sequential and random. The technique to be used is specified through the ACCESS clause of the SELECT statement.

Entry-Sequenced File Processing: An entry-sequenced file can only be processed sequentially; therefore, since the default is sequential, the ACCESS clause need not be specified.

Key-Sequenced File Processing: A key-sequenced file can be processed sequentially, randomly, or both sequentially and randomly. To process sequentially, ACCESS IS SEQUENTIAL is specified. To process randomly, ACCESS IS RANDOM is specified. To process both sequentially and randomly, ACCESS IS DYNAMIC is specified.

ACCESS IS DYNAMIC provides the greatest flexibility since all the capabilities of both sequential and random processing are supported. Within a given OPEN statement, processing can be switched from sequential to random and vice-versa, as many times as desired.

Current Record Pointer

The current record pointer (CRP), a conceptual pointer, is applicable only to key-sequenced files. The current record pointer indicates the next record to be accessed by a sequential request; the CRP has no meaning for random processing. The CRP is affected only by the OPEN, START and READ statements, it is not used or affected by the WRITE, REWRITE, or DELETE statements. The following are examples of how the CRP is affected by various COBOL statements.

Example 1:

Assuming a file has records with keys from 1 to 10, if the sequence of I/O operations on the file with ACCESS IS DYNAMIC and opened I-O is:

```
READ (After setting record key to 10)
WRITE (After setting record key to 44)
READ NEXT
```

the READ NEXT reads record 11 if the previous READ was successful. If the previous READ was not successful, the STATUS KEY will be set to 94 (No Current

Record Pointer) when the READ NEXT is attempted. This occurs independently of the successful intervening WRITE.

Generally, the last request on a file which establishes the CRP (OPEN, READ, or START) must have been successful in order for the sequential read to be successful.

Example 2:

In this example, ACCESS IS SEQUENTIAL is specified; therefore, records are retrieved in ascending key sequence starting at the position indicated by the CRP. (Assume this file has records with keys from 1 to 10.)

```

OPEN INPUT    (CRP at first record in file)
(set record
key to 10)

START        (CRP at record 10)

READ        (Read record 10)
(Set record
key to 5)

START        (CRP at record 5)

READ        (Read record 5; CRP at
record 6)

READ        (Read record 6; CRP at
record 7)

READ        (Read record 7; CRP at
record 8)

```

Note that the CRP can be changed randomly through the use of the START statement but all reading is done sequentially from that point. In this example, if the START request for record key 5 had failed with no record found (23), the three READ statements following would have failed with no current record pointer (94).

Example 3:

In this example ACCESS IS DYNAMIC is specified. Therefore, records are accessed randomly if a READ is specified and sequentially if READ NEXT is specified. (Assume this file has records with keys from 1 to 44.)

```

OPEN INPUT    (CRP is set to first record on
file)
(Set record
key to 5)

READ        (Read record 5; CRP at
record 6)

READ NEXT    (Read record 6; CRP at
record 7)

READ NEXT    (Read record 7; CRP at
record 8)
(Set record
key to 43)

START        (CRP at record 43)

READ NEXT    (Read record 43; CRP at
44)
(Set record
key to 47)

START        (Fails -- no record 47)

READ NEXT    (Fails -- no CRP)

```

The last READ NEXT failed because the preceding START was unsuccessful; in this file there is no record 47.

Example 4:

In this updating example, ACCESS IS DYNAMIC is specified; the REWRITE statement does not affect the CRP. (Assume this file has records with keys from 1 to 44.)

```

OPEN I-O      (CRP is at first record on
file)
(Set record
key to 10)

READ        (Read record 10; CRP at
record 11)
(Set record
key to 44)

REWRITE      (Updates record 44; CRP
remains at record 11)

READ NEXT    (Read record 11; CRP at
record 12)
(Set record
key to 74)

REWRITE      (Fails -- record not found in
this file)

READ NEXT    (Read record 12; CRP at
record 13).

```

Note that although the last REWRITE failed, the following READ NEXT was successful.

Table 11. File Status Values and Error Handling

First Character of FILE STATUS	No USE Declarative		USE Declarative	
	AT END or INVALID KEY clause	No AT END or INVALID KEY clause	AT END or INVALID KEY clause	No AT END or INVALID KEY clause
0	Return to next sentence	Return to next sentence	Return to next sentence	Return to next sentence
1	Return to AT END or INVALID KEY address	Return to next sentence	Return to AT END or INVALID KEY address	Return to next sentence after USE declarative is executed
2	Return to AT END or INVALID KEY address	Return to next sentence	Return to AT END or INVALID KEY address	Return to next sentence after USE declarative is executed
3	Write message and return to next sentence	Write message and return to next sentence	Return to next sentence after USE declarative is executed	Return to next sentence after USE declarative is executed
9	Return to next sentence	Return to next sentence	Return to next sentence after USE declarative is executed	Return to next sentence after USE declarative is executed

ERROR HANDLING

All errors on a VSAM file, whether logic errors caused by the COBOL programmer (for example, reading an unopened file), or I-O errors on the external storage media, return control to the COBOL program. The contents of FILE STATUS indicate the status of the last request on the file. It is strongly recommended that all files have a file status associated with them, and that the COBOL programmer check the contents of FILE STATUS after each request.

Table 11 describes the actions taken for all the combinations of AT END, INVALID KEY, and error declaratives for each value of FILE STATUS.

Note: Return is always to NEXT STATEMENT unless the request that caused the error contained an AT END or INVALID KEY clause. By omitting both the AT END and INVALID KEY clauses and the USE ERROR/EXCEPTION for the file, any type of error for the file can be intercepted by checking the FILE STATUS data name following each I/O request (including OPEN and CLOSE) for the file. This will simplify the exception-condition handling in the COBOL program.

Record Formats for VSAM Files

For VSAM files, processing is independent of whether or not the records on a file are of fixed-length format (that is, all records in the file are the same length) or of variable-length format.

Thus for example, the considerations which are discussed in "Record Formats For Non-VSAM Files" generally do not apply.

The following points should also be considered:

- For record handling purposes, the records are considered to be fixed-length when
 1. All the records in the file are the same size (or there is only one record description).
 2. No record contains an OCCURS clause with the DEPENDING ON option.
 Otherwise, the records are considered to be variable length.
- For variable-length the records, without OCCURS DEPENDING ON clauses, the following applies:

When a READ INTO statement is used, the size of the longest record for the file is moved to the input area. Coding considerations for records with the OCCURS DEPENDING ON option are discussed in "Table Handling Considerations."

executed. This same technique can be used for any request against the file (CLOSE, READ, etc.) to determine if such a request was attempted in his program.

Opening a VSAM File

Initial Loading of Records into a File

An unloaded file is one which has been defined but has never contained any records. An empty file is one which has contained records but from which all records have been deleted.

Initial loading is the process of writing records into an unloaded file.

It is strongly recommended that initial loading of records into a key-sequenced file be done sequentially. If the initial loading is done randomly, performance may be slower, not only for the initial loading process, but also for all processing done on that file later on. Random loading of records does not reserve free space in the file; therefore, the file will be dynamically reorganized when any subsequent records are inserted.

File Status Initialization

The value of 'Z' in Status Key 1 is reserved for the programmer's use. This permits his determining whether a request was made against his file. For example, if he initializes Status Key 1 to the value Z before attempting to OPEN his file he can then determine if his program actually attempted the OPEN by checking the contents of Status Key 1. If it is Z, the OPEN statement was not executed; if it is a value other than Z, the statement was

VSAM files can be opened INPUT, OUTPUT, EXTEND, or I-O. An unloaded file can be opened OUTPUT or EXTEND only. An empty file can be opened EXTEND, INPUT or I-O. The first READ request on an empty file gives an AT END condition (FILE STATUS=10). An empty file or a file with records cannot be opened OUTPUT.

If any of these rules are violated, the file is not opened and the FILE STATUS key is set to the appropriate value. Refer to Table 12 for FILE STATUS key values at open time. Table 13 describes file status at action request time.

A file which contains records can be opened EXTEND, INPUT, or I-O. If such a file is opened EXTEND and it is a key-sequenced file, the first record to be added must have its record key higher than the highest record key on the file when it was opened. If it is not higher, a logic error results, and the FILE STATUS key is equal to 92. For an entry-sequenced file, the records are added after the last record.

Since the USE declarative is executed only for files that are in open status, the only OPEN error which can cause the USE DECLARATIVE to be invoked is trying to open a file which is already in the open status. This is a logic error and causes file status to be set to 92. The open status of the file is not affected. However, if the file is defined as ACCESS IS DYNAMIC, the illegal OPEN statement causes the current record pointer to be undefined.



Table 12. File Status Key Values at OPEN

File Status	Probable Cause
30	I-O error
91	Incorrect password. Either an incorrect password was specified or a required password was not specified. If a file is opened OUTPUT, EXTEND, or I-O, the UPDATE password is required.
92	Logic error caused by opening an opened file, or by opening a locked file.
93	Resource not available. Caused by insufficient virtual storage, or the file is not available for the type of processing requested. ¹
95	Invalid or incomplete information in the ASSGN card, or the file was not found in the catalog. ²
96	Missing DLBL card

¹Indicates that the file was already opened by someone else and opening it for this request would violate the share options specified for the file.

²FILE STATUS 95 can also be caused by the following:

- an attempt to open a key-sequenced file as if it were an entry-sequenced file or vice versa.
- an attempt to open an unloaded file with the INPUT or I-O option.
- an attempt to open OUTPUT a file not in the unloaded state.
- record key length or displacement specification that does not match what was specified when the file was defined.

Table 13. File Status at Action Request Time

File Status	Probable Cause
00	Successful
10	A sequential READ statement encountered EOF.
21	A request was issued to change the record key during execution of a REWRITE statement, or a sequence error occurred for a sequentially-accessed key-sequenced file.
22	
23	
24	A request was issued to add a record whose record key was a duplicate of a record already on the file.
30	key-sequenced file only
34	Either a READ statement was issued for a record whose record key does not match any record on the file, or a REWRITE or DELETE statement was issued for a record not on the file.
92	A request was issued to write a record beyond the externally-defined boundaries of the file.
94	An I-O error occurred.
	A request was issued to write a record beyond the externally-defined boundaries of an entry-sequenced file.
	A logic error occurred. (See Note below.)
	No current record pointer for a sequential READ statement.

Note: File Status = 92 can be caused by the following:

- Any request issued against an unopened file.
- Any request issued which is not allowed for the OPEN option; for example, issuing a READ statement for a file opened OUTPUT, or a REWRITE statement for a file opened INPUT.
- Any attempt to write or rewrite a record longer than the maximum record size specified when the file was defined.
- Any action taken on a file after EOF has been encountered (entry-sequenced or key-sequenced file). If EOF is encountered on a key-sequenced file, a START or a READ statement can be issued to reset the CRP and continue processing. For example, a key-sequenced file with ACCESS IS SEQUENTIAL specified:

```

OPEN
READ      successful
READ      EOF encountered
READ      logic error
START     reset CRP
READ      successful
  
```

or, a key-sequenced file with ACCESS IS DYNAMIC specified:

```

OPEN
READ NEXT  successful
READ NEXT  EOF encountered
READ NEXT  logic error
READ       reset CRP (random READ)
READ NEXT  successful
  
```

- An attempt to rewrite when ACCESS IS SEQUENTIAL has been specified if the preceding action was not a successful READ operation.
- An attempt to delete when ACCESS IS SEQUENTIAL was specified if the preceding action was not a successful READ operation (key-sequenced file only).

WRITING RECORDS INTO A VSAM FILE

The COBOL WRITE statement is used to add a record to a file. (Existing records in the file are not replaced with this statement.) The record to be written must not be larger than the maximum record size specified when the file was defined.

Entry-Sequenced File Considerations for the WRITE Statement

Entry-Sequenced File records are written sequentially. If the file is not opened OUTPUT or EXTEND, FILE STATUS=92 results and the record is not written.

Key-Sequenced File Considerations for the WRITE Statement

When ACCESS IS SEQUENTIAL is specified, the file must be opened OUTPUT or EXTEND. If not, the WRITE statement is not executed and FILE STATUS is set to 92.

The records must be written in ascending key sequence. If the file is opened EXTEND, the record keys of the records to be added must be higher than the highest record key on the file when it was opened. The following example shows the action and resultant FILE STATUS when a file containing records whose keys are 2, 4, 6, 8, and 10 is opened EXTEND. (Refer to Table 13 explanations of FILE STATUS values at action request time.)

<u>ACTION</u>	<u>FILE STATUS</u>
WRITE RECORD KEY = 8	92
WRITE RECORD KEY = 9	92
WRITE RECORD KEY = 12	00
WRITE RECORD KEY = 11	21

If many records are to be added to a file, it is strongly recommended that sequential access be used. Performance is improved both for the process of adding the records and for later retrieval of them.

When ACCESS IS RANDOM or ACCESS IS DYNAMIC is specified, the file must be opened I-O or OUTPUT. If not, the WRITE statement is not executed and FILE STATUS is set to 92. The records can be written in any order.

REWRITING RECORDS ON A VSAM FILE

The COBOL REWRITE statement is used to replace existing records on the file.

Entry-Sequenced File Considerations for the REWRITE Statement

For successful REWRITE statement execution, the file must be opened I-O. The record to be rewritten must first be



read by the COBOL program, then updated by the REWRITE statement. (The length of the record being rewritten cannot be changed.) If there was no preceding READ statement, or if the preceding READ statement was not successful (EOF was reached), the REWRITE statement is not executed and FILE STATUS is set to 92.

Key-Sequenced File Considerations for the REWRITE Statement

For successful REWRITE statement execution, the file must be opened I-O. The length of the record can be changed, but the value of the record key cannot be changed.

When ACCESS IS SEQUENTIAL is specified, the record to be rewritten must first be read by the COBOL program, then updated by the REWRITE statement. The REWRITE statement is not successful if the preceding statement for the file was not a successful READ of this record. This causes file status to be set to 92.

When ACCESS IS RANDOM or ACCESS IS DYNAMIC is specified, the record does not need to be read by the COBOL program. The record is updated by moving its key to RECORDKEY and doing the REWRITE.

READING RECORDS ON A VSAM FILE

The COBOL READ statement is used to access records on a file. If the file is not opened INPUT or I-O, the READ statement is not executed and FILE STATUS is set to 92.

Entry-Sequenced File Considerations for the READ Statement

Records are read sequentially, in the order in which they were written.

Key-Sequenced File Considerations for the READ Statement

When ACCESS IS SEQUENTIAL is specified, records are read sequentially, beginning at the position of the current record pointer. If the current record pointer is undefined when the READ is executed, FILE STATUS is set to 94. The following example shows

successful and unsuccessful READ and START executions. (Assume this file has records with keys 1 through 8 and 20.)

OPEN I-O filename	CRP at first record on file
READ (move 10 to record key)	First record on file is read
START	START fails with no record found (23)
READ (move 20 to record key)	READ fails with no CRP (94)
START	START is successful
READ	Record 20 is read
READ	EOF is reached, FILE STATUS=10
READ (move 8 to record key)	Logic error, FILE STATUS=92
START	START is successful
READ	record 8 is read.

When ACCESS IS RANDOM is specified, records are read in the order specified by the program. To read records whose record key is 10, move 10 to the RECORD KEY field in the record and issue a READ statement.

When ACCESS IS DYNAMIC is specified, records can be read randomly or sequentially. The READ NEXT statement is used for sequential accessing, and the RFA statement is used for random accessing.

READ NEXT Statement

Records are read sequentially beginning at the position of the current record pointer. If the current record pointer is not defined when the READ NEXT statement is issued, FILE STATUS is set to 94. The current record pointer is considered undefined if the preceding START or READ statement was not successful.

For details on the effect of COBOL statements on the position on the current record pointer, refer to the section entitled "Current Record Pointer."

READ Statement

The READ statement reads records randomly using the value placed in the record key field.

USING THE START VERB

The START statement is only valid for key-sequenced files but not when ACCESS IS RANDOM is specified or when the file is opened OUTPUT or EXTEND.

In some of the preceding examples, the START verb was used to position the CRP. Then the READ (for ACCESS IS SEQUENTIAL) and READ NEXT (for sequential processing when ACCESS IS DYNAMIC) retrieves the record pointed to by the CRP as established by the START.

Example:

```
05 RECORD-KEY.
   10 GEN11.
       15 GEN12 PIC 99.
       15 GEN13 PIC 99.
   10 GEN14 PIC9.
```

In this example, GEN12, GEN11, or RECORD-KEY could be used as the data-name in the "KEY IS relational data-name" option of the START statement. The lengths would be 2, 4, and 5 respectively. GEN13 and GEN14 could not be used as they are not in the leftmost part of RECORD-KEY.

Assume that the value of RECORD-KEY is 01472:

- START filename KEY = GEN11 would position the CRP to the first record on the file whose key has 0147 in the first 4 bytes.
- START file-name KEY > GEN12 would position the CRP to the first record in the file whose key has the first two bytes greater than 01.

DELETE Statement

The DELETE is valid only for a key-sequenced file. The same considerations discussed under "Key-Sequenced File Considerations for the REWRITE Statement" apply to the DELETE statement.

COBOL Language Usage With VSAM

The COBOL language statements which are directly related to VSAM processing are in the section "DOS/VS COBOL Considerations" in the publication IBM DOS Full American National Standard COBOL. The following paragraphs are intended only to highlight and summarize the basic language statements used in writing a VSAM-file-processing COBOL program.

COBOL programmer can use VSAM in three basic ways: to create a file, to retrieve a file, and to update a file. However, prior to processing a VSAM file, it is an absolute necessity that the previously discussed Access Method Services functions be performed. Most significant to the COBOL programmer is whether the file is defined as an entry-sequenced file or as a key-sequenced file.

Creating a VSAM File

The minimum COBOL language statements required to create a VSAM file are summarized in Table 14.

Table 14. COBOL Statements for Creating a VSAM File

	Entry-Sequenced File	Key-Sequenced File
Environment Division	SELECT ASSIGN	SELECT ASSIGN ORGANIZATION IS INDEXED RECORD KEY
Data Division	FD entry LABEL RECORDS	FD entry LABEL RECORDS
Procedure Division	OPEN OUTPUT or OPEN EXTEND WRITE CLOSE	OPEN OUTPUT or OPEN EXTEND WRITE CLOSE

The following examples show the creation of VSAM files.

Example 1:

This example shows the creation of an entry-sequenced VSAM file. The FILE STATUS facility is used to monitor all I/O operations in the program.

VSAM

IDENTIFICATION DIVISION.

ENVIRONMENT DIVISION.

INPUT-OUTPUT SECTION.

FILE-CONTROL.

SELECT INREC
ASSIGN TO SYS005-UR-2540R-CARDIN.
SELECT OUTREC
ASSIGN TO SYS010-AS-OUTMAST
FILE STATUS IS CHK.

DATA DIVISION.

FILE SECTION.

FD INREC LABEL RECORDS ARE OMITTED
DATA RECORD IS INMASTER.
01 INMASTER PIC X(80).
FD OUTREC LABEL RECORDS ARE STANDARD
DATA RECORD IS OUTMASTER.
01 OUTMASTER PIC X(80).
WORKING-STORAGE SECTION.
77 CHK PIC XX.

PROCEDURE DIVISION.

PARA1.

OPEN INPUT INREC OUTPUT OUTREC.
IF CHK IS NOT = "00" GO TO CHKRTN.

PARA2.

READ INREC INTO OUTMASTER
AT END GO TO PARA4.

PARA3.

WRITE OUTMASTER.
IF CHK IS NOT = "00" GO TO CHKRTN.
GO TO PARA2.

PARA4.

CLOSE INREC OUTREC.
IF CHK IS NOT = "00" GO TO CHKRTN.

FINIT.

STOP RUN.

CHKRTN.

DISPLAY "ERROR. STATUS KEY
VALUE IS" CHK
GO TO FINIT.

Note that in this example any Status Key return other than 00 causes transfer of control to paragraph CHKRTN. This routine can determine the exact cause of the error by checking the Status Key. Once the cause is determined, instructions can be issued according to the user's desired response to each type of error.

Example 2:

This example shows the creation of a COBOL key-sequenced VSAM file. This

program performs the same function as example 1 except that now a key-sequenced file is being created. The records in the file "INREC" are in ascending key order.

IDENTIFICATION DIVISION.

ENVIRONMENT DIVISION.

INPUT-OUTPUT SECTION.

FILE-CONTROL.

SELECT INREC
ASSIGN TO SYS005-UR-2540R-CARDIN.
SELECT OUTREC
ASSIGN TO SYS010-OUTMAST
ORGANIZATION IS INDEXED
RECORD KEY IS ARG-1
FILE STATUS IS CHK.

DATA DIVISION.

FILE SECTION.

FD INREC LABEL RECORDS ARE OMITTED
DATA RECORD IS INMASTER
01 INMASTER PIC X(80).
FD OUTREC LABEL RECORDS ARE STANDARD
DATA RECORD IS OUTMASTER.
01 OUTMASTER.
05 FILLER PIC X.
05 ARG-1 PIC XXX.
05 REM PIC X(76).
WORKING-STORAGE SECTION.
77 CHK PIC XX.

PROCEDURE DIVISION.

PARA1.

OPEN INPUT INREC OUTPUT OUTREC.
IF CHK IS NOT = "00" GO TO CHKRTN.

PARA2.

READ INREC INTO OUTMASTER
AT END GO TO PARA4.

PARA3.

WRITE OUTMASTER.
IF CHK IS NOT = "00" GO TO CHKRTN.
GO TO PARA2.

PARA4.

CLOSE INREC OUTREC.
IF CHK IS NOT = "00" GO TO CHKRTN.

FINIT.

STOP RUN.

CHKRTN.

DISPLAY "ERROR. STATUS KEY VALUE
IS" CHK
GO TO FINIT.

Note that in this example any Status Key return other than 00 causes transfer of control to paragraph CHKRTN. This routine can determine the exact cause of the error

y checking the Status Key. Once the cause is determined, instructions can be issued according to the user's desired response to each type of error.

Retrieving a VSAM File

The minimum COBOL language statements required to retrieve a VSAM file are summarized in Table 15.

Table 15. COBOL Statements for Retrieving a VSAM File

	Entry-Sequenced File	Key-Sequenced File
Environment Division	SELECT ASSIGN	SELECT ASSIGN ORGANIZATION IS INDEXED RECORD KEY
Data Division	FD entry LABEL RECORDS	FD entry LABEL RECORDS
Procedure Division	OPEN INPUT READ ... AT END CLOSE	OPEN INPUT READ CLOSE

The following examples show the retrieval of records from VSAM files.

Example 3:

This example shows the retrieval of records from the entry-sequenced file created in example 1. The records are then printed.

```
IDENTIFICATION DIVISION.
.
.
ENVIRONMENT DIVISION
.
.
INPUT-OUTPUT SECTION.
FILE-CONTROL
  SELECT INREC
  ASSIGN TO SYS010-AS-INMAST
  FILE STATUS IS CHK.
  SELECT PREC
  ASSIGN TO SYS005-UR-1403-S-PRNTR
.
.
DATA DIVISION.
FILE SECTION.
FD INREC LABEL RECORDS ARE STANDARD
```

```
DATA RECORD IS INMASTER.
01 INMASTER PIC X(80).
FD PREC LABEL RECORDS ARE OMITTED
DATA RECORD IS POUT.
01 POUT PIC X(80).
WORKING-STORAGE SECTION.
77 CHK PIC XX.
PROCEDURE DIVISION.
PARA1.
  OPEN INPUT INREC OUTPUT PREC.
  IF CHK IS NOT = "00" GO TO CHKRTN.
PARA2.
  READ INREC INTO POUT AT END GO TO
  PARA4.
  IF CHK IS NOT = "00" GO TO CHKRTN
PARA3.
  WRITE POUT.
  GO TO PARA2.
PARA4.
  CLOSE OUTREC PREC.
  IF CHK IS NOT = "00" GO TO CHKRTN.
FINIT.
  STOP RUN.
CHKRTN.
  DISPLAY 'ERROR. STATUS KEY VALUE
  IS' CHK.
  GO TO FINIT.
```

Note that in this example any Status Key return other than 00 causes transfer of control to paragraph CHKRTN. This routine can determine the exact cause of the error by checking the Status Key. Once the cause is determined, instructions can be issued according to the user's desired response to each type of error.

Example 4:

This example shows the retrieval of records from the key-sequenced file created in example 2. Note that in the Procedure Division there is a switch from sequential processing to random processing; this is permitted since ACCESS IS DYNAMIC is specified in the ENVIRONMENT Division.

```
IDENTIFICATION DIVISION.
.
.
ENVIRONMENT DIVISION.
.
.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT INREC
  ASSIGN TO SYS010-INMAST
  ORGANIZATION IS INDEXED
  ACCESS IS DYNAMIC
  RECORD KEY IS ARG-1
  FILE STATUS IS CHK.
  SELECT PREC
  ASSIGN TO SYS005-UR-1403-S-PRNTR
.
.
```



```

DATA DIVISION.
FILE SECTION.
FD INREC LABEL RECORDS ARE STANDARD
  DATA RECORD IS INMASTER.
01 INMASTER.
  05 FILLER PIC X.
  05 ARG-1 PIC XXX.
  05 ARG-2 PIC XX.
  05 ARG-3 PIC XX.
  05 FILLER PIC X(72).
FD PREC LABEL RECORDS ARE OMITTED
  DATA RECORD IS POUT.
01 POUT PIC X(80).
WORKING-STORAGE SECTION.
77 CHK PIC XX.
PROCEDURE DIVISION.
PARA1.
  OPEN INPUT INREC OUTPUT PREC.
  IF CHK IS NOT = "00" GO TO CHKRTN.
PARA2.
  MOVE "003" TO ARG-1.
  START INREC.
PARA3.
  READ INREC NEXT RECORD AT END GO TO
  PARA4.
  IF CHK IS NOT = "00" GO TO CHKRTN.
  IF ARG-2 IS = "02" GO TO PARA4.
  IF ARG-3 IS NOT = "73" GO TO PARA3.
  WRITE POUT FROM INMASTER.
  GO TO PARA3.
PARA4.
  MOVE "101" TO ARG-1.
  READ INREC INVALID KEY GO TO CHKRTN.
  WRITE POUT FROM INMASTER.
  MOVE "103" TO ARG-1.
  READ INREC INVALID KEY GO TO CHKRTN.
  WRITE POUT FROM INMASTER.
PARA5.
  CLOSE INREC PREC.
  IF CHK IS NOT = "00" GO TO CHKRTN.
FINIT.
  STOP RUN.
CHKRTN.
  DISPLAY 'ERROR. STATUS KEY VALUE
  IS' CHK.
  GO TO FINIT.

```

Note that in this example any Status Key return other than 00 causes transfer of control to paragraph CHKRTN. This routine can determine the exact cause of the error by checking the Status Key. Once the cause is determined, instructions can be issued according to the user's desired response to each type of error.

Job Control Language for a VSAM File

JCL is simplified for VSAM since all VSAM files must be cataloged through Access Method Services.

The JCL to execute the program in example 1 is

```

// JOB
// ASSGN      SYS010,X'233'
// DLBL       OUTMAST,'PAYFILE',,VSAM
// EXTENT     SYS010,VSAMVOL
// EXEC       EXAMPLE,SIZE=50K

```

The volume on which the VSAM file was defined is mounted at address 233, the volume ID is VSAMVOL, and the file was given the name PAYFILE when it was defined. The SIZE parameter is required on the EXEC card for VSAM programs.

Converting Non-VSAM Files to VSAM Files

ISAM files can be converted to VSAM files so that they may be processed by a COBOL program using VSAM. The conversion is done through Access Method Services.

Essentially, the conversion process consists of defining a VSAM file as the target for the file being converted. Then through the appropriate JCL and the REPRO command, the conversion is accomplished.

For a complete description of the conversion process, see DOS/VS Utilities VSAM Access Method Services, and DOS/VS Data Management Guide.

Using ISAM Programs to Process VSAM Files

Once the file is converted the programmer can process the new VSAM file with his old ISAM program by converting his ISAM JCL to VSAM JCL. For more details on this procedure see DOS/VS Data Management Guide.

DETAILED FILE PROCESSING CAPABILITIES

The following topics are discussed within this chapter:

COBOL VSAM Control Blocks

DTF Tables

Error Recovery for Non-VSAM Files

Volume and File Label Handling

the Environment Division (SELECT, RERUN, and SAME statements) and the Data Division (FD and associated records). The File Control Block (FCB) is generated dynamically at execution time by the VSAM library subroutines. The user may wish to refer to fields in these blocks for debugging. The format of the VSAM control block (Access Method Control Block -- ACP) is not given here, as the knowledge of its contents is not needed by the COBOL user.

COBOL VSAM CONTROL BLOCKS

The compiler generates a File Information Block (FIB) from information in

FILE
PROC

CONTROL BLOCKS FOR VSAM

The following two control blocks are required to process input/output requests for VSAM files.

VSAM FILE INFORMATION BLOCK (FIB)

The file information block, a portion of the completed object module, is used at execution time by the ILBDINT0, ILBDVOC0, and ILBDVIO0 COBOL library subroutines for processing input/output verbs used with VSAM files. The FIB is built by phase 21.

Fixed Portion:

Displacement		Field	No. of Bytes	Description
Hex	Decimal			
0	0	IFIBID	1	FIB identification code: X'I'
1	1	IFIBLVL	1	FIB level number
2	2	INAMED	7	External name
9	9	INAMEDB	1	External name
A	10		1	Reserved
B	11	IORG	1	ORGANIZATION
Code:				
				Equate Bit
				<u>Bits</u> <u>Name</u> <u>Settings</u> <u>Meaning</u>
				0-7 IORGVPS 1000 1000 VSAM ADDRESSED
				IORGVIX 0100 1000 VSAM INDEXED
C	12	IACCESS	1	ACCESS MODE
Code:				
				Equate Bit
				<u>Bits</u> <u>Name</u> <u>Settings</u> <u>Meaning</u>
				0-7 IACCSEQ 1000 0000 SEQUENTIAL
				IACCRAN 0100 0000 RANDOM
				IACCDYN 0010 0000 DYNAMIC
D	13		1	Reserved
E	14	ISW1	1	Miscellaneous switches
Code:				
				Equate Bit
				<u>Bits</u> <u>Name</u> <u>Settings</u> <u>Meaning</u>
				0-7 ISOPTNL 1000 0000 OPTIONAL specified
				ISSAMREC 0010 0000 SAME RECORD AREA specified
F	15		1	Reserved
10	16		6	Reserved
16	22	IRECLEN	2	Number of bytes in longest 01-entry
18	24	IRECDBL	2	Displacement in TGT of record's first base locator cell
1A	26	IRECNBL	1	Number of base locators for RECORD AREA
1B	27		1	Reserved
1C	28	ISTATDBL	2	Displacement in TGT of base locator for STATUS data-name
1E	30	ISTATDDN	2	Displacement from base locator of STATUS data-name
20	32	ISTATLDN	2	Length of STATUS data-name
22	34		1	Reserved
23	35	IKEYNO	1	Number of entries in key list

24	36	IKEYFNLT	2	Length of each entry in key list
26	38	IPSWISW	1	Miscellaneous switches
27	39	IPSWNO	1	Number of entries in password list
28	40	IPSWENTL	2	Length of each entry in password list
2A	42		14	Reserved
38	56	IMISCAD	4	Address in variable length portion of FIB for miscellaneous clauses
3C	60	ILABELAD	4	Reserved
40	64	IKEYLSTA	4	Address of first key list entry
44	68	IPSWLSTA	4	Address of first password list entry
48	72		16	Reserved

Variable Length Portion:

Supplementary information for miscellaneous clauses (one for each clause):

Displacement		Field	No. of		Description
Hex	Decimal		Bytes		
0	0	IMSW1	2		Switch bytes.
Code:					
		Equate	Bit		
<u>Bits</u>	<u>Name</u>	<u>Settings</u>	<u>Meaning</u>		
0-7	IMRREOV	1000 0000	RERUN at end of volume		
8-15			Reserved		
2	2	IRERUNI	4		RERUN integer (field contains zeros if RERUN not specified)
6	6		2		Slack bytes
8	8	IRERUNN	8		External-name of RERUN clause

Key List Entry: (one per user-defined key--RECORD/ALTERNATE/RELATIVE)

Displacement		Field	No. of		Description
Hex	Decimal		Bytes		
0	0	KEYSW	1		Miscellaneous switches
Code:					
		Equate	Bit		
<u>Bits</u>	<u>Name</u>	<u>Settings</u>	<u>Meaning</u>		
0-7	IKEYCOMP	1000 0000	Key is USAGE COMP (binary)		
1	1	IKEYLDN	1		Length of key data-name
2	2	IKEYDBL	2		Displacement of key data-name's locator in TGT
4	4	IKEYDDN	2		Data-name displacement from locator

Password List Entry: (one per password)

0	0	IPSWDIXN	1	Associated index number
0 = none				
1 = primary				
1	1	IPSWDLDN	1	Length of password data-name
2	2	IPSWDDEL	2	Displacement of password data-name's locator in TGT
4	4	IPSWDDDN	2	Data-name displacement from locator



VSAM FILE CONTROL BLOCK (FCB)

The VSAM File Control Block is created by the ILBDINT0 COBOL library subroutine. It is used by the ILBDVIO0 and ILBDVOC0 subroutines to interface with the VSAM system control subroutines

Displacement		Field	No. of Bytes	Description
Hex	Decimal			
0	0	FCBID	1	FCB identification code: 'F'
1	1	FCBLVL	1	FCB level number
2	2	FOPENOPT	4	Save area for OPEN options
6	6	FCLOSOPT	4	Save area for CLOSE options
A	10		2	Reserved
C	12	FCOBRTN	4	Address of COBOL transmitter routine
10	16	FUSERR	4	Address of USE...ERROR declarative
14	20	FUSELIST	4	Address of USE declarative Exit List
18	24		6	Reserved
1E	30	FRECKEY	1	Number of RECORD KEY
1F	31	FADVANC	1	Reserved
20	32	FENDINV	4	Return address from INVALID KEY, AT END, or end-of-page
24	36		12	Reserved for compilation-dependent fields
30	48	FOPENOPS	4	Options for VSAM OPEN verb

Code:

Bits	Equate Name	Bit Settings	Meaning
0-7	FOPIN	1000 0000	INPUT
	FOPOUT	0100 0000	OUTPUT
	FOPIO	0010 0000	I-O
	FOPEXT	0001 0000	EXTEND
8-15	Reserved		
16-23	FOPUERR	1000 0000	USE...ERROR declarative address in FUSERR cell
24-31	Reserved		

34 52 FCLOSOPS 4 VSAM CLOSE options ?

Code:

Bits	Equate Name	Bit Settings	Meaning
0-7	FCLLOCK	0001 0000	LOCK
8-31	Reserved		

38 56 PSW1 4 Miscellaneous switches

Code:

Bits	Equate Name	Bit Settings	Meaning
0-7	FSOPEN	1000 0000	File is open
	FSLOCKED	0100 0000	File is closed with lock
	FSOPTNL	0010 0000	Optional file not present
	FSOKACT	0001 0000	Successful action has occurred since open
	FSEOF	0000 1000	Sequential read has encountered end-of-file
	FSVCORE	0000 0100	Main storage to process this open has been acquired
8-31	Reserved		Reserved

3C 60 FTRSTMT 4 Transmission statement switches

Code:

<u>Bits</u>	<u>Equate</u>	<u>Bit</u>	<u>Settings</u>	<u>Meanings</u>
0-7	FTREAD		0000 0100	READ statement
	FTWRITE		0000 1000	WRITE statement
	FTREWRT		0000 1100	REWRITE statement
	FTSTART		0001 0000	START
	FTDELET		0001 0100	DELETE statement
8-15	FTINVKEY		1000 0000	INVALID KEY
	FTATEND		0100 0000	AT END
	FTNEXT		0000 0010	NEXT
	FTKEY		0000 0001	KEY
16-23	FTRSCHGT		1000 0000	GREATER THAN
	FTRSCHGE		0100 0000	EQUAL TO
	FTRSCHGE		0010 0000	NOT LESS THAN
24-31	Reserved			

40	64	FSYSCBAL	4	Address of system control blocks address list
44	68	FSYSCBLL	4	Address of system control blocks lengths list
48	72	FSYSCBNO	2	Number of system control blocks (DTP, DCB, ACB)
4A	74	FKEYLEN	2	Length of KEY data-name
4C	76	FRECCNT	4	Record count for checkpoint subroutine, if RERUN specified
50	80	FFIBAD	4	Address of File Information Block (FIB)
54	84	FWORKAD	4	Address of system-dependent work area
58	88	FRECA	4	Address of current record area
5C	92	FSAMRECA	4	Address of SAME RECORD AREA
60	96	FSTATKEY	2	STATUS KEY work area
62	98	FLASTREQ	1	Last I/O statement

Code:

<u>Bits</u>	<u>Equate</u>	<u>Bit</u>	<u>Settings</u>	<u>Meanings</u>
0-7	FLASTRD		0000 0100	READ statement
	FLASTWRT		0000 1000	WRITE statement
	FLASTRWT		0000 1100	REWRITE statement
	FLASTSTR		0001 0000	START statement
	FLASTDLT		0001 0100	DELETE statement
	FLASTOPN		0001 1000	OPEN statement
	FLASTCLO		0001 1100	CLOSE statement

63 99 13 Reserved

FILE
PROC

DTF TABLES

Whenever COBOL imperative-statements (READ, WRITE, REWRITE, etc.) are used in a program to control the input and/or output of records in a file, that file must be defined by a DTF. A DTF is created by the compiler for each file opened in a COBOL program from information specified in the Environment Division, FD entry, and input/output statements in the source program. The DTF for each file is part of the object module that is generated by the compiler. It describes the characteristics of the logical file, indicates the type of processing to be used for the file, and specifies the storage areas and routines used for the file.

The DTF's generated for the permissible combinations of device type and COBOL file processing technique are as follows:

DTFCD	Card reader, punch -- organization and access sequential
DTFPR	Printer -- organization and access sequential
DTFMT	Tape -- organization and access sequential
DTFSD	Mass storage device -- organization and access sequential
DTFDA	Mass storage device -- organization direct, access sequential or random
DTFIS	Mass storage device -- organization indexed, access sequential or random

DTFDU 3540 diskette -- organization and access sequential

Because of their limited interest for the COBOL programmer, the contents and location of the fields of each of the DTF types are not discussed in this publication. However, there are certain fields which immediately precede the storage area allocated for the DTF which are pertinent. These fields are provided on the listing in hexadecimal if an abnormal termination occurs and the SYMDMP option is in effect. The SYMDMP option is described in detail in the chapter "Symbolic Debugging Features." Fields preceding the DTF are described below.

For magnetic tape files (DTFMT) or sequentially organized files on mass storage devices (DTFSD), a 26-byte Pre-DTF is reserved in front of the DTF. The fields of the Pre-DTF are shown in Table 16. If any option is not specified, the field will contain binary zeros.

When actual track addressing is used for files with direct organization and random access (DTFDA), a variable-length Pre-DTF is reserved. The fields of the Pre-DTF are shown in Table 17. If any option is not specified, the field will contain binary zeros.

When relative track addressing is used for files with direct organization and random access (DTFDA), a variable-length Pre-DTF is reserved. The fields of the Pre-DTF are shown in Table 18. If any option is not specified, the field will contain binary zeros.

Table 16. Fields Preceding DTFMT and DTFSD

2 bytes	Length of nonstandard label, if present
1 byte	Number of reels (as specified in the ASSIGN clause) when file is opened ¹
1 byte	Number of reels remaining (i.e., file not completely read) ¹
2 bytes	Maximum record length if records are variable, blocked and APPLY WRITE-ONLY is not specified.
4 bytes	Address of label declarative with BEGINNING REEL option UNIT
4 bytes	Address of label declarative with ENDING REEL option UNIT
4 bytes	Address of label declarative with ENDING FILE option
4 bytes	Address of label declarative with BEGINNING FILE option
1 byte	Switch -- PF if closed WITH LOCK; otherwise, the switch is used as shown in Table 17
3 bytes	Address of USE AFTER STANDARD ERROR declarative
DTFMT/DTFSD	
¹ For INPUT files with nonstandard labels only.	

Table 17. Fields Preceding DTFDA -- ACCESS IS RANDOM -- Actual Track Addressing

9-263 bytes	ACTUAL KEY ¹
8 bytes	SEEK Address ²
2 bytes	Error bytes ³
4 bytes	Address of file extent information
4 bytes	Address of label declarative with ENDING FILE option
4 bytes	Address of label declarative with BEGINNING FILE option
1 byte	Switch -- PF if closed WITH LOCK; otherwise the switch is used as shown in Table 17
3 bytes	Address of USE AFTER STANDARD ERROR declarative
DTFDA	
¹ ACTUAL KEY specified in last executed WRITE statement	
² In the form MBECCHRR	
³ This area is reserved by the Supervisor and assigned the name ERRBYTE. For a complete discussion, refer to the publication <u>DOS/VS Supervisor and I/O Macros</u> , Order No. GC24-5037.	

Table 18. Fields Preceding DTFDA -- ACCESS IS RANDOM -- Relative Track Addressing

5-258 bytes	ACTUAL KEY ¹
4 bytes	SEEK address ²
3 bytes	Last extent used ³
1 byte	Not used
2 bytes	Error bytes ⁴
1 byte	Index to last extent used in the Disk Extent Table
3 bytes	Address of Disk Extent Table in the DTF
4 bytes	Address of label declarative with ENDING FILE option
4 bytes	Address of label declarative with BEGINNING FILE option
1 byte	Switch -- FF if closed WITH LOCK; otherwise the switch is used as shown in Table 17
3 bytes	Address of USE AFTER STANDARD ERROR declarative
DTFDA	
¹ ACTUAL KEY specified in the last executed WRITE statement ² In the form TTR ³ In the form TTT ⁴ This area is reserved by the DOS/VS Supervisor and assigned the name ERRBYTE. For a complete discussion, refer to the publication <u>DOS/VS Supervisor and I/O Macros</u> .	

When actual track addressing is used for files with direct organization and sequential access (DTFDA), a 31-byte Pre-DTF is reserved. The fields of the Pre-DTF are shown in Table 19. If any option is not specified, the field will contain binary zeros.

When relative track addressing is used for files with direct organization and sequential access (DTFDA), a 31-byte

Pre-DTF is reserved. The fields of the Pre-DTF are shown in Table 20. If any option is not specified, the field will contain binary zeros.

For files whose organization is indexed, eight bytes are reserved preceding the DTF, as shown in Table 21. The fields preceding the DTFDU for the 3540 are shown in Table 22.

Table 19. Fields Preceding DTFDA -- ACCESS IS SEQUENTIAL -- Actual Track Addressing

8 bytes	SEEK address ¹
5 bytes	IDLOC ²
2 bytes	Error bytes ³
4 bytes	Address of file extent information
4 bytes	Address of label declarative with ENDING FILE option
4 bytes	Address of label declarative with BEGINNING FILE option
1 byte	Switch -- FF if closed WITH LOCK; otherwise the switch is used as shown in Table 17
3 bytes	Address of USE AFTER STANDARD ERROR declarative
DTFDA	
¹ In the form MBBCCHHR ² Address (returned by the system) of next record in the form CCHHR ³ This area is reserved by the DOS/VS Supervisor and assigned the name ERBYTE. For a complete discussion, refer to the publication <u>DOS/VS Supervisor and I/O Macros</u> .	



Table 20. Fields Preceding DTFDA -- ACCESS IS SEQUENTIAL -- Relative Track Addressing

4 bytes	SEEK address ¹
3 bytes	Last extent used ²
1 byte	Not used
4 bytes	IDLOC ³
1 byte	Not used
2 bytes	Error bytes ⁴
1 byte	Index to the last extent used in the Disk Extent Table
3 bytes	Address of Disk Extent Table in the DTF
4 bytes	Address of label declarative with ENDING FILE option
4 bytes	Address of label declarative with BEGINNING FILE option
1 byte	Switch -- PF if closed with LOCK; otherwise the switch is used as shown in Table 17
3 bytes	Address of USE AFTER STANDARD ERROR declarative
DTFDA	
¹ In the form TTTR ² In the form TTT ³ Address (returned by the system) of the next record in the form TTTR ⁴ This area is reserved by the DOS/VS Supervisor and assigned the name ERREBYTE. For a complete discussion, refer to the publication <u>DOS/VS Supervisor and I/O Macros</u> .	

Table 21. Fields Preceding DTFIS

2 bytes	Unused
2 bytes	Displacement of record key within record
1 byte	Switch -- PF if closed WITH LOCK; otherwise the switch is used as shown in Table 17
3 bytes	Address of USE AFTER STANDARD ERROR declarative
DTFIS	

Table 22. Fields Preceding DTFDU

4 bytes	Unused
1 byte	DTF switch -- FF if closed with LOCK
3 bytes	Address of USE AFTER STANDARD ERROR declarative
DTFDU	

Some files can be opened several different ways in one COBOL program.

For DTFCD and DTFPR, only one DTF will be generated for each file.

For DTFMT, a maximum of three DTF's may be needed -- one each for OPEN INPUT, OPEN INPUT REVERSED, and OPEN OUTPUT.

For DTFSD, a maximum of three DTF's may be needed -- one each for OPEN INPUT, OPEN OUTPUT, and OPEN I-O statements.

For DTFIS and DTFDA, only one DTF is needed.

Pre-DTF Switch

When used, this switch provides communication between the executing program and its input/output subroutines at execution time. The entire byte may be set to X'FF' to indicate that the file was closed WITH LOCK and cannot be reopened. Otherwise the switch is used as shown in Table 23.

ERROR RECOVERY FOR NON-VSAM FILES

COBOL allows the programmer to handle input/output errors through 1) the INVALID KEY clause for certain source language statements, and 2) the USE AFTER STANDARD ERROR declarative sentence.

Input/output errors caused by the program can be recovered from directly by the procedure specified in the INVALID KEY clause. That is, when the system determines that an invalid key condition exists, control is returned to the

programmer at the imperative-statement specified in the INVALID KEY clause. An invalid key condition can occur on files with direct or indexed organization and on sequentially organized disk files. The errors that cause an invalid key condition are shown in Table 24.

Table 23. Meaning of Pre-DTF Switch

Bit	Meaning, if ON
0	Turned ON the first time a DTFSD output file is opened. The entire DTF is saved for subsequent OPEN OUTPUT statements.
1	Turned ON when DTFDA or DTFSD files are opened I-O.
2	This bit is ON to indicate beginning of volume user label processing. The bit is set OFF when a file is opened to indicate to the user label processing subroutine (ILBDUSL0) that beginning-of-file user labels are to be processed. That subroutine sets the bit ON after beginning-of-file processing to indicate that all subsequent calls for this subroutine are for beginning-of-volume user label processing.
3	For output files with variable-length blocked records, this bit is turned OFF when a file is opened and ON for all WRITE's after the first.
4	Turned ON for spanned record processing on a DTFDA file.
5-7	Not used.



Table 24. Errors Causing an Invalid Key Condition

Organization	ACCESS	OPEN	I-O Verb	Condition
Sequential	[SEQUENTIAL]	OUTPUT	WRITE	End of extents reached.
Direct	[SEQUENTIAL]	OUTPUT	WRITE	Track address outside file extents.
Direct	RANDOM	INPUT	READ	No record found.
		OUTPUT	WRITE	Track address outside file extents.
		I-O	READ REWRITE	Track address outside file extents.
Indexed	[SEQUENTIAL]	INPUT I-O	START	No record found.
		OUTPUT	WRITE	Duplicate record; sequence check.
		INPUT	READ	No record found.
	RANDOM	I-O	REWRITE	
		I-O	WRITE	Duplicate record.

Other input/output errors cause the job to be cancelled unless the programmer has specified a USE AFTER STANDARD ERROR declarative. Control is transferred to this declarative section if the system determines that a "standard" error has occurred during input/output processing. In this declarative section, the programmer

may interrogate the COBOL error bytes if he has specified the GIVING option of the USE AFTER STANDARD ERROR declarative sentence. The meaning of these bytes for a specified combination of device type and file processing technique is shown in Table 25.

Table 25. Meaning of Error Bytes for GIVING Option of Error Declarative (Part 1 of 2)

Device	Organization	ACCESS	OPEN	I/O Verb	Condition	Byte	Result	
Unit record	Sequential	[SEQUENTIAL]			Input/output error	1	File must be closed and job must be terminated.	
Tape	Sequential	[SEQUENTIAL]	INPUT	READ	Wrong length record	2	Skip block if return is made to non-declarative portion.	
					Parity error	1	Skip block if return is made to non-declarative portion.	
					OUTPUT WRITE		All exceptional conditions are handled by the system.	
DASD	Sequential	[SEQUENTIAL]	INPUT I-O	READ	Wrong length record	2	Skip block if return is made to non-declarative portion.	
					Parity error	1	Skip block if return is made to non-declarative portion.	
					OUTPUT WRITE I-O	Parity error	1	Bad block written.
					Wrong length record	2	Bad block written.	
DASD	Direct	[SEQUENTIAL]	INPUT	READ	Wrong length record	2	Return to statement after READ.	
					Data check in count area	1	Return to statement after READ.	
					Data check for key and/or data	4	Return to statement after READ.	
DASD	Direct	RANDOM	INPUT I-O	READ	Same as ACCESS SEQUENTIAL (above).			
					OUTPUT WRITE	Wrong length record	2	Return to next statement; bad block written.
					Data check in count area	1	Return to next statement; bad block written.	
					Data check for key and/or data	4	Return to next statement; bad block written.	
					No room found	3	Return to next statement.	

Note: If no USE AFTER STANDARD ERROR routine is specified and one of the above conditions occurs, the programmer is notified of the condition and the job is cancelled.

FILE PROC

Table 25. Meaning of Error Bytes for GIVING Option of Error Declarative (Part 2 of 2)

Device	Organization	ACCESS	OPEN	I/O Verb	Condition	Byte	Result			
DASD	Direct	RANDOM	I/O	REWRITE	Wrong length record	2	Return to next statement; bad block written.			
					Data check in count area	1	Return to next statement; bad block written.			
					Data check in key and/or data	4	Return to next statement; bad block written.			
DASD	Indexed	[SEQUENTIAL]	INPUT I-O	READ REWRITE	DASD error	1	Return to next statement; bad block read or written.			
					Wrong length record	2	Return to next statement; bad block read or written.			
				START	DASD error	1	Continued processing of file permitted.			
				OUTPUT WRITE	DASD error	1	Return to next statement; bad block written.			
					Wrong length record	2	Return to next statement; bad block written.			
			Prime data area full		3	File must be closed.				
			Cylinder index full	4	File must be closed.					
				Master index full	5	File must be closed.				
			DASD	Indexed	RANDOM	INPUT I-O	READ REWRITE	DASD error	1	Return to next statement; bad block read or written.
								Wrong length record	2	Return to next statement; bad block read or written.
I-O WRITE	DASD error	1					Return to next statement; bad block written.			
	Wrong length record	2					Return to next statement; bad block written.			
	Overflow area full	6					Files must be closed.			
3540	Sequential	Sequential	INPUT	READ	Data check	1	Return to next statement.			
				OUTPUT WRITE	Equipment check	2	Bad block read or written up until bad physical record.			

Note: If no USE AFTER STANDARD ERROR routine is specified and one of the above conditions occurs, the programmer is notified of the condition and the job is cancelled.

If the programmer includes a USE AFTER STANDARD ERROR routine without specifying the GIVING option, he must call an assembler language routine within the declarative if he wishes to interrogate the error bits -- set either in the DTF (DTFMT, DTFSD, or DTFIS) or in the fields preceding the DTF (DTFDA).

Interrogation of these error bits should be made to the locations shown in Tables 26, 27, 28, 29, and 30.

Note: The byte and bit displacement in Tables 26, 27, 28, 29, and 30 is relative to zero.

Table 26. Location and Meaning of Error Bits for DTFMT

OPEN	Verb	Condition	Byte*	Bit
INPUT	READ	Wrong length record	3	1
		Parity error	2	6
OUTPUT	WRITE	Wrong length record	3	1
		Parity error	2	6

*Within the DTF.

Table 27. Location and Meaning of Error Bits for DTFSD

OPEN	Verb	Condition	Byte*	Bit
INPUT, I-O	READ	Wrong length record	3	1
		Parity error	2	6
OUTPUT, I-O	WRITE	Parity error	2	6

*Within the DTF.

FILE
PROC

Table 28. Location and Meaning of Error Bits for DTFDA

ACCESS	OPEN	Verb	Condition	Byte*	Bit		
[SEQUENTIAL]	INPUT	READ	Wrong length record	0	1		
			Data check in count area	1	0		
			Data check in key or data	1	3		
			No record found	1	2 or 4		
RANDOM	INPUT, I-O	READ	Same as sequential				
			OUTPUT	WRITE	Wrong length record	0	1
					No room found	0	4
					Data check in count area	1	0
	Data check in key or data	1			3		
	I-O	REWRITE	Wrong length record	0	1		
			Data check in count area	1	0		
			Data check in key or data	1	3		
			No record found	1	2 or 4		

*Within error bytes preceding DTF. See the section "DTF Tables" for the location of these bytes.

Table 29. Location and Meaning of Error Bits for DTFIS

ACCESS	OPEN	Verb	Condition	Byte*	Bit
[SEQUENTIAL]	INPUT, I-O	READ	DASD error	30	0
			Wrong length record	30	1
	OUTPUT	WRITE	DASD error	30	0
			Wrong length record	30	1
			Prime data area full	30	2
			Cylinder index full	30	3
			Master index full	30	4
RANDOM	INPUT, I-O	READ REWRITE	DASD error	30	0
			Wrong length record	30	1
	I-O	WRITE	DASD error	30	0
			Wrong length record	30	1
			Overflow area full	30	6

*Within the DTF.

Table 30. Location and Meaning of Error Bits for DTFDU

ACCESS	OPEN	Verb	Condition	Byte*	Bit
Sequential	Input	READ	Data check	3	3
	Output	WRITE	Equipment check	2	2

The following should be considered when processing tape input files:

1. Two types of errors are returned to the programmer: wrong length record and parity check. The COBOL error bytes, if requested, are set to reflect the error condition and control is transferred to the USE AFTER STANDARD ERROR declarative sentence. The error block is made available at data-name-2 of the GIVING option, if specified.

If a parity error is detected when a block of records is read, the tape is backspaced and reread 100 times before control is returned to the programmer. If the error persists, the block is considered an error block and is added to the block count found in the DTF table.

2. Normal return (to the non-declarative portion) from a USE AFTER STANDARD ERROR declarative section is through the invoked IOCS subroutine. Thus, the next sequential block is brought into storage permitting continued processing of the file. (The error block is bypassed.) A return through the use of a GO TO statement does not bring the next block into storage; therefore, it is impossible to continue processing the file.

The processing of a sequential disk file opened as input is the same as the previous discussion of tape files, except that the disk block is reread ten times before being considered an error block.

COBOL cannot handle nested errors on sequential files. If errors occur within an error declarative, results are unpredictable.



VOLUME AND FILE LABEL HANDLING

TAPE LABELS

Among the several types of tape labels allowed under the Disk Operating System Virtual Storage are: volume labels, standard file labels, user standard labels, and nonstandard labels. Unlabeled files are also permitted. The description of each type of label follows.

Volume Labels

A volume label is used whenever standard file labels are used. Logical IOCS requires a volume label with VOL1 as its first four characters on every standard or user standard labeled file. VOL2-VOL8 are also allowed, but must be written and checked by the programmer.

Standard File Labels

A standard file label is an 80-character label created when an output file is opened or closed, in part by IOCS using the TLBL control statement. The first three characters are HDR (header), EO (end-of-volume), or EOF (end-of-file). The fourth character is a 1, indicating the first of a possible eight labels. The remainder of the label is formatted into fields describing the file. Labels 2 through 8 in this field are bypassed on input, and are not created on output under the Disk Operating System Virtual Storage.

The contents of the fields of a standard file label are described in "Appendix B: Standard Tape File Labels." The relationship between the TLBL statement and a standard file label is shown in Figures 39 and 40.

User Standard Labels

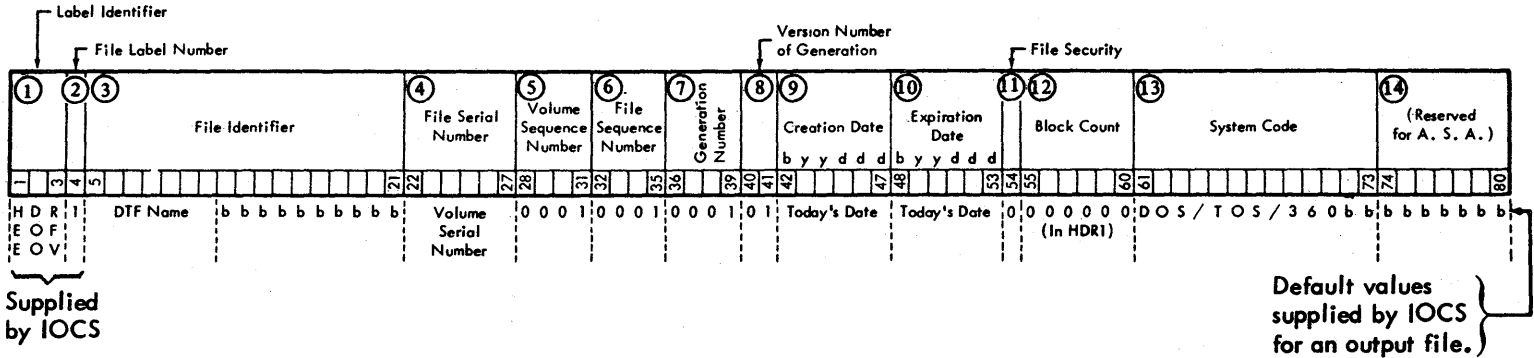
A user standard label is an 80-character label having UHL (user header label) or UTL (user trailer label) in the first three positions. The fourth position contains a number 1 through 8 which represents the relative position of the user label within a group of user labels. The contents of the remaining 76 positions are entirely up to the programmer. User labels, if present, follow HDR, EO, or EOF standard labels. On multivolume files, they may also appear at beginning-of-volume. User header labels are resequenced starting with one (UHL1) at the beginning of a new volume. Figure 41 shows the positioning of user labels on a file.

Nonstandard Labels

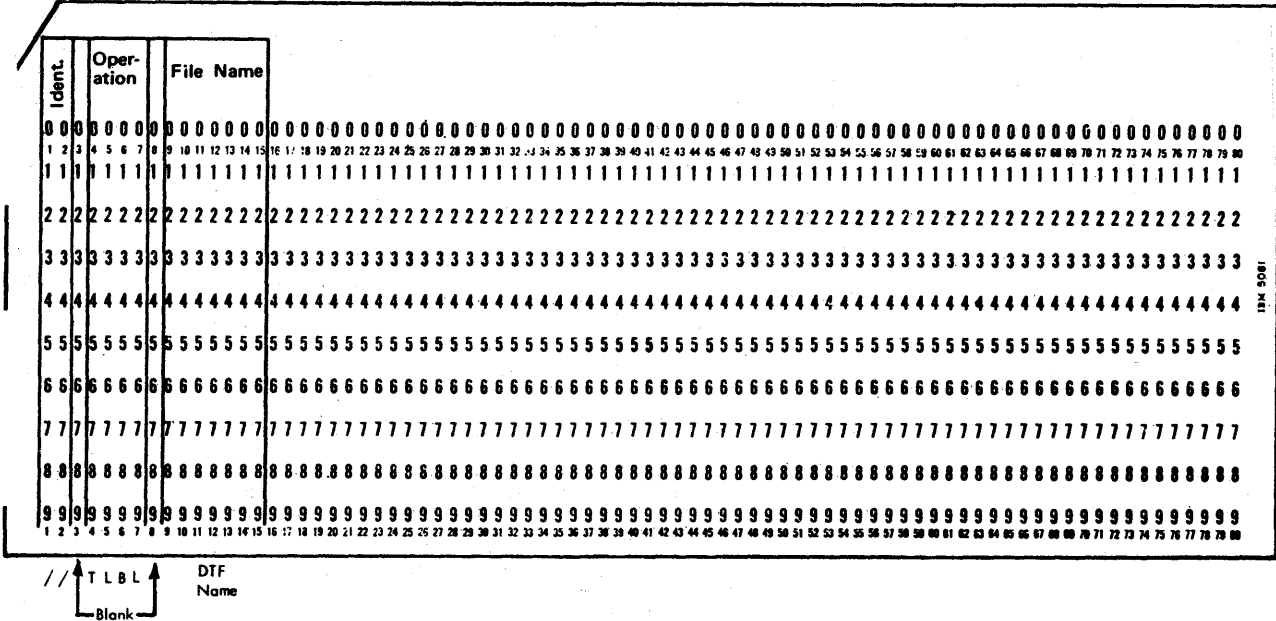
A nonstandard label may be any length. The contents of a nonstandard label is entirely programmer-dependent. It is the COBOL programmer's responsibility either to process or bypass nonstandard labels on input and to create them on output. Nonstandard label processing is not permitted on ASCII files. Figure 42 shows the positioning of nonstandard labels on a file.

Figure 40. Standard Tape File Label and TLBL Card (Showing Minimum Requirements)

Standard Tape File Label



Job Control TLBL Card



Header labels are written or read when the file is opened or when a volume switch occurs. Trailer labels are written when the physical end of the reel is reached, or when a CLOSE REEL or CLOSE file-name is issued. Trailer labels are read on each reel except the last when a tapemark is reached. For the last reel (i.e., EOF labels), trailer labels are not read until the file is closed.

For multivolume input files with nonstandard labels, the programmer must specify the integer-1 option of the source language ASSIGN clause, where integer-1 is the number of reels in the file. This number can be overridden at execution time by storing a nonzero integer in the special register NSTD-REELS before opening the file. The number of reels is then available to the programmer while the file is opened both in the special register NSTD-REELS and in the field reserved for this purpose which precedes the DTF table for DTFMT (see "DTF Tables" in this chapter). In addition, the number of reels remaining after each volume switch can also be found in the field reserved for this purpose which precedes the DTF table for DTFMT.

When processing a multivolume file with nonstandard labels (i.e., when the data-name option of the LABEL RECORDS clause is specified), if the programmer wishes to stop reading or writing before the physical end of a reel is reached, he must set a switch in the appropriate declarative section. In the Procedure Division, he can either CLOSE REEL or CLOSE FILE depending on the switch setting. Volume switching is done by LIOCS when CLOSE REEL is executed.

Sample Programs

Figure 43 illustrates the manner in which unlabeled input files on a multivolume volume are processed by a COBOL program. The input volume contains four files, only three of which are being used by the program. This unused file, which resides between the first and third file on the volume, must be bypassed during file processing. The program creates a single multivolume file with standard labels.

- ① All input files residing on the same volume are assigned to the same symbolic unit.

- ② The second file on the input reel is not used in this program and is bypassed through use of the POSITION option of the MULTIPLE FILE TAPE clause.
- ③ The first and second input files are closed by the execution of the CLOSE statement with the NO REWIND option, leaving the tape positioned in mid-reel for the next OPEN.
- ④ All volumes with the exception of the last volume of the multivolume output file are closed by a close statement with the REEL option. Volume switching is performed as noted in Step ⑧.
- ⑤ The second and third input files processed by the program are opened by an OPEN statement with the NO REWIND option.
- ⑥ At job completion, a standard CLOSE is issued to reposition the tapes of the closed files at their physical beginnings.
- ⑦ An LBLTYP control statement is included because a tape file requiring label information is to be processed.
- ⑧ Alternate assignments have been made for SYS011. Because these alternate assignments are in the sequence in which the ASSGN statements are submitted, the first volume of the output file will be on tape drive 282, the second on 283, and the third on 181. When the first CLOSE OUT-PUT REEL statement is executed, a standard EOJ label is written on the volume assigned to drive 282 and the reel is rewound and positioned at its physical beginning. The next WRITE RECO statement executed will then be written on the volume mounted on drive 283.
- ⑨ Although the file OUT-PUT consists of multiple volumes, only one TLBL control statement need be submitted.

Figure 44 is a sample program that illustrates the manner in which the multivolume file created in Figure 43 is read as an input file. The sample program also creates a multivolume file with standard labels.

- ① All output files residing on the same volume are assigned to the same symbolic unit.

The name field of the system-name in the ASSIGN clause is specified. This is the external-name by which the file

is known to the system. When specified, it is the name that appears in the filename field of the DLBL or TLBL job control statements.

- ② For the multivolume input file IN-PUT, the AT END option of the READ statement applies only to the last volume containing the EOF label. For prior volumes containing EOF labels, automatic volume switching will take place as indicated in the ASSGN control statements pertaining to the file IN-PUT.
- ③ The first and second file written on the volume are closed using the NO REWIND option of the CLOSE statement. This option leaves the tape positioned in mid-reel following the EOF label of the file just closed.
- ④ At job's completion, a standard CLOSE is issued to reposition the tapes of the closed files at their physical beginning.
- ⑤ A LBLTYP control statement is included because tape files requiring label information are being processed.
- ⑥ There are three TLBL control statements for the volume assigned to SYS013, one for each file referenced on the volume. The filename field of the TLBL control statements for these files contains the names used in the ASSIGN clauses of the COBOL source program, not the programmer logical unit name.
- ⑦ Alternate assignments have been made for SYS012 to handle the multiple volumes of the file IN-PUT.

Figure 45 illustrates the creation of an unlabeled multivolume file. The number of output volumes is determined dynamically during program execution. The program's input consists of the labeled multifile volume created in Figure 44.

- ① All input files residing on the same volume are assigned to the same symbolic unit.

The name field of the system-name of the ASSIGN clause is specified. These names will appear on the TLBL control statements that refer to these files.

The MULTIPLE FILE TAPE clause is not required for the multifile volume because each file is being processed in the sequence in which it appears on the reel. A rewind will not be executed for any file on the reel except for that processed last.

- ② The CLOSE statement for files IN-PUT-1 and IN-PUT-2, and the OPEN statement for files IN-PUT-2 and IN-PUT-3, use the NO REWIND option. This leaves the tape positioned in mid-reel for the multifile volume's next OPEN statement.
- ③ When it has been determined from the input data that a new output reel is required for the multivolume output file, a CLOSE OUT-PUT REEL statement is executed, processing is halted, and a message is issued to the operator which requests a new volume to be mounted.
- ④ At job's completion, a standard CLOSE is issued to reposition the tapes of the closed file at their physical beginning.
- ⑤ An LBLTYP control statement is included because tape files requiring label information are being processed.
- ⑥ There are three TLBL control statements for the volume assigned to SYS014, one for each file referenced on the volume. The filename field of the TLBL control statements for these files contains the names used in the ASSIGN clauses of the source program and not the programmer logical unit names.
- ⑦ Only one tape drive is assigned to the multivolume file OUT-PUT. Therefore, each time a volume is closed, processing must be halted and the operator informed to mount a new tape. This is illustrated in Step ③.

FILE
PROC

```

// JOB SAMPLE
* UNLABELED MULTIFILE VOLUME TO MULTIVOLUME FILE WITH STANDARD LABELS
// OPTION LOG,DUMP,LINK,LIST,LISTX,XREF,SYM,ERRS,NODECK
// EXEC FCOBOL

```

```

000010 IDENTIFICATION DIVISION.
000020 PROGRAM-ID. SAMPLE-1.
000030 ENVIRONMENT DIVISION.
000040 CONFIGURATION SECTION.
000050 SOURCE-COMPUTER. IBM-370.
000060 OBJECT-COMPUTER. IBM-370.
000070 INPUT-OUTPUT SECTION.
000080 FILE-CONTROL.
000090     SELECT INPUT1 ASSIGN TO SYS010-UT-3400-S-FILE1.
000100     SELECT INPUT2 ASSIGN TO SYS010-UT-3400-S-FILE2. } ①
000110     SELECT INPUT3 ASSIGN TO SYS010-UT-3400-S-FILE3.
000120     SELECT OUT-PUT ASSIGN TO SYS011-UT-3400-S.
000130 I-O-CONTROL.
000140     MULTIPLE FILE TAPE CONTAINS INPUT1 POSITION 1
000150                                     INPUT2 POSITION 3 } ②
000160                                     INPUT3 POSITION 4.
000170 DATA DIVISION.
000180 FILE SECTION.
000190 FD INPUT1
000200     RECORD CONTAINS 80 CHARACTERS
000210     LABEL RECORD IS OMITTED.
000220 01 REC1 PIC X(80).
000230 FD INPUT2
000240     RECORD CONTAINS 80 CHARACTERS
000250     LABEL RECORD IS OMITTED.
000260 01 REC2 PIC X(80).
000270 FD INPUT3
000280     RECORD CONTAINS 80 CHARACTERS
000290     LABEL RECORD IS OMITTED.
000300 01 REC3 PIC X(80).
000310 FD OUT-PUT
000320     RECORD CONTAINS 80 CHARACTERS
000330     BLOCK CONTAINS 3 RECORDS
000340     LABEL RECORD IS STANDARD.
000350 01 RECO PIC X(80).
000360 PROCEDURE DIVISION.
000370     OPEN INPUT INPUT1 OUTPUT OUT-PUT.
000380 READ1.
000390     READ INPUT1 INTO RECO AT END GO TO CLOSE1.
000400 A. WRITE RECO.
000410 B. GO TO READ1.
000420 CLOSE1.
000430     CLOSE INPUT1 WITH NO REWIND. ③
000440 C. CLOSE OUT-PUT REEL. ④
000450 D. OPEN INPUT INPUT2 WITH NO REWIND. ⑤
000460 READ2.
000470     READ INPUT2 INTO RECO AT END GO TO CLOSE2.
000480     PERFORM A.
000490     GO TO READ2.
000500 CLOSE2.
000510     CLOSE INPUT2 WITH NO REWIND. ③
000520     PERFORM C.
000530     OPEN INPUT INPUT3 WITH NO REWIND. ⑤

```

Figure 43. Processing an Unlabeled Multifile Volume (Part 1 of 2)

```
000540 READ3.  
000550     READ INPUT3 INTO RECO AT END GO TO CLOSE3.  
000560     PERFORM A.  
000570     GO TO READ3.  
000580 CLOSE3.  
000590     CLOSE INPUT3 OUT-PUT. (6)  
000600     STOP RUN.
```

```
// LBLTYP TAPE (7)  
// EXEC LNKEDT
```

```
// ASSGN SYS010,X'281'  
// ASSGN SYS011,X'282'  
// ASSGN SYS011,X'283',ALT}  
// ASSGN SYS011,X'181',ALT} (8)  
// TLEL SYS011,'MULTI-VOL FILE',99/214 (9)  
// EXEC
```

Figure 43. Processing an Unlabeled Multifile Volume (Part 2 of 2)

```
// JOB SAMPLE
* LABELED MULTIVOLUME FILE TO LABELED MULTIFILE VOLUME
// OPTION LOG,DUMP,LINK,LIST,LISTX,XREF,SYM,ERRS,NODECK
// EXEC FCOBOL
```

```
000010 IDENTIFICATION DIVISION.
000020 PROGRAM-ID. SAMPLE-2.
000030 ENVIRONMENT DIVISION.
000040 CONFIGURATION SECTION.
000050 SOURCE-COMPUTER. IBM-370.
000060 OBJECT-COMPUTER. IBM-370.
000070 INPUT-OUTPUT SECTION.
000080 FILE-CONTROL.
000090     SELECT IN-PUT ASSIGN TO SYS012-UT-3400-S.
000100     SELECT OUT-PUT1 ASSIGN TO SYS013-UT-3400-S-FILE1.
000110     SELECT OUT-PUT2 ASSIGN TO SYS013-UT-3400-S-FILE2.
000120     SELECT OUT-PUT3 ASSIGN TO SYS013-UT-3400-S-FILE3.
000130 DATA DIVISION.
000140 FILE SECTION.
000150 FD IN-PUT.
000160     RECORD CONTAINS 80 CHARACTERS
000170     BLOCK CONTAINS 3 RECORDS
000180     LABEL RECORD IS STANDARD.
000190 01 IN-REC.
000200     05 FILLER PIC X(4) .
000210     05 CODA PIC X.
000220     05 FILLER PIC X(6) .
000230     05 CODB PIC X.
000240         88 SW-FIL1 VALUE "9".
000250         88 SW-FIL2 VALUE "8".
000260     05 FILLER PIC X(68) .
000270 FD OUT-PUT1
000280     RECORD CONTAINS 80 CHARACTERS
000290     BLOCK CONTAINS 3 RECORDS
000300     LABEL RECORD IS STANDARD.
000310 01 OUT-REC1 PIC X(80) .
000320 FD OUT-PUT2
000330     RECORD CONTAINS 80 CHARACTERS
000340     BLOCK CONTAINS 3 RECORDS
000350     LABEL RECORD IS STANDARD.
000360 01 OUT-REC2 PIC X(80) .
000370 FD OUT-PUT3
000380     RECORD CONTAINS 80 CHARACTERS
000390     BLOCK CONTAINS 3 RECORDS
000400     LABEL RECORD IS STANDARD.
000410 01 OUT-REC3 PIC X(80) .
000420 WORKING-STORAGE SECTION.
000430 77 TAPE-NUMBER PIC 9 VALUE 0.
000440 PROCEDURE DIVISION.
000450     OPEN INPUT IN-PUT OUTPUT OUT-PUT1.
```

①

Figure 44. Reading a Multivolume File with Standard Labels; Creating a Multifile Volume with Standard Labels (Part 1 of 2)

```

000460 READ-IN.
000470 READ IN-PUT AT END GO TO END-OF-JOB. (2)
000480 A. MOVE IN-REC TO OUT-REC1.
000490 WRITE OUT-REC1.
000500 IF SW-FIL1 NEXT SENTENCE ELSE GO TO READ-IN.
000510 CLOSE OUT-PUT1 WITH NO REWIND. (3)
000520 OPEN OUTPUT OUT-PUT2.
000530 ADD 1 TO TAPE-NUMBER.
000540 B. PERFORM READ-IN.
000550 MOVE IN-REC TO OUT-REC2.
000560 WRITE OUT-REC2.
000570 IF SW-FIL2 NEXT SENTENCE ELSE GO TO B.
000580 CLOSE OUT-PUT2 WITH NO REWIND. (3)
000590 OPEN OUTPUT OUT-PUT3.
000600 ADD 1 TO TAPE-NUMBER.
000610 C. PERFORM READ-IN.
000620 MOVE IN-REC TO OUT-REC3.
000630 WRITE OUT-REC3.
000640 GO TO C.
000650 END-OF-JOB.
000660 CLOSE IN-PUT.
000670 IF TAPE-NUMBER = 0 CLOSE OUT-PUT1 GO TO D.
000680 IF TAPE-NUMBER = 1 CLOSE OUT-PUT2 ELSE CLOSE OUT-PUT3. (4)
000690 D. STOP RUN.

