

**USER'S REFERENCE MANUAL (U)
FOR
COMPILER, MONITOR SYSTEM-2 (CMS-2)
FOR USE WITH AN/UYK-7 COMPUTER**

COMPILER, ASSEMBLER, AND INSTRUCTION REPERTOIRE

FLEET COMBAT DIRECTION SYSTEMS SUPPORT ACTIVITY

San Diego, California 92147

CHANGE NOTICE

5

**THIS CHANGE SHOULD BE INCORPORATED INTO THE
BASIC PUBLICATION. DESTROY SUPERSEDED PAGES.**

LIST OF EFFECTIVE PAGES

Insert latest changed pages; dispose of superseded pages in accordance with applicable regulations.

NOTE: On a changed page, the portion of the text affected by the latest change is indicated by a vertical line, or other change symbol, in the outer margin of the page.

Total number of pages in this manual is 681 consisting of the following:

Page No.	†Change No.	Page No.	†Change No.
Title	4	II-4-2.....	3
A - D.....	5	II-4-3.....	0
i.....	0	II-4-4 - II-4-6.....	3
ii Blank.....	0	II-4-7 - II-4-16.....	0
iii - v.....	0	II-4-17.....	5
vi.....	3	II-4-18.....	0
vii.....	5	II-4-18A - II-4-18B.....	3
viii.....	4	II-4-19 - II-4-20.....	0
viiiA.....	4	II-4-21 - II-4-22.....	5
viiiB Blank.....	4	II-4-23 - II-4-24.....	0
ix.....	4	II-4-25.....	5
x.....	5	II-4-26.....	0
xi.....	0	II-4-27.....	5
xii.....	3	II-4-28.....	2
xiii.....	2	II-4-29 - II-4-33.....	0
xiv.....	5	II-4-34.....	5
xv.....	0	II-4-35.....	0
xvi.....	2	II-4-36.....	3
xvii - xviii.....	4	II-4-37 - II-4-40.....	0
II-1-1 - II-1-9.....	0	II-4-41 - II-4-42.....	2
II-1-10 Blank.....	0	II-4-43 - II-4-59.....	0
II-2-1.....	4	II-4-60.....	3
II-2-2 - II-2-24.....	0	II-4-61 - II-4-62.....	2
II-2-25 - II-2-26.....	4	II-4-63.....	3
II-2-27.....	0	II-4-64.....	0
II-2-28 Blank.....	0	II-4-65.....	5
II-3-1 - II-3-3.....	0	II-4-66.....	0
II-3-4.....	3	II-4-67 - II-4-68.....	5
II-3-5.....	0	II-4-68A.....	5
II-3-6.....	1	II-4-68B Blank.....	5
II-3-7.....	3	II-4-69.....	5
II-3-8.....	0	II-4-70.....	0
II-4-1.....	0	II-4-71 - II-4-72.....	5

†Zero in this column indicates an original page.

LIST OF EFFECTIVE PAGES (contd)

Page No.	†Change No.	Page No.	†Change No.
II-5-1 - II-5-2	4	II-6-18 - II-6-22	0
II-5-2A	4	II-6-23	2
II-5-2B Blank	4	II-6-24	0
II-5-3	3	II-6-25	1
II-5-4	4	II-6-26	0
II-5-4A	4	II-6-27	2
II-5-4B Blank	4	II-6-28	1
II-5-5	3	II-6-29 - II-6-30	0
II-5-6	1	II-6-31	1
II-5-7	2	II-6-32 - II-6-34	0
II-5-8 - II-5-10	0	II-6-35	3
II-5-11 - II-5-12	3	II-6-36 Blank	3
II-5-13	1	II-7-1 - II-7-3	0
II-5-14 - II-5-17	0	II-7-4 - II-7-5	5
II-5-18	5	II-7-6 - II-7-7	4
II-5-18A	5	II-7-8	5
II-5-18B Blank	5	II-7-9	0
II-5-19 - II-5-20	5	II-7-10	5
II-5-21	0	II-7-11	2
II-5-22	5	II-7-12	3
II-5-23 - II-5-25	0	II-7-12A - II-7-12B	4
II-5-26 - II-5-27	3	II-7-13 - II-7-16	0
II-5-28	0	II-7-17	2
II-5-29	5	II-7-18	0
II-5-30	0	II-7-19	2
II-5-31 - II-5-32	5	II-7-20 - II-7-21	0
II-5-33 - II-5-38	0	II-7-22	3
II-5-39 - II-5-40	3	II-7-23	5
II-5-41 - II-5-44	0	II-7-24	4
II-5-45 - II-5-46	3	II-7-24A	4
II-5-47	0	II-7-24B	5
II-5-48 - II-5-79	4	II-7-25 - II-7-26	3
II-5-80 Blank	4	II-7-27	0
II-6-1 - II-6-4	0	II-7-28	2
II-6-5 - II-6-7	2	II-7-29	1
II-6-8 - II-6-9	0	II-7-30	5
II-6-10	4	II-7-30A - II-7-30F	5
II-6-11 - II-6-15	0	II-7-31 - II-7-33	5
II-6-16	1	II-7-34 Blank	5
II-6-17	2	II-8-1 - II-8-2	0
		II-8-3	1
		II-8-4 - II-8-5	0

†Zero in this column indicates an original page.

LIST OF EFFECTIVE PAGES (contd)

Page No.	†Change No.	Page No.	†Change No.
II-8-6.	3	II-12-5	5
II-8-7 - II-8-11	0	II-12-6 - II-12-34	0
II-8-12 Blank	0	II-12-35 - II-12-50	1
II-9-1.	2	II-12-51 - II-12-54	0
II-9-2 - II-9-3	0	II-12-55	5
II-9-4.	2	II-12-56	0
II-9-5 - II-9-6	1	II-12-57 - II-12-68	1
II-9-7.	3	II-12-69 - II-12-72	0
II-9-8.	0	II-12-73 - II-12-78	1
II-9-9 - II-9-16	2	II-12-79 - II-12-82	0
II-9-17.	0	II-12-83	1
II-9-18 - II-9-21	2	II-12-84	5
II-9-22 Blank	0	II-12-85 - II-12-88	1
II-10-1 - II-10-2	3	II-12-89	5
II-10-3	0	II-12-90 - II-12-94	1
II-10-4 - II-10-5	5	II-12-95 - II-12-110	0
II-10-6 - II-10-10.	0	A-1	2
II-10-11	3	A-2	0
II-10-12 - II-10-13	0	A-3	5
II-10-14 Blank	0	A-4 Blank	5
II-11-1 - II-11-10.	0	B-1 - B-2	0
II-11-11 - II-11-12.	1	B-3	5
II-11-13 - II-11-14.	3	B-4 - B-6	0
II-11-14A - II-11-14B.	3	B-7	5
II-11-15 - II-11-22	0	B-8 - B-12	0
II-11-23	1	B-13 - B-14	5
II-11-24	5	B-15 - B-16	0
II-11-25	3	B-17 - B-18	5
II-11-26	5	B-19 - B-20	0
II-11-26A - II-11-26B.	3	B-21	5
II-11-27 - II-11-31.	0	B-22 - B-28	0
II-11-32 - II-11-49.	2	C-1	0
II-11-50	5	C-2 - C-6	3
II-11-51 - II-11-69.	2	D-1	5
II-11-70	5	D-2	2
II-11-71 - II-11-74.	2	E-1 - E-2	3
II-11-75 - II-11-76.	5	E-3 - E-4	5
II-11-77 - II-11-80.	2	E-4A	5
II-11-81	5	E-4B Blank	5
II-11-82 - II-11-83.	2	E-5 - E-6	5
II-11-84	5	E-7	4
II-12-1 - II-12-4	0	E-8	3
		E-9 - E-10	4
		E-10A	4
		E-10B	5
		E-11	5
		E-12 - E-14	3
		E-15 - E-17	5

†Zero in this column indicates an original page.

LIST OF EFFECTIVE PAGES (contd)

Page No.	†Change No.	Page No.	†Change No.
E-18 - E-19	2		
E-20 - E-24	3		
F-1	0		
F-2	5		
G-1 - G-8	0		
H-1	3		
H-2 Blank	0		
I-1 - I-3	2		
I-4	5		
I-5 - I-24	2		
I-25	5		
I-26 - I-30	2		
I-31	5		
I-32 - I-34	2		
I-35	5		
I-36 Blank	5		
K-1 - K-15	5		
K-16 Blank	5		

†Zero in this column indicates an original page.

FOREWORD

This is Revision 1 of Volume II of the two volume User's Reference Manual for Compiler Monitor System (CMS-2) for use with AN/UYK-7 Computer. This volume contains a description of the languages recognized by the Compiler and Assembler, including a comprehensive description of macro generation. It also contains a complete functional description of the AN/UYK-7 Computer Instruction Repertoire. (Volume I describes the Monitor, Loader, Librarian, Peripheral Utilities, and System Operation.)

Revision 1 incorporates changes 1, 2, and 3 to the original document (NAVSHIPS 0967-028-0060). The reader will find the changed pages clearly indicated throughout the volume.

A document related to Volume II is the Compiler Monitor System-2 (CMS-2) User's Reference Manual, Volume I, M-5012. This document describes the CMS-2 language as it is used in the preparation of programs for the CP-642B-hosted CMS-2/XCMS-2 compilers.

RECORD OF CHANGES

CHANGE NO.	DATE	TITLE OR BRIEF DESCRIPTION	ENTERED BY

TABLE OF CONTENTS

<u>Paragraph</u>	<u>Title</u>	<u>Page</u>
SECTION 1		
INTRODUCTION		
1.1	Purpose and Scope	II-1-1
1.2	Applicable Documents	II-1-2
1.3	System Capabilities	II-1-3
1.3.1	Hardware Requirements	II-1-3
1.3.2	Software Components	II-1-5
1.3.2.1	Monitor	II-1-6
1.3.2.2	Object Code Loader	II-1-6
1.3.2.3	Librarian	II-1-7
1.3.2.4	Peripheral Utilities	II-1-7
1.3.2.5	Compiler	II-1-7
1.3.2.6	Assembler	II-1-8
1.3.2.7	System Tape Generator	II-1-8
1.4	System Operation	II-1-8
1.4.1	System Load and Initiation	II-1-8
1.4.2	Standard Input Processing	II-1-9
1.4.3	Standard Output Processing	II-1-9
SECTION 2		
INTRODUCTION TO THE CMS-2 LANGUAGE		
2.1	Major Features of CMS-2	II-2-1
2.2	Program Structure	II-2-1
2.2.1	Organization and Classification of	
	Identifiers	II-2-3
2.2.1.1	Forward and Backward References	II-2-3
2.2.1.2	Local and Global Definitions	II-2-3
2.2.1.3	External References and Definitions	II-2-4
2.2.2	CMS-2 Elements	II-2-5
2.2.2.1	System Data Designs	II-2-7
2.2.2.2	System Procedures	II-2-8
2.2.2.2.1	Local Data Designs	II-2-9
2.2.2.2.2	Procedures	II-2-9
2.2.3	Range of Identifiers	II-2-9
2.3	Declarative Statements	II-2-10
2.3.1	Program Structure Declaratives	II-2-10
2.3.1.1	Procedure Structure Declaratives and	
	Linking	II-2-13
2.3.1.2	Reentrant System Procedures	II-2-14
2.3.2	Data Declarations	II-2-16
2.3.2.1	Switches	II-2-17
2.3.2.2	Variables	II-2-17
2.3.2.3	Tables	II-2-17

TABLE OF CONTENTS (Continued)

<u>Paragraph</u>	<u>Title</u>	<u>Page</u>
SECTION 2 (Continued)		
2.3.2.4	Arrays	II-2-20
2.3.3	Compiler Directive Declaratives	II-2-20
2.4	Dynamic Statements	II-2-24
2.4.1	Expressions	II-2-24
2.4.2	Statement Operators	II-2-24
2.4.3	Special Operators	II-2-25
2.5	High-Level Input/Output Statements	II-2-26
2.6	Program Debug Facilities	II-2-27
SECTION 3 BASIC DEFINITIONS		
3.1	CMS-2 Alphabet	II-3-1
3.2	Symbols	II-3-1
3.2.1	Operators	II-3-2
3.2.2	Identifiers	II-3-2
3.2.2.1	Statement Label	II-3-3
3.2.3	Constants	II-3-3
3.2.3.1	Numeric Constants	II-3-3
3.2.3.2	Hollerith Constant	II-3-4
3.2.3.3	Status Constants	II-3-5
3.2.3.4	Boolean Constants	II-3-6
3.3	Delimiters	II-3-6
3.4	Statements	II-3-6
3.5	Comments	II-3-6
3.5.1	Special Comments	II-3-7
3.6	Source Card Format	II-3-8
SECTION 4 DECLARATIVES		
4.1	Program Structure Declaratives	II-4-1
4.1.1	System Declarative (SYSTEM)	II-4-2
4.1.2	Head Declarative (HEAD)	II-4-2
4.1.3	End Head Declarative (END-HEAD)	II-4-3
4.1.4	System Data Design Declarative (SYS-DD)	II-4-4
4.1.5	End System Data Design Declarative (END-SYS-DD)	II-4-4
4.1.6	System Procedure Declarative (SYS-PROC)	II-4-5

TABLE OF CONTENTS (Continued)

<u>Paragraph</u>	<u>Title</u>	<u>Page</u>
SECTION 4 (Continued)		
4.1.7	Reentrant System Procedure Declarative (SYS-PROC-REN)	II-4-6
4.1.8	Local Data Design Declarative (LOC-DD)	II-4-6
4.1.9	Local Data Design Declarative (END-LOC-DD)	II-4-7
4.1.10	Automatic Data Design Declarative (AUTO-DD)	II-4-7
4.1.11	End Automatic Data Design Declarative (END-AUTO-DD)	II-4-8
4.1.12	Procedure (PROCEDURE) and End Procedure (END-PROC) Declaratives	II-4-8
4.1.13	Function (FUNCTION) and End Function (END-FUNCTION) Declaratives	II-4-11
4.1.14	End System Procedure Declarative (END-SYS-PROC)	II-4-13
4.1.15	End System Declarative (END-SYSTEM)	II-4-14
4.2	Data Declarations	II-4-14
4.2.1	Variable Declaration (VRBL)	II-4-15
4.2.1.1	Parameter Declaration (PARAMETER)	II-4-18A
4.2.2	Table (TABLE) Declaration	II-4-19
4.2.3	Field (FIELD) Declaration	II-4-25
4.2.4	Item-Area (ITEM-AREA) Declaration	II-4-30
4.2.5	Subtable (SUB-TABLE) Declaration	II-4-31
4.2.6	Like-Table (LIKE-TABLE) Declaration	II-4-34
4.2.7	End-Table (END-TABLE) Declaration	II-4-35
4.2.8	Packing Rules	II-4-37
4.2.8.1	No Packing (NONE)	II-4-38
4.2.8.2	Medium Packing (MEDIUM)	II-4-38
4.2.8.3	Dense Packing (DENSE)	II-4-39
4.2.9	Overlay (OVERLAY) Declaration	II-4-40
4.2.10	Data Referencing	II-4-44
4.2.10.1	Table Referencing	II-4-44
4.2.10.1.1	Whole Table Referencing	II-4-45
4.2.10.1.2	Item Referencing	II-4-45
4.2.10.1.3	Field Referencing	II-4-47
4.2.10.1.4	Item-Area Referencing	II-4-49
4.2.11	Transfer Declaratives (Switches)	II-4-50
4.2.11.1	Statement Switch (SWITCH) Declaratives	II-4-50
4.2.11.1.1	Index Switch	II-4-50
4.2.11.1.2	Item Switch	II-4-53
4.2.11.2	Procedure Switch (P-SWITCH) Declaratives	II-4-55
4.2.11.2.1	Index Procedure Switch	II-4-55
4.2.11.2.2	Double Procedure Switch	II-4-56
4.2.11.2.3	Item Procedure Switch	II-4-58

TABLE OF CONTENTS (Continued)

<u>Paragraph</u>	<u>Title</u>	<u>Page</u>
SECTION 4 (Continued)		
4.2.11.3	Switch Referencing	II-4-59
4.2.12	Local Indexes	II-4-59
4.2.13	Data (DATA) Declaration	II-4-61
4.3	Control Declaratives	II-4-62
4.3.1	Mode (MODE) Declaration	II-4-63
4.4	System Linkage	II-4-64
4.4.1	External Definition (EXTDEF) Operator	II-4-65
4.4.2	External Reference (EXTREF) Operator	II-4-66
4.4.3	Transient Reference (TRANSREF) Operator	II-4-66
4.4.4	Local Definition (LOCDEF) Operator	II-4-67
4.4.5	Applications of EXTDEF, EXTREF and TRANSREF	II-4-68
SECTION 5 DYNAMIC STATEMENTS		
5.1	Expressions	II-5-2
5.1.1	Arithmetic Expressions	II-5-2
5.1.1.1	Fractional Significance in Fixed- Point Operations	II-5-3
5.1.2	Relational Expressions	II-5-5
5.1.3	Boolean Expressions	II-5-6
5.1.4	Literal Expressions	II-5-8
5.2	Functional Modifiers	II-5-9
5.2.1	Absolute Value (ABS) Modifier	II-5-10
5.2.2	Bit (BIT) Modifier	II-5-10
5.2.3	Character (CHAR) Modifier	II-5-12
5.2.4	Count (CNT) Number of Bits	II-5-13
5.2.5	Core Address (CORAD) Modifier	II-5-14
5.2.6	File Position (FIL) Modifier	II-5-14
5.2.7	Record Position (POS) Modifier	II-5-15
5.2.8	Record Length (LENGTH) Modifier	II-5-15
5.3	Procedure Linking	II-5-15
5.3.1	Procedure Call	II-5-16
5.3.2	Function Call	II-5-18
5.3.3	Return (RETURN) Statement	II-5-19
5.3.4	Executive (EXEC) Statement	II-5-22
5.3.5	Procedure Switch Call	II-5-23
5.4	Replacement Statements	II-5-26

TABLE OF CONTENTS (Continued)

<u>Paragraph</u>	<u>Title</u> (Continued)	<u>Page</u>
5.4.1	Assignment (SET) Statement	II-5-26
5.4.1.1	Arithmetic Assignment Statement	II-5-27
5.4.1.2	Literal Assignment Statement	II-5-31
5.4.1.3	Status Assignment Statement	II-5-32
5.4.1.4	Boolean Assignment Statement	II-5-33
5.4.1.5	Multiword Assignment Statement	II-5-35
5.4.1.5.1	Multiword Table-to-Table Assignment Statement	II-5-35
5.4.1.5.2	Multiword Item-to-Item Assignment Statement	II-5-36
5.4.1.5.3	Single Word-to-Multiword Assignment Statement	II-5-36
5.4.2	Exchange Statement (SWAP)	II-5-38
5.4.3	Shift (SHIFT) Operation	II-5-39
5.4.4	Pack (PACK) Operation	II-5-40
5.5	Control Statements	II-5-41
5.5.1	GOTO Statement Name	II-5-41
5.5.2	GOTO Switch Name	II-5-42
5.5.3	STOP Statement	II-5-45
5.6	Decision Statements	II-5-45
5.6.1	Logical Decision Statement	II-5-46
5.6.2	Table Search Statement	II-5-48
5.6.2.1	FIND Statement	II-5-48
5.6.2.2	Search Decision Statement	II-5-50
5.6.2.3	Table Search Format	II-5-51
5.6.2.4	Table Search Examples	II-5-52
5.6.3	Validity Decision Statement	II-5-54
5.6.4	Parity Decision Statement	II-5-55
5.6.5	ELSE Statement	II-5-56
5.6.6	Nested Decision Statements	II-5-58
5.7	Statement Blocks	II-5-59
5.7.1	BEGIN Block	II-5-60
5.7.2	VARY Block	II-5-61
5.7.2.1	VARY Statement	II-5-62
5.7.2.1.1	Index Clause	II-5-64
5.7.2.1.2	WHILE Clause	II-5-65
5.7.2.1.3	UNTIL Clause	II-5-66
5.7.2.2	Resume (RESUME) Statement	II-5-66
5.7.2.3	End Vary Statement (END)	II-5-67

TABLE OF CONTENTS (Continued)

<u>Paragraph</u>	<u>Title</u> (Continued)	<u>Page</u>
5.7.2.4	Examples of VARY Blocks	II-5-67
5.7.3	FOR Block	II-5-72
5.7.3.1	FOR Statement	II-5-73
5.7.3.2	Value Block	II-5-76
5.7.3.3	FOR Block Examples	II-5-77

SECTION 6
INPUT/OUTPUT STATEMENTS

6.1	Input/Output Operations	II-6-1
6.1.1	INPUT Statement	II-6-4

TABLE OF CONTENTS (Continued)

<u>Paragraph</u>	<u>Title</u>	<u>Page</u>
SECTION 6 (Continued)		
6.1.2	OUTPUT Statement	II-6-7
6.1.3	FORMAT Declaration	II-6-8
6.2	Encode and Decode Operations	II-6-15
6.3	Nonstandard File Control	II-6-18
6.3.1	FILE Declaration	II-6-20
6.3.2	OPEN Statement	II-6-24
6.3.3	ENDFILE Statement	II-6-24
6.3.4	CLOSE Statement	II-6-26
6.4	Device State Checking	II-6-26
6.5	Device Positioning	II-6-28
6.5.1	Positioning by Files	II-6-28
6.5.2	Positioning by Records	II-6-30
6.6	File and Record Position Determination	II-6-31
6.7	Record Length Determination	II-6-32
6.8	Device Identification Operations	II-6-33
6.8.1	DEFID Statement	II-6-33
6.8.2	CHECKID Statement	II-6-35
SECTION 7 COMPILE-TIME SYSTEM FACILITIES		
7.1	Accessing the Compiler	II-7-1
7.2	Major and Minor Headers	II-7-3
7.3	Options Header Statement	II-7-5
7.3.1	SOURCE Option	II-7-6
7.3.2	OBJECT Option	II-7-7
7.3.3	LISTING Option	II-7-10
7.3.4	MONITOR Option	II-7-11
7.3.5	NONRT Option	II-7-12
7.3.6	Two-Level Diagnostics	II-7-12
7.3.7	MODEVRBL Option	II-7-12
7.3.8	STRUCTURED Option	II-7-12A
7.4	Allocation Header Statements	II-7-12B
7.4.1	Pooling Statements	II-7-12B
7.4.1.1	LOCDPOOL Statement	II-7-14
7.4.1.2	TABLEPOOL Statement	II-7-16
7.4.1.3	BASE Statement	II-7-17
7.4.1.4	DATAPOL Statement	II-7-18
7.4.2	EQUALS Statement	II-7-19
7.4.2.1	Defining a Tag	II-7-20
7.4.2.2	Establishing Relative Locations	II-7-21
7.4.3	NITEMS Statement	II-7-22
7.5	Library Retrieval Header Statements	II-7-22
7.5.1	LIBS Statement	II-7-23
7.5.2	Retrieval Selection Statements	II-7-23

TABLE OF CONTENTS (Continued)

<u>Paragraph</u>	<u>Title</u>	<u>Page</u>
SECTION 7 (Continued)		
7.5.3	Correcting Elements During Library Retrieval	II-7-24B
7.5.4	DEP Statement	II-7-25
7.5.5	Key Specification	II-7-25
7.6	Miscellaneous Header Statements	II-7-27
7.6.1	SYS-INDEX Statement	II-7-27
7.6.2	MEANS Statement	II-7-27
7.6.3	EXCHANGE Statement	II-7-28
7.6.4	DEBUG Statement	II-7-29
7.6.5	CSWITCH Declarations	II-7-30
7.6.5.1	CSWITCH Selection Declaration	II-7-30
7.6.5.2	CSWITCH Brackets	II-7-30A
7.6.5.3	CSWITCH Deletion	II-7-30B
7.6.5.4	CSWITCH Example	II-7-30B
7.6.6	EXECUTIVE Statement	II-7-31
7.6.7	CMODE Statement	II-7-32
7.6.8	SPILL Statement	II-7-32

SECTION 8
DEBUG STATEMENTS

8.1	Display Statement	II-8-2
8.2	Snap Statement	II-8-5
8.3	Range Declaration	II-8-7
8.4	Trace Statement	II-8-9
8.5	Procedure Trace (PTRACE)	II-8-11

SECTION 9
DIRECT CODE

9.1	Direct Code Statement Format	II-9-1
9.2	Direct Code Statement Repertoire	II-9-2
9.2.1	Direct Code Directives	II-9-2
9.2.2	Constants	II-9-6
9.2.2.1	Decimal Numbers	II-9-6
9.2.2.2	Octal Numbers	II-9-7
9.2.2.3	Floating-Point Numbers	II-9-7
9.2.2.4	Character Strings	II-9-8
9.2.2.5	Scaled Decimal Numbers	II-9-9
9.2.2.6	Scaled Octal Numbers	II-9-9
9.2.3	Data Expressions	II-9-10
9.2.4	Literals	II-9-11
9.2.5	Direct Constant Entries	II-9-11
9.2.6	Instruction Expressions	II-9-12
9.3	Processing Conventions	II-9-15

TABLE OF CONTENTS (Continued)

<u>Paragraph</u>	<u>Title</u>	<u>Page</u>
SECTION 10 COMPILER OUTPUTS		
10.1	Source Listing Format	II-10-1
10.2	Source and Mnemonic Listing Format	II-10-2
10.3	Local Cross-Reference Listings	II-10-4
10.4	Global Cross-Reference Listing	II-10-5
10.5	Symbol Analysis Format	II-10-6
10.6	Compiler Error Summary	II-10-12
SECTION 11 ASSEMBLER		
11.1	Assembler Functions	II-11-1
11.1.1	Input Language Structure	II-11-2
11.1.1.1	Label	II-11-2
11.1.1.2	Statements	II-11-5
11.1.1.2.1	Fields	II-11-5
11.1.1.2.2	Subfields	II-11-5
11.1.1.2.3	Omission of Subfields	II-11-5
11.1.1.2.4	Statement Continuation	II-11-6
11.1.1.2.5	Statement Termination and Notes	II-11-6
11.1.1.2.6	Blank Card Images	II-11-6
11.1.1.2.7	Language Structure Summary	II-11-6
11.1.1.3	Notations Used In This Section	II-11-7
11.1.1.4	Coding Control Statements	II-11-7
11.1.1.4.1	Comments	II-11-7
11.1.1.4.2	Printer Page Control	II-11-7
11.1.1.5	Directives	II-11-8
11.1.2	Addressing Sections	II-11-8
11.1.3	Segmentation	II-11-8
11.1.4	Assembly Base Addresses	II-11-9
11.1.5	Conditional Assembly	II-11-9
11.1.6	Library Usage	II-11-10
11.1.7	Macros	II-11-10
11.1.8	Expressions	II-11-11
11.1.9	Assembler Generation	II-11-11
11.1.9.1	Full-Word	II-11-11
11.1.9.2	Half-Word	II-11-11
11.1.10	Temporary Storage	II-11-11
11.1.11	Assembler Output	II-11-12
11.1.12	Assembly Time Allocation	II-11-12
11.1.13	Linking	II-11-12
11.2	Control Card	II-11-12
11.2.1	Start Assembly (ULTRA)	II-11-12

TABLE OF CONTENTS (Continued)