```

```

// LBLTYP TAPE (5)
// EXEC LNKEDT

```

```

// ASSGN SYS018,X'283'
// TLBL FILE1,'MULTI-FILE1 VOL' } (6)
// TLBL FILE2,'MULTI-FILE2 VOL' }
// TLBL FILE3,'MULTI-FILE3 VOL' }
// ASSGN SYS012,X'281'
// ASSGN SYS012,X'282',ALT } (7)
// ASSGN SYS012,X'181',ALT }
// TLBL SYS012,'MULTI-VOL FILE'
// EXEC

```

Figure 44. Reading a Multivolume File with Standard Labels; Creating a Multifile Volume with Standard Labels (Part 2 of 2)

```

// JOB SAMPLE
* LABELLED MULTIFILE VOLUME TO UNLABELED MULTIVOLUME FILE
// OPTION LOG,DUMP,LINK,LIST,LISTX,XREF,SYM,ERRS,NODECK
// EXEC FCOBOL

```

```

000010 IDENTIFICATION DIVISION.
000020 PROGRAM-ID. SAMPLE-3.
000030 ENVIRONMENT DIVISION.
000040 CONFIGURATION SECTION.
000050 SOURCE-COMPUTER. IBM-370.
000060 OBJECT-COMPUTER. IBM-370.
000070 INPUT-OUTPUT SECTION.
000080 FILE-CONTROL.
000090     SELECT IN-PUT-1 ASSIGN TO SYS014-UT-3400-S-FILE1.
000100     SELECT IN-PUT-2 ASSIGN TO SYS014-UT-3400-S-FILE2.
000110     SELECT IN-PUT-3 ASSIGN TO SYS014-UT-3400-S-FILE3.
000120     SELECT OUT-PUT ASSIGN TO SYS015-UT-3400-S.
000130 DATA DIVISION.
000140 FILE SECTION.
000150 FD IN-PUT-1
000160     RECORD CONTAINS 80 CHARACTERS
000170     BLOCK CONTAINS 3 RECORDS
000180     LABEL RECORD IS STANDARD.
000190 01 IN-REC1 PIC X(80).
000200 FD IN-PUT-2
000210     RECORD CONTAINS 80 CHARACTERS
000220     BLOCK CONTAINS 3 RECORDS
000230     LABEL RECORD IS STANDARD.
000240 01 IN-REC2 PIC X(80).
000250 FD IN-PUT-3
000260     RECORD CONTAINS 80 CHARACTERS
000270     BLOCK CONTAINS 3 RECORDS
000280     LABEL RECORD IS STANDARD.
000290 01 IN-REC3 PIC X(80).
000300 FD OUT-PUT
000310     RECORD CONTAINS 80 CHARACTERS
000320     BLOCK CONTAINS 3 RECORDS
000330     LABEL RECORD IS OMITTED.
000340 01 OUT-REC.
000350     05 FILLER PIC X(4).
000360     05 CODA PIC X.
000370         88 HI VALUE "9".
000380     05 FILLER PIC X(6).
000390     05 CODB PIC X.
000400         88 LO VALUE "8".
000410     05 FILLER PIC X(68).
000420 PROCEDURE DIVISION.
000430     OPEN INPUT IN-PUT-1 OUTPUT OUT-PUT.
000440 IN-1.
000450     READ IN-PUT-1 INTO OUT-REC AT END GO TO CLOSE1.
000460 TESTER.
000470     IF HI AND LO PERFORM CLOSE-OUT ELSE WRITE OUT-REC.
000480 A. GO TO IN-1.
000490 CLOSE1.
000500     CLOSE IN-PUT-1 WITH NO REWIND.
000510     OPEN INPUT IN-PUT-2 WITH NO REWIND.
000520 IN-2.
000530     READ IN-PUT-2 INTO OUT-REC AT END GO TO CLOSE2.
000540     PERFORM TESTER.
000550     GO TO IN-2.

```

Figure 45. Creating an Unlabeled Multivolume File (Part 1 of 2)

```

000560 CLOSE2.
000570     CLOSE IN-PUT-2 WITH NO REWIND.
000580     OPEN INPUT IN-PUT-3 WITH NO REWIND. } 2
000590 IN-3.
000600     READ IN-PUT-3 INTO OUT-REC AT END GO TO CLOSE3.
000610     PERFORM TESTER.
000620     GO TO IN-3.
000630 CLOSE-OUT.
000640     CLOSP OUT-PUT REEL.
000650     STOP "REMOVE TAPE ON SYS015 AND MOUNT NEW TAPE". } 3
000660 CLOSE3.
000670     CLOSE IN-PUT-3 OUT-PUT. 4
000680     STOP RUN.

```

```

// LBLTYP TAPE 5
// EXEC LNKEDT

```

```

// ASSGN SYS014,X'283'
// TLBL FILE1,'MULTI-FILE1 VOL'
// TLBL FILE2,'MULTI-FILE2 VOL' } 6
// TLBL FILE3,'MULTI-FILE3 VOL'
// ASSGN SYS015,X'282' 7
// EXEC

```

Figure 45. Creating an Unlabeled Multivolume File (Part 2 of 2)

MASS STORAGE FILE LABELS

The IBM Disk Operating System/Virtual Storage provides positive identification and protection of all files on mass storage devices by recording labels on each volume. These labels ensure that the correct volume is used for input, and that no current information is destroyed on output.

The mass storage labels always include one volume label for each volume and one or more file labels for each logical file on the volume. There may also be user header labels and user trailer labels.

Volume Labels

The volume label is an 80-byte data field preceded by a 4-byte key field. Both the key field and the first four bytes of the data field contain the label identifier VOL1. IOCS creates a standard volume label for every volume processed by the Disk Operating System/Virtual Storage. It is always the third record on cylinder 0, track 0. The format and contents of a standard volume label can be found in the publication DOS/VS Disk Labels.

Standard File Labels

A standard file label identifies a particular logical file, gives its location(s) on the mass storage device, and contains information to prevent premature destruction of current files. A standard file label for a file located on a mass storage device is a 140-character label created (OPEN/CLOSE OUTPUT) in part by IOCS using the DLBL control statement. The fields contained within the label follow three standard formats.

1. Format 1 is used for all logical files. The contents of the fields of a Format 1 label is discussed in "Appendix C: Standard Mass Storage Device Labels."
2. Format 2 is required for indexed files. The contents of the fields of a Format 2 label can be found in the publication DOS/VS Disk Labels.
3. Format 3 is required if a logical file uses more than three extents of any volume. The contents of the fields of a Format 3 label can be found in the publication DOS/VS Disk Labels.

User Labels

The programmer can include additional labels to further define his file. The labels are referred to as user standard labels. They cannot be specified for indexed files. A user label is an 80-character label containing UHL (user header label) or UTL (user trailer label) in the first three character positions. The fourth position contains a number 1 through 8 which represents the relative position of the user label with a group of user labels. The contents of the remaining 76 positions is entirely up to the programmer. User header and trailer labels are written on the first track of the first extent of each volume allocated by the programmer for the file. User header labels are resequenced starting with one (UHL1) at the beginning of each new volume.

LABEL PROCESSING CONSIDERATIONS

Files on Mass Storage Device Opened as Input

1. Standard labels checked
 - a. The volume serial numbers in the volume labels are compared to the file serial numbers in the EXTENT card.
 - b. Fields 1 through 3 in Format 1 label are compared to the corresponding fields in the DLBL card.
 - c. Each of the extent definitions in the Format 1 and Format 3 labels is checked against the limit fields supplied in the EXTENT card.
2. User labels checked
 - a. If user header labels are indicated for directly or sequentially organized files, they are read as each volume of the file is opened. After reading each label, the OPEN routine branches to the programmer's label routine if the appropriate USE AFTER STANDARD LABEL PROCEDURE declarative is specified in the source program. The LABEL RECORDS clause with the data-name option must be specified in the Data Division. The programmer's label routine then performs any processing required.

- b. If user trailer labels are indicated on a sequential file, they are read after reaching the end of the last extent on each volume when the file is closed, provided end-of-file has been reached. Trailer labels are processed by the programmer's label routine if the appropriate USE AFTER STANDARD LABEL PROCEDURE declarative is specified in the source program. The LABEL RECORDS clause with the data-name option must be specified in the Data Division.

2. User header labels created

- a. If user header labels are indicated by the presence of the appropriate USE AFTER STANDARD LABEL PROCEDURE declarative and the LABEL RECORDS clause with the data-name option, the programmer's label routine is entered to furnish the labels as each volume of the file is opened. This can be done for as many as eight user header labels per volume. As each label is presented, IOCS writes it out on the first track of the first extent of the volume.
- b. If user trailer labels are indicated by the presence of the appropriate USE AFTER STANDARD LABEL PROCEDURE declarative and the LABEL RECORDS clause with the data-name option, the programmer's label routine is entered to furnish the labels when the end of the last extent on each volume is reached. This can be done for as many as eight user trailer labels. The CLOSE statement must be issued to create trailer labels for the last volume of a sequential file or for a direct file.

Files on Mass Storage Devices Opened as Output

1. Standard labels created

- a. The volume serial numbers in the volume labels are compared to the file serial numbers in the EXTENT card.
- b. The extent definitions in all current labels on the volume are checked to determine whether any extend into those defined in the EXTENT card. If any overlap, the expiration date is checked against the current date in the Communication Region of the Supervisor. If the expiration date has passed, the old labels are deleted. If not, the operator is notified of the condition.
- c. The new Format 1 label is written with information supplied in the DLBL card. If an indexed file is being processed, the DTFIS routine supplies information for the Format 2 label.
- d. The information in the EXTENT card is placed in the Format 1 labels and, if necessary, in the additional Format 3 labels.

UNLABELED FILES

When a multivolume tape file is opened as INPUT and integer as specified in the ASSIGN clause is greater than 1, the compiler will generate the following message to the operator:

```
C126D IS IT EOF?
```

The operator must respond either with N if it is not the last reel, or with Y if it is the last reel. If it is end-of-file, control passes to the imperative-statement specified in the AT END phrase of the READ statement; if it is not end-of-file, processing of the next volume is initiated.

If the integer specified in the ASSIGN clause is not greater than 1, control always passes at end-of-volume to the imperative-statement specified in the AT END phrase of the READ statement.

FILE
PROC

The IBM DOS/VS COBOL Compiler and Library support the American National Standard Code for Information Interchange (ASCII) as well as EBCDIC. This support allows the user at object time to accept and create magnetic tapes in accordance with all of the following standards:

- American National Standard Code for Information Interchange, X3.4-1968.
- American National Standard Magnetic Tape Labels for Information Interchange, X3.27-1969.
- American National Standard Recorded Magnetic Tape for Information Interchange (800 CPI, NRZI), X3.22-1967.

COBOL LANGUAGE CONSIDERATIONS

The ASCII feature is supported by the following addition to IEM's implementation of American National Standard COBOL:

The system-name specified in the ASSIGN clause is now coded as

SYSnnn-UT-device-C[-buffer offset][-name]

where

Organization code C indicates that an ASCII-encoded tape file is to be processed.

Buffer offset is a two-character field that serves to indicate the size of the block prefix. A block prefix, if present, precedes each physical record and is not accessible to the COBOL programmer. This entry may only be present for ASCII tape files and is only required if a non-zero block prefix exists. For output files, buffer offset may be specified as 00 for F, U, or D-mode records, or as 04 for D-mode records only. A buffer offset of 04 on output means that the block prefix will contain the length of each physical record. For input files, buffer offset may be in the range 00 through 99.

FILE HANDLING

In processing ASCII files, the supported record formats are fixed, undefined, and variable. A variable-length record on an ASCII file is known as a D-format record. ASCII support does not extend to spanned records. Record formats are discussed in detail in the chapter "Record Formats."

For an ASCII file that contains a buffer offset field, the following considerations apply:

- If the BLOCK CONTAINS clause with the RECORDS option is specified, or if the BLOCK CONTAINS clause is omitted, the compiler compensates for the buffer offset field.
- If the BLOCK CONTAINS clause with the CHARACTERS option is specified, the programmer must include the buffer offset as part of the physical record.

Labels on ASCII files are processed as are the existing DOS/VS standard and user standard labels.

Nonstandard label procedures, however, are not supported. Therefore, USE BEFORE STANDARD LABEL PROCEDURES are not permitted for ASCII files. ASCII files on unlabeled tapes are supported. These unlabeled tapes may contain data in any of the supported record formats. A complete discussion of tape file labels can be found in the chapter "Advanced Processing Capabilities."

The ASCII option (organization code C in the ASSIGN clause) must not be specified for a file on which checkpoints are to be written.

Diagnostic messages associated with ASCII file handling are provided. At compile time, E-level messages are issued for files whose record descriptions contain data formats that are inconsistent with ASCII conversion. At object time, a message is issued if an invalid sign configuration is present during translation, and the job will be terminated.



OPERATIONAL CONSIDERATIONS

It should be noted that ASCII support causes translation from ASCII to EBCDIC on input and from EBCDIC to ASCII on output. Translation occurs automatically and is transparent to the COBOL programmer. Since an ASCII file is assumed to contain only ASCII characters, standard character substitution occurs when untranslatable configurations are present. The character X'1A' is substituted for invalid EBCDIC configurations during translation. An invalid ASCII configuration (high-order bit on) translates to the character X'3F'.

OBTAINING AN ASCII COLLATING SEQUENCE ON A SORT

If an ASCII collated sort is desired or numeric sort keys contain a sign in the

form of a leading overpunch or separate character, a Program Product IBM DOS/VS Sort/Merge program must be used. If sort files reside on a 3330 or 3340 device, the Sort program that supports these devices is required. The Program Product IBM DOS/VS Tape and Disk Sort/Merge, Program Number 5746-SM1 is designed specifically for use with a DOS/VS system.

To obtain an ASCII collated sort, the system-name in the ASSIGN clause for the sort work files should contain a C in the organization field. The class field may be specified as either UT or DA. (Since ASCII support causes translation from ASCII to EBCDIC on input, sort work files are not restricted to tapes.)

Note that for an ASCII collated sort, the buffer offset field is not permitted.

The ASCII collating sequence is listed in the publication IBM DOS Full American National Standard COBOL.

Logical records for files which are not VSAM files may be in one of four formats: fixed-length (format F), variable-length (format V), undefined (format U), or spanned (format S). All of these formats are not supported for all access methods. F-mode files must contain records of equal lengths. Files containing records of unequal lengths must be V-mode, S-mode, or U-mode. Files containing logical records that are longer than physical records must be S-mode.

The record format is specified in the RECORDING MODE clause in the Data Division. If this clause is omitted, the compiler determines the record format from the record descriptions associated with the file. If the file is to be blocked, the BLOCK CONTAINS clause must be specified in the Data Division.

The prime consideration in the selection of a record format is the nature of the file itself. The programmer knows the type of input his program will receive and the type of output it will produce. The selection of a record format is based on this knowledge as well as an understanding of the type of input/output devices on which the file is written and of the access method used to read or write the file.

Coding considerations for non-fixed length records are discussed in the chapter "Table Handling Considerations."

FIXED-LENGTH (FORMAT F) RECORDS

Format F records are fixed-length records. The programmer specifies format F records by including RECORDING MODE IS F in the file description entry in the Data Division. If the clause is omitted and both of the following are true:

- All records in the file are the same size
- BLOCK CONTAINS [integer-1 TO] integer-2... does not specify integer-2 less than the length of the maximum level-01 record

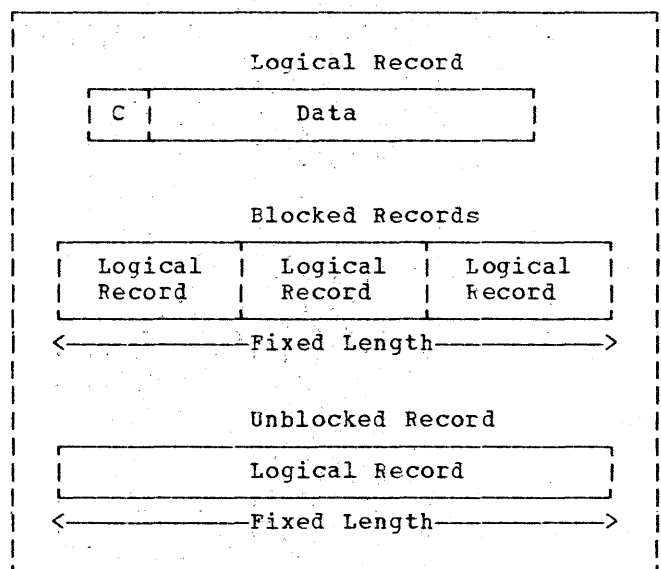
the compiler determines the recording mode to be F. All records in the file are the same size if there is only one record description associated with the file and it contains no OCCURS clause with the

DEPENDENT ON option, or if multiple record descriptions are all the same length.

The number of logical records within a block (blocking factor) is normally constant for every block in the file. When fixed-length records are blocked, the programmer specifies the BLOCK CONTAINS clause in the file description entry in the Data Division.

In unblocked format F, the logical record constitutes the block. The BLOCK CONTAINS clause is unnecessary for unblocked records.

Format F records are shown in Figure 46. The optional control character, represented by C, is used for stacker selection and carriage control. When carriage control or stacker selection is desired, the WRITE statement with the ADVANCING or POSITIONING option is used to write records on the output file. In this case one character position must be included as the first character of the record. This position will be automatically filled in with the carriage control or stacker select character. The carriage control character never appears when the file is written on the printer or punched on the card punch.



REC
FMTS

Figure 46. Fixed-Length (Format F) Records

UNDEFINED (FORMAT U) RECORDS

Format U is provided to permit the processing of any blocks that do not conform to F or V formats. Format U records are shown in Figure 47. The optional control character C, as discussed under "Fixed-Length (Format F) Records," may be used in each logical record.

The programmer specifies format U records by including RECORDING MODE IS U in the file description entry in the Data Division. U-mode records may be specified only for directly organized or standard sequential files.

If the RECORDING MODE clause is omitted, and BLOCK CONTAINS [integer-1 TO] integer-2... does not specify integer-2 less than the maximum level-01 record, the compiler determines the recording mode to be U if the file is directly organized and one of the following conditions exist:

- The FD entry contains two or more level-01 descriptions of different lengths.
- A record description contains an OCCURS clause with the DEPENDING ON option.
- A RECORD CONTAINS clause specifies a range of record lengths.

Each block on the external storage media is treated as a logical record. There are no record-length or block-length fields.

Note: When a READ INTO statement is used for a U-mode file, the size of the longest record for that file is used in the MOVE statement. All other rules of the MOVE statement apply.

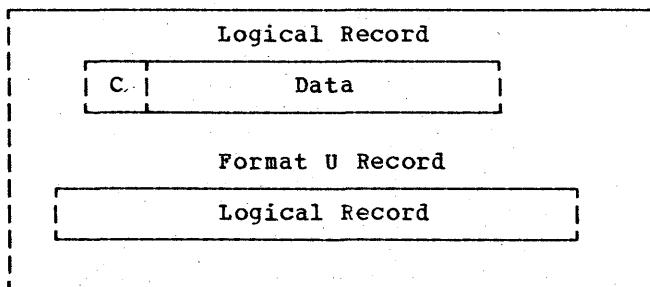


Figure 47. Undefined (Format U) Records

VARIABLE-LENGTH RECORDS

There are two types of variable-length record: D-format and V-format. A D-format record is a variable-length record on an

ASCII tape file. D-format records are processed in the same manner as V-format records on tape files.

The programmer specifies format V records by including RECORDING MODE IS V in the file description entry in the Data Division. V-mode records may only be specified for standard sequential files. If the RECORDING MODE clause is omitted and BLOCK CONTAINS [integer-1 TO] integer-2... does not specify integer-2 less than the maximum level-01 record, the compiler determines the recording mode to be V if the file is standard sequential and one of the following conditions exists:

- The FD entry contains two or more level 01 descriptions of different lengths.
- A record description contains an OCCURS clause with the DEPENDING ON option.
- A RECORD CONTAINS clause specifies a range of record lengths.

V-mode records, unlike U-mode or F-mode records, are preceded by fields containing control information. These control fields are illustrated in Figures 48 and 49.

The first four bytes of each block contain control information (CC):

- LL -- represents two bytes designating the length of the block (including the 'CC' field).
- BB -- represents two bytes reserved for system use.

The first four bytes of each logical record contain control information (cc):

- ll -- represents two bytes designating the logical record length (including the 'cc' field).
- bb -- represents two bytes reserved for system use.

For unblocked V mode records (see Figure 45) the data portion + CC + cc constitute the block.

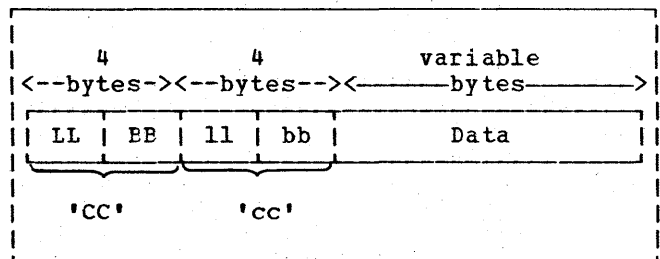


Figure 48. Unblocked V-Mode Records

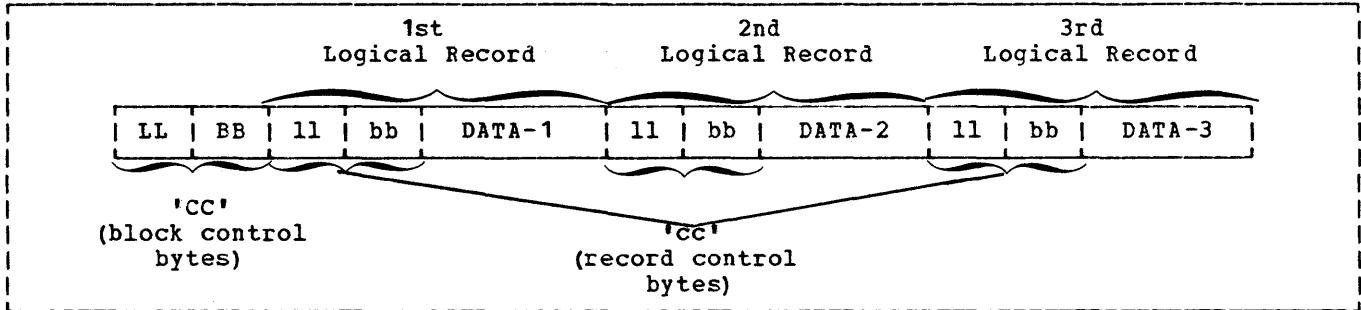


Figure 49. Blocked V-Mode Records

For blocked V-mode records (see Figure 49) the data portion of each record + the cc of each record + CC constitute the block.

The control bytes are automatically provided when the file is written and are not communicated to the programmer when the file is read. Although they do not appear in the description of the logical record provided by the programmer, the compiler will allocate input and output buffers which are large enough to accommodate them. When variable-length records are written on unit record devices, control bytes are neither printed nor punched. They appear, however, on other external storage devices as well as in buffer areas of storage. V-mode records moved from an input buffer to a working-storage area will be moved without the control bytes.

Note: When a READ INTO statement is used for a V-mode file, the size of the longest record for that file is used in the MOVE statement. All other rules of the MOVE statement apply.

Example 1:

Consider the following standard sequential file consisting of unblocked V-mode records:

```

FD VARIABLE-FILE-1
  RECORDING MODE IS V
  BLOCK CONTAINS 35 TO 80 CHARACTERS
  RECORD CONTAINS 27 TO 72 CHARACTERS
  DATA RECORD IS VARIABLE-RECORD-1
  LABEL RECORDS ARE STANDARD.

01 VARIABLE-RECORD-1.
  05 FIELD-A PIC X(20).
  05 FIELD-B PIC 99.
  05 FIELD-C OCCURS 1 TO 10 TIMES
    DEPENDING ON
    FIELD-B PIC 9(5).

```

The LABEL RECORDS clause is always required. The DATA RECORD(S) clause is never required. If the RECORDING MODE clause is omitted, the compiler determines the mode as V since the record associated with VARIABLE-FILE-1 varies in length depending on the contents of FIELD-B. The RECORD CONTAINS clause is never required. The compiler determines record sizes from the record description entries. Record length calculations are affected by the following:

- When the BLOCK CONTAINS clause with the RECORDS option is used, the compiler adds four bytes to the logical record length and four more bytes to the block length.
- When the BLOCK CONTAINS clause with the CHARACTERS option is used, the programmer must include each cc + CC in the length calculation (see Figure 45). In the definition of VARIABLE-FILE-1, the BLOCK CONTAINS clause specifies 8 more bytes than does the record contains clause. Four of these bytes are the logical record control bytes and the other four are the block control bytes.

Assuming that FIELD-B contains the value 02 for the first record of a file and FIELD-B contains the value 03 for the second record of the file, the first two records will appear on an external storage device and in buffer areas of storage as shown in Figure 50.

If the file described in Example 1 had a blocking factor of 2, the first two records would appear on an external storage medium as shown in Figure 51.



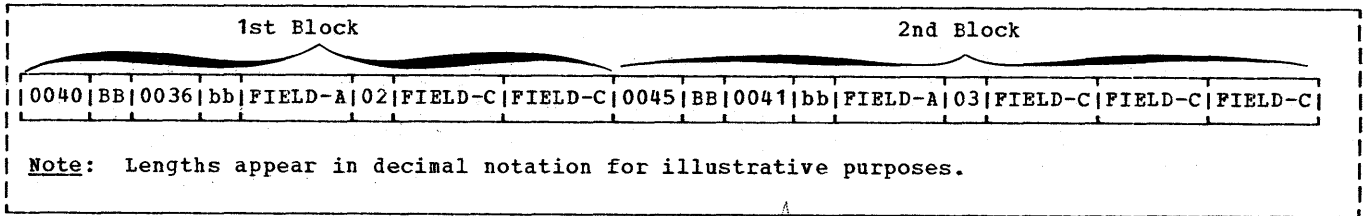


Figure 50. Fields in Unblocked V-Mode Records

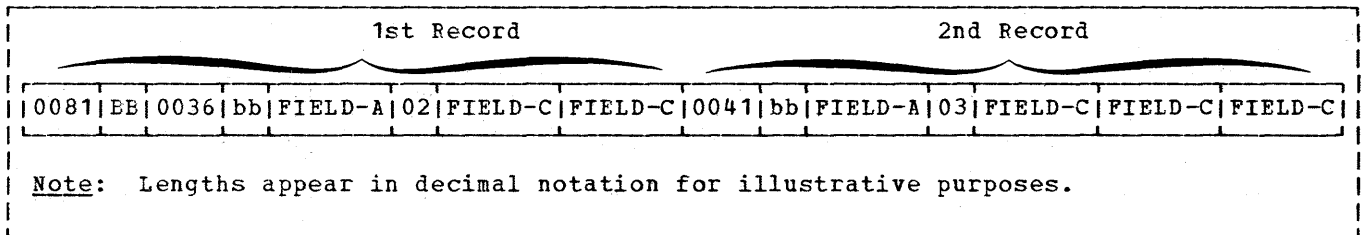


Figure 51. Fields in Blocked V-Mode Records

Example 2:

If VARIABLE-FILE-2 is blocked, with space allocated for three records of maximum size per block, the following FD entry could be used when the file is created:

```

FD VARIABLE-FILE-2
  RECORDING MODE IS V
  BLOCK CONTAINS 3 RECORDS
  RECORD CONTAINS 20 TO 100 CHARACTERS
  DATA RECORDS ARE VARIABLE-RECORD-1,
  VARIABLE-RECORD-2
  LABEL RECORDS ARE STANDARD.

01 VARIABLE-RECORD-1.
   05 FIELD-A PIC X(20).
   05 FIELD-B PIC X(80).

01 VARIABLE-RECORD-2.
   05 FIELD-X PIC X(20).
  
```

As mentioned previously, the RECORDING MODE, RECORD CONTAINS, and DATA RECORDS clauses are unnecessary. By specifying

that each block contains three records, the programmer allows the compiler to provide space for three records of maximum size plus additional space for the required control bytes. Hence, 316 character positions are reserved by the compiler for each output buffer. If this size is other than the maximum, the BLOCK CONTAINS clause with the CHARACTERS option should be specified.

Assuming that the first six records written are five 100-character records followed by one 20-character record, the first two blocks of VARIABLE-FILE-2 will appear on the external storage device as shown in Figure 52.

The buffer for the second block is truncated after the sixth WRITE statement is executed since there is not enough space left for a maximum size record. Hence, even if the seventh WRITE to VARIABLE-FILE-2 is a 20-character record, it will appear as the first record in the third block. This situation can be avoided by using the APPLY WRITE-ONLY clause when creating files of variable-length blocked records.

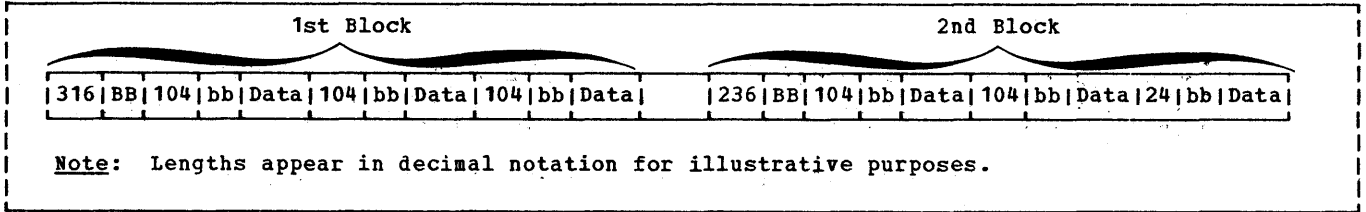


Figure 52. First Two Blocks of VARIABLE-FILE-2

APPLY WRITE-ONLY Clause

The APPLY WRITE-ONLY clause is used to make optimum use of buffer and external storage space when creating a standard sequential file with blocked V-mode records.

Suppose VARIABLE-FILE-2 is being created with the following PD entry:

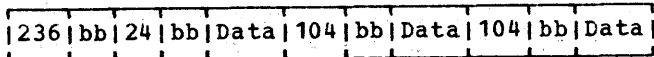
```

FD VARIABLE-FILE-2
  RECORDING MODE IS V
  BLOCK CONTAINS 316 CHARACTERS
  RECORD CONTAINS 20 TO 100 CHARACTERS
  DATA RECORDS ARE VARIABLE-RECORD-1,
  VARIABLE-RECORD-2
  LABEL RECORDS ARE STANDARD.

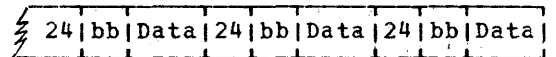
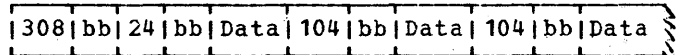
01 VARIABLE-RECORD-1.
  05 FIELD-A PIC X(20).
  05 FIELD-B PIC X(80).

01 VARIABLE-RECORD-2.
  05 FIELD-X PIC X(20).
  
```

The first three WRITE statements to the file create one 20-character record followed by two 100-character records. Without the APPLY WRITE-ONLY clause, the buffer is truncated after the third WRITE statement is executed, since the maximum size record no longer fits. The block is written as shown below:



Using the APPLY WRITE-ONLY clause will cause a buffer to be truncated only when the next record does not fit in the buffer. That is, if the next three WRITE statements to the file specify VARIABLE-RECORD-2, the block will be created containing six logical records, as shown below:



Note: When using the APPLY WRITE-ONLY clause, records must not be constructed in buffer areas. An intermediate work area must be used with a WRITE FROM statement.

SPANNED (FORMAT S) RECORDS

A spanned record is a logical record that may be contained in one or more physical blocks. Format S records may be specified for direct files and for standard sequential files assigned to magnetic tape or to mass storage devices.

When creating files with S-mode records, if a record is larger than the remaining space in a block, a segment of the record is written to fill the block. The remainder of the record is stored in the next block or blocks, as required.

When retrieving a file with S-mode records, only complete records are made available to the programmer.

Spanned records are preceded by fields containing control information. Figure 53 illustrates the control fields.

BDF (Block Descriptor Field):

- LL -- represents 2 bytes designating the length of the physical block (including the block descriptor field itself).
- BB -- represents 2 bytes reserved for system use.

REC
FMTS

SDF (Segment Descriptor Field):

- ll -- represents 2 bytes designating the length of the record segment (including the segment descriptor field itself).
- bb -- represents 2 bytes reserved for system use.

Note: There is only one block descriptor field at the beginning of each physical block. There is, however, one segment descriptor field for each record segment within the block.

Each segment of a record in a block, even if it is the entire record, is preceded by a segment descriptor field. The segment descriptor field also indicates whether the segment is the first, the last, or an intermediate segment. Each block includes a block descriptor field. These fields are not described in the Data Division; provision is automatically made for them. These fields are not available to the programmer.

A spanned blocked file may be described as a file composed of physical blocks of fixed length established by the programmer. The logical records may be either fixed or variable in length and that size may be smaller, equal to, or larger than the physical block size. There are no required relationships between logical records and physical block sizes.

A spanned unblocked file may be described as a file composed of physical blocks each containing one logical record or one segment of a logical record. The logical records may be either fixed or variable in length. When the physical block contains one logical record, the length of the block is determined by the logical record size. When a logical record has to be segmented, the system always writes the largest physical block possible. The system segments the logical record when the entire logical record cannot fit on the track.

Figure 54 is an illustration of blocked spanned records of SFILE. SFILE is described in the Data Division with the following file description entry:

```

FD SFILE
RECORD CONTAINS 250 CHARACTERS
BLOCK CONTAINS 100 CHARACTERS
.
.
.
    
```

Figure 54 also illustrates the concept of record segments. Note that the third block contains the last 50 bytes of REC-1 and the first 50 bytes of REC-2. Such portions of logical records are called record segments. It is therefore correct to say that the third block contains the last segment of REC-1 and the first segment of REC-2. The first block contains the first segment of REC-1 and the second block contains an intermediate segment of REC-1.

S-MODE CAPABILITIES

Formatting a file in the S-mode allows the programmer to make the most efficient use of external storage while organizing data files with logical record lengths most suited to his needs.

1. Physical record lengths can be designated in such a manner as to make the most efficient use of track capacities on mass storage devices.
2. The programmer is not required to adjust logical record lengths to maximum physical record lengths and their device-dependent variants when designing his data files.
3. The programmer has greater flexibility in transferring logical records across DASD types.

Spanned record processing will support the 2400, 3410, 3420 tape series, the 2311, 2314, 2319, 3330, and 3340 disk storage devices, and the 2321 data cell drive.

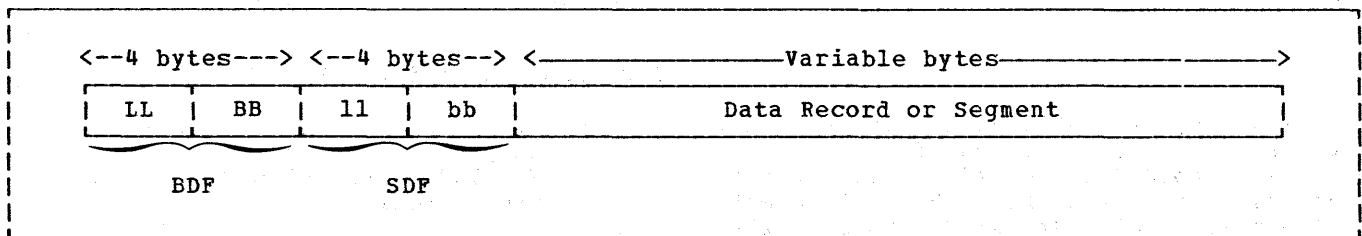


Figure 53. Control Fields of an S-Mode Record

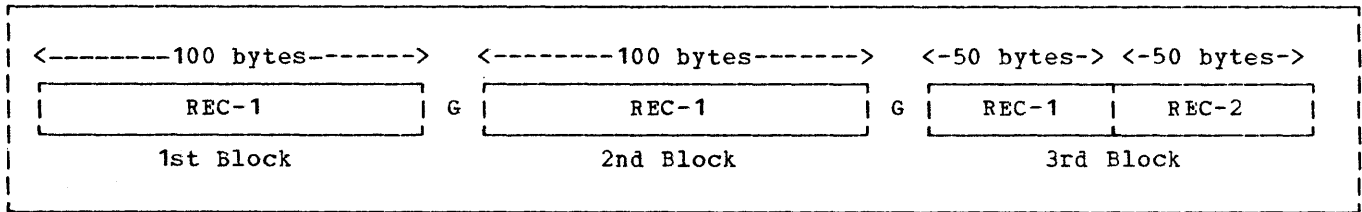


Figure 54. One Logical Record Spanning Physical Blocks

SEQUENTIALLY ORGANIZED S-MODE FILES ON TAPE OR MASS STORAGE DEVICES

When the spanned format is used for DTFMT or DTFSD files, the logical records may be either fixed or variable in length and are completely independent of physical record length. A logical record may span physical records. A physical record may contain one or more logical records and/or segments of logical records.

Source Language Considerations

The programmer specifies S-mode by describing the file with the following clauses in the file description (FD) entry of his COBOL program:

- BLOCK CONTAINS integer-2 CHARACTERS
- RECORD CONTAINS [integer-1 TO] integer-2 CHARACTERS
- RECORDING MODE IS S

The size of the physical record must be specified using the BLOCK CONTAINS clause with the CHARACTERS option. Any block size may be specified. Block size is independent of logical record size.

The size of the logical record may be specified by the RECORD CONTAINS clause. If this clause is omitted, the compiler will determine the maximum record size from the record descriptions under the FD.

Format S may be specified by the RECORDING MODE IS S clause. If this clause is omitted, the compiler will set the recording mode to S if the BLOCK CONTAINS integer-2 CHARACTERS clause was specified and either:

1. integer-2 is less than the largest fixed-length level-01 FD entry
2. integer-2 is less than the maximum length of a variable level-01 FD entry (i.e., an entry containing one or more OCCURS clauses with the DEPENDING ON option).

When the spanned recording mode is being used, each logical record is processed in a work area, not in the buffer. Logical records are always aligned on a double-word boundary. Therefore, the programmer is not required to add inter-record slack bytes for alignment purposes.

Except for the APPLY WRITE-ONLY clause, all the options for a variable file apply to a spanned file.

Processing Sequentially Organized S-Mode Files

Suppose a file has the following file description entry:

```
FD SPAN-FILE
   BLOCK CONTAINS 100 CHARACTERS
   LABEL RECORDS ARE STANDARD
   DATA RECORD IS DATAREC.

01 DATAREC.
   05 FIELD-A PIC X(100).
   05 FIELD-B PIC X(50).
```

Figure 55 illustrates the first four blocks of SPAN-FILE as they would appear on external storage devices (i.e., tape or mass storage) or in buffer areas of virtual storage.

Note:

1. The RECORDING MODE clause is not specified. The compiler determines the recording mode to be S since the block size is less than the record size.
2. The length of each physical block is 100 bytes, as specified in the BLOCK CONTAINS clause. All required control fields, as well as data, must be contained within these 100 bytes.
3. No provision is made for the control fields within the level-01 entry DATAREC.



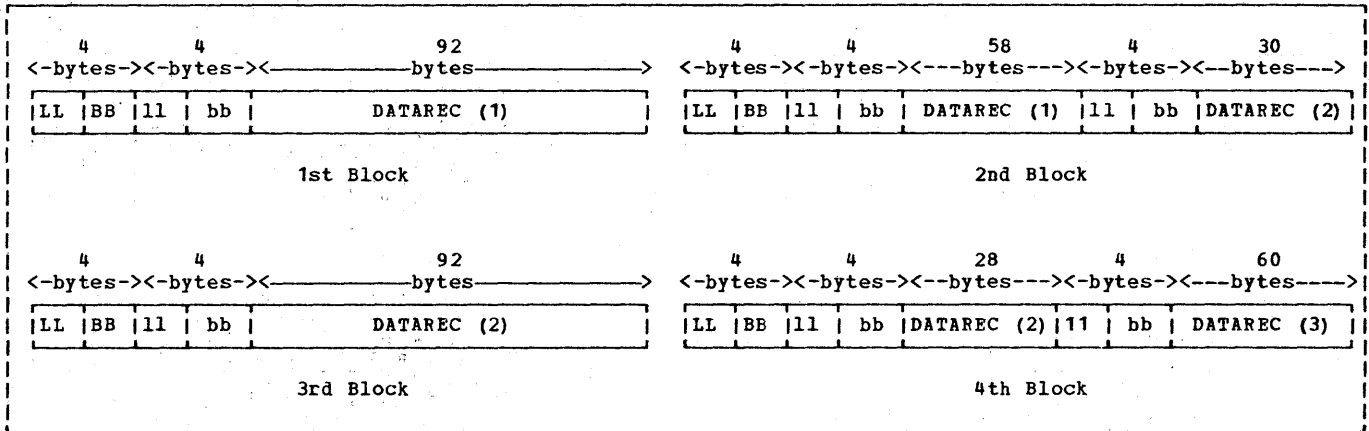


Figure 55. First Four Blocks of SPAN-FILE

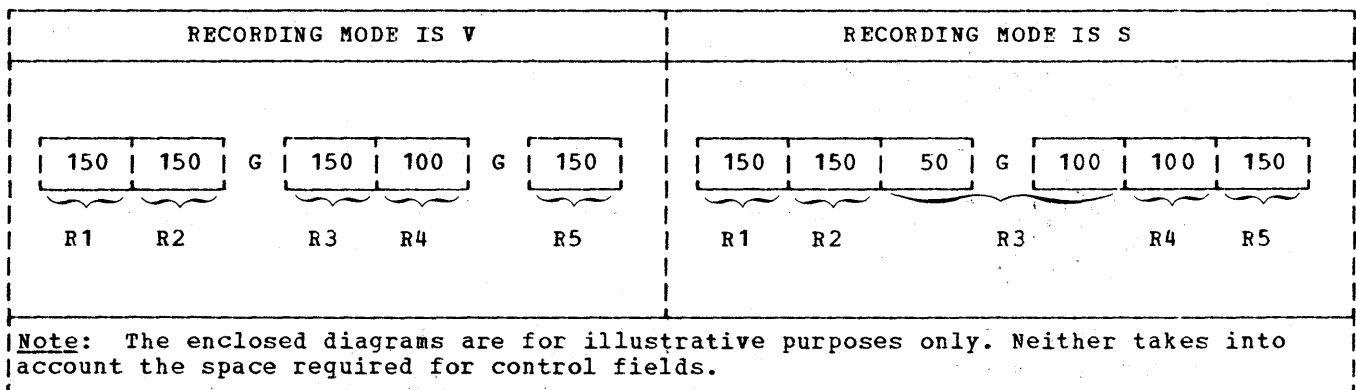


Figure 56. Advantage of S-Mode Records Over V-Mode Records

The preceding discussion dealt with S-mode records which were larger than the physical blocks that contained them. It is also possible to have S-mode records which are equal to or smaller than the physical blocks that contain them. In such cases, the RECORDING MODE clause must specify S (if so desired) since the compiler cannot determine this by comparing block size and record size.

One advantage of S-mode records over V-mode records is illustrated by a file with the following characteristics:

1. RECORD CONTAINS 50 TO 150 CHARACTERS
2. BLOCK CONTAINS 350 CHARACTERS
3. The first five records written are 150, 150, 150, 100, and 150 characters in length.

For V-mode records, buffers are truncated if the next logical record is too large to be completely contained in the block (see Figure 56). This results in more physical blocks and more inter-record gaps on the external storage device.

Note: For V-mode records, buffer truncation occurs:

1. When the maximum level-01 record is too large.
2. If APPLY WRITE-ONLY or SAME RECORD AREA is specified and the actual logical record is too large.

For S-mode records, all blocks are 350 bytes long and records that are too large to fit entirely into a block will be segmented. This results in more efficient use of external storage devices since the

number of inter-record gaps are minimized (Figure 56).

With the exception of the last block, the actual physical block size will always fall between the limits of specified block size and four bytes less than the specified block size, depending on whether or not the residual space of an incomplete block in the buffer is sufficient to add a segment length field and at least one byte of data. That is, specified block size - 4 ≤ actual block size ≤ specified block size.

The last block may be short when an incomplete block remains in the buffer at CLOSE time.

A second advantage of S-mode processing over that of V-mode is that the programmer is no longer limited to a record length that does not exceed the track capacity of the mass storage device selected. Records may span track, cylinders, and extents, but not volumes.

DTFMT and DTFSD spanned records differ from other formats because of an allocation of an area of storage known as the "logical record area." If logical records span physical blocks, COBOL will use this logical record area to assemble complete logical records. If logical records do not span blocks (i.e., they are contained within a single physical block) the logical record area is not used. Regardless, it is complete logical records that are made available to the programmer. Both READ and WRITE statements should be thought of as manipulating complete logical records and not record segments.

DIRECTLY ORGANIZED S-MODE FILES

When S-mode is used for a directly organized file, only unblocked records are permitted. Logical records may be either fixed or variable in length. A logical record will span physical records if, and only if, it spans tracks. A physical record will contain only one logical record or a segment of a logical record, or segments of two logical records and/or whole logical records. Records may span tracks, cylinders, and extents, but not volumes.

Source Language Considerations

The programmer specifies S-mode by describing the file with the following clauses in the file description (FD) entry of his COBOL program:

- BLOCK CONTAINS integer-2 CHARACTERS
- RECORD CONTAINS [integer-1 TO] integer-2 CHARACTERS
- RECORDING MODE IS S

The size of a logical record may be specified by the RECORD CONTAINS clause. If this clause is omitted, the compiler will determine the maximum record size from the record descriptions under the FD.

The spanned format may be specified by the RECORDING MODE IS S clause. If this clause is omitted, the compiler will set the recording mode to S if the BLOCK CONTAINS integer-2 CHARACTERS clause was

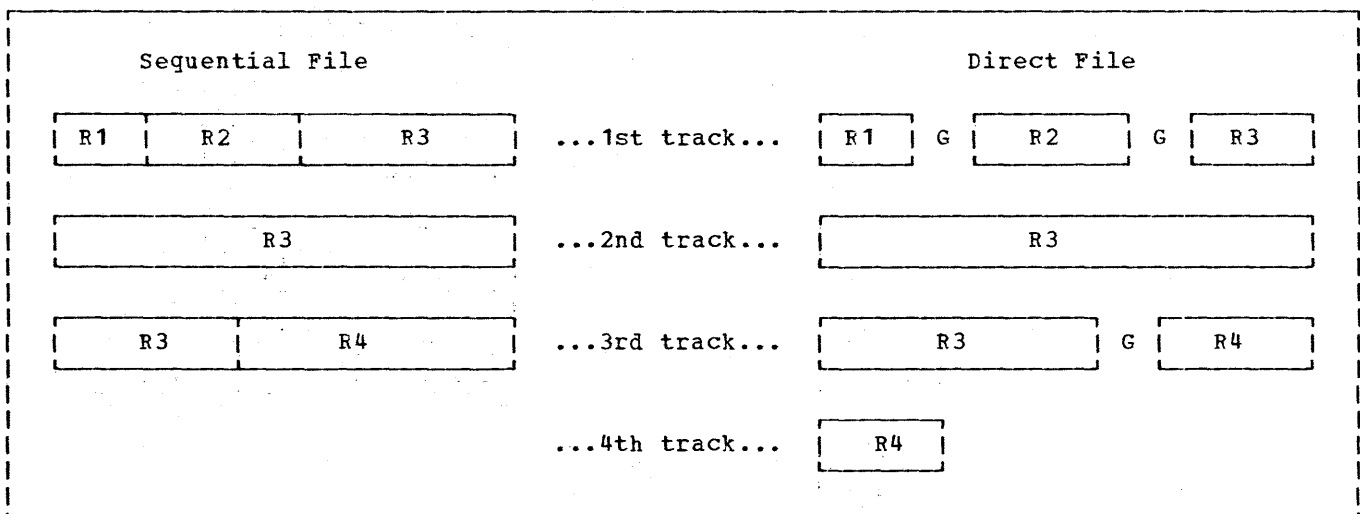


Figure 57. Direct and Sequential Spanned Files on a Mass Storage Device

REC
FMTS

specified and integer-2 is less than the greatest logical record size. This is the only use of the BLOCK CONTAINS clause. It is otherwise treated as comments.

The physical block size is determined by either:

1. The logical record length, or
2. The track capacity of the device being used.

If, for example, the track capacity of a mass storage device is 3625 characters, any record smaller than 3625 characters may be written as a single physical block. If a logical record is greater than 3625 characters, the record is segmented. The first segment may be contained in a physical block of up to 3625 bytes, and the remaining segments must be contained in succeeding blocks. In other words, a logical record will span physical blocks if, and only if, it spans tracks.

Figure 57 illustrates four variable-length records (R1, R2, R3, and R4) as they would appear in direct and sequential files on a mass storage device. In both cases, control fields have been omitted for illustrative purposes. For both files, assume:

1. BLOCK CONTAINS 3625 CHARACTERS (track capacity = 3625)
2. RECORD CONTAINS 500 TO 5000 CHARACTERS

In the sequential file, each physical block is 3625 bytes in length and is completely filled with logical records. The file consists of three physical blocks, occupies three tracks, and contains no inter-record gaps.

In the direct file, the physical blocks vary in length. Each block contains only

one logical record or one record segment. Logical record R3 spans physical blocks only because it spans tracks. The file consists of seven physical blocks, occupies more than three tracks, and contains three inter-record gaps.

Processing Directly Organized S-Mode Files

When processing directly organized files, there are two advantages spanned format has over the other record formats:

1. Logical record lengths may exceed the length restriction of the track capacity of the mass storage device. If, for example, the track capacity of a mass storage device is 2000 bytes, the length of each logical record for formats other than spanned is, by necessity, restricted to the track capacity.

Note: Even when the spanned format is used, the COBOL restriction on the length of logical records (i.e., a maximum length of 32,767 characters) must be adhered to.

2. For formats other than spanned, only complete logical records can be written on any single track. This means that if a track has only 1000 unoccupied bytes and the programmer attempts to add a record of 1100 bytes to this track, an INVALID KEY condition will occur. When the spanned format is used, a 1000 byte segment will be written on the specified track, and the remainder will be written on the next track. The segmenting is transparent to the programmer.

PART III

PROGRAMMING TECHNIQUES → PGM
TECH

USING THE SORT FEATURE → SORT

USING THE REPORT WRITER FEATURE → RPT
WTR

TABLE HANDLING CONSIDERATIONS → TBL
HDLN

This chapter describes techniques and hints for better COBOL programming.

Note: When OPT is in effect, the generated code is more suitable for running under VS, as the addressing scheme is designed to reduce possible page faults.

CODING CONSIDERATIONS FOR DOS/VS

These suggestions will aid DOS/VS efficiency:

- If a short subprogram is referenced only once or twice (and is not an exception condition routine), then its code should be incorporated in the calling program, if convenient.
- Subprograms and frequently used subroutines should be loaded near the programs which use them. This can be done via linkage editor control cards.
- Segmentation in many cases is no longer necessary or desirable.
- Data items of constant value should be grouped together. Data items whose values vary during execution should also be grouped together and should be separate from those of constant value, if feasible.
- FDs for files that will be opened at the same time should be grouped together.
- The most frequently referenced data items should be placed in the beginning of the Working Storage Section.
- The COBOL Procedure Division should be organized generally as follows:
 - All frequently used paragraphs or sections should be located near the routines that use them.
 - All infrequently used paragraphs or sections should be grouped together and apart from frequently used routines. The COUNT option can be used as an aid in this process.
- Avoid initializing data areas until just before they are needed.
- Reference data in the order in which it is stored.
- Use the OPTIMIZE feature if possible.

Further, the procedure is divided into 4K blocks, each of which is assigned a PBL. Since these blocks correspond to two pages each, the user may get some idea of the inter-page relationships in his program (although the first is not page aligned). The statement range for each PBL is given on the compiler output listing. This should help the user rearrange his program if he so desires.

- The REDEFINES clause should be used for its alternate grouping and alternate description capabilities rather than for merely saving space. Although it will save virtual space, it can lead to coding errors if not used carefully.

GENERAL CONSIDERATIONS

COPY

The COPY function should be used by an installation so that if a record format, for example, changes, each program does not need to be modified itself. Rather, the COPY library is updated and each program then recompiled.

Use of this function can lead to standardization of naming conventions and ease of maintenance.

SYNTAX CHECKING

The first several compilations of a program should use the CSYNTAX or SYNTAX feature to save compilation time.

Formatting the Source Program Listing

The lister feature increases significantly the usability of the source program listing, not only by producing cross-reference information, but by

formatting the listing to aid logic tracing. There are four statements that can be coded in any or all of the four divisions of a source program: SKIP1, SKIP2, SKIP3, and EJECT. These statements provide the programmer with the ability to control the spacing of a source listing and thereby improve its readability. These statements should not be used when the lister feature is used.

ENVIRONMENT DIVISION

RESERVE Clause

When using an additional buffer to process standard sequential or indexed

files, care must be taken to ensure that the buffer is filled before the execution of each WRITE or REWRITE statement.

APPLY WRITE-ONLY Clause

To make optimum use of buffer and external storage space allocated when creating a standard sequential file with blocked V-mode records, the programmer should use the APPLY WRITE-ONLY clause for the file. Using this clause causes a buffer to be truncated only when the next record does not fit in the buffer. (If APPLY WRITE-ONLY is not specified, the buffer is truncated when the maximum size record will not fit in the space remaining in the buffer.)

DATA DIVISION

OVERALL CONSIDERATIONS

FD Entries

File Description (FD) entries for the most active files should appear first, since the COBOL compiler assigns registers to files until it runs out of registers, and then reuses the last registers for all subsequent files. This does not apply when OPT is in effect, since in that case the compiler will determine the frequency of usage and assign registers accordingly.

Prefixes

Assign a prefix to each level-01 item in a program, and use this prefix on every subordinate item (except FILLER) to associate a file with its records and work areas. For example, MASTER is the prefix used here:

FILE SECTION.

FD MASTER-INPUT-FILE

.
.

01 MASTER-INPUT-RECORD.

.
.

WORKING-STORAGE SECTION.

01 MASTER-WORK-AREA.

05 MASTER-PAYROLL PICTURE 9(3).
05 MASTER-SSNO PICTURE 9(9).

If files or work areas have the same fields, use the prefix to distinguish between them. For example, if three files all have a date field, instead of DATE, DAT, and DA-TE, use MASTER-DATE, DETAIL-DATE, and REPORT-DATE. Using a unique prefix for each level-01 item and all subordinate fields makes it easier for a programmer unfamiliar with the program to find fields in the program listing, and to know which fields are logically part of the same record or area.

When using the MOVE statement with the CORRESPONDING option and referring to individual fields, redefine or rename "corresponding" names with the prefixed unique names. This technique eliminates excessive qualifying. For example:

```
01 MST-WORK-AREA.
05 SAME-NAMES.      (***)
   10 LAST-NAME PIC...
   10 FIRST-NAME PIC...
   10 PAYROLL PIC...
   .
   .
05 DIFF-NAMES REDEFINES SAME-NAMES.
   10 MST-LAST-NAME PIC...
   10 MST-FIRST-NAME PIC...
   10 MST-PAYROLL PIC...
01 RPT-WORK-AREA.
05 SAME-NAMES.      (***)
   10 PAYROLL PIC...
   10 FILLER PIC...
   10 FIRST-NAME PIC...
   10 FILLER PIC...
   10 LAST-NAME PIC...
```

PROCEDURE DIVISION.

```
.
.
IF MST-PAYROLL IS EQUAL TO HDQ-PAYROLL
AND MST-LAST-NAME
IS NOT EQUAL TO PRRV-LAST-NAME
MOVE CORRESPONDING
MST-WORK-AREA
TO RPT-WORK-AREA.
```

Note: Fields marked *** above must have exactly the same names for their subordinate fields to be considered "corresponding." The same names must not be the redefining ones or they will not be considered to correspond.

Level Numbers

The programmer should use widely incremented level numbers such as 01, 05, 10, 15, etc., instead of 01, 02, 03, 04, etc., in order to allow space for future insertions of group levels. For readability, indent level numbers. (The lister feature does this automatically, even if the original source program does not follow such indenting practices.)

Note that when using the SYMDMP option, level numbers appear "normalized" in the symbolic dump produced. For example, a group of data items described as:

```
01 RECORDA.
05 FIELD-A.
   10 FIELD-A1 PIC X.
   10 FIELD-A2 PIC X.
```



will appear as follows in SYMDMP output:

```
01 RECORDA...
02 FIELD-A...
03 FIELD-A1...
03 FIELD-A2...
```

Use level number 88 for codes. Thus, if the codes must be changed, the Procedure Division coding for tests need not be changed.

FILE SECTION

RECORD CONTAINS Clause

The programmer should use the RECORD CONTAINS clause with the integer CHARACTERS option in order to save himself, as well as any future programmer, the task of counting the data record description positions. In addition, the compiler can then diagnose errors if the data record description conflicts with the RECORD CONTAINS clause.

BLOCK CONTAINS Clause

If a block prefix exists on an ASCII file and the BLOCK CONTAINS clause is used in the COBOL program, the length of the block prefix must be included in the BLOCK CONTAINS clause.

WORKING-STORAGE SECTION

Separate Modules

In a large program, the programmer may wish to plan ahead for breaking the programs into separately compiled modules, as follows:

1. When using separate modules, an attempt should be made to combine entries of each Working-Storage Section into a single level-01 record (or a single level-01 record for each 32K bytes). Logical record areas can be indicated by using level-02, -03, etc., entries. A CALL statement with the USING option is more efficient when a single item is passed than when many level-01 and/or -77 items are passed. When this method is employed, mistakes are more easily avoided.

2. Areas which do not contain VALUE clauses should be separated from areas that do contain VALUE clauses. VALUE clauses (except for level-88 items) are invalid in the Linkage Section.
3. When the Working-Storage Section consists of one level-01 item without any VALUE clauses, the COPY statement can easily be used to include the item as the description of a Linkage Section in a separately compiled module.
4. See the chapter "Using the Segmentation Feature" for information on how to modularize the Procedure Division of a COBOL program; VS coding considerations should also be taken into account.

Locating the Working-Storage Section in Dumps

If the SYMDMP option is not used for program debugging, a method of locating the Working-Storage Section of a program in object-time dumps is to include the two following statements as the first and last Working-Storage statements, respectively, in the program.

```
77 FILLER PICTURE X(44), VALUE "PROGRAM
   XXXXXXXX WORKING-STORAGE BEGINS HERE".
01 FILLER PICTURE X(42), VALUE "PROGRAM
   XXXXXXXX WORKING-STORAGE ENDS HERE".
```

These two nonnumeric literals will appear in all dumps of the program, delimiting the Working-Storage Section. The program-name specified in the PROGRAM-ID clause should replace the XXXXXXXX in the literal.

The location and length of Working-Storage is given in the compiler output when SYM, LISTX, or CLIST is in effect.

REDEFINES Clause

REUSING DATA AREAS: Virtual storage can be used more efficiently by writing different data descriptions for the same data area. For example, the coding that follows shows how the same area can be used as a work area for the records of several input files that are not processed concurrently. Caution should be exercised when using this procedure, as it can lead to programming errors.

WORKING-STORAGE SECTION.

```
01 WORK-AREA-FILE1.  
    (largest record description for FILE1)  
.  
.  
01 WORK-AREA-FILE2 REDEFINES  
    WORK-AREA-FILE1.  
    (largest record description for FILE2)  
.  
.  
.
```

ALTERNATE GROUPINGS AND DESCRIPTIONS:

Program data can often be described more efficiently by providing alternate groupings or data descriptions for the same data. For example, a program references both a field and its subfields, each of which is more efficiently described with a different usage. This can be done by using the REDEFINES clause as follows:

```
01 PAYROLL-RECORD.  
05 EMPLOYEE-RECORD PICTURE X(28).  
05 EMPLOYEE-FIELD REDEFINES  
    EMPLOYEE-RECORD.  
    10 NAME PICTURE X(24).  
    10 NUMBERX PICTURE S9(5) COMP.  
05 DATE-RECORD PICTURE X(10).
```

The following illustrates how a table (TABLEA) can be initialized by having different data descriptions for the same data:

```
05 VALUE-A.  
    10 A1 PICTURE S9(9) COMPUTATIONAL  
        VALUE IS ZEROES.  
    10 A2 PICTURE S9(9) COMPUTATIONAL  
        VALUE IS 1.  
.  
.  
    10 A100 PICTURE S9(9) COMPUTATIONAL  
        VALUE IS 99.  
05 TABLEA REDEFINES VALUE-A  
    PICTURE S9(9) COMPUTATIONAL  
    OCCURS 100 TIMES.
```

Note: Caution should be exercised when redefining a subscript. If the value of the redefining data item is changed in the Procedure Division, no new calculation for the subscript is performed.

PGM
TECH

PICTURE Clause

DECIMAL-POINT ALIGNMENT: Procedure Division operations are most efficient when the decimal positions of the data items involved are aligned. If they are not, the compiler generates instructions to align the decimal positions before any operations involving the data items can be executed.

Assume, for example, that a program contains the following instructions:

WORKING-STORAGE SECTION.

77 A PICTURE S999V99.

77 B PICTURE S99V9.

.
. .
.

PROCEDURE DIVISION.

.
. .
.

ADD A TO B.

Time and internal storage space are saved by defining B as:

77 B PICTURE S99V99.

If it is inefficient to define B differently, a one-time conversion can be done, as explained in "Data Format Conversion" in this chapter.

FIELDS OF UNEQUAL LENGTH: When a data item is moved to another data item of a different length, the following should be considered:

- If the items are external decimal items, the compiler generates instructions to insert zeros in the high-order positions of the receiving field, when it is the larger.
- If the items are nonnumeric, the compiler may generate instructions to insert spaces in the low-order positions of the receiving field (or the high-order positions if the JUSTIFIED RIGHT clause is specified). This generation of extra instructions can be avoided if the sending field is described with a length equal to or greater than the receiving field.

SIGN USAGE: The presence or absence of a plus or minus sign in the description of an arithmetic field often can affect the efficiency of a program. The following paragraphs discuss some of the considerations.

Decimal Items: The sign position in an internal or external decimal item can contain:

1. A plus or minus sign. If S is specified in the PICTURE clause, a plus or minus sign is inserted when either of the following conditions prevail:
 - a. The item is in the Working-Storage Section and a VALUE clause has been specified.
 - b. A value for the item is assigned as a result of an arithmetic operation during execution of the program.

If an external decimal item is punched, printed, or displayed, an overpunch will appear in the low-order digit. In EBCDIC, the configuration for low-order zeros normally is a nonprintable character. Low-order digits of positive values will be represented by one of the letters A through I (digits 1 through 9); low-order digits of negative values will be represented by one of the letters J through R (digits 1 through 9).

2. A hexadecimal F. If S is not specified in the PICTURE clause, an F is inserted in the sign position when either of the following conditions prevail:
 - a. The item is in the Working-Storage Section and a VALUE clause has been specified
 - b. A value for the item is developed during the execution of the program.

An F is treated as positive, but is not an overpunch.

3. An invalid configuration. If an internal or external decimal item contains an invalid configuration in the sign position, and if the item is involved in a Procedure Division operation, the program will be abnormally terminated.

Note: If the SIGN clause is used and it specifies that the sign is LEADING, more object code will be generated when that data item is used with a verb. The additional code is needed to move the sign character to the TRAILING position before performing the operation.

Unsigned items (items for which no S has been specified) are treated as absolute values. Whenever a value (signed or unsigned) is stored in or moved in an

elementary move to an unsigned item, a hexadecimal F is stored in the sign position of the unsigned item. For example, if an arithmetic operation involves signed operands and an unsigned result field, compiler-generated code will insert an F in the sign position of the result field when the result is stored.

For internal and external decimal items used as input, it is the programmer's responsibility to ensure that the input data is valid. The compiler does not generate a test to ensure that the configuration in the sign position is valid.

When a group item is being moved, the data is moved without regard to the level structure of the group items involved. The possibility exists that the configuration in the sign position of a subordinate numeric item may be destroyed. Therefore, caution should be exercised in moving group items with subordinate numeric fields or with other group operations such as READ or ACCEPT.

USAGE Clause

DATA FORMAT CONVERSION: Operations involving mixed, elementary numeric data formats require conversion to a common format. This usually means that additional storage is used and execution time is increased. The code generated must often move data to an internal work area, perform any necessary conversion, and then execute the indicated operation. Often, too, the result may have to be converted in the same way. Table 31 indicates when data conversion is necessary.

If it is impractical to use the same data formats throughout a program, and if two data items of different formats are frequently used together, a one-time conversion can be effected. For example, if A is defined as a COMPUTATIONAL item and B as a COMPUTATIONAL-3 item, A can be moved to a work area that has been defined as COMPUTATIONAL-3. This move causes the data in A to be converted to COMPUTATIONAL-3. Whenever A and B are used in a Procedure Division operation, reference can be made to the work area rather than to A. When this technique is used, the conversion is performed only once, instead of each time an operation is performed.

PGM
TECH

Table 31: Data Format Conversion (Part 1 of 2)

Usage	Bytes Required	Boundary Alignment Required	Typical Usage	Converted for Arithmetic Operations	Special Characteristics
DISPLAY (external decimal)	1 per digit (except for V)	No	Input from cards, output to cards, listings	Yes	May be used for numeric fields up to 18 digits long. Fields over 15 digits require extra instructions if used in computations.
DISPLAY (external floating point)	1 per character (except for V)	No	Input from cards, output to cards, listings	Yes	Converted to COMP-2 format via COBOL library subroutine.
COMP-3 (internal decimal)	1 per 2 digits plus 1 byte for low-order digit and sign	No	Input to a report item Arithmetic fields Work areas	Sometimes when a small COMP-3 item is used with a small COMP item	Requires less space than DISPLAY. Convenient form for decimal alignment. Can be used in arithmetic computations without conversion. Fields over 15 digits require a subroutine when used in computations.
COMP (binary)	2 if $1 \leq N \leq 4$ 4 if $5 \leq N \leq 9$ 8 if $10 \leq N \leq 18$ where N is the number of 9's in the picture	Halfword Fullword Fullword	Subscripting Arithmetic fields	Sometimes for both mixed and unmixed usages	Rounding and testing for the ON SIZE ERROR condition are cumbersome if calculated result is greater than 9(9). Extra instructions are generated for computations if the SYNCHRONIZED clause is not specified. Fields of over nine digits require additional handling.

Table 31. Data Format Conversion (Part 2 of 2)

Usage	Bytes Required	Boundary Alignment Required	Typical Usage	Converted for Arithmetic Operations	Special Characteristics
COMP-1 (internal floating point)	4 (short-precision)	Fullword	Fractional exponentiation	No	Tends to produce less accurate results if more than 17 significant digits are required and if the exponent is large. Extra instructions are generated for computations if the SYNCHRONIZED clause is not specified. Requires floating-point feature.
COMP-2 (internal floating point)	8 (long-precision)	Double-word	Fractional exponentiation when additional precision is required	No	Same as COMP-1.

PGM
TECH

The following seven cases show how data conversions are handled on mixed elementary items for names, data comparisons, and arithmetic operations. Moves without the CORRESPONDING option to and from group items, as well as comparisons involving group items, are done without conversion.

COMPUTATIONAL-3 to COMPUTATIONAL:

To Move Data: Moves COMPUTATIONAL-3 data to a work area and then converts COMPUTATIONAL-3 data to COMPUTATIONAL data.

To Compare Data: Converts COMPUTATIONAL data to COMPUTATIONAL-3 or vice versa, depending on the size of the field.

To Perform Arithmetic Operations: Converts COMPUTATIONAL data to COMPUTATIONAL-3 or vice versa, depending on the size of the field.

Numeric DISPLAY to COMPUTATIONAL-3:

To Move Data: Converts DISPLAY data to COMPUTATIONAL-3 data.

To Compare Data: Converts DISPLAY data to COMPUTATIONAL-3 data.

To Perform Arithmetic Operations: Converts DISPLAY data to COMPUTATIONAL-3 data.

COMPUTATIONAL to COMPUTATIONAL-3:

To Move Data: Converts COMPUTATIONAL data to COMPUTATIONAL-3 data in a work area, and then moves the work area.

To Compare Data: Converts COMPUTATIONAL to COMPUTATIONAL-3 data or vice versa, depending on the size of the field.

To Perform Arithmetic Operations: Converts COMPUTATIONAL to COMPUTATIONAL-3 data or vice versa, depending on the size of the field.

Numeric DISPLAY to COMPUTATIONAL:

To Move Data: Converts DISPLAY data to COMPUTATIONAL-3 data and then to COMPUTATIONAL data.

To Compare Data: Converts DISPLAY to COMPUTATIONAL or converts both DISPLAY and COMPUTATIONAL data to COMPUTATIONAL-3 data.

To Perform Arithmetic Operations: Converts DISPLAY data to COMPUTATIONAL-3 or COMPUTATIONAL data.

COMPUTATIONAL to Numeric DISPLAY:

To Move Data: Converts COMPUTATIONAL data to COMPUTATIONAL-3 data and then to DISPLAY data.

To Compare Data: Converts DISPLAY to COMPUTATIONAL or both COMPUTATIONAL and DISPLAY data to COMPUTATIONAL-3 data, depending on the size of the field.

To Perform Arithmetic Operations: Depending on the size of the field, converts DISPLAY data to COMPUTATIONAL data, or both DISPLAY and COMPUTATIONAL data to COMPUTATIONAL-3 data in which case the result is generated in a COMPUTATIONAL-3 work area and then converted and moved to the DISPLAY result field.

COMPUTATIONAL-3 to Numeric DISPLAY:

To Move Data: Converts COMPUTATIONAL-3 data to DISPLAY data.

To Compare Data: Converts DISPLAY data to COMPUTATIONAL-3 data. The result is generated in a COMPUTATIONAL-3 work area and is then converted and moved to the DISPLAY result field.

Numeric DISPLAY to Numeric DISPLAY:

To Perform Arithmetic Operations: Converts all DISPLAY data to COMPUTATIONAL-3 data. The result is generated in a COMPUTATIONAL-3 work area and is then converted to DISPLAY and moved to the DISPLAY result field.

Internal Floating-point to Any Other: When an item described as COMPUTATIONAL-1 or COMPUTATIONAL-2 (internal floating-point) is used in an operation with another data format, the item in the other data format is always converted to internal floating-point. If necessary, the internal floating-point result is then converted to the format of the other data item.

SYNCHRONIZED Clause

As illustrated in Table 31, COMPUTATIONAL, COMPUTATIONAL-1 and COMPUTATIONAL-2 items have specific boundary alignment requirements. To ensure correct alignment, either the programmer or the compiler may have to insert slack bytes or the compiler must generate extra instructions to move the item to a correctly aligned work area when reference is made to the item.

The SYNCHRONIZED clause may be used at the elementary level to specify the automatic alignment of elementary items on their proper boundaries, or at the 01 level to synchronize all elementary items within the group. For COMPUTATIONAL items, if the PICTURE is in the range of S9 through S9(4), the item is aligned on a halfword boundary. If the PICTURE is in the range of S9(5) through S9(18), the item is aligned on a fullword boundary. For COMPUTATIONAL-1 items, the item is aligned on a fullword boundary. For COMPUTATIONAL-2 items, the item is aligned on a doubleword boundary. The SYNCHRONIZED clause and slack bytes are fully discussed in the publication IBM System/360 Disk Operating System: Full American National Standard COBOL.

Special Considerations for DISPLAY and COMPUTATIONAL Fields

NUMERIC DISPLAY FIELDS: Zeros are not inserted into numeric DISPLAY fields by the instruction set. When numeric DISPLAY data is moved, the compiler generates instructions that insert any necessary zeros into the DISPLAY fields. When numeric DISPLAY data is compared, and one field is smaller than the other, the compiler generates instructions to move the smaller item to a work area where zeros are inserted.

COMPUTATIONAL FIELDS: COMPUTATIONAL fields can be aligned on either a halfword or fullword boundary. If an operation involves COMPUTATIONAL fields of different lengths, the halfword field is automatically expanded to a fullword field. Therefore, mixed halfword and fullword fields require no additional operations.

COMPUTATIONAL-1 AND COMPUTATIONAL-2 FIELDS: If an arithmetic operation involves a mixture of short-precision and long-precision fields, the compiler generates instructions to expand the short-precision field to a long-precision field before the operation is executed.

COMPUTATIONAL-3 FIELDS: The compiler does not have to generate instructions to insert high-order zeros for ADD and SUBTRACT statements that involve COMPUTATIONAL-3 data. The zeros are inserted by the instruction set.

Data Formats in the Computer

The following examples illustrate how the various COBOL data formats appear in the computer in EBCDIC (Extended

Binary-Coded-Decimal Interchange Code) format. More detailed information about these data formats appear in the publication IBM System/370 Principles of Operation.

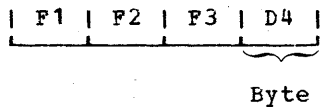
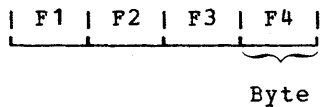
Numeric DISPLAY (External Decimal): Suppose the value of an item is -1234, and its PICTURE and USAGE clauses are:

PICTURE 9999 DISPLAY.

or

PICTURE S9999 DISPLAY.

The item appears in the computer in the following forms, respectively:



Hexadecimal F is treated arithmetically as positive; hexadecimal D represents a minus sign.

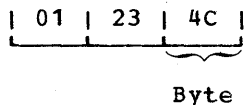
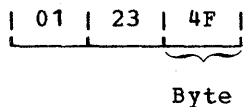
COMPUTATIONAL-3 (Internal Decimal): Suppose the value of an item is +1234, and its PICTURE and USAGE clauses are:

PICTURE 9999 COMPUTATIONAL-3.

or

PICTURE S9999 COMPUTATIONAL-3.

The item appears internally in the following forms, respectively:



Hexadecimal F is treated arithmetically as positive; hexadecimal C represents a plus sign.

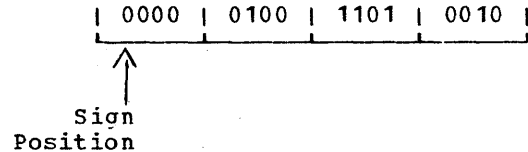
Note: Since the low-order byte of an internal decimal number always contains a sign field, an item with an odd number of digits can be stored more efficiently than an item with an even number of digits.

Note that a leading zero is inserted in the above example.

COMPUTATIONAL (Binary): Suppose the value of an item is 1234, and its PICTURE and USAGE clauses are:

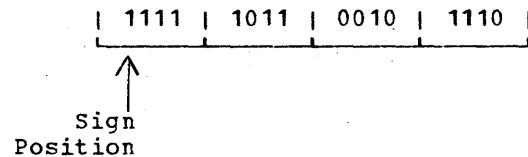
PICTURE S9999 COMPUTATIONAL.

The item appears internally in the following form:



A 0 in the sign position indicates that the number is positive. Negative numbers are represented in two's complement form; thus, the sign position of a negative number will always contain a 1.

For example -1234 would appear as follows:



Binary Item Manipulation: A binary item is allocated storage ranging from one halfword to two fullwords, depending on the number of 9's in its PICTURE. Table 32 is an illustration of how the compiler allocates this storage. Note that it is possible for a value larger than that implied by the PICTURE clause to be stored in the item. For example, PICTURE S9(4) implies a maximum value of 9,999, although it could actually hold the number 32,767.

Because most binary items are manipulated according to their allotted storage capacity, the programmer can ignore this situation. For the following reasons, however, he must be careful of his data:

1. When the ON SIZE ERROR option is used, the size test is made on the basis of the maximum value allowed by the picture of the result field. If a size error condition exists, the value of the result field is not altered and control is given to the imperative-statements specified by the error option.



Table 32. Relationship of PICTURE to Storage Allocation

PICTURE	Maximum Working Value	Assigned Storage
S9 through S9 (4)	32,767	One halfword
S9 (5) through S9 (9)	2,147,483,647	One fullword
S9 (10) through S9 (18)	9,223,372,036,854,775,807	Two fullwords

- When a binary item is displayed or exhibited, the value used is a function of the number of 9's specified in the PICTURE clause.
- When the actual value of a positive number is significantly larger than its picture value, a value of 1 could appear in the sign position of the item, causing the item to be treated as a negative number in subsequent operations.

Bits 1 through 7 are the exponent (characteristic) of the number.

Bits 8 through 31 are the fraction (mantissa) of the number.

This form of data is referred to as floating point. The example illustrates short-precision floating-point data (COMPUTATIONAL-1): In long-precision (COMPUTATIONAL-2), the fraction length is 56 bits. (For a detailed explanation of floating-point representation, see the publication IBM System/370 Principles of Operation.)

Figure 58 illustrates three binary manipulations. In each case, the result field is an item described as PICTURE S9 COMPUTATIONAL. One halfword of storage has been allocated, and no ON SIZE ERROR option is involved. Note that if the ON SIZE ERROR option had been specified, it would have been executed for cases B and C.

PROCEDURE DIVISION

The Procedure Division of a program can often be made more efficient or easier to debug by using some of the techniques described below.

COMPUTATIONAL-1 or COMPUTATIONAL-2 (Floating-point): Suppose the value of an item is +1234 and that its USAGE is COMPUTATIONAL-1, the item appears internally in the following form:

MODULARIZING THE PROCEDURE DIVISION

Modularization involves organizing the Procedure Division into at least three functional levels: a main-line routine, processing subroutines, and input/output subroutines. When the Procedure Division is modularized, programs are easier to maintain and document. In addition, modularization makes it simple to break down a program using the segmentation feature, resulting in a more efficient segmented program. Virtual storage implications should be taken into

```
|0|100 0011|0100 1101 0010 0000 0000 0000|
S 1      7 8                               31
```

- S is the sign position of the number.
- 0 in the sign position indicates that the sign is plus.
 - 1 in the sign position indicates that the sign is minus.

Case	Hexadecimal Result of Binary Calculation	Decimal Equivalent	Actual Decimal Value in Halfword of Storage	DISPLAY or EXHIBIT Value
A	0008	8	+8	8
B	000A	10	+10	0
C	C350	50000	-15536	6

Figure 58. Treatment of Varying Values in a Data Item of PICTURE S9

consideration when rearranging the Procedure Division. The COUNT option is useful in determining a rearrangement scheme.

Main-Line Routine

The main-line routine should be short and simple, and should contain all the major logical decisions of the program. This routine controls the order in which second-level subroutines are executed. All second-level subroutines should be invoked from the main-line routine by PERFORM statements.

Processing Subroutines

Processing subroutines should be broken down into as many functional levels as necessary, depending on the complexity of the program. These must be completely closed subroutines, with one entry point and one exit point. The entry point should be the first statement of the subroutine. The exit point should be the EXIT statement. Processing subroutines can PERFORM only lower level subroutines; return to the higher level subroutine (processing subroutine) must be accomplished by a GO TO statement that references the EXIT statement.

Input/Output Subroutines

The input/output subroutines should be the lowest level subroutines, since all higher level subroutines have access to them. There should be one OPEN subroutine and one CLOSE subroutine for the program, and only one functional (READ or WRITE) subroutine for each file. Having one READ or WRITE subroutine per file has several advantages:

1. Coding can be added to count records on a file, transform blanks into zeros, check for 9's padding, etc.
2. Input and output files can be reformatted without changing the logic of the program.
3. DEBUG statements can be added during testing to create input or to DISPLAY formatted output, instead of having to create a test file.

OVERALL CONSIDERATIONS

OPTIMIZE Option

If the OPTIMIZE option is in effect, the number of procedure blocks in a program cannot exceed 255. A procedure block is equivalent to approximately 4096 bytes of Procedure Division code.

If the COUNT option is in effect, the number of verb blocks in a program cannot exceed 32,767. A verb block consists of a set of verbs in which any verb (excluding ABEND) in the block is executed if and only if all verbs in the block are executed. The average program Procedure Division contains approximately three verbs per verb block.

INTERMEDIATE RESULTS

The compiler treats arithmetic statements as a succession of operations and sets up intermediate result fields to contain the results of these operations. Examples of such statements are the arithmetic statements and statements containing arithmetic expressions. See the appendix "Intermediate Results" in the publication IBM DOS Full American National Standard COBOL for a description of the algorithms used by the compiler to determine the number of places reserved for intermediate result fields.

Intermediate Results and Binary Data Items

If an operation involving binary operands requires an intermediate result greater than 18 digits, the compiler converts the operands to internal decimal before performing the operation. If the result field is binary, the result will be converted from internal decimal to binary.

If an intermediate result will not be greater than nine digits, the operation is performed most efficiently on binary data fields.

Intermediate Results and COBOL Library Subroutines

If a decimal multiplication operation requires an intermediate result greater than 30 digits, a COBOL library subroutine

PGM
TECH

is used to perform the multiplication. The result of this multiplication is then truncated to 30 digits.

A COBOL library subroutine is used to perform division if:

1. The divisor is equal to or greater than 15 digits.
2. The length of the divisor plus the length of the dividend is greater than 16 bytes.
3. The scaled dividend is greater than 30 digits. (A scaled dividend is a number that has been multiplied by a power of ten in order to obtain the desired number of decimal places in the quotient.)

Intermediate Results Greater Than 30 Digits

Whenever the number of digits in a decimal intermediate result is greater than 30, the field is truncated to 30 digits. A warning message will be generated during compilation, and program flow will not be interrupted at execution time. This truncation may cause a result to be incorrect.

If binary or internal decimal data is in agreement with its data description, no interrupt can occur because of an overflow condition in an intermediate result. This is due to the truncation described in the preceding paragraph.

If the possibility exists that an intermediate result field may exceed 30 digits, truncation can be avoided by the specification of floating-point operands (COMPUTATIONAL-1 or COMPUTATIONAL-2); however, accuracy may not be maintained.

Intermediate Results and Floating-point Data Items

If a floating-point operand has an intermediate result field in which exponent overflow occurs, the job will be abnormally terminated.

Regardless of how B and C are defined in the following statement, if A is a floating-point data item, no decimal places will be calculated in the intermediate result.

COMPUTE A = B / C

Intermediate Results and the ON SIZE ERROR Option

The ON SIZE ERROR option applies only to the final calculated results and not to intermediate result fields.

EXPONENTIATION

When the exponent is not a literal, one of the following three subroutines is invoked, depending on the base and the exponent:

1. If the base is not a floating-point item and the exponent is an integer item, a call to the subroutine ILBDXPRO is generated and the exponentiation is executed in packed decimal arithmetic.
2. If the base is a floating-point item and the exponent is an integer item, a call to the subroutine ILBDGPW0 is generated and the exponentiation is executed in floating-point arithmetic.
3. If the exponent is a floating-point item or has a PCITURE specifying decimal places, a call to the subroutine ILBDFPW0 is generated and the exponentiation is executed in floating-point arithmetic.

When the exponent is an integer literal, one of the following applies:

1. If the base is a floating-point item, a call to the subroutine ILBDGPW0 is generated and the exponentiation is executed in floating-point arithmetic.
2. If the base is not a floating-point item, an in-line loop is generated to perform the exponentiation unless the maximum possible result exceeds 30 digits, in which case a call to the subroutine ILBDXPRO is generated. In either case, the exponentiation is executed in packed decimal arithmetic.

Optimization Based on Execution Frequency

Additional optimization techniques may be used based on execution frequency statistics. These techniques are discussed in the chapter entitled "Execution Statistics".

PROCEDURE DIVISION STATEMENTS

COMPUTE Statement

The use of the COMPUTE statement generates more efficient code than does the use of individual arithmetic statements, since the compiler can keep track of internal work areas and does not have to store the results of intermediate calculations. It is the programmer's responsibility, however, to ensure that the data is defined with the level of significance required in the answer.

A black circular logo with the text "PGM" on the top line and "TECH" on the bottom line, both in white capital letters.

IF Statement

Nested and compound IF statements should be avoided as the logic is difficult to debug.

MOVE Statement

Performing a move operation for an item longer than 256 bytes requires the generation of more instructions than are required for a move operation for an item of 256 bytes or less.

For fields longer than 512 bytes, a MOVE LONG (MVCL) instruction is generated unless the first byte of the receiving field is used as a byte of the sending field. In this case, the object-time subroutine ILBDVMOO is called to perform the move.

When a MOVE statement with the CORRESPONDING option is executed, data items are considered as "corresponding" only if their respective data-names are the same, including all implied qualification up to, but not including, the data-names used in the MOVE statement itself.

For example:

```
01 AA          01 XX
05 BB          05 BB
10 CC          10 CC
10 DD          10 DD
05 EE          05 YY
10 FF          10 FF
```

The statement MOVE CORRESPONDING AA TO XX will result in moving CC, and DD, but not FF, since FF of EE does not correspond to FF of YY.

The compiler assumes that the data being moved conforms to PICTURE and USAGE specifications. If it does not, dissimilar results will occasionally occur because of the different code generated for various sending and receiving fields. This fact is most apparent when the sending field is COMPUTATIONAL, the value in the item exceeds the number of digits specified in the PICTURE clause, and the option NOTRUNC is in effect.

Note: The other rules for MOVE CORRESPONDING, of course, must still be satisfied.

NOTE Statement

When the NOTE statement is the first statement in a paragraph, it will cause the

whole paragraph to be treated as part of the NOTE. Programmer errors can be avoided by using the asterisk (*) in place of the NOTE statement.

PERFORM Statement

PERFORM is a useful statement if the programmer adheres to the following rules:

1. Always execute the last statement of a series of routines being operated on by a PERFORM statement. When branching out of the routine, make sure control will eventually return to the last statement of the routine, which should be an EXIT statement. Although no code is generated, the EXIT statement allows a programmer to immediately recognize the extent of a series of routines within the range of a PERFORM statement.
2. Always either PERFORM routine-name THRU routine-name-exit, or PERFORM section-name. A PERFORM paragraph-name can create problems for the programmer trying to maintain the program. For example, if one paragraph must be broken into two paragraphs, the programmer must examine every statement to determine whether this paragraph is within the range of the PERFORM statement. As a result, all statements referencing the paragraph-name must be changed to PERFORM THRU statements.
3. A PERFORM statement containing embedded PERFORMS or PERFORM VARYING with one or more AFTER options causes the compiler to generate complex code. If a series of simple PERFORM statements can accomplish the same function, the programmer would be wise to substitute these since more efficient code is generated.

READ INTO AND WRITE FROM OPTIONS

Always use READ INTO and WRITE FROM, and process all files in the Working-Storage Section for the following reasons:

1. Debugging is much simpler. Working-Storage areas are easier to locate in a dump than are buffer areas. And, if files are blocked, it is much easier to determine which record in a block was being processed when the abnormal termination occurred.

PGM
TECH

2. Trying to access a record-area after the AT END condition has occurred (for example, AT END MOVE HIGH-VALUE TO INPUT-RECORD) can cause problems if the record area is defined only in the File Section.

Note: The programmer should be aware that additional time is used to execute the move operation involved in each READ INTO or WRITE FROM instruction.

When a READ INTO statement is used for a V-mode or U-mode file, the size of the

longest record for that file is used in the MOVE statement. All other rules of the MOVE statement apply.

TRANSFORM Statement

The TRANSFORM statement generates more efficient code than the EXAMINE REPLACING BY statement when only one character is being transformed. The TRANSFORM statement, however, uses a 256-byte table.

To use the Sort Feature, statements are written in the COBOL source program. These statements are described in IBM DOS Full American National Standard COBOL. The Sort/Merge publications listed in the Preface of this manual contain information on the Sort/Merge feature.

When a SORT or MERGE statement is used in a program, the compiler generates linkages between the program, modules in the subroutine library, and the Sort/Merge program. The name of Sort/Merge called by COBOL is "SORT" and the user must include the proper one on the option.

Depending on the features specified and devices to be used by Sort/Merge, different Sort/Merge products should be used:

<u>Feature</u>	<u>Product Requirement</u>
VSAM	5746-SM1
MERGE	5746-SM1
ASCII-Collated Sort	5743-SM1,5746-SM1
Numeric Sort keys with Sign in the form of leading overpunch or separate character.	5743-SM1,5746-SM1
3330/3333 Sort Work files	5743-SM1,5746-SM1
3340 Sort Work files	5746-SM1
3400 Sort Work files	5743-SM1,5746-SM1
2311 Sort work files	SM-483,5743-SM1

The program product DOS/VS Sort/Merge, 5746-SM1, is designed specifically for use with DOS/VS.

Otherwise, IBM DOS Tape and Disk Sort/Merge, 360N-SM-483, can be used.

Additional job control statements must be included in the execution step of the job to describe the files used by the sort program. These statements are described below in "Sort Job Control Requirements."

Note: The Checkpoint/Restart Feature can be activated during a sorting operation by specifying the RERUN statement.

SORT/MERGE JOB CONTROL REQUIREMENTS

Three types of files can be defined for the Sort program in the execution job step: input, output, and work. Two types of files can be defined for the Merge program in the execution job step: input and output.

SORT INPUT AND OUTPUT CONTROL STATEMENTS

When the USING and/or GIVING options are specified, the compiler generates dummy Input and/or Output Procedures. Hence, the job control requirements for files named as operands of USING and GIVING are the same as those for files used as input to or output from the sorting operation in these procedures.

The following job control statements are required for files used as input to or output from the sorting operation:

```
ASSGN
    followed by
        VOL
        TPLAB
    or
        VOL
        DLAB
        XTENT
    or
        DLBL
        EXTENT
    or
        TLBL
```

The symbolic unit to which each sort input or output file is assigned in the source language ASSIGN clause is specified in an ASSGN control statement.

Note: ASSGN control statements are required only if the input/output devices used in an application have not been previously assigned the appropriate symbolic names.

If an input file contains standard labels, a TLBL or DLBL (or VOL and TPLAB or VOL and DLAB) statement(s) is required. The symbolic name of the device from which the input file is to be read must also be included on this statement.

One EXTENT control statement is required to define the limits of each area of a mass storage device from which an input file



will be read. EXTENT statements must include the symbolic unit name of the device containing the extent.

If the output file is to use standard labels, a TLBL or DLBL statement is required.

One EXTENT control statement must be used to define the limits of each area of a mass storage device onto which the output file is written. The symbolic name of the output unit must appear on this card.

Note: Because the USING and GIVING options generate dummy input and/or Output procedures, the rules on pooling of files in the Sort/Merge Programmer's Guide referenced above do not apply. No pooling of Sort input, output, and work files is allowed.

SORT WORK FILE CONTROL STATEMENTS

The Sort program requires at least one mass storage unit or three tape units as an intermediate sort work file. The symbolic units to which this file is assigned are normally consecutively numbered beginning with SYS001. Intermediate storage may be assigned on the following devices:

- IBM 2400 Series Magnetic Tape Units
- IBM 3400 Series Magnetic Tape Units¹
- IBM 2311 Direct-Access Storage Device
- IBM 2314/2319 Direct-Access Storage Facility
- IBM 3330/3333 Direct-Access Storage Facility¹
- IBM 3340 Direct-Access Storage Facility¹

Note: When variable-length or redefined-length records are being sorted, sort work files must not be assigned to 7-track tapes. 7-track tape work files can only be used to sort records whose keys are packed decimal or binary.

Device types may not be mixed; i.e., work units for a particular sort operation must all be of the same type.

If spanned records are being sorted and mass storage devices are being used as sort

¹Only supported by the DOS Sort/Merge Program Product, Program Number 5743-SM1 or the DOS/VS Sort/Merge Program Product, Program Number 5746-SM1 (see above).

work files, it is the programmer's responsibility to assign these work files to devices whose track sizes are larger than the logical record sizes of the records being sorted. A spanned record that is larger than the available track size can be sorted by assigning the work files to magnetic tape.

If a work unit is to use standard labels, a TLBL or DLBL control statement is required. The filename entry on these statements must be SORTWK1 through SORTWKn. The symbolic unit names assigned to the work areas to be allocated (SYS001, SYS002, etc.) must appear on these cards.

One EXTENT control statement must be included to define each work area on a mass storage device. The total work area required may be divided into as many as eight extents, which would require eight EXTENT control statements. When code SD is specified on the DLBL card, symbolic unit names on these statements must be in consecutive order (SYS001, SYS002, etc.). If SORT-OPTION is specified, the symbolic unit names must be in the same order as specified on SORTWK.

Amount of Intermediate Storage Required

When intermediate storage is assigned on a mass storage unit, at least twice the amount required to hold all input records should be assigned. This area may consist of from one to eight extents, and the extents may be assigned on no more than eight devices.

If tape intermediate storage is used, at least the minimum number of units (three) must be assigned. The input file can be as large as the number of records that can be written on one full reel of tape. Assigning more than three intermediate storage tape drives does not increase the maximum input file size, but does improve performance.

Improving Performance

Performance increases significantly if 50K of real storage is available for execution of the Sort program. At the 100K level, the performance is very high. If insufficient virtual storage is available, the Sort/Merge program will issue a message:

7054A "INSUFFICIENT CORE"

SORT-OPTION Clause

The "SORT-OPTION" clause is a means of specifying the options that have been selected for the associated sort/merge operation that cannot be specified via the SORT special registers. The format of the contents of the data-name is shown in Figure 58.1. This corresponds to the SORT/MERGE option statement. For more details on specific options for SORT, see IBM DOS/VS Sort/Merge Programmer's Guide, Order No. SC33-4028.

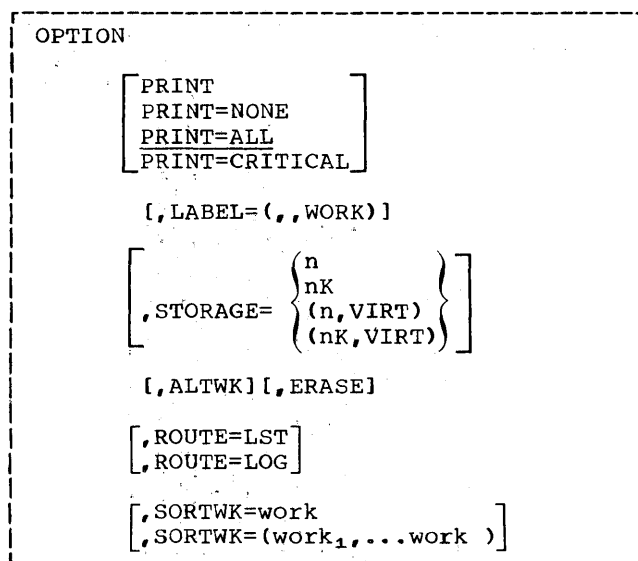


Figure 58.1. OPTION Control Statement to SORT/MERGE

PRINT Option

```
[  
  PRINT  
  PRINT=NONE  
  PRINT=ALL  
  PRINT=CRITICAL  
]
```

PRINT and PRINT=ALL specify that all messages are to be printed by the sort/merge program. This includes error and end-of-job messages, control card information, various size calculations, and other informative messages.

PRINT=NONE specifies that no messages are to be printed by the sort/merge program. This parameter is useful if you have no alternate message device and do not want messages listed with other printed output. A message device need not be assigned.

PRINT=CRITICAL specifies that only messages critical to the sort/merge program's operation are to be printed. These are error messages resulting from conditions that can cause program termination. For more details on these conditions and messages, refer to IBM DOS/VS Sort/Merge Programmer's Guide, Order No. SC33-4028.

Note: PRINT=ALL is assumed until the OPTION statement is read, therefore, if PRINT=NONE or PRINT=CRITICAL will be used, the OPTION statement should precede all others.

LABEL Option

```
[ LABEL=(, , work) ]
```

This operand specifies the type of labels associated with the work files. The two label types are:

S - standard labels
U - unlabeled

The default is S, standard labels.

Work must be replaced by S or U. This operand is required if the OPTION statement is specified, and unlabeled work files are used. If the operand is omitted, standard labels are assumed for all files.

When standard labels are used, the sort/merge program uses the DOS/VS system facilities to process these labels. Unlabeled tape files are processed by the sort/merge program. No user programming is required.

STORAGE Option

```
[  
  , STORAGE= {  
    n  
    nK  
    (n, VIRT)  
    (nK, VIRT)  
  }  
]
```

This option is required to specify to the sort/merge program how much storage to use and whether it can fix pages.

```
[ STORAGE=n  
  STORAGE=(n, VIRT) ]
```

n specifies the amount of storage to be made available to sort/merge (together with its user routines). n can be specified either as a decimal number of bytes, or as a decimal number of K (1024 bytes).



The default is the value of the SIZE parameter on the EXEC job control statement. If both SIZE and STORAGE are specified, the lower value is taken. If neither is specified, the default is the partition size or the required size calculated by sort/merge (but at least 64K), whichever is smaller. The sort/merge program terminates if n is less than 16K bytes. If n is greater than the partition size, it is ignored.

If the sort/merge program is invoked from another program, the defaults are calculated in a similar way, but the value of the SIZE parameter and the partition size are adjusted downwards by the difference between the address of the sort/merge load point and the beginning address of the partition.

VIRT

If VIRT is specified, the sort/merge program will not attempt to fix pages when running in virtual mode. It may be necessary to specify VIRT to prevent interference with other jobs running simultaneously, or to allow a user-written routine to fix pages. VIRT should be avoided wherever possible since it has an unfavorable effect on sort/merge performance. VIRT is ignored when the sort/merge program is running in real mode. The value in SORT-CORE-SIZE will be ignored if the OPTION clause is specified.

ALTKW Option

ALTKW specifies an alternate work drive (tape only) in a sorting job. This doubles the maximum input file size allowed. The address of the alternate device must be different from the address of all other devices used in the job.

ERASE Option

ERASE specifies that work data sets used during a sorting operation are to be erased at the end of the job. It is ignored if 2400-series tapes are used for work areas. If the sort operation terminates abnormally,

- ERASE will be performed unless the checkpoint facility has been specified;
- if ERASE is performed, and if a workfile has been pooled with output, the output file will also be erased.

Note that the sort program does not close work data sets, even when terminating normally.

ROUTE Option

```
[,ROUTE=LST  
,ROUTE=LOG]
```

,ROUTE=LST specifies that messages are to be routed to the SYSLST file by the sort/merge program. Messages requiring operator intervention will also be printed on SYSLOG if allocated to a DOS/VS supported console device.

,ROUTE=LOG specified that messages are to be routed to the console.

Note: The default is assumed until the OPTION card has been read.

SORTWK Option

```
[,SORTWK=work  
,SORTWK=(work1,...work )]
```

This operand specifies the logical unit numbers associated with the work files. The parameters within parentheses must be replaced by symbolic unit numbers of a maximum of three significant digits from 1 to 221, or by a comma. When a comma is coded, or if the operand is omitted, the sort program will use the default assignment.

At least one blank must follow the last operand.

SORT-OPTION Clause Examples

SORT-OPTION is SRTOPTN where SRTOPTN is defined in working-storage section. At entry to SORT/MERGE, the contents of SRTOPTN is as follows.

Example 1:

```
OPTION PRINT=ALL,STORAGE=26384,LABEL=(,U)
```

All messages are requested, the virtual storage available to the sort/merge program is 26,384 bytes, and the work volume is unlabeled.

Example 2:

OPTION STORAGE=32K,ERASE,ROUTE=LST,
SORTWK=(005,006)

The PRINT option is not specified, so all messages will be printed by default. The storage available to the sort/merge program is 32K bytes. Standard labels, by default, are assumed for all files. The data sets used by sort are to be erased on completion of the sort operation. All messages are to be routed to the printer. The logical numbers of the work files are SYS005 and SYS006.

Example 3:

OPTION SORTWK=(0 10,11,12,,14,15),ALTWK

1. Assume work=3 (specified on the SELECT statement associated with the SD file); then, using M from Figure 58.2 (M=3) since, in this example, no override for the alternate work unit is specified, allocate as follows:

SYS010, SYS011 and SYS012 are the logical unit numbers of the work files. SYS004 is the logical unit for the alternate work device by default. SYS(M+1)=SYS(3+1)=SYS004.

SYS014 and SYS015 are not used in this application since WORK=3.

This example shows how the values interact. The example may be understood as showing a sort operation which was set up to run with five work files, but which for this particular run, has only three work files. (Note the assumption that work=3.)

2. Assume WORK=5, then

SYS010, SYS011, SYS012, SYS004, and SYS014 are the logical unit numbers of the work files. SYS015 is the logical unit for the alternate device.

Example 4:

A convenient way to specify the OPTION card at execution time is to use the card as a data card on SYSIPT and in the program specify

SORT-OPTION IS EXEC-SORT
.
.
.
ACCEPT EXEC-SORT FROM SYSIPT.

OUTPUT FILE STATEMENTS

The TLBL or DLBL statement file-name must be SORTOUT. Multivolume and/or multiextent output on disk is accomplished by using DOS/VS standards: one DLBL card is supplied for the entire file followed by one EXTENT card for each separate extent that the file occupies on the disk pack or packs. Where the output file is a direct-access multiextent file, only the first EXTENT statement need contain the specified or defaulted symbolic unit name for the output file. Other EXTENT statements may specify any valid symbolic unit name. Figure 58.2 gives the file-names and default symbolic unit names in the sort/merge program.

Use of Device	Filename	Symbolic Unit Name
Work	SORTWK1	SYS001
	.	.
	.	.
	SORTWK9	SYS(M)
ALTWK	SORTALT	SYS(M+1)

M=the number of work files, as specified in the SELECT statement for the SD file.

Figure 58.2. File Name and Default Symbolic Unit Names



Statement	Operands	Comments
OPTION	PRINT={ALL NONE CRITICAL} or PRINT	Default=ALL
	STORAGE=n (n,VIRT) (nK,....)	Default. See discussion.
	LABEL=(, ,work)	Default=standard labels
	ALTWK	
	ERASE	
	ROUTE={LST LOG}	Default Ph0 msg on printer and console and Ph1-3 on console.
	SORTWK= $\left\{ \begin{array}{l} \text{work}_1 \\ \text{(work}_1, \dots, \text{work }) \end{array} \right\}$	Default=(1,2,...m)

Figure 58.3. SUMMARY OF SORT-OPTION Operands

SORT DIAGNOSTIC MESSAGES

The messages generated by the Sort/Merge Feature are listed in the sort publications referenced in the preface.

LINKAGE WITH THE SORT/MERGE FEATURE

To initiate a sort or merge operation, the COBOL object program includes the object time subroutines ILBDSRTO and ILBDMRGO and transfers control to them.

If the INPUT PROCEDURE option of the SORT statement is specified in the source program, exit E15 of the Sort/Merge program is used. At this exit, the record released by the programmer is passed to the Sort/Merge program. Since a dummy Input Procedure will be generated by the compiler when the USING option is specified, records in the USING file are also passed to the Sort/Merge program at exit E15. Records in the USING file of a Merge operation are passed at exit E32.

If the OUTPUT PROCEDURE option of the SORT statement is specified, exit E35 of the Sort/Merge program is used. At this exit, the record returned by the Sort/Merge program is passed to the programmer. Since a dummy Output Procedure is generated by the compiler when the GIVING option is specified, records are also returned at exit E35 and written on this file. Exit E32 is used for the output procedure option of the MERGE statement.

Completion Codes

The Sort/Merge program returns a completion code upon termination and this code is stored in the COBOL special register SORT-RETURN. The codes are:

- 0 -- Successful completion of Sort/Merge
- 02 -- Invalid OPEN -- USING file
- 04 -- Permanent I/O error -- USING file
- 06 -- Invalid OPEN -- GIVING file
- 08 -- Permanent I/O error -- GIVING file
- 10 -- Boundary violation -- GIVING file
- 12 -- Duplicate or out of sequence key -- GIVING file

16 -- Unsuccessful completion of Sort/Merge

Successful Completion: When a Sort/Merge application has been successfully executed, a completion code of zero is returned and the sort operation terminates.

Unsuccessful Completion: If the Sort program encounters an error during execution that will not allow it to complete successfully, it returns a completion code of 16 and terminates. (A possible error is an uncorrectable input/output error.) The sort publications contain a detailed description of the conditions under which this termination will occur.

The user may test the SORT-RETURN register for successful termination of the sort operation, as shown in the following example:

```
SORT SALES-RECORDS ON ASCENDING KEY,  
CUSTOMER-NUMBER, DESCENDING KEY DATE,  
USING FN-1, GIVING FN-2.
```

```
IF SORT-RETURN NOT EQUAL TO ZERO, DISPLAY  
"SORT UNSUCCESSFUL" UPON CONSOLE, STOP  
RUN.
```

Cataloging a Sort Program

When the CATAL option is used to catalog a sort program, the following should be observed:

- To avoid duplicate names when selecting a catalog name for his program, the programmer must be aware of the naming convention used by the compiler to generate the name of the dummy phase into which the phases of the Sort/Merge program will subsequently be loaded.

Naming Convention: The compiler generates the phase card for the dummy phase using the following convention:

- If the PROGRAM-ID name is 6, 7, or 8 characters in length, the dummy phase name consists of the first 6 characters plus 2 zero characters.
- If the PROGRAM-ID name is less than 6 characters in length, the name is padded with zeros to 8 characters.

- Since the system expects the first character of PROGRAM-ID to be alphabetic, the first character, if numeric, is converted as follows:

0 -> J
1-9 -> A-I

The hyphen is converted to zero if it appears as the second through eighth character.

CHECKPOINT/RESTART DURING A SORT

The Checkpoint/Restart Feature is available to the programmer using the COBOL SORT statement. The programmer uses the RERUN clause to specify that checkpoints should be taken during program execution. The control statement requirements for taking a checkpoint are discussed in the section entitled "Program Checkout." Checkpoint/Restart is not available during a merge operation.

The system-name specified in the RERUN clause as the sort checkpoint device must not be the same as any system-name used in

the source language ASSIGN clause, but follows the same rules of formation.

The RERUN clause is fully described in the publication IBM DOS Full American National Standard COBOL.

USING SORT IN A MULTIPHASE ENVIRONMENT

When the Sort program is invoked in a multiphase environment, the following should be noted:

1. It is the programmer's responsibility to ensure that the COBOL program containing the SORT statement is the highest phase in storage.
2. If two programs are compiled, link edited, and executed together, only one program may use the Sort feature. If both programs require Sort, the programs can be compiled separately and then the decks must be organized so that the dummy phase cards for Sort are both together at the end of the deck before they are link edited and executed.

REPORT Clause in a File Description (FD) Entry

A given report-name may appear in a maximum of two file description entries. The file description entries need not have the same characteristics, but both must be standard sequential. If the same report-name is specified in two file description entries, the report will be written on both files. For example:

```
ENVIRONMENT DIVISION.
  SELECT FILE-1 ASSIGN SYS005-UR-1403-S.
  SELECT FILE-2 ASSIGN SYS001-UT-2400-S.
  .
  .
DATA DIVISION.
FD FILE-1 RECORDING MODE F
  RECORD CONTAINS 121 CHARACTERS
  REPORT IS REPORT-A.
FD FILE-2 RECORDING MODE V
  RECORD CONTAINS 101 CHARACTERS
  REPORT IS REPORT-A.
```

For each GENERATE statement, the records for REPORT-A will be written on FILE-1 and FILE-2, respectively. The records on FILE-2 will not contain columns 102 through 121 of the corresponding records on FILE-1.

Summing Techniques

Execution time of an object program can be decreased by keeping in mind that Report Writer source coding is treated as though the programmer had written the program in COBOL without the Report Writer feature. Therefore, a complex source statement or series of statements will generally be executed faster than simple statements that perform the same function. The following example shows two coding techniques for the Report Section of the Data Division. Method 2 uses the more complex statements.

```
RD...CONTROLS ARE YEAR MONTH WEEK DAY.
```

Method 1:

```
01 TYPE CONTROL FOOTING YEAR.
  02 SUM COST.
01 TYPE CONTROL FOOTING MONTH.
  02 SUM COST.
01 TYPE CONTROL FOOTING WEEK.
  02 SUM COST.
01 TYPE CONTROL FOOTING ADAY.
  02 SUM COST.
```

Method 2:

```
01 TYPE CONTROL FOOTING YEAR.
  02 SUM A.
01 TYPE CONTROL FOOTING MONTH.
  02 A SUM B.
01 TYPE CONTROL FOOTING WEEK.
  02 B SUM C.
01 TYPE CONTROL FOOTING ADAY.
  02 C SUM COST.
```

Method 2 will execute faster. One addition will be performed for each day, one more for each week, and one for each month. In Method 1, four additions will be performed for each day.

Use of SUM

Unless each identifier is the name of a SUM counter in a TYPE CONTROL FOOTING report group at an equal or lower position in the control hierarchy, the identifier must be defined in the File, Working-Storage, or Linkage Sections as well as in a TYPE DETAIL report group as a source item or no summing will occur. A SUM counter is algebraically incremented just before presentation of the TYPE DETAIL report group in which the item being summed appears as a source item or the item being summed appeared in a SUM clause that contained an UPON option for this DETAIL report group. This is known as SOURCE-SUM correlation. In the following example, SUBTOTAL is incremented only when DETAIL-1 is generated.



FILE SECTION.

02 NO-PURCHASES PICTURE 99.

REPORT SECTION.

01 DETAIL-1 TYPE DETAIL.
02 COLUMN 30 PICTURE 99 SOURCE
NO-PURCHASES.

01 DETAIL-2 TYPE DETAIL.

01 ADAY TYPE CONTROL FOOTING
LINE PLUS 2.

02 SUBTOTAL COLUMN 30 PICTURE 999
SUM NO-PURCHASES.

01 MONTH TYPE CONTROL FOOTING
LINE PLUS 2 NEXT GROUP
NEXT PAGE.

SUM Routines

A SUM routine is generated by the Report
Writer for each DETAIL report group of the
report. The operands included for summing
are determined as follows:

- 1. The SUM operand(s) also appears in a
SOURCE clause(s) for the DETAIL report
group.
2. The UPON detail-name option was
specified in the SUM clause. In this
case, all the operands are included in
the SUM routine for only that DETAIL
report group, even if the operand
appears in a SOURCE clause in other
DETAIL report groups.

When a GENERATE detail-name statement is
executed, the SUM routine for that DETAIL
report group is executed in its logical
sequence. When GENERATE report-name
statement is executed and the report
contains more than one DETAIL report group,
the SUM routine is executed for each one.
The SUM routines are executed in the

sequence in which the DETAIL report groups
are specified.

The following two examples show the SUM
routines that are generated by the Report
Writer. Example 1 illustrates how operands
are selected for inclusion in the routine
on the basis of simple SOURCE-SUM
correlation. Example 2 illustrates how
operands are selected when the UPON
detail-name option is specified.

Example 1: The following statements are
coded in the Report Section:

01 DETAIL-1 TYPE DE ...
02 ...SOURCE A.
01 DETAIL-2 TYPE DE ...
02 ...SOURCE B.
02 ...SOURCE C.
01 DETAIL-3 TYPE DE ...
02 ...SOURCE B.
01 TYPE CF ...
02 SUM-CTR-1 ...SUM A, B, C.
01 TYPE CF ...
02 SUM-CTR-2 ...SUM B.

A SUM routine is generated for each
DETAIL report group, as follows:

SUM-ROUTINE FOR DETAIL-1
REPORT-SAVE
ADD A TO SUM-CTR-1.
REPORT-RETURN
SUM-ROUTINE FOR DETAIL-2
REPORT-SAVE
ADD B TO SUM-CTR-1.
ADD C TO SUM-CTR-1.
ADD B TO SUM-CTR-2.
REPORT-RETURN
SUM-ROUTINE FOR DETAIL-3
REPORT-SAVE
ADD B TO SUM-CTR-1.
ADD B TO SUM-CTR-2.
REPORT-RETURN

Example 2: This example uses the same coding as Example 1, with one exception: the UPON detail-name option is used for SUM-CTR-1, as follows:

```
01 TYPE CF ...
02 SUM-CTR-1 ...SUM A, B, C
    UPON DETAIL-2.
```

The following SUM routines would then be generated instead of those shown in the previous example:

SUM Routine for DETAIL-1

```
REPORT-SAVE
REPORT-RETURN
```

SUM Routine for DETAIL-2

```
REPORT-SAVE
  ADD A TO SUM-CTR-1.
  ADD B TO SUM-CTR-1.
  ADD C TO SUM-CTR-1.
  ADD B TO SUM-CTR-2.
REPORT-RETURN
```

SUM Routine for DETAIL-3

```
REPORT-SAVE
  ADD B TO SUM-CTR-2.
REPORT-RETURN
```

Output Line Overlay

The Report Writer output line is created using an internal REDEFINES specification, indexed by integer-1. No check is made to prevent overlay on any line. For example:

```
02 COLUMN 10 PICTURE X(23)
  VALUE "MONTHLY SUPPLIES REPORT".
02 COLUMN 12 PICTURE X(9)
  SOURCE CURRENT-MONTH.
```

A length of 27 in column 10, followed by a specification for column 12, will cause field overlay when this line is printed.

Page Breaks

The Report Writer page break routine operates independently of the routines that are executed after any control breaks (except that a page break will occur as the result of a LINE NEXT PAGE clause). Thus, the programmer should be aware of the following facts:

1. A Control Heading is not printed after a Page Heading except for first generation. If the programmer wishes to have the equivalent of a Control

Heading at the top of each page, he must include the information and data to be printed as part of the Page Heading. Since only one Page Heading may be specified for each report, he should be selective in considering his Control Heading because it will be the same for each page, and may be printed at inappropriate times (see "Control Footings and Page Format" in this chapter).

2. GROUP INDICATE items are printed after page and control breaks. Figure 56 contains a GROUP INDICATE clause and illustrates the execution output.

REPORT SECTION.
.
.
01 DETAIL-LINE TYPE IS DETAIL LINE NUMBER IS PLUS 1.
02 COLUMN IS 2 GROUP INDICATE PICTURE IS A(9) SOURCE IS MONTHNAME OF RECORD-AREA (MONTH).
.
.
.
(Execution Output)
JANUARY 15 A00... A02...
PURCHASES AND COST...
JANUARY 21 A03... A03...

Figure 59. Sample of GROUP INDICATE Clause and Resultant Execution Output

WITH CODE Clause

When more than one report is being written on a file and the reports are to be selectively written, a unique 1-character code must be given for each report. A mnemonic-name is specified in the RD-level entry for each report and is associated with the code in the Special-Names paragraph of the Environment Division.

Note: If a report is written with the CODE option, the report should not be written directly on a printer device.

This code will be written as the first character of each record that is written on the file. When the programmer wishes to write a report from this file, he needs



only to read a record, check the first character for the desired code, and have it printed if the desired code is found. The record should be printed starting from the third character, as illustrated in Figure 60.

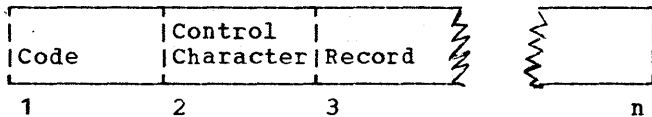


Figure 60. Format of a Report Record When the CODE Clause is Specified

The following example shows how to create and print a report with a code of A. A Report Writer program contains the following statements:

```
ENVIRONMENT DIVISION.
.
.
SPECIAL-NAMES.  "A" IS CODE-CHR-A
                 "B" IS CODE-CHR-B.
.
.
DATA DIVISION.
.
.
REPORT SECTION.
RD  REP-FILE-A   CODE CODE-CHR-A ...
.
.
RD  REP-FILE-B   CODE CODE-CHR-B ...
```

A second program could then be used to print only the report with the code of A, as follows:

```
DATA DIVISION.
FD  RPT-IN-FILE
    RECORD CONTAINS 122 CHARACTERS
    LABEL RECORDS ARE STANDARD
    DATA RECORD IS RPT-RCD.
01  RPT-RCD.
    05  CODE-CHR          PICTURE X.
    05  PRINT-PART.
        10  CTL-CHR       PICTURE X.
        10  RECORD-PART  PICTURE X(120).
FD  PRINT-FILE
    RECORD CONTAINS 121 CHARACTERS
    LABEL RECORDS ARE STANDARD
    DATA RECORD IS PRINT-REC.
```

```
01  PRINT-REC.
    05  FILLER          PICTURE X(121).
.
.
PROCEDURE DIVISION.
.
.
LOOP.  READ RPT-IN-FILE AT END
       GO TO CONTINUE.
       IF CODE-CHR = "A"
       WRITE PRINT-REC FROM PRINT-PART
       AFTER POSITIONING CTL-CHR LINES.
       GO TO LOOP.
CONTINUE.
.
.
.
```

Control Footings and Page Format

Depending on the number and size of Control Footings (as well as the page depth of the report), all of the specified Control Footings may not be printed on the same page if a control break occurs for a high-level control. When a page condition is detected before all required Control Footings are printed, the Report Writer will print the Page Footing (if specified), skip to the next page, print the Page Heading (if specified) and then continue to print Control Footings.

If the programmer wishes all of his Control Footings to be printed on the same page, he must format his page in the RD-level entry for the report (by setting the LAST DETAIL integer to a sufficiently low line number) to allow for the necessary space.

NEXT GROUP Clause

Each time a CONTROL FOOTING report group with a NEXT GROUP clause is printed, the clause is activated only if the report group is associated with the control that causes the break. This is illustrated in Figure 61.

```

RD  EXPENSE-REPORT CONTROLS ARE FINAL,
    MONTH, ADAY
    .
    .
01  TYPE CONTROL FOOTING DAY
    LINE PLUS 1 NEXT GROUP
    NEXT PAGE.
    .
    .
01  TYPE CONTROL FOOTING MONTH
    LINE PLUS 1 NEXT GROUP
    NEXT PAGE.
    .
    .

(Execution Output)

EXPENSE REPORT
    .
    .
January 31.....29.30
    (Output for CF ADAY)
January total.....131.40
    (Output for CF MONTH)

```

Figure 61. Activating the NEXT GROUP Clause

Note: The NEXT GROUP NEXT PAGE clause for the Control Footing DAY is not activated.

Floating First Detail

The first presentation of a body group (PH, PF, CH, CF, DE) that contains a

relative line as its first line will have its relative line spacing suppressed; the first line will be printed on either the value of FIRST DETAIL or INTEGER PLUS 1 of a NEXT GROUP clause from the preceding page. For example:

1. If the following body group was the last to be printed on a page


```

01 TYPE CF NEXT GROUP NEXT PAGE

```

 then this next body group


```

01 TYPE DE LINE PLUS 5

```

 would be printed on value of FIRST DETAIL (in PAGE clause).

2. If the following body group was the last to be printed on a page


```

01 TYPE CF NEXT GROUP LINE 12

```

 and after printing, line-counter = 40, then this next body group


```

01 TYPE DETAIL LINE PLUS 5

```

 would be printed on line 12 + 1 (i.e., line 13).

Report Writer Routines

At the end of the analysis of a report description (RD) entry, the Report Writer routines are generated, based on the contents of the RD. Each routine references the compiler-generated card number of its respective RD.



Subscripts

If a subscript is represented by a constant and if the subscripted item is of fixed length, the location of the subscripted data item within the table or list is resolved during compilation.

If a subscript is represented by a data-name, the location is resolved at execution time. The most efficient format in this case is COMPUTATIONAL, with a PICTURE size less than five integers.

The value contained in a subscript is an integer which represents an occurrence number within a table. Every time a subscripted data-name is referenced in a program, the compiler generates up to 16 instructions to calculate the correct displacement. Therefore, if a subscripted data-name is to be processed extensively, move the subscripted item to an unsubscripted work area, do all necessary processing, and then move the item back into the table. Even when subscripts are described as COMPUTATIONAL, subscripting takes time and storage.

Index-names

Index-names are compiler-generated items, one fullword in length, assigned storage in the TGT (Task Global Table). An index-name is defined by the INDEXED BY clause. The value in an index-name represents an actual displacement from the beginning of the table that corresponds to an occurrence number in the table. Address calculation for a direct index requires a maximum of four instructions; address calculation for a relative index requires a few more. Therefore, the use of index-names in referencing tables is more efficient than the use of subscripts. The use of direct indexes is faster than the use of relative indexes.

Index-names can only be referenced in the PERFORM, SEARCH, and SET statements.

Index Data Items

Index data items are compiler-generated storage positions, one fullword in length,

that are assigned storage within the COBOL program area. An index data item is defined by the USAGE IS INDEX clause. The programmer can use index data items to save values of index-names for later reference.

Great care must be taken when setting values of index data items. Since an index data item is not part of any table, the compiler is unable to change any displacement value contained in an index-name when an index data item is set to the value of an index-name or another index data item. See the SET statement examples later in this chapter.

Index data items can only be referenced in SEARCH and SET statements.

OCCURS Clause

If indexing is to be used to reference a table element and the Format 2 (SEARCH ALL) statement is also used, the KEY option must be specified in the OCCURS clause. A table element is represented by the subject of an OCCURS clause, and is equivalent to one level of a table. The table element must then be ordered upon the key(s) and data-name(s) specified.

DEPENDING ON Option

If a data item described by an OCCURS clause with the DEPENDING ON data-name option is followed by nonsubordinate data items, a change in the value of data-name during the course of program execution will have the following effects:

1. The size of any group described by or containing the related OCCURS clause will reflect the new value of data-name.
2. Whenever a MOVE to a field containing an OCCURS clause with the DEPENDING ON option is executed, the MOVE is done on the basis of the current contents of the object of the DEPENDING ON option.
3. The location of any nonsubordinate items following the item described with the OCCURS clause will be affected by the new value of

TBL
HDLNG

data-name. If the programmer wishes to preserve the contents of these items, the following procedure can be used: prior to the change in data-name, move all nonsubordinate items following the variable item to a work area; after the change in data-name, move all the items back.

Note: The value of data-name may change because a move is made to it or to the group in which it is contained; or the value of data-name may change because the group in which it is contained is a record area that has been changed by execution of a READ statement.

For example, assume that the Data Division of a program contains the following coding:

```
01 ANYRECORD.
   05 A PICTURE S999 COMPUTATIONAL-3.
   05 TABLEA PICTURE S999 OCCURS 100
      TIMES DEPENDING ON A.
   05 GROUPB.
```

Subordinate data items.
End of record.

GROUPB items are not subordinate to TABLEA, which is described by the OCCURS clause. Assuming that WORKB is a work area with the same data structure as GROUPB, the following procedural coding could be used:

```
MOVE GROUPB TO WORKB
```

```
Calculate a new value of A
```

```
MOVE WORKB TO GROUPB
```

The preceding statements can be avoided by placing the OCCURS clause with the DEPENDING ON option at the end of the record.

Note: data-name can also change because of a change in the value of an item that redefines or renames it. In this case, the group size and the location of nonsubordinate items as described in the two preceding paragraphs cannot be determined.

OCCURS CLAUSE WITH THE DEPENDING ON OPTION

If a record description contains an OCCURS clause with the DEPENDING ON option, the record length is variable. This is true for records described in an FD as well as in the Working-Storage section. A previous chapter discussed four different record formats of non-VSAM files. Three of them, V-mode, U-mode, and S-mode, as well

as VSAM files, may contain one or more OCCURS clauses with the DEPENDING ON option.

This section discusses some factors that affect the manipulation of records containing OCCURS clauses with the DEPENDING ON option. The text indicates whether the factors apply to the File or Working-Storage sections, or both.

The compiler calculates the length of V-mode records containing the OCCURS clause with the DEPENDING ON option at three different times, as follows (the first and third applies to FD entries only; the second to both FD and Working-Storage entries):

1. When a file is read and the object of the DEPENDING ON option is within the record.
2. When the object of the DEPENDING ON option is changed as a result of a move to it or any item within its group. (The length is not calculated when a move is made to an item which redefines or renames it.)
3. For an output file, after the record is written, the length is set to maximum to enable a full move of the next record to the buffer. Immediately after the move, the correct length is recalculated as in item 2.

Consider the following example:

```
WORKING-STORAGE SECTION.
```

```
77 CONTROL-1 PIC 99.
77 WORKAREA-1 PIC 9(6)V99.
.
.
.
01 SALARY-HISTORY.
   05 SALARY OCCURS 0 TO 10 TIMES
      DEPENDING ON
      CONTROL-1 PIC 9(6)V99.
```

The Procedure Division statement MOVE 5 TO CONTROL-1 will cause a recalculation of the length of SALARY-HISTORY. MOVE SALARY (5) TO WORKAREA-1 will not cause the length to be recalculated.

The compiler permits the occurrence of more than one level-01 record, containing the OCCURS clause with the DEPENDING ON option, in the same FD entry (see Figure 62). For non-VSAM files, if the BLOCK CONTAINS clause is omitted, the buffer size is calculated from the longest level-01 record description entry. In Figure 62, the buffer size is determined by the description of RECORD-1 (RECORD-1 need not

be the first record description under the FD).

During the execution of a READ statement, the length of each level-01 record description entry in the FD will be calculated (see Figure 62). The length of the variable portion of each record will be the product of the numeric value contained in the object of the DEPENDING ON option and the length of the subject of the OCCURS clause. In Figure 62, the length of FIELD-1 is calculated by multiplying the contents of CONTROL-1 by the length of FIELD-1; the length of FIELD-2, by the product of the contents of CONTROL-2 and the length of FIELD-2; the length of FIELD-3 by the contents of CONTROL-3 and the length of FIELD-3.

Since the execution of a READ statement makes available only one record type (i.e., RECORD-1 type, RECORD-2 type, or RECORD-3 type), two of the three record descriptions in Figure 62 will be inappropriate. In such cases, if the contents of the object of the DEPENDING ON option does not conform to its picture, the length of the corresponding record will be unpredictable. For the contents of an item to conform to its picture:

- An item described as USAGE DISPLAY must contain external decimal data.
- An item described as USAGE COMPUTATIONAL-3 must contain internal decimal data.
- An item described as USAGE COMPUTATIONAL must contain binary data.
- An item described as signed must contain signed data.
- An item described as unsigned must contain unsigned data.

The following example illustrates the length calculations made by the system when a READ statement is executed:

```
FD
.
.
.
01 RECORD-1.
05 A PIC 99.
05 B PIC 99.
05 C PIC 99 OCCURS 5 TIMES
    DEPENDING ON A.

01 RECORD-2.
05 D PIC XX.
05 E PIC 99.
05 F PIC 99.
05 G PIC 99 OCCURS 5 TIMES
    DEPENDING ON F.
```

WORKING-STORAGE SECTION.

```
.
.
.
01 TABLE-3.
05 H PIC99 OCCURS 10 TIMES DEPENDING
    ON B.

01 TABLE-4.
05 I PIC99 OCCURS 10 TIMES DEPENDING
    ON E.
```

When a record is read, lengths are determined as follows:

1. The length of C is calculated using the contents of field A. The length of RECORD-1=A+B+C.
2. The length of G is calculated using the contents of field F. The length of RECORD-2=D+E+F+G.
3. The length of TABLE-3 is calculated using the contents of field B.
4. The length of TABLE-4 is calculated using the contents of field E.

The programmer should be aware of several characteristics of the previously cited length calculations. The following example illustrates a group item (i.e., REC-1) whose subordinate items contain an OCCURS clause with the DEPENDING ON option and the object of that DEPENDING ON option.

TBL
HDLNG

```

FD INPUT-FILE
.
.
DATA RECORDS ARE RECORD-1 RECORD-2 RECORD-3.

01 RECORD-1.
05 CONTROL-1 PIC 99.
05 FIELD-1 OCCURS 0 TO 10 TIMES DEPENDING ON CONTROL-1 PIC 9(5).

01 RECORD-2.
05 CONTROL-2 PIC 99.
05 FIELD-2 OCCURS 1 TO 5 TIMES DEPENDING ON CONTROL-2 PIC 9(4).

01 RECORD-3.
05 FILLER PIC XX.
05 CONTROL-3 PIC 99.
05 FIELD-3 OCCURS 0 TO 10 TIMES DEPENDING ON CONTROL-3 PIC X(4).

```

Figure 62. Calculating Record Lengths When Using the OCCURS Clause with the DEPENDING ON Option

```

WORKING-STORAGE SECTION.
01 REC-1.
05 FIELD-1 PIC 9.
05 FIELD-2 OCCURS 5 TIMES DEPENDING ON
    FIELD-1 PIC X(5).

01 REC-2.
05 REC-2-DATA PIC X(50).

```

The results of executing a MOVE to the group item REC-1 will be affected by the following:

- The length of REC-1 may have been calculated at some time prior to the execution of this MOVE statement.
- The length of REC-1 may never have been calculated at all.
- After the move, since the contents of FIELD-1 have been changed, an attempt will be made to recalculate the length of REC-1. Correct recalculation, however, will only be made if the new contents of FIELD-1 conform to its picture (i.e., USAGE DISPLAY must contain an external decimal item, USAGE COMPUTATIONAL-3 must contain an internal decimal item and USAGE COMPUTATIONAL must contain a binary item. An item described as signed must contain signed data, and an item described as unsigned must contain unsigned data). In the preceding example, if FIELD-1 does not contain an external decimal item, the length of REC-1 will be unpredictable.

Note: According to the COBOL description, FIELD-2 can occur a maximum of five times. If, however, FIELD-1 contains an external decimal item whose value exceeds five, the

length of REC-1 will still be calculated. One possible consequence of this invalid calculation will be encountered if the programmer attempts to initialize REC-1 by moving zeros or spaces to it. This initialization would inadvertently delete part of the adjacent data stored in REC-2.

The following discussion applies to updating a record containing an OCCURS clause with the DEPENDING ON option and at least one other subsequent entry. In this case, the subsequent entry is another item containing an OCCURS clause with the DEPENDING ON option.

```

WORKING-STORAGE SECTION.
01 VARIABLE-REC.
05 FIELD-A PIC X(10).
05 CONTROL-1 PIC 99.
05 CONTROL-2 PIC 99.
05 VARY-FIELD-1 OCCURS 10 TIMES
    DEPENDING ON CONTROL-1 PIC X(5).
05 VARY-FIELD-2 OCCURS 10 TIMES
    DEPENDING ON CONTROL-2 PIC X(9).

01 STORE-VARY-FIELD-2.
05 VARY-FLD-2 OCCURS 10 TIMES
    DEPENDING ON CONTROL-2 PIC X(9).

```

Assume that CONTROL-1 contains the value 5 and VARY-FIELD-1 contains 5 entries.

In order to add a sixth field to VARY-FIELD-1 the following steps are required:

```

MOVE VARY-FIELD-2 TO STORE-VARY-FIELD-2.
ADD 1 TO CONTROL-1.
MOVE 'additional field' TO VARY-FIELD-1
    (CONTROL-1).
MOVE STORE-VARY-FIELD-2 TO VARY-FIELD-2.

```

SET Statement

The SET statement is used to assign values to index-names and to index data items.

When an index-name is set to the value of a literal, identifier, or an index-name from another table element, it is set to an actual displacement from the beginning of the table that corresponds to the occurrence number indicated by the second operand in the statement. The compiler performs the necessary calculations. If an index-name is set to another index-name for the same table, the compiler need make no conversion of the actual displacement value contained in the second operand.

However, when an index data item is set to another index data item or to an index-name, or when an index-name is set to an index data item, the compiler is unable to change any displacement value it finds, since an index data item is not part of any table. Thus, no conversion of values can take place. Remember this to avoid making programming errors.

For example, suppose that a table has been defined as:

```
01 A.
05 B OCCURS 2 INDEXED BY I1, I5.
10 C OCCURS 2 INDEXED BY I2, I6.
15 D OCCURS 3 INDEXED BY I3, I4.
20 E PIC X(20).
20 F PIC 9(5).
```

The table appears in storage as shown in Figure 63.

Suppose that a reference to D (2, 2, 3) is necessary. The following method is incorrect:

```
SET I3 TO 2.
SET INDX-DATA-ITM TO I3.
SET I3 UP BY 1.
SET I2, I1 TO INDX-DATA-ITM.
MOVE D (I1, I2, I3) TO WORKAREA.
```

The value contained in I3 after the first SET statement is 25, which represents the beginning point of the second occurrence of D. When the second SET statement is executed, the value 25 is placed in INDX-DATA-ITM, and the fourth SET statement moves the value 25 into I2 and I1. The third SET statement increases the value in I3 to 50. The calculation for the address D (I1, I2, I3) would then be as follows:

$$\begin{aligned} &(\text{address of D (1, 1, 1)}) + 25 + 25 + 50 \\ &= (\text{address of D (1, 1, 1)}) + 100 \end{aligned}$$

This is not the address of D (2, 2, 3).

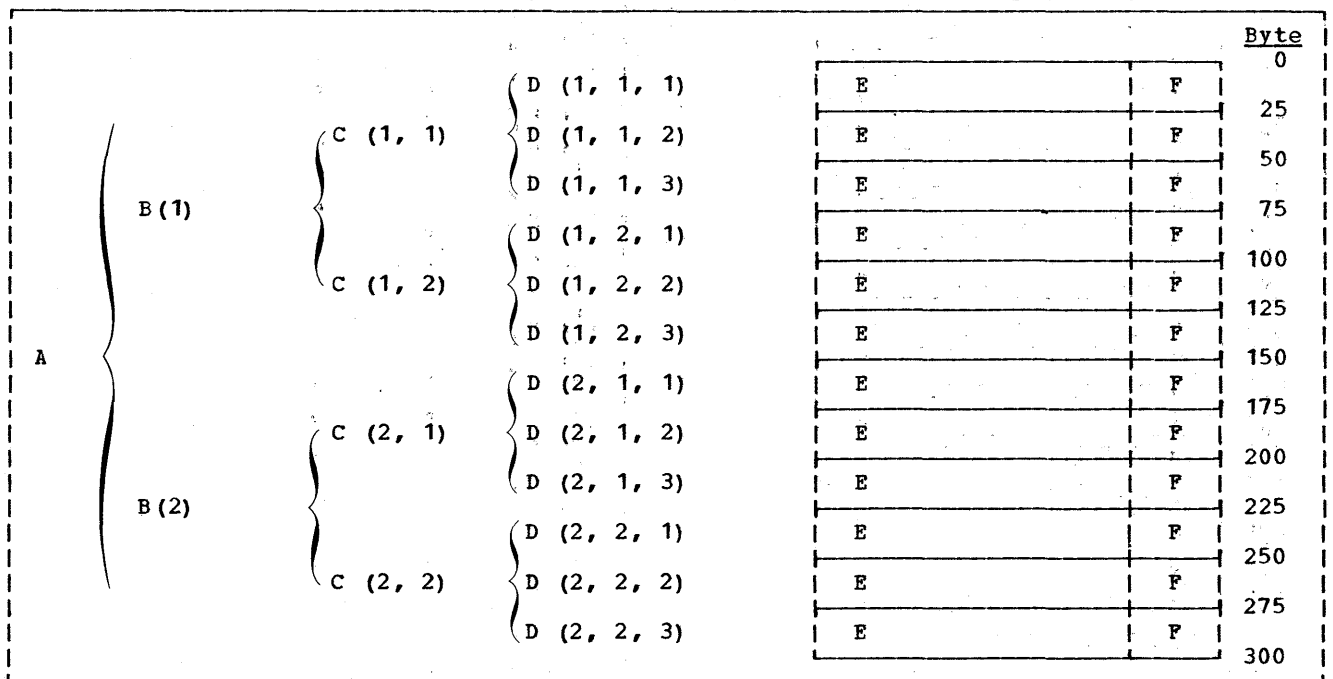


Figure 63. Table Structure in Virtual Storage

The following method will find the correct address:

```
SET I3 TO 2.  
SET I2, I1 TO I3.  
SET I3 UP BY 1.
```

In this case, the first SET statement places the value 25 in I3. Since the compiler is able to calculate the lengths of B and C, the second SET statement places the value 75 in I2, and the value 150 in I1. The third SET statement places the value 50 in I3. The correct address calculation will be:

$$\begin{aligned} & (\text{address of D (1, 1, 1)}) + 150 + 75 + 50 \\ & = (\text{address of D (1, 1, 1)}) + 275 \end{aligned}$$

The rules for the SET statement are shown in Table 33.

Use care when setting the value of index-names associated with tables described as OCCURS DEPENDING ON. If the table entry length is changed, the value contained within the index-name will become invalid unless a new SET statement corrects it.

Table 33. Rules for the SET Statement

Receiving \ Sending	Index-name	Index data item	Identifier or Literal
Index-name	Set to value corresponding to occurrence number ¹	Move without conversion	Set to value corresponding to occurrence number
Index data item	Move without conversion	Move without conversion	Illegal
Identifier	Set to occurrence number represented by index-name	Illegal	Illegal

¹If index-names refer to the same table element, move without conversion.



SEARCH Statement

Only one level of a table (a table element) can be referenced with one SEARCH statement. Note that SEARCH statements cannot be nested, since an imperative statement must follow the WHEN condition, and the SEARCH statement is itself conditional.

To write a series of statements that will search the 3-dimensional table defined in the discussion of the SET statement, the programmer could write:

```
.  
. .  
77  COMPARAND1 PIC X(5).  
77  COMPARAND2 PIC 9(5).  
  
01  A.  
    02 B OCCURS 2 INDEXED BY I1 I5.  
    03 C OCCURS 2 INDEXED BY I2 I6.  
    04 D OCCURS 3 INDEXED BY I3 I4.  
    05 E PIC X(5).  
    05 F PIC 9(5).  
. . .
```

(Initialize COMPARAND1 and COMPARAND2)

```
PERFORM SEARCH-TEST1 THRU SEARCH-EXIT1  
VARYING I1 FROM 1 BY 1 UNTIL I2 IS  
GREATER THAN 2.  
ENTRY-NOENTRY1.  
GO TO ERROR-RECOVERY1.
```

```
SEARCH-TEST1.  
SET I3 TO 1.  
SEARCH D WHEN E (I1, I2, I3) =  
COMPARAND1 AND F (I1, I2, I3) =  
COMPARAND2  
SET I5 TO I1  
SET I6 TO I2  
SET I2 TO 3  
SET I1 TO 3  
ALTER ENTRY-NOENTRY1 TO PROCEED  
TO ENTRY-PROCESSING1.  
SEARCH-EXIT1. EXIT.
```

```
.  
. .  
ERROR-RECOVERY1.  
. . .  
ENTRY-PROCESSING1.  
MOVE E (I5, I6, I3) TO OUTAREA1.  
MOVE F (I5, I6, I3) TO OUTAREA2.  
. . .
```

The PERFORM statement varies the indexes (I1 and I2) associated with table elements B and C; the SEARCH statement varies index I3 associated with table element D.

The values of I1 and I2 that satisfy the WHEN conditions of the SEARCH statement are saved in I5 and I6. I1 and I2 are then both set to 3, so that upon return from the SEARCH statement, control will fall through the PERFORM statement to the GO TO statement.

Subsequent references to the desired occurrence of table elements E and F make use of the index-names I5 and I6 in which the correct value was saved.

Format 1 SEARCH statements perform a serial search of a table. If it is certain that the "found" condition is beyond some intermediate point in the table, the index-names can be set at that point and only that part of the table be searched; this speeds up execution. If the table is large and must be searched from the first occurrence to the last, Format 2 (SEARCH ALL) is more efficient than Format 1, since it uses a binary search technique; however, the table must then be ordered.

In Format 1, the VARYING option allows the programmer to:

- Vary an index-name other than the first index-name stated for this table element. Thus, with two SEARCH statements, each using a different index-name, more than one value can be referenced in the same table element for comparisons, etc.
- Vary an index-name from another table element. In this case, the first index-name specified for this table is used for the SEARCH, and the index-name specified in the VARYING option is incremented at the same time. Thus, the programmer can search two table elements at once.

In Format 1, the WHEN condition can be any relation condition and there can be more than one. If multiple WHEN conditions are stated, the implied logical connective is OR -- that is, if any one of the WHEN conditions is satisfied, the imperative statement following the WHEN condition is executed. If all conditions are to be satisfied before exiting from the SEARCH, the compound WHEN condition with AND as the logical connective must be written.

SEARCH ALL Statement

The SEARCH ALL statement is used to search an entire table for an item without having to write a loop procedure. For example, a programmer-defined table may be the following:

```
01 TABLE.
   05 ENTRY-IN-TABLE OCCURS 90 TIMES
      ASCENDING KEY-1,KEY-2
      DESCENDING KEY-3
      INDEXED BY INDEX-1.
      10 PART-1 PICTURE 9(2).
      10 KEY-1 PICTURE 9(5).
      10 PART-2 PICTURE 9(6).
      10 KEY-2 PICTURE 9(4).
      10 PART-3 PICTURE 9(33).
      10 KEY-3 PICTURE 9(5).
```

A search of the entire table can be initiated with the following instruction:

```
SEARCH ALL ENTRY-IN-TABLE AT END GO TO
NOENTRY WHEN KEY-1 (INDEX-1) = VALUE-1
AND KEY-2 (INDEX-1) = VALUE-2 AND KEY-3
(INDEX-1) = VALUE-3 MOVE PART-1
(INDEX-1) TO OUTPUT-AREA.
```

The preceding instructions will execute a search on the given array TABLE, which contains 90 elements of 55 bytes and 3

keys. The primary and secondary keys (KEY-1 and KEY-2) are in ascending order whereas the least significant key (KEY-3) is in descending order. If an entry is found in which the three keys are equal to the given values (i.e., VALUE-1, VALUE-2, VALUE-3), PART-1 of that entry will be moved to OUTPUT-AREA. If matching keys are not found in any of the entries in TABLE, the NOENTRY routine is entered.

If a match is found between a table entry and the given values, the index (INDEX-1) is set to a value corresponding to the relative position within the table of the matching entry. If no match is found, the index remains at the setting it had when execution of the SEARCH ALL statement began.

Note: It is more efficient to test keys in order of significance (i.e., KEY-1 should be specified before KEY-2 in the WHEN statement). The WHEN statement can only test for equality, and only one side of the equation may be a key.

In Format 2, the SEARCH ALL statement, the table must be ordered on the key(s) specified in the OCCURS clause. Any key may be specified in the WHEN condition, but all preceding data-names in the KEY option

TBL
HDLNG

must also be tested. The test must be an "equal to" (=) condition, and the KEY data-name must be either the subject or object of the condition, or the name of a conditional variable with which the tested condition-name is associated. The WHEN condition can also be a compound condition, formed from one of the simple conditions listed above, with AND as the only logical connective. The KEY data item and the item with which it is compared must be compatible, as given in the rules of the relation test.

Compilation is faster if keys are tested in the SEARCH statement in the same order as they appear in the KEY option.

Note that if KEY entries within the table do not contain valid values, then the results of the binary search will be unpredictable.

Building Tables

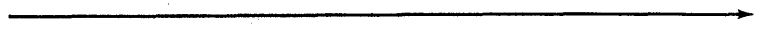
When reading in data to build an internal table:

1. Check to make sure the data does not exceed the space allocated for the table.
2. If the data must be in sequence, check the sequence.
3. If the data contains the subscript that determines its position in the table, check the subscript for a valid range.

When testing for the end of a table, use a named value giving the item count, rather than using a literal. Then, if the table must be expanded, only one value need be changed, instead of all references to a literal.

PART IV

LISTER FEATURE



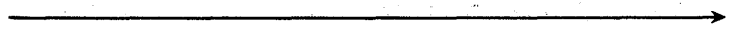
LISTE

SYMBOLIC DEBUGGING FEATURES



SYMDN

PROGRAM CHECKOUT



CHECK
OUT

EXECUTION STATISTICS



EXEC
STAT

This chapter describes the lister feature, a major new facility for optionally producing reformatted source listings with expanded, embedded cross referencing information to increase intelligibility and conserve space. Topics discussed in this chapter include:

- Overall operation of the lister feature
- The output source listing
- The output summary listing
- The optional reformatted output deck
- Using the lister feature

Features of the new source listing include:

- Standard indentation for all Data Division level numbers to show group structure, and for all IF statements and the like in the Procedure Division to show program logic.
- Alignment of PICTURE and VALUE clauses to highlight OCCURS and REDEFINES clauses.
- Two-way, embedded cross-references to eliminate indirect "lookups" (via a separate conventional SXREF listing).
- Reference letters to show the type of reference, indicate overall usage of a program item, and reduce the need to look up each reference.
- Footnotes on Procedure Division pages to show the definition of referenced data items, thereby eliminating more "lookups".
- Two-column Procedure Division pages to compact the listing and further reduce page turning.
- Cross-reference summary to show how, and how much, FD's and Procedure Division section's reference each other.
- Optional reformatted and renumbered source deck for manual use or for updating the BASIS library.

OVERALL OPERATION OF THE LISTER

The lister accepts source programs written in American National Standard COBOL and analyzes the source statements to establish inter-statement references, as well as the type of action resulting from the reference such as redefinition, interrogation, open/close, etc. After scanning the source statements, the lister performs all information transfers necessary for cross-referencing. Finally, the lister composes and prints the reformatted source code.

This reformatted source output follows indenting conventions imposed by the lister to increase readability, and contains cross references between data items and Procedure Division statements, between PERFORM statements and paragraph names, etc. Optionally, the lister produces a new source deck that matches the output listing except that the embedded cross-reference information is omitted.

Thus, the lister can be used to process source decks for uniformity of indenting and for highlighting of IFs, GO TOS, etc., or it can be used simply to obtain a cross-referenced source listing as permanent documentation of a production program, or as an aid in program analysis and debugging. Various options permit printing the Procedure Division listing in two columns to conserve space, and inclusion of BASIS and COPY statements.

The Listing

The reformatted output listing is divided into four parts:

1. A one-page introduction which describes briefly lister codes, conventions, uses
2. The Identification and Environment divisions
3. Detailed, cross-referenced, reformatted Data and Procedure divisions
4. The summary listing

These are described briefly below, and in greater detail in subsequent sections.



The Output Deck

The deck produced optionally by the lister may be saved either in card form or in a BASIS library. This output reflects the output listing, except that cross-reference information is omitted, and that card numbers replace statement numbers. The output deck is described in detail in a subsequent section of this chapter.

Reformatting of Identification and Environment Divisions

The lister reformats the Identification Division statements only by imposing indenting conventions. Statements are indented two spaces. Statements with continuations are indented four spaces.

Environment Division statements are reformatted by imposing indenting conventions and by appending cross-reference information to SELECT statements in the FILE CONTROL section. Thus, in reading the FILE CONTROL section, you receive direct references to the FILE DESCRIPTION statements in the Data Division.

Data Division Reformatting

The lister reformats the Data Division statements principally by imposing indenting conventions on them. In addition, it aligns PICTURE, VALUE, and other clauses vertically to improve readability and facilitate visual checking. This alignment generally highlights REDEFINES and OCCURS clauses, for example. All indenting is with respect to the left margin, which contains the statement number. The indenting conventions are:

- FDs are not indented
- For LEVEL 01 items, the indent is two spaces
- For LEVEL 02 items, indent is four spaces

Level 03 and lower items are each indented two from the last higher level item. Using this convention, the overall structure of each file and group item is immediately apparent when reading the listing. Level 77 items are not indented.

The most striking change in the appearance of the Data Division listing is the addition, at the right of each statement, of cross references that identify the statement number of each Data Division or Procedure Division statement that redefines, changes, reads, tests, or otherwise refers to the data item. When the number of such references is too great to fit on the line, the lister prints as many on the line as there is space for, and prints the remainder as a footnote at the bottom of the page.

Procedure Division Reformatting

The lister reformats the Procedure Division by applying indenting conventions to nested IFs, GO TOs, etc., and by appending cross references to sections and paragraphs, where appropriate, to indicate that the procedure is PERFORMED by another or similar action. It also appends references to the Data Division so that the data item being acted upon can be found quickly. Six codes are used in the Procedure Division:

A ALTER
B (ALTER) to PROCEED TO
E INPUT or OUTPUT procedure for Sort/Merge
G GO TO
P PERFORM
T (PERFORM) THRU

Summary Listing

The summary listing provides an overall view of the relationship among FDs, RDs, and SDs in the program. The entry for each of these major parts of the program consists of a title line showing the statement number and the name of the file, record, or section and a series of counts (by reference type) for each of the categories "intra", "from", and "to". Intra references are those within the section, file, or record, such as REDEFINES and PERFORM operations.

THE SOURCE LISTING

General Appearance

In looking at the source listing of the Identification, Environment, or Data Divisions, you will find that the pages may be considered as having three "columns". The leftmost contains a statement number, or is blank if the line is either a comment or a continuation of the preceding statement or line. The second column contains the reformatted COBOL statements. The third (not present in the Procedure Division) contains references to or from other statements in the source program. Thus, each line of the output listing contains a numbered source statement or its continuation, and a reference or series of references to all other statements in the source program that refer to it. If the series of references is too long to fit on the line, the lister prints as many as will fit, followed by a letter indicating a footnote. The footnote contains the remainder of the references.

The source listing of the Procedure Division is normally printed in double-column format, with each column divided as described above. This format also approximately doubles the span of logic that can be seen on one page or one facing-page spread.

Another characteristic of the source listing is that regardless of whether the source code follows indentation conventions, the lister indents statements according to their type, and according to hierarchy, where applicable. This feature of the lister makes file and record structure immediately visible, and also helps to identify groups of related statements such as IF/ELSE and nesting of IFs.

Format Conventions

New statements are indented from the left margin, which contains the statement number. The lister treats as new statements

- Division headers
- Section headers
- Paragraph names
- Level numbers

- Verbs
- ELSE statements
- OTHERWISE statements
- AT END statements (only when following SEARCH statements)

Indentation of the new statement is made according to the following rules:

1. Data Division
 - FDs and Level 77 items are not indented
 - Level 01 items are indented two spaces in the FILE SECTION or REPORT SECTION and are not indented in the LINKAGE or WORKING-STORAGE sections
 - Each subsequent lower level within an 01 item is indented two spaces more than the preceding higher level
2. Procedure Division
 - Section names are not indented
 - Paragraph names are indented two spaces
 - Unconditionally-executed verbs are indented four spaces
 - Verbs executed under a single condition such as IF or AT END are indented six spaces
 - The first IF statement in a nest of IF statements is indented two spaces; subsequent nested IF statements are indented an additional two spaces at each level
 - ELSE statements are indented to the same position as the IF statement to which they refer
3. Continuation lines (in all divisions) are indented six spaces with respect to the first line of the continued statement

Word spacing within a statement and on continuation lines is usually one space. Within the Data Division, however, PICTURE and VALUE clauses are aligned as nearly as possible into columns so that they may be found and compared easily.

Words are not split at the end of a statement or continuation line unless the word to be split is a nonnumeric literal that will not fit on a single continuation line.



References appear to the right of the statement or continuation line. References following paragraph names appear immediately to the right of the name, separated by a blank. References following other types of statements appear as far to the right as possible depending on the number of blanks available on the line. Each reference consists of a statement number and a type indicator. References in series are separated by commas, and are in ascending order.

Within the Data Division, a reference may also be an alphabetic footnote indicator. The footnote contains a series of references to REDEFINES and Procedure Division statements that refer to that data item.

Within the Procedure Division, the reference may also be a footnote indicator, but the footnote is different in appearance. In the Procedure Division, the footnote is actually an on-page replica of the Data Division statement referred to by the footnoted statement. This replica is complete with all other references to the data item from other portions of the program. To conserve space in the listing, the lister does not repeat a footnote if it appears at the bottom of either of the two preceding pages.

Type Indicators

As mentioned above, a reference consists of a statement number and a type indicator. The type indicator provides immediate information as to what is being done by the statement referred to.

Two sets of type indicators are used by the lister, one for the Data Division, and one for the Procedure Division. Within the Data Division, the type indicators are:

- U Data item unchanged (used as a source field)
- C Data item changed (such as ADD or MOVE)
- E Data item referred to by Environment Division statement (SELECT) or by Procedure Division input/output operation (READ, WRITE)
- D Data item REDEFINED or RENAMED
- Q Queried by IF, WHEN, or UNTIL
- R Referred to by a READ statement

W Referred to by a WRITE, GENERATE, DISPLAY, or similar statement

X Used as an index, subscript, or object of a DEPENDING ON statement

Within the Procedure Division, the type indicators are:

- A ALTER
- B (ALTER) TO PROCEED TO
- E INPUT or OUTPUT procedure (Sort or Merge feature)
- G GO TO
- P PERFORM
- T (PERFORM) THRU

THE SUMMARY LISTING

The summary listing is useful both as an analysis and as a troubleshooting aid. Using the summary listing, the data areas most referred to, the procedures that reference them most often and the nature of those references can be ascertained quickly. The number of references to undefined symbols and the number of incorrectly coded COBOL words can also be ascertained.