<u>Paragraph</u>	<u>Title</u>	<u>Page</u>
SECTION 11 (Continued)		
11.2.2	Stop Assembly (OFF)	II-11-14B
11.2.3	Disable Object Output Code (OFO)	II-11-14B
11.2.4	Sample Deck Using Control Cards	II-11-15
11.3	Source Statements	II-11-15
11.3.1	Label Field	II-11-15
11.3.1.1	Labels	II-11-16
11.3.1.2	Address Counter Declaration	II-11-18
11.3.1.3	Leading Asterisk (*)	II-11-18
11.3.1.4	Half-Word Instruction Labels	II-11-18
11.3.2	Operation Field	II-11-19
11.3.2.1	Processor Instruction Mnemonics	II-11-20
11.3.2.2	Input/Output Controller Command Mnemonics	II-11-20
11.3.2.3	Directive Mnemonics	II-11-20
11.3.3	Directives	II-11-21
11.3.3.1	ABS Directive	II-11-21
11.3.3.2	BYTE Directive	II-11-21
11.3.3.3	CHAR Directive	II-11-22
11.3.3.4	DO Directive	II-11-23
11.3.3.4A	EMBED Directive	II-11-25
11.3.3.5	END Directive	II-11-26
11.3.3.6	EQU Directive	II-11-26A
11.3.3.7	EVEN, ODD Directives	II-11-26B
11.3.3.8	FORM Directive	II-11-27
11.3.3.9	LCR Directive	II-11-28
11.3.3.10	LIBS Directive	II-11-29
11.3.3.11	LIB Directive	II-11-29
11.3.3.12	LINK Directive	II-11-31
11.3.3.13	LIST, ELIST, and NOLIST Directives	II-11-32
11.3.3.14	LIT Directive	II-11-33
11.3.3.15	LLT Directive	II-11-34
11.3.3.16	PXL Directive	II-11-35
11.3.3.17	RES Directive	II-11-35
11.3.3.18	RF\$ Directive	II-11-36
11.3.3.19	SEGEND Directive	II-11-37
11.3.3.20	SETADR Directive	II-11-38
11.3.3.21	WRD Directive	II-11-39
11.3.3.22	TAGTBL Directive	II-11-40
11.4	Macro Statements	II-11-40
11.4.1	MACRO and END Directives	II-11-40
11.4.1.1	Paraforms	II-11-42
11.4.1.2	Starred Labels Within Macros	II-11-45

TABLE OF CONTENTS (Continued)

<u>Paragraph</u>	<u>Title</u>	<u>Page</u>
SECTION 11		
(Continued)		
11.4.1.3	Operand Field	II-11-45
11.4.2	Other Macro-Oriented Directives	II-11-46
11.4.2.1	NAME Directive	II-11-46
11.4.2.2	GO Directive	II-11-48
11.4.3	Summary of Macro Usage	II-11-49
11.4.4	Special Consideration When Coding Macros	II-11-50
11.4.4.1	Comments	II-11-50
11.4.4.2	Labels on a Macro Reference Line	II-11-50
11.4.4.3	Address Counter Declarations Within a Macro	II-11-51
11.4.4.4	Externalizing Labels	II-11-51
11.4.4.5	Macro Reference Lines	II-11-53
11.4.4.6	Complex Macros	II-11-53
11.5	Address Counter Declarations	II-11-54
11.6	Expression Statements	II-11-55
11.6.1	Labels	II-11-56
11.6.2	Address Counter	II-11-56
11.6.3	Decimal Number	II-11-57
11.6.4	Octal Number	II-11-57
11.6.5	Floating Point Number	II-11-58
11.6.6	Fixed Point Number	II-11-59
11.7	Data Modes	II-11-59
11.7.1	Literals	II-11-59
11.7.2	Data Words	II-11-60
11.7.2.1	Constants	II-11-60
11.7.2.2	Character Strings	II-11-61
11.8	Operators	II-11-62
11.8.1	Symbols	II-11-63
11.8.1.1	Arithmetic Operators	II-11-65
11.8.1.1.1	*+ (Positive Exponentiation)	II-11-65
11.8.1.1.2	*- (Negative Exponentiation)	II-11-65
11.8.1.1.3	*/ (Binary Exponentiation or Scaling)	II-11-66
11.8.1.1.4	* (Arithmetic Product)	II-11-66
11.8.1.1.5	/ (Arithmetic Quotient)	II-11-67
11.8.1.1.6	// (Covered Quotient)	II-11-67
11.8.1.1.7	+ (Arithmetic Sum)	II-11-67
11.8.1.1.8	- (Arithmetic Difference)	II-11-67
11.8.1.2	Logical Operators	II-11-68
11.8.1.2.1	** (Logical Product)	II-11-68
11.8.1.2.2	++ (Logical Sum)	II-11-68
11.8.1.2.3	-- (Logical Difference)	II-11-68

TABLE OF CONTENTS (Continued)

<u>Paragraph</u>	<u>Title</u>	<u>Page</u>
SECTION 11 (Continued)		
11.8.1.3	Conditional Operators	II-11-68
11.8.1.3.1	= (Equal)	II-11-69
11.8.1.3.2	> (Greater Than)	II-11-69
11.8.1.3.3	< (Less Than)	II-11-69
11.8.2	Operator Priorities	II-11-70
11.8.3	Parenthetical Grouping	II-11-71
11.8.4	Relocatability	II-11-71
11.9	Assembler Outputs	II-11-73
11.9.1	Side-By-Side Listing	II-11-73
11.9.2	Error Codes	II-11-73
11.9.2.1	Expression (E)	II-11-73
11.9.2.2	Duplicate (D)	II-11-73
11.9.2.3	Undefined (U)	II-11-74
11.9.2.4	Instruction (I)	II-11-74
11.9.2.5	Relocation (R)	II-11-74
11.9.2.6	Truncation (T)	II-11-74
11.9.2.7	Overflow (O)	II-11-74
11.9.2.8	Name (N)	II-11-74
11.9.2.9	Level (L)	II-11-75
11.9.2.10	Floating Point (F)	II-11-75
11.9.2.11	Warning (W)	II-11-75
11.9.3	Generation Formats	II-11-75
11.9.4	Listing of Labels	II-11-76
11.9.4.1	Level 0	II-11-76
11.9.4.2	Level 1	II-11-76
11.9.4.3	LLT Sample Listing	II-11-76
11.9.4.4	Undefined Labels	II-11-76
11.9.4.5	Cross Reference Listing	II-11-77
11.10	Assembler Diagnostics and Status	II-11-77
11.10.1	Assembly Errors	II-11-77
11.10.2	Assembler Internal Errors	II-11-79
11.10.2.1	Core Overflow	II-11-79
11.10.2.2	Level Overflow	II-11-79
11.10.3	Library Call Errors	II-11-79
11.10.4	Peripheral Errors	II-11-79
11.11	Source Deck Organization	II-11-80
11.12	Special Considerations	II-11-83
SECTION 12 INSTRUCTION REPERTOIRE		
12.1	AN/UYK-7 Computer Functions	II-12-1
12.1.1	Register Format and Usage	II-12-2
12.1.1.1	Program Address Register	II-12-2

TABLE OF CONTENTS (Continued)

<u>Paragraph</u>	<u>Title</u>	<u>Page</u>
SECTION 12 (Continued)		
12.1.1.2	Addressable Registers, Control Memory	II-12-3
12.1.2	Modes of Operation	II-12-8
12.1.2.1	Interrupt State	II-12-8
12.1.2.2	Task State	II-12-8
12.1.2.3	Active Status Register	II-12-9
12.2	AN/UYK-7 Instruction Formats	II-12-12
12.2.1	Format I Instructions	II-12-12
12.2.2	Format II Instructions	II-12-13
12.2.3	Format III Instructions	II-12-13
12.2.4	Format IV-A Instructions	II-12-14
12.2.5	Format IV-B Instructions	II-12-14
12.2.6	Indirect Word	II-12-15
12.2.7	I/O Commands Formats	II-12-16
12.2.7.1	Normal Mode	II-12-16
12.2.7.2	ESI Mode	II-12-17
12.3	Symbolic Conventions	II-12-17
12.3.1	f - Function Code Designator	II-12-17
12.3.2	a - Arithmetic Code Designator	II-12-17
12.3.3	k - Operand Interpretation Code Designator	II-12-22
12.3.4	b - Index Register Code Designator	II-12-25
12.3.5	i - Indirect Address Code Designator	II-12-25
12.3.6	s - Base Register Code Designator	II-12-25
12.3.7	y - Operand Code Designator	II-12-25
12.3.8	f ₂ , f ₃ , f ₄ , - Subfunction Code Designators	II-12-26
12.3.9	m - Shift Counter Field	II-12-26
12.3.10	m - Monitor Interrupt Code Designator	II-12-26
12.3.11	c - Chain Flag Code Designator	II-12-26
12.3.12	j - Channel Number	II-12-26
12.4	Computer-Instruction Repertoire	II-12-26
12.4.1	Load and Store Instructions	II-12-35
12.4.2	Arithmetic Instructions	II-12-41
12.4.3	Jump Instructions	II-12-50
12.4.4	Instructions Involving Comparison Operations	II-12-57
12.4.5	Instructions Involving Logical Operations	II-12-62
12.4.6	Shift Instructions	II-12-68
12.4.7	Instructions Referencing Control Memory	II-12-71
12.4.8	Interrupt Handling Instructions	II-12-79
12.4.9	Miscellaneous Instructions	II-12-84
12.4.10	Extension Mnemonics	II-12-92
12.4.11	Input/Output Instructions	II-12-98

TABLE OF CONTENTS (Continued)

<u>Paragraph</u>	<u>Title</u>	<u>Page</u>
	APPENDIX A CHARACTER CODES	A-1
	APPENDIX B SUMMARY OF SYSTEM STATEMENTS	B-1
	APPENDIX C SUMMARY OF SERVICE ROUTINE CALLING SEQUENCES	C-1
	APPENDIX D CMS-2 COMPILER RESERVED WORD LIST	D-1
	APPENDIX E COMPILER ERROR MESSAGES AND LIMITS	E-1
	APPENDIX F SUMMARY OF ASSEMBLER ERROR CODES	F-1
	APPENDIX G AN/UYK-7 CONDENSED REPERTOIRE	G-1
	APPENDIX H CMS-2 SYSTEM TAPE DUPLICATION	H-1
	APPENDIX I SYSTEM MODIFICATION	I-1
	CMS-2 KEYWORD INDEX	K-1

LIST OF ILLUSTRATIONS

<u>Figure</u>	<u>Title</u>	<u>Page</u>
1-1	Typical Minimum Configuration	11-1-4
2-1	CMS-2 Compile-Time System	11-2-4
2-2	Structuring of Data Designs and Procedures	11-2-6
2-3	Source Deck Forms	11-2-7
2-4	System Procedure Design	11-2-8
2-5	Range of Program Identifiers	11-2-10
2-6	CMS-2 Program Structure Declaratives	11-2-11
2-7	A Compile-Time System Structure	11-2-12
2-8	Statement Execution Flow Involving Procedure Calls	11-2-15
2-9	Table Structure	11-2-18
2-10	Field Assignments for a Table	11-2-19
2-11	Table Storage Sequence	11-2-19
2-12	Parent Table Relationships	11-2-21
2-13	A Three-Dimensional Array	11-2-22
2-14	Array Storage Sequence	11-2-23
5-1	VARY Flow	11-5-63
6-1	Input/Output Data Flow	11-6-3
7-1	Elements of a Compile-Time System	11-7-2
11-1	Assembler Pass 1 Data Flow	11-11-3
11-2	Assembler Pass 2 Data Flow	11-11-4
11-3	Sample Deck Using Control Cards	11-11-15
11-4	Sample Deck to Assemble, Load, and Execute a Single Program	11-11-17
11-5	Sample Cross-Reference Listing	11-11-78
11-6	Source Deck Organization for a Single Program	11-11-60
11-7	Source Deck Organization for Assembling Using Library Input	11-11-31
11-8	Source Deck Organization for Two or More Dependent Programs or Segments	11-11-32
11-9	Source Deck Organization for Two or More Independent Programs or Segments	11-11-32
11-10	Source Deck Assembly Time Allocation	11-11-33

LIST OF TABLES

<u>Table</u>	<u>Title</u>	<u>Page</u>
4-1	Examples of Variable Declarations	II-4-20
4-2	Examples of Type-a Fields	II-4-32
5-1	Arithmetic Operators	II-5-2A
5-2	Relational Operators	II-5-6
5-3	Boolean Operators	II-5-6
5-4	Truth Table	II-5-8
6-1	CMS-2 Operating System Standard Files	II-6-2
7-1	Equals Expression Summary	II-7-20
9-1	Instruction Sub-field Valid Forms	II-9-19
11-1	Operators and Priorities of Operators	II-11-64
11-2	Data Modes for Operator Items	II-11-65
11-3	Relocation of Binary Items	II-11-72
11-4	Single and Double Precision Expressions	II-11-72
12-1	Central Processor Control Memory Address Assignments	II-12-4
12-2	AN/UYK-7 Computer Modes of Operation	II-12-8
12-3	Active Status Register	II-12-9
12-4	Instruction Repertoire Symbol Definitions	II-12-19
12-5	General Operand Interpretation (Memory To Arithmetic)	II-12-23
12-6	General Operand Interpretation (Arithmetic To Memory)	II-12-23
12-7	General Operand Interpretation (Normal Replace Instruction Interpretation)	II-12-24

SECTION 1

INTRODUCTION

1.1 PURPOSE AND SCOPE

This user's reference manual contains the information required by programmers and operators who wish to use or control the operation of the Compiler-Monitor System (CMS-2) developed for use with the AN/UYK-7 Computer. Univac Systems Programming Group developed this system for the Department of the Navy, Naval Ship Systems Command, under contract number N00024-70-C-1142.

This manual consists of two volumes with contents as follows:

Volume I - Monitor, Loader, Librarian, Peripheral Utilities, and System Operation - contains descriptions of command formats recognized by the Monitor, Loader, Librarian, and Peripheral Utilities. This volume also contains descriptions of calling sequences required to reference Monitor service routines and descriptions of operator commands recognized by the CMS-2.

Volume II - Compiler, Assembler, and Instruction Repertoire - contains a description of the languages recognized by the Compiler (both high level and low level) and the Assembler, including a comprehensive description of macro generation. Volume II also contains a complete functional description of the AN/UYK-7 Computer instruction repertoire.

While this manual contains comprehensive descriptions of command syntax, calling sequences, and messages generated by various components of the system, detailed descriptions of the functions performed by each component, the interfaces between the components, and functions performed by the hardware are all beyond the scope of this manual.

1.2 APPLICABLE DOCUMENTS

The following documents augment the content of this manual to provide a complete description of the Compiler-Monitor System 2 (CMS-2) developed for the AN/UYK-7 Computer:

NAVSHIPS 0967-029-5430
Program Specification for
Compiler-Monitor System
for use with the AN/UYK-7
Computer.

Consists of five parts containing the basic functional specifications for each of the five major components in the system:

Part 1 - Compiler

Part 2 - Librarian

Part 3 - Monitor

Part 4 - Loader

Part 5 - Peripheral Utilities

NAVSHIPS 0967-029-5440
Program Design Plan for
Compiler-Monitor System
for use with the AN/UYK-7
Computer.

Consists of five parts containing the detailed descriptions of functions performed by each of the five major components in the system:

Part 1 - Compiler

Part 2 - Librarian

Part 3 - Monitor

Part 4 - Loader

Part 5 - Peripheral Utilities

NAVSHIPS 0967-051-6291
AN/UYK-7 Digital Data Computer

Contains the hardware description of the computer used by the system.

Univac DSD Document, PX 3699A
UNIVAC[®] 1532 Input/Output
Console Technical Description

Contains the hardware description of the I/O console used by the system.

Univac DSD Document, PX 3662
UNIVAC[®] 1540/1541 Magnetic
Tape Units Technical Description

Contains the hardware description of the magnetic tape units used by the system.

Univac DPD Document, UP2543, Rev 1,
UNIVAC[®] 1004 Card Processor Reference

Contains the hardware description
of the card reader, card punch,
and high-speed printer used by
the system.

1.3 SYSTEM CAPABILITIES

CMS-2, also referred to as the system, provides optimum utilization of the AN/UYK-7 Computer and associated peripherals in a serial batch processing environment. The system is user oriented and designed to optimize the capabilities of the system for all users. Its major features are:

1. Provides the simplest possible operational characteristics consistent with the full utilization of the system.
2. Provides a simple yet flexible means of generating, storing, and updating computer programs at the individual installation.
3. Provides a broad and easily utilized system of program construction, manipulation, and debugging aids.

1.3.1 Hardware Requirements

The system operates on an AN/UYK-7 Computer and its associated peripherals. The minimum equipment required to efficiently operate the system consists of:

1. AN/UYK-7 Computer with 3 memory banks.
2. UNIVAC[®] 1004 Card Reader, Punch, and Printer or equivalent.
3. Six UNIVAC[®] 1240 or 1540 Magnetic Tape Transports or equivalent.
4. UNIVAC[®] 1532 Input/Output Console and Keyboard or equivalent.

Figure 1-1 illustrates a typical minimum configuration.

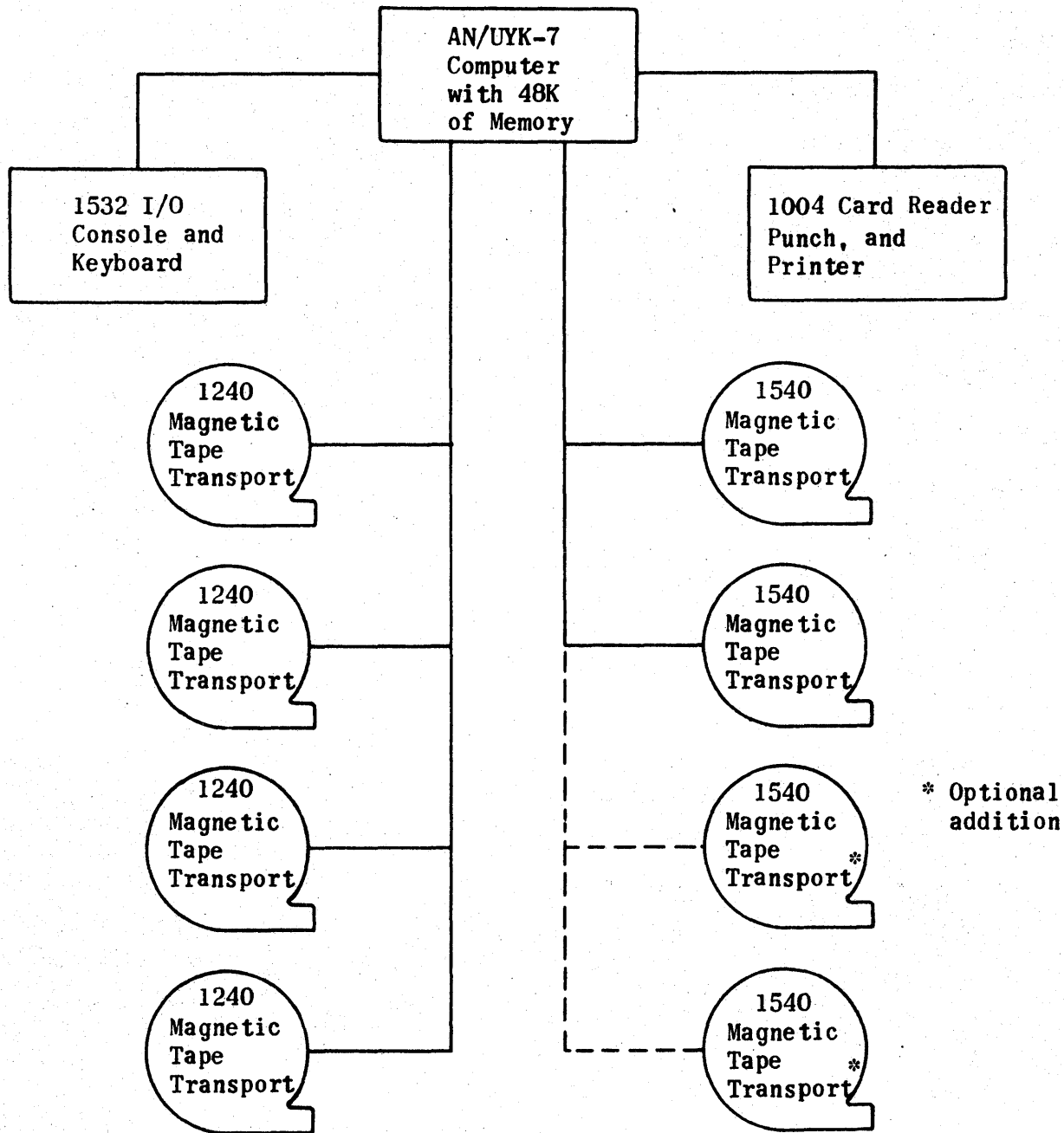


Figure 1-1. Typical Minimum Configuration

1.3.2 Software Components

A software component of CMS-2 is a program which is designed to perform a specific function (e.g., compile a program, build a library) using an external data-base (e.g., source, object code). The component is designed to operate under the Monitor (a component that performs control functions) which determines its final operational allocation and provides it with a centralized input/output capability. Thus, a component represents a module in the system which may be tailored to operate under the Executives with little anticipated change in the logic. These changes would necessarily reflect the idiosyncrasies of the particular executive, notably in the area of the Executive call. CMS-2, for the initial development, consists of the following components:

1. Monitor
2. Object Code Loader
3. Librarian
4. Peripheral Utilities
5. Compiler
6. Assembler
7. System Tape Generator

The system utilizes magnetic tape as the system storage medium. The tape format effectively provides the features which follow:

1. Efficient run time retrieval of programs.
2. Independent system initialization.
3. Potential system expansion.
4. Total system delivery.
5. Complete system update, maintenance and reproduction.

The system storage medium consists of two parts: 1) the operational library and 2) the system library.

The operational library contains the Monitor and components which make up the system; in addition, it contains directory information required for locating programs on the operational library, for relocatable loading (not instruction modification), and for initiating execution. This library occupies the first file on the system tape and is written in a format compatible with the NDRO bootstrap routine. The Monitor and the directory information are loaded when the bootstrap routine is initiated from the computer control console. This provides the independent system initialization.

The system Library contains the following:

1. Object code of all run time routines (implicit).
2. Object code of all intrinsic built-in routines.
3. Object code of Monitor and all components.
4. Compoools of selected data designs for Monitor and components.

1.3.2.1 Monitor

The Monitor (described in Volume I, Section 2) is a serial batch processing operating routine utilizing a single AN/UYK-7 unit processor. Major functions available through the Monitor are requested by the user through control cards described in Volume I, Section 2. The Monitor includes the Centralized Input/Output Module which supports I/O on the various devices attached to the AN/UYK-7. The Monitor also provides for handling of all classes of processor interrupts and interfaces to allow user programs to access vital interrupt information (e.g., floating point error). The Monitor is responsible for retrieval of components from the operational library on the system storage medium (system tape). It also maintains system core allocation algorithm.

1.3.2.2 Object Code Loader

The Object Code Loader (described in Volume I, Section 3) performs instruction modification to object code produced by the CMS-2 Compiler and CMS-2 Assembler. The Loader allows optimum code to be generated by the AN/UYK-7 language processors by combining independently compiled program segments under a common base register or registers, thus reducing the number of base register manipulation instructions which must be executed.

1.3.2.3 Librarian

The Librarian (described in Volume I, Section 4) provides a convenient, easy-to-use method of storing, retrieving, and updating both source statements and relocatable object code. The Librarian is capable of updating (i.e., adding, deleting, changing) both entire elements or individual items within an element.

1.3.2.4 Peripheral Utilities

The Peripheral Utilities (described in Volume I, Section 5) provide a variety of functions for manipulating data files on the peripheral devices. These functions include:

1. Position specified magnetic tape at the start of a file (designated by a file mark).
2. Position specified magnetic tape at the start of a record within a file.
3. Transfer tape data into memory (read tape).
4. Transfer memory data onto tape (write tape).
5. Compare the contents of two tapes and print out any differences.

1.3.2.5 Compiler

The Compiler (described in Volume II Sections 2 through 10) accepts both high and low level languages. The high level language is statement oriented and the low level is computer instruction mnemonic oriented. These languages describe the desired program, and the Compiler generates object code data that the Object Code Loader places into memory as an executable program. The Compiler input is called source code and the Compiler output is called object code.

1.3.2.6 Assembler

The Assembler (described in Volume II, Section 11) accepts a computer instruction mnemonic oriented language that gives the programmer absolute control of the structure of his program. In addition, the Assembler provides programmers with a level of assistance beyond that normally associated with an assembler class of language processors. The assembler accepts programmer definitions of pseudo-operations (called Macros) and then uses the definition whenever the programmer references the pseudo-operations. As with the Compiler, the Assembler input is called source code and the Assembler output is called object code.

1.3.2.7 System Tape Generator

The System Tape Generator (described in the System Programmer's Manual) provides an easy-to-use method of updating system tapes that have a directory scheme identical to the CMS-2 tape. The System Tape Generator accepts input that contains the necessary data to change tape directory information (e.g., number assigned to a new component, names of new records, names of records to be deleted), and then, in conjunction with the Object Code Loader, generates a new system tape complete with required directories.

1.4 SYSTEM OPERATION

System operation consists of initiating the system, preparing inputs to the system, and accepting outputs.

1.4.1 System Load and Initiation

System loading is initiated using the AN/UYK-7 Magnetic Tape Hardware Bootstrap Program. The resident Monitor is loaded at a specified location dependent on the number of operable memory banks currently available. The necessary allocation-dependent words in the resident Monitor are then initialized. The Monitor's data and the operational library directory for the Monitor are then loaded from the system tape, and the Monitor requests the current time, date, operating mode (open or closed shop), and standard selections of input/output devices. Any other flags or variables maintained by the Monitor and requiring initialization are preset.

1.4.2 Standard Input Processing

Once initiated, the Monitor starts reading card image data from the selected standard input device. This device is normally the card reader; however, the Monitor accepts commands to process data from either magnetic tape or the I/O console as standard input data. In each of these cases, the input data must be in card image format.

The data read from the standard input device consists of:

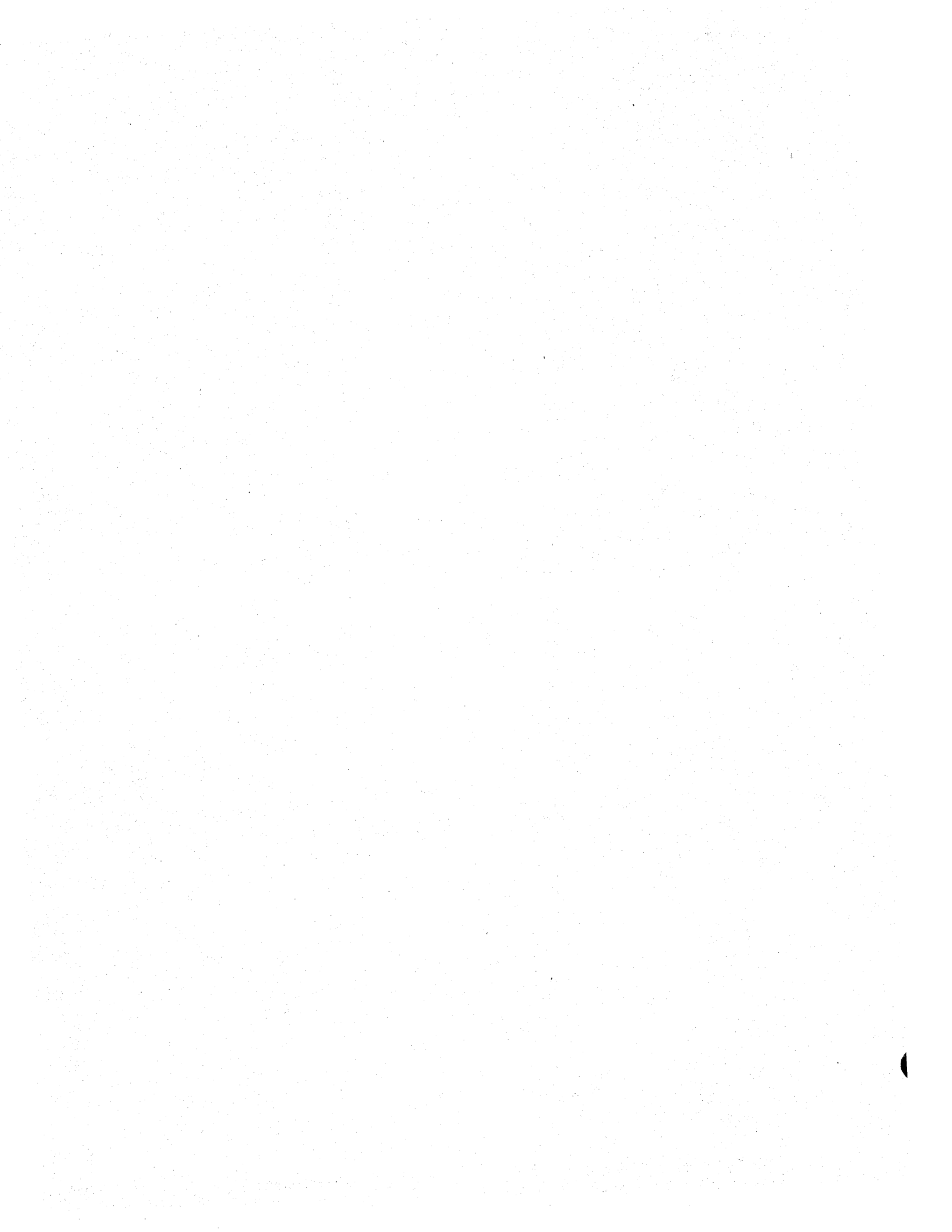
1. Monitor control cards which contain commands to the Monitor including commands to load and activate other system components or user programs.
2. Inputs to an activated system component or user program which contain both commands to the program and input data to be processed by the program.