General Appearance

Each division or section header, and each FD, RD, or SD begins a new entry in the summary listing. The entry consists of the header line and, beginning on the next line, the total number of each kind of reference to that section from within itself (INTRA), and from outside itself (FROM). These references are followed by similar information for references the section makes to others outside itself (TO).

THE OUTPUT DECK

By specifying the DECK option on the LST card, a new COBOL source deck can be produced that reflects the reformatted source listing. This deck may be saved in a BASIS library (used directly as input to the compiler) or punched onto cards. Because of reformatting, the new deck may contain more cards than the original, but the difference is not great enough to cause any appreciable increase in compilation time. The output deck differs from the listing as follows:

1. References, footnotes, and blank lines are omitted.
2. Literals will be repositioned, if needed, to assure proper continuation.
3. Statement numbers are converted to card numbers.
 - a. The statement number is multiplied by 10, and leading zeros are added as necessary to fill columns 1 through 6.
 - b. Comment and continuation cards are numbered one higher than the preceding card.
 - c. Statement-beginning cards are given the higher of the two numbers produced by the first two rules.

The new deck will permanently process all the BASIS INSERT and DELETE cards, and thus can be used to permanently update the Source Statement Library. This avoids having to resequence the update cards after they have been tested, and avoids the errors incurred during that resequencing process.

USING THE LISTER

Options

The format and contents of the listings and deck produced by the lister are determined by the options specified on the LST card. The LIST card may be placed anywhere between the EXEC statement and the first statement of the COBOL program. It may be placed between any other compiler option cards.

Two format options determine the dimensions and layout of the source and summary listings.

PROC=1col
2col

specifies that the source listing of the Procedure Division will be printed in either single or double column format. At least 132 print positions are required for double column format.

Three options pertain to the output deck:

DECK
NODECK

indicates whether an updated source deck is to be produced as a result of

the lister reformatting and/or the update basis library.

COPYPCH
NOCOPYPCH

will punch updated and reformatted copy libraries as a permanent part of the source when DECK is specified, and will punch out an updated and reformatted copy library when no updated source deck is requested.

LSTONLY
LSTCOMP

The LSTONLY option will give a reformatted listing and a deck, if DECK was specified, but will not compile the program. LISTCOMP will, in addition to listing the source, also compile the program as part of the job step.

PROGRAMMING CONSIDERATIONS

The lister is designed to operate most efficiently on syntactically correct COBOL source, and does not have the expanded error handling of the full compiler. It is therefore highly recommended that the user programs first be compiled using the SYNTAX option, and syntax errors corrected before invoking the lister feature. If the lister function is used and there are syntactical errors, the formatting may be unpredictable, and performance can be significantly impacted.

Further notes: Since Lister reformats the users COBOL program, compilation of the program, if LSTCOMP is in effect, will be different from a non-lister compilation of the same program. For example:

1. Lister sequence numbers may be different
2. SKIP/EJECT cards will have no functional value with LISTER
3. BASIS card will be dropped from the Lister listings
4. FIPS messages will be based on the reformatted Lister listings.
5. Suppress option of COPY will have no effect
6. Sequence checking will not take place for a Lister sum.
7. The Insert card indicator for BASIS will not be indicated on a lister listing.

A programmer using IBM DOS/VS COBOL under the DOS/VS System, has several methods available to him for testing and debugging his programs. Use of the symbolic debugging features is the easiest and most efficient method for testing and debugging and is described in detail in this chapter.

The chapter entitled "Program Checkout" contains information useful for testing and debugging programs run without the symbolic debugging features. It also contains information on compile-time debugging features, linkage editor and execution-time diagnostics as well as a description of taking checkpoints and restarting programs.

The chapter entitled "Execution Statistics" also contains information helpful in testing and debugging programs run both with and without the symbolic debugging features.

USE OF THE SYMBOLIC DEBUGGING FEATURES

There are three symbolic debugging options available to the programmer for object-time debugging: the statement number option, the flow trace option, and the symbolic debugging option. None of these features require source language coding; rather they are requested via the CBL card at compile time. Operation of the symbolic debug option is dependent upon execution-time control cards. Figure 9 illustrates the output generated for each of these features.

STATEMENT NUMBER OPTION

The statement number option facilitates debugging by providing the programmer with information about the statement being executed at the time of an abnormal termination of a job. It identifies the program containing the statement and provides the number of the statement and of the verb being executed.

This feature is requested at compile time via the STATE option of the CBL card. Note that STATE and STXIT, STATE and SYMDMP, and STATE and OPT are mutually exclusive options at compile-time and STATE and STXIT are mutually exclusive in an

execution-time run unit. The CBL card is discussed in detail in the chapter "Preparing COBOL Programs for Processing."

FLOW TRACE OPTION

The flow trace option provides the programmer with the facility for receiving a formatted trace (i.e., a list containing the program identification and statement numbers) corresponding to a variable number of procedures executed prior to an abnormal termination. The number of procedures to be traced is specified by the programmer. If the FLOW option is specified and the number of procedures is not specified, a trace of 99 procedures is provided.

A flow trace is printed only in the event of an abnormal termination. It is requested at compile time via the FLOW option of the CBL card. In a subprogram structure, once a FLOW specification has been made on a program, the subprograms for which a trace is desired should specify FLOW=0. The FLOW=0 specification enables subprograms to utilize the table space reserved previously for the trace; additional table space need not be allocated.

FLOW and STXIT, and FLOW and OPT are mutually exclusive options at compile-time and FLOW and STXIT are mutually exclusive in an execution-time run unit. The CBL card is discussed in the chapter "Preparing COBOL Programs for Processing."

SYMBOLIC DEBUG OPTION

The symbolic debug option produces a symbolic formatted dump of the object program's data area when the program abnormally terminates. It also enables the programmer to request dynamic dumps of specific data-names at strategic points during program execution. If two or more COBOL programs are link edited together and one of them terminates abnormally, the program causing termination and any callers compiled with the symbolic debug option, up to and including the main program, will be given a formatted dump. If any called program contains the SYMDMP option, the main program must be an ANS COBOL program.



Another feature of SYMDMP is that a check is made for a subscript which points out of the program area and for the length of a variable-length move out of the data area. If these address limits are reached, message C170I is issued and an abend dump is given.

The abnormal termination dump consists of the following parts:

1. Abnormal termination message, including the number of the statement and of the verb being executed at the time of an abnormal termination.
2. Selected areas in the Task Global Table.
3. Formatted dump of the Data Division including:
 - (a) for an SD, the statement number, the sort-file-name, the type, and the sort record.
 - (b) for an FD, the statement number, the file-name, the type, SYSnnn, DTF status, the contents of the Pre-DTF and DTF in hexadecimal, and the fields of the record.

for a VSAM file, the file-name, whether the file is open or closed, file organization, type of access, type of last input-output statement, the current contents of the FILE STATUS word, as well as the record fields.
 - (c) for an RD, the statement number, the report-name, the type, the report line, and the contents of PAGE-COUNTER and LINE-COUNTER if present.
 - (d) for an index-name, the name, the type, and the occurrence number in decimal.

Note: For DTFDA when ACCESS IS RANDOM, the actual key is not provided in the Pre-DTF.

The symbolic debug option is requested at compile time via the SYMDMP option of the CBL card. Note that SYMDMP and STXIT, SYMDMP and STATE, and SYMDMP and OPT are mutually exclusive options at compile time and SYMDMP and STATE and STXIT are mutually exclusive in a single execution-time run unit. The CBL card is discussed in the chapter "Preparing COBOL Programs for Processing."

Operation of the symbolic debug option is dependent on object-time control cards

placed in the input stream. These cards are discussed below.

Object-Time Control Cards

The operation of the symbolic debug option is determined by two types of control cards:

Program-control card -- required if abnormal termination and/or dynamic dumps are requested.

Line-control card -- required only if dynamic dumps are requested.

Syntax Rules: The fields of both the program-control card and the line-control card must conform to the following rules:

1. Control cards are essentially free form, i.e., parameters coded on these cards can start in any column. However, parameters may not extend beyond column 71.
2. Each parameter except the last must be immediately followed by a comma.
3. No commas are needed to account for optional parameters that are not specified.
4. All upper-case letters represent specifications that are to appear in the actual statement exactly as shown.
5. All lower-case letters represent generic terms that are to be replaced in the actual statement.
6. Brackets are used to indicate that a specification is optional and is not always required in the statement.
7. Brackets enclosing stacked items indicate that a choice of one item may, but need not, be made by the programmer.
8. Braces enclosing stacked items indicate that a choice of one item must be made by the programmer.
9. All punctuation marks and special characters shown in the statement formats other than hyphens, brackets, braces, and underscores, must be punched exactly as shown. This includes commas, parentheses, and the equal sign.
10. Underscoring indicates the default case.

Continuation Cards: To continue either the program-control card or the line-control card, a nonblank character must be coded in column 72 of the continued card.

SYMDMP

Individual keywords and data-names cannot be split between cards.

Control Statement Placement: The placement of the control cards in the input stream must be as follows:

1. If a main program is compiled with the SYMDMP option, the control cards must precede the programmer's data, if any, in the input stream:

```
// EXEC
  {Control Cards}
/*
  {Programmer's Data}
/*
/ε
```

2. If the main program is a non-COBOL program or a COBOL program compiled without the SYMDMP option, but at least one subprogram called by this main program is a COBOL program compiled with the SYMDMP option, the control cards must follow the programmer's data for the main program, if any:

```
// EXEC
  {Programmer's Data for Main
  Program}
/*
  {Control Cards}
/*
/ε
```

Program-Control Cards: A program-control card must be present at execution time for any program requesting a SYMDMP service. Program-control cards have the following format:

program-id,nnn

```
,SD[=filename] ,ENTRY , (HEX)
,MT[=filename] ,NOENTRY , (NOHEX)
```

program-id

is a one through eight character COBOL program-name. This program-name must be the name of a COBOL program compiled with the SYMDMP option. This parameter is required and must appear first on the program-control card.

nnn

is a 3-digit integer representing the programmer logical unit assigned to the dictionary file produced at compile time (i.e., the SYS005 file.) This parameter is required and must follow the "program-id". This value must be the same as the one specified in the ASSGN control statement for the dictionary file at execution time.

SD[=filename]
MT[=filename]

SD must be specified if the symbolic unit indicated by "nnn" is a disk file; MT must be specified if it is a tape file. "filename" is the name of the dictionary file produced at compile time. For a tape file, the "filename" parameter is ignored. For a disk file, if "filename" is not specified, IJSYS05 will be used. "filename" may be from one to seven characters in length. If "filename" is specified on the CBL card for a disk file, "filename" must also be specified on the program-control card and these names must be identical.



ENTRY
NOENTRY

ENTRY is used to provide a trace of a program-name when several programs are link edited together. Each time the program whose PROGRAM-ID matches the "program-id" parameter is entered, its name is displayed.

(HEX)
(NOHEX)

refers to the format of the Data Division area provided in the abnormal termination dump. If HEX is specified, level-01 items are provided in hexadecimal. Items subordinate to level-01 items are printed in EBCDIC, if possible. Level-77 items are provided both in EBCDIC and hexadecimal. If HEX is not specified, items subordinate to level-01 items and level-77 items are provided in EBCDIC. If unprintable, hexadecimal notation is provided.

Note: Parentheses are required.

Line-Control Cards: Line-control cards have the following format:

line-num [, (verb-num)][, ON n][, m][, k]

{ [, (HEX)]
[, (NOHEX)] , ALL }
{ { [, (HEX)] , name1 [THRU name2]... } }

line-num

corresponds to the generated card number prior to which the dump is desired. The dump is given before the first or only verb on that line. This parameter is required and must be the first on the line-control card.

verb-num

indicates the position of the verb on the specified statement before whose execution a dynamic dump is given. When "verb-num" is not specified, 1 is assumed; when specified, "verb-num" must follow line-num and may not exceed 15.

ON n [,m][,k]

is equivalent to the COBOL statement ON n AND EVERY m UNTIL k. This option limits the requested dynamic dumps to specified times. For example, "ON n" would result in one dump, given the nth time "line-num" is reached during execution. "ON n,m" would result in a dump the first time at the nth execution of "line-num" and thereafter at every mth execution until end-of-job.

(HEX)

(NOHEX)

refers to the format of the Data Division areas provided in the dynamic dump. If HEX is specified, level-01 items are provided in hexadecimal. Items subordinate to level-01 items are printed in EBCDIC, if possible. Level-77 items are printed both in EBCDIC and hexadecimal. If HEX is not specified, items subordinate to level-01 items and level-77 items are provided in EBCDIC. If unprintable, hexadecimal notation is provided. Note that if "name1" is specified and it represents a group item and HEX has not been specified, neither the group nor the elementary items in the group will be provided in hexadecimal.

name1 [THRU name2]

represents selected areas of the Data Division to be dumped. With the THRU option, a range of data-names appearing consecutively in the Data Division is dumped. "name1" and "name2" may be qualified but not subscripted. If the programmer wishes to see a subscripted item, specifying the name of the item without the subscript results in a dump of every occurrence of that item.

ALL

results in a dump of everything that would be dumped in the event of an abnormal termination. The purpose of ALL is to allow the programmer to receive a formatted dump at normal end-of-job. To do this, the generated statement number of the line on which a STOP RUN, EXIT PROGRAM, or GOBACK statement appears must be specified as the "line-num" parameter.

OVERALL CONSIDERATIONS

The end-of-file control card, slash asterisk (/*) must end the symbolic debug control card data set. If a run unit includes one or more programs that have been compiled with the SYMDMP option and no symbolic dump is required at execution time, the input data set must nevertheless be provided, although in this case it consists only of the end-of-file (/*) card.

If no executable output is produced as a result of the compilation (NOLINK, NODECK), any symbolic debugging options specified are suppressed.

SAMPLE PROGRAM -- TESTRUN

Figure 64 is an illustration of a program that utilizes the symbolic debugging features. In the following description of the program and its output, letters identifying the text correspond to letters in the program listing.

- (A) Because the SYMDMP option is requested on the CBL card, the logical unit SYS005 must be assigned at compile time.
 - (B) The CBL card specifications indicate that an alphabetically ordered cross-reference dictionary, a flow trace of 10 procedures, and the symbolic debug option are being requested.
 - (C) An alphabetically ordered cross-reference dictionary of data-names and procedure-names is produced by the compiler as a result of the SXREF specification on the CBL card.
 - (D) The file assigned at compile time to SYS005 to store SYMDMP information is assigned to SYS009 at execution time.
 - (E) The SYMDMP control cards placed in the input stream at execution time are printed along with any diagnostics.
- ① The first card is the program-control card where:
- (a) TESTRUN is the PROGRAM-ID.
 - (b) 9 is the logical unit to which the SYMDMP file is assigned.
 - (c) MT indicates that the SYMDMP file is on tape.
 - (d) (HEX) indicates the format of the abnormal termination dump.

② The second card is a line-control card which requests a (HEX) formatted dynamic dump of KOUNT, NAME-FIELD, NO-OF-DEPENDENTS, and RECORD-NO prior to the first and every fourth execution of generated card number 71.

③ The third card is also a line-control card which requests a (HEX) formatted dynamic dump of WORK-RECORD and B prior to the execution of generated card number 80.

Ⓕ The type code combinations used to identify data-names in abnormal termination and dynamic dumps are defined. Individual codes are illustrated in Table 34.

Ⓖ The dynamic dumps requested by the first line-control card.

Ⓕ The dynamic dumps requested by the second line-control card.

Ⓘ Program interrupt information is provided by the system when a program terminates abnormally.

Ⓙ The statement number information indicates the number of the verb and of the statement being executed at the time of the abnormal termination. The name of the program containing the statement is also provided.

Ⓚ A flow trace of the last 10 procedures executed is provided because FLOW=10 was specified on the CBL card.

Ⓛ Selected areas of the Task Global Table are provided as part of the abnormal termination dump.

Ⓜ For each file-name, the generated card number, the file type, SYSnnn, the DTF status, and the fields of the Pre-DTF and DTF in hexadecimal are provided.

Ⓝ The fields of records associated with each FD are provided in the format requested on the program-control card.

Ⓟ The contents of the fields of the Working-Storage Section are provided in the format requested on the program-control card.

Ⓠ The value associated with each of the possible subscripts are provided for data items described with an OCCURS clause.

Ⓡ Asterisks appearing within the EBCDIC representation of the value of a given field indicate that the type and the actual content of the field conflict.

Note: When using the SYMDMP option, level numbers appear "normalized" in the symbolic dump produced. For example, a group of data items described as:

```
01 RECORDA.  
05 FIELD-A.  
10 FIELD-A1 PIC X.  
10 FIELD-A2 PIC X.
```

will appear as follows in SYMDMP output:

```
01 RECORDA...  
02 FIELD-A...  
03 FIELD-A1...  
03 FIELD-A2...
```

Debugging TESTRUN

1. Referring to the statement number information ② provided by the symbolic debug option, it is learned that the abend occurred during the execution of the first verb on card 80.
2. Generated card number 80 contains the statement COMPUTE B = B + 1.
3. Verifying the contents of B at the time of the abnormal termination ③ it can be seen that the usage of B (numeric packed) conflicts with the value contained in the data area reserved for B (numeric display).
4. The abnormal termination occurred while trying to perform an addition on a display item.

More complex errors may require the use of dynamic dumps to isolate the problem area. Line-control cards are included in TESTRUN merely to illustrate how they are used and the output they produce.

Table 34. Individual Type Codes Used in SYMDMP Output

Code	Meaning
A	Alphabetic
B	Binary
D	Display
E	Edited
*	Subscripted Item
F	Floating Point
N	Numeric
P	Packed Decimal
S	Signed
OT	Overpunch Sign Trailing
OL	Overpunch Sign Leading
SL	Separate Sign Leading
ST	Separate Sign Trailing

```
// JOB DEBUGL
// OPTION NODECK, LINK, LIST, LISTX, SYM, ERRS
// ASSGN SYS005, X'483'
// EXEC FCOBOL
```

(A)

1 IBM DOS VS COBOL

REL 1.0

PP NO. 5746-CBI

07.52.05 10/02/73

```
CBI SXREF,FLCW=10,SYMDMP,QUOTE,SEQ
C0001 000010 IDENTIFICATION DIVISION.
C0002 000020 PROGRAM-ID. TESTRUN.
C0003 000030 AUTHOR. PROGRAMMER NAME.
C0004 000040 INSTALLATION. NEW YORK DEVELOPMENT CENTER.
C0005 000050 DATE-WRITTEN. SEPTEMBER 26,1973.
C0006 000060 DATE-COMPILED. 10/02/73
C0007 000070 REMARKS. THIS PROGRAM HAS BEEN WRITTEN AS A SAMPLE PROGRAM FOR
C0008 000080 COBOL USERS. IT CREATES AN OUTPUT FILE AND READS IT BACK
C0009 000090 AS INPUT.
C0010 000100
C0011 000110 ENVIRONMENT DIVISION.
C0012 000120 CONFIGURATION SECTION.
C0013 000130 SOURCE-COMPUTER. IBM-360-H50.
C0014 000140 OBJFCT-COMPUTER. IBM-370.
C0015 000150 INPUT-CUTPUT SECTION.
C0016 000160 FILE-CONTROL.
C0017 000170 SELECT FILE-1 ASSIGN TO SYS008-UT-2400-S.
C0018 000180 SELECT FILE-2 ASSIGN TO SYS008-UT-2400-S.
C0019 000190
C0020 000200 DATA DIVISION.
C0021 000210 FILE SECTION.
C0022 000220 FD FILE-1
C0023 000230 LABEL RECORDS ARE OMITTED
C0024 000240 BLOCK CONTAINS 5 RECORDS
C0025 000250 RECORDING MODE IS F
C0026 000255 RECORD CONTAINS 20 CHARACTERS
C0027 000260 DATA RECORD IS RECCRD-1.
C0028 000270 01 RECORD-1.
C0029 000290 05 FIELD-A PIC X(20).
C0030 000290 FD FILE-2
C0031 000300 LABEL RECORDS ARE OMITTED
C0032 000310 BLOCK CONTAINS 5 RECORDS
C0033 000320 RECCRD CONTAINS 20 CHARACTERS
C0034 000330 RECORDING MODE IS F
C0035 000340 DATA RECORD IS RECCRD-2.
C0036 000350 01 RECORD-2.
C0037 000350 05 FIELD-A PIC X(20).
```

(B)



Figure 64. Using the Symbolic Debugging Features to Debug the Program TESTRUN (Part 1 of 12)

```

C0038 000370 WORKING-STORAGE SECTION.
C0039 000380 01 FILLER.
C0040          02 KOUNT PIC S99 COMP SYNC.
00041          02 ALPHABET PIC X(26) VALUE "ABCDEFGHIJKLMNOPQRSTUVWXYZ".
00042          02 ALPHA REDEFINES ALPHABET PIC X OCCURS 26 TIMES.
C0043 000420          02 NUMER PIC S99 COMP SYNC.
C0044          02 DEPENDENTS PIC X(26) VALUE "01234012340123401234012340".
C0045          02 DEPEND REDEFINES DEPENDENTS PIC X OCCURS 26 TIMES.
C0046 000450 01 WORK-RECORD.
C0047 000460          05 NAME-FIELD PIC X.
C0048 000470          05 FILLER PIC X.
00049 000480          05 RECORD-NC PIC 9999.
C0050 000490          05 FILLER PIC X VALUE IS SPACE.
00051          05 LOCATION PIC AAA VALUE IS "NYC".
C0052 000510          05 FILLER PIC X VALUE IS SPACE.

```

```

C0053 000520          05 NO-OF-DEPENDENTS PIC XX.
00054          05 FILLER PIC X(7) VALUE IS SPACES.
C0055          01 RECCORDA.
00056          02 A PICTURE S9(4) VALUE 1234.
00057          02 B REDEFINES A PICTURE S9(7) COMPUTATIONAL-3.
C0058 000550 PROCEDURE DIVISION.
C0059          BEGIN.
00060          NOTE THAT THE FOLLOWING OPENS THE OUTPUT FILE
00061          TC BE CREATED AND INITIALIZES THE COUNTERS.
00062          STEP-1. OPEN OUTPUT FILE-1. MOVE ZERO TO KOUNT, NUMER.
00063          NOTE THAT THE FOLLOWING CREATES INTERNALLY THE
00064          RECORDS TO BE CONTAINED IN THE FILE, WRITES THEM
00065          ON TAPE, AND DISPLAYS THEM ON THE CONSOLE.
00066          STEP-2. ADD 1 TO KOUNT, NUMER. MOVE ALPHA (KOUNT) TO
00067          NAME-FIELD.
00068          MOVE DEPEND (KOUNT) TO NO-OF-DEPENDENTS.
00069          MOVE NUMER TO RECORD-NO.
00070          STEP-3. DISPLAY WORK-RECORD UPON CONSOLE.
00071          WRITE RECORD-1 FROM WORK-RECORD.
00072          STEP-4. PERFORM STEP-2 THRU STEP-3
00073          UNTIL KOUNT IS EQUAL TO 26.
00074          NOTE THAT THE FOLLOWING CLOSES THE OUTPUT FILE
00075          AND REOPENS IT AS INPUT.
00076          STEP-5. CLOSE FILE-1. OPEN INPUT FILE-2.
00077          NOTE THAT THE FOLLOWING READS BACK THE FILE
00078          AND SINGLES OUT EMPLOYEES WITH NO DEPENDENTS.
00079          STEP-6. READ FILE-2 RECORD INTO WORK-RECORD AT END GO TO STEP-8.
00080          COMPUTE B = B + 1.
00081          STEP-7. IF NO-OF-DEPENDENTS IS EQUAL TO "0" MOVE "Z" TO
00082          NO-OF-DEPENDENTS. EXHIBIT NAMED WORK-RECORD.
00083          GO TO STEP-6.
00084          STEP-8. CLOSE FILE-2.
00085          STEP-9. STOP RUN.

```

Figure 64. Using the Symbolic Debugging Features to Debug the Program TESTRUN
(Part 2 of 12)

INTRNL NAME	LVL	SOURCE NAME	BASE	DISPL	INTRNL NAME	DEFINITION	USAGE	R	O	Q	M
DNM=1-148	FD	FILE-1	DIF=01		DNM=1-148		DIFMT				F
DNM=1-179	01	RECORD-1	BL=1	000	DNM=1-179	DS 0CL20	GROUP				
DNM=1-200	02	FIELD-A	BL=1	000	DNM=1-200	DS 20C	DISP				
DNM=1-217	FD	FILE-2	DIF=02		DNM=1-217		DIFMT				F
DNM=1-248	01	RECORD-2	BL=2	000	DNM=1-248	DS 0CL20	GROUP				
DNM=1-269	02	FIELD-A	BL=2	000	DNM=1-269	DS 20C	DISP				
DNM=1-289	01	FILLER	BL=3	000	DNM=1-289	DS 0CL56	GROUP				
DNM=1-308	02	KOUNT	BL=3	000	DNM=1-308	DS 1H	COMP				
DNM=1-323	02	ALPHABET	BL=3	002	DNM=1-323	DS 26C	DISP				
DNM=1-341	02	ALPHA	BL=3	002	DNM=1-341	DS 1C	DISP	R	O		
DNM=1-359	02	NUMBER	BL=3	01C	DNM=1-359	DS 1H	COMP				
DNM=1-374	02	DEPENDENTS	BL=3	01E	DNM=1-374	DS 26C	DISP				
DNM=1-394	02	DEPEND	BL=3	01E	DNM=1-394	DS 1C	DISP	R	O		
DNM=1-410	01	WORK-RECORD	BL=3	038	DNM=1-410	DS 0CL20	GROUP				
DNM=1-434	02	NAME-FIELD	BL=3	038	DNM=1-434	DS 1C	DISP				
DNM=1-454	02	FILLER	BL=3	039	DNM=1-454	DS 1C	DISP				
DNM=1-473	02	RECORD-NO	BL=3	03A	DNM=1-473	DS 4C	DISP-NM				
DNM=1-492	02	FILLER	BL=3	03E	DNM=1-492	DS 1C	DISP				
DNM=2-000	02	LOCATION	BL=3	03F	DNM=2-000	DS 3C	DISP				
DNM=2-018	02	FILLER	BL=3	042	DNM=2-018	DS 1C	DISP				
DNM=2-037	02	NO-OF-DEPENDENTS	BL=3	043	DNM=2-037	DS 2C	DISP				
DNM=2-063	02	FILLER	BL=3	045	DNM=2-063	DS 7C	DISP				
DNM=2-082	01	RECORDA	BL=3	050	DNM=2-082	DS 0CL4	GROUP				
DNM=2-102	02	A	BL=3	050	DNM=2-102	DS 4C	DISP-NM				
DNM=2-113	02	B	BL=3	050	DNM=2-113	DS 4P	COMP-3	R			



Figure 64. Using the Symbolic Debugging Features to Debug the Program TESTRUN (Part 3 of 12)

MEMORY MAP	
TGT	00400
SAVE AREA	00400
SWITCH	00448
TALLY	0044C
SORT SAVE	00450
ENTRY-SAVE	00454
SORT CORE SIZE	00458
NSTD-REELS	0045C
SORT RET	0045E
WORKING CELLS	00460
SORT FILE SIZE	00590
SORT MODE SIZE	00594
PGT-VN TBL	00598
TGT-VN TEL	0059C
SORTAB ADDRESS	005A0
LENGTH OF VN TEL	005A4
LNKTH OF SORTAB	005A6
PGM ID	005A8
A(INIT1)	005B0
UPSI SWITCHES	005B4
DEBUG TABLE PTR	005BC
CURRENT PRIORITY	005C0
TA LENGTH	005C1
PROCEDURE BLOCK1 PTR	005C4
UNUSED	005C8
RESERVED	005CC
VSAM SAVE AREA ADDRESS	005D0
UNUSED	005D4
RESERVED	005DC
OVERFLCW CELLS	005F4
BL CELLS	005F4
DTFADR CELLS	00600
FIB CELLS	00608
TEMP STORAGE	00608
TEMP STORAGE-2	00610
TEMP STORAGE-3	00610
TEMP STORAGE-4	00610
BLL CELLS	00610
VLC CELLS	00614
SBL CELLS	00614
INDEX CELLS	00614
SUBADR CELLS	00614
ONCTL CELLS	0061C
PFMCTL CELLS	0061C
PFMSAV CELLS	0061C
VN CELLS	00620
SAVE AREA =2	00624
XSASW CELLS	00624
XSA CELLS	00624
PARAM CELLS	00624
RPTSAV AREA	00628
CHECKPT CTR	00628
IOPTR CELLS	00628
DEBUG TABLE	00628

Figure 64. Using the Symbolic Debugging Features to Debug the Program TESTRUN
(Part 4 of 12)

LITERAL POOL (HEX)

00690 (LIT+0) 00000001 001A1C5B 5BC2D6D7 C5D5405B 5BC2C3D3 D6E2C55B
 006A8 (LIT+24) 5BC2C6C3 D4E4D300 C0000000

DISPLAY LITERALS (BCD)

006B4 (LIT+36) 'WORK-RECORD'

FGT	00638
DEBUG LINKAGE AREA	00638
OVERFLOW CELLS	00640
VIRTUAL CELLS	00640
PROCEDURE NAME CELLS	00664
GENERATED NAME CELLS	00674
SUBDTF ADDRESS CELLS	00688
VNI CELLS	00688
LITERALS	00690
DISPLAY LITERALS	006B4
PROCEDURE BLOCK CELLS	006C0

REGISTER ASSIGNMENT

REG 6 BL =3
 REG 7 BL =1
 REG 8 BL =2

WORKING-STORAGE STARTS AT LOCATION 00100 FOR A LENGTH OF 00058.

0					
59	0006C0		START	EQU *	
	0006C0	58 F0 C 018		L	15,018(0,12) V(ILEEDBG4)
	0006C4	05 EF		BALR	14,15
	0006C6	58 F0 C 01C		L	15,01C(0,12) V(ILEDFLW1)
	0006CA	05 1F		BALR	1,15
	0006CC	003B		DC	X'003E'
62					
	0006CE	58 F0 C 018		L	15,018(0,12) V(ILEEDBG4)
	0006D2	05 EF		BALR	14,15
	0006D4	58 F0 C 01C		L	15,01C(0,12) V(ILEDFLW1)
	0006D8	05 1F		BALR	1,15
	0006DA	003E		DC	X'003E'
62					
	0006DC	58 F0 C 018		L	15,018(0,12) V(ILEEDBG4)
	0006E0	05 EF		BALR	14,15
	0006E2	58 20 D 1F4		L	2,1F4(0,13) EL=1
	0006E6	41 10 C 05F		LA	1,05F(0,12) LIT+7
	0006EA	58 00 D 200		L	0,200(0,13) DTF=1
	0006EE	18 40		LR	4,0
	0006F0	07 00		BCR	0,0
	0006F2	05 F0		BALR	15,0
	0006F4	50 00 F 008		ST	0,008(0,15)

STATISTICS	SOURCE RECORDS =	85	DATA ITEMS =	25	NC CF VERBS =	29		
STATISTICS	PARTITION SIZE =	655176	LINE COUNT =	56	BUFFER SIZE =	512		
OPTIONS IN EFFECT	PMAP RELOC ADR =	NONE	SPACING =	1	FLOW =	10		
OPTIONS IN EFFECT	LISTX	QUOTE	SYM	NOCATLR	LIST	LINK	NCSTXIT	NCLIE
OPTIONS IN EFFECT	NOCLIST	FLAGW	ZWE	NOSUFMAP	NOXREF	ERRS	SKREF	NCOPT
OPTIONS IN EFFECT	NOSTATE	TRUNC	SEQ	SYMDMP	NODECK	NOVERB	NOSYWTAX	NCIVL



Figure 64. Using the Symbolic Debugging Features to Debug the Program TESTRUN (Part 5 of 12)

CROSS-REFERENCE DICTIONARY

C

DATA NAMES	DEFN	REFERENCE
A	000056	
ALPHA	000042	000066
ALPHABET	000041	
B	000057	000080
DEPEND	000045	000068
DEPENDENTS	000044	
FIELD-A	000029	
FIELD-A	000037	
FILE-1	000017	000062 000071 000076
FILE-2	000018	000076 000079 000084
KCUNT	000040	000062 000066 000068 000072
LOCATION	000051	
NAME-FIELD	000047	000066
NC-OF-DEPENDENTS	000053	000068 000081
NUMER	000043	000062 000066 000069
RECORD-NO	000049	000069
RECORD-1	000028	000071
RECORD-2	000036	000079
RECORDA	000055	
WORK-RECORD	000046	000070 000071 000079 000082
BEGIN	000059	
STEP-1	000062	
STEP-2	000066	000072
STEP-3	000070	000072
STEP-4	000072	
STEP-5	000076	
STEP-6	000079	000083
STEP-7	000081	
STEP-8	000084	000079

CARD ERROR MESSAGE

```

00056 ILA2190I-W PICTURE CLAUSE IS SIGNED, VALUE CLAUSE UNSIGNED. ASSUMEL POSITIVE.
00066 ILA5011I-W HIGH ORDER TRUNCATION MIGHT OCCUR.
00066 ILA5011I-W HIGH ORDER TRUNCATION MIGHT OCCUR.

```

// EXEC LNKEDI

JCE DEBUGL DOS LINKAGE EDITOR DIAGNOSTIC OF INPUT

```

ACTION TAKEN MAP REL
LIST AUTOLINK IJFFZZN
LIST AUTOLINK ILBDADRO
LIST AUTOLINK ILBDDBGO
LIST INCLUDE IJJCPDV
LIST AUTOLINK ILBDDSP0
LIST AUTOLINK ILEDLSSO
LIST INCLUDE IJJCPDV
LIST AUTOLINK ILBDFLWO
LIST AUTOLINK ILEDIMLO
LIST AUTOLINK ILBDMNSO
LIST AUTOLINK ILBDSAEQ
LIST ENTRY

```

Figure 64. Using the Symbolic Debugging Features to Debug the Program TESTRUN
(Part 6 of 12)

PHASE	XFR-AD	LOCORE	HICORE	DSK-AD	ESD TYPE	LABEL	LOADED	REL-FR	
PHASE***	07D878	07D878	0803EF	05F OF 4	CSECT	TES1RUN	07D878	07D878	RELOCATABLE
					CSECT	IJFFBZZN	07E368	07E368	
					* ENTRY	IJFFZZZN	07E368		
					* ENTRY	IJFFBZZZ	07E368		
					* ENTRY	IJFFZZZZ	07E368		
					CSECT	ILEDSAE0	080268	080268	
					ENTRY	ILEDSAE1	0802B0		
					CSECT	ILEDMNS0	080260	080260	
					CSECT	ILBDDBG0	07EAC0	07EAC0	
					ENTRY	ILBDDPG5	07EFA2		
					ENTRY	ILBDDPG4	07F014		
					ENTRY	ILBDDBG7	07F038		
					ENTRY	ILBDDBG2	07ED8A		
					* ENTRY	ILBDDBG1	07EC1C		
					* ENTRY	ILBDDBG3	07F00A		
					* ENTRY	ILBDDBG6	07F024		
					ENTRY	STXIIPSW	07F0D0		
					* ENTRY	SORTLP	07F270		
					CSECT	ILBDFLW0	07FD70	07FD70	
					ENTRY	ILBDFLW1	07FE30		
					ENTRY	ILBDFLW2	07FF0C		
					CSECT	ILBDIML0	080208	080208	
					CSECT	ILBDADR0	07E718	07E718	
					* ENTRY	ILEDADR1	07E724		
					CSECT	ILBDDSP0	07F518	07F518	
					ENTRY	ILBDDSP1	07F918		
					CSECT	IJJCPDV	07E878	07E878	
					ENTRY	IJJCPDV1	07E878		
					* ENTRY	IJJCPDV2	07E878		
					CSECT	ILBDDSS0	07FA48	07FA48	
					ENTRY	ILBDDSS1	07FCA8		
					ENTRY	ILBDDSS2	07FCA0		
					ENTRY	ILBDDSS3	07FD60		
					ENTRY	ILBDDSS4	07FA6E		
					ENTRY	ILBDDSS5	07FE1A		
					ENTRY	ILBDDSS6	07FB7A		
					ENTRY	ILBDDSS7	07FB44		
					ENTRY	ILBDDSS8	07FA9E		
* UNREFERENCED SYMBOLS					WXTRN	ILEBSTN0			
					WXTRN	ILEBSR10			
					WXTRN	ILBDTEF3			

003 UNRESOLVED ADDRESS CONSTANTS

```
// ASSGN SYS008,X'482'
// ASSGN SYS009,X'483' ← (D)
// EXEC
```



Figure 64. Using the Symbolic Debugging Features to Debug the Program TESTRUN (Part 7 of 12)

SYMDMP CONTROL CARDS

- ① TESTRUN,009,MT,(HEX)
- ② 71,ON 1,4,(HEX),KOUNT,NAME-FIELD,NO-OF-DEPENDENTS,RECORD-NO
- ③ 80,(HEX),WORK-RECORD,B

NO ERRORS FOUND IN CONTROL CARDS

Ⓣ

TYPE CODES USED IN SYMDMP OUTPUT

CODE	MEANING
A	= ALPHABETIC
AN	= ALPHANUMERIC
ANE	= ALPHANUMERIC EDITED
D	= DISPLAY (STERLING NONREPORT)
DE	= DISPLAY EDITED (STERLING REPORT)
F	= FLOATING POINT (COMP-1/COMP-2)
FD	= FLOATING POINT DISPLAY (EXTERNAL FLOATING POINT)
NB	= NUMERIC BINARY UNSIGNED (COMP)
NB-S	= NUMERIC BINARY SIGNED
ND	= NUMERIC DISPLAY UNSIGNED (EXTERNAL DECIMAL)
ND-OL	= NUMERIC DISPLAY OVERPUNCH SIGN LEADING
ND-OT	= NUMERIC DISPLAY OVERPUNCH SIGN TRAILING
ND-SL	= NUMERIC DISPLAY SEPARATE SIGN LEADING
ND-ST	= NUMERIC DISPLAY SEPARATE SIGN TRAILING
NE	= NUMERIC EDITED
NP	= NUMERIC PACKED DECIMAL UNSIGNED (COMP-3)
NP-S	= NUMERIC PACKED DECIMAL SIGNED
*	= SUBSCRIPTED

Ⓠ

TESTRUN	AT CARD	000071				
LCC	CARD	LV	NAME	TYPE	VALUE	
07D978	000040	02	KOUNT	NB-S (HEX)	+01 0001	
07D9B0	000047	02	NAME-FIELD	AN	A	
07D9EE	000053	02	NO-OF-DEPENDENTS	AN	0	
07D9B2	000049	02	RECORD-NO	ND	0001	
TESTRUN	AT CARD	000071				
LCC	CARD	LV	NAME	TYPE	VALUE	
07D978	000040	02	KOUNT	NB-S (HEX)	+05 0005	
07D9B0	000047	02	NAME-FIELD	AN	E	
07D9EE	000053	02	NO-OF-DEPENDENTS	AN	4	
07D9B2	000049	02	RECORD-NO	ND	0005	
TESTRUN	AT CARD	000071				
LCC	CARD	LV	NAME	TYPE	VALUE	
07D978	000040	02	KOUNT	NB-S (HEX)	+09 0009	
07D9B0	000047	02	NAME-FIELD	AN	I	
07D9EE	000053	02	NO-OF-DEPENDENTS	AN	3	
07D9B2	000049	02	RECORD-NO	ND	0009	

Figure 64. Using the Symbolic Debugging Features to Debug the Program TESTRUN (Part 8 of 12)

TESTRUN LCC	AT CARD CARD	LV NAME	TYPE	VALUE
07D978	000040	02 KCUNT	NE-S (HEX)	+13 000D
07D9E0	000047	02 NAME-FIELD	AN	K
07D9EE	000053	02 NO-OF-DEPENDENTS	AN	2
07D9E2	000049	02 RECCRD-NO	ND	0013
TESTRUN LCC	AT CARD CARD	LV NAME	TYPE	VALUE
07D978	000040	02 KCUNT	NE-S (HEX)	+17 0011
07D9E0	000047	02 NAME-FIELD	AN	Q
07D9EE	000053	02 NO-CF-DEPENDENTS	AN	1
07D9E2	000049	02 RECCRD-NO	ND	0017
TESTRUN LCC	AT CARD CARD	LV NAME	TYPE	VALUE
07D978	000040	02 KCUNT	NE-S (HEX)	+21 0015
07D9E0	000047	02 NAME-FIELD	AN	U
07D9EE	000053	02 NO-CF-DEPENDENTS	AN	0
07D9E2	000049	02 RECCRD-NO	ND	0021
TESTRUN LCC	AT CARD CARD	LV NAME	TYPE	VALUE
07D978	000040	02 KCUNT	NE-S (HEX)	+25 0019
07D9E0	000047	02 NAME-FIELD	AN	Y
07D9EE	000053	02 NO-CF-DEPENDENTS	AN	4
07D9E2	000049	02 RECORD-NO	ND	0025
		(H)		
TESTRUN LCC	AT CARD CARD	LV NAME	TYPE	VALUE
07D9B0	000046	01 WORK-RECORD	(HEX)	C107F0F0 F0F140D5 E8C340F0 40404040 40404040
07D9E0	000047	02 NAME-FIELD	AN	A
07D9E1	000048	02 FILLER	AN	*
07D9E2	000049	02 RECORD-NO	ND	0001
07D9B6	000050	02 FILLER	AN	
07D9E7	000051	02 LOCATION	A	NYC
07D9EA	000052	02 FILLER	AN	
07D9EE	000053	02 NO-CF-DEPENDENTS	AN	0
07D9ED	000054	02 FILLER	AN	
07D9C8	000057	02 B	NP-S (HEX)	*1*2*3* F1F2F3C4



Figure 64. Using the Symbolic Debugging Features to Debug the Program TESTRUN (Part 9 of 12)

COBOL ABEND DIAGNOSTIC AIDS

INIERRUP1 CODE 7 LAST PSW ADDR BEFORE ABEND D007E1A8 (I)
 PROGRAM TESTRUN
 LAST CARD NUMBER/VERB NUMBER EXECUTED -- CARD NUMBER 000060/VERB NUMBER 01. (J)
 FLOW TRACE
 TESTRUN 000066 000070 000066 000070 000066 000070 000066 000070 000076 000079 (K)

DATA DIVISION DUMP OF TESTRUN

(L) TASK GLOBAL TABLE	LOC	VALUE
SAVE AREA	07DC78	0000F233 1B009101 00080168 8007E18E
	07DC9E	0000001A 0007DA60 5007E29A 0007D978
	07DCB8	4007E1A8 0007DEB0
SWITCH	07DCC0	3C10004E
TALLY	07DCC4	00000000
SCRT-SAVE	07DCC8	00000000
ENTRY-SAVE	07DCCC	0007DF38
SCRT-CORE-SIZE	07DCE0	00000000
NSID-REELS	07DCD4	0000
SCRT-RETURN	07DCD6	0000
WORKING CELLS	07DCD8	0007DA60 0007DB40 0000001A 0007DA60
	07DCF8	F0F2F640 E5E8C340 F0404040 40404040
	07DD18	40404040 40404040 4C404040 40404040
	07DD38	--SAME--
	07DD58	40404040 40404040 40404040 40404040
	07DD78	0000001A 01000000 0007E070 0007F518
	07DD98	0007D9AF 5007E29A 0007D978 0007DAD8
	07DDE8	0007DEB0 55E0F088 4780F030 18FE05EF
	07DDF8	0007FA48 0007E084 0007F518 4A50F060
	07DEF8	00000000 00000000 07070607 07070707
SCRT-FILE-SIZE	07DE08	00000000
SCRT-MODE-SIZE	07DE0C	00000000
PGT-VN TBL	07DE10	E2C1D4D7
IGT-VN TBL	07DE14	D3C54040
SORTAB ADDR	07DE18	0007D878
VN TBL LENGTH	07DE1C	0000
SORTAB LENGTH	07DE1E	0000
PROGRAM-ID	07DE20	TESTRUN
A(INIT1)	07DE28	0007D878
UPSI-SWITCHES	07DE2C	C9D3C2C4 D6E2E8F0
IGT-DBG TABLE	07DE34	00000228
CURRENT PRIORITY	07DE38	00
TRANSIENT AREA LENGTH	07DE39	689120
PROCEDURE-BLOCK	07DE3C	E0004780
UNUSED	07DE40	F0325810
RESERVED	07DE44	F0789101
VSAM SAVE AREA	07DE48	10004710
UNUSED	07DE4C	F0329601 00000000
RESERVED	07DE54	00000000 F0549110 000002E0 00000100
OVERFLOW CELLS	(NONE)	
EL CELLS	07DE6C	0007DAEC 0007DBA8 0007D978
DTFADR CELLS	07DE78	0007D9E8 0007DA60
FIB CELLS	(NONE)	
TEMP STORAGE	07DE80	00000000 0000026C
ELL CELLS	07DE88	00000000
VLC CELLS	(NONE)	
SEL CELLS	(NONE)	
INDEX CELLS	(NONE)	
OTHER (SEE MEMORY MAP)	07DE8C	0007D993 0007D9AF 0007E0AC 0007E0AC
	07DEAC	000006F8 D7C5D540 0A000A68 000009D2 5C29F0C8

Figure 64. Using the Symbolic Debugging Features to Debug the Program TESTRUN (Part 10 of 12)

DATA DIVISION DUMP OF TESTRUN

LCC	CARD	LV NAME	TYPE	VALUE
	000017	FD FILE-1 (M)		STANDARD SEQUENTIAL ASSIGNED TO SYS008, CLOSE
07E9D0			PRE-DIF	01010014 00000000 00000000 00000000 6C000000 00000000
07E9E8			DTFMT	00009200 0C000108 0007DA20 0C07DA28 0C07E368 1160E2E8 E2F0FCF8 4040C162
07DA08				00000000 00000000 00000000 86BCF018 41E0E0G1 58201044 0107DAD8 20000064
07EA28				0007DB40 0007DB40 00000014 0007DBA3 00640063 00000000 00000000 40680268
07EA48				01010014 00000000 00000000 00000000 6C000000 00000000 00006200 0C000108
	000028	01 RECORD-1 (N)		(HEX) D8C7F0F0 F1F740D5 E8C340F1 40404040 40404040
07DAEC			AN	C*0017 NYC 1
07EAEC	000029	02 FIELD-A		
	000018	FD FILE-2 (M)		STANDARD SEQUENTIAL ASSIGNED TO SYS008, OPEN INPUT
07DA48			PRE-DIF	01010014 00000000 00000000 00000000 6C000000 00000000
07DA60			DTFMT	00008200 0C000108 0007DA98 0007DAA0 0007E368 11E8E2E8 E2F0FCF8 40400272
07DA80				10000000 2407E1A2 C0000001 86BCF018 41E0E001 58201044 0207DC10 00000064
07EAA0				0007DBA8 0007DBA8 00000014 0007DC0E 00640063 00000000 000802E0 00080268
07DAC0				00000000 00000000 00000000 00000000 00000000 00000000 E907F0FG F2F640D5
	000036	01 RECORD-2 (N)		(HEX) C107F0F0 F0F140D5 E8C340F0 40404040 40404040
07EBA8			AN	A*0001 NYC 0
07EBA8	000037	02 FIELD-A		
	000039	01 FILLER (P)		(HEX) 001AC1C2 C3C4C5C6 C7C8C9D1 D2D3D4D5 D6D7D8D9 E2E3E4E5
07E978				E6E7E8E9 001AF0F1 F2F3F4F0 F1F2F3F4 F0F1F2F3 F4F0F1F2
07D990				F3F4F0F1 F2F3F4F0
07E9A8				
07D978	000040	02 KCUNT	NE-S	+26
07E97A	000041	02 ALPHABET	AN	ABCDEFGHIJKLMNOPQRSTUVWXYZ
	000042	02 ALPHA	*AN	
		(Q) (SUB1)		
07E97A		1		A
07E97E		2		B
07E97C		3		C
07E97D		4		D
07E97E		5		E
07E97F		6		F
07E980		7		G
07E981		8		H
07E982		9		I
07E983		10		J
07E984		11		K
07E985		12		L
07E986		13		M
07E987		14		N
07E988		15		O
07E989		16		P
07E98A		17		Q



Figure 64. Using the Symbolic Debugging Features to Debug the Program TESTRUN (Part 11 of 12)

DATA DIVISION DUMP OF TESTRUN

LCC	CARD	LV NAME	TYPE	VALUE
07D98B		18		R
07D98C		19		S
07D98D		20		T
07E98E		21		U
07D98F		22		V
07E990		23		W
07E991		24		X
07E992		25		Y
07D993		26		Z
07E994	000043	02 NUMER	NE-S	+26
07E996	000044	02 DEPENDENTS	AN	01234012340123401234012340
	000045	02 DEPEND	*AN	

LCC	VALUE
07E996	0
07D997	1
07E998	2
07D999	3
07E99A	4
07D99B	0
07E99C	1
07E99D	2
07E99E	3
07E99F	4
07E9A0	0
07D9A1	1
07E9A2	2
07D9A3	3
07E9A4	4
07E9A5	0
07E9A6	1
07E9A7	2
07E9A8	3
07E9A9	4
07E9AA	0
07D9AE	1
07E9AC	2
07E9AD	3
07E9AE	4
07E9AF	0

LCC	CARD	LV NAME	TYPE	VALUE
07D9B0	000046	01 WORK-RECORD	(HEX)	C107F0F0 F0F140D5 E8C340F0 40404040 40404040
07E9E0	000047	02 NAME-FIELD	AN	A
07D9E1	000048	02 FILLER	AN	*
07E9E2	000049	02 RECORD-NO	ND	0001
07E9E6	000050	02 FILLER	AN	
07E9E7	000051	02 LOCATION	A	NYC
07E9EA	000052	02 FILLER	AN	
07D9EB	000053	02 NO-OF-DEPENDENTS	AN	0

DATA DIVISION DUMP OF TESTRUN

LCC	CARD	LV NAME	TYPE	VALUE
07D9ED	000054	02 FILLER	AN	
	000055	01 RECORDA	(HEX)	F1F2F3C4
07E9C8	000056	02 A	ND-OI	+1234
07D9C8	000057	02 B	NP-S	*1*2*3*

END OF COBOL DIAGNOSTIC AIDS

Figure 64. Using the Symbolic Debugging Features to Debug the Program TESTRUN (Part 12 of 12)

A programmer using the DOS/VS COBOL Compiler and Library has several methods available to him for testing and debugging his programs. Use of the symbolic debugging features is the easiest and most efficient method for testing and debugging and is described in detail in the chapter "Symbolic Debugging Features." Using the execution statistics feature is another method for testing, debugging and optimizing a program, and is described in the chapter "Execution Statistics".

This chapter contains information useful for testing and debugging programs run without the symbolic debugging features. It also contains information on linkage editor and execution-time diagnostics as well as a description of taking checkpoints and restarting programs.

SYNTAX-CHECKING COMPILATION

The compiler checks the source text for syntax errors and then generates the appropriate error messages. With the syntax-checking feature, the programmer can request a compilation either conditionally, with object code produced only if no messages or just W- or C-level messages are generated, or unconditionally, with no object code produced regardless of message level.

Selected test cases run with the syntax-checking feature have resulted in a compilation-time saving of as much as 70%. For a discussion of the syntax-checking options, SYNTAX and CSYNTAX, see the section "CBL Statement -- COBOL Option Control Card."

IDENTIFICATION OF PROGRAM VERSIONS

One problem a programmer may have during checkout is associating a particular compilation listing with the object deck from that compilation and the output and/or dump from a particular run. To aid in this, the following facilities can be used:

1. Specify a DATE-COMPILED paragraph as part of the Environment Division. This is replaced by the actual date of compilation on the source listing (OPTION LIST).

2. The date and time of compilation are given in the header line of the compilation listing.
3. The date and time of compilation are punched into the object deck and will be found beginning at relative location X'EC' in the dump of the object module.
4. By moving the special register WHEN-COMPILED to an output record, the user may flag his output to identify it with a particular compilation. WHEN-COMPILED is described more fully in IBM DOS Full American National Standard COBOL.

DEBUG LANGUAGE

The COBOL debugging language is designed to assist the COBOL programmer in producing an error-free program in the shortest possible time. The following sections discuss the use of the debug language and other methods of program checkout.

The three debug language statements are TRACE, EXHIBIT, and ON. Any one of these statements can be used as often as necessary. They can be interspersed throughout a COBOL source program, or they can be contained in a packet in the input stream to the compiler.

Program checkout may not be desired after testing is completed. A debug packet can be removed after testing to eliminate the extra object program coding generated for the debug statements.

The output produced by the TRACE and EXHIBIT statements is listed on the system logical output device (SYSLSST).

The following discussions describe methods of using the debug language.

FLOW OF CONTROL

The READY TRACE statement causes the compiler-generated card numbers for each section-name and paragraph-name to be displayed. These card numbers are listed on SYSLSST at execution time when control passes to these sections and paragraphs.



Hence, the output of the READY TRACE statement appears as a list of card numbers. If VERB is specified, the actual paragraph-names and names of the verbs will be listed.

To reduce the length of the list and the time taken to generate it, a trace can be stopped with a RESET TRACE statement. The READY TRACE/RESET TRACE combination is helpful in examining a particular area of the program where the flow of control is difficult to determine, e.g., code consists of a series of PERFORM statements or nested conditional statements. The READY TRACE statement can be coded so that the trace begins before control passes to that area. The RESET TRACE statement can be coded so that the trace stops when the program has passed beyond the area.

Use of the ON statement with the TRACE statement allows conditional control of the tracing. When the COBOL compiler encounters an ON statement, it creates a counter which is incremented during execution, whenever control passes through that ON statement. For example, if an error occurs when a specific record is processed, the ON statement can be used to isolate the problem record. The statement should be placed where control passes through it only once for each record that is read. When the contents of the counter equal the number of the record (as specified in the ON statement), a trace can be taken on that record. The following example shows a method in which the 200th record could be selected for a TRACE statement.

```
Col.
1      Area A
-----
      RD-REC.
      .
      .
      .
DEBUG  RD-REC
      PARA-NM-1.    ON 200 READY TRACE.
                   ON 201 RESET TRACE.
```

If the TRACE statement were used without the ON statement, every record would be traced.

An example of a common program error is failing to break a loop or unintentionally creating a loop in the program. If many iterations of the loop are required before it can be determined that a program error exists, the ON statement can be used to initiate a trace after the expected number of iterations has been completed.

Note: If an error occurs during compilation of an ON statement, the diagnostic message may refer to the previous statement number.

DISPLAYING DATA VALUES DURING EXECUTION

A programmer can display the value of a data item during program execution by using the EXHIBIT statement. The EXHIBIT statement has three options:

1. EXHIBIT NAMED -- Displays the names and values of the data-names listed in the statement.
2. EXHIBIT CHANGED -- Displays the value of the data-names listed in the statement only if the value has changed since the last execution of the statement.
3. EXHIBIT CHANGED NAMED -- Displays the names and the values of the data-names only if the values have changed since the last execution of the statement.

Data values can be used to check the accuracy of the program. For example, using EXHIBIT NAMED, the programmer can display specified fields from records, compute the calculations himself, and compare his calculations with the output from his program. The coding for a payroll problem might be:

```
Col.
1      Area A
-----
      .
      GROSS-PAY-CALC.
      COMPUTE GROSS-PAY =
      RATE-PER-HOUR * (HRSWKD
      + 1.5 * OVERTIMEHRS).
      NET-PAY-CALC.
      .
      .
DEBUG  NET-PAY-CALC
      SAMPLE-1. ON 10 AND
      EVERY 10 EXHIBIT NAMED
      RATE-PER-HOUR, HRSWKD,
      OVERTIMEHRS, GROSS-PAY.
```

This coding will cause the values of the four fields to be listed for every tenth data record before net pay calculations are made. The output could appear as:

```
RATE-PER-HOUR = 4.00 HRSWKD = 40.0
OVERTIMEHRS = 0.0 GROSS-PAY = 160.00

RATE-PER-HOUR = 4.10 HRSWKD = 40.0
OVERTIMEHRS = 1.5 GROSS-PAY = 173.23
```

RATE-PER-HOUR = 3.35 HRSWKD = 40.0
OVERTIMEHRS = 0.0 GROSS-PAY = 134.00

Note: Decimal points are included in this example for clarity, but actual printouts depend on the data description in the program.

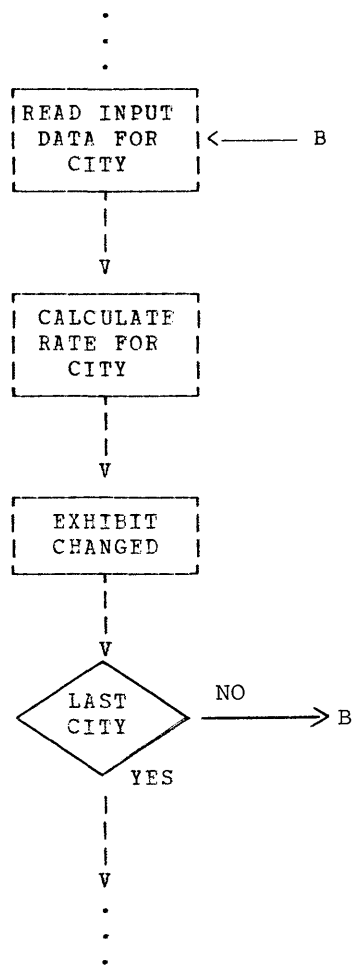
CHECK-
OUT

The preceding was an example of checking at regular intervals (every tenth record). A check of any unusual conditions can be made by using various combinations of COBOL statements in the debug packet. For example:

```
IF OVERTIMEHRS GREATER THAN 2.0
  EXHIBIT NAMED PAYRCDHRS...
```

In connection with the previous example, this statement could cause the entire pay record to be displayed whenever an unusual condition (overtime exceeding two hours) is encountered.

The EXHIBIT statement with the CHANGED option also can be used to monitor conditions that do not occur at regular intervals. The values of data-names are listed only if the value has changed since the last execution of the statement. For example, suppose the program calculates postage rates to various cities. The flow of the program might be:



```

STATE = 01 CITY = 01 RAIL = 10 BUS = 14 TRUCK = 12 AIR = 20
CITY = 02
CITY = 03 BUS = 06 AIR = 15
CITY = 04 RAIL = 30 BUS = 25 TRUCK = 28 AIR = 34
STATE = 02 CITY = 01 TRUCK = 25
CITY = 02 TRUCK = 20 AIR = 30
.
.
.
```

Figure 65. Sample Output of EXHIBIT Statement With the CHANGED NAMED Option

The EXHIBIT statement with the CHANGED option in the program might be:

EXHIBIT CHANGED STATE CITY RATE

The output from the EXHIBIT statement with the CHANGED option could appear as:

```
01 01 10
   02 15
   03
   04 10
02 01
   02 20
   03 15
   04
03 01 10
.
.
.
```

The first column contains the code for a state, the second column contains the code for a city, and the third column contains the code for the postage rate. The value of a data-name is listed only if it has changed since the previous execution. For example, since the postage rate to city 02 and city 03 in state 01 are the same, the rate is not printed for city 03.

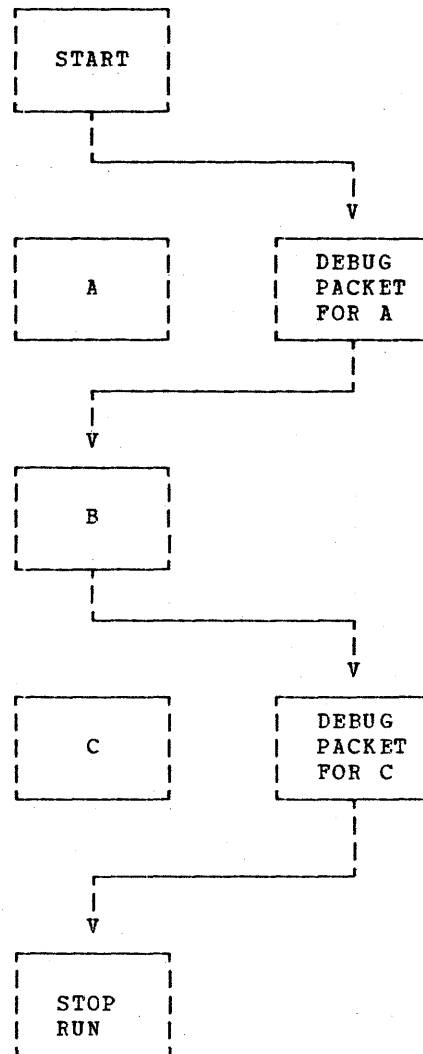
The EXHIBIT statement with the CHANGED NAMED option lists the data-name if the value has changed. For example, the program might calculate the cost of various methods of shipping to different cities. After the calculations are made, the following statement could appear in the program:

EXHIBIT CHANGED NAMED STATE CITY RAIL
BUS TRUCK AIR

The output from this statement could appear as shown in Figure 65. Note that a data-name and its value are listed only if the value has changed since the previous execution.

TESTING A PROGRAM SELECTIVELY

A debug packet allows the programmer to select a portion of the program for testing. The packet can include test data and can specify operations the programmer wants to be performed. When the testing is completed, the packet can be removed. The flow of control can be selectively altered by the inclusion of debug packets, as illustrated in the following example of selective testing of B:



In this program, A creates data, B processes it, and C prints it. The debug packet for A simulates test data. It is first in the program to be executed. In the packet, the last statement is GO TO B, which permits A to be bypassed. After B is executed with the test data, control passes to the debug packet for C, which contains a GO TO statement that transfers control to the end of the program, bypassing C.

TESTING CHANGES AND ADDITIONS TO PROGRAMS

If a program runs correctly, and changes or additions might improve its efficiency, a debug packet can be used to test changes without modifying the original source program.

If the changes to be incorporated are in the middle of a paragraph, the entire

paragraph with the changes included must be written in the debug packet. The last statement in the packet should be a GO TO statement that transfers control to the next procedure to be executed.

There are usually several ways to perform an operation. Alternative methods can be tested by putting them in debug packets.

The source program library facility can be used for program checkout by placing a source program in a library (see the chapter "Librarian Functions"). Changes or additions to the program can be tested by using the BASIS card and any number of INSERT and DELETE cards. Such changes or additions remain in effect only for the duration of the run.

A debug packet can also be used in conjunction with the BASIS card to debug a program or to test deletions or additions to it. The debug packet is inserted in the input stream immediately following the BASIS card and any INSERT or DELETE cards.

DUMPS

If a serious error occurs during execution of the problem program, the job is abnormally terminated; any remaining steps are bypassed; and a program phase dump is generated. The programmer can use the dump for program checkout. (However, any pending transfers to an external device may not be completed. For example, if a READY TRACE statement is in effect when the job is abnormally terminated, the last card number may not appear on the external device.) In cases where a serious error occurs in other than the problem program (e.g., Supervisor), a dump is not produced. Note that program phase dumps can be suppressed if the NODUMP option of the OPTION control statement has been specified for the job, or if NODUMP was specified at system generation time and is not overridden by the DUMP option for the current job.

HOW TO USE A DUMP

When a job is abnormally terminated due to a serious error in the problem program, a message is written on SYSLSST which indicates the:

1. Type of interrupt (e.g., program check)

2. Hexadecimal address of the instruction that caused the interrupt
3. Condition code
4. Reason for the interrupt (e.g., data exception)

The instruction address can be compared to the Procedure Division map. The contents of LISTX provide a relative address for each statement. The load address of the module (which can be obtained from the map of virtual storage generated by the Linkage Editor) must be subtracted from the instruction address to obtain the relative instruction address as shown in the Procedure Division map. The PMAP=nnnnn CBL option can be used to relocate LISTX addresses so that this calculation need not be done. If the interrupt occurred within the COBOL program, the programmer can use the error address and LISTX to locate the specific statement in the program which caused a dump to be taken. Examination of the statement and the fields associated with it may produce information as to the specific nature of the error.

Figure 64 is a sample dump which was caused by a data exception. Invalid data (i.e., data which did not correspond to its usage) was placed in the numeric field B as a result of redefinition. The following discussion illustrates the method of finding the specific statement in the program which caused the dump. Letters identifying the text correspond to letters in the program listing.

- (A) The program interrupt occurred at HEX LOCATION 07DFDC. This is indicated in the SYSLSST message printed just before the dump.
- (B) The linkage editor map indicates that the program was loaded into address 7D878. This is determined by examining the load point of the control section TESTRUN. TESTRUN is the name assigned to the program module by the source coding:

PROGRAM-ID. TESTRUN.

- (C) The specific instruction which caused the dump is located by subtracting the load address from the interrupt address (i.e., subtracting 7D878 from 7DFDC). The result, 764, is the relative interrupt address and can be found in the object code listing. In this case the instruction in question is an AP (add decimal).

CHECK
OUT

- Ⓓ The left-hand column of the object code listing gives the compiler-generated card number associated with the instruction. It is card 66. As seen in the source listing, card 66 contains the COMPUTE statement.

Additional details about reading a dump are found in the chapter "Interpreting Output."

ERRORS THAT CAN CAUSE A DUMP

A dump can be caused by one of many errors. Several of these errors may occur at the COBOL language level while others can occur at the job control level.

The following are examples of COBOL language errors that can cause a dump:

1. A GO TO statement with no procedure-name following it may have been improperly initialized with an ALTER statement. The execution of this statement will cause an invalid branch to be taken and results will be unpredictable.
2. Moves of or arithmetic calculations using numeric fields that have not been properly initialized.

For example, neglecting to initialize the object of an OCCURS clause with the DEPENDING ON option, or referencing data fields prior to the first READ statement may cause a program interrupt and a dump.

3. Invalid data placed in a numeric field as a result of redefinition.
4. Input/output errors that are nonrecoverable.
5. Items with subscripts whose values exceed the defined maximum value can destroy machine instructions when moved.
6. Attempting to execute an invalid operation code through a system or program error.
7. Generating an invalid address for an area that has address protection.
8. Subprogram linkage declarations that are not defined exactly as they are stated in the calling program.
9. Data or instructions can be modified by entering a subprogram and manipulating data incorrectly. A

COBOL subprogram can acquire invalid information from the main program, e.g., a CALL statement using a procedure-name and an ENTRY statement using a data-name.

10. An input file contains invalid data such as a blank numeric field or data incorrectly specified by its data description.

The compiler does not generate a test to check the sign position for a valid configuration before the item is used as an operand. The programmer can test for valid data by means of the numeric class test and, by using the TRANSFORM statement, convert it to valid data under certain conditions.

For example, if the units position of a numeric data item described as USAGE IS DISPLAY contained a blank, the blank could be transformed to a zero, thus forcing a valid sign.

11. Division by zero without an ON SIZE ERROR clause will cause a data exception.

LOCATING A DTF

One or more DTF's are generated by the compiler for each file opened in the COBOL program. All information about that file is found within the DTF or in the fields preceding the DTF. See the chapter "Detailed Processing Capabilities" for the type of information available and its location.

A particular DTF may be located in an execution-time dump as follows:

1. Determine the order of the DTF address cells in the TGT from the DTF numbers shown for each file-name in the glossary.

Note: Since the order is the same as the FD's in the Data Division, the order can be determined from the source program if the SYM option was not used (i.e., no glossary was printed).

2. Find the relative starting address of the block of DTF cells from the TGT listing in the Memory Map.
3. Calculate the absolute starting address of the block by adding the hexadecimal relocation factor for the beginning of the object module as given in the linkage editor MAP.

4. Allowing one fullword per DTF cell, count off the cells from the starting address found in step 3, using the order determined in step 1 to locate the desired DTF cell.
5. If more than one DTF is generated for a file, the above procedure should be followed using the PGT and the SUBDTF cells rather than the TGT and the DTFADR cells. The order of multiple DTF's in storage is dependent on the OPEN option as follows:
 - a. INPUT
 - b. OUTPUT
 - c. I-O or INPUT REVERSED

The following discussion illustrates the method of finding the DTF's in the sample program in Figure 66. Letters identifying the text refer to letters in the program listing.

- (E) The DTF for FILE-1 precedes the DTF for FILE-2.
- (F) DTFADR CELLS begin at relative location 600.
- (G) Since the relocation factor is 7D878, the DTFADR CELLS begin at location 7DE78 in the dump.
- (H) The DTF for FILE-1 begins at location 7D9E8, and the DTF for FILE-2 begins at location 7DA60.

LOCATING DATA

The location assigned to a given data-name may similarly be found by using the BL number and displacement given for that entry in the glossary, and then locating the appropriate one fullword BL cell in the TGT. The hexadecimal sum of the glossary displacement and the contents of the cell should give the relative address of the desired area. This can then be converted to an absolute address as described above.

Since the problem program in Figure 66 interrupted because of a data exception,

the programmer should locate the contents of field B at the time of the interrupt. This can be done as follows:

- (J) Locate data-name B in the glossary. It appears under the column headed SOURCE-NAME. Source-name B has been assigned to base locator 3 (i.e., BL =3) with a displacement of 050. The sum of the value of base locator 3 and the displacement value 50 is the address of data-name B.
- (K) The Register Assignment table lists the registers assigned to each base locator. Register 6 has been assigned to BL =3.
- (L) The contents of the 16 general registers at the time of the interrupt are displayed at the beginning of the dump. Register 6 contains the address 0007D978.
- (M) The location of data-name B can now be determined by adding the contents of register 6 and the displacement value 50. The result, 7D9C8, is the address of the leftmost byte of the 4-byte field B.

Note: Field B contains F1F2F3C4. This is external decimal representation and does not correspond to the USAGE COMPUTATIONAL-3 defined in the source listing.

- (N) The location assigned to a given data-name may also be found by using the BL CELLS pointer in the TGT Memory Map. Figure 64 indicates that the BL cells begin at location 7DE6C (add 5F4 to the load point address, 7D878, of the object module). The first four bytes are the first BL cell, the second four bytes are the second BL cell, etc. Note that the third BL cell contains the value 7D978. This is the same value as that contained in register 6.

Note: Some program errors may destroy the contents of the general registers or the BL cells. In such cases, alternate methods of locating the DTF's are useful.

CHECK-
OUT

```
// JOB DATACHK
// OPTION NODECK, LINK, LIST, LISTX, SYM, ERRS
// EXEC FCOBCL
```

1 IBM DOS VS COBCL

REL 1.0

PP NO. 5746-CE1

07.35.08 10/02/73

```
CEL QUOTE,SEQ
C0001 IDENTIFICATION DIVISION.
C0002 PROGRAM-ID. TESTRUN.
C0003 AUTHOR. PROGRAMMER NAME.
C0004 INSTALLATION. NEW YORK PROGRAMMING CENTER.
C0005 DATE-WRITTEN. SEPTEMBER 25,1973
C0006 DATE-COMPILED. 10/02/73
C0010 ENVIRONMENT DIVISION.
C0011 CONFIGURATION SECTION.
C0012 SOURCE-COMPUTER. IBM-370.
C0013 OBJECT-COMPUTER. IBM-370.
C0014 INPUT-OUTPUT SECTION.
C0015 FILE-CONTROL.
C0016 SELECT FILE-1 ASSIGN TO SYS008-UT-2400-S.
C0017 SELECT FILE-2 ASSIGN TO SYS008-UT-2400-S.
C0018 DATA DIVISION.
C0019 FILE SECTION.
C0020 FD FILE-1
C0021 LABEL RECORDS ARE OMITTED
C0022 BLOCK CONTAINS 5 RECORDS
C0023 RECORDING MODE IS F
C0024 RECORD CONTAINS 20 CHARACTERS
C0025 DATA RECORD IS RECORD-1.
C0026 01 RECORD-1.
C0027 05 FIELD-A PIC X(20).
C0028 FD FILE-2
C0029 LABEL RECORDS ARE OMITTED
C0030 BLOCK CONTAINS 5 RECORDS
C0031 RECORD CONTAINS 20 CHARACTERS
C0032 RECORDING MODE IS F
C0033 DATA RECORD IS RECORD-2.
C0034 01 RECCRD-2.
C0035 05 FIELD-A PIC X(20).
```

E

Figure 66. Sample Dump Resulting from Abnormal Termination (Part 1 of 6)

```

C0036      WORKING-STORAGE SECTION.
C0037      01 FILLER.
C0038          02 KOUNT PIC S99 COMP SYNC.
C0039          02 ALPHABET PIC X(26) VALUE IS "ABCDEFGHIJKLMNQRSTUUVWXYZ".
C0040          02 ALPHA REDEFINES ALPHABET PIC X OCCURS 26 TIMES.
C0041          02 NUMBER PIC S99 COMP SYNC.
C0042          02 DEPENDENTS PIC X(26) VALUE "01234012340123401234012340".
C0043          02 DEPEND REDEFINES DEPENDENTS PIC X OCCURS 26 TIMES.
C0044      01 WORK-RECORD.
C0045          05 NAME-FIELD PIC X.
C0046          05 FILLER PIC X.
C0047          05 RECORD-NO PIC 9999.
C0048          05 FILLER PIC X VALUE IS SPACE.
C0049          05 LOCATION PIC AAA VALUE IS "NYC".
C0050          05 FILLER PIC X VALUE IS SPACE.
C0051          05 NO-OF-DEPENDENTS PIC XX.
C0052          05 FILLER PIC X(7) VALUE IS SPACES.
C0053      01 RECCRDA.
C0054          02 A PICTURE S9(4) VALUE 1234.
C0055          02 B REDEFINES A PICTURE S9(7) COMPUTATIONAL-3.
C0056      PROCEDURE DIVISION.
C0057      BEGIN. READY TRACE.
C0058          NOTE THAT THE FOLLOWING OPENS THE OUTPUT FILE TO BE CREATED
C0059          AND INITIALIZES THE COUNTERS.
C0060      STEP-1. OPEN OUTPUT FILE-1. MOVE ZERO TO KOUNT, NUMBR.
C0061          NOTE THAT THE FOLLOWING CREATES INTERNALLY THE RECORDS TO BE
C0062          CONTAINED IN THE FILE, WRITES THEM ON TAPE, AND DISPLAYS
C0063          THEM ON THE CONSOLE.
C0064      STEP-2. ADD 1 TO KOUNT, NUMBR. MOVE ALPHA (KOUNT) TO
C0065          NAME-FIELD.
C0066          COMPUTE B = B + 1. ← 0
C0067          MOVE DEPEND (KOUNT) TO NO-OF-DEPENDENTS.
C0068          MOVE NUMBER TO RECORD-NO.
C0069      STEP-3. DISPLAY WORK-RECORD UPON CONSOLE. WRITE RECORD-1 FROM
C0070          WORK-RECORD.
C0071      STEP-4. PERFORM STEP-2 THRU STEP-3 UNTIL KOUNT IS EQUAL TO 26.
C0072          NOTE THAT THE FOLLOWING CLOSES THE OUTPUT FILE AND REOPENS
C0073          IT AS INPUT.
C0074      STEP-5. CLOSE FILE-1. OPEN INPUT FILE-2.
C0075          NOTE THAT THE FOLLOWING READS BACK THE FILE AND SINGLES
C0076          OUT EMPLOYEES WITH NO DEPENDENTS.
C0077      STEP-6. READ FILE-2 RECORD INTO WORK-RECORD AT END GO TO STEP-8.
C0078      STEP-7. IF NO-OF-DEPENDENTS IS EQUAL TO "0" MOVE "Z" TO
C0079          NO-OF-DEPENDENTS. EXHIBIT NAMED WORK-RECORD. GO TO STEP-6.
C0080      STEP-8. CLOSE FILE-2.
C0081      STCP RUN.

```



CHECK-
OUT

Figure 66. Sample Dump Resulting from Abnormal Termination (Part 2 of 6)

INTRNL NAME	LVL	SOURCE NAME	BASE	DISPL	INTRNL NAME	DEFINITION	USAGE	R	O	Q	M
DNM=1-148	FD	FILE-1	DTF=01		DNM=1-148		DTFMT				F
DNM=1-179	01	RECORD-1	BL=1	000	DNM=1-179	DS 0CL20	GROUP				
DNM=1-200	02	FIELD-A	BL=1	000	DNM=1-200	DS 20C	DISP				
DNM=1-217	FF	FILE-2	DTF=02		DNM=1-217		DTFMT				F
DNM=1-248	01	RECORD-2	BL=2	000	DNM=1-248	DS 0CL20	GROUP				
DNM=1-269	02	FIELD-A	BL=2	000	DNM=1-269	DS 20C	DISP				
DNM=1-289	01	FILLER	BL=3	000	DNM=1-289	DS 0CL56	GROUP				
DNM=1-308	02	KOUNT	BL=3	000	DNM=1-308	DS 1H	COMP				
DNM=1-323	C2	ALPHABET	BL=3	002	DNM=1-323	DS 26C	LISP				
DNM=1-341	02	ALPHA	BL=3	002	DNM=1-341	DS 1C	DISP	R	O		
DNM=1-359	02	NUMBER	BL=3	01C	DNM=1-359	DS 1H	COMP				
DNM=1-374	02	DEPENDENTS	BL=3	01E	DNM=1-374	DS 26C	DISP				
DNM=1-394	02	DEPEND	BL=3	01E	DNM=1-394	DS 1C	DISP	R	O		
DNM=1-410	01	WORK-RECORD	BL=3	038	DNM=1-410	DS 0CL20	GROUP				
DNM=1-434	02	NAME-FIELD	BL=3	038	DNM=1-434	DS 1C	DISP				
DNM=1-454	02	FILLER	BL=3	039	DNM=1-454	DS 1C	DISP				
DNM=1-473	02	RECORD-NO	BL=3	03A	DNM=1-473	DS 4C	DISP-NM				
DNM=1-492	02	FILLER	BL=3	03E	DNM=1-492	DS 1C	DISP				
DNM=2-000	02	LOCATION	BL=3	03F	DNM=2-000	DS 3C	DISP				
DNM=2-018	02	FILLER	BL=3	042	DNM=2-018	DS 1C	DISP				
DNM=2-037	02	NO-OF-DEPENDENTS	BL=3	043	DNM=2-037	DS 2C	DISP				
DNM=2-063	02	FILLER	BL=3	045	DNM=2-063	DS 7C	DISP				
DNM=2-082	01	RECORDA	BL=3	050	DNM=2-082	DS 0CL4	GROUP				
DNM=2-102	02	A	BL=3	050	DNM=2-102	DS 4C	DISP-NM				
DNM=2-113	02	B	BL=3	050	DNM=2-113	DS 4P	COMP-3				R

MEMORY MAP

TGT	00400
SAVE AREA	00400
SWITCH	00448
TALLY	0044C
SORT SAVE	00450
ENTRY-SAVE	00454
SORT CORE SIZE	00458
NSTD-REELS	0045C
SORT RET	0045E
WORKING CELLS	00460
SORT FILE SIZE	00590
SORT MODE SIZE	00594
PGT-VN TEL	00598
TGT-VN TEL	0059C
SORTAE ADDRESS	005A0
LENGTH OF VN TEL	005A4
LNTH OF SORTAE	005A6
PGM ID	005A8
A(INIT1)	005E0
UPSI SWITCHES	005E4
DEBUG TABLL PTR	005EC
CURRENT PRIORITY	005C0
TA LENGTH	005C1
PROCEDURE BLOCK1 PTR	005C4
UNUSED	005C8
RESERVED	005CC
VSAM SAVE AREA ADDRESS	005E0
UNUSED	005E4
RESERVED	005EC
OVERFLOW CELLS	005F4
EL CELLS	005F4
DTPADR CELLS	00600
FIB CELLS	00608
TEMP STORAGE	00608
TEMP STORAGE-2	00610
TEMP STORAGE-3	00610
TEMP STORAGE-4	00610
BLL CELLS	00610
VLC CELLS	00614
SBL CELLS	00614
INDEX CELLS	00614
SUBADR CELLS	00614
ONCTL CELLS	0061C
PFMCTL CELLS	0061C
PFMSAV CELLS	0061C
VN CELLS	00620
SAVE AREA =2	00624
XSASW CELLS	00624
XSA CELLS	00624
PARAM CELLS	00624
RPTSAV AREA	00628
CHECKPT CTR	00628
IOPTR CELLS	00628

Figure 66. Sample Dump Resulting from Abnormal Termination (Part 3 of 6)

REGISTER ASSIGNMENT

REG 6 BL =3
REG 7 BL =1
REG 8 BL =2

(K)

```

64      000708 48 30 C 03A      LH  3,03A(0,12)      LIT+2
        00070C 4A 30 6 000      AH  3,000(0,6)      DNM=1-308
        000710 4E 30 D 208      CVD 3,208(0,13)      TS=01
        000714 D7 05 D 208 D 208 XC  208(6,13),208(13) TS=01      TS=01
        00071A 94 0F D 20E      NI  20E(13),X'0F'   TS=01+6
        00071E 4F 30 D 208      CVB 3,208(0,13)      TS=01
        000722 40 30 6 000      STH 3,000(0,6)      DNM=1-308
        000726 48 30 C 03A      LH  3,03A(0,12)      LIT+2
        00072A 4A 30 6 01C      AH  3,01C(0,6)      DNM=1-359
        00072E 4E 30 D 208      CVD 3,208(0,13)      TS=01
        000732 D7 05 D 208 D 208 XC  208(6,13),208(13) TS=01      TS=01
        000738 94 0F D 20E      NI  20E(13),X'0F'   TS=01+6
        00073C 4F 30 D 208      CVB 3,208(0,13)      TS=01
        000740 40 30 6 01C      STH 3,01C(0,6)      DNM=1-359

64      000744 41 40 6 002      LA  4,002(0,6)      DNM=1-341
        000748 48 20 6 000      LH  2,000(0,6)      DNM=1-308
        00074C 4C 20 C 03A      MH  2,03A(0,12)      LIT+2
        000750 1A 42              AR  4,2
        000752 5E 40 C 038      S   4,038(0,12)      LIT+0
        000756 50 40 D 214      ST  4,214(0,13)      SES=1
        00075A 58 E0 D 214      L   14,214(0,13)     SBS=1
        00075E D2 00 6 038 E 000 MVC 038(1,6),000(14) DNM=1-434      DNM=1-341

66      000764 FA 30 6 050 C 03C (C)→ AP 050(4,6),03C(1,12) DNM=2-113      LIT+4
67

```

CHECK-OUT

// EXEC LNKEDT

```

PHASE  XFR-AD  LOCORE  HICORE  DSK-AD  ESD TYPE  LABEL  LOADED  REL-FR
PHASE*** 07D878 07D878 07F2AF 05F 0F 4 CSECT  TESTRUN  07D878 07D878 RELOCATELE ← (B)
CSECT    IJFFBZZN 07E278 07E278
* ENTRY  IJFFZZZN 07E278
* ENTRY  IJFFBZZZ 07E278
* ENTRY  IJFFZZZZ 07E278
CSECT    ILEDSAE0 07F128 07F128
ENTRY    ILEDSAE1 07F170
CSECT    ILELMNS0 07F120 07F120
CSECT    ILEDDSP0 07E628 07E628
ENTRY    ILEDDSP1 07EA28
CSECT    ILEDIML0 07F0C8 07F0C8
CSECT    ILEDDSS0 07EDA0 07EDAC
ENTRY    ILEDDSS1 07F000
ENTRY    ILEDDSS2 07EFF8
ENTRY    ILEDDSS3 07F0B8
ENTRY    ILEDDSS4 07EDC6
ENTRY    ILEDDSS5 07EE72
ENTRY    ILEDDSS6 07EED2
ENTRY    ILEDDSS7 07EE9C
ENTRY    ILEDDSS8 07EDF6
CSECT    IJJCPDV  07EB58 07EB58
ENTRY    IJJCPDV1 07EB58
* ENTRY  IJJCPDV2 07EB58
* UNREFERENCED SYMBOLS  WXTRN  SIXITPSW
                          WXTRN  ILBDBG2
002 UNRESOLVED ADDRESS CONSTANTS

```

Figure 66. Sample Dump Resulting from Abnormal Termination (Part 4 of 6)

(A)

(L)

DATACHK									
GR 0-F	0007DE78	0007DF80	00000001	00000001	0007D97A	5007E22C	0007D978	0007DB40	
	0007DBA8	0007E1FC	0007D878	0007D878	0007DEA0	0007DC78	0007E97A	0007E628	
FP REG	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	
CR 0-F	004000FF	0100DD00	FFFFFFFF	FFFFFFFF	00000000	00000000	00000000	00000000	
	00000000	00000000	00000000	00000000	00000000	00000000	C4000000	00000200	
COMREG	EG ADDR IS	000360							

CC0000	00000000	00000000	00000000	00000000	00000000	00000360	00000000	00000000	
CC0020	070E0000	000076CE	040D0000	000074DA	00000000	00000000	070C2000	0000071E	
CC0040	0000DE68	08000000	0000DE58	00000000	EFFCC798	012A88EA	040C0000	000009BE	
CC0060	040C0000	0000697A	C00C0000	0000980C	04C8G000	0000C09E	040C0000	000008F8	
CC0080	0000040C	00000000	00020007	00020001	1207F003	00000000	00000000	00000000	
CC00A0	00000000	00000000	C0000000	000002C0	C0000000	00000000	0000000E	00000000	
CC00C0	00000000	--SANE--							
CC0360	F1F061F0	F261F7F3	F000F000	00000000	00000000	00000000	C4C1E3C1	C3C8D240	
CC0380	0011D7FF	0007F2AF	0007F2AF	00000010	0017D7FF	F97F5CL3	A8A07CD0	00A63891	
CC03A0	38983D4A	3D4B0000	374C3750	375438F1	F0F0F2F7	F3F2F7F5	00003544	00000000	
CC03C0	3EF00000	363C36EC	372C0010	00000010	000070EC	00000000	000030DC	34440000	
CC03E0	00000000	036010E0	00000448	C0C00340	4040404C	40404000	40404040	40404000	
CC0400	0000528C	00002742	063E05FA	00003A40	00001F42	00004058	00004F5E	000073D8	
CC0420	0000F000	0300505E	0014002C	00030000	00005F78	00000000	000072F5	00005204	
CC0440	40800000	0000070E	00008350	00005100	000051B4	0000520C	00100010	00000000	
CC0460	00000000	000065F8	00000000	000030D0	000004F0	000070F5	00003C5A	00000000	
CC0480	00009812	000088C8	00000544	00000000	0000EC88	00007F50	000032FC	00003314	
CC04A0	00081018	00200000	00000000	00000000	00000000	000031A6	00000000	00000000	
CC04C0	00000000	00000000	00000005	03C10401	0000ED00	0000CDF0	0000D5F0	00005378	
CC04E0	000053E8	00000000	00000000	0007D7FF	FF010174	015502A0	FF000045	0000C6E0	
CC0500	0000C0A0	00003868	0000A1E8	00000000	000A0000	00001000	00002000	00003000	
CC0520	070D0000	000076C4	00000360	00000000	8000750E	900075C0	400053C8	00004F5E	
CC0540	00080700	00000000	00000000	00000000	04FC0552	00000000	00000000	00000000	
CC0560	0000318C	0000DEE8	00000008	00020406	080A0C0E	00183048	607890A8	00000000	
CC0580	00000000	C0830083	00830083	00830083	C0830083	00830083	C0830083	00830083	
CC05A0	00830083	--SANE--							
CC05CC	C0830083	00830083	000C0083	83838300	80020000	00007888	00000000	0000BEA5	
CC05E0	06E006E0	06E006E0	06E006E0	4EE0056A	06E006E0	06E006B0	06E006B0	06E006E0	
CC0600	06E006E0	06E006E0	06E006E0	06E006E0	41EB001F	41EB0010	18F69503	04454770	
CC0620	06604590	06D447F0	066006B0	06E006E0	06E006B0	06E006B0	06E041EB	001F06B0	
CC0640	06E041EB	001E4570	06A85890	041441FC	92161211	47700660	41F09314	94633006	
CC0660	42B0A001	58B00514	960CA00F	07FF960C	A00F4400	B528077F	920003CF	928305C9	

Figure 66. Sample Dump Resulting from Abnormal Termination (Part 5 of 6)

DATACHK

```

07D800 D7C8C1E2 C55C5C5C 071D2000 0007DFE2 0007E1FC 0007D878 0007E878 0007DEA0
07D820 0007EC78 0007D97A 0007E628 0007DE78 0007DF80 00000001 00000001 0007D97A
07D840 5007E22C 0007D978 0007DB40 0007DBA8 0000844B E56753BE 0C000000 00000000
07D860 00000000 00000000 00000000 00000000 00000000 00000000 05F00700 9C0EF00A
07D880 47F0F082 0007D878 0007E878 0011D7FF 0000E8E6 0000DB64 8C000015 80000015
07D8A0 0011D7FF 0007F8C8 0A16180C 00000000 182F07F1 0007D878 D7C8C1E2 C55C5C5C
07D8C0 00000000 00000000 00000000 00000000 00000000 00000000 G007E978 0007DAD8
07D8E0 0007EBAB 0007E1FC 0007E878 0007D878 0007DEA0 0007DC78 0C000000 58C0FC66
07D900 58E0C000 58D0F0CA 9500E000 4770F0A2 9610D048 92FFE000 47F0F0AC 98CEF03A
07D920 90ECD00C 185D989F F0BA9110 D0480719 07FF0700 0007E1FC 0C07D878 0C07D878
07D940 0007DEA0 0007DC78 0007EF08 0007E1E2 C3D6C2C6 F3F0F0F0 E3C5E2E3 D9E4D540
07D960 00000000 F1F061F0 F261F7F3 F0F74BF3 F54EF0F8 10604780 0001C1C2 C3C4C5C6
07D980 C7C8C9D1 D2D3D4D5 D6E7D8D9 E2E3E4E5 E6E7E8E9 0001F0F1 F2F3F4F0 F1F2F3F4
07D9A0 F0F1F2F3 F4F0F1F2 F3F4F0F1 F2F3F4F0 C107EB78 000040D5 E8C34000 004040C0
07D9C0 46404040 00000000 F1F2F3C4 00000000 01010014 00000000 00000000 00000000
07D9E0 6C000000 00000000 00009200 00000108 0007DA20 00000000 0007E278 1160E2E8
07LA00 E2F0F0F8 40400162 10000000 04000000 00000000 86BCF018 41E0E001 58201044
07LA20 0107EAD8 20000064 0007EB40 0007DB40 000C0014 0007DBA3 00640063 00000000

```

(H)

(M)

```

07DA40 00000000 4007F128 01010014 00000000 00000000 00000000 0C000000 00000000
07DA60 00008200 00000108 0007DA98 00000000 0007E278 1168E2E8 E2F0F0F8 40400272
07DA80 00000000 20000000 00000000 86BCF018 41E0E001 58201044 0267DEA8 0C000064
07LAA0 0007DC10 00000000 0000C014 00000000 00640063 00000000 0C07F170 0007F128
07LAC0 00000000 --SAME--
07DB00 00000000 00000000 00008000 00000107 0007EB90 00000000 0007E450 02050202
07DE20 0007ECF8 00000000 0207DD10 2C000014 4120E000 47000000 C000FFFF FFFF0000
07DB40 FFFFFFFF FFFFFFFF FFFFBEE1 20000082 4710F132 47F0F15E 91084015 00000000
07DB60 00008000 00000107 0007DBE8 00000000 0007F3E8 02050202 0007CCE0 00000000
07DB80 0207DD38 20000050 4120E000 47000000 0000F0C4 18E44BE0 F24C430E 0C008900
07DEA0 00198800 00190600 12004780 F074420E 00009601 F23247F0 F0464800 F23047F0
07DEBC 014C9108 40154710 F0949180 40154710 F0E84100 000847F0 F0C418E4 4EE0E24E
07DEE0 430E0001 89000019 88000019 43E00000 89E00019 88E00019 19E04780 F08C4100
07DC00 001447F0 F0C44100 000C1B40 58E40000 1A4041EE 000012EE 4780F132 50E0F1F2
07DC20 58140038 5010F20A 4110F20A 5010F202 D201F200 40069108 40154780 F11C18E4
07DC40 4BE0F250 D201F210 E0005810 F2420A00 91801002 4710F114 0A079108 40204710
07DC60 F1A85820 F20A4122 000058E0 F23E58F0 F1F207FF 47F0F104 0000F233 1E009101
07DC80 F2324780 F1444800 F22E9400 F2325820 F23A58E0 F23E58F4 0100A0A9 41E0E11C
07DCA0 47F0F16E 41E0F14C 9101F233 4780F16E 0010004E 00000000 0G000000 0C000000
07DCC0 7010004E 00000000 00000000 0007EF08 00000000 00000000 0007D9E8 0007DE40
07DCE0 0000D8E6 0007D9E8 5007E22C F16E9025 F21290EC 40404040 40404040 40404040
07DED0 40404040 --SAME--
07DE60 40404040 40404040 40404040 40400000 00000000 0007DF78 0007DE40 01000000
07DE80 4007DF54 0007E628 0007DE78 0007DF80 0007DB40 0000D8E6 0007E9E8 5007E22C
07DEA0 0007D978 0007DB40 0007DBA8 0007E1FC 0007D878 0007D878 0C07DEA0 0007DF80
07DEC0 0007E628 0007DE78 0007E000 0007DB40 00014720 0007DF78 0007EDA0 4007E88E
07DEE0 50002000 4A50F060 4A20F060 183047F0 F036E200 50002000 00000000 00000000
07DE00 07070607 07070707 00000000 00000000 E2C1D4D7 D3C54040 C007D878 0C000000
07DE20 E3C5E2E3 D9E4D540 0007D878 C9D3C2C4 D6E2E8F0 F3F09002 D0689120 E0004780
07DE40 F0325810 F0789101 10004710 F0329601 00C00000 00000000 F0549110 0C0002E0
07DE60 00000100 000002A8 D4C5E3C8 0007DB40 0007DBA8 0007D978 0007E9E8 0007DEA0
07DE80 00000000 0000001C 00000000 0007E97A 9802E068 47E00002 0007E05E D7C5D540
07DEA0 0007F120 0007E628 0007F0C8 0007DF70 0007E05E 0007E100 0C07E14C 0007E1A4
07DEC0 0007E07E 0007E092 0007E146 0007E174 0007E05E 400E1F88 00000001 1C000C1A
07DEE0 5E5BC2D6 D7C5D540 5E5BC2C3 D3D6E2C5 5E5BC2C6 C3D4E4D3 C0000000 E6D6D9D2
07DF00 60D9C5C3 D6D9C4BE 58F0C004 051F0001 4004F5F7 404040D2 9640D048 58F0C004
07DF20 051F0001 4004F6F0 40404076 5820D1F4 4110C040 5800D200 184005F0 50002008
07DF40 4500F00C 0007D9E8 0A024100 D20058F0 C00805EF 5810D200 96101620 502001F4
07DF60 5870D1F4 D2016000 C038D201 601CC038 58F0C004 051F0001 4004F6F4 40404004
07DF80 4830C03A 4A306000 4E30D208 D705D208 D208940F D20E4F30 D2084030 60004830
07DFA0 C03A4A30 601C4E30 D208D705 D208D208 94CFE20E 4F30D208 4030601C 41406002
07DFC0 48206000 4C20C03A 1A425B40 C0385040 D21458E0 D214D200 6038E000 FA306050
07DFE0 C03C4140 601E4820 60004C20 C03A1A42 5B40C038 5040D218 58E0E218 D2000403
07E000 E0009240 60444830 601C4E30 D208F331 603AC20E 96F0603D 58F0C004 051F0001
07E020 4004F6F9 404040F0 58F0C004 051F0002 00C00014 0D0001FC 0C38FFFF D2137000
07E040 60385810 D2001841 58F01010 45E0F00C 5020E1F4 5870D1F4 5810D220 07F158F0
07E060 C004051F 0C014004 F7F14040 400F5800 D2205000 D21C5800 C0205000 D2204830
07E080 60004930 C03E58F0 C024078F 5810C00C 07F15800 D21C5000 D22058F0 C004051F
07E0A0 00014004 F7F44040 40585810 D20094EF 10201801 18404110 C04805F0 5000F008
07E0C0 4500F00C 00000000 0A025800 D2004110 C0500A02 4110C040 5800D204 184005F0
07E0E0 5000F008 4500F00C 00000000 0A024100 D20458F0 C00805EF 5810D204 96101020
07E100 58F0C004 051F0001 4004F7F7 40404040 5810C204 58F0C028 91201010 07F1F841
07E120 41F0C028 D2021025 F00158F0 101045E0 F0085020 D1F85880 D1F8D213 60388000
07E140 58F0C018 07FF5810 C01C07F1 58F0C004 051F0001 4004F7F8 40404080 5820C02C

```

(N)

(G)

CHECK OUT

Figure 66. Sample Dump Resulting from Abnormal Termination (Part 6 of 6)

EXECUTION STATISTICS

The DOS/VS COBOL Compiler provides several methods for testing, debugging, and optimizing programs. Use of the symbolic debugging features is an efficient method for testing and debugging a program, and is described in the chapter "Symbolic Debugging Features". The chapter entitled "Program Checkout" contains information useful for testing and debugging programs without the symbolic debugging features. The OPT option, described in the chapter "Preparing COBOL Programs for Processing", is an efficient method for automatically optimizing a program,

This chapter describes execution statistics -- how they may be obtained, some sample output, and some uses of the output.

OBTAINING EXECUTION STATISTICS

Execution statistics are invoked via the CBL card at compile time. No source language coding changes are required. The execution frequency statistics option, COUNT, facilitates testing, debugging, and optimizing by providing the programmer with verb counts at the following times.

- STOP RUN
- GOBACK in the main program
- Abnormal termination of a job

When COUNT is specified, the following items should be taken into account:

1. If COUNT and STIXIT are desired, either STIXIT must be requested in the program unit requesting COUNT, or, the program unit requesting COUNT must be entered before the program unit requesting STIXIT.
2. When COUNT is specified, the compiler divides the program into blocks of verbs. When the statistics are printed, the last block of verbs executed in each program unit is indicated. If the program abnormally terminates, the statement causing the abnormal termination can be determined (by using the symbolic debugging features, for example). The programmer should then subtract one from the verb count for each verb flagged which follows the abending verb.

3. To obtain execution statistics if COUNT is requested for one of many program units, either all programs must be compiled by at least DOS/VS Release 2 compiler, or the program must terminate in a program unit compiled on at least a DOS/VS COBOL Release 2 compiler, or the program must terminate in at least a DOS/VS COBOL library Release 2 subroutine.
4. If COUNT is requested, the user must specify the SIZE parameter on his load module EXEC card. The dynamic space required for COUNT is approximately 512 bytes plus 80 bytes per program unit being monitored, and four bytes per count block (see the compiler output statistics). The requirements for each program unit are rounded to the next 128-byte boundary.
5. The OTHERWISE verb is treated as if the user coded the ELSE verb.

Debugging and Testing

The execution statistics clearly identify the following areas of the program:

- Untested and weakly tested areas of the program
- The last blocks entered and executed
- Possible sources of unnecessary code
- The most heavily used parts of the program; that is, those parts most susceptible to changes.

OPTIMIZATION METHODS

Based on execution frequency and timer statistics, the following types of optimization can be implemented by the user:

- Resequencing the program
- Insight into SYMDMP
- Common expression elimination
- Backward movement
- Unrolling
- Jamming
- Unswitching
- Incorporating procedures inline
- Tabling
- Efficiency guidelines

Note, however, that each optimization technique can result in more inefficient code if the statistics used in optimizing the program are not representative of the normal program flow. In addition, it is recommended that any optimization methods implemented be documented in the program.

Resequencing the Program

The COBOL Procedure Division should be organized as follows:

1. All frequently-used paragraphs or sections should be located near the routines that use them.
2. All infrequently-used paragraphs or sections should be grouped together and apart from frequently-used routines.
3. The most frequently-referenced data items should be placed in the beginning of the Working-Storage Sections.

Insight into SYMDMP Output

The area where dynamic symbolic dumps are to be used can be pointed to by the execution statistics. Knowledge of what area of code is executed and how often it is executed should give the user information on what sections should be further investigated.

Common Expression Elimination

This technique is designed to eliminate unnecessary arithmetic calculations. An arithmetic expression calculation is considered unnecessary if it represents a value calculated elsewhere that will always be used without modification. One such example would be an arithmetic expression whose operands are not redefined or reevaluated, but the expression is recalculated.

Backward Movement

This technique facilitates moving calculations and other operations from an area of code frequently executed to an area less frequently executed. For example, an expression calculated within a PERFORMed procedure (using a Format 2, 3, or 4 PERFORM statement) which always yields the same value for that PERFORM statement could be calculated in-line or in another procedure which would be PERFORMed just prior to the regularly PERFORMed procedure. Another example might be an expression which is calculated in many procedures which are often PERFORMed in succession. This expression could be removed from all the procedures and calculated just once prior to the procedures.

Unrolling

Procedures which are frequently executed may be expanded so that the statements within the procedure are repeated, with slight modification, to reduce the procedure overhead. For example,

```
PERFORM YEARLY-GROSS-CALC VARYING
      WEEK-NO
      FROM 1 BY 1 UNTIL WEEK-NO
      GREATER THAN 52.

YEARLY-GROSS-CALC.
      ADD GROSS-SALARY (WEEK-NO) TO
      YEARLY-GROSS
```

could be replaced by

```
PERFORM YEARLY-GROSS-CALC VARYING
      WEEK-NO
      FROM 1 BY 4 UNTIL WEEK-NO
      GREATER THAN 52.

YEARLY-GROSS-CALC.
      ADD      GROSS-SALARY (WEEK-NO),
      GROSS-SALARY (WEEK-NO+1),
```

EXEC
STAT

GROSS-SALARY (WEEK-NO+2), GROSS
SALARY (WEEK-NO+3)
YEARLY-GROSS.

In addition, indexing might be useful in
this example.

Jamming

In some instances, two procedures can be
merged into one procedure, thereby saving
some procedure overhead. An example of
this might be replacing

```
MOVE 0 TO WEEK-NUM.
PERFORM YEARLY-GROSS-CAL 52 TIMES.
MOVE 0 TO WEEK-NUM.
PERFORM YEARLY-NET-CAL 52 TIMES.
.
.
YEARLY-GROSS-CAL.
  ADD 1 TO WEEK-NUM.
  ADD GROSS-SALARY (WEEK-NUM) TO
  YEARLY-GROSS.
YEARLY-NET-CAL.
  ADD 1 TO WEEK-NUM.
  ADD NET-SALARY (WEEK-NUM) TO
  YEARLY-NET.
```

by

```
MOVE 0 TO WEEK-NUM.
PERFORM YEARLY-CAL 52 TIMES.
.
.
YEARLY-CAL.
  ADD 1 TO WEEK-NUM.
  ADD GROSS-SALARY (WEEK-NUM) TO
  YEARLY-GROSS.
  ADD NET-SALARY (WEEK-NUM) TO
  YEARLY-NET.
```

Unswitching

Procedures may contain tests that result
in the same action for any set of
executions of that procedure. In such a
case, the test can be removed from the
procedure and the procedure duplicated.
For example, if "SWITCH" is not changed
within the loop, replace

```
COUNT=0
PERFORM JOBS-TOTAL-CAL JOB-NUM
TIMES.
.
.
JOB-TOTAL-CAL.
  ADD 1 TO COUNT.
```

```
ADD JOB-COST (COUNT) TO
TOTAL-JOB-COST.
IF SWITCH = 0 ADD JOB-EXPENSE
(COUNT) TO TOTAL-EXPENSES ELSE
ADD JOB-EXPENSE (COUNT) OVERHEAD TO
TOTAL-EXPENSES.
ADD JOB-INCOME (COUNT) TO
TOTAL-INCOME.
IF SWITCH = 0 ADD JOB-PROFIT (COUNT)
TO TOTAL-PROFITS.ELSE
COMPUTE TOTAL-PROFITS =
TOTAL-PROFITS + JOB-INCOME (COUNT)
- JOB-COST (COUNT) - JOB-EXPENSE
(COUNT) - OVERHEAD.
```

by

```
COUNT = 0
IF SWITCH = 0
  PERFORM JOB-TOTAL-CAL-0 JOB-NUM
  TIMES ELSE
  PERFORM JOB-TOTAL-CAL-1 JOB-NUM
  TIMES.
.
.
JOB-TOTAL-CAL-0.
  ADD 1 TO COUNT.
  ADD JOB-COST (COUNT) TO
  TOTAL-JOB-COST.
  ADD JOB-EXPENSE (COUNT) TO
  TOTAL-EXPENSES.
  ADD JOB-INCOME (COUNT) TO
  TOTAL-INCOME.
  ADD JOB-PROFIT (COUNT) TO
  TOTAL-PROFITS.
JOB-TOTAL-CAL-1.
  ADD 1 TO COUNT
  ADD JOB-COST (COUNT) TO
  TOTAL-JOB-COST
  ADD JOB-EXPENSE (COUNT), OVERHEAD TO
  TOTAL-EXPENSE
  ADD JOB-INCOME (COUNT) TO
  TOTAL-INCOME
  COMPUTE TOTAL-PROFITS =
  TOTAL-PROFITS + JOB-INCOME (COUNT)
  - JOB-COST (COUNT) - JOB-EXPENSE
  (COUNT) - OVERHEAD.
```

Incorporating Procedures Inline

Based on module size, number of
repetitions, modification activities,
future expansion considerations, and
frequency statistics, small procedures can
be moved in-line to minimize overhead
requirements.

Tabling

This technique is designed to replace
many IF statements by one table look-up

statement, or by one computed GO TO statement. For example, if the same data-item is tested in many successive IF statements to set the value of another data-item to some constant, and the range of tested values of the original data-item is limited, then a predetermined table of values could be used to assign the value of the second data-item. Similarly, many consecutive statements of the form

```
IF data-item-1=some-constant GO TO
  some-procedure
```

could be replaced by one computed GO TO statement.

Efficiency Guidelines

Based on execution frequency statistics, the following types of coding inefficiencies may be removed.

1. Unaligned decimal places in arithmetic or numeric comparison operands.
2. Different size operands in moves, comparisons, or arithmetic operations.
3. Mixed usage in arithmetic or numeric comparison operands.
4. Display usage in arithmetic operands or one numeric operand and one display operand in a comparison.
5. SYNC missing for COMP or COMP-1, -2, or -4 items.
6. Inefficient COMP type picture; that is, no sign, or more than 9 digits in a COMP item and no sign, even number of digits, or more than 16 digits in COMP-3 items.
7. Certain calls to object-time subroutines.
8. Indexing instead of subscripting and vice versa.
9. Noncomputational subscripts.

DIAGNOSTIC MESSAGES

Diagnostic messages are generated by the compiler and listed on SYSLST when errors are found in the source program.

Note: Diagnostic messages (except FIPS diagnostic messages) are suppressed when the NOERRS option is in effect.

WORKING WITH DIAGNOSTIC MESSAGES

1. Approach the diagnostic messages in the order in which they appear on the source listing. It is possible to get compound diagnostic messages. Frequently, an earlier diagnostic message indicates the reason for a later diagnostic message. For example, a missing quotation mark for an alphabetic or alphanumeric literal could involve the inclusion of some clauses not intended for that particular literal. This could cause an apparently valid clause to be diagnosed as invalid because it is not complete, or because it is in conflict with something that preceded it.
2. Check for missing or superfluous punctuation, or other errors of this type.
3. Frequently, a seemingly meaningless message is clarified when the valid syntax or format of the clause or statement in question is referenced.
4. Statement numbers are generated when a verb or procedure-name is encountered.

GENERATION OF DIAGNOSTIC MESSAGES

The compiler scans the statement, element by element, to determine whether the words are combined in a meaningful manner. Based upon the elements that have already been scanned, there are only certain words or elements that can be correctly encountered.

If the anticipated elements are not encountered, a diagnostic message is produced. Some errors may not be uncovered until information from various sections of the program is combined and the inconsistency is noted. Errors uncovered in this manner can produce a slightly different message format than those uncovered when the actual source text is still available. The message that is made unique through that particular error may not contain, for example, the actual source statement that produced the error.

Errors that appear to be identical are diagnosed in a slightly different manner, depending on where they were encountered by the compiler and how they fit within the context of valid syntax. For example, a period missing from the end of the Working-Storage section header is diagnosed specifically as a period required. There is no other information that can appear at

EXEC
STAT

that point. However, if at the end of a data item description entry, an element is encountered that is not valid at that point, such as the digits 02, it is diagnosed as invalid. Any clauses associated with the 02 entry which conflict with the clauses in the previous entry (the one that contained the missing period), are diagnosed. Thus, a missing period produces a different type of diagnostic message in one situation than in the other.

If an error occurs during compilation of an ON statement, the diagnostic message may refer to the previous statement number.

Notes:

- If an E-level diagnostic is generated, the LINK option is cancelled, and any linkage editor control statements in the job stream are invalid. For this reason, the following message is issued by the Job Control Processor following the first linkage editor control statement encountered:

1S1n D STATEMENT OUT OF SEQUENCE.
I

- If a D-level diagnostic is generated and the error is a compiler error, the job will terminate via the CANCEL macro and produce a dump.
- The following messages will not be issued during a SYNTAX-only compilation or during a CSYNTAX compilation if a C-level error in the diagnostic number ILA0xxx to ILA4xxx range was encountered:

ILA5001I COMPILER ERROR. COMPILATION
ABANDONED.

ILA5002I COMPILER ERROR. COMPILATION
ABANDONED.

ILA5003I DIVISOR IS ZERO. RESULT WILL BE
ALL 9'S.

ILA5004I ALPHANUMERIC SENDING FIELD TOO
BIG. 18 LOW ORDER BYTES USED.

ILA5005I COMPILER ERROR. COMPILATION
ABANDONED.

ILA5006I COMPILER ERROR. COMPILATION
ABANDONED.

ILA5007I COMPILER ERROR. COMPILATION
ABANDONED.

ILA5008I COMPILER ERROR. COMPILATION
ABANDONED.

ILA5009I COMPILER ERROR. COMPILATION
ABANDONED.

ILA5010I HIGH ORDER TRUNCATION OF THE
CONSTANT DID OCCUR.

ILA5011I HIGH ORDER TRUNCATION MIGHT
OCCUR.

ILA5012I LOST INTERMEDIATE RESULT
ATTRIBUTES IN 'XINTR' TABLE.
COMPILATION ABANDONED.

ILA5013I ILLEGAL COMPARISON OF TWO NUMERIC
LITERALS. STATEMENT DISCARDED.

ILA5014I KEY IN SEARCH ALL AT INVALID
OFFSET. STATEMENT DISCARDED.

ILA5015I INVALID USE OF SPECIAL REGISTER.
SUBSTITUTING-TALLY.

ILA5016I MORE THAN 255 SUBSCRIPT ADDRESS
CELLS USED. PROGRAM CANNOT
EXECUTE CORRECTLY.

ILA5017I INVALID ADVANCING OPTION FOR A
DTFCD FILE. USING STACKPR1.

ILA5018I INTEGER IN POSITIONING OPTION NOT
BETWEEN 0 AND 3. 1 ASSUMED.

ILA5019I PUNCH STACKER SELECT SPECIFIED
FOR A DTFPR FILE. USING 'SKIP
TO CHANNEL 1'.

ILA5020I IDENTIFIER NAME(S) IN EXHIBIT
EXCEEDS MAXIMUM. TRUNCATED TO
120 CHARACTERS.

ILA5021I INTEGER IN ADVANCING OR
POSITIONING OPTION NOT
POSITIVE. POSITIVE ASSUMED.

ILA5022I MORE THAN 2-DIGIT INTEGER IN
ADVANCING OPTION. USING
INTEGER 1.

ILA5023I EOP INVALID FOR DOUBLE-BUFFERED
FILE. IGNORED.

ILA5024I END-OF-PAGE OPTION REQUESTED FOR
NON-DTFPR FILE. IGNORED.

ILA5025I ADVANCING OR POSITIONING OPTION
ILLEGAL FOR NON-SEQUENTIAL
FILE. IGNORED.

ILA5026I EXHIBIT OPERAND GREATER THAN 256
BYTES. LENGTH OF 256 ASSUMED.

ILA5027I NEGATIVE OR ZERO SUBSCRIPT
INVALID. CHANGED TO POSITIVE
1.

ILA5028I RESULT FIELD WILL HAVE POSITIVE
SIGN.

ILA5029I STOP RUN GENERATED AFTER LAST
STATEMENT.

ILA5030I INSTEAD OF AN MVCL INSTRUCTION,
AN MVC OR A CALL TO AN
OBJECT-TIME SUBROUTINE HAS BEEN
GENERATED BECAUSE THE FIELDS
OVERLAP DESTRUCTIVELY.

ILA5031I AN MVCL INSTRUCTION HAS BEEN
GENERATED FOR A MOVE INVOLVING
AT LEAST ONE LINKAGE SECTION
DATA-NAME. IF THE FIELDS
OVERLAP DESTRUCTIVELY THE MOVE
WILL NOT BE PERFORMED.

In addition, no message of the form
ILA6xxx will be issued.

CHECK-
OUT

LINKAGE EDITOR OUTPUT

The Linkage Editor produces diagnostic messages, console messages, and a storage map. For a complete description of output and error messages from the Linkage Editor, see the publication DOS/VS System Control Statements. Output resulting from the link editing of a COBOL program is discussed in the chapter "Interpreting Output."

EXECUTION TIME MESSAGES

When an error condition that is recognized by compiler-generated code occurs during execution, an error message is written on SYSLSST and often SYSLOG.

Messages that normally appear on SYSLOG are provided with a code indicating from which partition the message originated.

A complete list of execution-time messages can be found in "Appendix I: Diagnostic Messages."

RECORDING PROGRAM STATUS

When a program is expected to run for an extended period of time, provision should be made for taking checkpoint information periodically during the run. A checkpoint is the recording of the status of a problem program and storage (including input/output status and the contents of the general registers). Thus, it provides a means of restarting the job at an intermediate checkpoint position rather than at the beginning, if for any reason processing is terminated before the normal end of the program. For example, a job of higher priority may require immediate processing, or some malfunction (such as a power failure) may occur and cause an interruption. Checkpoints are taken using the COBOL RERUN clause.

Restart is a means of resuming the execution of the program from one of the checkpoints rather than from the beginning of the job. The ability to restart is provided through the RSTRT job control statement. Full details on using this statement are in DOS/VS System Control Statements.

RERUN CLAUSE

The presence of the RERUN clause in the source program causes the CHKPT macro instruction to be issued at the specified interval. When the CHKPT macro instruction is issued, the following information is saved:

1. Information for the Restart and other supervisor or job control routines.
2. The general registers.
3. Bytes 8 through 10, and 12 through 45 of the Communication Region.
4. The problem program area.
5. All file protection extents for files assigned to mass storage devices if the extents are attached to logical units contained in the program for which checkpoints are taken.

Since the COBOL RERUN clause provides a linkage to the system CHKPT macro instruction, any warnings and restrictions on the use of this macro instruction also apply to the use of the RERUN clause. See the publication DOS/VS Supervisor and I/O Macros for a complete description of the CHKPT macro instruction.

TAKING A CHECKPOINT

In order to take a checkpoint, the programmer must specify the source language RERUN clause and must define the file upon which checkpoint records are to be written (e.g., ASSGN, EXTENT, etc.) Checkpoint information must be written on a 2311, 2314, 2319, 3330, or 3340 mass storage device or on a magnetic tape -- either 7- or 9-track. Checkpoint records cannot be imbedded in one of the problem program's output files, i.e., the program must establish a separate file exclusively for checkpoint records. Checkpoints cannot be written on VSAM files.

In designing a program for which checkpoints are to be taken, the programmer should consider the fact that, upon restarting, the program must be able to continue as though it had just reached that point in the program at which termination occurred. Hence, the programmer should ensure that:

1. File handling is such as to permit easy reconstruction of the status of the system as it existed at the time of checkpoint was taken. For example, when multifile reels are used, the operator should be informed (by message) as to which file is in use at the time a checkpoint is to be taken. He requires this information at restart time.
2. The contents of files are not altered between the time of the checkpoint and the time of the restart. For sequential files, all records written on the file at the time the checkpoint is taken should be unaltered at restart time. For nonsequential files, care must be taken to design the program so that a restart will not duplicate work that has been completed between checkpoint time and restart time. For example, suppose that checkpoint 5 is taken. By adding an amount representing the interest due, account XYZ is updated on a direct-access file that was opened with the I-O option. If the program is restarted from checkpoint 5 and if the interest is recalculated and again added to account XYZ, incorrect results will be produced.

If the program is modular in design, RERUN statements must be included in all modules that handle files for which checkpoints are to be taken. (When an entry point of a module containing a RERUN statement is encountered, a COBOL subroutine, ILBDCKP0, is called. ILBDCKP0 enters the files of the module into the

list of files to be repositioned.)
Repositioning to the proper record will not
occur for any files that were defined in
modules other than those containing RERUN
statements. Moreover, a restart from any
given checkpoint may not reposition other
tapes on which checkpoints are stored.
Note, too, that only one disk checkpoint
file can be used.

RESTARTING A PROGRAM

If the programmer requests checkpoints
in his job by means of the COBOL RERUN
clause, the following message is given each
time a checkpoint is taken:

```
OC001 CHKPT nnnn HAS BEEN TAKEN ON  
      SYSxxx
```

nnnn

is the 4-character identification of
the checkpoint record.

To restart a job from a checkpoint, the
following steps are required:

1. Replace the // EXEC statement with a
// RSTRT statement. The format of the
RSTRT statement is discussed in the

chapter "Preparing COBOL Programs For
Processing." All other job control
statements applicable to the job step
should be the same as when the job was
originally run. If necessary, the
channel and unit addresses for the //
ASSGN control statements may be
changed.

2. Rewind all tapes used by the program
being restarted, and mount them on
devices assigned to the symbolic units
required by the program. If
multivolume files are used, mount (on
the primary unit) the reel being used
at the time that the checkpoint was
taken, and rewind it. If multifile
volumes are used, position the reel to
the start of the file referenced at
the time the checkpoint is being
taken.
3. Reposition any card file so that only
cards not yet read when the checkpoint
was taken are in the card reader.
4. Execute the job.
5. A checkpointed program can be
restarted only in the same partition.
The virtual partition must start at
the same location as when the program
was checkpointed and its end address
must not be lower than at that time.



APPENDIX A: SAMPLE PROGRAM OUTPUT

The following is a sample COBOL program and the output listing resulting from its compilation, link editing, and execution. The program creates a blocked, unlabeled, standard sequential file, writes it out on tape, and then reads it back in. It also does a check on the field called NO-OF-DEPENDENTS. All data records in the file are displayed. Those with a zero in the NO-OF-DEPENDENTS field are displayed with the special character Z. The records

of the file are not altered from the time of creation, despite the fact that the NO-OF-DEPENDENTS field is changed for display purposes. The individual records of the file are created using the subscripting technique.

The output formats illustrated in the listing are described in the chapter "Interpreting Output."

```
// JOB SAMPLE
// OPTION NODECK, LINK, LIST, LISTX, SYM, ERRS
// EXEC FCOBOL
```

1 IBM DOS VS COBOL

REL 1.0

PP NO. 5746-CB1

08.17.32 10/02/73

```
CB1 QUOTE,OPT,SXREF,LVL=A
00001 000010 IDENTIFICATION DIVISION.
00002 000020 PROGRAM-ID. TESTRUN.
00003          AUTHOR. PROGRAMMER NAME.
00004          INSTALLATION. NEW YORK DEVELOPMENT CENTER
00005          DATE-WRITTEN. APRIL 18,1973
00006          DATE-COMPELED. 10/02/73
00007          REMARKS. THIS PROGRAM HAS BEEN WRITTEN AS A SAMPLE PROGRAM FOR
00008          COBOL USERS. IT CREATES AN OUTPUT FILE AND READS IT BACK
00009          AS INPUT.
00010 000100
00011 000110 ENVIRONMENT DIVISION.
00012 000120 CONFIGURATION SECTION.
00013 000130 SOURCE-COMPUTER. IBM-360-H50.
00014 000140 OBJECT-COMPUTER. IBM-370.
00015 000150 INPUT-OUTPUT SECTION.
00016 000160 FILE-CONTROL.
00017 000170     SELECT FILE-1 ASSIGN TO SYS008-UT-2400-S.
00018 000180     SELECT FILE-2 ASSIGN TO SYS008-UT-2400-S.
00019 000190
00020 000200 DATA DIVISION.
00021 000210 FILE SECTION.
00022 000220 FD FILE-1
00023 000230     LABEL RECORDS ARE OMITTED
00024 000240     BLOCK CONTAINS 5 RECORDS
00025 000250     RECORDING MODE IS F
00026 000255     RECORD CONTAINS 20 CHARACTERS
00027 000260     DATA RECORD IS RECORD-1.
00028 000270 01 RECORD-1.
00029          05 FIELD-A PIC X(20).
00030 000290 FD FILE-2
00031 000300     LABEL RECORDS ARE OMITTED
00032 000310     BLOCK CONTAINS 5 RECORDS
00033 000320     RECORD CONTAINS 20 CHARACTERS
00034 000330     RECCORDING MODE IS F
00035 000340     DATA RECORD IS RECORD-2.
00036 000350 01 RECORD-2.
00037          05 FIELD-A PIC X(20).
```

```
00038 000370 WORKING-STORAGE SECTION.
00039 000380 01 FILLER.
00040          02 KOUNT PIC S99 COMP SYNC.
00041 000400 02 ALPHABET PIC X(26) VALUE IS "ABCDEFGHIJKLMNOPQRSTUVWXYZ".
00042 000410 02 ALPHA REDEFINES ALPHABET PIC X OCCURS 26 TIMES.
00043 000420 02 NUMBR PIC S99 COMP SYNC.
00044 000430 02 DEPENDENTS PIC X(26) VALUE "01234012340123401234012340".
00045 000440 02 DEPEND REDEFINES DEPENDENTS PIC X OCCURS 26 TIMES.
00046 000450 01 WORK-RECORD.
00047 000460 05 NAME-FIELD PIC X.
00048 000470 05 FILLER PIC X VALUE IS SPACE.
00049 000480 05 RECORD-NO PIC 9999.
00050 000490 05 FILLER PIC X VALUE IS SPACE.
00051 000500 05 LOCATION PIC AAA VALUE IS "NYC".
00052 000510 05 FILLER PIC X VALUE IS SPACE.
00053 000520 05 NO-OF-DEPENDENTS PIC XX.
00054 000530 05 FILLER PIC X(7) VALUE IS SPACES.
00055 000540
00056 000550 PROCEDURE DIVISION.
00057 BEGIN.
00058 000570 NOTE THAT THE FOLLOWING OPENS THE OUTPUT FILE TO BE CREATED
00059 000580 AND INITIALIZES COUNTERS.
00060 000590 STEP-1. OPEN OUTPUT FILE-1. MOVE ZERO TO KOUNT, NUMBR.
00061 000600 NOTE THAT THE FOLLOWING CREATES INTERNALLY THE RECORDS TO BE
00062 000610 CONTAINED IN THE FILE, WRITES THEM ON TAPE, AND DISPLAYS
00063 000620 THEM ON THE CONSOLE.
00064 000630 STEP-2. ADD 1 TO KOUNT, NUMBR. MOVE ALPHA (KOUNT) TO
00065 000640 NAME-FIELD.
00066 000650 MOVE DEPEND (KOUNT) TO NO-OF-DEPENDENTS
00067 000660 MOVE NUMBR TO RECORD-NO.
00068 000670 STEP-3. DISPLAY WORK-RECORD UPON CONSOLE. WRITE RECORD-1 FROM
00069 000680 WORK-RECORD.
00070 000690 STEP-4. PERFORM STEP-2 THRU STEP-3 UNTIL KOUNT IS EQUAL TO 26.
00071 000700 NOTE THAT THE FOLLOWING CLOSES THE OUTPUT FILE AND REOPENS
00072 000710 IT AS INPUT.
00073 000720 STEP-5. CLOSE FILE-1. OPEN INPUT FILE-2.
00074 000730 NOTE THAT THE FOLLOWING READS BACK THE FILE AND SINGLES
00075 000740 OUT EMPLOYEES WITH NO DEPENDENTS.
00076 000750 STEP-6. READ FILE-2 RECORD INTO WORK-RECORD AT END GO TO STEP-8.
00077 000760 STEP-7. IF NO-OF-DEPENDENTS IS EQUAL TO "0" MOVE "Z" TO
00078 000770 NO-OF-DEPENDENTS. EXHIBIT NAMED WORK-RECORD. GO TO STEP-6.
00079 000780 STEP-8. CLOSE FILE-2.
00080 000790 STOP RUN.
```

INTRNL NAME	LVL	SOURCE NAME	BASE	DISPL	INTRNL NAME	DEFINITION	USAGE	R	O	Q	M
DNM=1-148	FD	FILE-1	DTF=01		DNM=1-148		DTFMT				F
DNM=1-179	01	RECORD-1	BL=1	000	DNM=1-179	DS 0CL20	GROUP				
DNM=1-200	02	FIELD-A	BL=1	000	DNM=1-200	DS 20C	DISP				
DNM=1-217	FD	FILE-2	DTF=02		DNM=1-217		DTFMT				F
DNM=1-248	01	RECORD-2	BL=2	000	DNM=1-248	DS 0CL20	GROUP				
DNM=1-269	02	FIELD-A	BL=2	000	DNM=1-269	DS 20C	DISP				
DNM=1-289	01	FILLER	BL=3	000	DNM=1-289	DS 0CL56	GROUP				
DNM=1-308	02	KOUNT	BL=3	000	DNM=1-308	DS 1H	COMP				
DNM=1-323	02	ALPHABET	BL=3	002	DNM=1-323	DS 26C	DISP				
DNM=1-341	02	ALPHA	BL=3	002	DNM=1-341	DS 1C	DISP	R	O		
DNM=1-359	02	NUMBR	BL=3	01C	DNM=1-359	DS 1H	COMP				
DNM=1-374	02	DEPENDENTS	BL=3	01E	DNM=1-374	DS 26C	DISP				
DNM=1-394	02	DEPEND	BL=3	01E	DNM=1-394	DS 1C	DISP	R	O		
DNM=1-410	01	WORK-RECORD	BL=3	038	DNM=1-410	DS 0CL20	GROUP				
DNM=1-434	02	NAME-FIELD	BL=3	038	DNM=1-434	DS 1C	DISP				
DNM=1-454	02	FILLER	BL=3	039	DNM=1-454	DS 1C	DISP				
DNM=1-473	02	RECORD-NO	BL=3	03A	DNM=1-473	DS 4C	DISP-NM				
DNM=1-492	02	FILLER	BL=3	03E	DNM=1-492	DS 1C	DISP				
DNM=2-000	02	LOCATION	BL=3	03F	DNM=2-000	DS 3C	DISP				
DNM=2-018	02	FILLER	BL=3	042	DNM=2-018	DS 1C	DISP				
DNM=2-037	02	NO-OF-DEPENDENTS	BL=3	043	DNM=2-037	DS 2C	DISP				
DNM=2-063	02	FILLER	BL=3	045	DNM=2-063	DS 7C	DISP				

MEMORY MAP

TGT	003F8
SAVE AREA	003F8
SWITCH	00440
TALLY	00444
SORT SAVE	00448
ENTRY-SAVE	0044C
SORT CORE SIZE	00450
NSTD-REELS	00454
SORT RET	00456
WORKING CELLS	00458
SORT FILE SIZE	00588
SORT MODE SIZE	0058C
PGT-VN TBL	00590
TGT-VN TBL	00594
SORTAB ADDRESS	00598
LENGTH OF VN TBL	0059C
LNPTH OF SORTAB	0059E
PGM ID	005A0
A(INIT1)	005A8
UPSI SWITCHES	005AC
DEBUG TABLE PTR	005B4
CURRENT PRIORITY	005E8
TA LENGTH	005E9
PRBL1 CELL PTR	005BC
UNUSED	005C0
RESERVED	005C4
VSAM SAVE AREA ADDRESS	005C8
UNUSED	005CC
RESERVED	005D4
OVERFLOW CELLS	005EC
BL CELLS	005EC
DTFADR CELLS	005F8
FIB CELLS	00600
TEMP STORAGE	00608
TEMP STORAGE-2	00610
TEMP STORAGE-3	00610
TEMP STORAGE-4	00610
BLL CELLS	00610
VLC CELLS	00614
SBL CELLS	00614
INDEX CELLS	00614
SUBADR CELLS	00614
ONCTL CELLS	0061C
PFMCTL CELLS	0061C
PFMSAV CELLS	0061C
VN CELLS	00620
SAVE AREA =2	00624
XSASW CELLS	00624
XSA CELLS	00624
PARAM CELLS	00624
RPTSAV AREA	00628
CHECKPT CTR	00628
IOPTR CELLS	00628
DEBUG TABLE	00628

LITERAL POOL (HEX)

00640 (LIT+0) 00000001 001A5B5B C2D6D7C5 D5405E5B C2C3D3D6 L2C55E5B
00658 (LIT+24) C2C6C3D4 E4D35B5B C0000000

DISPLAY LITERALS (BCD)

00664 (LTL+36) 'WORK-RECORD'

PGT	00628
DEBUG LINKAGE AREA	00628
OVERFLOW CELLS	00628
VIRTUAL CELLS	0062C
PROCEDURE NAME CELLS	00638
GENERATED NAME CELLS	00638
SUBDTF ADDRESS CELLS	0063C
VNI CELLS	0063C
LITERALS	00640
DISPLAY LITERALS	00664
PROCEDURE BLOCK CELLS	00670

REGISTER ASSIGNMENT

REG 6 BL =3
REG 7 BL =1
REG 8 BL =2

WORKING-STORAGE STARTS AT LOCATION 00100 FOR A LENGTH OF 00050.

PROCEDURE BLOCK ASSIGNMENT

PBL = REG 11

PBL =1 STARTS AT LOCATION 000674 STATEMENT 60

```

0
57
60      000674      PN=02      EQU      *
60      000674      PN=03      EQU      *
60      000674      START      EQU      *
000674      58 B0 C 048      L      11,048(0,12)      PBL=1
000678      58 20 D 1F4      L      2,1F4(0,13)      BL =1
00067C      41 10 C 01E      LA      1,01E(0,12)      LIT+6
000680      58 00 D 200      L      0,200(0,13)      DTF=1
000684      18 40      LR      4,0
000686      05 F0      BALR     15,0
000688      50 00 F 008      ST      0,008(0,15)
00068C      45 00 F 00C      BAL      0,00C(0,15)
000690      00000000      DC      X'00000000'
000694      0A 02      SVC      2
000696      41 00 D 200      LA      0,200(0,13)      DTF=1
00069A      58 F0 C 008      L      15,008(0,12)      V(ILEDIML0)
00069E      05 EF      BALR     14,15
0006A0      58 10 D 200      L      1,200(0,13)      DTF=1
0006A4      96 10 1 020      OI      020(1),X'10'
0006A8      50 20 D 1F4      ST      2,1F4(0,13)      BL =1
0006AC      58 70 D 1F4      L      7,1F4(0,13)      BL =1
60
0006B0      D2 01 6 000 C 018      MVC     000(2,6),018(12)      DNM=1-308      LIT+0
0006B6      D2 01 6 01C C 018      MVC     01C(2,6),018(12)      DNM=1-359      LIT+0
64
64      0006BC      PN=04      EQU      *
0006BC      48 30 C 01A      LH      3,01A(0,12)      LIT+2
0006C0      4A 30 6 000      AH      3,000(0,6)      D,M=1-308
0006C4      4E 30 D 210      CVD     3,210(0,13)      TS=01
0006C8      D7 05 D 210 D 210      XC      210(6,13),210(13)      TS=01      TS=01
0006CE      94 0F D 216      NI      216(13),X'0F'      TS=01+6
0006D2      4F 30 D 210      CVB     3,210(0,13)      TS=01
0006D6      40 30 6 000      STH     3,000(0,6)      DNM=1-308
0006DA      48 30 C 01A      LH      3,01A(0,12)      LIT+2
0006DE      4A 30 6 01C      AH      3,01C(0,6)      DNM=1-359
0006E2      4E 30 D 210      CVD     3,210(0,13)      TS=01
0006E6      D7 05 D 210 D 210      XC      210(6,13),210(13)      TS=01      TS=01
0006EC      94 0F D 216      NI      216(13),X'0F'      TS=01+6
0006F0      4F 30 D 210      CVB     3,210(0,13)      TS=01
0006F4      40 30 6 01C      STH     3,01C(0,6)      DNM=1-359
64
0006F8      41 40 6 002      LA      4,002(0,6)      DNM=1-341
0006FC      48 20 6 000      LH      2,000(0,6)      DNM=1-308
000700      4C 20 C 01A      MH      2,01A(0,12)      LIT+2
000704      1A 42      AR      4,2
000706      5E 40 C 018      S      4,018(0,12)      LIT+0
00070A      50 40 D 21C      ST      4,21C(0,13)      SBS=1
00070E      58 E0 D 21C      L      14,21C(0,13)      SBS=1
000712      D2 00 6 038 E 000      MVC     038(1,6),000(14)      DNM=1-434      DNM=1-341
000718      41 40 6 01E      LA      4,01E(0,6)      DNM=1-394
00071C      48 20 6 000      LH      2,000(0,6)      DNM=1-308
000720      4C 20 C 01A      MH      2,01A(0,12)      LIT+2
000724      1A 42      AR      4,2
000726      5B 40 C 018      S      4,018(0,12)      LIT+0
00072A      50 40 D 220      ST      4,220(0,13)      SBS=2
00072E      58 F0 D 220      L      15,220(0,13)      SBS=2
000732      D2 00 6 043 F 000      MVC     043(1,6),000(15)      DNM=1-394      DNM=1-394
000738      92 40 6 044      MVI     044(6),X'40'      DNM=2-37+1
67
00073C      48 30 6 01C      LH      3,01C(0,6)      DNM=1-359
000740      4E 30 D 210      CVD     3,210(0,13)      TS=01
000744      F3 31 6 03A D 216      UNPK    03A(4,6),216(2,13)      DNM=1-473      TS=07
00074A      96 F0 6 03D      OI      03D(6),X'F0'      DNM=1-473+3
    
```

```

68          00074E          PN=05  EQU  *
68          00074E 58 F0 C 00C          L    15,00C(0,12)          V(ILEEDSP0)
          000752 05 1F          BALR  1,15
          000754 0002          DC    X'0002'
          000756 00          DC    X'00'
          000757 000014          DC    X'000014'
          00075A 0D0001FC          DC    X'0D0001FC'          BL =3
          00075E 0038          DC    X'0038'
          000760 FFFF          DC    X'FFFF'

68          000762 D2 13 7 000 6 038          MVC  000(20,7),038(6)          DNM=1-179          DNM=1-410
          000768 58 10 D 200          L    1,200(0,13)          DTF=1
          00076C 18 41          LR    4,1
          00076E 58 F0 1 010          L    15,010(0,1)
          000772 45 E0 F 00C          BAL  14,00C(0,15)
          000776 50 20 D 1F4          ST   2,1F4(0,13)          BL =1
          00077A 58 70 D 1F4          L    7,1F4(0,13)          BL =1
          00077E 58 10 D 228          L    1,228(0,13)          VN=01
          000782 07 F1          BCR  15,1

70          000784          PN=06  EQU  *
70          000784 D2 03 D 224 D 228          MVC  224(4,13),228(13)          PSV=1          VN=01
          00078A 41 00 B 11E          LA   0,11E(0,11)          GN=01
          00078E 50 00 D 228          ST   0,228(0,13)          VN=01
          000792          GN=01  EQU  *
          000792 48 30 6 000          LH   3,000(0,6)          DNM=1-308
          000796 49 30 C 01C          CH   3,01C(0,12)          LIT+4
          00079A 47 80 B 12E          BC   8,12E(0,11)          GN=02
          00079E 47 F0 B 048          BC   15,048(0,11)          PN=04
          0007A2          GN=02  EQU  *
          0007A2 D2 03 D 228 D 224          MVC  228(4,13),224(13)          VN=01          PSV=1

73          0007A8          PN=07  EQU  *
73          0007A8 58 10 D 200          L    1,200(0,13)          DTF=1
          0007AC 94 EF 1 020          NI   020(1),X'EF'
          0007B0 18 01          LR    0,1
          0007B2 18 40          LR    4,0
          0007B4 41 10 C 026          LA   1,026(0,12)          LIT+14
          0007B8 07 00          BCR  0,0
          0007BA 05 F0          BALR  15,0
          0007BC 50 00 F 008          ST   0,008(0,15)
          0007C0 45 00 F 00C          BAL  0,00C(0,15)
          0007C4 00000000          DC    X'00000000'
          0007C8 0A 02          SVC  2
          0007CA 58 00 D 200          L    0,200(0,13)          DTF=1
          0007CE 41 10 C 02E          LA   1,02E(0,12)          LIT+22
          0007D2 0A 02          SVC  2

73          0007D4 41 10 C 01E          LA   1,01E(0,12)          LIT+6
          0007D8 58 00 D 204          L    0,204(0,13)          DTF=2
          0007DC 18 40          LR    4,0
          0007DE 05 F0          BALR  15,0
          0007E0 50 00 F 008          ST   0,008(0,15)
          0007E4 45 00 F 00C          BAL  0,00C(0,15)
          0007E8 00000000          DC    X'00000000'
          0007EC 0A 02          SVC  2
          0007EE 41 00 D 204          LA   0,204(0,13)
          0007F2 58 F0 C 008          L    15,008(0,12)          V(ILBDIML0)
          0007F6 05 EF          BALR  14,15
          0007F8 58 10 D 204          L    1,204(0,13)          DTF=2
          0007FC 96 10 1 020          OI   020(1),X'10'
    
```


76							
76	000800		FN=08	EQU	*		
	000800	58 10 D 204		L	1,204(0,13)		DTF=2
	000804	91 20 1 010		TM	C10(1),X'20'		
	000808	47 10 B 1BE		BC	1,1BE(0,11)		GN=03
	00080C	18 41		LR	4,1		
	00080E	41 F0 C 010		LA	15,010(0,12)		GN=03
	000812	D2 02 1 025 F 001		MVC	025(3,1),001(15)		
	000818	58 F0 1 010		L	15,010(0,1)		
	00081C	45 E0 F 008		EAL	14,008(0,15)		
	000820	50 20 D 1F8		ST	2,1F8(0,13)		BL =2
	000824	58 80 D 1F8		L	8,1F8(0,13)		EL =2
	000828	D2 13 6 038 8 000		MVC	038(20,6),000(8)		DNM=1-410
	00082E	47 F0 B 1C2		BC	15,1C2(0,11)		GN=04
	000832		GN=03	EQU	*		
76							
	000832	47 F0 B 208		BC	15,208(0,11)		PN=010
	000836		GN=04	EQU	*		
77							
	000836		PN=09	EQU	*		
77							
	000836	95 F0 6 043		CLI	043(6),X'F0'		DNM=2-37
	00083A	47 70 B 1DA		BC	7,1DA(0,11)		GN=05
	00083E	95 40 6 044		CLI	044(6),X'40'		DNM=2-37+1
77							
	000842	47 70 B 1DA		BC	7,1DA(0,11)		GN=05
	000846	92 E9 6 043		MVI	043(6),X'E9'		DNM=2-37
	00084A	92 40 6 044		MVI	044(6),X'40'		DNM=2-37+1
	00084E		GN=05	EQU	*		
78							
	00084E	58 10 C 038		L	1,038(0,12)		LIT+32
	000852	50 10 D 22C		ST	1,22C(0,13)		PRM=1
	000856	41 20 D 22C		LA	2,22C(0,13)		PRM=1
	00085A	58 F0 C 00C		L	15,00C(0,12)		V(ILBDDSP0)
	00085E	05 1F		BALR	1,15		
	000860	8001		DC	X'8001'		
	000862	10		DC	X'10'		
	000863	00000B		DC	X'00000B'		
	000866	0C00003C		DC	X'0C00003C'		LIT+36
	00086A	0000		DC	X'0000'		
	00086C	00		DC	X'00'		
	00086D	000014		DC	X'000014'		
	000870	0D0001FC		DC	X'0D0001FC'		BL =3
	000874	0038		DC	X'0038'		
	000876	FFFF		DC	X'FFFF'		
78							
	000878	47 F0 B 18C		BC	15,18C(0,11)		PN=08
79							
	00087C		PN=010	EQU	*		
79							
	00087C	58 10 D 204		L	1,204(0,13)		DTF=2
	000880	94 EF 1 020		NI	020(1),X'EF'		
	000884	18 01		LR	0,1		
	000886	18 40		LR	4,0		
	000888	41 10 C 026		LA	1,026(0,12)		LIT+14
	00088C	07 00		BCR	0,0		
	00088E	05 F0		BALR	15,0		
	000890	50 00 F 008		ST	0,008(0,15)		
	000894	45 00 F 00C		BAL	0,00C(0,15)		
	000898	00000000		DC	X'00000000'		
	00089C	0A 02		SVC	2		
	00089E	58 00 D 204		L	0,204(0,13)		DTF=2
	0008A2	41 10 C 02E		LA	1,02E(0,12)		LIT+22
	0008A6	0A 02		SVC	2		
80							
	0008A8	0A 0E		SVC	14		

0008AA	50 D0 5 008	INIT2	ST	13,008(0,5)	
0008AE	50 50 D 004		ST	5,004(0,13)	
0008B2	58 20 C 004		L	2,004(0,12)	VIR=1
0008B6	95 00 2 000		CLI	000(2),X'00'	
0008BA	07 79		BCR	7,9	
0008BC	92 FF 2 000		MVI	000(2),X'FF'	
0008C0	96 10 D 048		OI	048(13),X'10'	SWT+0
0008C4	50 E0 D 054	INIT3	ST	14,054(0,13)	
0008C8	05 F0		BALR	15,0	
0008CA	91 20 D 048		TM	048(13),X'20'	SWT+0
0008CE	47 E0 F 016		BC	14,016(0,15)	
0008D2	58 00 B 048		L	0,048(0,11)	
0008D6	98 2D B 050		LM	2,13,050(11)	
0008DA	58 E0 D 054		L	14,054(0,13)	
0008DE	07 FE		BCR	15,14	
0008E0	96 20 D 048		OI	048(13),X'20'	SWT+0
0008E4	41 60 0 004		LA	6,004(0,0)	
0008E8	41 10 C 000		LA	1,000(0,12)	
0008EC	41 70 C 003		LA	7,003(0,12)	VIR=1-1
0008F0	05 50		BALR	5,0	
0008F2	58 40 1 000		L	4,000(0,1)	
0008F6	1E 4E		ALR	4,11	
0008F8	50 40 1 000		ST	4,000(0,1)	
0008FC	87 16 5 000		BXLE	1,6,000(5)	
000900	41 10 C 010		LA	1,010(0,12)	PN=01
000904	41 70 C 017		LA	7,017(0,12)	IIT+0-1
000908	05 50		BALR	5,0	
00090A	58 40 1 000		L	4,000(0,1)	
00090E	1E 4B		ALR	4,11	
000910	50 40 1 000		ST	4,000(0,1)	
000914	87 16 5 000		BXLE	1,6,000(5)	
000918	41 80 D 1F4		LA	8,1F4(0,13)	OVF=1
00091C	41 70 D 20F		LA	7,20F(0,13)	TS=01-1
000920	05 10		BALR	1,0	
000922	58 00 8 000		L	0,000(0,8)	
000926	1E 0B		ALR	0,11	
000928	50 00 8 000		ST	0,000(0,8)	
00092C	87 86 1 000		BXLE	8,6,000(1)	
000930	58 60 D 1FC		L	6,1FC(0,13)	BL =3
000934	58 70 D 1F4		L	7,1F4(0,13)	BL =1
000938	58 80 D 1F8		L	8,1F8(0,13)	BL =2
00093C	D2 03 D 228 C 014		MVC	228(4,13),014(12)	VN=01
000942	58 E0 D 1E0		L	14,1E0(0,13)	
000946	90 6D E 060		STM	6,13,060(14)	
00094A	58 E0 D 054		L	14,054(0,13)	
00094E	07 FE		BCR	15,14	
000000	05 F0	INIT1	BALR	15,0	
000002	07 00		BCR	0,0	
000004	90 0E F 00A		STM	0,14,00A(15)	
000008	47 F0 F 082		BC	15,082(0,15)	
00000C	00000000		DC	30F'0'	
000084	58 CQ F 0C6		L	12,0C6(0,15)	
000088	58 E0 C 004		L	14,004(0,12)	VIR=1
00008C	58 D0 F 0CA		L	13,0CA(0,15)	
000090	95 00 E 000		CLI	000(14),X'00'	
000094	47 70 F 0A2		BC	7,0A2(0,15)	
000098	96 10 D 048		OI	048(13),X'10'	SWT+0
00009C	92 FF E 000		MVI	000(14),X'FF'	
0000A0	47 F0 F 0AC		BC	15,0AC(0,15)	

0000A4	98 CE F 03A	LM	12,14,03A(15)	
0000A8	90 EC D 00C	STM	14,12,00C(13)	
0000AC	18 5D	LR	5,13	
0000AE	98 9F F 0BA	LM	9,15,0BA(15)	
0000B2	91 10 D 048	TM	048(13),X'10'	SWT+0
0000B6	07 19	BCR	1,9	
0000B8	07 FF	BCR	15,15	
0000BA	07 00	BCR	0,0	
0000BC	000008C4	ADCON	L4(INIT3)	
0000C0	00000000	ADCON	L4(INIT1)	
0000C4	00000000	ADCON	L4(INIT1)	
0000C8	00000628	ADCON	L4(PGT)	
0000CC	000003F8	ADCON	L4(TGT)	
0000D0	00000674	ADCON	L4(START)	
0000D4	000008AA	ADCON	L4(INIT2)	
0000D8	C3D6C2D6F3F0F0F0	DC	X'C3D6C2D6F3F0F0F0'	
0000E0	E3C5E2E3D9E4D540	DC	X'E3C5E2E3D9E4D540'	
0000E8	00000000	DC	X'00000000'	
0000EC	F1F061F0F261F7F3	DC	X'F1F061F0F261F7F3'	
0000F4	FOF84BF1F74BF3F2	DC	X'FOF84BF1F74BF3F2'	

STATISTICS	SOURCE RECORDS =	80	DATA ITEMS =	22	NO OF VERBS =	28		
STATISTICS	PARTITION SIZE =	655176	LINE COUNT =	56	BUFFER SIZE =	512		
OPTIONS IN EFFECT	PMAP RELOC ADR =	NCNE	SPACING =	1	FLOW =	NONE		
OPTIONS IN EFFECT	LISTX	QUOTE	SYM	NOCATALR	LIST	LINK	NOSIXIT	NOLLB
OPTIONS IN EFFECT	NOCLIST	FLAGW	ZWB	NOSUPMAP	XREF	ERRS	SXRLF	OPT
OPTIONS IN EFFECT	NOSTATE	TRUNC	SEQ	NOSYMDMP	NODECK	NOVERB	NOSYNTEX	LVL=A

CROSS-REFERENCE DICTIONARY

DATA NAMES	DEFN	REFERENCE
ALPHA	000042	000064
ALPHABET	000041	
DEPEND	000045	000066
DEPENDENTS	000044	
FIELD-A	000029	
FIELD-A	000037	
FILE-1	000017	000060 000068 000073
FILE-2	000018	000073 000076 000079
KCUNT	000040	000060 000064 000066 000070
LOCATION	000051	
NAME-FIELD	000047	000064
NC-OF-DEPENDENTS	000053	000066 000077
NUMBR	000043	000060 000064 000067
RECORD-NO	000049	000067
RECORD-1	000028	000068
RECORD-2	000036	000076
WORK-RECORD	000046	000068 000076 000078

PROCEDURE NAMES	DEFN	REFERENCE
BEGIN	000057	
STEP-1	000060	
STEP-2	000064	000070
STEP-3	000068	000070
STEP-4	000070	
STEP-5	000073	
STEP-6	000076	000078
STEP-7	000077	
STEP-8	000079	000076

CARD ERROR MESSAGE

00064	ILA5011I-W	HIGH ORDER TRUNCATION MIGHT OCCUR.
00064	ILA5011I-W	HIGH ORDER TRUNCATION MIGHT OCCUR.

FEDERAL INFORMATION PROCESSING STANDARDS (FIPS) DIAGNOSTIC MESSAGES

LINE	NUMBER	MESSAGE
C0006	ILA8003I-W	DATE-COMPILED PARAGRAPH IS AN EXTENSION TO FIPS LEVEL A.
C0025	ILA8002I-W	RECORDING MODE IS CLAUSE IS AN EXTENSION TO ALL FIPS LEVELS.
C0034	ILA8002I-W	RECORDING MODE IS CLAUSE IS AN EXTENSION TO ALL FIPS LEVELS.
C0054	ILA8003I-W	SPACES IS AN EXTENSION TO FIPS LEVEL A.
C0060	ILA8003I-W	COMMA OR SEMICOLON AS PUNCTUATION IS AN EXTENSION TO FIPS LEVEL A.
C0062	ILA8003I-W	COMMA OR SEMICOLON AS PUNCTUATION IS AN EXTENSION TO FIPS LEVEL A.
C0062	ILA8003I-W	COMMA OR SEMICOLON AS PUNCTUATION IS AN EXTENSION TO FIPS LEVEL A.
C0064	ILA8003I-W	COMMA OR SEMICOLON AS PUNCTUATION IS AN EXTENSION TO FIPS LEVEL A.
C0064	ILA8003I-W	MULTIPLE RESULTS IN ADD STATEMENT IS AN EXTENSION TO FIPS LEVEL A.
C0068	ILA8003I-W	UPON OPTION OF DISPLAY STATEMENT IS AN EXTENSION TO FIPS LEVEL A.
C0068	ILA8002I-W	UPON CONSOLE OPTION OF DISPLAY STATEMENT IS AN EXTENSION TO ALL LEVELS.
C0068	ILA8003I-W	FROM OPTION OF WRITE STATEMENT IS AN EXTENSION TO FIPS LEVEL A.
C0070	ILA8003I-W	UNTIL OPTION OF PERFORM STATEMENT IS AN EXTENSION TO FIPS LEVEL A.
C0076	ILA8003I-W	INTO OPTION OF READ STATEMENT IS AN EXTENSION TO FIPS LEVEL A.
C0078	ILA8002I-W	EXHIBIT STATEMENT IS AN EXTENSION TO ALL FIPS LEVELS.
END OF COMPILATION		

// EXEC LNKEDT

JOE SAMPLE DOS LINKAGE EDITOR DIAGNOSTIC OF INPUT

```

ACTION TAKEN MAP REL
LIST AUTOLINK IJFFBZZN
LIST AUTOLINK ILBDDSP0
LIST AUTOLINK IJJCPDV
LIST AUTOLINK ILBDDSS0
LIST INCLUDE IJJCPDV
LIST AUTOLINK ILBDIML0
LIST AUTOLINK ILBDMNS0
LIST AUTOLINK ILBDSAE0
LIST ENTRY
    
```

10/02/73	PHASE	XFR-AD	LOCORE	HICORE	DSK-AD	ESD TYPE	LABBL	LOADED	REL-FR	
	PHASE***	07D878	07D878	07F1FF	05F OF 4	CSECT	TESTRUN	07D878	07D878	RILLOCATABLE
						CSECT	IJFFBZZN	07E1C8	07E1C8	
						* ENTRY	IJFFZZZN	07E1C8		
						* ENTRY	IJFFBZZZ	07E1C8		
						* ENTRY	IJFFZZZZ	07E1C8		
						CSECT	ILBDSAE0	07F078	07F078	
						ENTRY	ILBDSAE1	07F0C0		
						CSECT	ILBDMNS0	07F070	07F070	
						CSECT	ILBDIML0	07F018	07F018	
						CSECT	ILBDDSP0	07E578	07E578	
						ENTRY	ILBDDSP1	07E978		
						CSECT	ILBDDSS0	07ECF0	07ECF0	
						ENTRY	ILBDDSS1	07EF50		
						ENTRY	ILBDDSS2	07EF48		
						ENTRY	ILBDDSS3	07F008		
						ENTRY	ILBDDSS4	07ED16		
						ENTRY	ILBDDSS5	07EDC2		
						ENTRY	ILBDDSS6	07EE22		
						ENTRY	ILBDDSS7	07EDEC		
						ENTRY	ILBDDSS8	07ED46		
						CSECT	IJJCPDV	07EAA8	07EAA8	
						ENTRY	IJJCPDV1	07EAA8		
						* ENTRY	IJJCPDV2	07EAA8		
						WXTRN	STXITPSW			
						WXTRN	ILBDDBG2			

* UNREFERENCED SYMBOLS

002 UNRESOLVED ADDRESS CONSTANTS

// ASSGN SYS008,X'483'
// EXEC

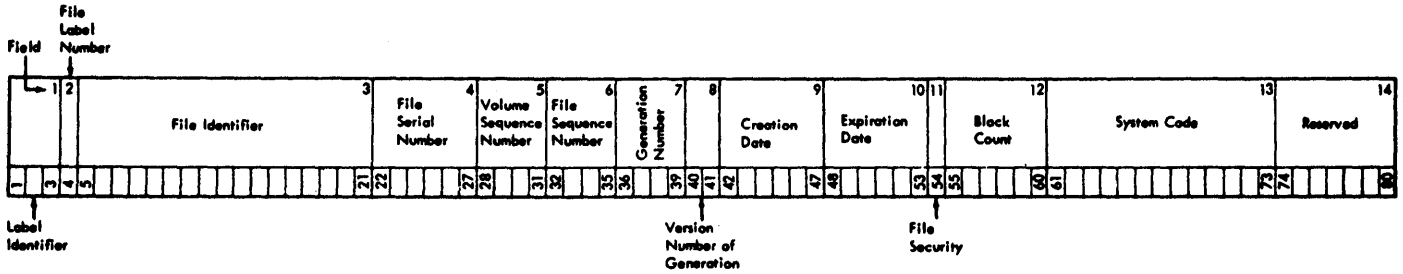
WORK-RECORD = A 0001 NYC Z
WORK-RECORD = B 0002 NYC 1
WORK-RECORD = C 0003 NYC 2
WORK-RECORD = D 0004 NYC 3
WORK-RECORD = E 0005 NYC 4
WORK-RECORD = F 0006 NYC Z
WORK-RECORD = G 0007 NYC 1
WORK-RECORD = H 0008 NYC 2
WORK-RECORD = I 0009 NYC 3
WORK-RECORD = J 0010 NYC 4
WORK-RECORD = K 0011 NYC Z
WORK-RECORD = L 0012 NYC 1
WORK-RECORD = M 0013 NYC 2
WORK-RECORD = N 0014 NYC 3
WORK-RECORD = O 0015 NYC 4
WORK-RECORD = P 0016 NYC Z
WORK-RECORD = Q 0017 NYC 1
WORK-RECORD = R 0018 NYC 2
WORK-RECORD = S 0019 NYC 3
WORK-RECORD = T 0020 NYC 4
WORK-RECORD = U 0021 NYC Z
WORK-RECORD = V 0022 NYC 1
WORK-RECORD = W 0023 NYC 2
WORK-RECORD = X 0024 NYC 3
WORK-RECORD = Y 0025 NYC 4
WORK-RECORD = Z 0026 NYC Z

ECJ SAMPLE

BG
BG A 0001 NYC 0
BG B 0002 NYC 1
BG C 0003 NYC 2
BG D 0004 NYC 3
BG E 0005 NYC 4
BG F 0006 NYC 0
BG G 0007 NYC 1
BG H 0008 NYC 2
BG I 0009 NYC 3
BG J 0010 NYC 4
BG K 0011 NYC 0
BG L 0012 NYC 1
BG M 0013 NYC 2
BG N 0014 NYC 3
BG O 0015 NYC 4
BG P 0016 NYC 0
BG Q 0017 NYC 1
BG R 0018 NYC 2
BG S 0019 NYC 3
BG T 0020 NYC 4
BG U 0021 NYC 0
BG V 0022 NYC 1
BG W 0023 NYC 2
BG X 0024 NYC 3
BG Y 0025 NYC 4
BG Z 0026 NYC 0
BG EOJ SAMPLE
00.56.19,DURATION 00.03.42



APPENDIX B: STANDARD TAPE FILE LABELS

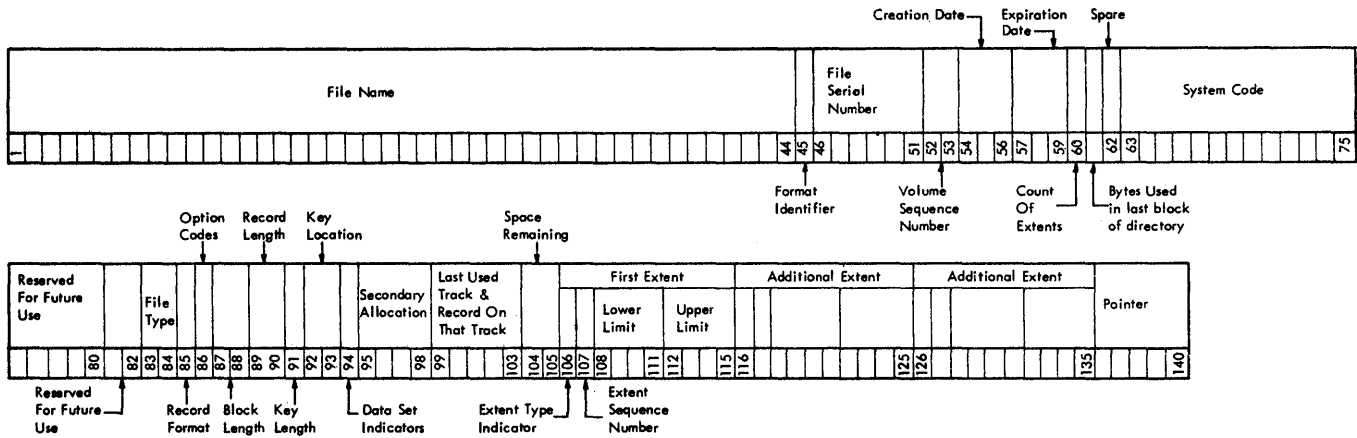


The standard tape file label format and contents are as follows:

<u>Field</u>	<u>Name and Length</u>	<u>Description</u>
1.	<u>LABEL IDENTIFIER</u> 3 bytes, EBCDIC	Identifies the type of label. HDR = Header (beginning of a data file) EOF = End-of-file (end of a set of data) EOV = End-of-volume (end of the physical reel)
2.	<u>FILE LABEL NUMBER</u> 1 byte, EBCDIC	Always a 1.
3.	<u>FILE IDENTIFIER</u> 17 bytes, EBCDIC	Uniquely identifies the entire file, may contain only printable characters. Some other systems will not accept embedded blanks in the file identifier.
4.	<u>FILE SERIAL NUMBER</u> 6 bytes, EBCDIC	Uniquely identifies a file/volume relationship. This field is identical to the volume serial number in the volume label of the first or only volume of a multivolume file or a multifile set. This field will normally be numeric (000001 to 999999), but may contain any six alphanumeric characters.
5.	<u>VOLUME SEQUENCE NUMBER</u> 4 bytes	Indicates the order of a volume in a given file or multifile set. The first must be numbered 0001, and subsequent numbers must be in proper numeric sequence.
6.	<u>FILE SEQUENCE</u> 4 bytes	Assigns numeric sequence to a file within a multifile set. The first must be numbered 0001.
7.	<u>GENERATION TIME</u> 4 bytes	Uniquely identifies the various editions of the file. May be from 0001 to 9999 in proper numeric sequence.
8.	<u>VERSION NUMBER OF GENERATION</u> 2 bytes	Indicates the version of a generation of a file.