In general, when the Monitor passes control to either another system component or user program, the component or program (as applicable) starts reading data from the standard input devices; for example, the card images following the image commanding the Monitor to load and activate the Compiler make up the source code input to the Compiler. On the other hand, most of the card images following the image commanding the Monitor to load and activate the Librarian are commands to the Librarian directing its manipulation of data stored on magnetic tape.

1.4.3 Standard Output Processing

The system has three basic outputs:

1. Standard output consisting of object code cards normally produced on the card punch. The Monitor accepts a command to place this data on magnetic tape in object code card image format instead of punching cards.
2. Hardcopy output consisting of high-speed printer-oriented data, such as program listings, normally produced on the UNIVAC[®] 1004 High Speed Printer. The Monitor accepts a command to place this data on magnetic tape in printer format instead of printing the data.
3. Console messages consisting of up to 72 characters of operator-oriented data (request to mount a tape) typed out on the I/O console.



SECTION 2

INTRODUCTION TO THE CMS-2 LANGUAGE

2.1 MAJOR FEATURES OF CMS-2

CMS-2 is a problem-oriented compiler language developed to meet the needs of real-time data processing and scientific applications. Its major features are described below:

- a. CMS-2 permits program modularization and adherence to the concepts of structured programming.
- b. Input to the CMS-2 Compiler is statement-oriented, rather than card-oriented. The source card format is free-form and may be arranged for user convenience.
- c. A broad range of data types is definable in CMS-2. These types include fixed-point, floating-point, Boolean, Hollerith (character), and status.
- d. CMS-2 permits direct reference to, and manipulation of, character and bit strings.
- e. Programs may include segments of symbolic machine language, referred to as direct code.

The remainder of this section presents a number of definitions, discusses various concepts fundamental to the CMS-2 language, and presents a summary of the specific capabilities of the language.

2.2 PROGRAM STRUCTURE

A CMS-2 program is composed of an orderly set of statements. These statements are composed of various symbols that are separated by delimiters. Three categories of symbols are processed: operators, identifiers, and constants. The operators are language primitives assigned by the Compiler to indicate

specific operations or definitions within a program. Identifiers are the unique names assigned by the programmer to data units, program elements, and statement labels. Constants are known values, and may be numeric (decimal or octal), Hollerith strings, status values, or Boolean.

CMS-2 statements are written in a free format and terminated by a dollar sign. Several statements may be written on one card, or one statement may cover many cards. A statement label may be placed at the beginning of a statement for reference purposes.

The collection of program statements developed by the programmer for input to the CMS-2 Compiler is known as the source code for a program and is composed of the following two basic types of statements:

1. Declarative statements - Provide basic control information to the Compiler and define the structure of the data associated with a particular program.
2. Dynamic statements - Cause the Compiler to generate executable machine instructions (object code) for a program.

These instructions, when executed at program run time, manipulate the data to solve the desired problem.

Declarative statements defining the data for a program are grouped into units called data designs. Data designs consist of the precise definition of temporary and permanent data storage areas, input areas, output areas, and special data units such as program switches. The dynamic statements that cause manipulation of data or express calculations to solve the programmer's problems are grouped into procedures. These data designs and procedures may be further grouped or classified to form elements of a CMS-2 program. At compile-time, the CMS-2 Compiler recognizes a system as any collection of program elements that may be compiled as an entity independent of any interfacing program elements. A compile-time system may comprise an entire execution package or it may be only a small part of a large program.

Before presenting any further discussion concerning the classification and grouping of procedures and data designs into elements and the combining of these elements to form systems, several concepts fundamental to the CMS-2 language must be explored.

2.2.1 Organization and Classification of Identifiers

The CMS-2 Compiler uses several conventions to classify data definitions and program identifiers that are defined in a user's program. These techniques assist the programmer in structuring his program and simplify the development and maintenance of the programs.

2.2.1.1 Forward and Backward References

The order in which definitions and references to these definitions appear in the source input to the Compiler is quite important. All data units are defined in data designs. Within the data design where it is defined, an identifier may generally be referenced either before it is defined (a forward reference) or after it is defined (a backward reference). However, references to data from outside a data design can only be backward; that is, the data must have already been defined before it can be referenced. Since data definitions always appear in data designs, and since data references usually appear in procedures, procedures generally follow the data designs defining the data referenced by the procedure.

References to statement labels within procedures and calls to procedures may be forward or backward, but must obey the following local/global limitations.

2.2.1.2 Local and Global Definitions

The Compiler further structures the referencing of identifiers by classifying all identifiers in a program as either local or global. Local definitions are those identifiers that can be referenced only from within the system element where they are defined. Global definitions are those identifiers that can be referenced both from inside the element where defined and from outside by subsequent system elements in the source input stream.

Figure 2-1 is a pictorial representation of a CMS-2 compile-time system consisting of three elements: A, B, and C. Since a definition in the CMS-2 language is said to be local if it is valid only within a single element of the system, any definition valid within element B of Figure 2-1 is said to be local to element B. A global definition in the system of Figure 2-1 is valid within elements A, B, and C.

An alternative definition of the term "system" can be derived from this local-global concept, i.e., a system is the largest global area within a CMS-2 compilation.

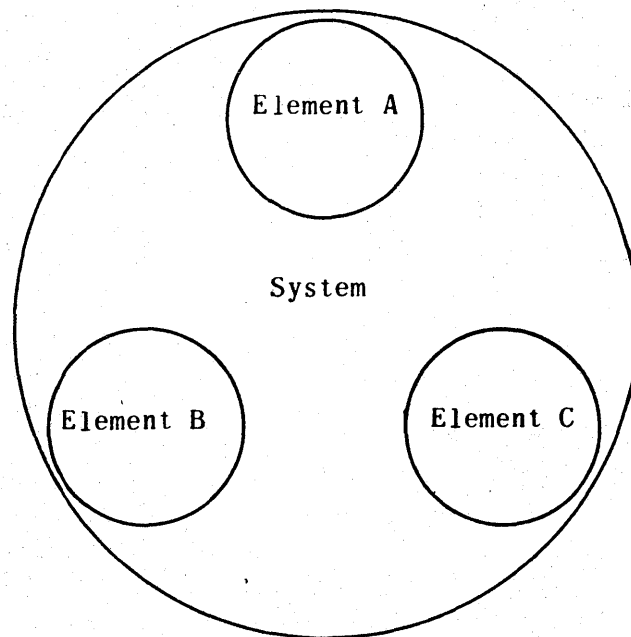


Figure 2-1. CMS-2 Compile-Time System

2.2.1.3 External References and Definitions

The CMS-2 Compiler provides the capability of compiling one or more elements of a large system independently. For example, all three elements of the system of Figure 2-1 could be compiled together as a single compile-time system. Alternatively, elements A and B could be compiled together as a

compile-time system and then element C could be compiled separately as another compile-time system. The compiler-produced output in each case is the computer-executable instructions (object code) for the various system elements. Later, the object code for elements A, B, and C may be combined by a relocatable linking loader program and executed together.

Presumably, there is some cross-referencing of data and procedures between the three elements of our example. In order to compile element C separately, any references made by element C to definitions in elements A and B must be handled in a special manner by the Compiler and the Loader. References of this type are called external references because they involve definitions that are external to element C and, in this case, external to the compile-time system as well. Those definitions in elements A and B that are referenced externally by element C are called external definitions because they are definitions that are available to elements external to A and B.

There are various ways in which definitions and references may be declared external. In some cases the Compiler will automatically treat a definition or a reference as being external. In other cases, external references and definitions must be explicitly declared by the programmer.

It should be noted that only global definitions may be externally referenced or defined. Local definitions are never valid outside, or external to, the element in which they are defined.

2.2.2 CMS-2 Elements

As described in the previous paragraphs, data designs may be grouped or classified to form elements of a CMS-2 program. One or more elements then make up a compile-time system. The ordering and content of program elements is subject to the rules governing range and classification of definitions.

The two types of elements within a compile-time system are system data designs and system procedures. System data designs contain global data definitions. System procedures contain one or more procedures and may also include local data design packages. A local data design, as the name implies, contains data definitions that are local to the system procedure in which the local data

design appears. This structuring of data designs and procedures into program elements within a system is illustrated in Figure 2-2.

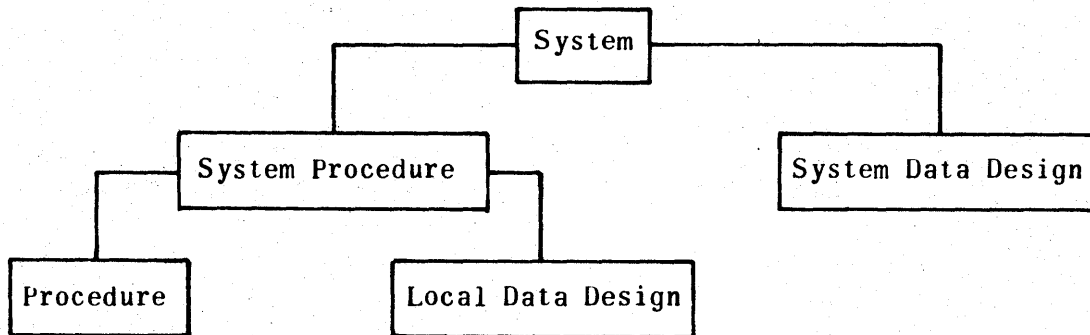


Figure 2-2. Structuring of Data Designs and Procedures

The hierarchy shown in Figure 2-2 indicates that, within a system, system data designs are equal in importance to system procedures; they are the program elements of a system. Keeping in mind the restrictions against forward referencing, a source deck may take various forms, as illustrated in Figure 2-3.

The technique illustrated in Figure 2-3 (A) is often used in constructing a program. Since definitions within a system data design are global to the balance of the system, system procedures may appear in any order following the system data design(s) defining the referenced data. Interspersing data designs and procedures as in Figure 2-3 (B) (C), however, has an advantage, especially in a large system, of maintaining data definitions in meaningful groups close to the associated procedures.

Note that Figure 2-3 illustrates several examples of a compile-time system, but these systems might be only a small part of an entire execution package. In addition, each compile-time system of Figure 2-3 might be further broken down into two or more compile-time systems. In this manner, corrections may be made to a particular system procedure, which may then be recompiled without compiling the entire execution package again.

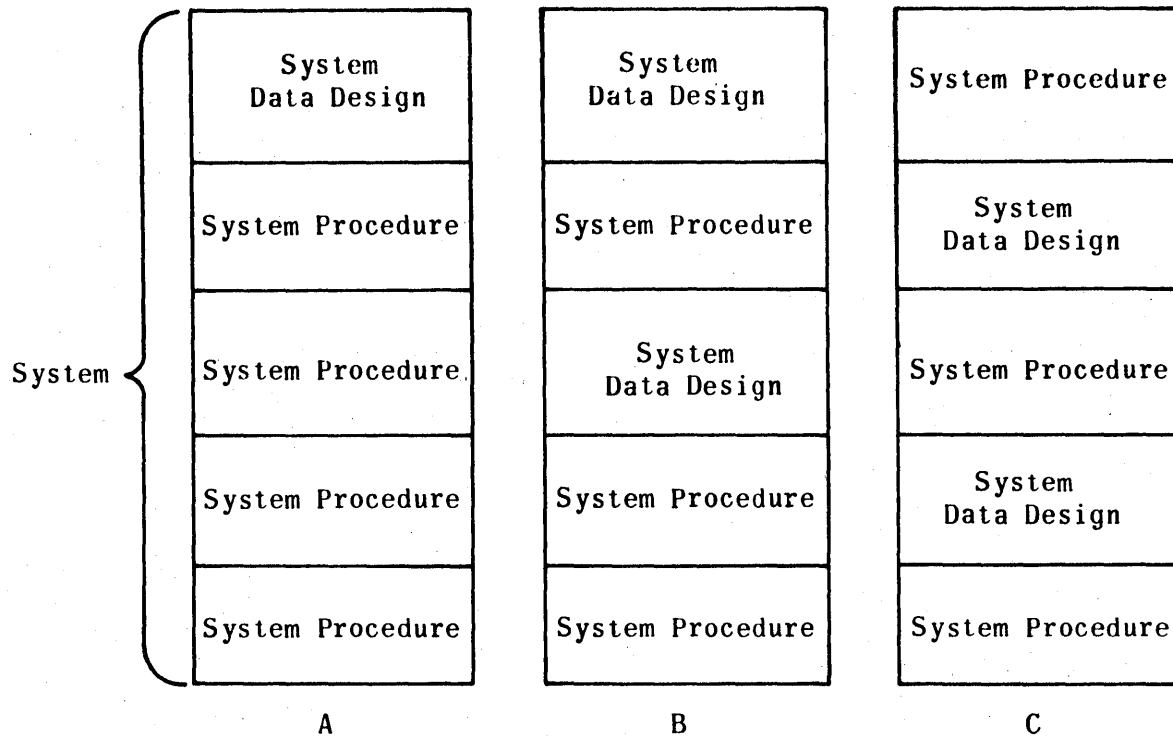


Figure 2-3. Source Deck Forms

System data designs and system procedures are the smallest program units that may be compiled individually. A compile-time system may consist of a single system data design or system procedure, but it cannot consist of a single local data design or procedure.

2.2.2.1 System Data Designs

Data designs contain descriptions of the attributes of the various data units (e.g., tables and variables) and their relationship to each other. As the Compiler processes these descriptions, it assigns and reserves core storage locations for subsequent references to the data units. Data designs may contain value information as well, which will cause the Compiler to generate object code to preset the data.

M-5035

Definitions within a system data design are global to the system. They are all automatically externally defined by the Compiler. There is, therefore, no need to specifically externally define any data within a system data design.

2.2.2.2 System Procedures

System procedures are composed of procedures and local data designs. A system procedure usually contains one procedure with a name identical to that of the system procedure name. This procedure is known as the prime procedure of that system procedure. The prime procedure entry point is automatically externally defined by the Compiler and is global to the system. Other system procedures, and data designs may reference prime procedures at will. Thus, the prime procedure of a system procedure is considered a global procedure (hence, the term system procedure).

A system procedure may contain more than one procedure, as illustrated in Figure 2-4.

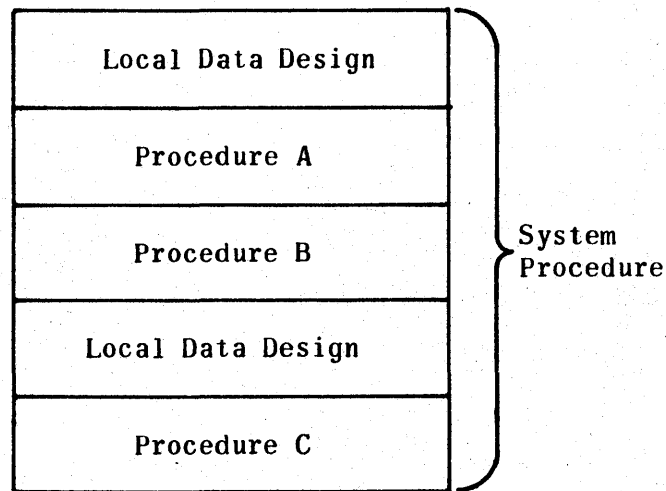


Figure 2-4. System Procedure Design

For example, the system procedure of Figure 2-4 contains three procedures and two local data designs. If this system procedure were named "B", procedure B would be the prime procedure of the system procedure and its name would be

global to the entire system. However, procedures A and C, along with the data units of the two data designs, would be local to the system procedure and could not be referenced from outside the system procedure.

2.2.2.2.1 Local Data Designs. The difference between system data designs and local data designs is that, while system data design definitions are global to the system and automatically externally defined, local data design definitions are local to the system procedure within which they are contained; any necessary external definitions must be explicitly indicated within the data design. In addition, a local data design may not be compiled separately from its associated system procedure.

The local data design is intended to be used for the definition of data units referenced only by the procedures within its system procedure. The use of local data designs reduces the possibility of duplication of data names in a large system because of their limited range of definition.

2.2.2.2.2 Procedures. Procedures contain CMS-2 statements and machine-language statements. They may not contain data definitions or data values for previously defined data. Procedures contain the statements from which the Compiler generates the instructions that actually perform the steps necessary to the solution of the problem. They must be included within a system procedure element at compile-time.

2.2.3 Range of Identifiers

As can be seen from the previous discussions, the organization of CMS-2 statements into system data designs and system procedures to form the elements of a program is closely related to the rules concerning classification of identifier definitions and references. These rules on the range of identifiers (i.e., local/global definitions and forward/backward references) are summarized in Figure 2-5.

	<u>Identifiers</u>	<u>Range Within Which They Can Be Referenced</u>
Global	Prime Procedures	Throughout the compile-time system.
	Data defined in a System Data Design	Within that system data design and in all system elements that follow.
Local	Local Procedures	Within the same system procedure.
	Data defined in a Local Data Design	Within that local data design and the remainder of the system procedure containing the local data design.
	All Statement Labels	Within the system procedure.

Figure 2-5. Range of Program Identifiers

2.3 DECLARATIVE STATEMENTS

The CMS-2 declarative statements provide the Compiler with information about program structure and data element definitions. Declaratives generally do not result in executable code. Declaratives are classified in three categories: program structure declaratives, data declaratives, and Compiler directive (or program control) declaratives.

2.3.1 Program Structure Declaratives

In the development of a CMS-2 program, the dynamic and data definition statements are organized into procedure and data design packages. CMS-2 program structure declaratives are used to define the source program organization by specifically delimiting the structure type as shown in Figure 2-6. An example of the correct organization of program structure declaratives for a compile-time system is presented in Figure 2-7.

<u>Beginning Delimiter</u>	<u>Ending Delimiter</u>	<u>Purpose</u>
SYSTEM	END-SYSTEM	Delimits a compile-time system
SYS-DD	END-SYS-DD	Delimits a system data design within a compile-time system
SYS-PROC	END-SYS-PROC	Delimits a system procedure within a compile-time system
LOC-DD	END-LOC-DD	Delimits a local data design within a system procedure
PROCEDURE	END-PROC	Delimits a procedure within a system procedure
FUNCTION	END-FUNCTION	Delimits a function within a system procedure
SYS-PROC-REN	END-SYS-PROC	Delimits a reentrant system procedure within a compile-time system
AUTO-DD	END-AUTO-DD	Delimits a reentrant data design within a reentrant system procedure
HEAD	END-HEAD	Delimits a header package within a compile-time system

Figure 2-6. CMS-2 Program Structure Declaratives

II-2-12

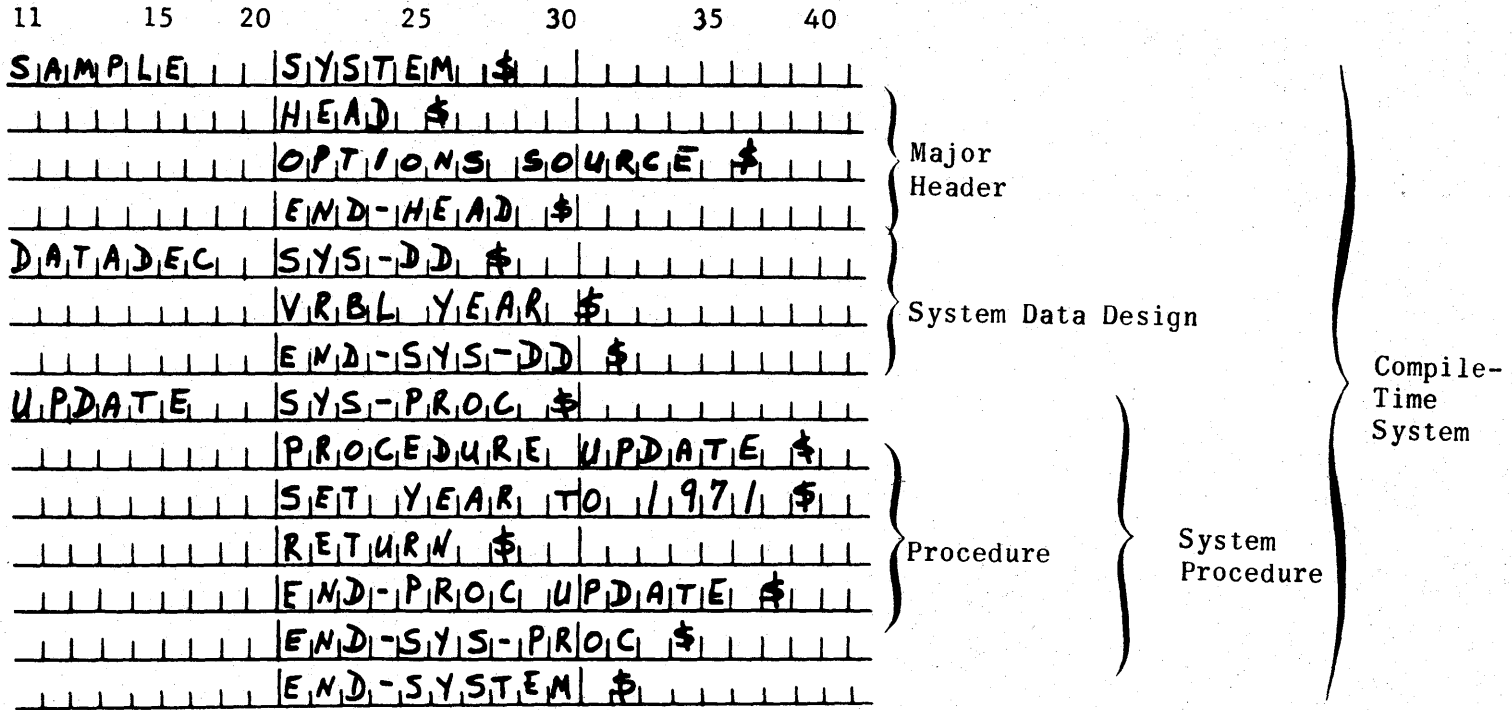


Figure 2-7. A Compile-Time System Structure

2.3.1.1 Procedure Structure Declaratives and Linking

The dynamic statements that describe the processing operations of a program are grouped into blocks of statements called procedures. The overall purpose of a program, its design, and to some extent, its size, influence the programmer's decision as to whether one or several procedures will be declared. The transfer of program control from one procedure to another requires the observance of procedure linking rules for such transfers.

The concept of procedure linking may best be described by posing a situation from which those linking requirements desirable for use by a programmer may be derived. As a program design develops, it becomes apparent to the programmer/designer that there is a requirement to execute a given set of statements at several points (within several procedures) in the total program. As each of these points is encountered, it would be advantageous to have a program control capability of branching to a common routine (procedure), processing, and returning to the next instruction following the program control branch point (or procedure call). Along with this procedure call should be a capability of simultaneously and automatically passing that data, from the calling procedure to the called procedure where the data is processed. This automatic data transfer is defined as input of parameters, that is, data input to the called procedure from the calling procedure.

Upon completion of processing by the called procedure, it also should be possible to automatically pass the results of the processing from the called procedure to the calling procedure when program control is returned to the calling procedure. This is defined as output of parameters, that is, data output to the calling procedure from the called procedure.

Additionally, there should be a capability of specifying an instruction address (statement label) to which the called procedure may transfer program control in the event it does not perform its normal processing due to invalid input data or processing checks indicating invalid or illogical results. This is defined as an abnormal exit (abnormal return).

The foregoing is the capability available to provide linkage among all procedures. Furthermore, all or part of these linkage capabilities may be used, depending upon the requirements of the program. The syntactical requirements for defining a procedure and making a procedure call are presented in Sections 4 and 5, respectively. Figure 2-8 is a schematic representation of the procedure linkage concept.

2.3.1.2 Reentrant System Procedures

Certain programming applications require that one or more of the procedures comprising the program package or system for that application be structured such that they may be shared by more than one task concurrently. Procedures of this type are said to be reentrant procedures.

The principal characteristic of a reentrant routine is that it must be divided into two logically and physically distinct parts: a constant part and a variable part. The constant part (instruction part) is loaded into memory once and services all tasks requiring this routine. One copy of the variable part (data area) belongs to each task that is being serviced. This copy is usually created (that is, it is allocated memory space) when the task is initiated.

Within the CMS-2 language, a programmer has the capability of declaring a system procedure to be reentrant. In this case, the object code generated by the Compiler for all procedures within this system procedure will be invariant (constant) under execution. In addition, a special type of local data design called an automatic data design may be declared within a reentrant system procedure. An automatic data design is used for the definition of temporary storage and procedure parameters used by the reentrant procedures within the system procedure. Within a reentrant system procedure, the Compiler automatically performs the required separation of the constant part (procedures) and the variable part (automatic data designs). Multiple copies of the variable part may then be loaded into memory along with a single copy of the constant part and the reentrant system procedure may be executed simultaneously by more than one task or central processor.