<u>Field</u>	<u>Name and Length</u>	<u>Description</u>												
9.	<u>CREATION DATE</u> 6 bytes	Indicates the year and the day of the year that the file was created. <table border="1"> <thead> <tr> <th><u>Position</u></th> <th><u>Code</u></th> <th><u>Meaning</u></th> </tr> </thead> <tbody> <tr> <td>1</td> <td>blank</td> <td>none</td> </tr> <tr> <td>2-3</td> <td>00-99</td> <td>year</td> </tr> <tr> <td>4-6</td> <td>001-366</td> <td>day of year</td> </tr> </tbody> </table> <p>(e.g., January 31, 1973 would be entered as 73031).</p>	<u>Position</u>	<u>Code</u>	<u>Meaning</u>	1	blank	none	2-3	00-99	year	4-6	001-366	day of year
<u>Position</u>	<u>Code</u>	<u>Meaning</u>												
1	blank	none												
2-3	00-99	year												
4-6	001-366	day of year												
10.	<u>EXPIRATION DATE</u> 6 bytes	Indicates the year and the day of the year when the file may become a scratch tape. The format of this field is identical to field 9. On a multifile reel processed sequentially, all files are considered to expire on the same day.												
11.	<u>FILE SECURITY</u> 1 byte	Indicates security status of the file. 0 = No security protection. 1 = Security protection. Additional identification of the file is required before it can be processed.												
12.	<u>BLOCK COUNT</u> 6 bytes	Indicates the number of data blocks written in the file from the last header label to the first trailer label, exclusive of tapemarks. Count does not include checkpoint records. This field is used in trailer labels.												
13.	<u>SYSTEM CODE</u> 13 bytes	Uniquely identifies the operating system.												
14.	<u>RESERVED</u> 7 bytes	Reserved. Should be recorded as blanks.												

APPENDIX C: STANDARD MASS STORAGE DEVICE LABELS



Format 1: This format is common to all data files on disk.

Field Name and Length

Description

1. **FILE NAME**
44 bytes, alphanumeric EBCDIC. This field serves as the key portion of the file label. It can consist of three sections:
 1. **File ID** is an alphanumeric field assigned by the programmer and identifies the file. It can be 1 through 35 bytes in length if generation and version numbers are used, or 1 through 44 bytes in length if they are not used.
 2. **Generation Number**. If used, this field is separated from File ID by a period. It has the format Gnnnn, where G identifies the field as the generation number and nnnn (in decimal) identifies the generation of the file.
 3. **Version Number of Generation**. If used, this section immediately follows the generation number and has the format Vnn, where V identifies the field as the version of generation number and nn (in decimal) identifies the version of generation of the file.

Note: IBM DOS/VS System compares the entire field against the filename given in the DLBL card. The generation and version numbers are treated differently by the IBM OS/VS System.

Fields 2 through 33 constitute the DATA portion of the file label.

<u>Field</u>	<u>Name and Length</u>	<u>Description</u>
2.	<u>FORMAT IDENTIFIER</u> 1 byte, EBCDIC numeric	1 = format 1
3.	<u>FILE SERIAL NUMBER</u> 6 bytes, alphanumeric EBCDIC	Uniquely identifies a file/volume relationship. It is identical to the volume serial number of the first or only volume of a multivolume file.
4.	<u>VOLUME SEQUENCE NUMBER</u> 2 bytes, binary	Indicates the order of a volume relative to the first volume on which the data file resides.
5.	<u>CREATION DATE</u> 3 bytes, discontinuous binary	Indicates the year and the day of the year the file was created. It is of the form YDD, where Y signifies the year (0-99) and DD the day of the year (1-366).
6.	<u>EXPIRATION DATE</u> 3 bytes, discontinuous binary	Indicates the year and the day of the year the file may be deleted. The form of this field is identical to that of field 5.
7a.	<u>EXTENT COUNT</u> 1 byte, binary	Contains a count of the number of extents for this file on this volume. If user labels are used, the count includes the user label track as a separate extent. This field is maintained by the Disk Operating System.
7b.	<u>BYTES USED IN LAST BLOCK OF DIRECTORY</u> 1 byte, binary	Used by IBM Operating System Virtual Storage only for partitioned (library structure) data sets. Not used by the Disk Operating System Virtual Storage.
7c.	<u>SPARE</u> 1 byte	Reserved for future use.
8.	<u>SYSTEM CODE</u> 13 bytes	Uniquely identifies the operating system.
9.	<u>RESERVED</u> 7 bytes	Reserved for future use.
10.	<u>FILE TYPE</u> 2 bytes	The contents of this field uniquely identify the type of data file.

<u>Hex Code</u>	<u>Meaning</u>
4000	Sequential organization
2000	Direct organization
8000	Indexed organization
0200	Library organization
0000	Organization not defined in the file label

<u>Field</u>	<u>Name and Length</u>	<u>Description</u>																																					
11.	<u>RECORD FORMAT</u> 1 byte	The contents of this field indicate the type of records contained in the file.																																					
		<table border="1"> <thead> <tr> <th><u>Bit Position</u></th> <th><u>Content</u></th> <th><u>Meaning</u></th> </tr> </thead> <tbody> <tr> <td rowspan="3">0 and 1</td> <td>01</td> <td>Variable-length records</td> </tr> <tr> <td>10</td> <td>Fixed-length records</td> </tr> <tr> <td>11</td> <td>Undefined format</td> </tr> <tr> <td rowspan="2">2</td> <td>0</td> <td>No track overflow</td> </tr> <tr> <td>1</td> <td>File is organized using track overflow (IBM OS/VS only)</td> </tr> <tr> <td rowspan="2">3</td> <td>0</td> <td>Unblocked records</td> </tr> <tr> <td>1</td> <td>Blocked records</td> </tr> <tr> <td rowspan="2">4</td> <td>0</td> <td>No truncated records</td> </tr> <tr> <td>1</td> <td>Truncated records in file</td> </tr> <tr> <td rowspan="3">5 and 6</td> <td>01</td> <td>Control character ASA code</td> </tr> <tr> <td>10</td> <td>Control character machine code</td> </tr> <tr> <td>00</td> <td>Control character not stated</td> </tr> <tr> <td rowspan="2">7</td> <td>0</td> <td>Records are written without keys</td> </tr> <tr> <td>1</td> <td>Records are written with keys</td> </tr> </tbody> </table>	<u>Bit Position</u>	<u>Content</u>	<u>Meaning</u>	0 and 1	01	Variable-length records	10	Fixed-length records	11	Undefined format	2	0	No track overflow	1	File is organized using track overflow (IBM OS/VS only)	3	0	Unblocked records	1	Blocked records	4	0	No truncated records	1	Truncated records in file	5 and 6	01	Control character ASA code	10	Control character machine code	00	Control character not stated	7	0	Records are written without keys	1	Records are written with keys
<u>Bit Position</u>	<u>Content</u>	<u>Meaning</u>																																					
0 and 1	01	Variable-length records																																					
	10	Fixed-length records																																					
	11	Undefined format																																					
2	0	No track overflow																																					
	1	File is organized using track overflow (IBM OS/VS only)																																					
3	0	Unblocked records																																					
	1	Blocked records																																					
4	0	No truncated records																																					
	1	Truncated records in file																																					
5 and 6	01	Control character ASA code																																					
	10	Control character machine code																																					
	00	Control character not stated																																					
7	0	Records are written without keys																																					
	1	Records are written with keys																																					
12.	<u>OPTION CODES</u> 1 byte	Bits within this field are used to indicate various options used in building the file.																																					
		<table border="1"> <thead> <tr> <th><u>Bit Position</u></th> <th><u>Meaning</u></th> </tr> </thead> <tbody> <tr> <td>0</td> <td>If on, indicates data file was created using write validity check.</td> </tr> <tr> <td>1-7</td> <td>Unused.</td> </tr> </tbody> </table>	<u>Bit Position</u>	<u>Meaning</u>	0	If on, indicates data file was created using write validity check.	1-7	Unused.																															
<u>Bit Position</u>	<u>Meaning</u>																																						
0	If on, indicates data file was created using write validity check.																																						
1-7	Unused.																																						
13.	<u>BLOCK LENGTH</u> 2 bytes, binary	Indicates the block length for fixed-length records, or maximum block size for variable-length blocks.																																					
14.	<u>RECORD LENGTH</u> 2 bytes, binary	Indicates the record length for fixed-length records, or the maximum record length for variable-length records.																																					
15.	<u>KEY LENGTH</u> 1 byte, binary	Indicates the length of the key portion of the data records in the file.																																					
16.	<u>KEY LOCATION</u> 2 bytes, binary	Indicates the high-order position of the data record.																																					

<u>Field</u>	<u>Name and Length</u>	<u>Description</u>														
17.	<u>DATA SET INDICATORS</u> 1 byte	Bits within this field are used to indicate the following: <table border="1"> <thead> <tr> <th><u>Bit Position</u></th> <th><u>Meaning</u></th> </tr> </thead> <tbody> <tr> <td>0</td> <td>If on, indicates that this is the last volume on which this file normally resides. This bit is used by the DOS/VS DTFSR routine only. None of the other bits in this byte are used by the DOS/VS.</td> </tr> <tr> <td>1</td> <td>If on, indicates that the data set described by this file must remain in the same absolute location on the direct-access device.</td> </tr> <tr> <td>2</td> <td>If on, indicates that block length must always be a multiple of eight bytes.</td> </tr> <tr> <td>3</td> <td>If on, indicates that this data file is security protected; a password must be provided in order to access it.</td> </tr> <tr> <td>4-7</td> <td>Space. Reserved for future use.</td> </tr> </tbody> </table>	<u>Bit Position</u>	<u>Meaning</u>	0	If on, indicates that this is the last volume on which this file normally resides. This bit is used by the DOS/VS DTFSR routine only. None of the other bits in this byte are used by the DOS/VS.	1	If on, indicates that the data set described by this file must remain in the same absolute location on the direct-access device.	2	If on, indicates that block length must always be a multiple of eight bytes.	3	If on, indicates that this data file is security protected; a password must be provided in order to access it.	4-7	Space. Reserved for future use.		
<u>Bit Position</u>	<u>Meaning</u>															
0	If on, indicates that this is the last volume on which this file normally resides. This bit is used by the DOS/VS DTFSR routine only. None of the other bits in this byte are used by the DOS/VS.															
1	If on, indicates that the data set described by this file must remain in the same absolute location on the direct-access device.															
2	If on, indicates that block length must always be a multiple of eight bytes.															
3	If on, indicates that this data file is security protected; a password must be provided in order to access it.															
4-7	Space. Reserved for future use.															
18.	<u>SECONDARY ALLOCATION</u> 4 bytes, binary	Indicates the amount of storage to be requested for this data file at end-of-extent. This field is used by IBM OS/VS only. It is not used by DOS/VS routines.														
19.	<u>LAST USED TRACK AND RECORD ON THAT TRACK</u> 5 bytes, discontinuous binary	Indicates the last occupied track in a consecutive file organization data file. This field has the format CCHHR. It is all binary zeros if the last track in a consecutive data file is not on this volume, or if it is not consecutive organization.														
20.	<u>AMOUNT OF SPACE REMAINING ON LAST TRACK USED</u> 2 bytes, binary	A count of the number of bytes of available space remaining on the last track used by this data file on this volume.														
21.	<u>EXTENT TYPE INDICATOR</u> 1 byte	Indicates the type of extent with which the following fields are associated: <table border="1"> <thead> <tr> <th><u>Hex Code</u></th> <th><u>Meaning</u></th> </tr> </thead> <tbody> <tr> <td>00</td> <td>Next three fields do not indicate any extent.</td> </tr> <tr> <td>01</td> <td>Prime area (indexed) or consecutive area, etc., (i.e., the extent containing the user's data records).</td> </tr> <tr> <td>02</td> <td>Overflow area of an indexed file.</td> </tr> <tr> <td>04</td> <td>Cylinder index or master index area of an indexed file.</td> </tr> <tr> <td>40</td> <td>User label track area.</td> </tr> <tr> <td>80</td> <td>Shared cylinder indicator.</td> </tr> </tbody> </table>	<u>Hex Code</u>	<u>Meaning</u>	00	Next three fields do not indicate any extent.	01	Prime area (indexed) or consecutive area, etc., (i.e., the extent containing the user's data records).	02	Overflow area of an indexed file.	04	Cylinder index or master index area of an indexed file.	40	User label track area.	80	Shared cylinder indicator.
<u>Hex Code</u>	<u>Meaning</u>															
00	Next three fields do not indicate any extent.															
01	Prime area (indexed) or consecutive area, etc., (i.e., the extent containing the user's data records).															
02	Overflow area of an indexed file.															
04	Cylinder index or master index area of an indexed file.															
40	User label track area.															
80	Shared cylinder indicator.															

<u>Field</u>	<u>Name and Length</u>	<u>Description</u>
22.	<u>EXTENT SEQUENCE NUMBER</u> 1 byte, binary	Indicates the extent sequence in a multi-extent file.
23.	<u>LOWER LIMIT</u> 4 bytes, discontinuous binary	The cylinder and the track address specifying the starting point (lower limit) of this extent component. This field has the format CCHH.
24.	<u>UPPER LIMIT</u> 4 bytes	The cylinder and the track address specifying the end point (upper limit) of this extent component. This field has the format CCHH.
25-28.	<u>ADDITIONAL EXTENT</u> 10 bytes	These fields have the same format as the fields 21 through 24, above.
29-32.	<u>ADDITIONAL EXTENT</u> 10 bytes	These fields have the same format as fields 21 through 24, above.
33.	<u>POINTER TO NEXT FILE LABEL WITHIN THIS LABEL SET</u> 5 bytes, discontinuous binary	The disk address (format CCHHR) of a continuation label is needed to further describe the file. If field 9 indicates indexed organization, this field will point to a Format 2 file label within this label set. Otherwise, it points to a Format 3 file label, and then only if the file contains more than three extent segments. If no additional file label is pointed to, this field contains all binary zeros.

APPENDIX D: TRACK FORMATS FOR DIRECT-ACCESS STORAGE DEVICES

The track format for the 2311, 2314, 2319, 2321, 3330, and 3340 direct-access storage devices is illustrated in Figure 67. The names of the fields are given in the following discussion.

Index Marker: All tracks start with an index marker. It is a signal to the hardware that indicates beginning of the track.

Home Address: The home address, preceded by a gap, follows the index marker. The home address uniquely identifies each track by specifying the cylinder and head number.

Track Descriptor Record (Record 0): Record 0 consists of two parts: a count portion and a data portion. The count portion is the same as it is for any other record (see the following description of count for record 1. The 8-byte data portion is used to record information used by LIOCS. The information in the data portion depends on the data organization (direct or indexed) that is being used.

For direct organization, this portion in the form of CCHRR contains the address of the last record on the track and the number of bytes remaining on the track. This information is used to determine whether there is space for another record on the track. For indexed organization, the data portion contains the address of the last record in the cylinder overflow area and the number of tracks remaining in the cylinder overflow area. Record 0 is then used as the cylinder overflow control record.

Address Marker: All records after record 0 will be preceded by a 2-byte address marker. The address marker is a signal to the hardware that a record is starting.

Data Records: Data records can consist of a count and data portion for sequential organization, or a count, key, and data

portion for direct and indexed organizations.

1. **Count Portion.** The count portion contains the identification of each record, the key length, and the data length.
 - a. **Identification.** Each record is identified with its cylinder number, head number, or record number. The cylinder and head numbers will be the same as those of the home address. The record number will indicate a particular record on the track. That is, the first record after record 0 will be record 1, followed by record 2, etc. This 5-byte binary field in the form of CCHRR is often referred to as the record ID.
 - b. **Key Length.** The key length is specified in an 8-bit byte; its length can range from 0 to 255. This field will contain a zero if there is no key.
 - c. **Data Length.** The data length is specified in the 16 bits of the next two bytes.

Note: It is the count portion that identifies the presence or absence of a key, in addition to indicating the data length. In this way, each record is unique and self formatting.
2. **Key Portion.** The key portion of the record is normally used to store the control field of the data record such as a man number. Direct and indexed files must have a key portion.
3. **Data Portion.** The data portion of the record contains the data record.

Note that all records, including the data record, terminate with a 2-byte cyclic check. The hardware uses this cyclic check to ensure that it correctly reread what it had written. The cyclic check is cumulative and is appended to each record when it is written. Upon reading the record, the cyclic check is again accumulated and then compared with the appended cyclic check. If they do not agree, a data check is initiated.

The first byte of the count portion of each record and the home address is reserved for a flag byte. If a track

becomes defective, a utility program may be used to transfer the data to an alternate track. (Cylinders 200 through 202 are reserved for alternate tracks on the 2321. Strips 6 through 9 of subcell 19 of each cell are reserved for alternate tracks on the 2321.) In this case, a flag bit within the byte is set on to indicate that this is a defective track and the address of an alternate track will be placed in the record ID of record 0. Subsequent references to this defective track will result in the Supervisor accessing record 0 for the address of the alternate track.

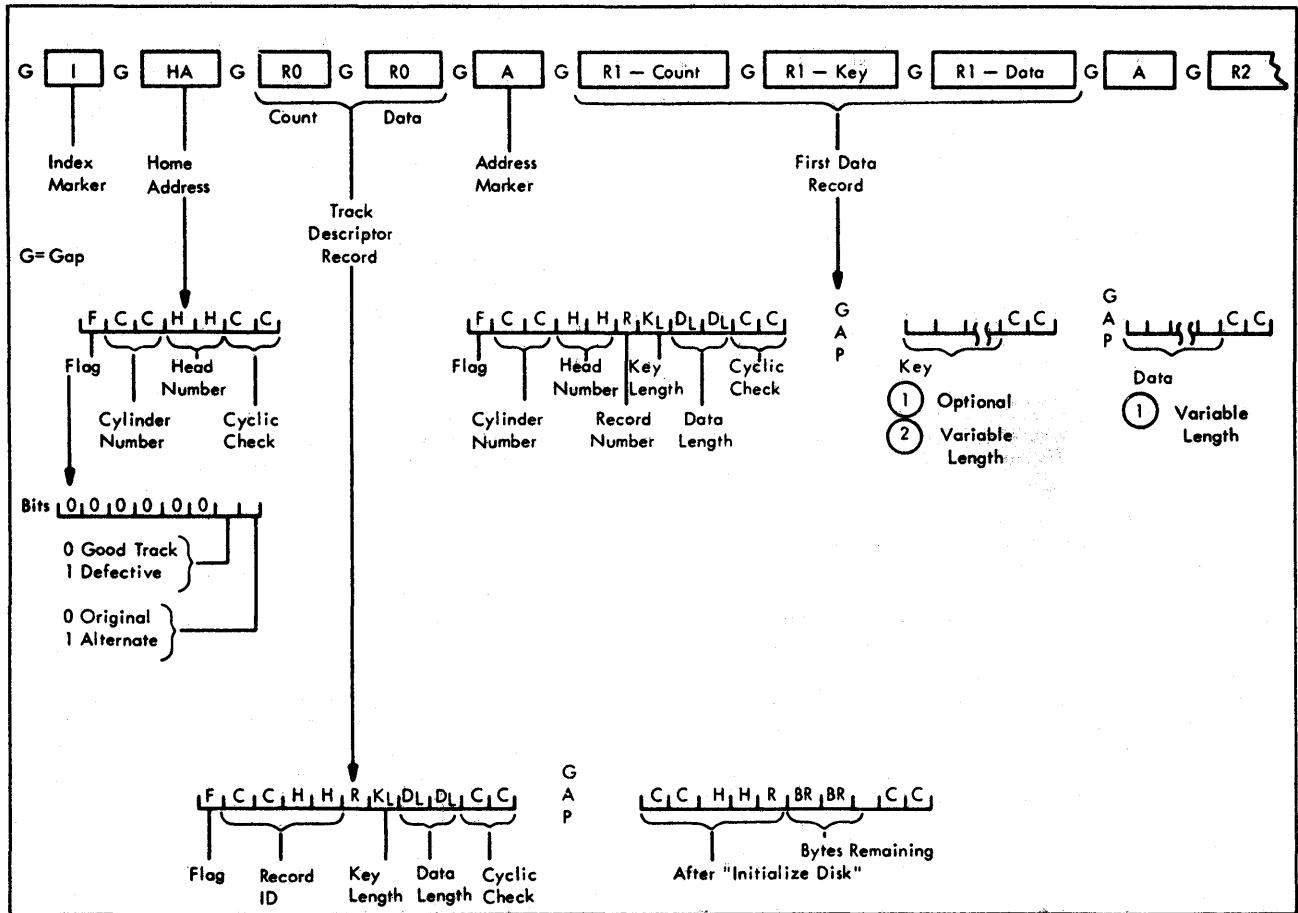


Figure 67. Track Format

The IBM DOS/VS COBOL Object-Time Subroutine Library, Program Number 5746-LM4, is packaged with the DOS/VS COBOL Compiler and also available as a separate product. It provides subroutines to be link edited with object modules produced by DOS/VS COBOL Compiler. It also provides subroutines that can be dynamically fetched during problem program execution.

There are several major categories of COBOL library subroutines:

- Input/output verb routines
- ASCII support routines
- Conversion routines
- Arithmetic verb routines
- Sort/Merge Feature interface routines
- Checkpoint (RERUN) routines
- Segmentation Feature routines
- Other verb routines
- Object-time debugging routines
- Object-time execution statistics routines
- Optimizer routines
- Transient routines

The following sections describe some of the more commonly used subroutines.

INPUT/OUTPUT SUBROUTINES

The input/output subroutines are used for the COBOL verbs DISPLAY (TRACE and EXHIBIT), ACCEPT, STOP (literal), READ, WRITE, REWRITE, OPEN, CLOSE, DELETE, and START printer spacing, printer overflow, input/output errors, disk formatting and extent handling, and tape and sequential disk labels.

Printer Spacing

The ILBDSPA0 subroutine is used to control printer spacing when the WRITE statement with the BEFORE/AFTER ADVANCING or POSITIONING option is specified in the source program.

Tape and Sequential Disk Labels

The ILBDUSL0 and ILBDNSL0 subroutines are used when user or nonstandard labels, respectively, are to be processed (LABEL RECORDS ARE data-name).

CLOSE WITH LOCK Subroutine

The ILBDCLK0 subroutine is given control on an OPEN if the file is ever closed with lock in the program. It checks whether the OPEN statement is used to open a file previously closed with lock. If the file was previously closed with lock, it issues an object-time message and terminates the current job.

WRITE Statement Subroutines

The ILBDVBL0 subroutine is used to write variable-length blocked records.

The ILBDDIO0 subroutine is used for writing files with direct organization (DTFDA).

The ILBDISM0 subroutine is used for writing files with indexed organization.

READ Statement Subroutines

The ILBDDSR0 subroutine is used to read sequentially the records of a directly organized file.

The ILBDDIO0 subroutine is used to read randomly the records of a directly organized file.

The ILBDISMO subroutine is used to read an indexed file.

REWRITE Statement Subroutines

The ILBDDIO0 subroutine is used to update records on a directly organized file.

The ILBDISMO subroutine is used to update an indexed file.

DISPLAY (EXHIBIT and TRACE) Subroutines

The ILBDDSP0 subroutine formats one or more operands into printed lines, performing conversions as needed.

The ILBDOSY0 and ILBDASY0 subroutines open SYSLST and/or SYSPCH and/or SYSIPT if there are DISPLAY or ACCEPT statements in a label declarative.

ACCEPT and STOP (literal) Statement Subroutines

The ILBDACP0 subroutine is used to handle ACCEPT statements for both SYSIPT and the console, as well as the STOP (literal) statement. The ILBDACP0 subroutine does not format or convert operands. For operands greater than 80 characters in length, any remainder in excess of the nearest multiple of 80 is ignored when accepting data from SYSIPT.

CLOSE Subroutine

The ILBDCRD0 subroutine is given control when a CLOSE UNIT statement is issued for a sequential input file with direct organization.

Multiple File Tape Subroutine

The ILBDMFT0 subroutine is given control when a reel contains more than one file and there are no standard labels.

Tape Pointer Subroutine

The ILBDIML0 subroutine locates the pointer to the physical tape drive associated with the logical unit for a particular tape file.

Input/Output Error Subroutines

The ILBDSAEO subroutine is used for processing input/output errors that occur on tape and sequential disk.

The ILBDDAEO subroutine is used for processing input/output errors that occur on directly organized files.

The ILBDISE0 subroutine is called whenever an input/output error occurs during the processing an indexed file.

The ILBDABX0 subroutine is used to issue a STXIT macro instruction causing control to be passed to it if there is an error on a unit-record device.

Disk Extent Subroutines

The ILBDFMT0 subroutine writes record 0 (R0) on each track of each extent of a directly organized file opened as output, and writes an end-of-file (EOF) record as the last record in the file. This subroutine is called after the file has been opened.

The ILBDXTN0 subroutine stores for subsequent use the extent information for directly organized files.

3886 OCR Subroutine

The ILBDOCR0 subroutine is used to perform I/O operations for the 3886 Optical Character Reader.

VSAM Subroutines

The ILBDINT0 subroutine does initialization for VSAM processing.

The ILBDVOC0 performs VSAM open and close functions.

The ILBDVIO0 performs all action requests for VSAM files (for example, READ, WRITE, REWRITE, START, DELETE).

These routines may call the Checkpoint subroutine and \$\$BCOBR1 discussed later in this chapter.

Auxiliary Subroutines

Certain input/output subroutines use auxiliary subroutines as follows:

<u>Auxiliary Routine</u>	<u>Used By</u>
ILBDMOVO	ILBDSPA0, ILBDNSL0, ILBDVBL0
ILBDIDA0	ILBDFMT0, ILBDDSR0

ASCII SUPPORT SUBROUTINES

The subroutine described below handles functions necessary for files written in ASCII. Other functions are handled by code generated by the compiler or by the subroutine ILBDSPA0.

Separately Signed Numeric Subroutine

The ILBDSSN0 subroutine is called to check the validity of signs described as TRAILING SEPARATE CHARACTER or LEADING SEPARATE CHARACTER.

CONVERSION SUBROUTINES

Eight numeric data formats are permitted in COBOL: five external (for input and

output) and three internal (for internal processing).

The five external formats are:

- External or zoned decimal
- External floating-point
- Sterling display
- Numeric edited
- Sterling report

The three internal formats are:

- Internal or packed decimal
- Binary
- Internal floating-point

The conversions from internal decimal to external decimal, from external decimal to internal decimal, and from internal decimal to numeric edited are performed in-line. The other conversions are performed by the COBOL library subroutines shown in Table 35.

Table 35. Functions of COBOL Library Conversion Subroutines

Subroutine Name and Entry Points	Conversion	
	From	To
ILEDEFL2	External floating-point	Internal decimal
ILEDEFL1	External floating-point	Binary
ILBDEFLO	External floating-point	Internal floating-point
ILBDBID0 ¹	Binary	Internal decimal
ILBDBID1 ¹		
ILBDBID2 ¹		
ILEDBIE0 ¹	Binary	External decimal
ILBDBIE1 ¹		
ILBDBIE2 ¹		
ILBDBII0 ²	Binary	Internal floating-point
ILBDBII1 ²		
ILBDTEF0 ²	Binary	External floating-point
ILBDTEF1 ²		
ILBDTEF2	Internal decimal	External floating-point
IFBDTEF3	Internal floating-point	External floating-point
ILBDIDB0	Internal decimal	Binary
ILBDIDB1	External decimal	Binary
ILBDDCI1	Internal decimal	Internal floating-point
ILBDDCI0	External decimal	Internal floating-point
ILBDIFD0	Internal floating-point	Internal decimal
ILBDIFD1	Internal floating-point	External decimal
ILBDIFB1	Internal floating-point	Binary integer and a power of 10 exponent
ILBDIFB2 ³		
ILBDIFE0 ³	Internal floating-point	Binary
ILBDIDRO	Internal decimal	Sterling report
ILBDIDTO	Internal decimal	Sterling non-report
ILBDSTIO	Sterling non-report	Internal decimal

¹The entry points used depend on whether the double-precision number is in registers 0 and 1, 2 and 3, or 4 and 5, respectively.

²The entry points are for single-precision binary and double-precision binary, respectively.

³This entry point is used for calls from other COBOL library subroutines.

ARITHMETIC VERB SUBROUTINES

Most arithmetic operations are performed in-line. However, involved calculations with very large numbers, such as decimal multiplication of two 30-digit numbers, are performed by COBOL library arithmetic subroutines. These subroutine names and their functions are shown in Table 36.

SORT/MERGE FEATURE INTERFACE ROUTINE

Communication between the Sort/Merge program and the COBOL program is maintained by ILBDSRT0 and ILBDMRG0.

CHECKPOINT (RERUN) SUBROUTINE

The ILBDCKP0 subroutine issues the checkpoint macro instruction, which will write checkpoint records on a programmer-specified tape or disk checkpoint device. There are two calling sequences to this subroutine. The first, ILBDCKP1, is activated during initialization when the addresses of all files in the program are entered in a table. The second, ILBDCKP2, is required to take checkpoints during a sorting operation.

If RERUN is requested during a sorting operation, ILBDSRT0 must gather a list of physical IOCS files in use by the Sort program every time Sort exits at E11, E21, and E31. ILBDSRT0 then calls the checkpoint subroutine which will take a checkpoint of all active files.

SEGMENTATION FEATURE SUBROUTINE

The Segmentation Feature requires an object time subroutine, ILBDSEM0. The ILBDSEM0 subroutine performs the following functions when segments are needed:

1. Loads and initializes independent segments not in storage.
2. Loads overlayable segments not in storage.
3. Initializes independent segments if the segment is in storage.
4. Branches to desired entry points.

OTHER VERB ROUTINES

There are also COBOL library subroutines for comparisons, the verbs MOVE and TRANSFORM, and other features of the COBOL language.

Compare Subroutines

The ILBDVCO0 subroutine compares two operands, one or both of which is variable in length. Each may exceed 256 bytes.

The ILBDIVL0 subroutine is used in comparisons involving the figurative constant ALL 'literal', where literal is greater than one character.

Table 36. Functions of COBOL Library Arithmetic Subroutines

Subroutine Name	Function
ILBDXMU0	Internal decimal multiplication (30 digits * 30 digits = 60 digits)
ILBDXDIO	Internal decimal division (60 digits/30 digits = 30 digits)
ILBDXPRO	Decimal fixed-point exponentiation
ILBDFPW0	Floating-point exponentiation
ILBDGPW0 ¹	Floating-point exponentiation

¹The ILBDGPW0 entry point is used if the exponent has a PICTURE clause specifying an integer. The ILBDFPW0 entry point is used in all other cases.

MOVE Subroutines

The ILBDVMO0 subroutine is used when one or both operands is variable in length and in-line instructions cannot be generated (for example, fields overlap, etc). Each may exceed 256 bytes. The subroutine has two entry points, depending on the type of MOVE: ILBDVMO0 (left-justified) and ILBDVMO1 (right-justified).

The ILBDANF0 subroutine is used to move the figurative constant ALL 'literal', where literal is greater than one character.

The ILBDANE0 subroutine is used to perform a right-or left-justified alphanumeric edited move.

The ILBDSMV0 subroutine handles moves to right-justified receiving fields either greater than 512 bytes in length or variable in length.

TRANSFORM Subroutine

The ILBDVTR0 subroutine transforms variable-length items using the ILBDTRN0 transform table.

Class Test Subroutine

The ILBDCLS0 subroutine is used to perform class tests for variable-length items and those fixed-length items longer than 256 bytes.

Note: The following tables are placed in the library for use by the in-line coding generated by the compiler and the subroutines called for by both the class test and TRANSFORM:

ILBDATB0 -- Alphabetic class test
ILBDETBO -- External decimal class test
ILBDITB0 -- Internal decimal class test
ILBDUTB0 -- Unsigned internal decimal
ILBDWTB0 -- Unsigned external decimal

SEARCH Subroutine

The ILBDSCH0 subroutine processes each search argument key according to type.

Main Program or Subprogram Subroutine

The ILBDMNS0 subroutine is a 1-byte switch tested in the code generated for EXIT PROGRAM, GOBACK, INIT1, and INIT2.

The ILBDSET0 subroutine must be called by a non-American National Standard COBOL program prior to any call to an American National Standard COBOL program. When calling ILBDSET0, standard linkage conventions must be observed; there are no parameters to be passed. The ILBDSET0 subroutine sets the 1-byte switch (ILBDMNS0) to X'FF'. This switch is tested in the American National Standard COBOL program to determine whether it is a main or a called program. The name of this subroutine can be changed to any name desired by the COBOL user.

OBJECT-TIME DEBUGGING SUBROUTINES

Three options are available for object-time debugging. These are the statement number option (STATE), the flow trace option (FLOW), and the symbolic debug option (SYMDMP). The subroutines for the first two options provide debugging information when a program terminates abnormally; the subroutines for the third option provide debugging information either at abnormal termination or dynamically during execution of a program. All of the subroutines are under the control of and are serviced by the Debug Control Subroutine (ILBDDBG0). This section discusses (1) the Debug Control Subroutine, and (2) the subroutines that are called in response to each of the three debugging options.

Debug Control Subroutine

The ILBDDBG0 subroutine is included in the load module whenever the CBL control card for a program contains at least one of the debugging options, or when the CBL control card for a program requests execution statistics.

Statement Number Subroutine

The ILBDSTN0 subroutine provides the number of the statement and the number of the verb being executed when abnormal termination occurs. If abnormal termination occurs during execution of an instruction outside of the COBOL program, the statement number that is provided is that of the last COBOL instruction executed.

Flow Trace Subroutine

Space is allocated at compile time for a flow trace table using the programmer-specified number in the FLOW option of the CBL card. (If FLOW=0 was specified for a subprogram, no space is allocated; rather the subprogram shares the table space reserved by that program preceding it in the calling sequence for which a FLOW specification was made.)

Each time the flow trace subroutine ILBDFLW0 receives control from the COBOL program, it inserts the executing program's identification as well as the card number of the current procedure into the next available position in the table. When the end of the table is reached, subsequent entries overlay the first set of entries. The procedure is repeated until the end of the program or until abnormal termination. If abnormal termination occurs, the subroutine produces a list of each entry of the table, beginning with the earliest entry.

Symbolic Debug Subroutines

The symbolic debug subroutines provide a formatted symbolic dump, either dynamically at execution time, or at abnormal termination.

The following subroutines perform initialization and process debug control cards:

ILBDMP10, ILBDMP11, ILBDMP12, ILBDMP13, ILBDMP14, and either ILBDMP01 or ILBDMP02.

To provide a dump at abnormal termination, the following subroutines are used:

ILBDMP20, ILBDMP21, ILBDMP22, ILBDMP23, ILBDMP24, ILBDMP25 and ILBDMP01 or ILBDMP02. These subroutines are not included in the load module at link edit time; they are loaded dynamically during program execution.

The ILBDADRO subroutine tests the validity of an address calculated for a subscripted identifier or the validity of the starting and ending addresses of a variable-length identifier used as the receiving field in a MOVE statement.

OBJECT-TIME EXECUTION STATISTICS SUBROUTINES

The object-time execution statistics subroutines enable the printing of execution statistics when a program terminates normally (via STOP RUN or GOBACK in the main program) and when a program terminates abnormally. In addition, when COUNT is requested, the debug control subroutine (described above) is also included in the load module.

COUNT Initialization Subroutine

The ILBDTC00 subroutine is called from the debug control subroutine to get space for an initialize the table and chains which service the COUNT options.

COUNT Frequency Subroutine

The ILBDCT10 subroutine maintains the execution frequency statistics.

COUNT Termination Subroutine

The ILBDTC20 subroutine is included in all COBOL load modules. It determines if execution frequency statistics were requested.

COUNT Print Subroutine

The ILBDTC30 subroutine formats and prints the execution frequency statistics.

OPTIMIZER SUBROUTINES

GO TO ... DEPENDING ON Subroutine

The ILBDGDO0 subroutine is called only when the optimization option (OPT) has been specified. It is used to more efficiently process GO TO statements with the DEPENDING ON option in both segmented and nonsegmented programs.

Optimizer DISPLAY Subroutine

The ILBDDSS0 subroutine is used to print or type certain data types on SYSLST or the console, respectively.

TRANSIENT SUBROUTINES

The IBM DOS/VS COBOL Object-Time Subroutine Library includes routines that are dynamically fetched during program execution. These routines are as follows:

Symbolic Debug Subroutines

With the exception of ILBDDBG0, the symbolic debug subroutines described previously are transient routines.

SYMDMP Error Message Subroutine

The \$\$BCOBEM subroutine prepares SYMDMP error messages.

Error Message Subroutine

The \$\$BCOBER subroutine prepares input/output error messages.

Error Message Print Subroutine

The \$\$BCOBER1 subroutine prints the error messages prepared by \$\$BCOBER and provides a dump if the DUMP option is in effect.

Reposition Tape Subroutine

The \$\$BFCMUL subroutine resets the PUB pointer for a particular (SYSnnn) device to the same as that saved earlier by the subroutine ILBDIML0.

Note: If dynamically fetched subroutines are required during problem program execution, the Subroutine Library must be installed on the object machine. If dynamically fetched subroutines are not required during problem program execution, the object-time subroutines can be link edited on the source machine; the Subroutine Library must in this case be installed on the source machine.

APPENDIX F: SYSTEM AND SIZE CONSIDERATIONS

This appendix contains information concerning system and size requirements for the DOS/VS COBOL compiler, execution time considerations, and the Sort/Merge Feature. Additional information used in estimating the virtual and auxiliary storage requirements is contained in the publication IBM DOS/VS COBOL Compiler and Library, Installation Reference Material.

MINIMUM MACHINE REQUIREMENTS FOR THE COMPILER

1. A System/370 supported by DOS/VS. A minimum of 60K bytes of virtual storage is required.
2. Five work files. The system logical unit SYSLNK must be assigned to a single area (extent) on a 2314, 2319, 3330, or 3340 mass storage device. Four programmer logical units (SYS001 through SYS004) must reside on 2400, 3410, 3420 tape units, or on 2314, 2319, 3330, or 3340 mass storage devices. At least one programmer logical unit as well as the operating system must reside on a mass storage device (that is, a 2311, 2314, 2319, 3330, or 3340). If the three remaining logical units reside on tape, there must be a separate tape unit for each file. If they reside on mass storage devices, there must be enough space on those devices. An additional logical unit, SYS005, must be assigned if the symbolic debug option (SYMAMP) is being used. Logical unit SYS006 must be assigned for the FIPS flagger.

Work file assignments must be made as follows:

SYSLNK - mass storage device
SYS001 - mass storage device
SYS002 - mass storage device or tape unit
SYS003 - mass storage device or tape unit
SYS004 - mass storage device or tape unit
SYS005 - mass storage device or tape unit
SYS006 - mass storage device unit

Note that SYSLNK need only be assigned at compile time if the CATAL or LINK option is in effect.

The filenames for SYSLNK and SYS001 through SYS006 on the TLBL or DLBL statements are IJSYSLN, IJSYS01, IJSYS02, IJSYS03, IJSYS04, IJSYS05, and IJSYS06, respectively. If the "filename" parameter of the SYMDMP option is specified, this filename is used instead of IJSYS05 on DLBL statements.

3. A device, such as a printer keyboard, for direct operator communication.
4. A device, such as a card reader, for the job input stream.
5. A device, such as a printer or tape unit, for system output files.
6. The floating-point arithmetic feature, if floating-point literals or calculations are used.

SOURCE PROGRAM SIZE CONSIDERATIONS

Compiler Capacity

This section contains information which must be considered in determining the limitations on the size of a COBOL source program in a specific virtual storage size. It also contains information to aid the programmer in determining how his source program affects usage of space at compilation time.

The capacity of the COBOL compiler is limited by two general conditions: (1) the total table requirement may be greater than the space available and (2) the fact that an individual table (with the exception of the ADCON and cross-reference tables) may need to be longer than 32,767 bytes. If either of these conditions are met during compilation, one of the following error messages will be issued:

ILA0001I-D NO MORE TABLE SPACE AVAILABLE. COMPILATION ABANDONED.

ILA0003I-D A TABLE HAS EXCEEDED THE MAXIMUM SIZE. COMPILATION ABANDONED.

ILA6007I-D TABLE HAS EXCEEDED MAXIMUM SIZE. LISTX, OBJECT MODULE, AND DECK WILL BE INCOMPLETE. INCREASE PARTITION.

In each case, compilation is terminated. However, in the first and third cases, or in the case of overflow of the ADCON or cross-reference table, the program may be recompiled with a larger size parameter.

The compiler will accept and compile a 1500 card program in the minimum storage of 60K. In this configuration, the minimum size compiler input/output areas must be allocated. If both LINK and DECK are specified, more storage is required for buffer space, which reduces the space available to a given program. Within this configuration, the compiler will accept programs much larger than 1500 cards; the specific size limitation for any storage size depends entirely on the statement mix in that program, but the limiting factors are described in the next section.

The overall critical limit using the minimum buffer specification may be expressed as follows:

$$2 (\text{number of pn's} + \text{gn's} + \text{literals} + \text{virtuals}) + 8A + S (L + 5D + 8V + 3P) \leq 14336 + C$$

where the number of virtuals is the number of calls to COBOL object-time subroutine entry points and subprograms specified in a CALL statement, and V is the number of unique such names; also

A = number of entries in the ADCON table as defined below

S = 1 if the Segmentation Feature is required and NOOPT is in effect; otherwise 0

L = length of optimized literals

D = number of segment discontinuities in the Procedure Division

P = number of PERFORM exits and altered GO TO statements

C = any storage over 60K assigned to the program

It should be noted that the number of gn's is reduced when using OPT.

Within this configuration, assuming no Report Section, the compiler will accept for example:

300 procedure references assuming an average procedure-name length of 12 characters

25 OCCURS clauses with the DEPENDING ON option

10 files, assuming an average of 3 subordinate record entries

Effective Storage Considerations

The performance of the compiler is affected by the amount of storage it is allocated. The compiler will take advantage of any extra storage it is assigned. Furthermore, the use of a BUF parameter tailored to the work file device type in use is recommended. The following CBL parameters positively affect compile-time performance:

OPT
SYNTAX (CSYNTAX)
NOLIB
BUF

The amount of virtual storage within the compiler's partition and the limitation on the size of an individual internal table are two factors that limit the capacity of the compiler. The limitation on the size of internal tables can, in some instances, be overcome by the spilling over of some tables onto external devices. However, spilling over may cause a severe degradation of performance. The storage limitation should not be reached by any reasonable use of the language. However, within a limited storage capacity excessive use of certain features and combination of features in the language could make compilation impossible. Some of the features that significantly affect storage usage are:

1. ADCON Table

Each entry occupies 8 bytes. This table is not limited to the maximum size of 32,767 bytes. Entries are based on:

- Number of 4096-byte segments in the Working-Storage Section
- Number of 4096-byte segments in a file buffer area
- Number of referenced procedure-names
- Number of implicit procedure-name references such as those generated by IF, SEARCH, and GENERATE statements, ON SIZE ERROR, INVALID KEY, and AT END options, the OCCURS clause with the DEPENDING ON option, USE sentences, and the Segmentation Feature
- Number of files

The size of this table is significantly reduced when using OPT.

2. Procedure-Name Table

This table contains the number of definitions written in a section and unresolved procedure references. Procedure references are resolved at the end of a section if the definition of the procedure-name is in that section or a preceding section. Therefore, forward references beyond a section impact space.

3. OCCURS DEPENDING ON Table

This table contains an entry for each unique object of an OCCURS clause with the DEPENDING ON option. The size of an entry is (2 + length of name + length of each qualifier) bytes.

4. Index Table

An entry is made for each INDEXED BY clause consisting of 11 bytes for each index.

5. File Table

An entry is made for each file specified in the program. Each entry occupies 60 bytes of storage.

6. Report Writer Tables

A considerable amount of information is maintained concerning each RD such as controls, sums, headings, footings, routines to be generated, etc. The contents of the table is increased by the existence of qualification and subscripting in the Report Section. Approximately 30 reports can be processed, without exceeding the limit of a table.

7. Operand Table

Entries are made depending on the number of operands in a statement. This table could reach its limits by the use of compound nested IF statements or GO TO DEPENDING ON statements with an excessive number of branch points.

8. Dictionary Table

An entry is made for each procedure-name and each data-name in the program. A procedure entry consists of (7 or 9 + length of name) bytes. A data entry consists of (length of name + n) bytes, where n is determined by the attributes of the

data item. Some of the features that contribute to the value n are:

- One byte for each character in a numeric edited or alphanumeric edited item PICTURE clause.
- Five bytes for an elementary item with a sterling report PICTURE clause.
- Three bytes for an item subordinate to an OCCURS clause.

In the statistics output, an indication is given if spill of this table occurred. If spill occurred, increasing the partition size assigned to the compiler should increase performance.

9. Literal Tables

The total length of all literals (after optimization) may not exceed 32511 bytes. No more than 16255 literals may be specified.

If the segmentation feature is used, an area corresponding to the total length of all optimized literals must be kept free during the time the ADCON table is being built. Therefore, a segmented program with literals may need more storage.

10. Miscellaneous Tables

The existence of the following items causes entries to be made into tables and impacts the total space required for compilation.

- SAME (RECORD) AREA clause
- Subscripting
- Intermediate Arithmetic Results
- Complex Arithmetic Expressions
- Complex Logical Expressions
- APPLY clauses
- Special-Names
- RERUN clauses
- Error messages
- XREF
- Segmentation feature
- VERBSUM/VERBREF

EXECUTION TIME CONSIDERATIONS

The amount of virtual storage must be sufficient to accommodate at least:

- The selected control program
- Support for the file processing techniques used

- Load module to be executed

- Dynamic storage for VSAM, 3886 processing, and COUNT.

When the OPTIMIZE option is specified, the number of procedure blocks in the program cannot exceed 255. A procedure block is approximately 4096 bytes of Procedure Division code.

COBOL programs compiled with any of the symbolic debugging options (STATE, FLOW, SYMDMP) have different requirements at execution time than similar programs compiled without these options. The following differences should be noted:

- If the SYMDMP option is in effect, the work file required at compile time (SYS005) must be present at execution time.
- The size of the load module will increase by about 3200 bytes if the SYMDMP option is in effect. In addition, since the object-time subroutine that provides SYMDMP output is invoked dynamically, the programmer must provide space in the partition amounting to $S + V$. When only an abnormal termination dump is required, $S = 4000$ and $V = 0$; that is, 4000 extra bytes must be available. When dynamic dumps are required, $S = 11,000$ and V is approximately $25 * \text{number of line-control cards} + 10 * \text{the number of identifiers specified on these line-control cards}$.
- The size of the load module will increase by $4500 + V$ bytes if the FLOW option is in effect. V is a variable factor that depends upon the number specified by the programmer on the CBL card. V is calculated using the formula:

$$V = 92 + 4 * nn + 8 * p$$

where "nn" is any number from 0 through 99, and "p" is the number of procedure-names in the program.

- The size of the load module will increase by $4600 + V$ bytes when the STATE option is in effect. V is approximately $5 * \text{the number of COBOL statements in the program}$.
- When both SYMDMP and FLOW are in effect, the size of the load module will increase by the amount it would for FLOW alone, and the size of the partition increases by the amount it would for SYMDMP alone.

- A SIZE parameter must be specified on the EXEC card for VSAM and 3886 processing and if COUNT is requested on the CBL card.

COBOL programs with the execution frequency option COUNT have the following additional requirements:

- The size of the load module will increase by about $6000 + V$ bytes (if any of the symbolic debugging options are in effect) to $8900 + V$ bytes (if the symbolic debugging options are not in effect). V is calculated using the formula:

$$V = (54 * \text{pgm}) + (8 * \text{nvb}) + (7 * \text{npr}) + ((4 + \text{sym}) * \text{vbl}) + \text{pnl}$$

where

pgm is the number of COBOL program units being monitored by COUNT

nvb is the number of verbs in the program units

npr is the number of procedure-names plus inserted procedure-names in the program

sym is zero unless SYMDMP is in effect, then it becomes two

vbl is the number of verb blocks in the program (which can be estimated as 1/3 the number of verbs in the program)

pnl is the sum of the lengths of the procedure-names.

- The increase in dynamic storage is estimated using the formula

$$D = 512 + (72 * \text{pgm}) + (4 * \text{vbl})$$

where

pgm is the number of COBOL programs being monitored by COUNT

vbl is the number of verb blocks in the program (which can be estimated as 1/3 the number of verbs in the program).

MULTIPROGRAMMING CONSIDERATIONS

In a system which supports the batch-job foreground (NPARTS = 2 or more) and private core-image library options, the Linkage Editor can execute in any foreground partition (as well as the background

partition) provided a minimum of 14K or 64K of storage is assigned to the partition. When executing in a foreground partition, a private core image library must be assigned.

In the multiprogramming environment described above, the COBOL compiler can be executed in any partition having a minimum of 64 bytes in the following manner:

1. At system generation time, link edit the compiler in the background partition and place it in the system core image library.
2. Link edit the compiler in each desired foreground partition and place the output in a private core image library assigned to that partition.
3. When executing the compiler in a foreground partition, assign the appropriate private core image library.

SORT FEATURE CONSIDERATIONS

The DOS/VS SORT/MERGE Program Product, Program Number 5746-SM1, must be executed under control of DOS/VS. It requires the following minimum machine configuration:

1. The DOS/VS SORT/MERGE Program Product uses 16K bytes; additional storage is

needed for DOS/VS and for user-written routines. user-written routines, (that is, the COBOL program, etc).

Note: Performance often increases significantly if 50K is available for operation of the Sort/Merge program. At the 100K level, the performance could be even higher.

2. Standard instruction set.
3. At least one 2314, 2319, 3330, 3333, or 3340 work file. (System residence requirements may necessitate having an additional disk storage unit for sorting.)
4. One IBM 1403, 1443, or 3211 Printer, or one IBM operator communication device (for example, 3215).
5. One IBM 1442, 2501, 2520, 2540, 3505, 3525, or 2560 Card Reader, or one IBM 2400 or 3400 Series Magnetic Tape Unit (7- or 9-track) assigned to SYSIPT and SYSRDR.
6. Three IBM 2400 or 3400 Series Magnetic Tape Units for work files when tape units are to be used for intermediate storage.

For specific size, device, and work file requirements of the other Sort/Merge products, see the respective Programmer's Guides as noted in the preface.

COMMUNICATION REGION

The Communication Region is a 46-byte storage area in a unique area within the Supervisor for each partition used by the Supervisor and the COBOL compiler. The structure of the Communication Region is illustrated in Figure 68.

Fields in the Communication Region are addressed relative to the first byte of the region. An asterisk (*) identifies the fields available to the COBOL programmer.

<u>Byte(s)</u>	<u>Meaning</u>
0-7*	Calendar date supplied during the IPL procedure or by the DATE control statement. This field can be used for dating printed output of the COBOL program via the special register CURRENT-DATE. The date can be in one of two forms: mm/dd/yy or dd/mm/yy where <u>mm</u> is month, <u>dd</u> is day, and <u>yy</u> is year. The form is chosen by the installation at system generation time.
8,9	Address of the background program label area.
10-13	Reserved for control program use.
14-22*	User area for inter-program or intra-program communication. This field can be referenced in a COBOL program executing in the background via the special register COM-REG. All eleven bytes are initialized to binary zeros when a JOB control statement is encountered.

<u>Byte(s)</u>	<u>Meaning</u>
23*	User program switch indicators (UPSI). The condition-name associated with the status of the UPSI switches can be specified in the COBOL program via the Special-Names paragraph of the Environment Division. UPSI byte switches are set by the UPSI control statement. The condition-name associated with each may be tested in the Procedure Division of the COBOL program. UPSI byte switches are initialized to binary zeros when a JOB control statement is encountered.
24-31	Jobname for programs located in the operand field of the JOB control statement.
32-35	Address of the uppermost byte of the program area.
36-39	Address of the uppermost byte of the last phase loaded into the program area.
40-43	Address of the uppermost byte used in loading any phase of the program.
44,45	Length of the program label area.

Note: When FIPS is in effect, the COBOL compiler uses byte 12.

	Date			User Area			Job Name																
	Mo/Day/Yr			(Inter -or Intraprogram Communication)			(Entered from Job Control)																
	or		Reserved		Program Switches (UPSI)																		
	Day/Mo/Yr																						
Bytes	0	7	8	9	10	13	14			22	23	24			31	32	35	36	39	40	43	44	45

Address of first byte supplied in register 1 by COMRG

Figure 68. Communication Region in the Supervisor

This appendix illustrates the necessary job control statements and their sequence for five typical programs:

1. Creating a Direct File
2. Retrieving and Updating a Direct File
3. Creating an Indexed File
4. Retrieving and Updating an Indexed File
5. Sorting an Unlabeled Tape File

In all five programs the programmer has requested the following compiler options through the OPTION control statement:

- NODECK -- No punched card output for the object program is needed.
- LINK -- The object module is to be linkage edited.
- LIST -- The COBOL source statements are to be printed on SYSLST.
- LISTX -- A Procedure Division map with global tables, literal pool, and register assignments is to be printed on SYSLST.
- SYM -- A Data Division map is to be printed on SYSLST.
- ERRS -- The diagnostic messages of the COBOL compiler are to be printed on SYSLST.

The EXEC FCOBOL statement calls for execution of the FCOBOL compiler.

By using the CBL card, the programmer indicates that in this source program the quotation mark (") is used for nonnumeric literals.

The ASSIGN clause in the COBOL source program specifies a system-name with the following fields:

(Non-VSAM)
SYSnnn-class-device-organization[-name]

(VSAM)
SYSnnn[-class][-device][-organization]
[-name]

The ASSGN control statement for a file must specify the same logical unit as the SYSnnn field of system-name. The ASSGN

statement assigns the logical unit to a specific hexadecimal address. The address specified must be associated with the device whose number is given in the device field of system-name.

The DLBL control statement for a labeled file on a mass storage device must contain the same name as system-name. This is the name by which the file is known to the control program. (The name field of system-name is optional. If name is omitted, the DLBL statement must specify the logical unit (SYSnnn) as the file-name.) The code field of the DLBL statement must correspond to the class and organization fields of system-name as follows:

DLBL "code"	ASSIGN "class"	ASSIGN "organization"
SD	DA or UT	S
		AS (entry-sequenced file) omitted (key-sequenced file)
DA	DA	A or U, D or W
ISC	DA	I
ISE	DA	I

The first EXTENT control statement for a file on a mass storage device must specify the same logical unit as the SYSnnn field of system-name. (Subsequent EXTENT statements for the same file, if they immediately follow the first, may omit this field.) The type of the extent must be compatible with the organization field of system-name as follows:

EXTENT "type"	ASSIGN "organization"
1 (data area, no split cylinder)	S, A, U, I, D, W AS
2 (overflow area for indexed file)	I
3 (index area for indexed file)	I
4 (data area, split cylinder)	S, A, U, I, D, W

DIRECT FILES

The following two examples illustrate the job control statements necessary for programs that create and update a direct file.

In the COBOL source programs, the programmer has written:

```
SELECT DA-FILE ASSIGN TO
  SYS015-DA-2311-A-MASTER...
```

```
SELECT CARD-FILE ASSIGN TO
  SYS007-UR-2540R-S...
```

In the READFILE source program, the programmer has written:

```
SELECT PRINT-FILE ASSIGN TO
  SYS008-UR-2403-S...
```

(Note the relationship between the system-names in the source programs and the control statements.)

The LBLTYP statement defines the amount of storage to be reserved to process labels for the DA file. The file has one extent.

The EXEC LNKEDT statement causes the object program to be link edited.

An ASSGN control statement assigns logical unit SYS007 to the hexadecimal address 00C -- a 2540R Card Reader.

In the updating program, another ASSGN statement assigns logical unit SYS008 to the hexadecimal address 00E -- a 1403 Printer.

The next series of statements identify the direct file completely.

The ASSGN statement identifies the file as residing on logical unit SYS015, which has the hexadecimal address of 192 -- a 2311 Disk Drive.

The DLBL statement specifies the filename as MASTER, with an expiration date of the 365th day of 1973, and that the file has direct organization (DA).

The EXTENT statement specifies that the file residing on logical unit SYS015 has a serial number 111111, that the extent is a data area with no split cylinder and that this is the first (and only) extent for the file (type and sequence number 1,0), that the file begins on relative track 1020 (track 0 of cylinder 102), and that the file occupies 100 tracks.

(Note that in the EXTENT statement, the relative track number (1020) is not required for the input DA file of the updating program, since the system will use the file labels for this information.)

The EXEC statement begins execution of the problem program, and is followed by input data.

The /* statements indicate end-of-data, the /% statement indicates end-of-job.

Creating a Direct File

```
// JOB CREATEDA
// OPTION NODECK, LINK, LIST, LISTX, SYM, ERRS
// EXEC FCOBOL
  CBL QUOTE
```

```
  {COBOL source deck}
/*
// LBLTYP NSD(01)
// EXEC LNKEDT
// ASSGN SYS007,X'00C'
// ASSGN SYS015,X'192'
// DLBL MASTER,,74/365,DA
// EXTENT SYS015,111111,1,0,1020,100
// EXEC
```

```
  {input data cards}
/*
/%
```

Retrieving and Updating a Direct File

```
// JOB READFILE
// OPTION NODECK, LINK, LIST, LISTX, SYM, ERRS
// EXEC FCOBOL
  CBL QUOTE
```

```
  {COBOL source deck}
/*
// LBLTYP NSD(01)
// EXEC LNKEDT
// ASSGN SYS007,X'00C'
// ASSGN SYS008,X'00E'
// ASSGN SYS015,X'192'
// DLBL MASTER,,74/365,DA
// EXTENT SYS015,111111,1,0,1020,100
// EXEC
```

```
  {input data cards}
/*
/%
```

INDEXED FILES

The following two examples illustrate the job control statements necessary for programs that create and update an indexed file.

In the CREATEIS source program, the programmer has written:

```
SELECT IS-FILE ASSIGN TO
  SYS015-DA-2314-I-MASTER
ACCESS IS SEQUENTIAL
RECORD KEY IS REC-ID.
```

In the RANDIS source program, the programmer has written:

```
SELECT IS-FILE ASSIGN TO
  SYS015-DA-2314-I-MASTER
ACCESS IS RANDOM
NOMINAL KEY IS KEY-ID
RECORD KEY IS REC-ID.

SELECT PRINT-FILE ASSIGN TO
  SYS008-UR-1403-S
RESERVE NO ALTERNATE AREAS.
```

In both source programs, he has written:

```
SELECT CARD-FILE ASSIGN TO
  SYS007-UR-2540R-S.
```

I-O-CONTROL.

```
APPLY MASTER-INDEX TO 2311 ON IS-FILE.
```

(Note the relationship between the source program statements and the job control statements.)

The LBLTYP statement defines the amount of storage reserved to process labels for the indexed file. The file has three extents: a master index extent, a cylinder index extent, and a data extent.

The EXEC LNKEDT statement causes the object module to be link edited.

An ASSGN control statement assigns logical unit SYS007 to the hexadecimal address 00C -- a 2540R Card Reader.

In the retrieval program, another ASSGN statement assigns logical unit SYS008 to the hexadecimal address 00E -- a 1403 Printer.

The next ASSGN statement assigns logical unit SYS015 to the hexadecimal address 193 -- a 2314 Disk Drive.

The DLBL statement names the file as MASTER, and indicates the expiration date as the 365th day of 1974. In the file creation program, the file label is indexed sequential using Load Create (code ISC); in

the retrieval program, the file label is indexed sequential using Load Extension, Add or Retrieve (code ISE).

The first EXTENT statement is identified as a master index (type and sequence numbers are 4,0), and the relative track is 900 (the extent begins on cylinder 180 track 0), and the extent is 20 tracks long.

The second EXTENT statement is identified as a cylinder index (type and sequence number are 4,1), the relative track is 1820 (the extent begins on cylinder 91, track 0), and the extent is 20 tracks long.

(Note that the extents assigned to master and cylinder indexes must be contiguous, and that the master index must precede the cylinder index on the disk pack. Also note, that if a master index is not requested, the first extent is that for the cylinder index, which would be type 4, sequence number 1.)

The third EXTENT statement is identified as a data area (type 1) and is the third extent named for this file. The relative track is 0020 (the extent begins on cylinder 1, track 0), and the extent is 1760 tracks long.

End-of-data is indicated with the /* statement; end-of-job is indicated with the /& statement.

Creating an Indexed File

```
// JOB CREATEIS
// OPTION NODECK, LINK, LIST, LISTX, SYM, ERRS
// EXEC FCOBOL
  CBL QUOTE
```

{COBOL source deck}

```
/*
// LBLTYP NSD(03)
// EXEC LNKEDT
// ASSGN SYS007,X'00C'
// ASSGN SYS015,X'193'
// DLBL MASTER,,74/365,ISC
// EXTENT SYS015,111111,4,0,1800,20
// EXTENT SYS015,111111,4,1,1820,20
// EXTENT SYS015,111111,1,2,0020,1760
// EXEC
```

{input data card}

```
/*
/ &
```

Retrieving and Updating an Indexed File

```
// JOB RANDIS
// OPTION NODECK, LINK, LIST, LISTX, SYM, ERRS
// EXEC FCOBOL
```

```
{COBOL source deck}
// LBLTYP NSD(03)
// EXEC LNKEDT
// ASSGN SYS007, X'00C'
// ASSGN SYS008, X'00E'
// ASSGN SYS015, X'193'
// DLBL MASTER, , 73/365, ISE
// EXTENT SYS015, 111111, 4, 0, 1800, 20
// EXTENT SYS015, 111111, 4, 1, 1820, 20
// EXTENT SYS015, 111111, 1, 2, 0020, 1760
// EXEC
```

```
{input data cards}
```

```
/*
/ &
```

FILES USED IN A SORT OPERATION

The following example illustrates the job control statements necessary for a program that sorts an unlabeled tape file.

In the COBOL source program, the programmer has written:

```
SELECT NET-FILE-IN ASSIGN TO
  SYS007-UT-2400-S.
```

```
SELECT NET-FILE-OUT ASSIGN TO
  SYS008-UT-2400-S.
```

```
SELECT NET-FILE ASSIGN TO 3
  SYS001-UT-2400-S.
```

NET-FILE-IN is the input file;
NET-FILE-OUT is the output file; NET-FILE
is the sort work file, which utilizes three
tape units.

(Note the relationship between the system-names in the COBOL source program and the control statements.)

The EXEC LNKEDT statement causes the job to be link edited.

The first two ASSGN control statements assign the logical unit SYS007 to hexadecimal address 181, and logical unit SYS008 to hexadecimal address 182. SYS007 is the sort input file, and SYS008 is the sort output file.

The last three ASSGN statements assign logical unit SYS001 to hexadecimal address 183, logical unit SYS002 to hexadecimal address 281, and logical unit SYS003 to hexadecimal address 282. SYS001, SYS002, and SYS003 are the logical units that must be used for sort work files. The sort work files must be assigned to 9-track tape units. At this installation, 9-track tape drives are associated with hexadecimal addresses 183, 281, and 282.

Sorting an Unlabeled Tape File

```
// JOB SORTCOB
// OPTION NODECK, LINK, LIST, LISTX, SYM, ERRS
// EXEC FCOBOL
  CBL QUOTE
```

```
{COBOL source deck}
// EXEC LNKEDT
// ASSGN SYS007, X'181'
// ASSGN SYS008, X'182'
// ASSGN SYS001, X'183'
// ASSGN SYS002, X'281'
// ASSGN SYS003, X'282'
// EXEC
/ &
```

This appendix contains information on how to generate a listing of compile-time diagnostic messages.

COMPILE-TIME MESSAGES

The user can request a complete listing of the diagnostics generated by this compiler simply by compiling a program with a PROGRAM-ID of ERRMSG. For a description of the formats of compiler diagnostics and information about generating this listing, see the chapter entitled "Output" in this publication.

OPERATOR MESSAGES

This section lists the messages issued to SYSLOG by the IBM DOS/VS COBOL Compiler and Library. All of the messages listed are also issued on SYSLST.

The following messages are issued during compilation on SYSLOG. They are also printed on SYSLST with the prefix ILA.

C100I PARTITION IS LESS THAN 60K

Explanation: At least 60K is required to compile using DOS/VS COBOL. Probable user error.

System Action: The compilation is terminated.

Programmer Response: Not applicable.

Operator Response: Use the ALLOC command to allocate at least 60K to the partition. If the problem recurs, do the following to complete your problem determination action before calling IBM for programming support:

1. Execute the MAP command and save the output.
2. Have the source deck, control cards, output listing, and console sheet available.

C101I DEVICE NOT ASSIGNED - SYSnnn.

Explanation: (nnn is either 001, 002, 003, or 004.) The specified logical unit is unassigned and must be assigned. Probable user error.

System Action: The compilation is terminated.

Programmer Response: Not applicable.

Operator Response: Use the ASSGN command to assign a physical unit (magnetic tape or disk) to the file indicated. If the problem recurs, do the following to complete your problem determination action before calling IEM for programming support:

1. Execute the LISTIO command and save the output.
2. Have the source deck, control cards, output listing, and console sheet available.

C102I UNSUPPORTED DEVICE TYPE - SYSnnn.

Explanation: (nnn is either 001, 002, 003, or 004.) The specified file must be a tape or disk file for SYS002 through SYS004. SYS001 should be assigned to disk; however, in small, simple programs that do not require dictionary spill, it is sometimes possible to compile with the spill file (SYS001) assigned to tape. If any spill does occur, an input/output error may occur. Compile-time statistics will say "DICTIONARY SPILL HAS OCCURRED". No mention is made of dictionary spill in the compile-time statistics if spill does not occur. Probable user error.

System Action: The compilation is terminated.

Programmer Response: Not applicable.

Operator Response: Use the ASSGN command to assign the appropriate physical unit to the file indicated -- SYS001 should be assigned to a magnetic tape or disk unit. If the problem recurs, do the following to complete your problem determination action before calling IBM for programming support:

1. Execute the LISTIO command and save the output.
2. Have the source deck, control cards, output listing, and console sheet available.

C103I END OF FILE ON SYSIPT.

Explanation: End-of-file was encountered in the initialization phase; no source statements were found. Probable user error.

System Action: The compilation is terminated.

Programmer Response: Not applicable.

Operator Response: Ensure that a /* card does not precede the source deck, or add the source deck to the job stream. If the problem recurs, do the following to complete your problem determination action before calling IBM for programming support:

1. Execute the LISTIO command and save the output.
2. Have the source deck, control cards, output listing, and console sheet available.

C104I SYS001 FILE NOT ASSIGNED TO DISK

Explanation: In small, simple programs that do not require dictionary spill, it is sometimes possible to compile with the spill file (SYS001) assigned to tape. However, if any spill does occur, an input/output error may occur. Any compilation which spills the dictionary will contain a message in compile-time statistics. User error.

System Action: The compilation continues.

Programmer Action: Not applicable.