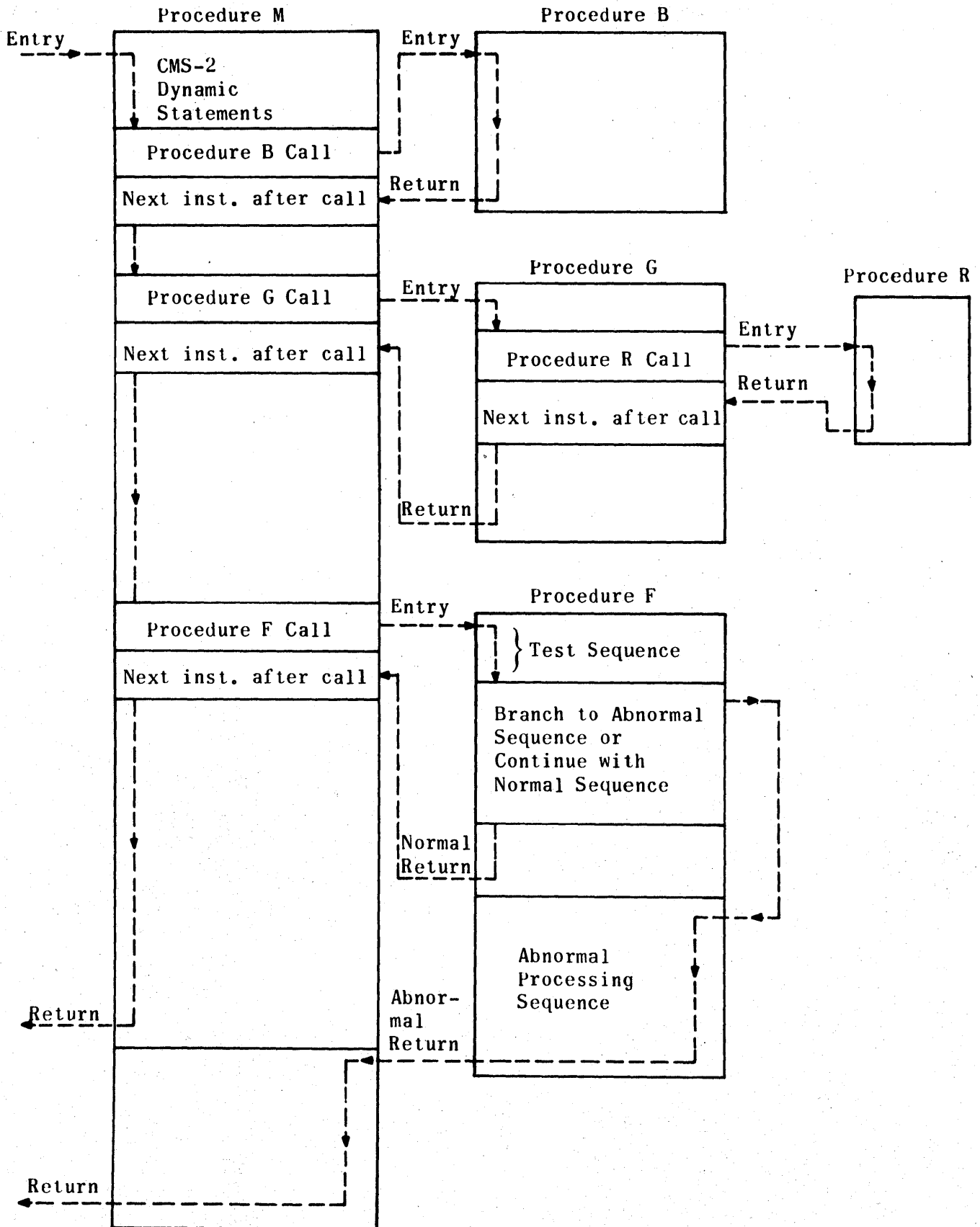


Figure 2-8. Statement Execution Flow Involving Procedure Calls

It must be clearly understood that the Compiler provides only this separation capability. The responsibility for loading these programs into memory and allocating space for automatic data designs is properly a function of loaders, monitors, and executive programs. Furthermore, the CMS-2 language and Compiler provides the capability, through this separation function, of implementing such sophisticated programming techniques as recursion and reentrance after suspension. However, much of the responsibility for this type of programming must be borne by the programmer/designer and the executive program for the application.

2.3.2 Data Declarations

Data declarations by the programmer define the format, structure, and order of data elements within a compile-time system. The three major data types are as follows:

1. Switches:
 - a. Statement switches.
 - b. Procedure switches.
2. Variables:
 - a. Computational:
 1. Integer.
 2. Fixed-point.
 3. Floating-point.
 - b. Non-computational:
 1. Hollerith.
 2. Boolean.
 3. Status.
3. Tables:
 - a. One-dimensional.
 - b. Multidimensional (array).
 - c. Subtables.

- d. Like-tables.
- e. Item areas.
- f. Fields.

2.3.2.1 Switches

Switches provide for the transfer of program control to a specific location within a compile-time system. Switches contain a set of identifiers, or switch points, to facilitate program transfers and branches. The switch points represent program addresses of statement labels or procedure names. Transfer of control to a particular switch point is usually determined by the value of a programmer-supplied index.

2.3.2.2 Variables

A variable is a singular piece of data. It may be one bit or multiple bits or words. A variable may be preset to a desired value within the definition statement. The variable may contain a constant value or its value may continuously change during program execution. Multiple variables having identical attributes may be defined in a single declarative statement. Data types that may be specified for a variable are arithmetic (fixed- or floating-point), Hollerith (character string), status (defined states of condition), or Boolean (true or false). An initial value, or preset, may be specified for the variable in the declarative statement.

2.3.2.3 Tables

Tables hold ordered sets of identically structured information. The common unit of data structure in a table is the item. An item consists of k computer words where k is selected by the programmer or Compiler. A table may contain n items, where n is programmer selected. Thus the size of the table in number of required computer words for storage becomes the product of n and k . A table structure is illustrated in Figure 2-9.

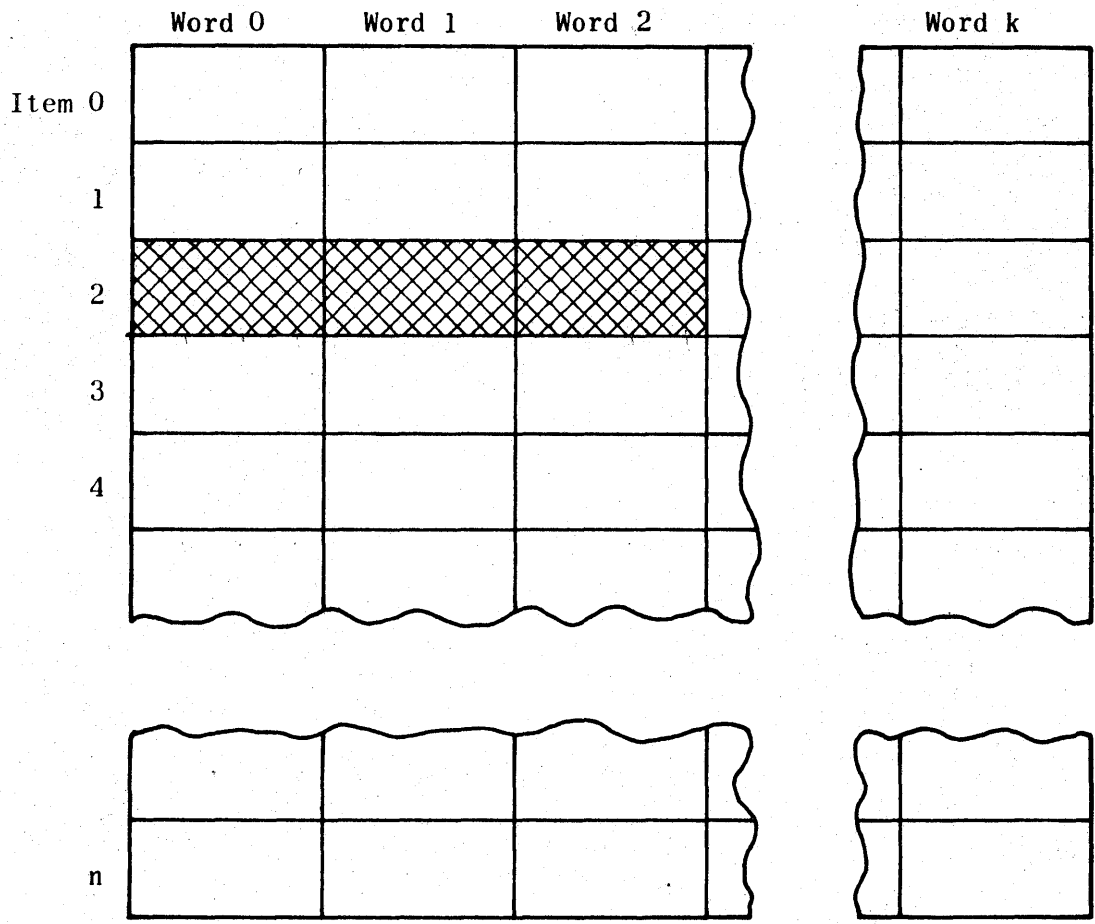


Figure 2-9. Table Structure

Items may be subdivided into fields. Fields are the smallest subdivision of a table. A field may be a partial word, a full word, or a multi-word subdivision. Data types that may be specified for a field are arithmetic (fixed- or floating-point), Hollerith (character string), status (defined states of condition), or Boolean (true or false). Fields may overlap each other. Data may be preset into a field. An example of field assignments is illustrated in figure 2-10.

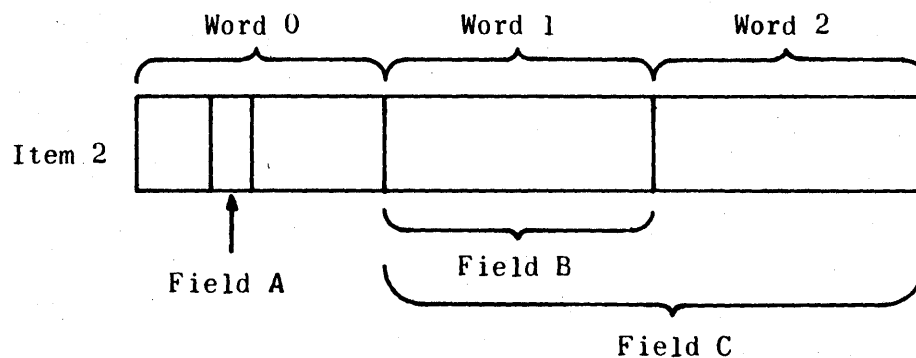


Figure 2-10. Field Assignments for a Table

CMS-2 tables may be defined as horizontal or vertical. This specification by the programmer dictates the manner in which the table words will be stored in core. The words of a horizontally declared table are stored such that words 0 of all items are stored sequentially, followed by words 1 of all items, etc. The words of a vertically defined table are stored such that all words of item 0 are stored sequentially, followed by all words of item 1, etc. Figure 2-11 illustrates the storage pattern for horizontal and vertical storage.

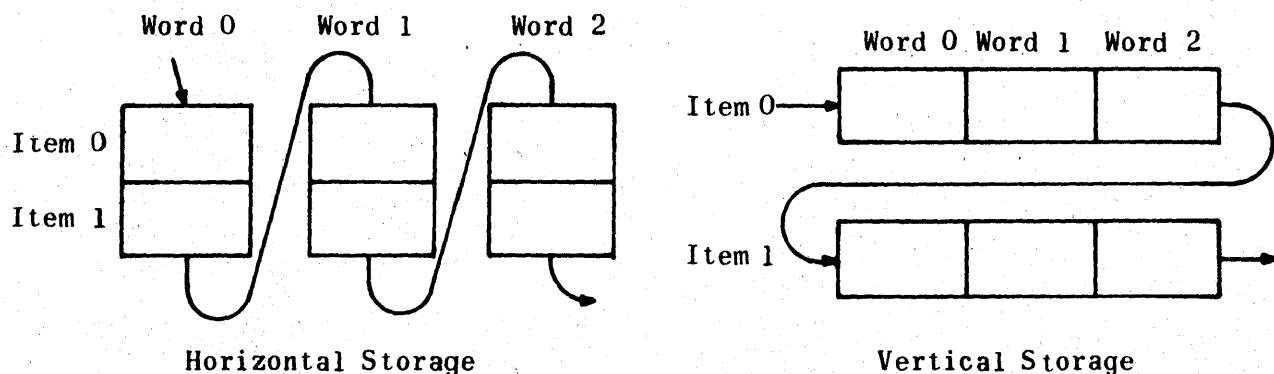


Figure 2-11. Table Storage Sequence

The CMS-2 table structure also allows the programmer to define a subset of adjacent items within a table as a subtable. The programmer may also allocate outside the table a working storage area, designated as an item-area, which will automatically take on the same field format as that defined for the table items. Additionally, the programmer may declare tables known as like-tables having identical field format as the parent table but having a different number of items. Figure 2-12 illustrates these described relationships to the parent table.

2.3.2.4 Arrays

An array is an extension of the table concept for storing ordered sets of identically structured information previously defined as items. Arrays may be conceptually visualized as rows, columns, and planes of items. An example of an array (three-dimensional) is presented in Figure 2-13. As with tables, the basic structural unit of an array is the item. The array item may consist of k computer words with fields defined as desired. The pattern for storage of an array within core is illustrated in Figure 2-14.

2.3.3 Compiler Directive Declaratives

Certain CMS-2 declarative statements specify control information to the Compiler. These declaratives direct Compiler action as to allocation mode, listing options, system index registers, program debug features, base numbering system interpretation, data pooling requirements, and the computer mode of operation within which the designated program is expected to run. These declarative statements may be located in major headers if the control applies to the entire compile-time system, in minor headers if the control applies to a system element, or within a system element. The rules for placement and range of effective action for the individual declaratives are defined fully in Section 7.

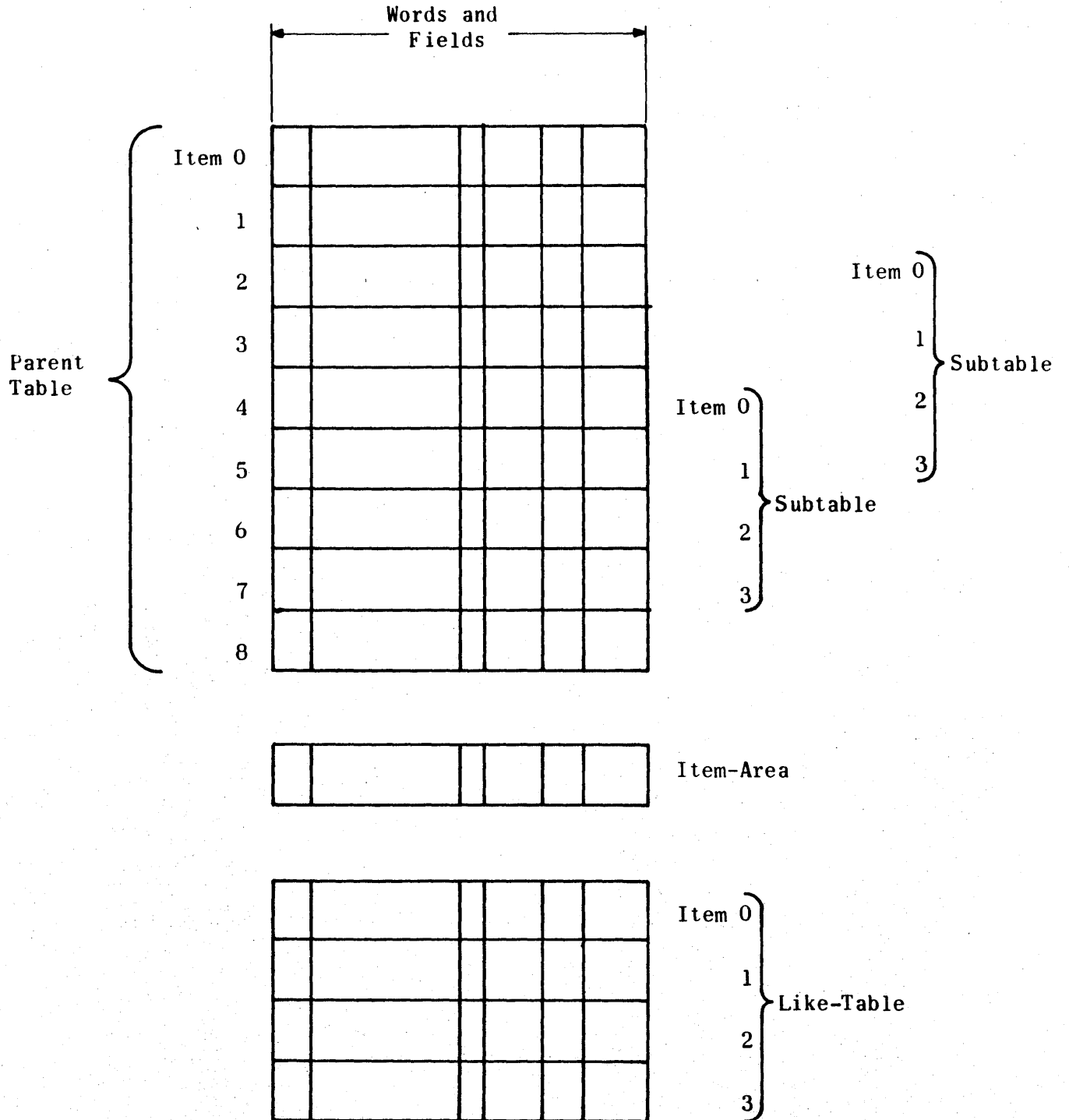


Figure 2-12. Parent Table Relationships

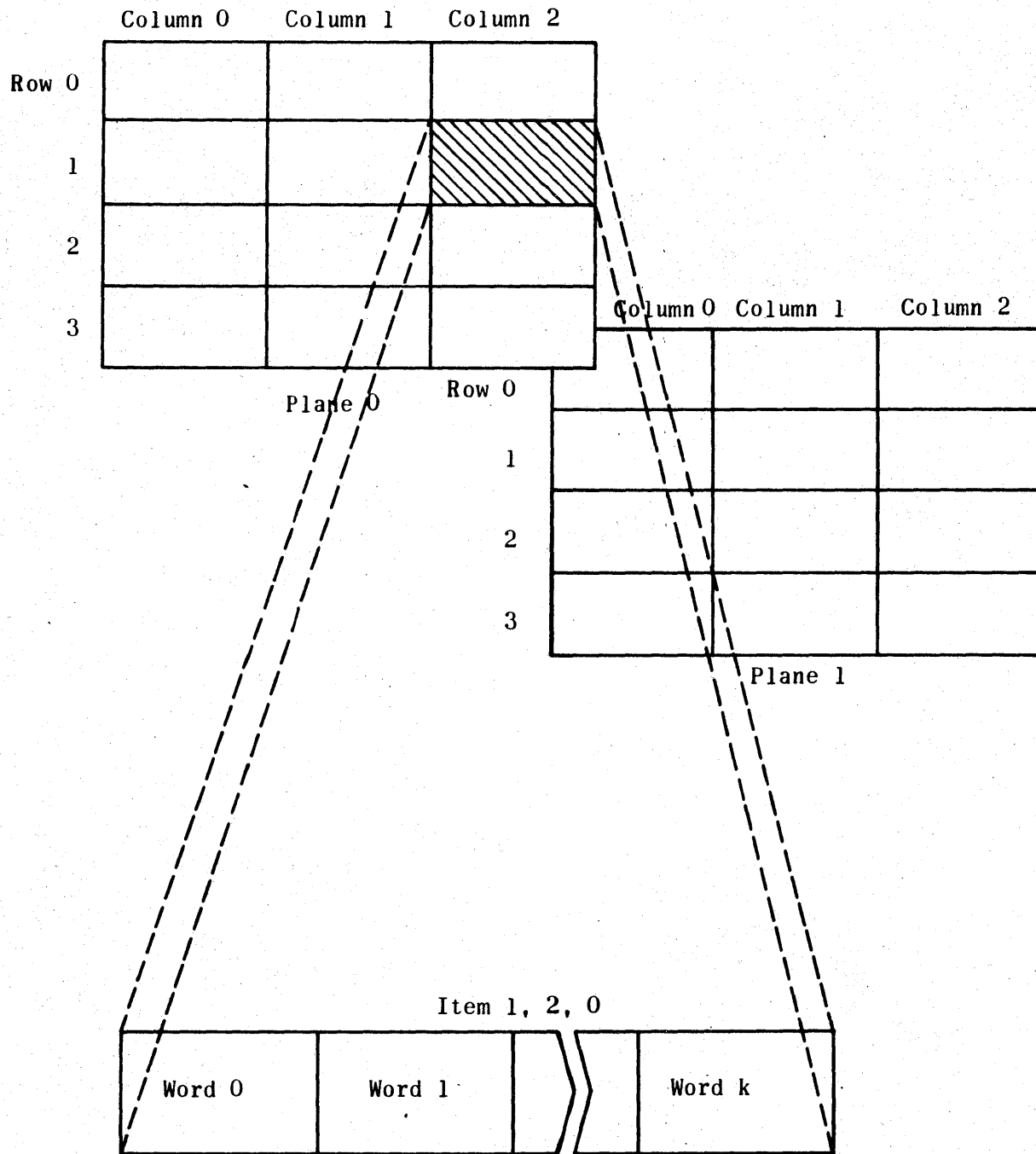


Figure 2-13. A Three-Dimensional Array

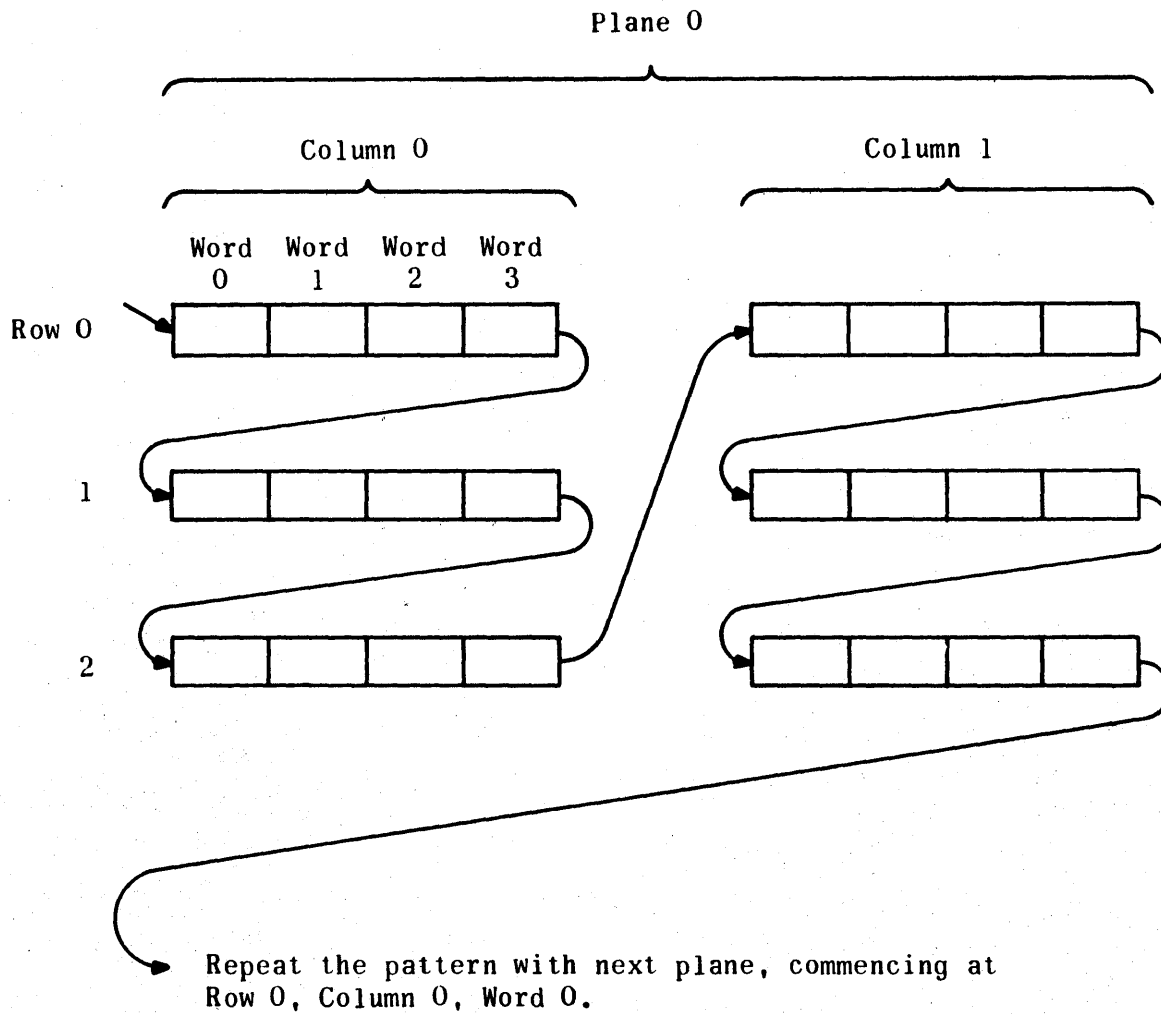


Figure 2-14. Array Storage Sequence

2.4 DYNAMIC STATEMENTS

CMS-2 dynamic statements specify processing operations and result in executable code generation by the Compiler. A dynamic statement consists of an operator followed by a list of operands and additional operators. An operand may be a single name, a constant, a data-element reference, or an expression.

2.4.1 Expressions

Arithmetic expressions may include standard addition, subtraction, multiplication, and division operators, as well as exponentiation, mixed-mode values, and inline redefinition of the scaling of fixed-point numbers. An algebraic hierarchy of operation evaluation is used. A relational expression performs a comparison between two similar operands as specified by a relational operator. There are four types of comparisons available:

1. Arithmetic, involving the comparison of signed arithmetic values (fixed, floating, or mixed).
2. Hollerith, involving a left-to-right, character-by-character comparison.
3. Boolean, involving single bit comparisons.
4. Status, involving the comparison of status values.

Arithmetic operators used in CMS-2 are + (addition), - (subtraction), / (division), * (multiplication), ** (exponentiation), and .. (inline scaling).

Relational operators are EQ (equal), NOT (not equal), LT (less than), GT (greater than), LTEQ (less than or equal) and GTEQ (greater than or equal).

Boolean operators used in CMS-2 are AND, OR, XOR (exclusive or) and COMP (logical not). A CMS-2 expression may include algebraic, relational, and Boolean operators.

2.4.2 Statement Operators

The CMS-2 statement operators allow the programmer to write his program in an easy-to-learn, problem-oriented language. Major CMS-2 operators and their functions are summarized on the following page.

<u>Operator</u>	<u>Function</u>
SET	Performs calculations or assigns a value to one or more data units. The assignment may be arithmetic, Hollerith, status, Boolean, or multi-word.
SWAP	Exchanges the contents of two data units.
GOTO	Alters program flow directly or via a statement switch.
IF	Expresses a test situation for conditional execution of one or more statements.
VARY	Establishes a program loop to repeat execution of a specified group of statements.
FIND	Searches a table for data that satisfies specified conditions.
PACK	Transfers bit strings into a data area.
SHIFT	Shifts a string of bits.
FOR	Selectively executes one of a set of statement blocks based on the value of a controlling expression.

2.4.3 Special Operators

Special operators are available in CMS-2 to facilitate references to data structures and operations on them. These operators and their functions are summarized below.

<u>Operator</u>	<u>Function</u>
BIT	To reference a string of bits in a data element.
CHAR	To reference a character string.
CORAD	To reference a core address.
ABS	To obtain the absolute value of an expression.
POS, FIL	To position a magnetic tape file.
LENGTH	To obtain an input/output file length.
CNT	To obtain a count of bits set.
CAT	To concatenate character strings.

2.5 HIGH-LEVEL INPUT/OUTPUT STATEMENTS

CMS-2 high-level input/output (I/O) statements permit the program to communicate with various hardware devices while running in a non-real-time environment under a Monitor system. When CMS-2 I/O statements are used by the programmer, the Compiler generates specific calls to run-time routines that must be loaded with the user's program. The run-time routines are designed to link with the Monitor system and communicate with its I/O drivers. I/O declarative and dynamic statement operators and their associated functions are summarized below.

<u>Operator</u>	<u>Function</u>
FILE	Defines the environment and pertinent information concerning an input or output operation, and reserves a buffer area for record transmission.
OPEN	Prepares an external device for I/O operations.
CLOSE	Deactivates a specified file and its external device, if appropriate.
INPUT	Directs an input operation from an external device to a FILE buffer area.
OUTPUT	Directs an output operation from a FILE buffer area to an external device.
FORMAT	Defines the desired conversion between external data blocks and internal data definitions.
ENCODE	Directs transformation of data elements into a common area, with conversion in accordance with a specified FORMAT.
DECODE	Directs unpacking of a common area and transmittal to data units as specified by a FORMAT declaration.
ENDFILE	Places an end-of-file mark on appropriate recording mediums.
CHECKID	Directs checking an ID header or label on a file.
DEFID	Directs the output of an ID header on a file.

2.6 PROGRAM DEBUG FACILITIES

CMS-2 debug statements may be placed in the source language of a user's program to assist in program checkout. These statements may reference any data units defined within the system. Machine code is generated by the Compiler to call on run-time debug routines. The debug routines communicate with the Monitor system during program execution to print the desired checkout data onto the system output device (high-speed printer).

Five program checkout statements are provided. Output code is generated only if the corresponding statements are enabled in the program header information. A programmer may then control and select the debug tools as needed. The debug operators and their functions are summarized below.

<u>Operator</u>	<u>Function</u>
DISPLAY	Causes the contents of machine registers and/or specified data units to be formatted and printed on the system output.
SNAP	The contents of a data unit are printed and stored. Subsequent executions cause a printout only when the data contents are modified.
RANGE	A high and low value are specified for a data unit. Each time the data is modified in the program, a message is printed if the value falls outside the range.
TRACE	A printout is generated for the execution of each CMS-2 statement between TRACE and END-TRACE boundaries.
PTRACE	Each CMS-2 procedure call encountered in the program being executed is identified by calling and called procedure names.

SECTION 3

BASIC DEFINITIONS

A CMS-2 program consists of an ordered set of sentences composed of symbols and delimiters. The symbols and delimiters are formed using characters from the CMS-2 alphabet.

3.1 CMS-2 ALPHABET

The CMS-2 alphabet consists of letters, digits, and marks as described below:

- a. Letter - One of the 26 letters of the English alphabet, A through Z, written in capital letter form.
- b. Digit - One of the ten Arabic numerals, 0 through 9.
- c. Mark - Any additional special character that may be input to the Compiler via the Monitor I/O routines. The commonly used marks that have significance to the CMS-2 Compiler are listed below, along with their common name:

+	(plus))	(right parenthesis)
-	(minus)	\$	(dollar sign)
/	(slash)	,	(comma)
*	(asterisk)	'	(prime)
.	(decimal point, period)	Δ	(space)
((left parenthesis)		(space)

3.2 SYMBOLS

CMS-2 symbols are composed of strings of one or more letters, digits, or marks from the CMS-2 alphabet. There are three types of symbols:

1. Operators - Indicating operations or specifications.
2. Identifiers - Names by which programs reference their environment.
3. Constants - Words that represent unchanging values (constants in the mathematical sense).

3.2.1 Operators

Operators are symbols that denote an action or delineation to the Compiler. They tell the Compiler "what to do" or "what it is" as opposed to other symbols that tell "where it is" or "how much it is".

The following symbols are examples of CMS-2 operators; the symbols are divided into five categories:

<u>Arithmetic</u>	<u>Relational</u>	<u>Boolean</u>	<u>Dynamic</u>	<u>Declarative</u>
+	EQ	AND	PROCEDURE	TABLE
-	LT	OR	FIND	FIELD
/	GT	COMP	SET	LOC-DD

A special class of operators provides machine control interface. These symbols are entirely machine dependent. For the AN/UYK-7 Computer, these symbols are: KEY1, KEY2, KEY3, STOP, STOP5, STOP6, and STOP7.

3.2.2 Identifiers

Identifiers are arbitrary names used to label various units of a CMS-2 program so that these units may be referred to by unique names. A name is composed of from one to eight letters and digits; the first character of a name must be a letter. All CMS-2 identifiers (except statement labels, procedure names, and abnormal exits) must be defined by or within a data declaration, which associates the identifier with its specific attributes.

In order to prevent ambiguities in the source input for a CMS-2 program, the Compiler does not allow the programmer to declare or define identifiers that duplicate operator symbols in the CMS-2 language. Appendix D presents a list of those symbols which are reserved words. These reserved words are not available to the programmer for use as identifiers. In addition, any programmer expecting to make use of CMS-2 run-time routines (high-level debug, input/output, or mathematical routines) should avoid the use of identifiers beginning with the characters "RT". This will prevent possible conflict at load time with global identifiers defined and referenced within the CMS-2 run-time library.

3.2.2.1 Statement Label

A statement label is a special identifier in a CMS-2 program; the statement label is used to label a dynamic statement. A statement label derives its definition by context, since it is always followed immediately by a period. When reference is made to the statement label during an operation within the program, the period is omitted and the label is then known as a statement name. More than one statement label may be applied to a dynamic statement.

NOTE

Statement labels may appear only on dynamic statements. Hence, the period following a name signifying a statement label may be used only between the PROCEDURE and END-PROC declarations and may never be used with direct code statement labels.

3.2.3 Constants

A constant denotes a value that is known at compilation time. CMS-2 programs manipulate the following four types of data:

1. Numeric values consisting of rational numbers.
2. Hollerith or literal values consisting of strings of characters from the CMS-2 alphabet.
3. Status values consisting of independent sets of arbitrarily named conditions.
4. Boolean values consisting of the two values: true or false.

3.2.3.1 Numeric Constants

A numeric value, positive or negative, may be represented by a decimal or octal constant as described below:

1. Decimal - Consists of one or more base-10 digits (0-9). This is the normal mode of the Compiler. The number enclosed in parentheses, preceded by the letter D, is also acceptable and may be used when a non-decimal mode is specified to the Compiler (see Section 7).

2. Octal - Consists of one or more base-8 digits (0-7) enclosed in parentheses and preceded by the letter O.

These constants may be preceded by a plus sign if positive and must be preceded by a minus sign if negative.

A radix point appearing within or at the beginning of the constant identifies the constant as a mixed number or fraction. The number of fractional bits attributed to the constant equals:

1. $3 \cdot 2^{n+1}$ truncated to an integer, if constant is decimal, or
2. $3 \cdot n$, if constant is octal

where n is the number of fractional digits. If the radix point is omitted or occurs at the end of the constant, it identifies the constant as an integer (whole number).

Examples

- a. -94 (negative decimal integer)
- b. O(77) (positive octal integer)
- c. 88.1 (positive mixed decimal number)
- d. -O(.64) (negative octal fraction)
- e. -D(492.3) (negative mixed decimal number)

To avoid writing many zeros, it is sometimes convenient to express a very large or very small numeric constant as a coefficient multiplied by an exponent.

Examples

- a. $.00023_8 = .23_8 * 10_8^{-3} = O(.23E-3)$
- b. $1800000_{10} = 18_{10} * 10^5 = 18E5$
- c. $15000_{10} = 1.5_{10} * 10^4 = 1.5E4$
- d. $7300_8 = 7.3_8 * 10_8^3 = O(7.3E3)$

Both the coefficient and the exponent must have the same base. If the number is octal, it must be preceded by the O descriptor.

3.2.3.2 Hollerith Constant

A Hollerith constant is composed of a string of characters enclosed by parentheses and preceded by the descriptor H.

Examples

1. H(NOTNOW)
2. H(REWINDΔΔ)
3. H()LAST)

In the second example, the two blanks are considered part of the constant. The third example illustrates the use of a right parenthesis as part of the constant within a string of characters. Each right parenthesis must be represented by two consecutive right parentheses since the string is terminated by a single right parenthesis. Encoding this constant results in the characters:

)LAST

Any character, including blank, is a valid character in the Hollerith set and may be used in source programs to construct character-string constants.

NOTE

If a Hollerith constant appears as the last term of a parenthesized expression, at least one blank must separate the right parenthesis signifying the end of the Hollerith constant from the right parenthesis signifying the end of the expression.

3.2.3.3 Status Constants

A status constant is a mnemonic used to describe one of the possible values of a data unit. The Compiler assigns a unique value (beginning with zero) to each status constant that is associated with a data unit. In subsequent statements, as the programmer sets and tests the data unit using the mnemonic, the Compiler substitutes the assigned value to differentiate possible conditions. Status constants must be unique for a given data unit but may be reused for other data units.

A status constant may be composed of any characters of the CMS-2 alphabet with the exception of a single prime ('). The status constant may have the same number of characters as an identifier. Status constants are always enclosed by single primes, as illustrated on the following page.

'REPAIR'
'STANDBY'
'ALERT'
'AIRBORNE'

3.2.3.4 Boolean Constants

A Boolean constant denotes one of the logical values of Boolean algebra (true or false) and is represented as a binary integer:

<u>Logical</u>	<u>Binary</u>
True	1
False	0

3.3 DELIMITERS

Blanks serve to separate symbols in a CMS-2 program. When used as a separator, a single blank accomplishes the same result as a sequence of two or more blanks. All marks described in paragraph 3.1 may be used as delimiters. Some marks, such as \$, have unique delimiting uses. When a mark appears between two CMS-2 symbols, blanks are not needed as separators although they may be used if desired.

3.4 STATEMENTS

CMS-2 statements are dynamic and declarative and are composed of a string of symbols and delimiters. In general, a declaration defines a structural configuration of data and a dynamic statement defines the processing operation that manipulates the data. All CMS-2 statements are terminated by a dollar sign (\$). More than one statement may appear on one card and a statement may be continued on several cards. Null statements are recognized by the compiler when a statement terminator is immediately followed by a statement terminator and is otherwise syntactically correct.

3.5 COMMENTS

Comments, intended as clarifying text, have no operational effect on a program and may be included in either of the following two ways.

1. Within a statement by enclosing the comment within two consecutive single-prime symbols, as illustrated below:

```
VRBL Z I'NTEGER" 14 "BITS" S'IGNED" $
```

NOTE

A symbol may not be broken by this type of comment; i.e., V"A"R"IA"BL"E" would not result in the symbol VRBL.

2. As a separate statement the use of the operator, COMMENT:

```
COMMENT THIS ROUTINE COMPUTES SQUARE ROOTS $
```

NOTE

A dollar sign may be expressed within either type of comment by coding it as two consecutive dollar signs.

3.5.1 Special Comments

Comments beginning in card column 11 (the first column of the statement field) are treated as special cases by the Compiler. If the statement is one of the following, the Compiler performs the indicated action on the listing:

<u>Input</u>	<u>Action</u>
COMMENT△(EJECT \$	Eject to the top of the next listing page.
COMMENT△(LINE* \$	Print a line of asterisks (*).
COMMENT△(SKIPn \$	Skip n lines, where n is a number from 1 to 9.

If a comment statement beginning in card column 11 is not one of the special indicators, the Compiler replaces the word COMMENT with a single asterisk (*) in column 11 and lists the comment after skipping a line.

Input:

```
COMMENT THIS IS AN EXAMPLE $
```

Output:

```
* THIS IS AN EXAMPLE $
```

These special comments allow programmers to produce listings that have a greater narrative format than listings without the special comments feature.

3.6 SOURCE CARD FORMAT

All CMS-2 source cards contain a card identification field and a statement field. The card identification field occupies columns 1 through 10; the statement field occupies columns 11 through 80.

Card columns 1 through 10 have no operational effect on the Compiler. However, the suggested use of the card identification field is as follows:

- a. Columns 1 through 4 - Program identification.
- b. Columns 5 through 8 - Card sequence number.
- c. Columns 9 through 10 - Insert number.

The statement field has a free format. Statement labels, operands, operators, etc., may occur anywhere in columns 11 through 80. Each statement is terminated with a \$. There can be more than one statement per card or a statement may require several cards. No continuation card indicator is needed when a CMS-2 statement exceeds one card. The statement continues in columns 11 through 80 of each succeeding card until a dollar sign is encountered. If a symbol or contiguous string of characters is to span two cards, the first part must end in column 80 of card 1 and the second part must start in column 11 of card 2. For example, if the eight-character symbol STMTLAB1 is started in column 78 of one card, the remaining five characters must begin in column 11 of the next card.

While packing of statements on cards reduces the size of the input deck, the CMS-2 Compiler does not format the listing of the input statements. Packed statements will appear in the same form on the listing.

SECTION 4

DECLARATIVES

The CMS-2 declarative statements provide the Compiler with information about program structure and data element definitions. Declaratives generally do not result in executable object code. Declaratives may be divided into three groups: program structure declaratives, data declaratives, and control declaratives.

4.1 PROGRAM STRUCTURE DECLARATIVES

The following program structure declaratives are used to define the organization of a CMS-2 program:

- a. SYSTEM and END-SYSTEM statements delimit a compile-time system.
- b. HEAD and END-HEAD statements delimit headers within a compile-time system.
- c. SYS-DD and END-SYS-DD statements delimit a system data design within a compile-time system.
- d. SYS-PROC (or SYS-PROC-REN) and END-SYS-PROC statements delimit a system procedure within a compile-time system.
- e. LOC-DD and END-LOC-DD statements delimit local data designs within a system procedure.
- f. AUTO-DD and END-AUTO-DD statements delimit automatic data designs within a system procedure.
- g. PROCEDURE and END-PROC statements delimit procedures within a system procedure.
- h. FUNCTION and END-FUNCTION statements delimit functions within a system procedure.

Each of these statements is discussed in detail in the following paragraphs in the order in which they generally occur in a CMS-2 source program (see Section 2 for further information on program organization).

4.1.1 System Declarative (SYSTEM)

The SYSTEM declarative specifies the beginning of a compile-time system. This must always be the first statement of a CMS-2 source program.

Format

```
name SYSTEM key-specification comments $
```

Explanation

Name The identifier by which this system is known.

SYSTEM Declares a compile-time system to be known by the identifier above.

Key specification Optional (see Section 7 for explanation).

Comments Programmer remarks. Optional.

4.1.2 Head Declarative (HEAD)

The HEAD declarative serves to identify a group of major or minor Compiler control statements. The major header control statements of a compile-time system must immediately follow the SYSTEM declarative and must be terminated with an END-HEAD declarative. Minor header control statements immediately precede the system data design or system procedure to which they apply. Since the HEAD declarative is primarily for library control purposes, its use in source input to the Compiler is generally optional (see Section 7 for further information on the HEAD statement).

Format

```
name HEAD key-specification comments $
```

Explanation

Name The identifier by which this header is known.

HEAD Declares a major or minor header.

Key specification Optional (see Section 7 for explanation).

Comments Programmer remarks. Optional.

4.1.3 End Head Declarative (END-HEAD)

This declarative terminates a major or minor header within a compile-time system. Its use is required after major header control statements but is optional after minor header control statements.

Format

END-HEAD name \$

Explanation

END-HEAD Declares the end of a header.

Name The header identifier. Optional.

Example

```

SAMPLE | SYSTEM $ |
|-----|-----|
|       | OPTIONS SOURCE $ |
|-----|-----|
|       | END-HEAD $ |
|-----|-----|
H1 | HEAD $ |
|-----|-----|
|       | ATTR MEANS A 32 S 10 $ |
|-----|-----|
|       | END-HEAD H1 $ |
|-----|-----|
DATA1 | SNS-DD $ |
|-----|-----|

```

4.1.4 System Data Design Declarative (SYS-DD)

This declarative specifies the beginning of a collection of data element definitions that are global to the system; that is, these data elements are known to all system procedures that follow in the compile-time system. A system data design is a basic element of a CMS-2 program.

Format

name SYS-DD key-specification comments \$

Explanation

Name The identifier by which this system data design is known.

SYS-DD The system data design declarative.

Key specification Optional (see Section 7 for explanation).

Comments Programmer remarks. Optional.

4.1.5 End System Data Design Declarative (END-SYS-DD)

This declaration terminates a system data design within a compile-time system.

Format

END-SYS-DD name \$

Explanation

END-SYS-DD Declares the end of a system data design.

Name The identifier by which this system data design is known.

4.1.7 Reentrant System Procedure Declarative (SYS-PROC-REN)

This declarative specifies the beginning of a special type of system procedure known as a reentrant system procedure. The generated code produced by the Compiler for all procedures within a reentrant system procedure is invariant under execution (see Section 2 for a further explanation of reentrant code). A reentrant system procedure consists of one or more procedures and may contain one or more local data designs or automatic data designs.

Format

Name SYS-PROC-REN key-specification comments \$

Explanation

Name	The identifier by which the system procedure is known. It also identifies the prime procedure within the system procedure.
SYS-PROC-REN	Initiates a reentrant system procedure.
Key specification	Optional (see Section 7 for explanation).
Comments	Programmer remarks. Optional.

4.1.8 Local Data Design Declarative (LOC-DD)

This declarative specifies the beginning of a set of data element definitions that are valid only within the system procedure in which this local data design appears. Such data elements must be defined in a local data design before they may be referenced by a dynamic statement within a procedure.

Format

name LOC-DD comments \$

Explanation

Name	The identifier by which the local data design is known. Optional. The name has relevance when LOCDDPOOL is requested. The user must provide a LOC-DD name at compile time if he wants to reference the pooled local data design by AC name at load time.
LOC-DD	Declares the start of a local data design.
Comments	Programmer remarks. Optional.

4.1.9 Local Data Design Declarative (END-LOC-DD)

This declarative specifies the end of a local data design within a system procedure.

Format

END-LOC-DD name \$

Explanation

END-LOC-DD Declares the end of a local data design.

Name The identifier by which the local data design is known.
Optional.

4.1.10 Automatic Data Design Declarative (AUTO-DD)

This declarative specifies the beginning of a set of data element definitions that are valid only within the reentrant system procedure in which this automatic data design appears. An automatic data design may appear only within a reentrant system procedure (i.e., it must follow a SYS-PROC-REN declaration). All formal input and output parameters and temporary data storage areas used by reentrant procedures that follow must be declared between the AUTO-DD and END-AUTO-DD declaratives. The allocation for these data areas must be provided dynamically prior to or during execution; hence, automatic data designs may not contain switch definitions or preset data.

Format

name AUTO-DD comments \$

Explanation

Name The identifier by which the automatic data design is known.

AUTO-DD Declares the start of an automatic data design within a reentrant system procedure.

Comments Programmer remarks. Optional.

NOTE

Automatic data designs are functionally similar to local data designs except that they may be used only within a reentrant system procedure and may not contain preset data or switches. Automatic data designs should contain the definitions of all data units that are modified during execution of reentrant procedures. Local data designs may be used within reentrant system procedures for defining switches and preset data that are not modified during execution.

4.1.11 End Automatic Data Design Declarative (END-AUTO-DD)

This declarative specifies the end of an automatic data design within a reentrant system procedure.

Format

END-AUTO-DD name \$

Explanation

END-AUTO-DD Declares the end of an automatic data design.

Name The identifier by which the automatic data design is known.

4.1.12 Procedure (PROCEDURE) and End Procedure (END-PROC) Declaratives

A procedure is the basic organizational unit of dynamic statements in a CMS-2 program; it establishes the rules of data manipulation in processing a problem. Procedures specify a sequence of operations which appear only once in the source program but which may be invoked at various points throughout the program. The PROCEDURE declarative specifies the beginning entry point of a procedure and supplies further identifying information to the Compiler. The end of the procedure is indicated by the END-PROC declarative. Procedures may be called by name from other procedures within the same system procedure, or from other system procedures if the procedure is externally defined. Procedures may have input, output or exit parameters, which are passed from or to the calling procedure.

Format

```

PROCEDURE name INPUT formal-parameters OUTPUT
      formal-parameters EXIT abnormal-exit-name(s) $
      .
      .
      .
steps of the procedure
      .
      .
      .
END-PROC name $

```

Explanation

PROCEDURE	Delimits (with associated END-PROC) a procedure and establishes an entry point for the procedure.
Name	An identifier by which the procedure is referenced.
INPUT	Optional. Specifies that the list of formal parameters that follows is input to the procedure.
Formal Parameters	Optional. Data unit names separated by commas. They are input or output parameters that have been previously defined in a data design. They establish the structure of parameters and provide a legality check on procedure calls. Formal parameters may be variables, system indexes, entire tables, like-tables, subtables, or item areas. They may not be subscripted data units, expressions, constants or functionally modified data units. A formal parameter may not be a LOC-INDEX. (See paragraph 4.2.2, Example 5 for an explanation of the allowable use of the CORAD operator in the formal parameter list.)

OUTPUT Optional. Specifies that the list of formal parameters that follows is output as a result of the procedure operation.

EXIT Optional. Specifies that one or more abnormal exit names follow.

Abnormal Exit Name(s) Optional. Identifies the abnormal exit(s). Abnormal exit names appear only as operands of a RETURN operation within the procedure and must be unique. If more than one is specified, they must be separated by commas.

END-PROC Specifies the end of the procedure identified by name.

Example

```

PROCEDURE TEST INPUT V1, V2 OUTPUT VALUE
EXIT BADV1, BADV2 $

```

(steps of procedure)

```

END-PROC TEST $

```

In this example, the name of the procedure is TEST. It has two formal input parameters, V1 and V2, which will contain input values when the procedure is entered. The formal output parameter, VALUE, will be set appropriately by the procedure prior to returning to the calling program. The two abnormal exit parameters require alternative return points in the calling program to be specified when procedure TEST is called. Refer to Section 5 for further information on procedure calls.

4.1.13 Function (FUNCTION) and End Function (END-FUNCTION) Declaratives

The function is a special class of procedure. While a procedure call has a specific CMS-2 statement form (see Section 5), the function is called implicitly by using its name in a dynamic statement in much the same way as a data unit is referenced (see Section 5). The steps of the function are delimited by the FUNCTION and END-FUNCTION declarations. A function must have at least one input parameter and always results in a single output value. The function must specify a data unit name or an expression as a parameter on a RETURN statement to indicate the output value (see Section 5). A function may be the prime procedure or the only procedure of a system procedure if desired.

Format

```

FUNCTION name (formal input parameters) type $
.
.
.
steps of the function
.
.
.
END-FUNCTION name $

```

Explanation

FUNCTION	Specifies the beginning of a function definition.
Name	The identifier used to reference the function.
Formal Input Parameters	The names of variables, tables, like-tables, subtables, or item-areas that have been previously defined in a data design. A function must have at least one formal input parameter; if more than one is specified, they must be separated by commas.

Type Optional. Specifies the attributes of the output value of the function. When this information is omitted, the attributes of the output value are determined by the implied mode of the Compiler for variables (a signed 16-bit integer, unless superseded by a MODE declaration). When specified, the type parameter must be one of the following:

F	Floating-point value
B	Boolean value
H followed by the number of characters	Hollerith value having the indicated number of characters (not to exceed eight)
S followed by a list of status constants separated by commas	Status value that can assume any of the states corresponding to the listed status constants
A followed by the total number of bits (not to exceed 64), the designator S or U (signed or unsigned), and the number of fractional bits	Fixed-point value
I followed by the total number of bits (not to exceed 64), and the designator S or U (signed or unsigned)	Integer value

Steps of the Function CMS-2 dynamic statements that perform the operations of the function.

END-FUNCTION Specifies the end of the function definition.

Example

```

FUNCTION TPOS(AZM) I 32 S $
SET ALPHA TO (3*AZM/15)**BETA $
IF ALPHA GTEQ 50 THEN
GOTO TPOS1 $
RETURN (4*ALPHA) $
TPOS1. SET ALPHA TO 4*(ALPHA/3) $
RETURN (ALPHA) $
END-FUNCTION TPOS $

```

In this example, the output value of the function is a signed 32-bit integer that depends on the value of the input parameter AZM.

4.1.14 End System Procedure Declarative (END-SYS-PROC)

This declarative specifies the end of a system procedure.

Format

END-SYS-PROC name \$

Explanation

END-SYS-PROC Declares the end of the system procedure.

Name The name of this system procedure assigned by the SYS-PROC statement.

4.1.15 End System Declarative (END-SYSTEM)

This statement declares the end of a compile-time system.

Format

END-SYSTEM name \$

Explanation

END-SYSTEM Declares the end of the preceding system.

Name The name of the system assigned by the SYSTEM statement.

4.2 DATA DECLARATIONS

Data declarations define the structure and order of data elements within a compile-time system and provide a means for referencing these elements. A thorough understanding of data declarations is necessary to write efficient and accurate CMS-2 programs.

There are five major types of data declarations in CMS-2:

1. Variables.
2. Tables (together with fields, item-areas, subtables, and like-tables).
3. Indexes.
4. Switches.
5. Files.

These declarations, with the exception of files, are discussed in the following paragraphs. Files are described in Section 6 with the input/output statements.

Within any range of definition, no two data elements may have the same name. A name defined in a system data design may not be duplicated within the compile-time system (or within any system to be loaded simultaneously in which the name is externally defined). A data definition made within a local data design or a statement label specified within a procedure may not be duplicated within that system procedure.

An exception is made for fields. The definition of a field is always local to a table. A field name may not be duplicated within the same table, but may appear in as many tables as desired.

4.2.1 Variable Declaration (VRBL)

A variable is a single quantity of data. The variable name identifies a given location containing the quantity of data. A variable may contain a constant value or its value may continually change during program execution. The data may occupy one bit, part of a word, one whole word, or many words.

Variables may be any of the following six data types:

- | | | |
|-------------------|---|-------------------|
| 1. Integer | } | Computational |
| 2. Fixed-Point | | |
| 3. Floating-Point | | |
| 4. Boolean | } | Non-computational |
| 5. Hollerith | | |
| 6. Status | | |

Format

VRBL name(s) type (R) initial-value V(x,y) \$

Explanation

VRBL Indicates that the definition of one or more variables follows.

Name(s) A unique identifier by which the variable is referenced. Multiple names (maximum 25) may be specified by separating them with commas and enclosing the list in parentheses. The Compiler then allocates locations for each of the names according to the description (implied or specific) that follows.

Type

Optional. May be one of the following:

F	Floating-point
B	Boolean variable
H, followed by the number of characters (not to exceed 132)	Hollerith variable having the indicated number of characters
S, followed by a list of status constants separated by commas	Status variable that can assume any of the states corresponding to the listed status constants
A, followed by the total number of bits (not to exceed 64), the designator S or U (signed or unsigned), and the number of fractional bits	Fixed-point variable
I, followed by the total number of bits (not to exceed 64), and the designator S or U (signed or unsigned)	Integer variable

In the absence of a preceding MODE declarative statement and in the absence of one of the above listed data type specifications, the Compiler will imply and allocate the data type as a signed 16-bit integer.

NOTE

The descriptors H, A and I may be followed by tags representing the various numeric parameters. Tag is defined in Section 7 under the EQUALS definition.

4. VRBL FLOAT F P -O(17.77) \$

The variable FLOAT is a floating-point variable. The descriptor P indicates that the initial value of the variable is -17.77 octal.

5. VRBL (CTR,CTR1,CTR2,PTR1) A 24 U 12 P 0 \$

The variables CTR, CTR1, CTR2, PTR1 are all fixed-point (A). They each contain 24 unsigned bits and 12 fractional bits. The initial value of all the variables is 0 as indicated by the descriptor P.

6. VRBL INTR I 5 U \$

The variable INTR is an unsigned integer variable of 5 bits.

7. VRBL BOOL B P 1 \$

The variable BOOL is a Boolean variable initially set to the true (1) condition.

8. VRBL GAS S 'FULL','LOW','EMPTY' P 'FULL' \$

The variable GAS is a status variable that may assume the three states FULL, LOW, and EMPTY. The Compiler will assign values to these three states and assign these values to the variable as it subsequently encounters these states in conjunction with the variable GAS (for example, see the SET statement). The variable will be preset to the value associated with FULL.

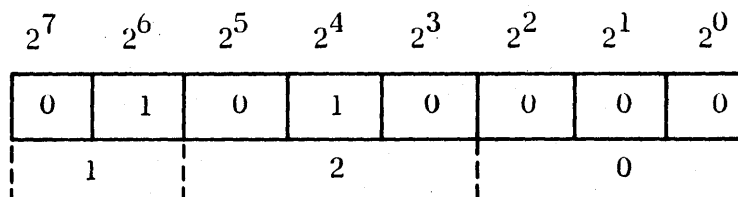
Note that the states of a status variable are always enclosed within single primes.

9. VRBL FLT F (R) \$

The variable FLT is a floating-point variable. (R) specifies that those AN/UYK-7 instructions providing floating-point operations with rounding are to be used.

10. VRBL COURSE I 8 U P 112.5 V(180,7) \$

Variable COURSE is an unsigned integer variable, eight bits long. The variable is to be preset with the value 112.5. V(180,7) specifies that bit 2^7 is to represent 180. From this specification, bit $2^6 = 90$, $2^5 = 45$, $2^4 = 22.5$, $2^3 = 11.25$, etc. Thus, to preset the value to 112.5 requires bits 2^6 and 2^4 to be set ($90 + 22.5 = 112.5$), as illustrated below.



The internal representation will be the octal number 120. See Table 4-1 for further examples.

4.2.1.1 Parameter Declaration (PARAMETER)

The PARAMETER statement declares a variable as in the VRBL declaration but in addition associates with it registers to be used for parameter passage on any call to a procedure (does not apply to function calls) with a formal parameter defined via this PARAMETER statement.