Operator Response: Use the ASSGN command to assign SYS001 to a disk unit. If the problem recurs, do the following to complete your problem determination action before calling IBM for programming support:

1. Execute the LISTIO command and save the output.
2. Have the source deck, control cards, output listing, and console sheet available.

C105I W-CANNOT OPEN SYS005 -- SYMDMP IGNORED.

Explanation: The SYMDMP option has been specified, but the file needed for symbolic debug cannot be opened since SYS005 is unassigned. Probable user error.

System Action: The SYMDMP option is canceled, and the compilation continues.

Programmer Response: Not applicable.

Operator Response: Use the ASSGN command to assign SYS005 to a physical unit. If the problem recurs, do the following to complete your problem determination before calling IBM for programming support:

1. Execute the LISTIO command and save the output.
2. Have the source deck, control cards, output listing, and console sheet available.

C106I SYS006 IS NOT A DISK. NOLVL ASSUMED.

Explanation: The specified logical unit is not assigned to a disk.

System Action: Compilation continues with NOLVL.

Programmer Response: Not applicable.

Operator Response: Use the ASSGN command to assign SYS006 to a disk unit.

OBJECT-TIME MESSAGES

The following messages are normally issued on SYSLOG.

C110A STOP literal

Explanation: The programmer has issued a STOP literal statement in the COBOL source program.

System Action: Awaits operator response.

Programmer Response: Not applicable.

Operator Response: Operator should respond with end-of-block, or with any character in order to proceed with the program.

C111A AWAITING REPLY

Explanation: This message is issued in connection with the Full American National Standard COBOL ACCEPT statement.

System Action: Awaits operator response.

Programmer Response: Provide the operator with instructions.

Operator Response: The operator should reply as specified by the programmer.

The following messages are issued on SYSLOG and SYSLST prior to cancellation of the job. If the DUMP option is specified, a partial dump is taken from the problem program origin to the highest storage location of the last phase loaded. When this occurs, the eight bytes immediately preceding the DTF are destroyed. The messages have the form:

CmmmI SYSnnn filename DTFaddress text

where:

<u>mmm</u>	<u>text</u>
112	DATA CHECK
113	WRONG LENGTH RECORD
114	PRIME DATA AREA FULL
115	CYLINDER INDEX TOO SMALL
116	MASTER INDEX TOO SMALL
117	OVERFLOW AREA FULL
118	DATA CHECK IN COUNT
119	DATA CHECK IN KEY OR DATA
120	NO ROOM FOUND
121	DASD ERROR
122	DASD ERROR WHILE ATTEMPTING TO WRITE RECORD ZERO
123	FILE CANNOT BE OPENED AFTER CLOSE WITH LOCK
124	CYLINDER AND MASTER INDEX TOO SMALL

nnn is equal to 001 through 255

filename is seven or less characters and is generated from the file-name specified in the SELECT sentence.

address is the hexadecimal address of the file's DTF table.

mmm and text correspond as follows:

Explanation: Condition indicated occurred on SYSnnn.

System Action: The job is cancelled.

Programmer Response: Rerun the job or add a user Declarative Section to the Procedure Division of the source program to handle errors within the program.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, compiler output, and console sheet available.

Operator Response: Not applicable.

C125I NO EXTENTS

Explanation: During CLOSE UNIT processing, no extent is found for the next volume.

System Response: The job is cancelled.

Programmer Response: Rerun job with proper EXTENT (XTENT) statements.

Operator Response: Not applicable.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, compiler output, and console sheet available.

The following message is issued on SYSLOG:

C126D SYSnnn IS IT EOF?

Where nnn is equal to 001 through 255

Explanation: A tapemark was just read on an unlabeled tape file described at compilation time as having more than one reel.

System Action: Awaits response from operator.

Programmer Response: Not applicable.

Operator Response: The operator must respond either with N if is end of volume, or with Y if it is end of file.

The following messages are issued on SYSLOG and SYSLIST:

C127D NO EOF RECORD WRITTEN IN PRIME DATA AREA

Explanation: During CLOSE processing of an ISAM file opened OUTPUT, no room was found to write EOF record.

Programmer Response: Rerun the job with the proper EXTENT.

If the problem recurs, do the following to complete your problem determination action before calling IBM for programming support. Have source deck, control cards, compiler output, and console sheet available.

Operator Response: Not applicable.

C128D UNRECOVERABLE I/O ERROR

Explanation: This is probably a hardware error on tape.

Programmer Response: Not applicable.

Operator Response: Rerun the job.

If the problem recurs, do the following to complete your problem determination action before calling IBM for programming support. Have source deck, control cards, compiler output, and console sheet available.

C129I VSAM SUBROUTINE ERROR. CANCELING JOB.

Explanation: The subroutine has encountered an unrecoverable error.

Action: The program is canceled with a dump.

Programmer Response: Submit an APAR with the dump.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C130I INPUT/OUTPUT ERROR. FILE STATUS SET TO xx NEAR REL LOC.
xxxxxx.

Explanation: An I/O error has occurred on the file being accessed by the COBOL statement at or near the relative location given in the message, and the user has no USE declarative for that file.

Action: Control returns to COBOL at the statement following the COBOL request that caused the error.

Programmer Response: If the error occurred on a READ operation, processing can continue. If the error occurred on a WRITE operation, there may be a loss of data.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C131I UNABLE TO OPEN FILE 'SYSnnn'. CANCELING.

Explanation: The VSAM OPEN or CLOSE request gave a return code of X'68' or X'6C' because of invalid time stamps in the VSAM catalog or VSAM file. The VSAM catalog or file should be recreated. See DOS/VS Supervisor and I/O Macros for more detail on the OPEN/CLOSE return codes.

Action: The job is canceled.

Programmer Response: Recreate the VSAM catalog and/or file.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C140I INVALID SEPARATE SIGN CONFIGURATION

Explanation: During execution of a COBOL program, an invalid sign was detected for a separately signed item.

Action: The job is terminated.

Programmer Response: Probable user error. Correct program's input data before reexecuting.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, compiler output, and data available.

The following messages (C150I-C170I) are listed on SYSLST. The messages have the form:

CmmmI { program-id } text
 { card/verb number }

Messages C150I through C162I may appear interspersed among the SYMDMP control cards at the point at which the error is encountered. Program-id is provided for all messages except C150I through C152I. For these, the card/verb number of the corresponding line-control card is given instead. The program-id associated with C150I through C152I can be determined from the nearest preceding program-control card.

Messages C153I through C155I may also appear in the midst of the dump output if the error condition is not recognized until dumping has started.

C150I IDENTIFIER NOT FOUND.

Explanation: An identifier specified on the line-control card cannot be found in the program or is invalid. Level-66 and

level-88 items and items defined under an RD are invalid requests.

Action: The dump request for this identifier is ignored.

Programmer Response: Probable user error. Before reexecuting, ensure that no requests have been made on the line-control card for the dumping of identifiers that have not been defined or that are invalid.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C151I CARD NUMBER NOT FOUND

Explanation: The card number specified on the line-control card is not within range of the Procedure Division.

Action: The line-control card which specifies the nonexistent card number is skipped.

Programmer Response: Probable user error. Ensure that any card number specified on a line-control card is within range of numbers specified for source program before reexecuting.

If the problem recurs. do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C152I VERB NUMBER NOT FOUND

Explanation: The verb number specified on a line-control card does not exist on the card specified.

Action: The nearest verb number on the card specified is used.

Programmer Response: Probable user error. Correct verb number specification before reexecuting.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C153I NO ROOM TO DUMP.

Explanation: If this message immediately follows a program-control card, sufficient storage is not available for the debug subroutine or for the 72 bytes of data required for each program in the run unit. If this message follows an abnormal termination message, one or more of the following is not available in free storage or in the COBOL Procedure Division: a contiguous block of 4000 bytes, a contiguous block of 1800 bytes, or a contiguous block of 512 bytes.

Action: No Data Division dump for the indicated program and, in some instances, no statement number information, is provided.

Programmer Response: Probable user error. Increase the size of the partition before reexecuting. See "System Configuration" for information about storage requirements for symbolic debugging.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C154I I/O ERROR ON DEBUG FILE.

Explanation: An input/output error has occurred on the debug file. Note that such an error may be the result of a file other than the debug file being mounted on the logical unit specified.

Action: SYMDMP output is cancelled for the indicated program.

Response: Hardware, operator, or user JCL error. Before reexecuting, check logical unit number specified on program-control card against current mounting, as well as the ASSGN, DLBL, and EXTENT cards of compilation.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C155I WRONG DEBUG FILE FOR PROGRAM.

Explanation: The file corresponding to the filename and/or logical unit number provided on the program-control card is not the debug file created for this program at compile time.

Action: SYMDMP output is cancelled for the indicated program.

Programmer Response: Probable user error. Before reexecuting, ensure that the filename and/or logical unit specified on the program-control card corresponds to that of the debug file created at compile time.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C156I NO ROOM FOR DYNAMIC DUMP.

Explanation: Sufficient storage is not available to store the line-control card information during execution.

Action: Dynamic dumping is cancelled for the indicated program.

Programmer Response: Probable user error. Increase size of partition or decrease number of line-control cards before reexecuting.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C157I INVALID FILENAME.

Explanation: If the "filename" parameter is specified for a disk file on the CBL card at compile time, the same "filename" must also be specified on the program-control card. "Filename" may be from one to seven characters in length; the first character must be a letter.

Action: All SYMDMP output is cancelled for the indicated program.

Programmer Response: Probable user error. Correct "filename" specification on the program-control card before reexecuting.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C158I INVALID LOGICAL UNIT.

Explanation: The logical unit parameter on the program-control card must be specified, must be an integer between 0 and 244, and must match the one specified in the ASSGN control statement for the debug at compile time.

Action: All SYMDMP output is cancelled for the indicated program.

Programmer Response: Probable user error. Correct logical unit specification on program-control card before reexecuting.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C159I MISSING PARAMETERS.

Explanation: A non-continued line-control card ends with (HEX), OF, IN, or THRU. Possibly a continuation punch is missing in column 72.

Action: A HEX or THRU option ending a card is ignored. When a card ends with OF or IN, the word is ignored and the identifier that is dumped is the first one encountered whose qualifiers match those preceding the word OF or IN.

Programmer Response: Probable user error. Check line-control card for keypunch errors before reexecuting.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C160I INVALID OPTION.

Explanation: An element used as an optional parameter on a program-control card is not one of the legal program-control card options.

Action: The element is ignored.

Programmer Response: Probable user error. Correct syntax of program-control card before reexecuting.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C161I SUBSCRIPTING ILLEGAL.

Explanation: The "name" parameter of the line-control card may not be subscripted.

Action: The subscripts are ignored. Every occurrence of the identifier is dumped.

Programmer Response: Probable user error. Specify the name of the item without the subscript before reexecuting. This will result in a dump of every occurrence of the item.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C162I ON PARAMETER TOO BIG.

Explanation: Neither the n, m, nor k parameter of the ON option may exceed 32767.

Action: The number is reduced to 32767.

Programmer Response: Probable user error. Correct invalid parameter before reexecuting.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C163I FLOW TRACE NON-CONTIGUOUS. MORE THAN 10 PROGRAMS ENCOUNTERED

Explanation: A non-contiguous flow trace will result if FLOW option is effective in a subprogram structure of more than 10 programs compiled with the FLOW option.

Action: The FLOW trace is terminated upon encountering the eleventh PROGRAM-ID. Tracing resumes only upon returning to one of the original ten programs.

Programmer Response: Probable user error. If trace is absent for a program where it is critical, recompile one or more of the programs where the flow is non-critical without the FLOW option and reexecute.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C164I FLOW TRACE IN EFFECT BUT NO PROCEDURES TRACED.

Explanation: Abnormal termination has taken place before any COBOL statement with a procedure-name has been traced.

Action: No tracing is done.

Programmer Response: Probable user error. If trace is desired, recompile the program after inserting additional procedure-names.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C165I SYMDMP/STATE/FLOW/COUNT INTERNAL ERROR. EXECUTION CANCELLED.

Explanation: Abnormal termination occurred during execution of one of the debugging subroutines.

Action: The job is cancelled.

Programmer Response: Internal logic error.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C169I STATE OPTION CANCELLED.

Explanation: compiler or logic error has occurred during STATE option processing. Under certain conditions, this error may result from other user errors. For example, a loop might destroy some of the information required by the STATE subroutines; an invalid branch might cause a non-existent priority-number to be stored in the TGT, etc.

Action: STATE output is cancelled.

Programmer Response: Probable user error. Possible compiler error or user error. Correct other known errors (if any) before attempting reexecution.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C170I INVALID ADDRESS.

Explanation: The address calculated for a subscripted identifier, or a starting or ending address of a variable-length identifier used as the receiving field in a MOVE statement is invalid.

Action: A symbolic dump is produced.

Programmer Response: Probable user error. Possible compiler error or user error. Correct other known errors (if any) before attempting reexecution.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C171I SPACE NOT FOUND FOR THE COUNT CHAIN. CONTINUING.

Explanation: A GETVIS macro was unsuccessful due to lack of space.

Action: Execution continues. Execution statistics are not provided for the last indicated program unit.

Programmer Response: Probable user error. Allocate more space on EXEC card before attempting reexecution.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C172I SPACE NOT FOUND FOR THE VERBSUM TABLE. CONTINUING.

Explanation: A GETVIS macro was unsuccessful due to lack of space.

Action: Execution continues. Verb summary statistics are not provided for the program.

Programmer Response: Probable user error. Allocate more space on EXEC card before attempting reexecution.

C173I FREEVIS FAILED. EXECUTION CANCELLED.

Explanation: A FREEVIS macro was unsuccessful.

Action: Execution is terminated.

Programmer Response: Probable user error. Allocate more space on EXEC card before attempting reexecution.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

C175I INVALID COUNT TABLE ENTRY. EXECUTION CANCELLED.

Explanation: A count table entry in the object module is not one of the following: end-of-table indicator, procedure-id, or verb-id.

Action: Execution is terminated.

Programmer Response: Probable user error. Possible compiler or user error. Check your program for routines that may have moved data into the count table area. Correct other known errors (if any) before attempting execution.

If the problem recurs, do the following before calling IBM for programming support: have source deck, control cards, and compiler output available.

COBOL OBJECT PROGRAM UNNUMBERED MESSAGES

xxx...

Explanation: This message is written on the console and is recognizable because it is not preceded by a message code and action indicator. It is issued by an object program originally coded in COBOL. The message text is supplied by the object program and may indicate alternative action to be taken.

System Action: The job continues.

Operator Response: Operator response, if any is needed, is determined by the message text.

APPENDIX J: COBOL 3886 OPTICAL CHARACTER READER SUPPORT

This appendix contains information on the 3886 Optical Character Reader, Model 1* (denoted as "the OCR"). Topics discussed include:

- 3886 OCR processing
- COBOL considerations for 3886 OCR processing.
- Status key values
- Sample program

This discussion assumes familiarity with these IBM 3886 Optical Character Reader publications:

IBM 3886 OCR General Information Manual, Order No. GA21-9146 -- for terminology, device capabilities, and the formats of the header and data records.

IBM 3886 OCR Input Document Design and Specifications, Order No. GA21-9148 -- for document design considerations and detailed specifications.

In addition, the applicable portions of the following manuals should be referenced:

IBM DOS/VS Supervisor and I/O Macros, Order No. GC33-5373 -- for describing documents using the DFR and DLINT macros.

IBM DOS/VS System Generation, Order No. GC33-5377

IBM DOS/VS Data Management Guide, Order No. GC24-5062.

IBM DOS/VS Program Planning Guide for the IBM 3886 Optical Character Reader, Model 1, Order No. GC21-5059

3886 OCR PROCESSING

The 3886 OCR, Model 1 is a general purpose online device that satisfies a broad range of data entry requirements. The OCR accepts documents sized from 3 inches by 3 inches to 9 inches by 12 inches. It can read machine-printed

*This device should not be confused with the 3886, Model 2, an offline Optical Character Reader with output to tape. Information is included in this chapter on processing the tapes produced by the Model 2.

alphabetic characters, numeric characters, and certain special characters in a wide variety of fonts, as well as hand-printed numeric characters.

The OCR reads documents one line at a time, under program control. Additional features, all under program control, include:

- document marking
- line marking
- document eject (with stacker selection)
- line reread (for the current line, and with a different format if desired)

Note: The OCR cannot read previous lines; reading can proceed from top to bottom on the document only.

IMPLEMENTING AN OCR OPERATION

Document Design

The OCR form that will be used for input should be prepared independently of the COBOL program. Document design criteria are described in detail in IBM 3886 Optical Character Reader, Input Document Design Guide and Specifications.

The most important aspects of document design are:

1. The locations of lines which can be read. These lines are identified by "timing marks." Lines not associated with timing marks are always ignored by the OCR. Note that lines may be almost anywhere on the document, and need not be at regular intervals.
2. The location of fields to be read. Fields, (strings of related characters) should be identified in document design. They should be described using the DFR and DLINT macros. (See section entitled "Document Description".)
3. The form identifier. This field should be a pre-printed code at a common location on the first readable line of each format. This field can be ignored by programming or DLINIT

specification if desired; it should, however, be included in the form design so as to allow for later form changes or intermixing of forms in batches without disruption of operations.

Document Description

Documents are described in the system with the Define Format Record (DFR) and Define Line Type (DLINT) macros. These macros should be coded independently of the COBOL program.

The DFR macro identifies, by name, a collection of DLINT macros, and establishes various default field scanning options for them. Each different DFR grouping identifies a different document, or a largely different way of scanning the same document (for example, a document in a different font).

DFR and DLINT macros, after assembly and linkage editing, are preserved in loadable form until called for by the application program.

Each DLINT macro describes the scanning of a line, by field, in terms of

1. The starting and ending points of fields on a line (in tenths of an inch).
2. The field lengths (in characters).
3. The font code to be used (OCR-A, OCR-B, Gothic, or hand-printed numerics, all with various additional options).
4. Field editing (blank fill, blank suppression, zero fill, left or right justification, special character suppression).
5. Field character delimiters (a character to end a field scan).

Note that the DLINT macro may specify either standard mode or image mode. In standard mode, all DLINT options are valid, and the data record is of a fixed format, according to the field lengths in characters. In image mode, the field length and all EDIT keywords are invalid. The data record begins with 14 parameters, each two bytes long, indicating the length of the fields that follow. Because of this variable format in the data record, it is recommended that image mode be used only in applications for which standard mode is unsuitable.

COBOL Support

COBOL supports the OCR with a subprogram (invoked by CALL statements), Data Division COPY statement library material (to fully describe the parameter area required by the subprogram), and Procedure Division COPY statement library material (to provide procedures that simplify invocation of the subprogram).

File Description

The file is described by the Data Division COPY statement member. (See sample program for format.) All fields and codes are included, with descriptive names and default values. The programmer need only modify those fields that are not appropriate for the application.

The file description ("OCR-FILE" in the COPY statement member) includes all fields that the programmer must provide to the subprogram, the OCR-STATUS-KEY returned by the subprogram, and fields that describe the header and data records returned by the device. Note that the file is described through data records rather than the usual COBOL PD.

Note: The header and data records are not constructed under program control and are not altered after reading. Their contents are fully described in IBM 3886 Optical Character Reader General Information Manual.

Record Description

The COBOL record descriptions are based on the DLINT formats, either in image mode or in standard mode.

If standard mode scanning is specified, the data record is returned in a fixed format according to the DLINT macro; that is, contiguous fields, from left to right, in the same order as in the DLINT macro, each with a specified length in bytes. If image mode scanning is specified, however, the field lengths are returned at the beginning of the data record.

The programmer may describe the data records to be read by the application program by following the Data Division COPY statement request with statement(s) of the form:

05 dataname REDEFINES OCR-DATA-RECORD

The structure of each record description should follow each such statement starting with a level number greater than 5. (See sample program for example.)

Procedural Code

The COBOL source statements control the file, read lines, and recover from errors. The subprogram CALL statement requirements are described in the Procedure Division COPY statement member. This member provides paragraphs which the COBOL programmer can PERFORM to set the proper operation code, CALL the subprogram, and pass control to a programmer-supplied exception routine if an exception occurs. The programmer should COPY these paragraphs into his program.

The programmer must move parameter information to the file area (OCR-FILE-CONTROL-AREA), and then issue a PERFORM statement for the appropriate procedure.

If an exception occurs, the COPY statement member passes control to the procedure-name OCR-EXCEPTION-ROUTINE. If operations are to be retried in this routine, the programmer should issue the appropriate CALL (not PERFORM) statement and test the OCR-STATUS-KEY value afterwards.

Return from the OCR-EXCEPTION-ROUTINE would normally be to OCR-CALL-EXIT (after a successful retry or recovery). Control is then returned to the invoking PERFORM statement.

JCL Considerations

Programs using the IBM-supplied 3886 processing subroutines must have a SIZE parameter specified on the EXEC card and cannot run in REAL mode. The user must specify the SIZE parameter equal to the size of his problem program to free the remainder of his partition for use as the page pool. Each opened 3886 file requires at least 2K bytes of the page pool.

Subprogram Interface

The IBM-supplied COPY members provide a data area ('OCR-FILE') and CALL statements using this area for parameter interface to the OCR subprogram. The data area has the following format:

- ```

01 OCR-FILE.
 05 OCR-FILE-CONTROL-AREA
 10 OCR-FILE-ID PIC X(8) VALUE
 'SYSnnn'.
 (Unique file name; also, must
 agree with JCL ASSGN
 statement)

 10 OCR-FORMAT-RECORD-ID PIC X(8)
 VALUE "xxxxxxxx".
 (DFR phase name, used for
 'OPEN' or 'SETDV')

 10 OCR-OPERATION PIC X(5).
 ("OPEN", "CLOSE", "READ",
 "READO", "WAIT", "SETDV",
 "MARKL", "MARKD", or "EJECT"
 (left justified)).

 10 OCR-STATUS-KEY PIC 99.
 (also referred to as exception
 code.)

 10 OCR-LINE
 15 OCR-LINE-NUMBER PIC 99.
 (Line number (0-33) passed
 to "MARKL", "READ", or
 "EJECT")

 15 OCR-LINE-FORMAT PIC 99.
 (Line format number (0-63)
 passed to "READ")

 10 OCR-MARK PIC 99.
 (Mark option (1-15) passed to
 "MARKL" or "MARKD".)

 10 OCR-STACKER PIC 9.
 (Pocket number (1-2) passed to
 "EJECT".)

 05 OCR-HEADER-RECORD PIC X(20).
 (Header information returned from
 "READ" or "WAIT".)

 05 OCR-DATA-RECORD PIC X(130).
 (Data record returned from "READ"
 or "WAIT".)

```

(For descriptions of these operations, see the section "Statements for Invoking 3886 I/O Functions".)

**Note:** If the CALL statement does not have one, and only one, parameter following the USING option, the subprogram will return control immediately to the user (with a value of 8 in register 15, . No error indication will be available through COBOL.

Table 37 contains OCR status key values and their meanings. Table 38 is a guide to which operations cause status key values 00 through 99. Table 39 supplies the user responses to status key values.

Table 37. OCR Status Key Values and User Actions

| Status Key Code | Meaning                                                                                                                                                                                                                                                                                                                                                                     |
|-----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 00              | Successful completion                                                                                                                                                                                                                                                                                                                                                       |
| 10              | End-of-file <sup>1</sup>                                                                                                                                                                                                                                                                                                                                                    |
| 3x              | I/O error or related error where: <sup>2</sup><br>x = 1 -- Mark Check<br>= 2 -- Nonrecovery<br>= 3 -- Incomplete Scan<br>= 4 -- Mark Check and Equipment Check<br>= 9 -- Permanent Error                                                                                                                                                                                    |
| 9y              | Other error where:<br>y = 2 -- logic error, that is, file not open (except OPEN), file already open (for OPEN), WAIT issued, but no READO pending, WAIT not issued for pending READO.<br>= 3 -- insufficient storage available (OPEN) or failure in storage release (CLOSE)<br>= 5 -- invalid parameter (other than operation code)<br>= 9 -- unrecognizable operation code |

<sup>1</sup>The end-of-file condition is raised after the listed I/O commands if:  
-- the operator has pressed the END-OF-FILE button, and  
-- no documents remain in the read station, and  
-- no errors are outstanding

If // ASSGN SYSxxx,IGN has been specified, EOF is given only on READ and WAIT commands. While the end-of-file condition is active, commands (other than CLOSE) are only checked for validity.

<sup>2</sup>If any I-O errors, or certain system errors occur during the OPEN operation, the job is canceled by the system.

Table 38. Possible Status Key Values, By Operation

| OCR-OPERATION Value | OPEN | CLOSE | READ | READO | WAIT | MARKL | MARKD | EJECT | SETDV | other |
|---------------------|------|-------|------|-------|------|-------|-------|-------|-------|-------|
| 00                  | X    | X     | X    | X     | X    | X     | X     | X     | X     |       |
| 10                  |      |       | X    |       | X    | X     | X     | X     | X     |       |
| 31                  |      |       |      |       |      |       |       | X     |       |       |
| 32                  |      |       | X    | X     | X    | X     | X     | X     | X     |       |
| 33                  |      |       | X    |       | X    |       |       |       |       |       |
| 34                  |      |       |      |       |      |       |       | X     |       |       |
| 39                  |      |       | X    | X     | X    | X     | X     | X     | X     |       |
| 92                  | X    | X     | X    | X     | X    | X     | X     | X     | X     |       |
| 93                  | X    |       |      |       |      |       |       |       |       |       |
| 95                  | X    |       | X    | X     |      | X     | X     | X     |       |       |
| 99                  |      |       |      |       |      |       |       |       |       | X     |

Table 39. User Responses to Status Key

| Status Key | Meaning                        | Response                                                                                                                                                                                                                                  |
|------------|--------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 00         | Successful (no EOF)            | The operation has completed properly.                                                                                                                                                                                                     |
| 10         | End-of-file                    | Do EOF processing and close the file, or have operator ready 3886 and continue processing. See Note 1.                                                                                                                                    |
| 31         | Mark Check                     | Attempt to reread the line, or eject document and prepare to process next document.                                                                                                                                                       |
| 32         | Nonrecovery Error              | Eject document and prepare to process next document.                                                                                                                                                                                      |
| 33         | Incomplete Scan                | Reread the line using a different DLINT, or using an image-mode DPR.                                                                                                                                                                      |
| 34         | Mark Check and Equipment Check | See Note 2.                                                                                                                                                                                                                               |
| 39         | Permanent Error                | See Note 2. One of the following has occurred: Command Reject, Bus Out Check, Equipment Check, Non-Initialized, RCP error, or Invalid Format.                                                                                             |
| 92         | Logic error                    | See Note 3. One of the following operation order errors has occurred:<br>-- OPEN issued on file already open<br>-- file not open (all operations except OPEN)<br>-- WAIT issued but no READO in progress<br>-- READO not followed by WAIT |
| 93         | Insufficient storage           | See Note 3. The GETVIS issued by the COBOL subroutine has failed. Check that the SIZE parameter is large enough.                                                                                                                          |
| 95         | Invalid parameter              | See Note 3. A parameter required by the last operation was invalid (too large, too small, or contained invalid characters).                                                                                                               |
| 99         | Unrecognizable operation       | See Note 3. The OCR-OPERATION parameter contained an illegal operation code.                                                                                                                                                              |

Notes:

1. Serious I-O error conditions exist. No more I/O should be performed on the device after any of these errors are encountered. The program should indicate the error, perform error recovery, and issue a STOP RUN.
2. A serious programming error has occurred, or there is a problem in the program environment. The program should indicate the error, perform clean-up, and issue a STOP RUN.
3. WAIT and READ commands return data and header records only for the following codes: 00, 10, 31, and 33. For other codes, the contents of the header and data record areas are unpredictable.

STATEMENTS FOR INVOKING 3886 I/O FUNCTIONS

OPEN Function (Equivalent to OPEN Macro)

OPEN makes a logical file available to your program and loads the appropriate format record into the 3886. The statement format for OPEN is:

PERFORM OCR-OPEN

The subprogram requires these fields:  
OCR-FILE-ID, OCR-OPERATION ('OPEN'),  
OCR-FORMAT-RECORD-ID

The subprogram will return: OCR-STATUS-KEY

CLOSE Function (Equivalent to DOS CLOSE Macro)

CLOSE deactivates any 3886 files used by your program. These files must be closed before the program can be terminated. The statement format for CLOSE is:

PERFORM OCR-CLOSE

The subprogram requires these fields:  
OCR-FILE-ID, OCR-OPERATION ('CLOSE')

The subprogram will return: OCR-STATUS-KEY

READ Function (Equivalent to DOS READ and WAITF Macros)

READ allows one line of data to be read from the document. The statement format for READ is:

PERFORM OCR-READ

The subprogram requires these fields:  
OCR-FILE-ID, OCR-OPERATION ('READ'),  
OCR-LINE-NUMBER, OCR-LINE-FORMAT

The subprogram will return:  
OCR-STATUS-KEY, OCR-HEADER-RECORD,  
OCR-DATA-RECORD

Note: The READ function combines the functions of READO and WAIT. I/O overlap is not allowed within the issuing task.

READO Function (Equivalent to DOS READ Macro)

READO (read overlapped) initiates the reading of one line of data from the document. WAIT must subsequently be issued to complete the request. The statement format for READO is:

PERFORM-OCR-READ-OVERLAPPED

The subprogram requires these fields:  
OCR-FILE-ID, OCR-OPERATION ('READO'),  
OCR-LINE-NUMBER, OCR-LINE-FORMAT

The subprogram will return: OCR-STATUS-KEY

Note: A successful READO function must be followed by a WAIT request for that same OCR-FILE area. No intervening I/O operations for that file are allowed.

WAIT Function (Equivalent to DOS WAITF Macro)

WAIT completes the action of the preceding READ. The statement format for WAIT is:

PERFORM OCR-READ

The subprogram requires these fields:  
OCR-FILE-ID, OCR-OPERATION ('WAIT'),

The subprogram will return:  
OCR-STATUS-KEY, OCR-HEADER-RECORD,  
OCR-DATA-RECORD

The WAIT function causes the active task to be placed in the WAIT condition, if necessary, until the preceding READO operation is completed. It must be issued only after a successful READO, with no intervening commands for that file.

MARKL Function (Equivalent to DOS CNTRL Macro with LMK Option)

MARKL is used to mark a line on the document. The statement format for MARKL is:

PERFORM OCR-MARK-LINE

The subprogram requires these fields:  
OCR-FILE-ID, OCR-OPERATION ('MARKL'),  
OCR-LINE-NUMBER, OCR-MARK

The subprogram will return: OCR-STATUS-KEY

MARKD Function (Equivalent to DOS CNTRL Macro with DMK Option)

MARKD is used to mark the document (in the Page Mark location).

The statement format for MARKD is:

PERFORM OCR-MARK-DOCUMENT

The subprogram requires these fields:  
OCR-FILE-ID, OCR-OPERATION ('MARKD'),  
OCR-MARK

The subprogram will return: OCR-STATUS-KEY

EJECT Function (Equivalent to DOS CNTRL Macro, with ESP Option)

EJECT is used to eject the document into a specified stacker, with optional validation of its total number of timing marks. The statement format for EJECT is:

PERFORM OCR-EJECT

The subprogram requires these fields:  
OCR-FILE-ID, OCR-OPERATION ('EJECT'),  
OCR-STACKER, OCR-LINE-NUMBER

The subprogram will return: OCR-STATUS-KEY

SETDV (Set Device by Loading a Format Record) Function (Equivalent to DOS SETDEV Macro)

SETDV allows format records to be changed during execution of the program. The statement format for SETDV is:

PERFORM OCR-SET-DEVICE

The subprogram requires these fields:  
OCR-FILE-ID, OCR-OPERATION ('SETDV'),  
OCR-FORMAT-RECORD-ID

The subprogram will return: OCR-STATUS-KEY

COBOL 3886 Library Routine

The COBOL 3886 library routine is invoked in response to the CALL statement. For the proper execution of this routine GETTVIS=YES must be specified at system generation. An illegal SVC results if GETTVIS=NO is specified.

Table 40 contains a list of CALL statements used for invoking 3886 I/O functions (if the IBM-supplied COPY member is not used).

All OCR CALL statements have the format CALL 'ILBDOCRO' USING OCR-FILE, where OCR-FILE is used as follows:



Table 40. CALL Statements for Invoking 3886 I/O Functions

| Function<br>(OCR-OPERATION) | Set by User                                                        | Subroutine Returns                                     |
|-----------------------------|--------------------------------------------------------------------|--------------------------------------------------------|
| OPEN                        | OCR-FILE-ID<br>OCR-OPERATION<br>OCR-FORMAT-RECORD-ID               | OCR-STATUS-KEY                                         |
| CLOSE                       | OCR-FILE-ID<br>OCR-OPERATION                                       | OCR-STATUS-KEY                                         |
| READ                        | OCR-FILE-ID<br>OCR-OPERATION<br>OCR-LINE-NUMBER<br>OCR-LINE-FORMAT | OCR-STATUS-KEY<br>OCR-HEADER-RECORD<br>OCR-DATA-RECORD |
| READO                       | OCR-FILE-ID<br>OCR-OPERATION<br>OCR-LINE-NUMBER<br>OCR-LINE-FORMAT | OCR-STATUS-KEY                                         |
| WAIT                        | OCR-FILE-ID<br>OCR-OPERATION                                       | OCR-STATUS-KEY<br>OCR-HEADER-RECORD<br>OCR-DATA-RECORD |
| MARKL                       | OCR-FILE-ID<br>OCR-OPERATION<br>OCR-LINE-NUMBER<br>OCR-MARK        | OCR-STATUS-KEY                                         |
| MARKD                       | OCR-FILE-ID<br>OCR-OPERATION<br>OCR-LINE-NUMBER<br>OCR-MARK        | OCR-STATUS-KEY                                         |
| EJECT                       | OCR-FILE-ID<br>OCR-OPERATION<br>OCR-LINE-NUMBER<br>OCR-STACKER     | OCR-STATUS-KEY                                         |
| SETDV                       | OCR-FILE-ID<br>OCR-FORMAT-RECORD-ID<br>OCR-OPERATION               | OCR-STATUS-KEY                                         |

PROCESSING TAPES FROM THE OCR 3886, MODEL 2

Tape records produced from the IBM 3886, Model 2 are almost identical in format to the header and data records returned by the Model 1. The main differences between the records are:

- Model 2 tapes contain a document trailer record after the line output records for each document. The content of this trailer record differs from that of line output records.
- The codes used in certain fields of the header record differ between the two models.

Because of the similarity, however, the Data Division COPY statement member defined for the Model 1 may be tailored to describe the Model 2 tape records. To do this, punch out the COPY statement member, modify it according to the installation requirements, and recatalog it. The COPY statement member may then be included as a data record, under an FD for the input tape file.

Specific information on the formats and contents of the Model 2 tape records is contained in IBM 3886 Optical Character Reader, General Information Manual.

```

CBL LIB
00001 *****91547000
00002 ***** S A M P L E O C R P R O G R A M *****91548000
00003 *****91549000
00004 IDENTIFICATION DIVISION 91550000
00005 PROGRAM-ID. SAMPLE
00006 ***** THIS PROGRAM IS THE COEOL EQUIVALENT OF THE 91551200
00007 * ASSEMBLY LANGUAGE SAMPLE PROGRAM 'DOCLIST', 91551400
00008 * CONTAINED IN THE DOS/VS PROGRAM PLANNING GUIDE 91551600
00009 * FOR THE IBM 3886 OPTICAL CHARACTER READER, MODEL 1 91551800
00010 * (ORDER NO. GC21-5059) 91551900
00011 ENVIRONMENT DIVISION. 91552000
00012 INPUT-CUTPUT SECTION. 91552200
00013 FILE-CONTROL. 91552400
00014 SELECT PRINTER, ASSIGN TO SYS009-UR-1403-S. 91552600
00015 DATA DIVISION. 91553000
00016 FILE SECTION. 91553200
00017 FD PRINTER LABEL RECORDS ARE OMITTED. 91553400
00018 01 PRINT-RECORD. 91553600
00019 05 FILLER PIC X. 91553800
00020 05 PRINT-LINE PIC X(130). 91553900
00021 WORKING-STORAGE SECTION. 91554000
00022 77 PRINT-CONTROL PIC 9 VALUE 1. 91554200
00023 77 MSG-PERMANENT-ERROR PIC X(24) VALUE 91555000
00024 'PERMANENT ERROR OCCURRED'. 91556000
00025 77 MSG-MARK-CHECK PIC X(19) VALUE 91556100
00026 'MARK CHECK OCCURRED'. 91556200
00027 77 MSG-MARK-AND-EQUIP-CHECK PIC X(39) VALUE 91556600
00028 'MARK CHECK AND EQUIPMENT CHECK OCCURRED'. 91556700
00029 77 MSG-INCOMPLETE-SCAN PIC X(24) VALUE 91556800
00030 'INCOMPLETE SCAN OCCURRED'. 91556900
00031 77 MSG-NONRECOVERY-ERROR PIC X(26) VALUE 91557000
00032 'NONRECOVERY ERROR OCCURRED'. 91557200
00033 77 MSG-BAD-DATA PIC X(50) VALUE 91557400
00034 'THE FOLLOWING LINE WAS MISREAD. THE LINE HEADER ='. 91557600
00035 01 MSG-TERMINATION. 91557700
00036 05 FILLER PIC X(44) VALUE 91557800
00037 'TERMINAL ERROR OCCURRED - OCR-STATUS-KEY = '. 91558100
00038 05 MSG-TERM-STATUS-KEY PIC XX. 91558300
00039 01 OCR-FILE COPY ILEDOCRD. 91558400
00040 C ***** ILEDOCRD - OCR DATA DESCRIPTION *****

```

Figure 69. Sample OCR Program (Part 1 of 5)

```

00041 C *****90037000
00042 C ***** OCR 3886 FILE FORMAT *****90047000
00043 C *****90057000
00044 C 01 OCR-FILE. 90067000
00045 C 05 OCR-FILE-CONTROL-AREA. 90069000
00046 C 10 OCR-FILE-ID PIC X(8) VALUE 'SYS010'. 90077000
00047 C 10 OCR-FORMAT-RECORD-ID PIC X(8) VALUE 'FRLGDFR1'. 90087000
00048 C 10 OCR-OPERATION PIC X(5) VALUE 'OPEN'. 90097000
00049 C 88 OCRO-CPEN VALUE 'OPEN'. 90107000
00050 C 88 OCRO-CLOSE VALUE 'CLOSE'. 90117000
00051 C 88 OCRO-READ VALUE 'READ'. 90127000
00052 C 88 OCRO-READ-OVERLAPPED VALUE 'READO'. 90137000
00053 C 88 OCRO-WAIT VALUE 'WAIT'. 90147000
00054 C 88 OCRO-MARK-LINE VALUE 'MARKL'. 90157000
00055 C 88 OCRO-MARK-DOCUMENT VALUE 'MARKD'. 90167000
00056 C 88 OCRO-EJECT VALUE 'EJECT'. 90177000
00057 C 88 OCRO-SETDEV VALUE 'SETDV'. 90187000
00058 C 10 OCR-STATUS-KEY PIC 99 VALUE 0. 90197000
00059 C 88 OCRS-SUCCESSFUL VALUE 00. 90217000
00060 C 88 OCRS-END-OF-FILE VALUE 10. 90227000
00061 C 88 OCRS-IO-ERRORS VALUE 30 THRU 39. 90257000
00062 C 88 OCRS-MISC-ERROR VALUE 30. 90267000
00063 C 88 OCRS-MARK-CHECK VALUE 31. 90277000
00064 C 88 OCRS-NONRECOVERY-ERROR VALUE 32. 90287000
00065 C 88 OCRS-INCOMPLETE-SCAN VALUE 33. 90297000
00066 C 88 OCRS-MARK-AND-EQUIP-CHECK VALUE 34. 90307000
00067 C 88 OCRS-PERMANENT-ERROR VALUE 39. 90317000
00068 C 88 OCRS-SPECIAL-ERRORS VALUE 90 THRU 99. 90317400
00069 C 88 OCRS-LOGIC-ERROR VALUE 92. 90323000
00070 C 88 OCRS-RESOURCE-UNAVAILAELE VALUE 93. 90325000
00071 C 88 OCRS-INVALID-PARAMETER VALUE 95. 90326000
00072 C 88 OCRS-INVALID-OPERATION VALUE 99. 90326200
00073 C 10 OCR-LINE. 90327000
00074 C 15 OCR-LINE-NUMBER PIC 99 VALUE 1. 90337000
00075 C 15 OCR-LINE-FORMAT PIC 99 VALUE 1. 90347000
00076 C 10 OCR-MARK PIC 99 VALUE 0. 90357000
00077 C 10 OCR-STACKER PIC 9 VALUE 1. 90367000
00078 C * 90377000
00079 C * ***** HEADER AND DATA RECORD AREAS ***** 90387000
00080 C * FILLED IN BY SUCCESSFUL 'READ' AND/OR 'WAIT'. 90397000
00081 C * (NOTE - 'READO' DOES NOT ALTER THESE AREAS) 90407000
00082 C * 90417000
00083 C 05 OCR-HEADER-RECORD VALUE ZEROS. 90427000
00084 C 10 OCRH-LINE-NUMBER PIC 99. 90437000
00085 C 10 OCRH-LINE-FORMAT PIC 99. 90447000
00086 C 10 OCRH-LINE-SCAN-COUNT PIC 9. 90457000
00087 C 10 OCRH-LINE-STATUS PIC 9. 90467000
00088 C 88 OCRH-LINE-GOOD VALUE 0. 90477000
00089 C 88 OCRH-LINE-BLANK VALUE 1. 90487000
00090 C 88 OCRH-LINE-GROUP-ERASE VALUE 3. 90497000
00091 C 88 OCRH-LINE-CRITICAL-ERR VALUE 2. 90507000
00092 C 88 OCRH-LINE-NON-CRITICAL-ERR VALUE 4. 90517000
00093 C 88 OCRH-LINE-COMBINED-ERR VALUE 6. 90527000
00094 C 88 OCRH-LINE-INVALID VALUE 7. 90537000
00095 C 88 OCRH-END-OF-PAGE VALUE 5. 90547000
00096 C 10 OCRH-FIELD-INFO. 90557000
00097 C 15 OCRH-FIELD-STATUS PIC 9 OCCURS 14. 90567000
00098 C 88 OCRH-FIELD-GOOD VALUE 0. 90577000
00099 C 88 OCRH-FIELD-REJECT-CHARS VALUE 2. 90587000
00100 C 88 OCRH-FIELD-WRONG-LENGTH VALUE 4. 90597000
00101 C 88 OCRH-FIELD-COMBINED-ERR VALUE 6. 90607000
00102 C 88 OCRH-FIELD-BLANK VALUE 8. 90617000
00103 C 88 OCRH-FIELD-BLANK-SUP VALUE 4. 90627000
00104 C 05 OCR-DATA-RECORD. 90637000
00105 C 10 OCR-STANDARD-MODE-RECORD 90647000
00106 C 15 OCR-STANDARD-FIELD-CHAR PIC X OCCURS 130. 90657000
00107 C 10 OCR-IMAGE-MODE-RECORD 90667000
00108 C REDEFINES OCR-STANDARD-MODE-RECORD. 90677000
00109 C 15 OCR-IMAGE-FIELD-LENGTH PIC 99 OCCURS 14. 90687000
00110 C 15 OCR-IMAGE-FIELD-CHAR PIC X OCCURS 102. 90697000
00111 C ***** END OF 3886 DATA DIVISION COPY MEMBER ***** 90699000
00112 C 05 NOTICE-OF-PAYMENT-DUE REDEFINES OCR-DATA-RECORD. 91561400
00113 C 10 LINE-1. 91561600
00114 C 15 L1-POLICYHOLDER-NAME PIC X(20). 91561800
00115 C 15 FILLER PIC X(15).
00116 C 10 LINE-2 REDEFINES LINE-1. 91561900
00117 C 15 L2-CITY-AND-STATE PIC X(20). 91562200
00118 C 15 L2-POLICY-NUMBER PIC X(8). 91562400
00119 C 15 L2-AMOUNT-DUE PIC 9(4)V99. 91562600
00120 C 15 L2-PAYMENT-VERIFY-CODE PIC 9. 91562700
00121 C 10 LINE-3 REDEFINES LINE-1. 91562800
00122 C 15 L3-AMOUNT-PAID PIC 9(5)V99. 91563100

```

Figure 69. Sample OCR Program (Part 2 of 5)

|       |                                                                  |          |
|-------|------------------------------------------------------------------|----------|
| 00123 | PROCEDURE DIVISION.                                              | 91563400 |
| 00124 | STOP RUN.                                                        | 91563700 |
| 00125 | P10-START.                                                       | 91564000 |
| 00126 | MOVE 'SYS010' TO OCR-FILE-ID.                                    | 91565000 |
| 00127 | MOVE 'FORMAT' TO OCR-FORMAT-RECORD-ID.                           | 91566000 |
| 00128 | PERFORM OCR-OPEN.                                                | 91567000 |
| 00129 | OPEN OUTPUT PRINTER.                                             | 91568000 |
| 00130 | P10-HEAD.                                                        | 91569000 |
| 00131 | MOVE ALL '*' TO PRINT-LINE.                                      | 91570000 |
| 00132 | PERFORM PRINT-ROUTINE.                                           | 91571000 |
| 00133 | MOVE 1 TO OCR-STACKER.                                           | 91572000 |
| 00134 | P10-READ.                                                        | 91573000 |
| 00135 | PERFORM OCR-READ.                                                | 91574000 |
| 00136 | IF OCRS-NONRECOVERY-ERROR, GO TO P10-EOP-ERR.                    | 91574200 |
| 00137 | IF OCRH-LINE-GOOD, GO TO P10-GOOD.                               | 91575000 |
| 00138 | IF OCRH-LINE-BLANK, GO TO P10-GOOD.                              | 91576000 |
| 00139 | IF OCRH-LINE-NON-CRITICAL-ERR, GO TO P10-GOOD.                   | 91579000 |
| 00140 | IF OCRH-END-OF-PAGE, GO TO P10-EOP.                              | 91580000 |
| 00141 | ***** IF OCRH HAS ANY OTHER CODE, CONSIDER THE DATA AS BAD ****  | 91581000 |
| 00142 | P10-BAD.                                                         | 91582000 |
| 00143 | MOVE MSG-BAD-DATA TO PRINT-LINE.                                 | 91583000 |
| 00144 | PERFORM PRINT-ROUTINE.                                           | 91584000 |
| 00145 | MOVE 2 TO OCR-STACKER.                                           | 91584200 |
| 00146 | P10-GOOD.                                                        | 91585000 |
| 00147 | MOVE OCR-DATA-RECORD TO PRINT-LINE.                              | 91585200 |
| 00148 | PERFORM PRINT-ROUTINE.                                           | 91585400 |
| 00149 | MOVE 1 TO PRINT-CONTROL.                                         | 91585600 |
| 00150 | ADD 1 TO OCR-LINE-NUMBER, OCR-LINE-FORMAT.                       | 91585800 |
| 00151 | IF OCRH-LINE-NUMBER IS LESS THAN 3, GO TO P10-READ.              | 91585900 |
| 00152 | P10-EOP.                                                         | 91586200 |
| 00153 | MOVE 3 TO OCR-LINE-NUMBER.                                       | 91586300 |
| 00154 | PERFORM OCR-EJECT.                                               | 91586400 |
| 00155 | P10-EOP-ERR.                                                     | 91586600 |
| 00156 | MOVE 1 TO OCR-LINE-NUMBER, OCR-LINE-FORMAT.                      | 91586700 |
| 00157 | MOVE 3 TO PRINT-CONTROL.                                         | 91586800 |
| 00158 | GO TO P10-HEAD.                                                  | 91587100 |
| 00159 | ***** EXCEPTION PROCESSING ROUTINE *****                         | 91587300 |
| 00160 | OCR-EXCEPTION-ROUTINE.                                           | 91587400 |
| 00161 | IF OCRS-END-OF-FILE, GO TO P20-EOP.                              | 91587600 |
| 00162 | IF OCRS-MARK-CHECK,                                              | 91587700 |
| 00163 | MOVE MSG-MARK-CHECK TO PRINT-LINE,                               | 91587800 |
| 00164 | GO TO P20-RETURN.                                                | 91587900 |
| 00165 | IF OCRS-NONRECOVERY-ERROR,                                       | 91588100 |
| 00166 | MOVE MSG-NONRECOVERY-ERROR TO PRINT-LINE,                        | 91588500 |
| 00167 | GO TO P20-RETURN.                                                | 91588700 |
| 00168 | IF OCRS-INCOMPLETE-SCAN,                                         | 91588900 |
| 00169 | MOVE MSG-INCOMPLETE-SCAN TO PRINT-LINE,                          | 91589100 |
| 00170 | GO TO P20-RETURN.                                                | 91589300 |
| 00171 | IF OCRS-MARK-AND-EQUIP-CHECK,                                    | 91589500 |
| 00172 | MOVE MSG-MARK-AND-EQUIP-CHECK TO OCR-LINE,                       | 91589700 |
| 00173 | GO TO P20-PRINT-EOF.                                             | 91590000 |
| 00174 | IF OCRS-PERMANENT-ERROR,                                         | 91591000 |
| 00175 | MOVE MSG-PERMANENT-ERROR TO PRINT-LINE,                          | 91592000 |
| 00176 | GO TO P20-PRINT-EOF.                                             | 91593000 |
| 00177 | ***** IF NONE OF THE ABOVE ERRORS, GIVE TERMINATION MESSAGE **** | 91594000 |
| 00178 | MOVE OCR-STATUS-KEY TO MSG-TERM-STATUS-KEY.                      | 91595000 |
| 00179 | MOVE MSG-TERMINATION TO PRINT-LINE.                              | 91596000 |
| 00180 | GO TO P20-PRINT-EOF.                                             | 91597000 |
| 00181 | P20-RETURN.                                                      | 91598000 |
| 00182 | PERFORM PRINT-ROUTINE.                                           | 91598200 |
| 00183 | GO TO OCR-CALL-EXIT.                                             | 91598400 |
| 00184 | P20-PRINT-EOF.                                                   | 91600000 |
| 00185 | PERFORM PRINT-ROUTINE.                                           | 91601000 |
| 00186 | P20-EOF.                                                         | 91602000 |
| 00187 | PERFORM OCR-CLOSE.                                               | 91603000 |
| 00188 | CLOSE PRINTER.                                                   | 91604000 |
| 00189 | STOP RUN.                                                        | 91605000 |
| 00190 | PRINT-ROUTINE.                                                   | 91607000 |
| 00191 | WRITE PRINT-RECORD AFTER ADVANCING PRINT-CONTROL.                | 91609000 |
| 00192 | OCR-COPIED-PROCEDURES. COPY ILBLOCRP.                            | 91610000 |

Figure 69. Sample OCR Program (Part 3 of 5)

```

00193 C ***** ILBDOCRP - OCR 3886 PROCEDURES
00194 C *****
00195 C ***** O C R 3 8 8 6 P R O C E D U R E S *****90757000
00196 C *****90767000
00197 C *****90777000
00198 C * THE 3886 OCR SUBROUTINE USES OCR-FILE FIELDS AS FOLLOWS 90778000
00199 C * 90779000
00200 C * ALL OPERATIONS REQUIRE 90780000
00201 C * OCR-FILE-ID = THE UNIQUE NAME USED TO IDENTIFY THE FILE 90781000
00202 C * TO THE SUBROUTINE AND TO THE SYSTEM 90782000
00203 C * OCR-OPERATION = THE CODE FOR THE REQUESTED OPERATION 90783000
00204 C * ALL OPERATIONS RETURN 90784000
00205 C * OCR-STATUS-KEY = RETURN CODE FOR VARIOUS OCCURRENCES 90785000
00206 C * 90786000
00207 C * OCR-OPEN ('OPEN ') ALSO REQUIRES 90786200
00208 C * OCR-FORMAT-RECORD-ID = LIBRARY NAME OF DFR TO LOAD 90786400
00209 C * OCR-CLOSE ('CLOSE') REQUIRES NO ADDITIONAL PARAMETERS 90786600
00210 C * OCR-READ ('READ ') ALSO REQUIRES 90786800
00211 C * OCR-LINE-NUMBER (1-33) = LINE TO READ (ON DOCUMENT) 90786900
00212 C * OCR-LINE-FORMAT (1-63) = DLINT NUMBER (IN CURRENT DFR) 90787900
00213 C * AND RETURNS (IF OCRS-SUCCESSFUL) 90788100
00214 C * OCR-HEADER-RECORD = HEADER RECORD, AS RETURNED BY THE 3886 90788300
00215 C * OCR-DATA-RECORD = DATA FROM DOCUMENT, FROM 3886 90788500
00216 C * OCR-READ-OVERLAPPED ('READO') HAS SAME REQUIREMENTS AS OCR-READ 90788800
00217 C * OCR-WAIT ('WAIT ') RETURNS SAME PARAMETERS AS OCR-READ 90789800
00218 C * OCR-MARK-LINE ('MARKL') ALSO REQUIRES 90790000
00219 C * OCR-LINE-NUMBER (1-33) = LINE TO MARK (ON DOCUMENT) 90790200
00220 C * OCR-MARK (1-15) = SUM OF DESIRED MARK CODES (8421) 90790400
00221 C * OCR-MARK-DOCUMENT ('MARKD') ALSO REQUIRES 90790600
00222 C * OCR-MARK (1-15) = SUM OF DESIRED MARK CODES (8421) 90790700
00223 C * OCR-EJECT ('EJECT') ALSO REQUIRES 90791700
00224 C * OCR-STACKER (1-2) = STACKER TO SELECT (A OR B) 90791900
00225 C * OCR-LINE-NUMBER (0-33) = NUMBER OF LINES ON DOCUMENT 90792100
00226 C * FOR VALIDATION (IF 0, NO VALIDATION WILL OCCUR) 90792500
00227 C * OCR-SET-DEVICE ('SETDV') ALSO REQUIRES 90792600
00228 C * OCR-FORMAT-RECORD-ID = LIBRARY NAME OF DFR TO LOAD 90793600
00229 C * 90793800
00230 C * 90794000
00231 C * 1. THE TERMS DFR AND DLINT ARE USED TO REFER TO THE EXPANDED 90794200
00232 C * CODE, IN LOADABLE FORM, OF THE RESPECTIVE SYSTEM MACROS. 90794400
00233 C * 2. OCR-WAIT MAY BE REQUESTED AFTER, AND ONLY AFTER, A 90795300
00234 C * SUCCESSFUL OCR-READ-OVERLAPPED REQUEST. NO INTERVENING 90795500
00235 C * I/O COMMANDS WILL BE ALLOWED ON THAT SAME FILE. 90795700
00236 C * 3. THE PROCEDURES PROVIDED BELOW AUTOMATICALLY FILL IN 90795900
00237 C * THE OCR-OPERATION FIELD, CALL THE SUBROUTINE, AND TEST 90796100
00238 C * THE OCR-STATUS-KEY AFTER RETURN. IF ANY EXCEPTIONAL 90796400
00239 C * CONDITIONS OCCUR, THEY PASS CONTROL TO THE ROUTINE 90796600
00240 C * OCR-EXCEPTION-ROUTINE, WHICH THE PROGRAMMER MUST PROVIDE. 90796700
00241 C * THE PROGRAMMER MAY AVOID EXCEPTION ROUTINE INVOCATION BY 90797900
00242 C * ADDING THE FOLLOWING PHRASE TO THE COPY STATEMENT: 90798100
00243 C * REPLACING OCR-EXCEPTION-ROUTINE BY OCR-CALL-EXIT 90798300
00244 C * 4. ALTHOUGH OCR-STATUS-KEY MAY INDICATE THAT THE DESIRED 90798300
00245 C * OPERATION WAS SUCCESSFUL, THE VALIDITY OF THE DATA OBTAINED
00246 C * SHOULD BE DETERMINED BY TESTING OCRH-LINE-STATUS.
*****90798700

```

Figure 69. Sample OCR Program (Part 4 of 5)

|         |                                                        |          |
|---------|--------------------------------------------------------|----------|
| 00247 C | OCR-3886-PROCEDURES.                                   | 90799700 |
| 00248 C | OCR-OPEN.                                              | 90800700 |
| 00249 C | MOVE 'OPEN ' TO OCR-OPERATION OF OCR-FILE.             | 90807000 |
| 00250 C | PERFORM OCR-CALL THRU OCR-CALL-EXIT.                   | 90817000 |
| 00251 C | OCR-CLOSE.                                             | 90827000 |
| 00252 C | MOVE 'CLOSE' TO OCR-OPERATION OF OCR-FILE.             | 90837000 |
| 00253 C | PERFORM OCR-CALL THRU OCR-CALL-EXIT.                   | 90847000 |
| 00254 C | OCR-READ.                                              | 90857000 |
| 00255 C | MOVE 'READ ' TO OCR-OPERATION OF OCR-FILE.             | 90867000 |
| 00256 C | PERFORM OCR-CALL THRU OCR-CALL-EXIT.                   | 90877000 |
| 00257 C | OCR-READ-OVERLAPPED.                                   | 90887000 |
| 00258 C | MOVE 'READO' TO OCR-OPERATION OF OCR-FILE.             | 90897000 |
| 00259 C | PERFORM OCR-CALL THRU OCR-CALL-EXIT.                   | 90907000 |
| 00260 C | OCR-WAIT.                                              | 90917000 |
| 00261 C | MOVE 'WAIT ' TO OCR-OPERATION OF OCR-FILE.             | 90927000 |
| 00262 C | PERFORM OCR-CALL THRU OCR-CALL-EXIT.                   | 90937000 |
| 00263 C | OCR-MARK-LINE.                                         | 90947000 |
| 00264 C | MOVE 'MARKL' TO OCR-OPERATION OF OCR-FILE.             | 90957000 |
| 00265 C | PERFORM OCR-CALL THRU OCR-CALL-EXIT.                   | 90967000 |
| 00266 C | OCR-MARK-DOCUMENT.                                     | 90977000 |
| 00267 C | MOVE 'MARKD' TO OCR-OPERATION OF OCR-FILE.             | 90987000 |
| 00268 C | PERFORM OCR-CALL THRU OCR-CALL-EXIT.                   | 90997000 |
| 00269 C | OCR-EJECT.                                             | 91007000 |
| 00270 C | MOVE 'EJECT' TO OCR-OPERATION OF OCR-FILE.             | 91017000 |
| 00271 C | PERFORM OCR-CALL THRU OCR-CALL-EXIT.                   | 91027000 |
| 00272 C | OCR-SET-DEVICE.                                        | 91037000 |
| 00273 C | MOVE 'SETDV' TO OCR-OPERATION OF OCR-FILE.             | 91047000 |
| 00274 C | PERFORM OCR-CALL THRU OCR-CALL-EXIT.                   | 91057000 |
| 00275 C | OCR-CALL.                                              | 91067000 |
| 00276 C | CALL 'ILEDOCR0' USING OCR-FILE.                        | 91077000 |
| 00277 C | IF NOT OCRS-SUCCESSFUL OF OCR-FILE,                    | 91087000 |
| 00278 C | GO TO OCR-EXCEPTION-ROUTINE.                           | 91097000 |
| 00279 C | OCR-CALL-EXIT. EXIT.                                   | 91107000 |
| 00280 C | ***** END OF 3886 PROCEDURE DIVISION COPY MEMBER ***** | 91109000 |

Figure 69. Sample OCR Program (Part 5 of 5)



This index is supplemented with entries from the index of IBM DOS Full American National Standard COBOL. These entries are identified by an asterisk (\*).

(Where more than one page reference is given, the major reference appears first.)

- \* 22,205,42
- \* in arithmetic expressions\*  
(see also asterisks, used in PICTURE clause)\*
- \*\* in arithmetic expressions\*
- /\* 15
- /& 15
- { (see braces)\*
- [ (see brackets)\*  
(see pound sign)\*
- . (see period)\*
- ... (see ellipsis)\*
- < in relation conditions\*
- ( and ) in\*  
  arithmetic expressions\*  
  compound conditions\*  
  PICTURE clause\*  
  subscripting and indexing\*
- + (see plus symbol)\*
- \$ (see currency symbol, dollar sign)\*
- ; in Data Division and Procedure Division entries\*  
(see also semicolon)\*
- (see either hyphen, or minus symbol)\*
- / in  
  arithmetic expressions\*  
  sterling report items\*
- , (see comma)\*
- > in relation conditions\*
- " in  
  COMPUTE statement\*  
  relation conditions\*
- ' or ' in nonnumeric literals\*  
(see also quotation mark)\*
  
- A, in PICTURE clause\*  
  alphabetic items\*  
  alphabetic symbol\*  
  alphanumeric edited items\*  
  alphanumeric items\*
- abbreviations\*
- abnormal termination 251-253
- absolute\*  
  column number\*  
  line spacing in a report\*  
  values in MOVE statement\*
- ACCEPT statement 73,\*  
  subroutines 290
- Access Method Services 130
- ACCESS MODE and VSAM capabilities\*
- ACCESS MODE clause\*
- ACCESS MODE clause, VSAM\*
  
- accessing a direct file 99-117,\*  
  randomly 101  
  sequentially 100
- accessing an indexed file 118-122,\*  
  randomly 121-122  
  sequentially 121
- accessing a sequential file 99,\*
- acknowledgement\*
- action request time file status values (VSAM) 138
- ACTION statement 42
- actual decimal point\*
- actual key 101-117  
  actual track addressing 101-110, 116-117  
    considerations for specific devices 116-118  
    sample program 106-110
- relative track addressing 101-106  
  sample program 111-115
- structures 102
- ACTUAL KEY clause 101-102  
  description and format 102,\*  
  READ statement\*  
  REWRITE statement\*  
  SEEK statement\*  
  WRITE statement\*
- actual track addressing 101-110,116-117  
  considerations for specific devices 116-118  
  sample program 106-110
- ADCON table 298
- ADD statement (Librarian) 51,52  
  description and formats 51,\*
- adding a record to a prime track 120
- adding records to an indexed file 119-120, 121-122
- adding source statements to a book 51
- addition operator\*
- addressing direct files  
  actual track addressing 101-110, 116-117  
    sample program 106-110
- relative track addressing 101-106  
  sample program 111-115
- addressing schemes 99-122,\*  
  direct 99-117,\*  
  indexed 118-122,\*  
  sequential 99,\*
- advantage of S-mode records over V-mode records 186
- algebraic value in a sign condition\*
- algorithm\*
- alignment of data items\*  
  comparisons\*



decimal point\*  
 editing\*  
 File Section Items\*  
 JUSTIFIED clause\*  
 Linkage Section Items\*  
 PICTURE clause 196,\*  
 SYNCHRONIZED clause 200,\*  
 USING option\*  
 VALUE clause\*  
 Working-Storage Items 194,\*  
 ALL literal figurative constant\*  
   description\*  
   MOVE statement\*  
   relation condition\*  
   STOP statement\*  
 alphabetic character\*  
 alphabetic class test\*  
 alphabetic collating sequence\*  
 alphabetic data items\*  
 alphabetized cross-reference listing  
   (SXREF) 37,36,67,\*  
 alphanumeric character\*  
 alphanumeric collating sequence\*  
 alphanumeric data item\*  
   allowable symbols\*  
   class test\*  
   description\*  
   internal representation\*  
   JUSTIFIED clause\*  
   MOVE statement 205,\*  
   relation condition\*  
   USAGE clause 197,\*  
   VALUE clause\*  
 alphanumeric edited character\*  
 alphanumeric edited item\*  
 alphanumeric literals\*  
 ALT parameter of ASSGN job control  
   statement 25,24  
 ALTER statement  
   in a called program 75  
   in debug packets\*  
   description and format\*  
   effect on GO TO statement\*  
   segmentation\*  
   sort procedure\*  
 altering characters\*  
 altering execution sequence\*  
 altering usage of data items\*  
 alternative grouping of data\*  
   REDEFINES clause\*  
   RENAMES clause\*  
 amount of intermediate storage required for  
   sort 208  
 AND logical operator\*  
 APOST option 37  
 apostrophe\*  
 application programs 12  
 APPLY clause\*  
   CORE-INDEX option 101,99,\*  
   CYL-INDEX option 101,99,\*  
   CYL-OVERFLOW option 101,99,\*  
   EXTENDED-SEARCH option 101,99\*  
   MASTER-INDEX option 101,99,\*  
   WRITE-ONLY option 100,99,183,192,\*  
   programming technique\*  
   WRITE-VERIFY option 99,\*  
 arabic numerals\*  
 Area A and Area B in reference format\*  
 arguments  
   passed to a called assembler language  
   program 79  
   passed to a called COBOL program 76,77  
 arithmetic expressions\*  
   characters used\*  
   COMPUTE statement 204,\*  
   conditions\*  
   description\*  
   evaluation rules\*  
 arithmetic operators\*  
 arithmetic statements\*  
   ADD\*  
   COMPUTE 204,\*  
   CORRESPONDING option 205,\*  
   DIVIDE\*  
   GIVING option\*  
   intermediate results\*  
   MULTIPLY\*  
   overlapping operands\*  
   REMAINDER option\*  
   ROUNDED option\*  
   SIZE ERROR option\*  
   SUBTRACT\*  
 arithmetic subroutines 293  
 ascending sequence\*  
   ASCII character set\*  
   EBCDIC character set\*  
   sort\*  
   table handling\*  
 ASCII considerations\*  
 ASCII subroutines 291  
 ASCII tape files 177-178  
 assembler language routine for  
   accomplishing overlay 82  
 assembler language subprograms 78-81  
 assembler sublibrary of source statement  
   library 48  
 ASSGN control statement 24-25,20  
 ASSIGN clause 24,19  
   ASCII considerations\*  
   description and format 24,\*  
   NSTD-REELS special register 166,\*  
   sort\*  
     file in GIVING option\*  
     sort work units\*  
     system-name\*  
 ASSIGN clause, VSAM\*  
 assigning storage for compiler work file  
   buffers 36  
 assignment of input/output devices 19-20  
 assumed\*  
   decimal point\*  
   description\*  
   numeric edited items\*  
   numeric items\*  
   sterling nonreport items\*  
   decimal scaling positions\*  
   pound and shilling separators\*  
 asterisk (\*)  
   arithmetic expressions\*  
   comments\*  
   in job deck 22  
   in PHASE statement 42  
   PICTURE clause\*  
     check protect symbol\*  
     numeric edited items\*

- sterling report items\*
  - in source program 205
- AT END phrase\*
  - READ statement\*
  - RETURN statement\*
  - SEARCH statement\*
- AUTHOR paragraph\*
- AUTOLINK feature 43
- Automatic Library Look-Up (AUTOLINK) 43
  - and ACTION control statement 42,43
  - and PHASE control statement 41,43
- auxiliary subroutines 291

- B, in PICTURE clause\*
  - alphanumeric edited items\*
  - numeric edited items\*
  - space symbol\*
  - sterling report items\*
- background program 12
  - label area 301
- EASIS card 57,48
  - used for debugging 250
- batched-job mode 12
- BEFORE REPORTING declarative\*
- binary collating sequence\*
- binary data item 197-202,\*
  - in PICTURE clause 196,\*
  - description\*
  - intermediate results 203
  - internal representation\*
  - MOVE statement 205,\*
  - relation condition\*
  - SYNCHRONIZED clause 200,\*
  - USAGE clause 197,\*
- BKEND control statement 49
- BLANK WHEN ZERO clause\*
- BLOCK CONTAINS clause 194
  - ASCII considerations\*
  - VSAM\*
- block descriptor field 183
- block-length field
  - V-mode records 180
- block prefix 177
  - and BLOCK CONTAINS clause 194
- blocked records 180,181,\*
  - APPLY WRITE-ONLY clause\*
  - BLOCK CONTAINS clause 183,\*
  - inter-record slack bytes\*
  - recording mode\*
- body print group\*
- books in the source statement library
  - cataloging 48-49
  - retrieving 49
  - updating 49-51
- boundary alignment\*
- braces 22
- braces and brackets in formats\*
- brackets 22,23
- BUF option 36
- buffer\*
  - allocation\*
  - ASCII considerations\*
  - combined function processing\*
  - offset 177
  - truncation\*

- buffer allocation VSAM\*
- building tables 225
- bypassing label processing\*
- byte, contents of\*
- C, in sterling PICTURE\*
- CALL statement 75-76
  - boundary alignment in\*
    - in segmented program 90
    - 3886 OCR processing\*
- Called program 75
- calling an assembler language
  - subprogram 78-82
- calling and called programs 75-87
- calling cataloged procedures 54
- calling program 75
- capacity records 99-100
- capitalized words in formats\*
- carriage control character\*
- CATAL option 33,45
- cataloged procedures, calling 54
- cataloging
  - a book 48-49
  - a module 46-47
  - a program phase 45-46,40,41
  - a segmented program 92
  - a sort program 209
- CATALP control statement 53
- CATALR control statement 47-48
- CATALR option 39
- CATALS control statement 48-49
- CBL statement 36-39
- changing description of data items\*
- changing installation defaults 39
- character set\*
- character string\*
- characters, significant for various
  - options 39
- checking standard labels 162,174
  - DLAB control statement 30
  - DLBL control statement 28
  - TLBL control statement 27
  - TPLAB control statement 30
  - VOL control statement 30
- check protect symbol\*
- checkpoint/restart during a sort 210
- checkpoint subroutine 293
- checkpointing a COBOL program 262-263
- checkpoints during a sort operation 210
  - control statement requirements 263
- CHKPT macro instruction 262
- class test subroutine 294
- classes of data\*
- CLIST option 37
- CLOSE control statement 26
  - description and format 26,\*
  - OPEN REVERSED statement\*
  - random file options\*
  - sequential file options\*
- CLOSE statement, VSAM\*
- CLOSE UNIT subroutine 290
- CLOSE WITH LOCK subroutine 289
- COBOL execution output 72
- COBOL language usage with VSAM 141
- COBOL library subroutines 289-296
- COBOL option card (CBL card) 36-39

COBOL sublibrary of source statement  
   library 48  
 COBOL support for 3886 Optical Character  
   Reader 320  
 COBOL VSAM control blocks 145-149  
 CODE clause\*  
 coding considerations for DOS/VS 191,\*  
 coding form, COBOL\*  
 collating sequence\*  
   ASCII\*  
   EBCDIC\*  
   for sort\*  
 COLUMN clause\*  
 combined function processing\*  
 comma, exchanging with period\*  
 comma\*  
   in editing\*  
   in statement formats 23  
   PICTURE string\*  
   as punctuation\*  
 commands, job control 40  
 comment control statement 22  
 comments in job control statements 22  
 comments on the phase map 71  
 common exit point for procedures\*  
 common processing facilities, VSAM\*  
   current record pointer\*  
   INTO/FROM identifier option\*  
   INVALID KEY condition\*  
   status key\*  
 Communication Region 301  
   DATE control statement 26  
 COMP items\*  
 COMP-1 items\*  
 COMP-2 items\*  
 COMP-3 items\*  
 COMP-4 items\*  
 comparison\*  
 compare subroutines 293  
 compilation 17  
   foreground 300,12  
   job steps 13  
   of copied text\*  
   of debugging packet\*  
   options for 36-38,32-33  
   work files required for 297  
 compile and edit job 14  
 compile, edit, and execute job 14  
 compile-only job 14  
 compile-time debugging packet\*  
 compiler capacity 297-298  
 compiler diagnostic messages 67  
   generation of 260,307  
   working with 260  
 compiler directing statements\*  
   BASIS\*  
   COPY\*  
   DEBUG\*  
   description\*  
   DELETE\*  
   EJECT\*  
   ENTER\*  
   INSERT\*  
   list of\*  
   NOTE\*  
   SKIP\*  
 compiler-generated card number  
   on diagnostic messages 67  
   on object code listing 66  
   on source statements 59  
 compiler machine requirements 297  
 compiler messages 67,260,307  
 compiler options  
   CBL card 36-39,59  
   current-date\*  
   OPTION control statement 32-33  
   quotation mark\*  
   sequence checking\*  
   truncation of binary items\*  
 compiler output 59-69  
   from a segmented program 91  
 compiler statistics 66  
 compiler work files 297  
 completion codes from sort program 209  
 computational items  
   conversions involving 197-200  
   internal representation of 200-202  
   special considerations for 200  
 COMPUTATIONAL usage\*  
 COMPUTATIONAL-1 items\*  
 COMPUTATIONAL-1 usage\*  
 COMPUTATIONAL-2 items\*  
 COMPUTATIONAL-2 usage\*  
 COMPUTATIONAL-3 items\*  
 COMPUTATIONAL-3 usage\*  
 COMPUTATIONAL-4 items\*  
 COMPUTATIONAL-4 usage\*  
 COMPUTE statement  
   programming technique 204  
   subroutines 293  
 computer-name\*  
   OBJECT-COMPUTER paragraph\*  
   SOURCE-COMPUTER paragraph\*  
   System/370 instruction\*  
 COM-REG 301  
 condensed object listing 37  
 condition-name\*  
 condition-name condition\*  
 conditional sentence, definition\*  
 conditional statement\*  
 conditional variable\*  
 conditions\*  
   compound conditions\*  
   PERFORM statement\*  
   SEARCH statement\*  
   test conditions\*  
 Configuration Section\*  
   copying\*  
   description and format\*  
   OBJECT-COMPUTER paragraph\*  
   SOURCE-COMPUTER paragraph\*  
   SPECIAL-NAMES paragraph\*  
   and System/370 instruction generation\*  
 considerations when using overlay  
   structures 81  
 CONSOLE\*  
   ACCEPT statement\*  
   DISPLAY statement\*  
   SPECIAL-NAMES paragraph\*  
 constant\*  
 continuation of  
   job control statements  
     DLAB control statement 30  
     TPLAB control statement 30  
   line-control cards 230  
   program-control cards 230

control blocks, COBOL VSAM 145-149  
control breaks\*  
control bytes\*  
    BLOCK CONTAINS clause\*  
    inter-record slack bytes\*  
    S-mode and V-mode records\*  
control card, COBOL option 36  
CONTROL clause\*  
control fields  
    S-mode records 183,184  
    V-mode records 180-181  
control footings and page format 214  
control hierarchy\*  
control program 11  
CONTROL report group\*  
    CONTROL clause\*  
    GENERATE statement\*  
    LINE clause\*  
    NEXT GROUP clause\*  
    PAGE LIMIT clause\*  
    report group description entry\*  
    summation\*  
    TYPE clause\*  
controls in report writer\*  
control sections 42  
control statement placement  
    job control statements 23  
    linkage editor control statements 41  
    symbolic debug control  
    statements 230-231  
    UPDATE function 52  
conventional use of linkage registers 78  
conversion subroutines 291-292  
converting elementary data items 197-200  
converting non-VSAM files to VSAM  
files 144  
converting track addresses  
    in a COBOL source program  
    relative to actual 103,106  
    in EXTENT control statement  
    actual to relative 29  
    relative to actual 29  
copy function of Librarian 45  
COPY statement 49  
copying\*  
core image directory 45  
core image library  
    private 55,18,32  
    and Linkage Editor 40,300  
    system 45-46  
    and Linkage Editor 40,41  
CORE-INDEX option of the APPLY clause\*  
correspondence of arguments and parameters  
    assembler language subprograms 79  
    COBOL subprograms 77  
CORRESPONDING option\*  
counter updating\*  
CR, in PICTURE clause\*  
creating a direct file 101  
    actual track addressing 101-110,  
    116-117  
    sample program 106-110  
    relative track addressing 101-110  
    sample program 111-115  
    sample job decks 304,303  
creating an indexed file 121  
    sample job deck 305,303  
creating a VSAM file  
    examples of 141-144  
    language statements required for 141  
creating standard mass storage file  
labels 175,174  
    DLAB control statement 30  
    DLBL control statement 28  
    PARSTD option 33  
    STDLABEL option 33  
creating standard tape file  
labels 162,163-166  
    PARSTD option 33  
    STDLABEL option 33  
    TLBL control statement 27  
    TPLAB control statement 30  
creating user labels 162,165,174,175  
    USRLABEL option 33  
credit symbol\*  
cross-footing\*  
cross-reference dictionary 67  
    alphabetically ordered 37  
    source ordered 33  
CSYNTAX option 38  
CURRENCY-SIGN clause\*  
    description and format\*  
    international considerations\*  
    restriction\*  
currency symbol in PICTURE clause\*  
dollar sign\*  
    description\*  
    numeric edited items\*  
pound sign\*  
(see also CURRENCY SIGN clause)\*  
CURRENT-DATE 301,26  
current record pointer, VSAM 134  
cyclic check 288  
cylinder index 119  
cylinder overflow area 119  
CYL-INDEX option, APPLY clause\*  
CYL-OVERFLOW option, APPLY clause\*  
C01 through C12 function-names defined\*  
D, in sterling PICTURE clause\*  
D-mode records 177,180  
data, locating in a dump 253  
data access, VSAM 128  
data conversion 197-200,\*  
    DISPLAY statement 198,199,\*  
    EXAMINE statement\*  
    first character of program-name\*  
    GIVING option\*  
    MOVE statement\*  
    TRANSFORM statement\*  
data description clauses\*  
    BLANK WHEN ZERO  
    data-name\*  
    FILLER\*  
    JUSTIFIED\*  
    OCCURS\*  
    PICTURE 196,\*  
    REDEFINES 194,\*  
    RENAMES\*  
    SYNCHRONIZED 200,\*  
    USAGE 197,\*  
    VALUE\*  
data description entry\*