NOTE

The PARAMETER declaration is designed to accommodate users with previously assembled sets of utility routines that require specific A-registers for input parameters and which output results in specific A-registers. Because of this intended use the compiler does not assume the responsibility for destruction of partial results held in A-registers when formal input or output lists in the procedure declaration contain a mix of PARAMETER variables and normal variables, or when expressions are used as actual parameters when calling a procedure with PARAMETER variables in its formal input list.

Format

PARAMETER name type (R) initial-value V(x,y) , register-number \$

Explanation

PARAMETER Indicates that the definition of a variable follows.

Name

Type

R

Initial Value

V(x,y)

} The explanation of these terms are given in paragraph 4.2.1 for the VRBL declaration. Type may not specify more than two words.

Register Number An integer (0-7) or an EQUALS tag having a value (0-7).

Example

A comparison of the code generated when using the PARAMETER statement and that generated when using the VRBL statement:

PARAMETER X I 32 S, 0 \$	VRBL X I 32 S \$
PARAMETER Y I 32 S, 6 \$	VRBL Y I 32 S \$

Procedure declaration:

PROCEDURE P INPUT X OUTPUT Y \$	PROCEDURE P INPUT X OUTPUT Y \$
SA A0, X, K3	:
:	:
LA A6, Y, K3	:
END-PROC	END-PROC

Procedure call:

P INPUT A OUTPUT B \$	P INPUT A OUTPUT B \$
LA A0, A, K3	LA A0, A, K3
LBJ B6, P	SA A0, X, K3
SA A6, B, K3	LBJ B6, P
	LA A0, Y, K3
	SA A0, B, K3

4.2.2 Table (TABLE) Declaration

A table is an ordered set of data consisting of equal and adjacent subsets (basic units) called items. There is no limit to the number of items within a table.

All items of any one table contain exactly the same number of words and have the same data structure. Items are identified sequentially within a table, the first item number being 0 and the last $n-1$, where n equals the number of items. An optional counter (the major index) which is Compiler-allocated with maintenance responsibility residing with the programmer, may be specified. If maintained, it will contain the actual number of items within the table that contains meaningful data at any given time.

A one-dimensional table is arranged in either a vertical or horizontal alignment. The vertical table arrangement of data permits rapid searches on selected words of any one item. The horizontal table arrangement of data permits rapid searches on one word or one field of all items.

TABLE 4-1. EXAMPLES OF VARIABLE DECLARATIONS

TYPE OF VARIABLE	FORMAT OF VARIABLE DECLARATIONS						
	VRBL	Name(s)	Data Type	Rounding	Initial Value	Preset Scaling	\$
Integer	VRBL VRBL VRBL VRBL	INTA (INTB, INTC) INTD INTE	I 5 U I 16 S I INTDBITS U I 8 U	X	P 0(12345) P 1.25	 V(.25,0)	\$ \$ \$ \$
Floating-Point	VRBL VRBL VRBL	FLTA FLTB FLTC	F F F	(R)	P 12.75 P 15E9	X	\$ \$ \$
Fixed-Point	VRBL VRBL	FIXA (FIXB, FIXC)	A 12 S 4 A FIXBITS U FIXFRAC	X	P 0	X	\$ \$
Boolean	VRBL VRBL VRBL	BOOLA BOOLB (BOOLC, BOOLD)	B B B	X	P 1 P 0	X	\$ \$ \$
Status	VRBL VRBL	STA (STB, STC, STD)	S 'LOW', 'MEDIUM', 'HIGH' S 'NONE', 'ONE', 'FEW', 'MANY'	X	P 'NONE'	X	\$ \$
Hollerith	VRBL VRBL VRBL	HOLA HOLB (HOLC, HOLD)	H 7 H HOLCHAR H 1	X	P H(ABCD)	X	\$ \$ \$
Implied	VRBL	IMPL		X		X	\$

NOTES

1. Tags such as INTDBITS, FIXBITS, FIXFRAC, and HOLCHAR may be used as indicated but must be assigned integer values by an EQUALS declaration.
2. If the type of variable is implied (type-structure omitted) the attributes of the variable are determined by the implied mode of the Compiler to be a signed 16-bit integer, unless superseded by a MODE declaration.

A multidimensional table (array) is stored forward in memory, in order of increasing absolute location, with the leftmost subscript (which represents the row or item) varying most rapidly. Thus, a two-dimensional array may be said to be stored in a columnar fashion. All the defined subcomponents of a given table (fields, subtables, like-tables, and item-areas) must be defined between the TABLE declaration and its associated END-TABLE declaration. Subtables and like-tables are not allowed as subcomponents of an array. Subtables, like-tables, and item-areas cannot be included in a table that is variable in length or that uses indirect addressing (INDIRECT).

In the following discussion of table format, the term tag is indicated as an option to specify integer values for the various parameters associated with the declaration of a table. When this option is exercised, the value supplied by the tag must be provided by an EQUALS declarative (see Section 7).

Format

TABLE name storage-type words-per-item or packing-descriptor
INDIRECT number-of-items or dimensions major-index-name \$

Explanation

TABLE	Specifies a TABLE declaration.
Name	An identifier unique to the TABLE.
Storage Type	Specifies the storage alignment desired according to one of the following types:
	H Horizontal arrangement
	V Vertical arrangement
	A Array arrangement in n-dimensions
Words-per-Item	An integer constant or tag that specifies the number of words contained in each item. This value cannot be 0.

Packing Descriptor Specifies that the Compiler will compute the number of words per item necessary to contain the specified fields, as follows:

<u>Descriptor</u>	<u>Result</u>
NONE	Each field is assigned at least a full word.
MEDIUM	Each field is assigned the smallest available, directly referable word fragment that will hold the data.
DENSE	Fields will be packed by the Compiler in a dense manner, making optimum use of all bits in a word.
(data type)	The Compiler will assign the number of words required to accommodate the attributes of the specified data type (see explanation under FIELD declaration for data type). This parameter permits referencing a single piece of data by item. Field declarations must specify starting bit and word number when used with typed-item tables.

INDIRECT Optional. Specifies that the table is indirectly referenced. Furthermore, no core allocation is made, the names and definitions are preserved, and the core allocation can be accomplished dynamically.

4. TABLE | RAY, A, 2, INDIRECT, 2, 3, 4, \$

This example defines an array, RAY, of which each item contains two words. Its dimensions are 2 by 3 by 4, or 24 items. The word INDIRECT indicates that no core allocation is to be made. All references to table RAY will be made by indirect addressing.

Following is an example of the use of INDIRECT and a major index.

Assume that a procedure processes data tables of fixed format and varying lengths up to a maximum of 100 entries. The data design associated with the procedure contains a table declaration:

5. TABLE | TABA, H, 2, INDIRECT, 100, J1, \$

This declaration specifies a horizontal table of two words per item, 100 items maximum, that has not been allocated core storage. The table has a major index J1, which will contain the actual number of items at run-time as provided by the dynamic source statements. This declaration may be accompanied by FIELD declarations as desired.

The procedure itself might require as input the location of the calling program's data table and the number of items in the following table:

PROCEDURE, OUTDO, INPUT, CORAD, (TABA), J1, \$

The calling procedure puts data in its associated data table. The procedure OUTDO is then called:

OUTDO, INPUT, CORAD, (MYBUF), BUFITEMS, \$

where MYBUF is the name of the caller's data buffer which has a structure compatible with TABA, and BUFITEMS contains the number of items in MYBUF.

The procedure OUTDO thus has its TABA defined dynamically as the user's data table for this operation of OUTDO. Each subsequent request for OUTDO would provide similar entry conditions (see Section 5).

6. TABLE TRACK V 1 0 \$

The table TRACK is vertical with one word per item. TRACK is a "null" table because it is declared with zero items. No core locations will be reserved.

7. TABLE FLOATDAT V (F) 1000 \$

Table FLOATDAT has been declared vertical and 1,000 items long; the type of data associated with the table is floating-point. The Compiler will assign two words per item. References to the table may be by item and the Compiler will utilize the declared data type (floating-point).

8. Following is an example of the use of tags within a table declaration:

```
NUMWORDS EQUALS 1023 $
```

```
NUMITEMS EQUALS 25 $
```

```
TABLE LARGE V NUMWORDS INDIRECT NUMITEMS $
```

In the above example, the names NUMWORDS and NUMITEMS represent 1023 and 25 respectively. The statement is interpreted as:

```
TABLE LARGE V 1023 INDIRECT 25 $
```

The table LARGE is arranged vertically, with 1,023 words per item and 25 items. Dynamic core allocation is specified by the INDIRECT specifier.

4.2.3 Field (FIELD) Declaration

The items of a given table may be further subdivided into units called fields. Field configurations are identical for all items of a given table. A field may occupy a partial word, a whole word, or more than one word. A field defined as part of a word must be wholly contained within that word; i.e., it may not cross word boundaries. A field occupying more than one word will

be allocated an integral number of words. Multiword fields are not permitted in horizontal tables. Field definitions within an item are completely independent of one another and may, therefore, overlap. It is not necessary for all of the data within an item to be completely defined by fields. A field name is always associated with a particular table, like-table, subtable or item-area.

Format

There are two basic formats for defining fields:

1. Type a

```
FIELD name data-type (R) word-location starting-bit-position
      initial-value V(x,y) $
```

2. Type b

```
FIELD name data-type (R) initial-value V(x,y) $
```

Explanation

FIELD Specifies the FIELD declaration.

Name The identifier used to reference the field within the table. Field names are local to the table within which they are defined. The same names may, therefore, be used for fields within various tables. The same name may not be duplicated within the same table definition.

<u>Data Type</u>	<u>Descriptor</u>	<u>Meaning</u>
	F	Floating-point field.
	B	Boolean field.
	H followed by the number of characters (not to exceed 132).	Hollerith field having the indicated number of characters.
	S followed by a list of status constants separated by commas.	Status field, which can have any of the states from the listed status constants.

<u>Descriptor</u>	<u>Meaning</u>
A followed by the number of bits (not to exceed 64), designator S/U (signed or unsigned), and the number of fractional bits.	Fixed-point field.
l followed by the number of bits (not to exceed 64) and the designator S/U (signed or unsigned).	Integer field.

NOTE

If the data type specification is omitted, the field will be assigned the implied mode (i.e., by a MODE declaration or Compiler-provided signed 16-bit integer).

(R)	Optional. This parameter is meaningful only when a floating-point data type is specified. If specified, it indicates that AN/UYK-7 floating-point instructions with rounding are to be used for arithmetic operations of this field.
Word Location	An integer or tag indicating the word of the item in which the field occurs. This value must not be greater than 255.
Starting Bit Position	An integer or tag specifying the most significant bit or sign bit of the field. In an n-bit word, the positions are numbered from left (n-1) to right (0).
Initial Value	Optional. This field may be preset with this parameter. If specified, this parameter consists of a P followed by one or more constants. These constants must be a value compatible with the data type specified for the

7. FIELD FARITH F (R) 0 31 \$

In this example, field FARITH is a floating-point data type, two words long, starting at bit 31 of word 0. The (R) signifies that arithmetic operations using this field must employ floating-point instructions with rounding. Note that a floating-point field must never be defined in a horizontal table since the AN/UYK-7 floating-point format requires two adjacent computer words.

Examples of Type-b Fields

8. FIELD BOOL B \$

This is a Compiler-packed Boolean type field. The associated table must have a packing descriptor.

9. FIELD VALUE I 16 S P -145, 3(19), 4(73) \$

The integer field VALUE will be allocated or packed according to the packing descriptor specified in the table declaration. The first occurrence of this field in an item will be preset to -145, the next three occurrences will be preset to 19, and the following 4 occurrences will be preset to 73.

See Table 4-2 for further examples.

4.2.4 Item-Area (ITEM-AREA) Declaration

An item-area is a data set with a structure identical to that of an item of the associated (parent) table, with the same number of words and the same field configuration. There may be any number of item-areas associated with a parent table, but they are physically separated from the table. The item-area is a convenient working storage area, assigned by the Compiler, where a single item of a table may be temporarily stored for examination, manipulation, or accumulation of data.

Format

ITEM-AREA area-name(s) \$

Explanation

ITEM-AREA Specifies the ITEM-AREA declaration.

Area Name(s) The unique name or names, separated by commas, of areas of working storage. Each area has the same format as an item of the parent table; therefore, only the name of each item-area associated with the parent table need be specified.

Examples

1. ITEM-AREA BUFF1 \$

A single item-area named BUFF1 is defined.

2. ITEM-AREA BUFF2, WORK1, WORKST \$

There are three item-areas named BUFF2, WORK1, and WORKST.

4.2.5 Subtable (SUB-TABLE) Declaration

A subtable is a set of adjacent items, wholly contained within the parent table. Its item size and field configurations are identical with those defined for the parent table. Except that it lies within the confines of the parent table, a subtable is itself a table having its own optional major index and stipulated maximum number of items. Subtable definitions within a parent table are independent of one another and may overlap. Subtables may only be defined for single-dimensional tables (horizontal or vertical tables).

Format

SUB-TABLE name initial-item-number-of-parent-table
maximum-number-of-items-in-subtable major-index-name \$

TABLE 4-2. EXAMPLES OF TYPE-a FIELDS

M-5035

TYPE OF FIELD	FORMAT OF FIELD DECLARATION							
	FIELD Name	Data Type	Rounding	Location	Start Bit	Initial Values	Scaling	\$
Integer	FIELD INTA	I 5 U	X	2	15	P 3(10)	V(1,0)	\$
	FIELD INTB	I 64 S		INTBWL	INTBST	P 0		\$
Floating Point	FIELD FLTA	F	X	1	31		X	\$
Fixed Point	FIELD FIXA	A 6 U 2		0	5	P 9.548		
	FIELD FIXB	A 32 S 16	FIXBLOC	31	\$			
Boolean	FIELD BOOLA	B	X	1	0	P 1,0,1		\$
	FIELD BOOLB	B		0	BOOLBBIT			\$
Status	FIELD STA	S 'ON', 'OFF'	X	1	14	P 'ON'		\$
Hollerith	FIELD HOLA	H 40		3	31	P H(ABC)		
	FIELD HOLB	H 3	0	28	\$			
Implied	FIELD IMPLA		X	1	15			\$

NOTES

1. Tags such as INTBWL, INTBST, FIXBLOC, and BOOLBBIT may be used as indicated but must be assigned integer values by an EQUALS declaration.
2. If the type of field is implied, the attributes of the field are determined by the implied mode of the Compiler to be a signed 16-bit integer, unless superseded by a MODE declaration.
3. If a field starts in the middle of a word, it must be wholly contained within that word--it may not cross word boundaries.

II-4-32

Explanation

SUB-TABLE	Specifies a SUB-TABLE declaration.
Name	A unique identifier by which the subtable is referenced.
Initial Item Number of Parent Table	Establishes the base item of the subtable by specifying the item number of the table at which the subtable is to start. The number can be either an integer or a tag predefined by an EQUALS statement.
Maximum Number of Items in Subtable	Either an integer or a tag that specifies the size of the subtable.
Major Index Name	Optional. Specifies the name of the major index of the subtable. If used, it is handled in the same manner as the major index of a table (see paragraph 4.2.2).

Examples

1. SUB-TABLE BLIP A 20 \$ | | | | | | | | | | | | | | | | | | | | | |

In this example, the subtable named BLIP starts at the initial item number 4 of the parent table and is 20 items long.

2. SUB-TABLE SCAN OVAL ROUND SPOOK \$ | | | | | | | | | | | | | | | | | | | | | |

The characteristics of subtable SCAN are defined by means of tags whose values are supplied by the EQUALS declaration. OVAL provides the initial item number, and ROUND, the number of items. SPOOK is the name of the major index.

4.2.6 Like-Table (LIKE-TABLE) Declaration

The like-table serves the same purpose for an entire table or specified number of items of the table as the item-area does for an item of the table. Like-tables have configurations identical to the tables that they duplicate. The use of like-tables allows the elimination of duplicate field definitions when defining multiple tables with identical formats. LIKE-TABLE must be enclosed within the TABLE and END-TABLE brackets to which the same field definitions apply. A LIKE-TABLE is valid in single dimension tables only.

Format

LIKE-TABLE name number-of-items major-index-name \$

Explanation

LIKE-TABLE	Specifies that a LIKE-TABLE specification follows.
Name	A unique identifier by which this table is referenced.
Number of Items	Optional. An integer or tag specifying the maximum number of items in the table. If this parameter is omitted, the number of items of the like-table will be the same as the number of items of the parent table.
Major Index Name	Optional. Specifies the name of the major index of the like-table. This parameter, if used, is handled in the same manner as the major index of a table (see paragraph 4.2.2).

Example

```
TABLE MTRK V 12 690 $
.
.
LIKE-TABLE STRK 3 $
END-TABLE MTRK $
```

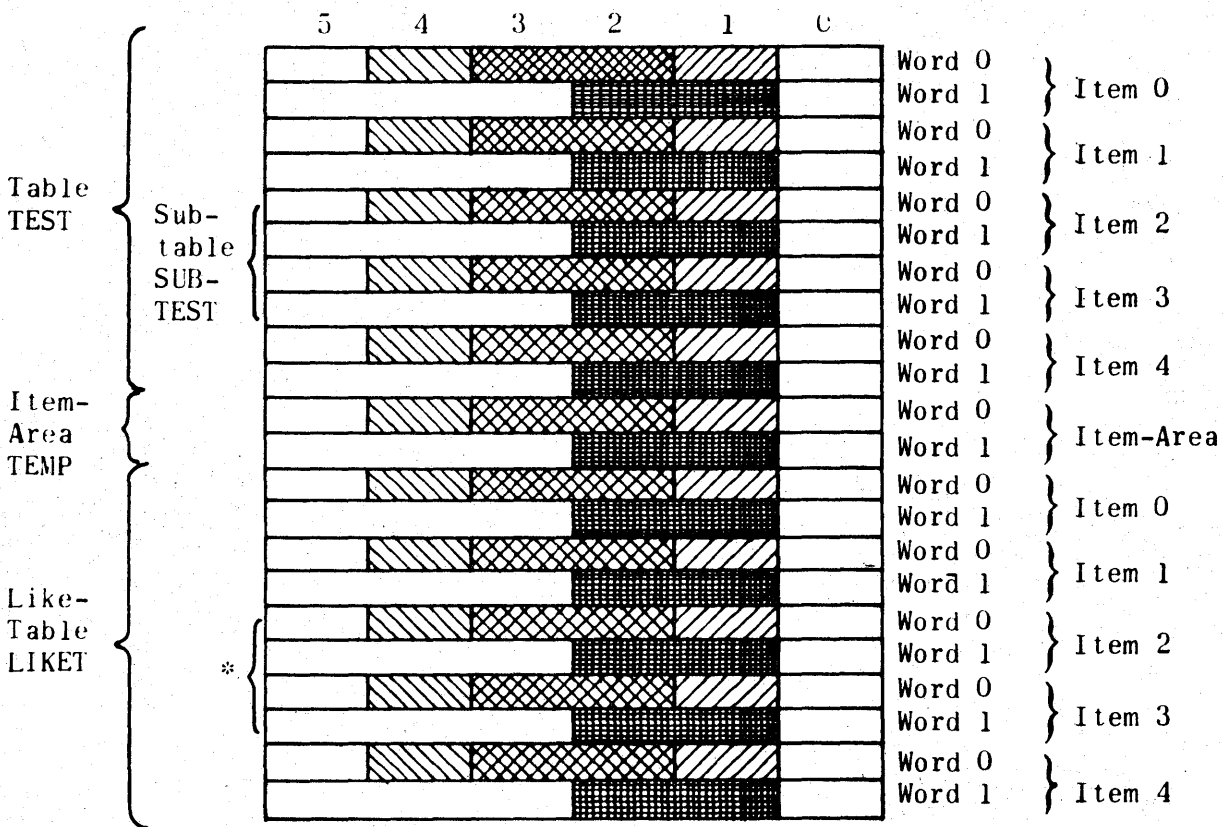
In this example, the like-table STRK is declared to have the identical format and field declarations as the parent table MTRK, except that it has only three items.

The following input:





```

TABLE | TEST | V | 2 | 5 | TEST | END | $ |
FIELD | VALUE1 | F | 3 | 4 | 0 | 3 | $ |
FIELD | VALUE2 | F | 3 | 4 | 0 | 4 | $ |
FIELD | VALUE3 | F | 2 | 1 | 1 | 2 | $ |
ITEM-AREA | TEMP | $ |
SUB-TABLE | SUBTEST | 2 | 2 | $ |
LIKE-TABLE | LIKET | 5 | $ |
END-TABLE | TEST | $ |
  
```

produces the following in core:



Note the following facts in conjunction with the resultant area:

1. Table TEST is a vertical table (all words of an item are together) of five items of two words each.
2. Item-area TEMP is allocated two words just like the items of table TEST.
3. Like-table LIKET is allocated five items identical to the items of table TEST.
4. Subtable SUBTEST exists within table TEST, occupying items 2 and 3.
5. Field VALUE1 () and VALUE2 () are identified with word 0 of every item within the table, like-table, and item-area. Notice the overlapping of the fields ().
6. Field VALUE3 () is identified with word 1 of every item within the table, like-table and item-area.
7. Subtable SUBTEST does not exist within the like-table LIKET at the corresponding position (*) because there is no way for the Compiler to differentiate between the two subtables.

4.2.8 Packing Rules

As fields and variables are defined in data declaration statements, the programmer specifies various attributes of the data units. These attributes are used by the Compiler to determine the proper allocation of these data units consistent with the AN/UYK-7 memory structure and addressing scheme. When Compiler packing of fields is specified in a table declaration, or when allocation of variables is performed by the Compiler, a set of rules governs the manner in which the fields and variables are packed into AN/UYK-7 memory words.

Three different packing algorithms are used by the Compiler: no packing, medium packing, and dense packing. In the descriptions of these algorithms that follow, the term magnitude is used to refer to the number of bits required to represent an arithmetic data unit (I or A type) excluding the sign bit. For example, a signed 32-bit integer (I 32 S) has 31 magnitude bits; an unsigned 32-bit integer (I 32 U) has 32 magnitude bits.

NOTE

In the case of no packing or medium packing, when a programmer specifies the number of bits in a field or variable, the Compiler will guarantee a data unit allocation of at least that number of bits; additional magnitude bits may be provided. Hence, the length specification of an arithmetic data unit should always be regarded as the minimum number of bits required to contain that data unit. For all types of packing, multiword data units must always start in bit 31 of a word.

4.2.8.1 No Packing (NONE)

No packing applies only to fields in tables declared with the NONE packing descriptor. Such fields will be packed as follows:

<u>Type</u>	<u>Packing</u>
Boolean (B) and Status (S)	Allocated a full-word.
Hollerith (H)	Allocated the least number of full-words required to contain the specified number of characters (four characters per word).
Floating-point (F)	Allocated two full-words.
Fixed-point (I or A)	Allocated one full-word if the magnitude is less than 32 bits; allocated two full-words if the magnitude is less than 64 bits.

4.2.8.2 Medium Packing (MEDIUM)

Medium packing applies to all variables and all fields in tables declared with the MEDIUM packing descriptor. Medium packing is performed as follows.

<u>Type</u>	<u>Packing</u>
Boolean (B) and Status (S)	Allocated a quarter-word.
Hollerith (H)	Allocated the least number of full-words required to contain the specified number of characters (four characters per word) if greater than two characters; allocated a quarter-word if one character; allocated a half-word if two characters.
Floating-point (F)	Allocated two full-words.
Fixed-point (I or A)	Allocated a quarter-word if unsigned and less than nine bits in magnitude; allocated a half-word if less than 16 bits in magnitude; allocated a full word if less than 32 bits in magnitude; allocated two full words if less than 64 bits in magnitude. Illustrating these rules: an I 8 U receives a quarter-word; an I 7 S, I 8 S, I 15 U, or I 16 S receives a half-word; an I 16 U or I 32 S receives a full word; an I 32 U or I 64 S receives two words; and an I 64 U is illegal.

4.2.8.3 Dense Packing (DENSE)

Dense packing applies only to fields in tables which are programmer-packed, to tables declared with the DENSE packing descriptor, or to variables that are overlaid (see paragraph 4.2.9). Dense packing is performed as follows:

<u>Type</u>	<u>Packing</u>
Boolean (B)	Allocated a single bit.
Status (S)	Allocated the number of bits required to hold the largest status constant (e.g., if seven states are defined, three bits would be allocated).

<u>Type</u>	<u>Packing</u>
Hollerith (H)	Allocated the number of quarter-words required to contain the specified number of characters if less than five; otherwise, allocated the required integral number of words.
Floating-point (F)	Allocated two full-words.
Fixed-point (I or A)	Allocated the number of bits required to contain the data unit if less than 32 bits in magnitude; otherwise allocated two full-words.

4.2.9 Overlay (OVERLAY) Declaration

The OVERLAY statement allows the user to control the allocation of variables and fields which are to be densely packed by the Compiler. If the OVERLAY involves fields, the OVERLAY statement must follow the associated field declarations within the TABLE, END-TABLE brackets. If the OVERLAY involves variables, it implies dense packing, and the OVERLAY statement must follow the associated data declarations within the data design.

Any number of fields or variables can be named, separated by commas, on the right side of an OVERLAY. All are allocated consecutive storage space in the order in which they are named, subject to the following restriction: a field or a variable defined as part of a word will be allocated so that it is wholly contained within a word. Compatibility in size is the programmer's responsibility.

Any number of OVERLAY statements are permitted and names may appear in more than one OVERLAY statement but logical inconsistencies must be avoided. Programmer confusion in using the OVERLAY statement can be avoided by interpreting the word OVERLAY in this statement as meaning "overlay with."

When using OVERLAY, the following rules apply:

1. If the data unit name is not a field, the variable list must not include fields.
2. If the data unit name is a field, all of the names in the field list must be fields; only fields within the same table may be overlaid.

3. If the overlay list exceeds the size of the data unit being overlaid, a warning message is output by the Compiler; however, the data allocation will still be performed.
4. All nonfield OVERLAYS must be outside table definition brackets but within the data design. Field OVERLAY's must be within the table brackets.
5. All names within an overlay must be previously defined within the data design.
6. A name cannot be used more than once on the right side of an OVERLAY statement.
7. If a name is to appear on the right side of one OVERLAY and to the left side of another OVERLAY, the statement containing the name on the right side must be declared first.

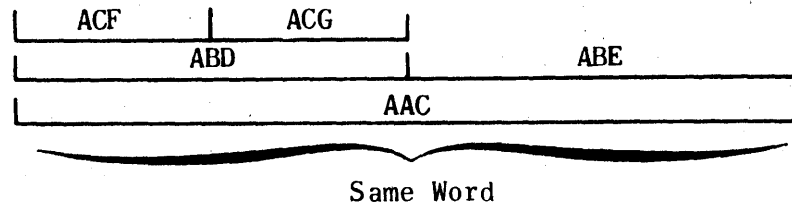
Format

data-unit-name OVERLAY variable-list or field-list \$

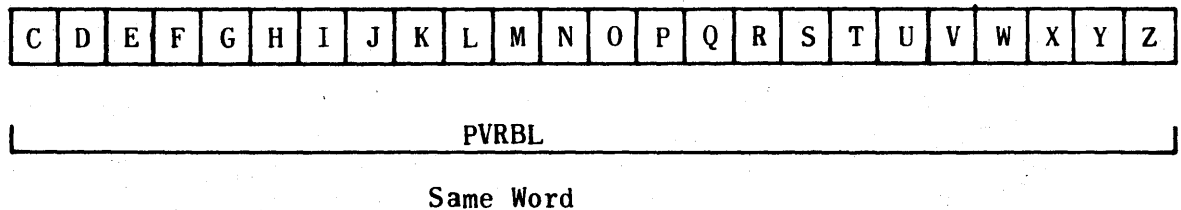
Explanation

Data Unit Name	The name of a variable, table, field, like-table, item-area or subtable.
OVERLAY	Indicates that the variable or field list to the right of the operator is to overlay the data unit to the left of the operator.
Variable List	Contains the names of one or more variables, tables, like-tables or item-areas, separated by commas.
Field List	Contains the names of one or more fields separated by commas.

The results of statements 2 through 8 would be:

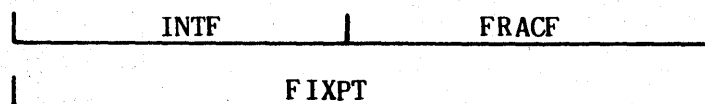


The results of statements 9 through 11 would be:



In the above example, a series of flags (Boolean variables) have been declared so that they occupy a single variable (word). This might be advantageous if it is frequently necessary to check to see if all flags are cleared. Also, since all flags are contained in a single variable, they can be quickly passed on procedure or function calls. Such packing results in more object code when data is manipulated, and care must be exercised in weighing the advantages gained over the resulting disadvantages.

The results of statements 13 through 16 would be:



4.2.10 Data Referencing

Data units are referenced in their entirety by name (identifier), a specific occurrence of an n-dimensional unit by name and subscripts, or a particular part of a data unit by use of a functional modifier. Variables and lists (tables, subtables, like-tables and item-areas), when treated as entities, are referenced by name only. Each of these data units has its own unique name, as established in a declaration. Fields, items, and words are always referenced with their associated table, item-area, like-table, or subtable. Because of this, a field name by itself is never meaningful. To identify the subdivision of a larger data unit, the additional descriptive information is enclosed in parentheses after the name of the larger data unit.

All lists are indexed (or subscripted) from 0 through N-1, where N equals the number of entries. Thus, entry 0 is the first entry in the list, entry 1 is the second entry, etc.

4.2.10.1 Table Referencing

Tables may be accessed in a variety of ways:

- a. Whole table referencing:
 1. Set every word of the given table to the specified value.
 2. Set every word of one table to the value of the corresponding word in another table.
- b. Item referencing - Set an item of the given table to the specified value(s).
- c. Field referencing (where a word is a special case of a field corresponding to one word) - Set the given field to the specified value.

The method of addressing may also be determined by the category into which the table falls:

- a. Horizontal or vertical (one-dimensional).
- b. Array (multidimensional).

4.2.10.1.1 Whole Table ReferencingFormat

name

Explanation

Name The name of the table, subtable, or like-table.

Examples

(See paragraph 5.4 for rules applicable to multiword set statements.)

1. SET TABL1 TO 5 \$

For this example, every word of TABL1 will be set to 5.

2. SET TABL1 TO TABL2 \$

In this example, every word of TABL1 will be set to the value of the corresponding word of TABL2. If TABL2 is longer than TABL1, the transfer of values will stop at the end of TABL1. If TABL2 is shorter than TABL1, the transfer of values will stop at the end of TABL2, with the excess words of TABL1 unaffected.

4.2.10.1.2 Item Referencing. There are two ways to address items, depending on the type of table in which the items occur:

1. Horizontal or vertical (one-dimensional):

Format

name(i)

Explanation

Name The name of the table, subtable, or like-table.

i The item indicator. It may be a data unit, tag, constant, or arithmetic expression.

Examples

a. SET TABL1(0) TO 5 \$

All words of the first item (0) of TABL1 will be set to 5.

b. SET TABL1(VAL) TO 5 \$

The words of the item of TABL1 indicated by the value of VAL will be set to 5.

2. Array (multidimensional):

Format

name (d_1, d_2, \dots, d_7)

Explanation

Name The name of an array.

d_i The index corresponding to the associated dimension of the table. The number of indexes specified must correspond to the number of dimensions of the table (at least 1 but no more than 7). Each index may be a constant, tag, data unit, or arithmetic expression.

Examples

a. TABL1

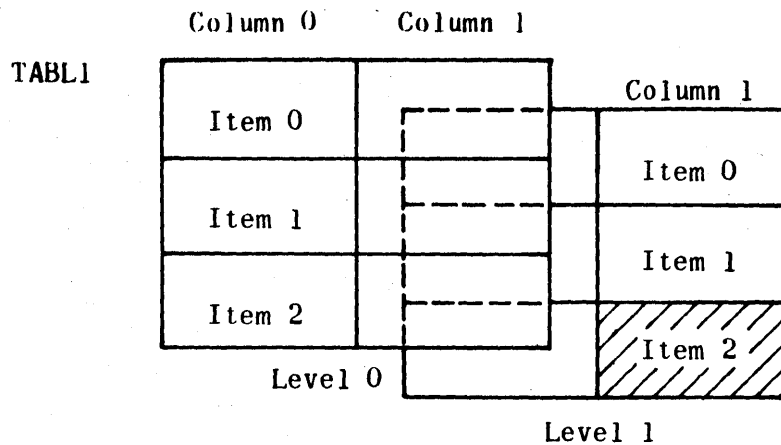
Column 0 Column 1

Item 0	Item 0
Item 1	Item 1
Item 2	Item 2

SET TABL1(1,1) TO 5 \$

All the words of item 1 in column 1 (the shaded area) of TABL1 will be set to 5.

b.



SET TABL1(2,1,1) TO 5

All the words of item 2 in column 1 of level 1 (the shaded area) of TABL1 will be set to 5.

4.2.10.1.3 Field Referencing. There are two ways in which fields may be addressed, depending on the type of table in which the fields occur:

1. Horizontal or vertical (one-dimensional):

Format

name(i,f)

Explanation

Name	The name of the table, subtable, or like-table.
i	The item indicator. It may be a data unit, constant, tag or arithmetic expression.
f	The field indicator. To specify a field previously defined within the table, f must be the field name. A word is a special type of field which may be indicated by a constant, data unit, tag, or arithmetic expression.

Examples

a. SET, TABL1(2,FLD) TO 5 \$

The predefined field FLD of item 2 of TABL1 will be set to 5.

b. SET, TABL1(2,1) TO 5 \$

Word 1 of item 2 of TABL1 will be set to 5.

2. Array (multidimensional):

Format

name(d₁, ..., d_n,f)

Explanation

- Name The name of the table.
- d_i The index corresponding to the associated dimension of the table, as previously described.
- f The field indicator. To specify a field previously defined within the table, f must be the field name. A word is a special case of field that may be indicated by a constant, data unit, tag or arithmetic expression.

Examples

a. SET, TABL1(2,1,1,FLAG) TO BUFLAG \$

The contents of BUFLAG will be placed in the field FLAG of item 2 in column 1 of level 1 of TABL1.

b. SET, TABL2(1,0,1) TO 5 \$

Word 1 of item 1 of column 0 will be set to 5.

c. SET TAB(2*TAB1(I,FLD1),FLD) TO 0 \$

The contents of field FLD of the item represented by the expression 2*TAB1(I,FLD1) will be set to 0.

d. TABL1 Whole table.
TABL1(4) Item 4.
TABL1(4,FLD) Field FLD of Item 4.
TABL1(IND1,2) Word 2 of item specified by IND1.
ARRAY2(0,1,FLD) Field FLD of item 0 in col. 1.
ARRAY3(IND1,IND2,IND3,TAG1) Word specified by TAG1 of of item specified by IND1 of column specified by IND2 of level specified by IND3.

NOTE

The interpretation of the referencing is governed by the structure declared in the TABLE declarative.

4.2.10.1.4 Item-Area Referencing. An item-area is addressed by its name. The name alone addresses the total item. A field specification may be included to address a field or word.

Format

name

or

name (f)

Explanation

Name The name of an item-area.

f The field indicator. To specify a field previously defined within the parent table, f must be the field name. A word is a special case of field that may be indicated by a constant, data unit, tag, or arithmetic expression.

Examples

1. SET ITEMA TO 5 \$

Every word of ITEMA will be set to 5.

2. SET ITEMA (FLD) TO 5 \$

Field FLD of ITEMA will be set to 5.

3. SET ITEMA (2) TO 0 \$

Word 2 of ITEMA will be set to 0.

4.2.11 Transfer Declaratives (Switches)

The transfer declaratives allow the establishment of switches for determining indirect linkage within procedures and for transferring control from one procedure to another. There are two classes of transfer declaratives: statement switches and procedure switches.

4.2.11.1 Statement Switch (SWITCH) Declaratives

The statement switch is a collection of statement labels to which control may be transferred, depending on various conditions encountered during processing. For purposes of identification and selection, the switch is a unit identified by a switch name. Since statement switches are collections of statement labels, and since statement labels are always local to the system procedure in which they are defined, switch declarations must fall within local data design (LOC-DD) brackets. There are two types of statement switches: index and item.

4.2.11.1.1 Index Switch. The index switch defines a transfer of control that is determined by a user-supplied index.

program control to the statement BOLT or the statement SWCH, depending upon a numerical input of 0 or 1, respectively.

The number of switch points in the left-hand column may be greater than the number in the right-hand column. In this case, only one statement name is specified.

2. SWITCH SW1, SW2 \$
 ALPHA, UPDATE \$
 BETA, INCR \$
 GAMMA, DECR \$
 DELTA \$
 END-SWITCH SW1, SW2 \$

Switch SW1 may be referenced by index values of 0, 1, 2 and 3.

Switch SW2 may use index values of 0, 1 or 2.

4.2.11.1.2 Item Switch. The item switch defines switch points that are accessed by a constant as specified in the definition. The Compiler performs a compare between the value of the variable name contained in the switch statement and the constants of the definition. Control will then be transferred to the switch point that corresponds to the matching constant. Validity checking of the data unit name is not necessary as program control will continue with the next instruction if a match is not found. This means, however, that the instruction sequence following must be applicable to a not-found condition. The variable specified in the item switch declaration must be defined prior to its reference in the switch declaration and must not exceed two words in length.

Format

```

SWITCH name (variable-name) $
constant, switch-point $
.
.
.
constant, switch-point $
END-SWITCH name $

```

Explanation

SWITCH	Specifies the beginning of a switch definition.
Name	An identifier used to reference the switch.
Variable Name	The name of a variable whose value is to be compared against the list of constants in the left-hand column.
Constant	A CMS-2 constant of two words or less in length. This constant must agree in type with the variable.
Switch Point	A statement name.
END-SWITCH	Specifies the termination of the switch.

Example

```

SWITCH SWOFF (FINISH) $
H(FINISH) ELEMENT $
H(STOP) UNCOND $
H(TERM) DONE $
END-SWITCH SWOFF $

```

This declaration defines switch SWOFF with switch points ELEMENT, UNCOND, and DONE. A reference to switch SWOFF will transfer control to one of these switch points, depending upon the value of the variable named FINISH. If FINISH is equal to H(TERM), control will transfer to the statement DONE.

4.2.11.2 Procedure Switch (P-SWITCH) Declaratives

The P-SWITCH is a collection of procedure names to which a call may be made, depending on conditions encountered during execution. The list of procedure names identifies the procedures accessible by the switch. Procedure switches are declarations, and can fall within the data design brackets (SYS-DD or LOC-DD) or may stand alone (i.e., within a SYS-PROC but outside a LOC-DD or procedure). There are three types of P-SWITCHes: index procedure, double procedure, and item procedure.

4.2.11.2.1 Index Procedure Switch

Format

```
P-SWITCH name INPUT formal-parameters OUTPUT
formal-parameters $
switch-point $
.
.
.
switch-point $
END-SWITCH name $
```

Explanation

P-SWITCH	Specifies the P-SWITCH declaration.
Name	The identifier by which the p-switch is referenced.
INPUT	Optional. Specifies that formal input parameters for each p-switch procedure follow.
OUTPUT	Optional. Specifies that formal output parameters for each p-switch procedure follow.
Formal Parameters	A list of names (single identifiers), separated by commas, which are to be input or output to the procedures.

Switch Point The name of a procedure accessible by the switch. The switch points are indexed by a value within the range 0 through n-1, where 0 corresponds to the first switch and n equals number of switch points. The program must include validity checking of the index if it can exceed the range 0 through n-1 (see paragraph 5.5).

END-SWITCH Specifies the termination of the switch.

This declaration allows the use of input and output parameter transfers. If formal parameters are specified, they must be identical for every procedure of the p-switch. No abnormal exits are allowed. Transfer to the procedures specified is activated by a procedure-switch linking statement (see paragraph 5.3).

Example

P-SWITCH	TRIG	INPUT	ANG,	SIDE	OUTPUT	SOL	\$
	SIN	\$					
	COS	\$					
	TAN	\$					
	END-SWITCH	TRIG	\$				

This declaration defines procedure switch TRIG whose input parameters are ANG and SIDE and whose output parameter is SOL. A reference to switch TRIG transfers control to one of the procedures SIN, COS or TAN, depending upon a numeral index of 0, 1 or 2, respectively.

4.2.11.2.2 Double Procedure Switch. Two procedure switches may be defined in a single declaration.

Format

```

P-SWITCH name-a, name-b $
      switch-point,  switch-point $
      .
      .
      .
      switch-point,  switch-point $
END-SWITCH name-a, name-b $

```

Explanation

P-SWITCH Specifies a P-SWITCH declaration.

Name The name of a procedure. A switch point is indexed by a value within the range of 0 through n-1 where n is the number of switch points.

END-SWITCH Specifies the termination of the switch.

Multiple procedure switch declarations do not allow formal input or output parameters. Name-b and associated switch points define a second procedure switch.

Example

```

P-SWITCH PLANE, TRAIN $
      AIR, RAIL $
      PROP, STATION $
      FOG $
END-SWITCH PLANE, TRAIN $

```

This declaration defines independent procedure switches PLANE and TRAIN. A reference to switch PLANE will transfer program control to one of the procedures AIR, PROP or FOG depending upon a numerical input of 0, 1 or 2, respectively.

4.2.11.2.3 Item Procedure SwitchFormat

```

P-SWITCH name (variable-name) INPUT formal-parameters
OUTPUT formal-parameters $
constant, procedure-name $
.
.
.
constant, procedure-name $
END-SWITCH name $

```

Explanation

P-SWITCH	Specifies a P-SWITCH declaration.
Name	An identifier used to reference the P-SWITCH.
Variable Name	The name of a variable whose value is to be compared against the list of constants in the left-hand column. When a match is found, the procedure that is paired with the constant will be accessed.
INPUT	Optional. Specifies that the names which follow are the formal input parameters.
OUTPUT	Optional. Specifies that the names which follow are the formal output parameters.
Formal Parameters	A list of the formal names, separated by commas, of the input and output parameters.
Constant	Any allowable CMS-2 constant of two words or less. This constant must agree in type with the variable.
Procedure Name	Identifies the procedures accessible by the switch.
END-SWITCH	Specifies the termination of the list of procedure names.

Example

P-SWITCH	LINK	(MTYPE)	INPUT	ADDR	OUTPUT	RESULT	\$
	0	(1,2)		MTPA			\$
	0	(2,2)		MTPB			\$
	0	(3,2)		MTPC			\$
			END-SWITCH	LINK			\$

A procedure switch linking statement would invoke the procedure item switch. This would cause the contents of the variable MTYPE to be compared against the list of constants in the LINK procedure switch table. A match generates a procedure call to the procedure associated with the constant. The formal parameters are ADDR and RESULT.

4.2.11.3 Switch Referencing

Statement switches (index and item) are referenced by a GOTO statement. Paragraph 5.5 provides examples of such referencing. Procedure switches (P-SWITCH) are referenced by a procedure switch call. Paragraph 5.3 provides examples of such referencing.

4.2.12 Local Indexes

Identifiers may be used to refer to machine index registers within the range of a procedure by means of index declaration statements. Two types of indexes may be declared: system indexes and local indexes.

System indexes are global identifiers that must be declared in a major header (see Section 7).

Local indexes are declared for use within a procedure by means of the LOC-INDEX statement.

Format

LOC-INDEX name(s) \$

Explanation

LOC-INDEX Indicates that the following name or names are to refer to the Compiler-assigned index register(s).

Name An identifier(s) of the index register(s). It must not be previously defined in a data declaration. This name can be an actual (not a formal) parameter in a procedure call. Multiple names are separated by commas.

The declaration should immediately follow the PROCEDURE statement.

Examples

```
LOC-INDEX ALPHA $
LOC-INDEX K, BAKER, CLAN $
```

The following conventions apply to declaration and use of indexes:

1. Two index registers (B6 and B7) are reserved for Compiler use and will never be assigned to a user-declared index.
2. Five index registers (B1 through B5) may be assigned specific data names by SYS-INDEX statements.
3. Up to five index registers may be assigned by the Compiler for use in a procedure by LOC-INDEX statements.
4. The sum of the number of index registers assigned by LOC-INDEX and SYS-INDEX statements will never exceed five.
5. There is no restriction on the number of local indexes which may be defined by LOC-INDEX statements. However, if index registers are not available, temporary locations in memory will be assigned.
6. In non-arithmetic operations the Compiler manipulates index registers as 16-bit, unsigned, integer data units. When index registers are used as operands in arithmetic expressions, AN/UYK-7 sign bit considerations require that the result of the expression must not exceed 15 bits in magnitude.

4.2.13 Data (DATA) Declaration

A DATA declaration may be used to assign a preset value to a previously defined data unit. EQUALS tags may be used to represent numeric constants (see Section 7).

NOTE

The DATA declaration is accepted by the AN/UYK-7 CMS-2 Compiler to provide additional compatibility with other CMS-2 compilers. However, its implementation is not fully compatible with other CMS-2 language implementations; nor can its continued existence in this or future CMS-2 implementations be assured. It is strongly recommended that the variable and field preset capability and the extensive direct code preset features available with this compiler be used in place of the DATA declaration. Each DATA declaration generates full word preset values; no partial word variables should be preset via the DATA declaration.

Format

name DATA constant-a constant-b \$

Explanation

- | | |
|------------|---|
| Name | An identifier of a table, subtable, variable, liketable, or item-area. After the first declaration, name is optional when presetting sequential words of a multiword data unit. When presetting these units, a user must be aware of the word allocation format (see Example 2 on the following page). |
| Constant-a | A numeric integer constant, Hollerith constant, or tag assigned as an initial value for the named data unit. If name identifies a variable, the type of this constant must agree with the type of the variable. |
| Constant-b | Optional. A numeric integer constant or tag that is used when a dual preset value may be applied to a data unit. When constant-b is specified, constant-a assigns a value to the upper halfword of the data unit and constant-b assigns a value to the lower halfword of the named data unit. Constant-b is not allowed when constant-a is Hollerith. |

A numeric constant may be followed by a scaling specifier (a comma followed by a positive integer constant). The scaling specifier must be given if the preset value is to have any fractional precision.

Examples

1. TAC DATA 77 \$

The whole word, referenced by the data unit name TAC, has an initial preset value of 77.

2. One DATA declaration per data unit name presets only the first word of the unit. To preset several or all words, the following format is employed, using table DICT as an example:

DICT DATA -64 \$
DATA 7 0 \$
DATA 11 \$

The first word of table DICT has an initial value of -64. The second word has an initial value of 7 in the upper half and 0 in the lower half. The third word has an initial preset value of 11. When using this method of table presetting, the user must be concerned with word allocation format so that a data reference will give the proper preset value.

3. HOLVB DATA H(TWOWDS) \$

This declaration will preset the Hollerith variable HOLVB with the characters TWOWDS (left-adjusted with two trailing blanks).

4.3 CONTROL DECLARATIVES

A variety of declaratives are available for use in specifying various Compiler control information. Most of these declarations control allocation or code generation on an element-wide basis. They appear primarily in major or minor headers, and are therefore described in Section 7. The MODE declaration is closely related to the data declaration process and is therefore described in the following sections.


```

2. MODE FIELD A 32 S 16 $
TABLE LOCALTRK H 11 100 LOCTR $
FIELD TRKNO I 32 S 0 31 $
FIELD ED S 'FRIEND', 'FOE', 'UNKNOWN' 1 31 $
FIELD X 1 2 31 $
FIELD Y 1 3 31 $
FIELD T 1 4 31 $
.
.
END-TABLE LOCALTRK $

```

In this example, fields X1, Y1 and T1 have undeclared data-type attributes; therefore, the MODE-declared attributes of A 32 S 16 would be assigned to each. Note that it is still necessary to specify the word location and starting bit for those three fields since Compiler packing was not specified.

4.4 SYSTEM LINKAGE

System linking is the process by which program information known to one basic CMS-2 element may be communicated to another basic element. It may be required when the information is local to a system procedure within a SYSTEM or unknown because the elements were compiled under different SYSTEM headers. The following items may be linked between two basic elements:

- a. Tables (and associated items).
- b. Variables.
- c. P-switches.
- d. Procedures.
- e. Functions.
- f. Files.

The capability to link the above items between and within basic CMS-2 elements is provided by the EXTREF, EXTDEF, TRANSREF, or LOCREF operators. This capability eliminates the need to expand the local concept of certain items and/or include entire elements within SYSTEM compiles when only a few unknown items are referenced.

Because segments containing items a through f can be linked at load time, it is unnecessary to externally reference a procedure (unless it has input/output parameters or exits) or to define a procedure in a segment in order to call it in that segment. Although the compiler will not flag such an "undefined" procedure, the loader will do so if that procedure remains undefined at load time.

4.4.1 External Definition (EXTDEF) Operator

The name and associated declaration following the operator EXTDEF is to be considered as global so that it may be referenced within any basic element of any SYSTEM compilation.

Format

(EXTDEF) identifier-identification \$

Explanation

EXTDEF. Specifies an external definition.

Identifier-Identification A symbol and its associated definition defined totally within this SYS-PROC.

Examples

1. (EXTDEF) VRBL X, F \$

This example specifies that VRBL X, defined in a LOC-DD of the SYS-PROC, is referenced by other SYS-PROC's and is to be considered global.

2. (EXTDEF) VRBL (XX, YY, ZZ) F \$

This example declares that variables XX, YY, and ZZ within this SYS-PROC are global floating-point data units.

Explanation

TRANSREF Specifies a transient reference.
Identifier-Identification A symbol and its associated identification that is referenced from, but not located within, this SYS-PROC.

Examples

```

1 (TRANSREF) VRBL F1 F $
1 (TRANSREF) VRBL (F2, F3) F $

```

4.4.4 Local Definition (LOCREF) Operator

The procedure or function declaration following the operator LOCREF is considered local to the system procedure in which it is contained, so that forward references to the procedure or function receive proper parameter linkage. The declaration containing the (LOCREF) modifier must appear in a local data design prior to the procedure or function containing the local forward reference. Formal parameters must be defined prior to the declaration.

Format

(LOCREF) procedure or function declaration \$

Explanation

LOCREF Specifies a local procedure or function declaration.

Example

```

1. VRBL VB1 I 8 U $
1 (LOCREF) PROCEDURE PRA INPUT VB1 $

```

This example specifies that procedure PRA is a procedure local to the system procedure and is called locally within this system procedure. Its formal input is VB1.

4.4.5 Applications of EXTDEF, EXTREF, and TRANSREF

```

1. A      SYS-PROC $
      LOC-DD $
      (EXTDEF) TABLE SUM H 1 10 $
      FIELD SUM1 A 32 S 4 10 31 $
      (EXTDEF) ITEM-AREA SUMA $
      ITEM-AREA SUMB $
      END-TABLE SUM $

      .
      .

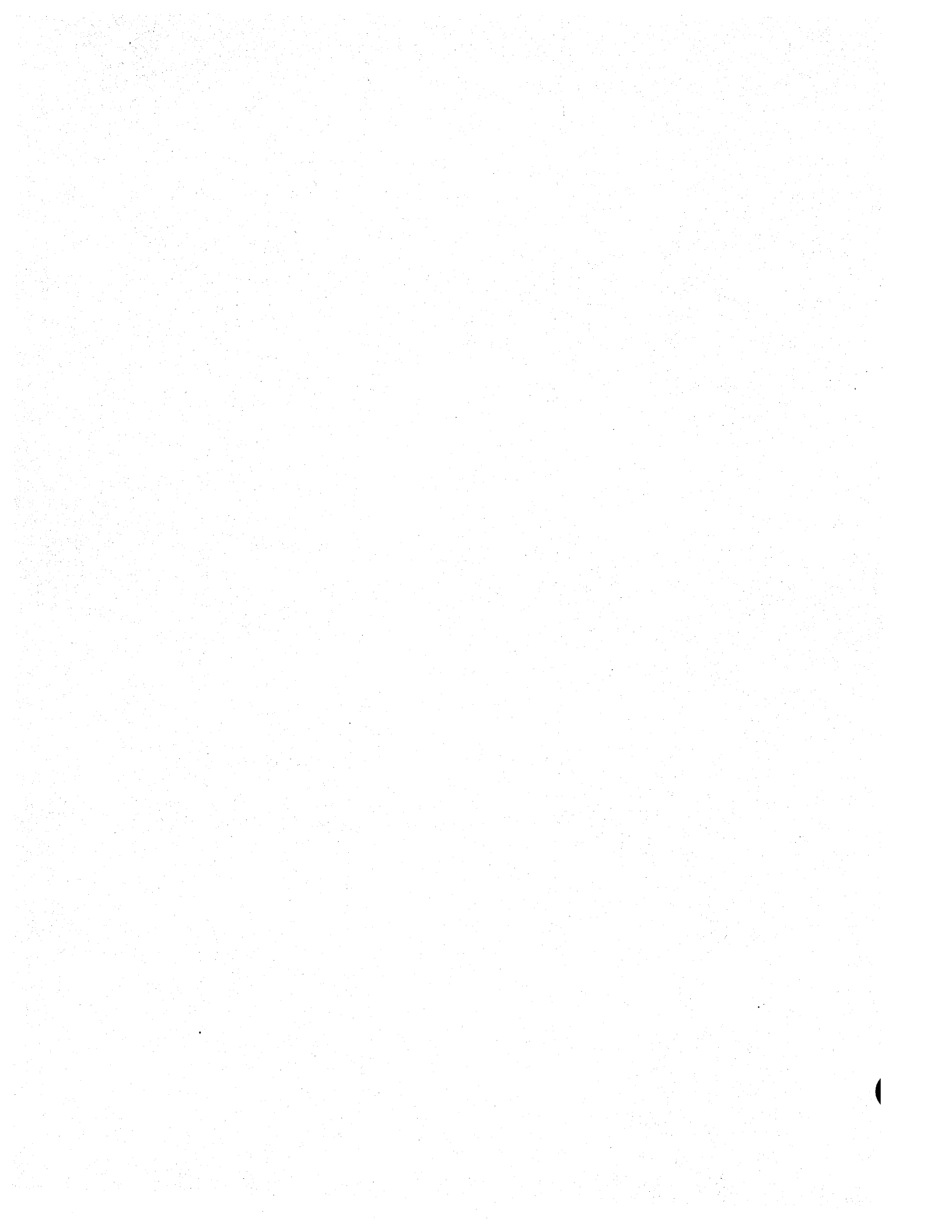
      B      SYS-PROC $
      LOC-DD $
      (EXTREF) TABLE SUM H 1 10 $
      FIELD SUM1 A 32 S 4 10 31 $
      LIKE-TABLE SUML 5 $
      (EXTREF) ITEM-AREA SUMA $
      ITEM-AREA SUMB $
      END-TABLE SUM $
  
```

Table SUM is linked between system procedure A and system procedure B. Since the field definition for SUM1 is included in the table description, SUM1 is also linked between A and B. A reference to SUM (0, SUM1) in either system procedure will access the same data unit.

Subunits of tables, such as like-tables, subtables, and item-areas are not automatically linked between system procedures. In this example, a reference to SUMB will access different data units. However, linkage can be achieved for subunits of tables. Item-area SUMA is externally defined in system procedure A and externally referenced in system procedure B, thus providing the ability to reference the same data unit from either system procedure.