Data Division\*

- considerations for 3540 Diskette unit files 124
- considerations for VSAM\*
- data extents
  - direct files 28,30
  - indexed files 28,30
- data files 15
- data flow logic in a CALL structure 78
- data format conversion 197-200
- data formats in the computer 200-202
- data item\*
- data item alignment\*
- data item description entry\*
- data management 98,112
- data manipulation statements\*
  - EXAMINE\*
  - MOVE\*
  - TRANSFORM\*
- data-name\*
- data-name clause\*
- data organization, VSAM 128,\*
- DATA RECORDS clause\*
- DATA RECORDS clause, VSAM
- data reference methods\*
- data sets for symbolic debugging\*
- data transformation\*
- DATE-COMPILED 64
- DATE control statement 26
  - and Communication Region 301
- DATE-WRITTEN paragraph\*
- DEBUG card\*
- debug control subroutine 294
- debug packet 250
- debugging language 229-232,247-251
- debugging TESTRUN 233
- decimal point alignment 196,\*
  - MOVE statement 205,\*
  - period insertion character\*
  - rounding\*
  - size error\*
- DECIMAL-POINT IS COMMA clause\*
- DECK option 32
- declaratives\*
  - error processing\*
  - EXIT statement in\*
  - label handling\*
  - report writer\*
  - section\*
    - description and format\*
  - USE sentence\*
- defaults\*
  - ACCESS MODE clause\*
  - APPLY CYL/MASTER-INDEX clause\*
  - BLOCK CONTAINS clause\*
  - changing 39
  - cylinder overflow area\*
  - name-field in system-name\*
  - page format in Report Writer\*
  - printer spacing\*
  - priority number\*
  - quotation mark character\*
  - record size\*
  - recording mode\*
  - segment limit\*
  - sequence checking\*
  - truncation of binary items\*
  - USAGE clause\*
- DEFINE command 130-134
  - defining a VSAM data space: DEFINE SPACE 132
  - defining a VSAM file: DEFINE CLUSTER 133
  - defining a VSAM master catalog: DEFINE MASTERCATALOG 131
- Define The File (DTF) 98
- DEL statement 51,52
- DELETE card 51-52
  - used for copying 57,\*
  - used for debugging 250
- DELETE statement, VSAM 141,\*
- deleting source statements
  - for one run only 51
  - from a book 48
- DEPENDING ON option of OCCURS clause
  - and Table Handling Feature 217-223
  - and variable-length records 218-220
- depth of a report page\*
- descending sequence\*
  - in sort\*
  - in table handling\*
- description and formats of job control statements 22-23
- DETAIL report group\*
  - description\*
  - GENERATE statement\*
  - LINE clause\*
  - NEXT GROUP clause\*
  - SUM counters\*
  - TYPE clause\*
- detailed file processing
  - capabilities 145-176
- determining the location of the libraries 55
- determining the priority of the last segment loaded into the transient area 92
- device assignment 19-20
  - duration of effect 24
- device support, VSAM 130
- diagnostic messages
  - compiler 67,260,307
  - execution-time 261
  - Federal Information Processing Standard (FIPS) 68
  - generation of 260
  - linkage editor 71,261
  - object time 309-317,260-261
  - operator 307-309,73,72
  - sort 209
- direct data organization 97,\*
- direct files 99-118,97
  - accessing techniques 100,101,\*
  - ACTUAL KEY clause 101,\*
  - actual track addressing 101-110, 116-117
  - APPLY EXTENDED-SEARCH clause 101,\*
  - ASSIGN clause 100,101,\*
  - BLOCK CONTAINS clause\*
  - error processing 106,111,\*
  - initiating access 100,\*
  - invalid key condition\*
  - random access 101,\*
  - READ statement\*
  - recording mode\*
  - relative track addressing 101-106, 111-115

REWRITE statement\*  
   sample job decks 304,303  
   sequential access 100,\*  
   user labels\*  
 WRITE statement\*  
 direct indexing\*  
 direct linkage 82  
 direct organization (DTFDA) 99-118,97  
 disk extent subroutines 290  
 Diskette input/output unit (3540)  
   processing 123-125  
 DISPLAY items  
   conversions involving 197-200  
   internal format of 201  
   special considerations for 200  
 DISPLAY statement\*  
 DISPLAY usage\*  
   alignment\*  
   alphabetic items\*  
   alphanumeric items\*  
   ASCII considerations\*  
   description\*  
   edited items\*  
   numeric items\*  
   SYNCHRONIZED clause\*  
 DISPLAY-ST usage\*  
 DISPLAY statement subroutines 290  
 displaying data values during  
   execution 248  
 DIVIDE statement\*  
 division, arithmetic operation\*  
 division by zero\*  
 division header\*  
 division operator\*  
 division of a program, definition\*  
 division/remainder method of  
   randomizing 103-106  
   used to create a direct file  
     actual track addressing 107-110  
     relative track addressing 112-114  
 DLAB control statement 30  
 DLBL control statement 28  
   identifying private libraries 55  
 document description (OCR) 320  
 document design (OCR) 319  
 dollar sign\*  
 DOS/VS COBOL Unresolved External  
   References 71  
 double slash 22  
 double spacing\*  
 doubleword\*  
 DTF  
   creation of 98,150  
   locating in a dump 252-253  
 DTF tables 150-155  
 dummy files\*  
 dummy segment 89  
 DUMP option 32  
 dumps  
   errors that can cause 252  
   how to use 251  
   symbolic 229-245  
   system 251-259  
 duplication of names\*  
  
 E, in  
   external floating-point items\*  
   floating-point numeric literals\*  
 EBCDIC collating sequence (Extended Binary  
   Coded Decimal Interchange Code)\*  
 edit and execute job 14  
 editing 17-18,40-42,\*  
 editing character\*  
 edit-only job 14  
 editor, linkage 11  
 effective storage considerations 298  
 EJECT 59,191,325,\*  
 ellipsis 23  
 ellipsis (...) in formats 23,\*  
 ELSE option\*  
   IF statement\*  
   nested IF statements\*  
   ON statement\*  
 END DECLARATIVES.\*  
 END statement 51,52  
 end-of-data control statement 15  
 end of file\*  
 end-of-job control statement 15  
 end of page condition\*  
 end of volume positioning\*  
 ENTER statement\*  
 ENTRY control statement 42  
 entry point in a called program 76,77  
 entry-sequenced files 127  
   considerations for READ statement 140  
   considerations for REWRITE  
     statement 139  
   considerations for WRITE statement 139  
 ENTRY statement 76  
   in an overlay structure 83  
 Environment Division\*  
   ASCII considerations\*  
   Configuration Section\*  
   Input-Output Section\*  
   international considerations\*  
   organization\*  
   segmentation considerations\*  
   sort considerations\*  
   System/370 card devices\*  
   System/370 instructions\*  
   3540 Diskette unit file  
     considerations 123  
 equal sign (=)\*  
   in COMPUTE statement\*  
   in relation condition\*  
 equal size operands in a relation  
   condition\*  
 equivalents\*  
   reserved words and abbreviations\*  
   THROUGH and THRU\*  
 error bytes\*  
 error conditions, arithmetic operations\*  
 error handling (VSAM) 36  
   file status values  
     at OPEN 138  
     at action request time 138  
     status key 136,130  
 error messages, diagnostic (see messages)  
 error message subroutine 295  
 ERROR procedures, VSAM\*  
 error processing declaratives\*

error recovery for non-VSAM files 155  
   on unit-record devices 35  
   using an assembler language routine 159-161  
   using error declarative section 155-158  
   using INVALID KEY 155,156  
 errors that can cause a dump 252  
 ERRS option 33  
 evaluation rules\*  
 EXAMINE statement\*  
 EXCEPTION/ERROR procedure, VSAM\*  
   CLOSE statement\*  
   DELETE statement\*  
   description and format\*  
   OPEN statement\*  
   READ statement\*  
   REWRITE statement\*  
   START statement\*  
   WRITE statement\*  
 EXEC control statement 15  
 EXEC FCOBOL statement 17,15  
 EXEC LNKEDT statement 17,15  
 execute-only job 14  
 execution, order of\*  
 execution output 72-73  
 execution time  
   considerations 299-300  
   machine requirements 297-298  
   messages 261  
 EXHIBIT statement 248-249,247  
   subroutine 290  
 exit point for procedures\*  
 EXIT PROGRAM statement 76  
   symbolic debugging\*  
 EXIT statement\*  
   description and format\*  
   PERFORM statement\*  
   PROGRAM option\*  
   with sort procedures\*  
 explanatory comments\*  
 exponent\*  
   floating-point items\*  
   floating-point numeric literals\*  
 exponentiation 204  
 Extended Binary Coded Decimal Interchange Code (EBCDIC)\*  
   collating sequence\*  
   nonnumeric literals\*  
 extended search 100  
 EXTENDED-SEARCH option of the APPLY clause 100,\*  
 extended source program library facility 57  
 EXTENT control statement 28-30  
 external data\*  
 external decimal items\*  
 external floating-point items\*  
 external-name 76  
   in ASSIGN clause\*  
 external reference 71  
   unresolved 71  
   weak 71  
  
 F-mode records 179,\*  
 FCOBOL 17  
  
 Features of the DOS/VS Compiler 7  
 FD\*  
 Federal Information Processing Standard (FIPS) 39,\*  
 figurative constants\*  
 file\*  
   and volume portability (VSAM) 130  
   definition\*  
   disposition of\*  
     CLOSE statement\*  
     OPEN statement\*  
   entry-sequenced 127  
   FILE-CONTROL paragraph\*  
   file description entry\*  
   inter-record slack bytes\*  
   I-O-CONTROL paragraph\*  
   key-sequenced 127  
   file control block (FCB) 148  
   FILE-CONTROL paragraph\*  
     ACCESS MODE clause\*  
     ACTUAL KEY clause\*  
     ASSIGN clause\*  
     copying\*  
     FILE-LIMIT clause\*  
     format\*  
     NOMINAL KEY clause\*  
     PROCESSING MODE clause\*  
     RECORD KEY clause\*  
     RESERVE clause\*  
     SELECT clause\*  
     sort considerations\*  
     TRACK-AREA clause\*  
   file description (OCR) 320  
   file description entry\*  
     BLOCK CONTAINS clause\*  
     description and format\*  
     DATA RECORDS clause\*  
     LABEL RECORDS clause\*  
     RECORD CONTAINS clause\*  
     RECORDING MODE clause\*  
     REPORT clause\*  
     report writer\*  
     sort\*  
     VALUE OF clause\*  
   file description entry, VSAM\*  
   file information area for OCR\*  
   file information block (FIB) 146  
   file integrity 116  
   FILE-LIMIT clause\*  
   file-name  
     arguments 79  
     definition\*  
   file organization 97-98  
     direct 99-118,97  
     indexed 118-122,97-98  
     sequential 99,97  
     VSAM 127  
   file processing for 3540 Diskette unit files 123  
   file processing technique\*  
   file retention  
     on direct-access storage devices 28  
     on tape devices 27  
   File Section 194,\*  
   ASCII considerations\*  
   boundary alignment\*  
   content\*  
   copying\*

file description entry\*  
format\*  
naming data\*  
record description entry\*  
report writer considerations\*  
sort considerations\*  
file size for sort\*  
files sharing same storage area\*  
FILE STATUS clause, VSAM 136,\*  
file status values (VSAM)  
  at OPEN 138  
  at action request time 138  
file table 299  
FILLER\*  
  CORRESPONDING option\*  
  inter-record slack bytes\*  
  record description entry\*  
FINAL control\*  
  CONTROL clause\*  
  TYPE clause\*  
final phase of sort\*  
FIPS diagnostic messages 68  
FIPS flagger description 39,\*  
fixed insertion editing\*  
fixed-length records 179  
fixed partitioned multiprogramming 12  
fixed point numeric item\*  
fixed point numeric literal\*  
fixed portion of a segmented program 89,\*  
FLAGE option 36  
FLAGW option 36  
floating first detail 215  
floating insertion editing\*  
floating-point data items and intermediate  
  results 204  
floating-point numeric literal\*  
flowchart\*  
flow diagram of overlay logic 84  
FLOW option  
  description 37,229  
  restriction with OPT 229,37  
  restriction with STXIT 38,229  
flow trace option (see FLOW option)  
flow trace subroutine 295  
footing report groups\*  
foreground compilation 300,12  
foreground programs 12  
form overflow\*  
format\*  
  DISPLAY statement output\*  
  EXHIBIT statement output\*  
  logical records\*  
  report page\*  
  statement 22  
format control of the source program  
  listing\*  
format F records 179  
format notation  
  job control statements 22-23  
  symbolic debug control statements 230  
format S records 183-188  
format U records 180  
format V records 180-183  
formats of blocked and unblocked  
  records 118  
formulas for converting actual to relative  
  track addresses 29  
formulas for converting relative to actual  
  track addresses 29  
fractions, internal floating-point items\*  
FROM identifier option, VSAM  
full FIPS flagging 39,\*  
fullword\*  
function-name\*  
functional commands (VSAM) 130  
functions of COBOL library arithmetic  
  subroutines 293  
GENERATE statement\*  
generation of diagnostic messages 260  
generic terms 22  
GIVING option of error declarative 156-158  
global table 65  
glossary 64  
GOBACK statement 76  
GO TO...DEPENDING ON subroutine 295  
GO TO MORE-LABELS\*  
GO TO statement\*  
GOBACK statement\*  
group\*  
  collating sequence for sort\*  
  contents\*  
  report\*  
GROUP INDICATE clause\*  
group item\*  
halfword\*  
halting execution\*  
header labels\*  
heading print groups\*  
hierarchy\*  
high-intermediate FIPS flagging 39,\*  
HIGH-VALUE (HIGH-VALUES) figurative  
  constant\*  
how to use a dump 251  
hyphen in statement formats 22  
I-O files\*  
I-O-CONTROL entry, VSAM\*  
I-O-CONTROL paragraph\*  
IBM sterling representation\*  
IBM-supplied processing programs 12  
ID Division header\*  
Identification Division\*  
identification field of COBOL source  
  statements 50  
identification of program versions 247  
identifier, definition\*  
identifying records\*  
IF statement 205  
IGN parameter of ASSGN job control  
  statement 25,\*  
ILBDCKP0 subroutine 263,293  
ILBDDUM0 93  
ILBDMNS0 75,294  
ILBDSEMO subroutine 94,293  
ILBDSET0 75,294  
ILBDSRT0 subroutine 209,293  
imperative statements\*  
implementing an OCR operation 319  
improving efficiency 122



improving performance (Sort) 208  
 in-line parameter list 81  
 INCLUDE control statement 42  
 incrementing\*  
 indentation of level numbers\*  
 independent overflow area 119  
 independent segment 89  
 index data items 217  
     MOVE statement\*  
     relation condition\*  
     USAGE clause description\*  
 index-names 217-223  
     description 217,\*  
     modifying values in\*  
     MOVE statement\*  
     OCCURS clause\*  
     PERFORM statement 217,\*  
     relation condition\*  
     SEARCH statement 217,\*  
     SET statement 217,\*  
     value in 217,\*  
 INDEX option of the USAGE clause\*  
 index table 299  
 INDEXED BY option of the OCCURS clause\*  
 indexed files 118-122  
     adding records to 120-121  
     sample job decks 303,305-306  
 indexed organization (DTFIS) 97-98,118-122  
     improving efficiency when using 122  
 indexes 119  
 indexing tables\*  
 indirect addressing 102-106  
 individual type codes used in SYMDMP  
     output 234  
 initial loading of records into a file  
     (VSAM) 137  
 Initial Program Loader (IPL) 11  
 initializing\*  
     index values\*  
     report writer special registers\*  
     sort special registers\*  
     values of data items\*  
 INITIATE statement\*  
 initiating\*  
     access to a mass storage file\*  
     file processing\*  
     report processing\*  
 in-line parameter list 81  
 Input  
     compiler 17  
     Job Control Processor 22  
     Linkage Editor 17,40,41  
         for a segmented program 92  
 input files\*  
     APPLY EXTENDED-SEARCH\*  
     effect of close options\*  
     error handling\*  
     inter-record slack bytes\*  
     intra-record slack bytes\*  
     label handling\*  
     OPEN statement\*  
     READ statement\*  
     record size\*  
 input format for source programs\*  
 INPUT PROCEDURE option 209  
 input/output areas (buffers)\*  
 input/output control statements for  
     sort 207  
 Input/Output Control System (IOCS) 98  
 input/output error subroutines 290  
 input/output verb subroutines 289  
 Input-Output Section\*  
     copying\*  
     description and format\*  
     File-Control paragraph\*  
     I-O-Control paragraph\*  
     sort considerations\*  
 input/output statements\*  
     ACCEPT\*  
     CLOSE\*  
     DISPLAY\*  
     OPEN\*  
     READ\*  
     REWRITE\*  
     SEEK\*  
     START\*  
     WRITE\*  
 input phase of sort 209,\*  
 INSERT card 51,52  
     used for copying\*  
     used for debugging 250  
 insertion\*  
     asterisks\*  
     commas\*  
     currency symbol\*  
     periods\*  
     sign control symbols\*  
     spaces\*  
     zeros\*  
 insertion character, and item size\*  
 insertion editing\*  
 installation defaults, changing 39  
 INSTALLATION paragraph\*  
 integer, description\*  
 integer literals\*  
 inter-record slack bytes\*  
 intermediate results 203-204  
     arithmetic statements\*  
     compound conditions\*  
 intermediate storage required for sort 208  
 internal data\*  
 internal decimal items\*  
     allowable characters\*  
     class test\*  
     collating sequence\*  
     description\*  
     internal representation\*  
     MOVE statement\*  
     relation condition\*  
     slack bytes\*  
     SYNCHRONIZED clause\*  
     USAGE clause\*  
 internal floating-point items\*  
     collating sequence\*  
     description\*  
     internal representation\*  
     MOVE statement\*  
     relation condition\*  
     USAGE clause\*  
 internal representation\*  
     alphabetic and alphanumeric items\*  
     external decimal items\*  
     numeric items\*  
     sterling items\*  
 international currency considerations\*  
 interpreting output 59-74

interrupts, errors causing 252  
 intra-record slack bytes\*  
 INVALID KEY condition 155  
   direct organization 156,99  
   indexed organization 156  
   READ\*  
   REWRITE\*  
   SEEK\*  
   standard sequential organization 156,99  
   START\*  
   WRITE\*  
 INVALID KEY condition, VSAM\*  
   DELETE statement\*  
   description\*  
   READ statement\*  
   REWRITE statement\*  
   START statement\*  
   WRITE statement\*  
 IOCS 98  
 IPL (Initial Program Loader) 11

job 13  
   types 14  
 job control commands 40  
 job control considerations  
   for accomplishing overlay 84  
   for Optical Character Reader 321  
   for sort program 207-208  
   for symbolic debug option 230-231,232  
 job control language for VSAM files 144  
 Job Control Processor 11,22  
   options 32-33  
 job control requirements for 3540 Diskette  
   unit files 124  
 JOB control statement 15,16  
 job control statements 22-36  
   comments in 22  
   definition 13  
   format notation 22-23  
   formation of 22  
   overlay considerations 84  
   sequence of 23  
   sort considerations 207-208  
   symbolic debug option  
   considerations 230-231,232  
 job deck 15,23  
 job definition 13  
 job definition statements 15  
 job processing 17  
 job step 13  
 justification\*  
   and JUSTIFIED clause\*  
   and MOVE statement\*  
 JUSTIFIED clause\*

KEY clauses 121,\*  
   ACTUAL\*  
   NOMINAL 122,\*  
   RECORD 122,\*  
 KEY option\*  
   OCCURS clause\*  
   SEARCH ALL statement\*  
   SORT statement\*

key-sequenced files 127  
   considerations for READ statement 140  
   considerations for REWRITE  
   statement 140  
   considerations for WRITE statement 139  
 key words\*  
 keys\*  
   for SORT statement\*  
   for START statement\*  
   for table SEARCH\*

label area, reserving storage for 30  
 label definition  
   DLAB control statement 30  
   DLBL control statement 28  
   TLBL control statement 27  
   TPLAB control statement 30  
 label handling\*  
 label processing 162-175  
   mass storage file labels 174-175  
   tape file labels 162-173  
 label processing considerations  
   mass storage file labels 174-175  
   tape file labels 165-166  
 label processing subroutines 289,290  
 LABEL RECORDS clause\*  
 LABEL RECORDS clause, VSAM\*  
 language considerations  
   for ASCII tape files 177  
   for VSAM 141  
   for 3540 Diskette unit files 123-124  
 LBLTYP control statement 30  
 leading zeros, suppression\*  
 left justification\*  
 length\*  
   binary items\*  
   BLOCK CONTAINS clause\*  
   data-name in APPLY CORE-INDEX\*  
   external decimal items\*  
   external floating-point items\*  
   internal decimal items\*  
   internal floating-point items\*  
   RECORD CONTAINS clause\*  
   and standard data format\*  
 level indicator\*  
   file description entry\*  
   reference format\*  
   report writer feature\*  
   sort feature\*  
   summary of\*  
 level numbers 193  
   data description entry\*  
   indentation of\*  
   reference format\*  
   special\*  
   use\*  
 level number 01 items\*  
 level number 02-49 items\*  
 level number 66 items\*  
 level number 77 items\*  
 level number 88 items\*  
 LIB option 39  
 Librarian 45-58  
 libraries  
   core-image 45  
   planning 45

- private 55
- procedure 53
- relocatable 46
- source statement 48
- system 45
- library-name\*
- library subroutines 289-296
- LINE clause\*
- line-control cards 231-232
- LINE-COUNTER special register\*
- line overlay (Report Writer) 213
- lines in a report\*
- lines, spacing of\*
  - program output\*
  - report\*
  - source program\*
- LINK option 32,40,260
- linkage 75-77
  - in a called program 76
  - in a calling program 75-76
  - correspondence of arguments and parameters 77
  - entry points 76
  - with the sort feature 209
- linkage conventions 90-93
  - argument list 79
  - assembler subprogram 82
  - overlay considerations 83
  - in-line parameter list 81
  - lowest level subprogram 81
  - register use 78
  - save area 79
- linkage statements for subprograms\*
- link editing 17-18,11
  - a segmented program, example of 92
  - in the foreground 306,12,32,41
  - with overlay 83
  - without overlay 77
- Linkage Editor 11
- linkage editor control statements 40-44
  - fields of 40
  - generated by compiler for segmentation 91,92
  - overlay considerations 83
  - placement of 41
- linkage editor diagnostic of input 71
  - linkage editor input deck 18
- linkage editor input for a segmented program 92
- linkage editor messages 71,261
- linkage editor output 69-73,17-18
  - from a segmented program 92-94
- linkage registers 78
- Linkage Section\*
  - boundary alignment\*
  - content\*
  - copying\*
  - data item description entry\*
  - error declarative\*
  - format\*
  - intra-record slack bytes\*
  - record description entry\*
  - structure\*
  - use of FILLER\*
  - USING option of the CALL statement\*
  - VALUE clause\*
- Linkage Section and OCR processing\*
- linkage with the Sort Feature 209-210
- LIOCS 98
- LIST option 32
- LISTIO control statement 31
- list of compiler features 7
- LISTX option 32
- literal\*
  - description\*
  - as a function-name\*
  - MOVE statement\*
  - nonnumeric\*
  - numeric\*
  - permissible comparisons\*
  - VALUE clause\*
- literal pool 65
- literal tables 299
- Loader, Initial Program (IPL) 11
- loader, relocating 43
- locating a DTF 252-253
- locating data in a dump 253
- locating the Working-Storage Section in dumps 194
- location of the libraries, determining 55
- location of the system libraries 56
- location of slack bytes\*
- LOG option 32
- logic module 114
- logic, overlay 84
- logical connectives in compound conditions\*
- Logical Input/Output Control System (LIOCS) 98
- logical operators\*
- logical record 98
  - description\*
  - redefining\*
  - renaming\*
  - size of\*
  - slack bytes in\*
  - spanning physical blocks 183-188
- long-precision internal floating point items\*
- low FIPS flagging 39,\*
- low-intermediate FIPS flagging 39,\*
- LOW-VALUE (LOW-VALUES) figurative constant\*
  - in ACTUAL KEY clause\*
  - description\*
  - MOVE statement\*
  - track identifier\*
  - TRANSFORM\*
- lower-case words in formats\*
- lowest level program 81
- LVL option 39
- machine considerations 297,300
- magnetic tape\*
- magnitude of floating-point items\*
- main-line routine 203
- main program or subprogram subroutine 294
- MAINT, procedure library catalog 53
- maintenance function of Librarian 46-54
- major control break\*
- mantissa\*
  - description\*
  - internal representation\*
  - PICTURE clause\*

mass storage devices 97  
   error information\*  
   list of\*  
 mass storage files\*  
   CLOSE statement\*  
   OPEN statement\*  
   READ statement\*  
   REWRITE statement\*  
   SEEK statement\*  
   START statement\*  
   WRITE statement\*  
 mass storage file labels 281-285  
 master index 119  
 maximum length\*  
   arithmetic operands\*  
   binary items\*  
   data description entries\*  
   external decimal items\*  
   floating-point items\*  
   internal decimal items\*  
   items in EXHIBIT statement\*  
   keys in table handling\*  
   numeric edited items\*  
   PICTURE character string\*  
   record\*  
   table elements\*  
 maximum number\*  
   index-names\*  
   keys\*  
   procedure-names in GO TO statement\*  
 maximum size\*  
 maximum value\*  
   ACTUAL KEY track identifier\*  
   binary item and PICTURE clause\*  
   floating-point items\*  
   NSTD-REELS special register\*  
   ON statement integer\*  
   RERUN clause integer\*  
   RESERVE clause integer\*  
   subscript\*  
 MEMORY SIZE clause\*  
 MERGE Feature 207-210  
 MERGE interface subroutine 293  
 messages, diagnostic  
   compiler 67,260,307  
   execution-time 261  
   Federal Information Processing Standard  
   (FIPS) 68  
   generation of 260  
   linkage editor 71,261  
   object-time 309-317,73,260-261  
   operator 307-309,73,72  
   sort 209  
   VSAM 130  
 methods of data reference\*  
 minimum machine requirements for the  
   compiler 297  
 minimum value\*  
 minor control break\*  
 minus symbol\*  
   arithmetic expressions\*  
   collating sequence\*  
   external floating-point literals\*  
   indexing\*  
   PICTURE clause\*  
   SIGN clause\*  
   as unary operator\*  
 miscellaneous tables 299  
  
 mnemonic-name\*  
   ACCEPT statement\*  
   assignment of\*  
   CODE clause\*  
   DISPLAY statement\*  
   SPECIAL-NAMES paragraph\*  
   WRITE statement\*  
 mode F records 179,\*  
 mode U records 180,\*  
 mode V records 180-183,\*  
 modification\*  
   library text\*  
     DELETE and INSERT cards 51,52,\*  
   sort records\*  
     after sorting\*  
     before sorting\*  
 modularizing  
   the Data Division 194  
   the Procedure Division 202-203  
   when using the Segmentation Feature 89  
 module 11  
   input to Linkage Editor 40,41  
 MOVE statement 205  
   CORRESPONDING option\*  
   description and formats\*  
   permissible moves\*  
   sort special registers\*  
   sterling items\*  
 MOVE statement subroutines 294  
 MTC control statement 31  
 multifile volumes  
   examples  
     creating 166-167,170-171  
     input processing 166,168-169  
   TLBL control statement 27  
 multiphase program 18,13,14  
 MULTIPLE FILE TAPE clause\*  
 multiple file tape subroutine 290  
 multiple phases 41  
 multiple results\*  
   ADD statement\*  
   SUBTRACT statement\*  
 multiplication\*  
 multiplication operator\*  
 MULTIPLY statement\*  
 multiprogramming 12,300  
 multivolume processing\*  
   ASSIGN clause\*  
   CLOSE statement\*  
   NSTD-REELS special register\*  
   reading\*  
   writing\*  
 multivolume tape files  
   example of creating 167,172-173  
   with nonstandard labels 166  
 mutually exclusive CBL options 39,37-38  
  
 name\*  
   data item\*  
   description of\*  
   field in system-name\*  
   indexing of\*  
   job\*  
   procedure\*  
   qualification\*  
   record\*  
   subscripting of\*

naming conventions  
  used by sort 209  
  used by segmentation 91-92  
negative operand in a sign condition\*  
negative value\*  
  DISPLAY statement\*  
  external floating point items\*  
  numeric edited items\*  
  PERFORM statement\*  
  SIGN clause\*  
  sign condition\*  
nested\*  
  IF statements\*  
  OCCURS clauses\*  
  PERFORM statements\*  
  REDEFINES clauses\*  
NEXT GROUP clause 214  
  description and format\*  
  effect of PRINT-SWITCH\*  
NOAUTO option 42,43  
NOCATALR option 39  
NOCLIST option 37  
NODECK option 32  
NODUMP option 32  
NOERRS option 33  
NOLIB option 39  
NOLINK option 32  
NOLIST option 32  
NOLISTX option 32  
NOLOG option 32  
NOLVL option 39  
NOMAP option 18,43,42  
NOMINAL KEY clause 121,122  
  description and format\*  
  indexed files\*  
  READ statement\*  
  REWRITE statement\*  
  START statement\*  
  WRITE statement\*  
noncontiguous data items\*  
nonnumeric literals\*  
  continuation of\*  
  description\*  
  EXAMINE statement\*  
  MOVE statement\*  
  relation condition\*  
  TRANSFORM statement\*  
  VALUE clause\*  
nonnumeric operands\*  
  MOVE statement\*  
  relation condition\*  
nonstandard labels 162,\*  
  declarative\*  
  GO TO MORE-LABELS\*  
  LABEL RECORDS clause\*  
  multiple reel files\*  
  NSTD-REELS special register\*  
  reversed reading\*  
  system procedures\*  
nonstandard tape file labels 162  
  multivolume file considerations 166  
  restriction for ASCII files 177  
non-VSAM files, converting to VSAM 144  
NOOPTIMIZE (NOOPT) option 37  
NOREL option 42  
normalized level numbers 193  
NOSEQ option 36  
NOSTATE option 38  
NOSTXIT option 37  
NOSUPMAP option 36  
NOSXREF option 37  
NOSYM option 33  
NOSYNTAX option 38  
NOT condition construction\*  
NOT logical operator\*  
NOTE statement 205  
NOTRUNC option 37  
NOVERB option 39  
NOXREF option 33  
NOZWB option 37  
NSTD-REELS 166  
null report group\*  
numeric character\*  
  description\*  
  PICTURE clause\*  
numeric class test\*  
numeric data item\*  
  BLANK WHEN ZERO clause\*  
  class test\*  
  EXAMINE statement\*  
  fixed-point\*  
  floating-point\*  
  internal representation\*  
  MOVE statement\*  
  PICTURE clause\*  
  relation condition\*  
  VALUE clause\*  
numeric edited items\*  
  BLANK WHEN ZERO clause\*  
  description\*  
  MOVE statement\*  
  relation condition\*  
  USAGE clause\*  
numeric item, description\*  
numeric literal\*  
  continuation\*  
  description\*  
  MOVE statement\*  
  relation condition\*  
  VALUE clause\*  
numeric operands\*  
  ADD statement\*  
  COMPUTE statement\*  
  DISPLAY statement\*  
  DIVIDE statement\*  
  EXAMINE statement\*  
  MOVE statement\*  
  MULTIPLY statement\*  
  relation conditions\*  
  SUBTRACT statement\*  
object code  
  listing 66  
  optimized 37  
OBJECT-COMPUTER paragraph\*  
  computer-name\*  
  copying\*  
  description and format\*  
  SEGMENT-LIMIT clause\*  
  System/370 instruction generation\*  
object module 69  
  produced by the compiler for  
  Segmentation 91-92

- object of a condition\*
- object program definition\*
- object-time control cards for symbolic debugging feature 230
- object-time debugging subroutines 294-295
- object-time messages 309-317,260-261,73
- obtaining an ASCII collating sequence on a sort 178
- OCCURS clause 217-220
  - DEPENDENDING ON option 217-218
  - DEPENDENDING ON table 299
  - description and formats\*
  - direct indexing\*
  - redefining restriction\*
  - relative indexing\*
  - renaming restriction\*
  - slack bytes\*
  - value restriction\*
  - with Table Handling Feature 217-220
- OCR processing 319,\*
- OF qualifier connective\*
  - with indexes and subscripts\*
  - with name\*
- OFF/ON STATUS clause\*
- OMR (optical mark read) processing\*
- ON statement 248,260
- ON/OFF STATUS clause\*
- ON SIZE ERROR option and intermediate results 204
- OPEN statement\*
  - declaratives\*
  - description and format\*
- OPEN statement, VSAM
- operands, overlapping\*
- operand table 299
- operator communication
  - ACCEPT statement 73
  - job control commands 40
  - PAUSE control statement 34
  - STOP statement 72
- operator intervention between job steps 34
- operator messages 307-309,72
  - ACCEPT statement 73
  - STOP statement 72
- Optical Character Reader, 3886
  - COBOL library subroutine for 290,325
  - COBOL support 320
  - document description 320
  - document design 319
  - file description 320
  - JCL considerations 321
  - procedural code 321
  - processing 319
  - processing tapes from the OCR 3886, Model 2 326
  - record description 320
  - sample program 327-331
  - statements for invoking functions
    - CLOSE function 324
    - EJECT function 325
    - MARKD function 325
    - MARKL function 324
    - OPEN function 324
    - READ function 324
    - READO function 324
    - SETDV function 325
    - WAIT function 324
  - Status Key values
    - by operation 322
    - user actions 322
    - user responses 323
  - subprogram interface 321
- optical mark read processing\*
- OPTIMIZE (OPT) option
  - description 37
  - restriction with FLOW 229,37
  - restriction with NOLINK and NODECK 38
  - restriction on number of procedure blocks 18
- optimized object code 38
- optimizer DISPLAY subroutine 295
- optimizer subroutines 295
- OPTION control statement 32-33
  - duration of effect 33
- OPTIONAL (SELECT clause) 24
- optional words in formats\*
- options available during link-editing 44
- options for compilation
  - CBL card 36-39
  - mutually exclusive 39,37-38
  - OPTION control statement 32-33
- OR logical operator in compound conditions\*
- order of evaluation for compound conditions\*
- order of execution\*
- organization\*
  - COBOL program\*
  - data\*
  - Data Division\*
  - Data Division entries\*
  - Environment Division\*
  - field of system-name\*
  - Identification Division\*
  - Procedure Division\*
- ORGANIZATION clause, VSAM
- organization of files 97-98
  - direct 98-118,97
  - indexed 118-122,97-98
  - sequential 99,97
  - VSAM 127
- origin point of phase 41
- output
  - COBOL execution 72
  - compiler 59-74,17
  - complete sample program 265-278
  - EXHIBIT statement 248-249
  - from a segmented program 91-94
  - linkage editor 69-72,17-18
  - phase execution 72-73
  - system 73
  - TRACE statement 247
- output files\*
  - CLOSE options\*
  - error handling\*
  - inter-record slack bytes\*
  - intra-record slack bytes\*
  - label handling\*
  - OPEN statement\*
  - WRITE statement\*
- output line overlay 213
- output phase of sort\*
- OUTPUT PROCEDURE option 209
- output source listing format\*
- overall considerations for symbolic debugging feature 232

overall description of VSAM\*  
 overflow area 119-120  
 overflow records\*  
 overlapping data groupings\*  
 overlapping operands\*  
   arithmetic statements\*  
   MOVE statement\*  
   TRANSFORM statement\*  
 overlay 14  
   assembler language subroutine for 82  
   using Segmentation Feature 89-94  
   using subprogram linkage 81-87  
 overlay logic 83  
 overlay structures 81-87  
   job control considerations 84  
   linkage editor 83-84  
   PHASE statement 41  
   provided by Segmentation Feature 89-94  
 overlayable fixed segment 89  
 overlaying procedures\*

P, in PICTURE clauses\*  
 packed decimal format\*  
 padding in a physical record\*  
 page breaks 213  
 page change in a report\*  
 page condition\*  
 PAGE-COUNTER special register\*  
 PAGE FOOTING report group\*  
 page format in Report Writer\*  
 PAGE HEADING report group\*  
 pairing\*  
   ELSE in nested IF statements\*  
   parentheses in arithmetic expressions\*  
   parentheses in subscripts and indexes\*  
   symbols in arithmetic expressions\*  
   symbols in compound conditions\*  
 paragraph\*  
   DATE-COMPILED\*  
   definition\*  
   FILE-CONTROL\*  
   I-O-CONTROL\*  
   OBJECT-COMPUTER\*  
   Procedure Division\*  
   PROGRAM-ID\*  
   SOURCE-COMPUTER\*  
   SPECIAL-NAMES\*  
 paragraph-name\*  
   qualification\*  
   reference format\*  
   rules for forming\*  
 parameters for OCR processing  
 parameter list 76,77,79  
 parentheses\*  
   arithmetic expressions\*  
   conditions\*  
   in statement formats 23  
   PICTURE clause\*  
   punctuation rules\*  
   subscripting and indexing\*  
 parity checking\*  
 PARSTD option 33  
 partial list of prime numbers 105  
 PASSWORD clause, VSAM 130,134,\*  
 PAUSE control statement 34

PBL (procedure block locator) 66  
 pence\*  
 PERFORM statement 205,\*  
   CALL statement\*  
   debug packets\*  
   declarative section\*  
   description and formats\*  
   flowcharts\*  
   in a segmented program 93,\*  
   sort procedures\*  
 period\*  
   and comma exchanged\*  
   data description entry\*  
   division header\*  
   END DECLARATIVES\*  
   fixed-point numeric literals\*  
   floating-point numeric literals\*  
   in statement formats 23  
   paragraph-name\*  
   PICTURE clause\*  
 permanent segment 89  
 permissible  
   comparisons\*  
   moves\*  
   specifications for the first eight bytes  
   of the actual key 102  
   symbol pairs\*  
 phase  
   definition of 11  
   origin point 41  
 PHASE control statement 41  
   generated by compiler for  
   Segmentation 90,92,93  
   using overlay 83  
 phase execution 18  
   output 72-73  
 phase map 71  
 Physical Input/Output Control System  
 (PIOCS) 98  
 physical file, definition\*  
 physical record\*  
   definition\*  
   size specification\*  
 PICTURE clause 196-197  
   allowable symbols\*  
   ASCII considerations\*  
   categories of data\*  
   character string\*  
   classes of data\*  
   editing\*  
   precedence of symbols in\*  
   repetition of symbols\*  
 PIOCS 98  
 planning the libraries 45  
 plus symbol\*  
   arithmetic expressions\*  
   collating sequence\*  
   indexing\*  
   PICTURE clause  
     description\*  
     external floating-point items\*  
     fixed insertion editing\*  
     floating insertion editing\*  
     numeric edited items\*  
     precedence in\*  
     sterling items\*  
 SIGN clause\*  
 unary operator\*

PMAP option 37  
 pocket select characters\*  
 positioning a file\*  
 positive operand in sign condition\*  
 positive value\*  
 possible specifications for X'ss' in the  
   ASSGN control statement 25  
 pound-report-string\*  
 pound-separator-string\*  
 pound sign\*  
 pre-DTF switch 155  
 prefixes 193  
 preparing COBOL programs for processing 19  
 prime area 118  
 prime numbers 103,104,105  
 print line size for report\*  
 PRINT-SWITCH\*  
 printer spacing subroutine 289  
 priority numbers 89,90  
   ALTER statement\*  
     called programs 90,\*  
     description 89,90,\*  
     information for use 89,90,\*  
     PERFORM statement 93,\*  
     section header\*  
     segment limit 91,\*  
 private core image library 55,18,32  
   and Linkage Editor 40,300  
 private libraries 55  
 private relocatable library 42,43,55  
 problem program area 18  
 procedural code (OCR) 321  
 procedure block 203  
 procedure block locator (PBL) 66  
 procedure branching statements\*  
   ALTER statement\*  
   EXIT statement\*  
   GO TO statement\*  
   PERFORM statement\*  
   STOP statement\*  
 Procedure Division\*  
   considerations for VSAM\*  
   content\*  
   copying\*  
   definition\*  
   Diskette unit (3540) file  
     considerations 124  
   header 76  
   modularizing 202  
   organization\*  
   Report Writer considerations\*  
   Sort considerations\*  
   statement list\*  
   statements 204-205,\*  
   sterling considerations\*  
   structure\*  
   table handling considerations\*  
   USING option on the division header\*  
 procedure library 53  
 procedure-name  
   arguments 79  
   definition\*  
   table 298  
 procedures, Declarative\*  
 processing 3540 Diskette unit  
   files 123-125  
 processing  
   capabilities summary, VSAM\*  
   COBOL files on mass storage devices 97  
   direct files 100-101  
   indexed files 121  
   programs 11  
   sequential files 99  
     ASCII tape files 177-178  
     tapes from the 3886 OCR, Model 2 326  
 PROCESSING MODE clause\*  
 Processor, Job Control 11,22  
   options 32-33  
 program, control 11  
 program-control cards 231  
 Program Global Table (PGT) 66  
 PROGRAM-ID paragraph  
   and program linkage 75-76  
   and Segmentation 91-92  
 Program Loader, Initial (IPL) 11  
 program-name\*  
 program status, recording 262  
 program switches 35  
   Communication Region 301  
 program termination\*  
 program versions, identification 247  
 programmer logical units 19,20  
 programming techniques 191-206  
   Data Division 193-202  
   Environment Division 192  
   general considerations 191  
   Procedure Division 202-206  
 programs  
   application 12  
   IBM-supplied 12  
   processing 11  
   system service 11  
 PSERV, procedure library 54  
 punctuation character\*  
  
 quadruple spacing of source program  
   listing\*  
 qualification\*  
   condition-names\*  
   description\*  
   index-names\*  
   names\*  
   subscripts\*  
 qualifier connective, definition\*  
 qualifier, definition\*  
 quotation mark option\*  
   default\*  
   nonnumeric literals\*  
   program-name\*  
 QUOTE (QUOTES) figurative constant\*  
 QUOTE option 37  
 quotient\*  
  
 random access\*  
   ACCESS MODE clause\*  
   CLOSE statement\*  
   definition\*



direct files 101,\*  
 error processing\*  
 indexed files 121-122,\*  
 READ statement\*  
 REWRITE statement\*  
 SEEK statement\*  
 WRITE statement\*  
 random file processing\*  
   effect of CLOSE options\*  
   function of a read\*  
 randomizing  
   for the 2311 Disk Drive 116  
   for the 2321 Data Cell Drive 117  
 randomizing techniques 102-106  
   sample programs 107-110,112-115  
 range of a PERFORM statement\*  
 range of values\*  
   condition-name\*  
   priority numbers\*  
   sequence numbers on DELETE card\*  
 RCE (read column eliminate) processing\*  
 RD\*  
 READ INTO statement 205  
 READ NEXT statement (VSAM) 140  
 READ statement\*  
   description and format\*  
   error processing\*  
 READ statement, VSAM 140,\*  
   description and formats\*  
   entry-sequenced file considerations 140  
   key-sequenced file considerations 140  
   processing capabilities\*  
 READ statement subroutines 289  
 reading backwards\*  
 reading nonstandard labels\*  
 READY/RESET TRACE statement\*  
 READY TRACE statement 247-248  
 receiving data item\*  
   justification\*  
   MOVE statement\*  
 record\*  
   description\*  
   level number\*  
   naming\*  
   slack bytes\*  
   RECORD CONTAINS clause 194  
   description and format\*  
   Report Writer\*  
   Sort\*  
 RECORD CONTAINS clause, VSAM\*  
 record description (OCR) 320  
 record description entry\*  
 record formats 179-188  
   format F 179  
   format S 183-188  
   format U 180  
   format V 180-183  
   VSAM files 136  
 RECORD KEY clause 122  
 RECORD KEY clause, VSAM\*  
   description and format\*  
   START statement\*  
 record length for sort records\*  
 record size default\*  
   ACCEPT statement\*  
   DISPLAY statement\*  
 report writer\*  
 record zero (R0) 99  
 recording capacities of mass storage  
 devices 97  
 recording mode\*  
   ASCII considerations\*  
   default\*  
   specification\*  
   types\*  
 RECORDING MODE clause  
   description and format\*  
   ASCII considerations\*  
 RECORDING MODE clause invalid, VSAM\*  
 recording program status 262  
 REDEFINES clause 194,\*  
   APPLY CORE-INDEX option\*  
   description and format\*  
   position when used\*  
   SYNCHRONIZED clause\*  
   VALUE clause\*  
 redefining subscripts 194  
 reference format\*  
 references, unresolved external 71  
 register assignment 66  
 register use for linkage 78  
 regrouping data items\*  
 relation character\*  
 relation conditions\*  
 relational-operators\*  
 relationship of PICTURE to storage  
   allocation 202  
 relative indexing\*  
 relative LINE clause\*  
 RELEASE statement in sort\*  
 relocatable library 46-47  
   cataloging a module 46  
   directory 46  
   INCLUDE statement 42  
   maintenance functions 46  
   private 42,43,55  
 relocating loader feature 43  
 relocation factor 39  
 remainder, definition\*  
 REMARKS paragraph\*  
 RENAMES clause\*  
 renaming\*  
   data items\*  
   logical records\*  
 REP statement 51,52  
 repetition of symbols in PICTURE\*  
 replacing source statements in a book 51  
 REPORT clause 211  
 REPORT clause invalid, VSAM\*  
 report, description\*  
 report description entry\*  
 report file, definition\*  
 REPORT FOOTING report group\*  
 report group description entry\*  
 report groups\*  
 REPORT HEADING report group\*  
 report page format effect on  
   LINE-COUNTER special register\*  
   PAGE-COUNTER special register\*  
   PAGE LIMIT clause\*  
 Report Section\*  
   content\*  
   COPY statement\*  
   formats\*  
   structure\*  
   VALUE clause\*

Report Writer Feature 211-215  
     Data Division considerations\*  
     Procedure Division considerations\*  
 Report Writer routines, generation of 215  
 report-name\*  
 Report Writer tables 299  
 reposition tape subroutine 296  
 required words for formats\*  
 RERUN clause  
     ASCII considerations\*  
     and RSTRT control statement 262-263,34  
     and Sort Feature 210  
     subroutine 293  
 RESERVE clause 192  
 RESET control statement 34-35,25  
 RESET TRACE statement 248  
 restarting a checkpointed  
     program 262-263,34  
 retrieving a book from the source statement  
     library 49  
     BASIS card 57  
     COPY statement 49,57  
     modifying using INSERT and DELETE  
         cards 57,58  
 retrieving a direct file 100  
     sample job deck 304  
 retrieving an indexed file 121  
     sample job deck 306  
 retrieving a program phase 45-46  
 retrieving a VSAM file 143  
 return code\*  
     multi-volume files\*  
     sort\*  
 return from sort\*  
     input procedure\*  
     output procedure\*  
 RETURN statement in sort\*  
 reversed reading of a file\*  
 rewinding a tape file\*  
 REWRITE statement\*  
     description and format\*  
     error processing\*  
 REWRITE statement, VSAM 139,\*  
     description and format\*  
     entry-sequenced file considerations 139  
     key-sequenced file considerations 140  
     processing capabilities\*  
 REWRITE statement subroutines 289  
 rewriting mass storage files\*  
 right justification\*  
 rolling counters forward\*  
 root phase 18  
     in overlay structure 81  
 root phase overlay 18  
 root segment 90,91,92  
     including subprograms in 90  
 ROUNDED option in arithmetic statements\*  
     ADD\*  
     COMPUTE\*  
     description\*  
     DIVIDE\*  
     MULTIPLY\*  
     SUBTRACT\*  
 rounding\*  
 RSTRT control statement 34,262-263  
 rules for the SET statement 223  
 R0 (record zero) 99  
  
 S, in PICTURE clause\*  
 S-mode records 183-188  
 SAME clause\*  
 SAME clause, VSAM\*  
 sample job decks 303-306  
 sample label and file extent information  
     for mass storage files 30  
 sample linkage routines used with a calling  
     subprogram 80  
 sample linkage routines used with a lowest  
     level subprogram 81  
 sample logical unit assignments 20  
 sample OCR program 327-331  
 sample programs\*  
     creation of a direct file 106-115,\*  
     creation of an indexed file\*  
     OCR 327-331  
     report writer\*  
     retrieval of a direct file\*  
     retrieval of an indexed file\*  
     sort\*  
     table handling\*  
 sample program output 265-278  
 sample program with symbolic debugging  
     features 235-246  
 save area 79  
 scaling, effect on rounding\*  
 scaling position character (P)\*  
 scientific decimal item\*  
 SEARCH ALL statement 221  
 SEARCH statement 224-225  
     subroutine 294  
 section header\*  
 section-name\*  
 SEEK statement\*  
 segment classification 89,\*  
 segment descriptor field 184  
 segment limit 91  
 Segmentation Feature 89-94  
     ALTER statement\*  
     called programs 90,\*  
     classifying segments 89,\*  
     control of 89,\*  
     fixed portion 91,\*  
     GO TO statement\*  
     independent segments 89,90,\*  
     operation 90  
     output from a segmented program 91  
     overlayable fixed segments 89,\*  
     PERFORM statement 93,\*  
     permanent segments 89,\*  
     priority numbers 89,\*  
     program organization\*  
     restrictions on program flow\*  
     segment limit 89,\*  
     with sort 93  
 segmentation subroutine 293  
 segments 89-90  
 SELECT clause  
     ASSGN control statement 28  
     COPY statement\*  
     DLBL control statement 28  
     EXTENT control statement 28-30  
     formats\*  
     GIVING option of SORT statement\*  
     programming technique 181  
     TLBL control statement 27  
     VOL control statement 30

SELECT OPTIONAL clause 24  
 semicolon\*  
   data description entry\*  
   source program\*  
 sentence\*  
   conditional\*  
   description\*  
   termination\*  
 SEPARATE CHARACTER option of SIGN clause\*  
 separately signed numeric subroutine 291  
 separator\*  
 SEQ option 36  
 sequence-check source statements 36  
 sequence checking compilation default\*  
 sequence-number-field for copying\*  
 sequence number in a source program\*  
 sequence of job control statements 23  
 sequential access 99,\*  
   ACCESS MODE clause 99,\*  
   ACTUAL KEY clause 101,\*  
   APPLY WRITE-ONLY clause\*  
   ASSIGN clause\*  
   description 99,\*  
   direct files 100,\*  
   indexed files 121,\*  
   NOMINAL KEY clause\*  
   recording mode\*  
   RECORDING MODE clause\*  
   RESERVE clause\*  
   sequential files 97,99,\*  
   size of records\*  
 sequential data organization 97,\*  
 sequential disk and tape labels  
   subroutines 289  
 sequential files 97,99,\*  
 sequential multivolume files 99,\*  
 sequential organization (DTFSD) 99,97  
 serial search of a table\*  
 service function of Librarian 45  
 service programs, system 11  
 Services, Access Method 130  
 SET command 26,59  
 SET statement 221-222  
   description and formats\*  
   rules for 223  
   with index data items\*  
   with indexes\*  
 setting values for index-names\*  
 severity levels of diagnostic messages 67  
 sharing storage between files\*  
 shilling representation\*  
 short-precision internal floating-point  
   items\*  
 sign\*  
   class condition\*  
   internal representation\*  
   literals\*  
   MOVE statement\*  
   PICTURE clause\*  
   relation condition\*  
   SIGN clause\*  
   sign condition\*  
   sterling items\*  
   subscripts\*  
   unary operator\*  
 SIGN clause 196  
   ASCII considerations\*  
   class condition\*  
   description and format\*  
   sign condition\*  
   7-track tape, restriction when used as sort  
   work files 208  
   sign usage 196  
   significant characters for various  
   options 39  
   simple insertion editing\*  
   single digit level number\*  
   single-program mode 12  
   size considerations 297  
   SIZE ERROR option in arithmetic statements\*  
   SKIP1 59,191  
   SKIP2 59,191  
   SKIP3 59,191  
   slack bytes\*  
   slash ampersand (/&) 15  
   slash asterisk (/\*) 15  
   sort diagnostic messages 209  
   sort\*  
     ascending and descending sequence\*  
     ASCII considerations 178,\*  
     checkpoints\*  
     collating sequence\*  
     control of procedures\*  
     Data Division considerations\*  
     Environment Division considerations\*  
     file description entry\*  
     final phase\*  
     input phase\*  
     keys\*  
     modification of records\*  
     optimizing performance 208-209,\*  
     Procedure Division considerations\*  
     sample program\*  
     special registers 209,\*  
     work units\*  
   Sort/Merge Feature 207-210  
     intermediate storage required 208  
     linkage 209  
     machine requirements 300  
     in a multiphase environment 210  
     in a segmented program 93  
     obtaining an ASCII collating  
       sequence 178  
     performance, improving 208-209  
     products 207  
   sort-file\*  
     COPY statement\*  
     description entry\*  
     SELECT clause\*  
   sort input/output control statements 207  
   sort/merge interface subroutine 293  
   sort/merge job control requirements 207  
   sort/merge work files 208  
   sort keys, restriction 207,178  
   sorting an unlabeled tape file 306,303  
     sample job deck 306  
   SORT-RETURN 209  
   sort special registers 209,\*  
   SORT statement\*  
     description and format\*  
     GIVING option 207,\*  
     INPUT PROCEDURE option\*  
     OUTPUT PROCEDURE option\*  
     RELEASE statement\*  
     RETURN statement\*  
     USING option 207 \*

SORTWK1, SORTWK2 in sort\*  
 SOURCE clause\*  
 SOURCE-COMPUTER paragraph\*  
 source program\*  
   definition\*  
   reference format\*  
   size considerations\*  
   structure\*  
 source program library facility 49,\*  
   COPY statement 49,\*  
   extended 57,\*  
     BASIS 57,\*  
     DELETE and INSERT 57,\*  
 source statement library 48-52  
   cataloging a book 48  
   directory 48  
   maintenance functions 48  
   updating books 50  
 space allocation  
   EXTENT control statement 28-29  
   XTENT control statement 30  
 SPACEn option 37  
 space insertion\*  
 SPACE (SPACES) figurative constant\*  
 spacing of source program listing 191  
 spanned records 183-188  
   on directly organized files 187-188  
   on sequentially organized files 185-186  
   and Sort Feature 208  
 special characters\*  
   CURRENCY-SIGN clause\*  
   in formats\*  
 special considerations when using overlay  
   structures 81  
 special insertion editing\*  
 special level numbers\*  
 special-names, definition\*  
 SPECIAL-NAMES paragraph\*  
 special registers\*  
   report writer\*  
     LINE-COUNTER\*  
     PAGE-COUNTER\*  
     PRINT-SWTICH\*  
   sort\*  
     SORT-CORE-SIZE\*  
     SORT-FILE-SIZE\*  
     SORT-MODE-SIZE\*  
     SORT-RETURN 209,\*  
   system\*  
     COM-REG 301,\*  
     CURRENT-DATE 301,\*  
     NSTD-REELS 166,\*  
     TALLY\*  
     TIME-OF-DAY\*  
 specifications for the first eight bytes of  
   the actual key 102  
 stacked items in formats\*  
 standard data format\*  
 standard file labels  
   example of multiple volume  
     creation 166,168-169  
     mass storage 174-175  
     format 1 281-286  
   tape 162,163-164,279-280  
 STANDARD option of the LABEL RECORDS  
   clause\*  
 standard sequential file\*  
   APPLY WRITE-ONLY clause\*  
   ASSIGN clause\*  
   BLOCK CONTAINS clause\*  
   definition\*  
   effect of CLOSE options\*  
   error declarative capabilities\*  
   labels\*  
   reading\*  
   recording mode\*  
   RESERVE clause\*  
   writing\*  
 standard system procedures\*  
   error routines\*  
   label handling\*  
 START statement 122  
   description and format\*  
   error processing\*  
   indexed files\*  
   NOMINAL KEY\*  
 START statement, VSAM 141,\*  
   description and format\*  
   processing capabilities\*  
 STATE option  
   description 38,229  
   restriction with OPT 38,37,229  
   restriction with STXIT 38,37,229  
   restriction with SYMDMP 38,229  
 statement formats  
   job control 22-23  
   symbolic debug option 229  
 statement number option (see STATE option)  
 statement number subroutine 294  
 statistics 66  
 Status Key  
   OCR 322,323  
   VSAM 138-139,136,130  
 STD LABEL option 33  
 sterling currency\*  
 STOP RUN statement\*  
 STOP statement 72  
   subroutines 290  
 storage required for sort\*  
 structures of the actual key 102  
 STXIT option 37  
   restriction with SYMDMP 38,37  
   restriction with STATE 38,37,229  
 subordinate phases 18  
 subprogram interface (OCR) 321  
 subprogram linkage statements 75-77,\*  
   CALL 75,\*  
   ENTRY 76,\*  
   EXIT PROGRAM\*  
   GOBACK\*  
   STOP RUN\*  
   termination considerations\*  
   USING option\*  
 subprogram structures 75-87  
 subroutines 289-296  
 subscripts 217  
   condition-name\*  
   description and formats\*  
   qualification\*  
   qualifier\*  
   redefining 194  
   restrictions on use\*  
 substitution\*  
   comma for period\*  
   dollar sign\*  
 subtotalling in a report\*

SUBTRACT statement\*  
 CORRESPONDING option\*  
 description and formats\*  
 GIVING option\*  
 overlapping operands\*  
 ROUNDED option\*  
 SIZE ERROR option\*  
 subtraction operator\*  
 SUM clause\*  
 SUM counters 211  
 definition\*  
 GENERATE statement\*  
 INITIATE statement\*  
 RESET clause\*  
 resetting to zero\*  
 SUM routines 212-213  
 summary reporting\*  
 summation in a report\*  
 summing techniques 211  
 Supervisor 11  
 SUPMAP option 36  
 suppress spacing\*  
 suppressing messages  
 FLAGE option 36  
 NOERRS option 33  
 suppression of\*  
 library entry listing\*  
 report group printing\*  
 sequence checking\*  
 spacing in WRITE statement\*  
 zeros in PICTURE clause\*  
 suppression and replacement editing\*  
 suppression symbols\*  
 switch-status condition\*  
 description and format\*  
 system-names\*  
 SXREF option  
 description 37  
 example 67  
 suppressing 37  
 SYM option 33  
 symbol pairs\*  
 symbolic debug option (see SYMDMP option)  
 symbolic debugging features 229-246  
 FLOW option 229  
 STATE option 229  
 SYMDMP option 229-232  
 symbolic debugging subroutines 295  
 symbolic names  
 of input/output devices 19,20,21  
 of phases 42  
 symbolic portion of ACTUAL KEY\*  
 symbols\*  
 arithmetic expressions\*  
 floating-point literals\*  
 PICTURE clause\*  
 relation conditions\*  
 sterling currency formats\*  
 used to define compiler-generated  
 information 66  
 SYMDMP option  
 description 229-230,38  
 restriction with NOLINK and NODECK 38  
 restriction with OPT 37,229  
 restriction with STATE 38  
 restriction with STXIT 37,38,229  
 SYNCHRONIZED clause 200  
 synonyms 116  
 syntax checking 191,247  
 SYNTAX option 38  
 syntax rules  
 job control statements 22-23  
 symbolic debug control statements 230  
 SYSCLB 19,21,55  
 SYSIN 19,21  
 SYSIPT 19,21  
 SYSLNK 19,21,40  
 SYSLOG 19,21  
 SYSLSL 19,21,59  
 SYSOUT 19,21,24  
 SYSPCH 19,21  
 SYSRDR 19,21  
 on same device as SYSIPT 15,17,19  
 SYSRES 19,21,55  
 SYSRLB 19,21,55  
 SYSSLB 19,21,55  
 system closing conventions\*  
 system and size considerations 297-300  
 system libraries, relative location 56  
 system logical units 19  
 system message identification codes 73  
 system-name\*  
 ASSIGN clause\*  
 RERUN clause\*  
 Sort Feature\*  
 system-name restriction for RERUN on a sort  
 file 210  
 system output 73  
 system service programs 11-12  
 system special registers\*  
 SYS000 through SYS221 19,21  
 SYS005 38,232,297  
 table element 217  
 Table Handling Feature 217-225  
 ascending/descending sequence\*  
 Data Division considerations\*  
 indexing 217,\*  
 Procedure Division considerations\*  
 sample program\*  
 subscripting 217,\*  
 tables 298-299  
 taking a checkpoint 262  
 TALLY special register\*  
 ACCEPT statement\*  
 description\*  
 DISPLAY statement\*  
 EXAMINE statement\*  
 SOURCE clause\*  
 subscript\*  
 SUM clause\*  
 tape device, error information\*  
 tape file labels 279-280  
 tape pointer subroutine 290  
 table and sequential disk labels  
 subroutines 289  
 Task Global Table (TGT) 65  
 TERMINATE statement  
 termination of\*  
 execution\*  
 main programs\*  
 report processing\*  
 sort processing\*  
 subprograms\*

test conditions\*  
 testing a program selectively 250  
 testing additions and changes to  
   programs 250-251  
 THEN  
   used in IF statement\*  
   used in sentences\*  
 THRU reserved word\*  
   and PERFORM statement\*  
   and THROUGH, equivalence of\*  
   and VALUE clause\*  
 TIME-OF-DAY special register\*  
 TLBL control statement 26  
   standard tape file labels 163,164  
 TPLAB control statement 30  
 TRACE statement 247  
 track 97  
 track addresses, converting  
   in a COBOL source program  
     relative to actual 103,106  
   in EXTENT control statement  
     actual to relative 29  
     relative to actual 29  
 track addressing 99-100  
   actual 101-110,116-117  
     sample program 106-110  
   relative 101-106  
     sample program 111-115  
 TRACK-AREA clause\*  
 track formats for direct-access storage  
   devices 287-288  
 track index 119  
 trailer labels\*  
 transfer of control\*  
   ALTER statement\*  
   CALL statement\*  
   calling and called programs\*  
   DECLARATIVES\*  
   end of series of procedures\*  
   EXIT statement\*  
   GO TO statement\*  
   GO TO MORE LABELS\*  
   GOBACK statement\*  
   operating system\*  
   operator\*  
   PERFORM statement\*  
   RELEASE statement\*  
   RETURN statement\*  
   segments, among\*  
   SORT feature\*  
 TRANSFORM statement 206  
   ASCII considerations\*  
   description and format\*  
   subroutine 294  
 transient area 92  
 transient subroutines 295-296  
 translation from ASCII to EBCDIC 178  
 transmission errors\*  
 triple spacing\*  
 TRUNC option 37  
 truncation\*  
   alphanumeric and alphanumeric items\*  
   arithmetic operations\*  
   of buffers\*  
   of COMPUTATIONAL items 37  
   floating insertion editing\*  
   MOVE statement\*  
   numeric items\*  
  
 TYPE clause\*  
 type codes used in SYMDMP output 234  
 types of jobs 14  
  
 U-mode records 180  
 undefined records 180  
 unique names\*  
   indexing\*  
   qualification\*  
   subscripting\*  
 unit, definition\*  
   in formats\*  
   in storage\*  
 unit record volume\*  
   CLOSE options\*  
   description\*  
   error information\*  
   list\*  
 unlabeled files 175  
   example of multifile volume  
     processing 166,168-169  
   example of multivolume file  
     creation 167,172-173  
     sorting 306  
 unnumbered messages, COBOL object-time 317  
 unresolved external references 71  
 unsigned numeric operands\*  
   considered positive\*  
   relation condition\*  
   sign condition\*  
 unsigned items 196-197  
 UPDATE function 50-53  
   ADD statement 50,51,52  
   control statement placement 52  
   DEL statement 51  
   END statement 51  
   invalid operand defaults 52  
   REP statement 51  
   UPDATE statement 50-51  
 UPDATE statement 50-52  
 updating a book in the source statement  
   library 50-52  
 updating a direct file 101  
   sample job deck 304  
 updating an indexed file 120,121  
   sample job deck 306  
 upper-case letters in statement formats 22  
 UPSI byte 37  
 UPSI control statement 35  
   Communication Region 301  
 UPSI switches 301  
 UPSI-0 through UPSI-7 35  
 USAGE clause 197  
   alteration by redefining\*  
   ASCII considerations\*  
   default\*  
   description and format\*  
   index data items\*  
   internal representations\*  
   use of coding form\*  
   use of symbolic debugging features 229  
 USE statement\*  
   description and formats\*  
   error processing\*  
   label processing\*  
   report writer\*

- use of SUM 211
- user-created libraries\*
- user error procedures\*
- user header label (UHL)\*
- user labels
  - mass storage files 174,175
  - tape files 162,165
- user program status indicator bits\*
- user program switch indicators 301
- user standard labels 162
- USING option
  - of CALL statement 75
  - of ENTRY statement 76
  - on Procedure Division header 76
- using ISAM programs to process VSAM files 144
- using the Report Writer feature 211-216
- using the Segmentation feature 89
- using the Sort/Merge feature 207
- using the START verb (VSAM) 141
- USRLABEL option 33
- utility data sets
  - required by compiler 297
  - required by sort program 300,208
- UTL (user trailer label)\*
  
- V, in PICTURE clause\*
- V-mode records 180-183
- valid mass storage devices\*
- VALUE clause\*
  - condition-names\*
  - description and formats\*
  - example\*
  - report writer data items\*
  - sterling items\*
- VALUE OF clause\*
- variable-length records 180-183
- variable length table\*
- various options, significant characters for 39
- VERB option 39
- vertical positioning of a printed line\*
- Virtual Storage Access Method (see VSAM)
- VSAM catalog 128
- VSAM messages 130
- VOL control statement 30
- volume and file portability (VSAM) 130
- volume labels
  - mass storage 174
  - tape 162
- volume positioning, CLOSE statement\*
- volume switch\*
  - CLOSE options\*
  - label processing\*
  - READ statement\*
  - WRITE statement\*
- VSAM (Virtual Storage Access Method) 127-144
  - Access Method Services 130
  - DEFINE command 130-131
    - defining a VSAM data space:
      - DEFINE SPACE 132
    - defining a VSAM file:
      - DEFINE CLUSTER 133
    - defining a VSAM master catalog:
      - DEFINE MASTERCATALOG 131-132
  - functional commands 130
  - catalog 128
  - COBOL language usage with VSAM 141
  - COBOL VSAM control blocks 145
  - converting non-VSAM files to VSAM files 144
  - creating a VSAM file 141
    - examples of 141-144
    - language statements required for 141
  - current record pointer 134
  - data access 128
  - Data Division considerations\*
    - data organization 128,129
  - DEFINE command 130-131
  - DELETE statement 141
  - device support 130
  - entry-sequenced files 127-128
  - error handling 136
    - file status values
      - at OPEN 138
      - at action request time 138
  - error processing 130
    - status key 138-139,136,130
  - file and volume portability 130
  - file control block (FCB) 148
  - FILE-CONTROL paragraph\*
  - file information block (FIB) 146
  - file organization 127
  - file processing techniques 134
  - file status values
    - at OPEN 138
    - at action request time 138
  - initial loading of records into a file 137
  - I-O-CONTROL paragraph\*
  - job control language for a VSAM file 144
  - key-sequenced files 127
  - messages 130
  - opening a VSAM file 137
  - overall description\*
  - Procedure Division considerations\*
    - processing capabilities summary\*
  - reading records on a VSAM file 140
    - entry-sequenced file considerations for the READ statement 140
    - key-sequenced file considerations for the READ statement 140
    - READ NEXT statement 140
    - READ statement 141
  - record formats for VSAM files 136
  - retrieving a VSAM file 143
  - rewriting records on a VSAM file 139-140
    - entry-sequenced file considerations for the REWRITE statement 139-140
    - key-sequenced file considerations for the REWRITE statement 140
  - security 130
  - service programs (Access Method Services) 130
  - status key 138-139,136,130
  - subroutines 290
  - using ISAM programs to process VSAM files 144
  - using the START verb 141
  - valid mass storage devices\*

writing records into a VSAM file 139  
 entry-sequenced file considerations  
 for the WRITE statement 139  
 key-sequenced file considerations  
 for the WRITE statement 139

weak external reference 71  
 WHEN-COMPILED special register 26,7,\*  
 WITH CODE clause 213-214  
 word\*  
 characters used in\*  
 continuation of\*  
 definition\*  
 separators\*  
 types\*  
 word boundary\*  
 work files  
 required by compiler 297,38  
 required by sort program 300,208  
 Working-Storage Section 194,\*  
 boundary alignment\*  
 condition-name entries\*  
 content\*  
 COPY statement\*  
 data item description entry\*  
 formats\*  
 level-numbers in\*  
 locating in a dump 194  
 naming data\*  
 record description entry\*  
 renaming entries in\*  
 structure\*  
 used in error processing\*  
 use of FILLER\*  
 values of items\*  
 Working-Storage Section and OCR processing\*  
 WRITE FROM statement 205  
 WRITE-ONLY option of APPLY clause\*  
 WRITE statement\*  
 combined function processing\*  
 description and formats\*  
 error processing\*  
 multivolume sequential files\*  
 system-name organization field\*  
 WRITE statement, VSAM 139,\*  
 description and format\*  
 entry-sequenced file considerations 139  
 key-sequenced file considerations 139  
 processing capabilities\*  
 WRITE statement subroutines 289  
 WRITE-VERIFY option of APPLY clause\*  
 writing user labels\*

X, in PICTURE clause\*  
 XREF dictionary 67  
 XREF option 33  
 X'ss' parameter of ASSGN job control  
 statement  
 possible specifications for 25  
 XTENT control statement 30

Z, in PICTURE clause\*  
 ZERO (ZEROES, ZEROS) figurative constant\*  
 description\*  
 MOVE statement\*  
 relation condition\*  
 replacing numeric literal\*  
 TRANSFORM statement\*  
 zero divisor\*  
 zero insertion\*  
 zero operand\*  
 DIVIDE statement\*  
 internal floating-point items\*  
 relation condition\*  
 sign condition\*  
 zero, simple insertion editing\*  
 zero suppression and replacement editing\*  
 zone bits, external decimal items\*  
 zoned decimal format\*  
 ZWE option 37

01-49 level numbers\*  
 6, in sterling PICTURE clause\*  
 66 level number\*  
 7, in sterling PICTURE clause\*  
 77 level number\*  
 7-track tape, restriction when used as sort  
 work files 208  
 8, in sterling PICTURE clause\*  
 88 level number\*  
 9, in PICTURE clause\*  
 alphanumeric edited items\*  
 description\*  
 numeric edited items\*  
 numeric items\*  
 precedence\*  
 sterling items\*  
 2319, 3211, 3330, 3410, 3420, 3505, 3525  
 support 7,\*  
 2560, 3504, 3881, 5425 support 7,\*  
 3203, 3340, 3540, 5203 processing 7,\*  
 3540 processing 123-125  
 3886 processing (see Optical Character  
 Reader)





IBM DOS/VS COBOL Compiler and  
Library Programmer's Guide  
SC28-6478-1

**Reader's  
Comment  
Form**


Your comments about this publication will help us to improve it for you. Comment in the space below, giving specific page and paragraph references whenever possible. All comments become the property of IBM.

Please do not use this form to ask technical questions about IBM systems and programs or to request copies of publications. Rather, direct such questions or requests to your local IBM representative.

If you would like a reply, please provide your name, job title, and business address (including ZIP code).

**Fold on two lines, staple, and mail.** No postage necessary if mailed in the U.S.A. (Elsewhere, any IBM representative will be happy to forward your comments.) Thank you for your cooperation.

Fold and Staple

  
First Class Permit  
Number 439  
Palo Alto, California

---

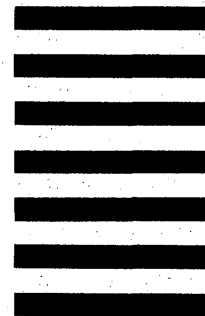
**Business Reply Mail**

No postage necessary if mailed in the U.S.A.

---

Postage will be paid by:

**IBM Corporation  
System Development Division  
LDF Publishing—Department J04  
1501 California Avenue  
Palo Alto, California 94304**



Fold and Staple



**International Business Machines Corporation  
Data Processing Division  
1133 Westchester Avenue, White Plains, New York 10604  
(U.S.A. only)**

**IBM World Trade Corporation  
821 United Nations Plaza, New York, New York 10017  
(International)**

## READER'S COMMENTS

**TITLE:** IBM DOS/VS COBOL  
Compiler and Library  
Programmer's Guide

**ORDER NO.** SC28-6478-0

Your comments assist us in improving the usefulness of our publications; they are an important part of the input used in preparing updates to the publications. All comments and suggestions become the property of IBM.

Please do not use this form for technical questions about the system or for requests for additional publications; this only delays the response. Instead, direct your inquiries or requests to your IBM representative or to the IBM Branch Office serving your locality.

Corrections or clarifications needed:

| <i>Page</i> | <i>Comment</i> |
|-------------|----------------|
|-------------|----------------|

Please include your name and address in the space below if you wish a reply.

Thank you for your cooperation. No postage necessary if mailed in the U.S.A.

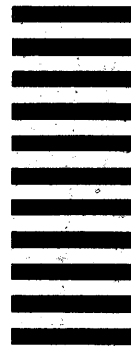
cut along this line

fold

fold

FIRST CLASS  
PERMIT NO. 33504  
NEW YORK, N.Y.

**BUSINESS REPLY MAIL**  
NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES



POSTAGE WILL BE PAID BY . . .

IBM CORPORATION  
1271 Avenue of the Americas  
New York, New York 10020

Attention: PUBLICATIONS

fold

fold



**International Business Machines Corporation**  
**Data Processing Division**  
**1133 Westchester Avenue, White Plains, New York 10604**  
**(U.S.A. only)**

**IBM World Trade Corporation**  
**821 United Nations Plaza, New York, New York 10017**  
**(International)**

IBM DOS/VS COBOL P.G. Printed in U.S.A. SC28-6478-1



International Business Machines Corporation  
Data Processing Division  
1133 Westchester Avenue, White Plains, New York 10604  
(U.S.A. only)

IBM World Trade Corporation  
821 United Nations Plaza, New York, New York 10017  
(International)