An externally referenced table must have the same field definitions as that of the externally defined table. Only like-tables, subtables and item-areas may be added or

deleted from the definition. As an example of this, SUML can be included in system procedure B without being in system procedure A as long as it has the same attributes as table SUM.



```

BETA      SYS-PROC  B. STRANGLER 25 MAR 71 $
          LOC-DD $
(EXTREF)  VRBL P1 F $
(EXTREF)  VRBL P2 F $
          VRBL A1 F $
          VRBL A2 F $
(EXTREF)  PROCEDURE ALPHA INPUT P1 OUTPUT P2 $
          END-LOC-DD $
          .
          .
          ALPHA INPUT A1 OUTPUT A2 $
          .
          .
          END-SYS-PROC BETA $
HAND     SYS-PROC  J. DOE 25 MAR 70 $
          LOC-DD $
(EXTDEF)  VRBL P1 F $
(EXTDEF)  VRBL P2 F $
          END-LOC-DD $
(EXTDEF)  PROCEDURE ALPHA INPUT P1 OUTPUT P2 $
          .
          .
          END-PROC ALPHA $
          .
          .

```

Procedure ALPHA is in SYS-PROC HAND. SYS-PROC BETA initiates a procedure call to ALPHA by means of ALPHA INPUT A1 OUTPUT A2\$. To indicate this cross-referencing to the Compiler, procedure ALPHA and its associated formal parameters, variables P1 and P2, are flagged as external references in SYS-PROC BETA and as external definitions in SYS-PROC HAND.

same base register. This capability allows assigning a base register to one SYS-DD when SYSTEM A is loaded for execution and a transient register to the other SYS-DD. It was not necessary to attach the operator EXTDEF to the variables in SYSTEM B since they are within SYS-DD's and, hence, global.

Use of the TRANSREF and EXTREF operators allows the Compiler to generate the appropriate object code for SYSTEM A. The allocation is determined by the system Loader.

```
4. SPCX      SYS-PROC $
           LOC-DD $
           VRBL VX1 A 16 S 10 $
(LOCREF) PROCEDURE PRX1 OUTPUT VX1 $
           VRBL VX2 H 2 $
(LOCREF) FUNCTION FNX2 (VX2) H 4 $
           VRBL (VA3,VA4) A 32 S 24 $
           VRBL (HA5,HA6) H 4 $
           •
           •
           END-LOC-DD $
           PROCEDURE PRCA1 $
           •
           •
           PRX1 OUTPUT VA3 $
           •
           •
```

```
SET HA5 TO FNX2(HA6) $  
  ●  
  ●  
END-PROC PRCA1 $  
PROCEDURE PRX1 OUTPUT VX1 $  
  ●  
  ●  
END-PROC PRX1 $  
FUNCTION FNX2 (VX2) H 4 $  
  ●  
  ●  
END-FUNCTION FNX2 $  
  ●  
  ●  
END-SYS-PROC SPCX $
```

Procedure PRX1 and function FNX2 are local to system procedure SPCX; both are called prior to their definition by procedure PRCA1. Including the declarations in the local data design prior to PRCA1 permits more accurate error checking and generation for their parameter passage with respect to scaling and type.

SECTION 5

DYNAMIC STATEMENTS

Dynamic statements specify processing operations within procedures and functions. They perform calculations, manipulate data, and direct control of the program.

This type of statement consists of an operator followed by a list of operands and additional operators. An operand may be a single name, a constant, a data unit reference or an expression. Expressions may be arithmetic, Boolean, relational, or literal.

Dynamic statements have two possible forms: simple and compound. A simple statement comprises a single dynamic statement followed by its statement terminator (\$). A compound statement consists of two or more dynamic statements, each separated by the connector THEN, followed by a single \$ statement terminator.

With the exception of decision statements (Paragraph 5.6), there are no restrictions on the number or types of dynamic statements which may be compounded. With this exception, the connector THEN is exactly equivalent to the terminator \$.

Example

The compound statement

```
SET A TO B THEN PROCA INPUT X  
THEN L2. SET E TO F THEN GOTO L1 $
```

is equivalent to the simple statements:

```
SET A TO B $ PROCA INPUT X  
$L2. SET E TO F $ GOTO L1 $
```

Simple and compound statements may also be grouped into a statement block by the formation of a BEGIN, VARY, or FOR block (Paragraph 5.7). Statement blocks are required in order to nest decision statements (Paragraph 5.6.6).

5.1 EXPRESSIONS

The arithmetic, Boolean, relational, and literal expressions used in dynamic statements are described in the following paragraphs.

NOTE

Variables, constants, local and system indexes, field references, typed item references, item word references, function references, and functionally modified data units may be used as operands in CMS-2 expressions. Tables, sub-tables, like-tables, and untyped item references may not be used as expression operands.

5.1.1 Arithmetic Expressions

An arithmetic expression consists of two or more arithmetic data units or constants (operands) connected by arithmetic operators. The operators and their hierarchy of execution are defined in Table 5-1. Operations involving the level-1 operators are evaluated first, followed by evaluation of operators of levels 2 and 3.

TABLE 5-1. ARITHMETIC OPERATORS

HIERARCHY OF EXECUTION	OPERATOR	FUNCTION
1	-	Unary minus
1	**	Exponentiation
2	*	Multiplication
2	/	Division
3	+	Addition
3	-	Subtraction

If expressions involve more than one operator of the same hierarchy, execution is performed from left to right in the order in which the operators are encountered. For example, $A*B/C$ is evaluated as $(A*B)/C$. The one exception to this rule occurs with expressions involving the exponentiation and unary minus operators (level-1). In this case, execution proceeds from right to left. For example, $-X**-Y$ is equivalent to $-(X**(-Y))$, unless X is a constant in which case the sign is part of the constant. Note that the unary minus is the only operator that may directly follow another operator.

Example

$A+B*C/D**3$ is evaluated in four steps:

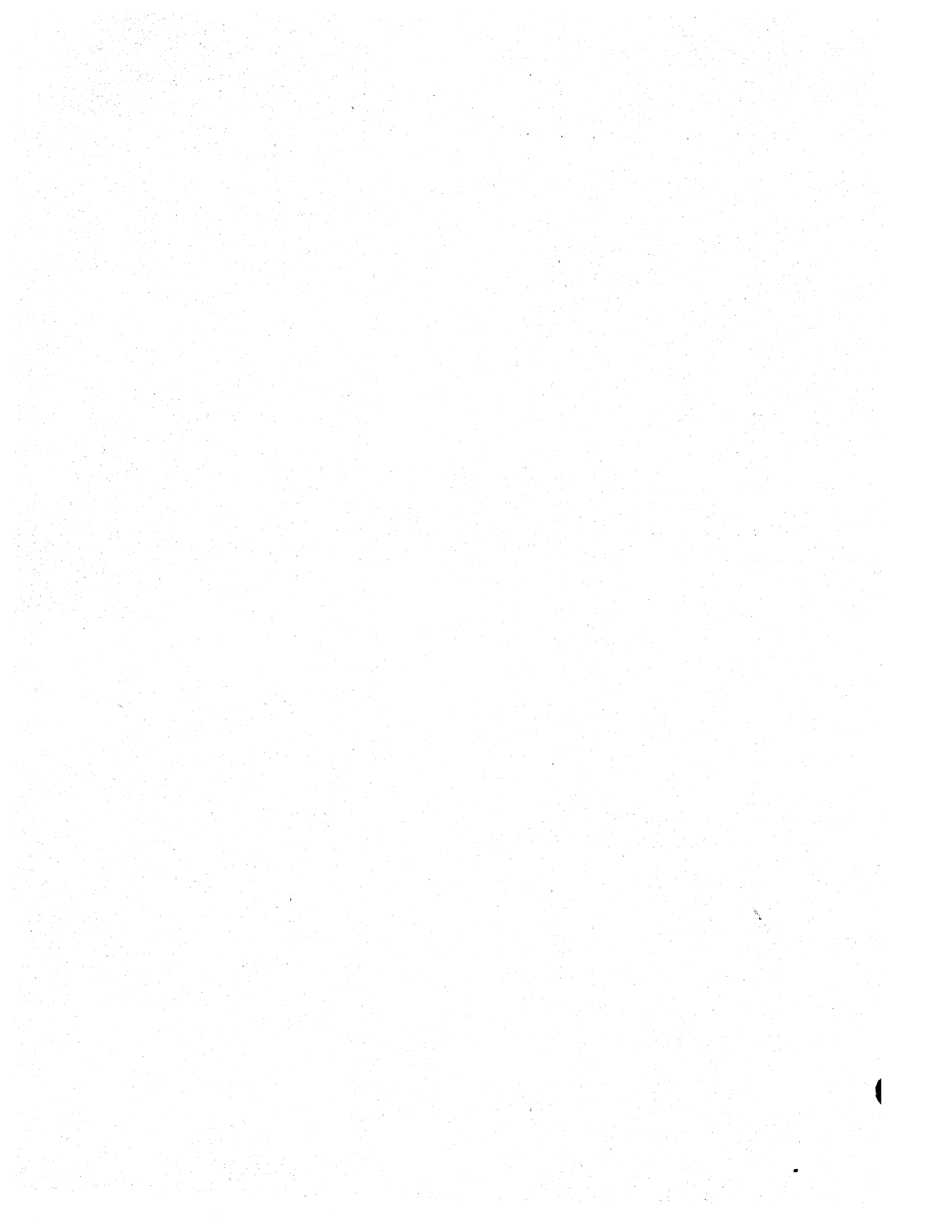
1. $B*C$
2. $D**3$
3. The result of Step 1 divided by the result of Step 2.
4. A plus the result of Step 3.

When operations are specified by parentheses, those within the innermost parentheses are performed first.

Example

$D*((A+B)**C)$ is evaluated in three steps:

1. $A+B$
2. The result of Step 1 raised to the power of C .
3. D multiplied by the result of Step 2.



Arithmetic operations are performed in one of two modes: floating-point or fixed-point. Floating- and fixed-point data units may be mixed within an expression. However, an operation is performed in fixed-point mode only if both operands are fixed-point. Exponentiation involving a scaled exponent is performed in floating-point mode.

The radix point of a fixed-point operand is determined by a data declaration or by an in-line definition. Using an in-line definition, the radix point is specified following the operand with the scaling specifier (...). An in-line definition overrides a data-declaration definition.

Example

A..5+B..7

In this example, A has a radix point of 5 and B has a radix point of 7.

Only an integer constant, an EQUALS tag, or a name defined in a MEANS declaration may follow the scaling specifier. An in-line definition is valid only for a particular reference. Any succeeding operand reference utilizes the radix point definition of the data declaration unless the radix point is again defined in-line.

Precision of fixed-point arithmetic operations is dependent upon the function of the statements.

5.1.1.1 Fractional Significance in Fixed-Point Operations

The rules for determining fractional significance in operations between two fixed-point operands A and B are described below. An operand may be a data unit, a constant or the result of a subexpression. The radix point of a data unit is the number of fractional bits defined in the data declaration or the in-line defined scaling specifier.

The radix point of a mixed or fractional constant equals:

1. $3.2 * n + 1$ truncated to an integer, if constant is decimal, or
2. $3 * n$, if constant is octal

where n is the number of fractional digits. The radix point of an intermediate result or subexpression is determined by application of the scaling rules.

In the discussion of scaling rules, the following abbreviations are used:

<u>Abbreviation</u>	<u>Meaning</u>
x	Radix point of A.
y	Radix point of B.
z	Radix point of receptacle in a replacement statement, or radix point of simulated receptacle in relational expression or programmer supplied override value.
$\min(x, y)$	The smaller value of x and y.

For relational expressions, FOR-expressions, or replacement statements with floating-point receptacles, the following are applied to determine the radix point of the simulated receptacle.

1. If $x \neq 0$ and $y \neq 0$, then $z = \min(x, y)$.
2. If $x = 0$ and $y = 0$, then $z = 0$.
3. If neither of the above is true, then z equals the nonzero scale factor (x or y, whichever is nonzero).

The programmer may override the Compiler determined value of z by enclosing an expression within parenthesis followed by the scaling specifier (..) and then specifying the desired value of z.

Example

$(A+B/C+D)..5$

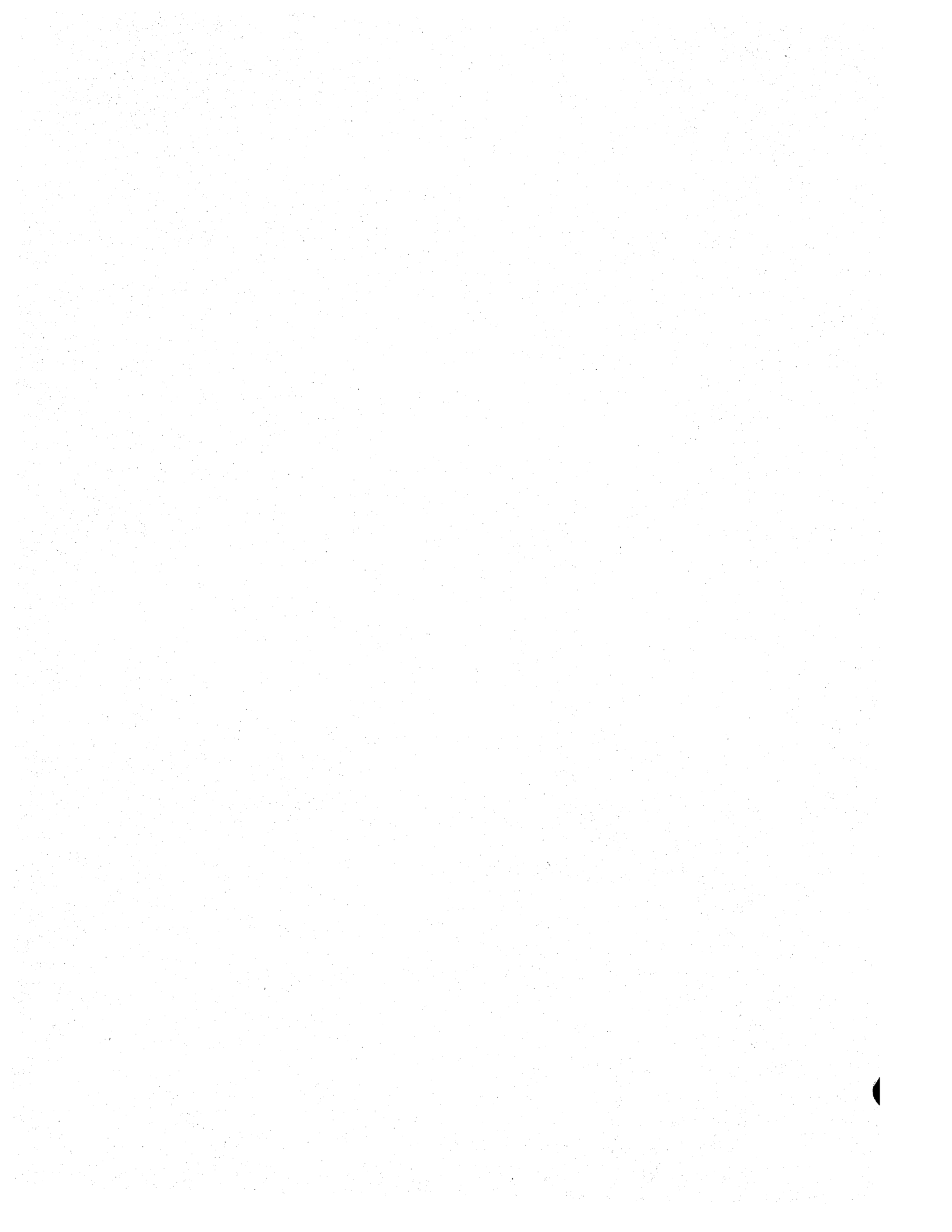
This specifies that each operation within the parentheses will be performed in accordance with the scaling rules for fixed-point arithmetic with "z" equal to 5.

The scaling rules for fixed-point arithmetic are now described as follows:

1. Addition and Subtraction ($A \pm B$)
 - a. If $x = y$, the radix point of the result is x.
 - b. If $\min(x, y)$ is greater than z, the radix point of the result is $\min(x, y)$.
 - c. If neither rule a nor rule b is true, the radix point of the result is z.

2. Multiplication ($A*B$)
 - a. No alignment prior to multiplication in relational expressions.
 - b. If x is greater than z , and A is the result of a previous multiplication, operand A is aligned to z prior to the multiplication. If y is greater than z , and B is the result of a previous multiplication, operand B is aligned to z prior to the multiplication.
 - c. The radix point of the result (product) is the sum of the radix points of the operands after application of rule a or rule b.
3. Division (A/B)
 - a. If y is greater than z , then B is aligned to z prior to the division.
 - b. A is aligned to $y + z$ so that the result (quotient) will have scaling equivalent to z .
4. Absolute value and complementation
 - a. There is no adjustment of scale factors; i. e., the scaling of the result equals the scaling of the operand.
5. Exponentiation
 - a. There is no adjustment; i. e., scaling of the result equals (exponent)*(operand scaling).

If both operands are constants, such as in addition, subtraction, multiplication and division, the above rules apply with the following consideration. Each of the constant operands has a user implied radix point as described at the beginning of this discussion. The rules produce an implied radix point for the result. On the other hand, constants are converted within the compiler to double precision binary constants with maximum precision. Attributed to each internal representation of the constant is the compiler generated radix point. Compiler evaluation of the constant expression is performed in strict double precision mode. The scaling rules applied to the compiler generated radix points yield a compiler radix point for the result of the constant expression. If the resultant radix point derived from the user implied radii is greater than the resultant radix point derived from compiler generated radii, then the final radix point for the evaluated constant expression is the compiler resultant radix point; otherwise, it is the resultant user implied radix point.



5.1.2 Relational Expressions

A relational expression performs a comparison between two operands via relational operators (see Table 5-2). If an operand of a relational expression is an arithmetic expression, the operations of the arithmetic expression are executed first. Comparisons between two arithmetic operands will be performed in fixed-point mode only if both operands are fixed-point. When comparisons are made between Hollerith operands, the shorter operand determines the number of characters to be compared. If one of the Hollerith operands is a constant, the necessary blank filling on the right is made to form equal length operands. A relational expression always results in a Boolean true or false value.

Example

$(A+B+C)*D \text{ EQ } E+F$

This expression is evaluated by comparing the result of $(A+B+C)*D$ with the result of $E+F$.

TABLE 5-2. RELATIONAL OPERATORS

OPERATOR	DEFINITION	OPERAND TYPES COMPARED
EQ	Equal	Arithmetic, Status, Hollerith, Boolean
NOT	Not equal	Arithmetic, Status, Hollerith, Boolean
LT	Less than	Arithmetic, Status, Hollerith
GT	Greater than	Arithmetic, Status, Hollerith
LTEQ	Less than or equal to	Arithmetic, Status, Hollerith
GTEQ	Greater than or equal to	Arithmetic, Status, Hollerith

5.1.3 Boolean Expressions

A Boolean expression consists of two or more operands connected by Boolean operators. The operands can be considered bit strings, i.e., a string of one or more consecutive bits, each having the Boolean value true or false, which is internally represented as 1 or 0 respectively. Operator definitions and hierarchy of evaluation are defined in Table 5-3.

TABLE 5-3. BOOLEAN OPERATORS

HIERARCHY OF EXECUTION	OPERATOR	DEFINITION
1	COMP	Complement or negation
2	AND	Logical multiply or intersection
3	OR	Logical add or union
3	XOR	Logical difference or exclusive OR

If Boolean operations are contained within parentheses, the innermost operation is executed first.

Example

A OR (COMP(A AND (B OR C))) AND C

The expression is evaluated as follows:

1. B or C
2. A AND the result of Step 1
3. COMP the result of Step 2
4. The result of Step 3 AND C
5. A OR the result of Step 4

Operands associated with logical operations in a Boolean expression may be of any type (i.e., Hollerith, numeric, status, or Boolean) or they may be relational expressions. If no Boolean operators are used, the operand of a Boolean expression must be a Boolean variable, Boolean constant, Boolean function, Boolean functional modifier, or a relational expression. If two operands result in bit strings of lengths a and b, where a and b are not equal, the length of each bit string is assumed to be the maximum of a and b with the shortest bit-string filled with zeros on the left. All bit strings are right-justified before the binary Boolean operations are performed. In arithmetic operations, bit-strings are assumed unsigned, fixed-point data with no scaling.

The primary use of Boolean expressions is in IF and FIND statements, which select statement execution options based on relational comparisons. The Boolean expression is also useful for manipulating bit strings and assigning values to Boolean data units.

When the value of a relational expression is used as an operand in a Boolean expression, each bit of the required bit string for the operand is assigned the value true (1) or false (0). When a Boolean data unit is used as an operand in a Boolean expression, it is always assumed to be a single bit in length. If the functional modifier BIT specifies a single bit of a data unit, that data unit is considered a Boolean operand.

When relational and Boolean operators are mixed in a Boolean expression, the relational operations are performed first and the resultant Boolean values are evaluated according to the Boolean operators.

Example

A LT B AND C EQ D

1. A LT B (true or false)
2. C EQ D (true or false)
3. (result of 1) AND (result of 2)

The results of Boolean operations can be shown in a truth table. Referring to Table 5-4, the A and B columns represent the assignment of truth values for these variables. The remaining columns show the truth values resulting from the Boolean operations. For example, if A and B are both false (represented internally as 0) then COMP B would be true, A AND B would be false, A OR B would be false, and A XOR B would be false.

TABLE 5-4. TRUTH TABLE

A	B	COMP B	A AND B	A OR B	XOR
0	0	1	0	0	0
0	1	0	0	1	1
1	0	1	0	1	1
1	1	0	1	1	0

5.1.4 Literal Expressions

A literal expression is similar to other expressions in that it specifies a single literal value expressed in terms of literal operators and associated operands. The operands that are allowed with literal expressions are: fields and variables typed as Hollerith, Hollerith functions, the functional modifiers BIT and CHAR, and Hollerith constants.

5.2.1 Absolute Value (ABS) Modifier

ABS is used for referencing the absolute value of a data element or arithmetic expression; it is an open function.

Format

ABS (data unit)

Explanation

ABS Specifies an absolute value operation on the specified element.

Data Unit A data unit or arithmetic expression.

Examples

1. ABS(ALPHA)

The example refers to the absolute value of the variable ALPHA.

2. ABS(CLASS(SIZE))

This example refers to the absolute value of the field, SIZE, in the item-area CLASS.

5.2.2 Bit (BIT) Modifier

BIT is used to reference a string of one or more bits in a data unit. Data unit bits are numbered with bit 0 specifying the leftmost bit. This function may be open if the starting bit and length specifications are positive integer constants and do not require movement across a word boundary.

Format

BIT (starting-bit, number-of-bits)(data-unit)

Explanation

BIT Indicates that bit specifications for a data element follow.

Starting Bit A numeric constant, data unit, or arithmetic expression. This bit specifies the initial bit position of the string.

Number of Bits Optional. A numeric constant, data unit, or arithmetic expression. It specifies the number of bits in the string. If this option is omitted, the number of bits is assumed to be 1.

Data Unit The name of a data unit. The data unit cannot be a system or local index.

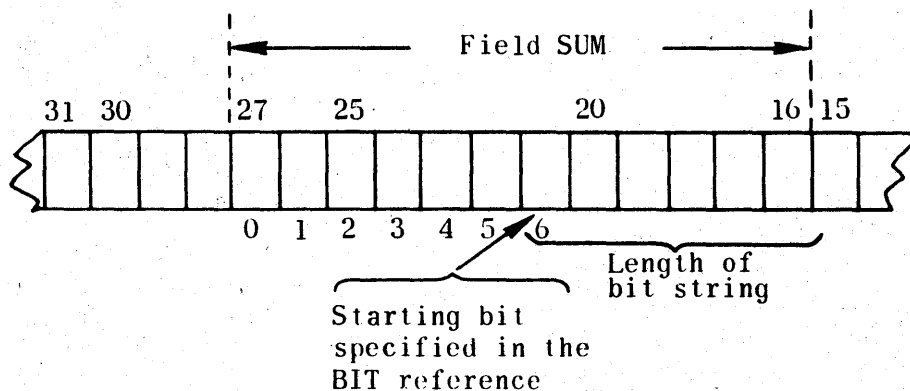
Examples

1. BIT(0,5),(ALPHA)

The string begins in bit 0 (the leftmost bit) and is five bits long. ALPHA is the name given to the variable by a previous declaration.

2. BIT(6,6),(BETA(N,SUM))

This bit string begins in bit 6 and is six bits long. The table (subtable or like-table) is BETA, the item index is N, and the field referenced is SUM. In this example, assume that field SUM has been defined such that it occupies bits 27-16 of a table word. Thus, the bit reference applies to word bits 21-16, as illustrated below:



Examples

1. CHAR(4,3)(BETA(A,SUM))

The character string begins in character position 1 and is three characters long. The table (subtable or like-table) is BETA, the item is 4, and the field referenced is SUM. This example can be visualized like Example 2 for BIT, the difference being that this uses character counts where there are eight bits per character.

2. CHAR(1)(BETA(N,SUM))

This character string begins in character position 1 and is one character long. The table (subtable or like-table) is BETA, the item index is N, and the field referenced is SUM.

5.2.4 Count (CNT) Number of Bits

CNT furnishes the count of the number of bits set (equal to 1) in the specified data element. CNT results in an integer value and may be used in a numeric or status expression. This is an open function.

Format

CNT (one-word data-unit)

Explanation

CNT Specifies the counting of bits set to 1 in the designated data unit.

One-word data-unit The name of a data unit contained in one word.

Example

CNT(TBLE(XX,FLDPOS))

The number of bits that are set in field FLDPOS of item XX of table TBLE will be counted.

M-5035

5.2.5 Core Address (CORAD) Modifier

CORAD is used to reference the core address of a data element. It is an open function.

Format

CORAD(data-unit or statement-name)

Explanation

CORAD Specifies the core address of the following data element or statement name.

Data Unit or Statement Name A data unit or statement identifier.

NOTE

The CORAD modifier always results in an unsigned, 16-bit value which represents the SY address of the data unit or statement name referenced. Under no circumstances will CORAD result in an 18-bit absolute address.

5.2.6 File Position (FIL) Modifier

FIL is used for file positioning. See Section 6 for usage examples. This is a closed function.

Format

FIL (name)

Explanation

FIL Specifies the positioning of a device to a particular file.

Name The identifier of a file.

5.2.7 Record Position (POS) Modifier

POS is used for record positioning. This is a closed function. See paragraphs 6.5.1 and 6.5.2 for usage examples.

Format

POS(name)

Explanation

POS Specifies the position of a file named within the parentheses.

Name The identifier of a file.

5.2.8 Record Length (LENGTH) Modifier

LENGTH is used to determine the length of the last record of an input or output operation; it is a closed function. See paragraph 6.7 for usage examples.

Format

LENGTH(name)

Explanation

LENGTH Specifies the length of a record for the file named.

Name The identifier of a file.

5.3 PROCEDURE LINKING

Procedure linking is accomplished through the procedure call, function call, the procedure switch call, and the return statement. This capability provides for program segmentation and increased efficiency through the elimination of statement duplication. Paragraph 4.1.12 gives instructions for declaring a procedure.

5.3.1 Procedure Call

A procedure call establishes transfer of control to a named procedure and may assign actual parameter values to the formal parameters defined in the procedure declaration.

All procedure input parameter linking is accomplished by transferring the values contained in the actual input parameters of the calling statement to the formal input parameters of the called procedure declarative statement. That is, an actual parameter and its corresponding formal parameter are usually distinct programmer-declared data units allocated to different locations in core. Therefore, modification of a formal input parameter in the procedure does not affect the value of the actual input parameter in the calling procedure. Procedure output parameter linking is accomplished by transferring the values contained in the formal output parameters of the called procedure's declarative statement to the actual output parameters specified in the calling statement upon procedure return. If an actual parameter is omitted, or if the same data unit is specified as both the actual parameter and the corresponding formal parameter, no transfer of values is performed. Addresses of data units may be transferred (simulating a call by name) by using the CORAD operator (see paragraph 5.2).

Format

```
name INPUT actual-input-parameter(s) OUTPUT
      actual-output-parameter(s) EXIT statement-name(s) $
```

Explanation

Name	Identifies the procedure to be executed.
INPUT	Optional. Specifies that the following list of actual parameters is to be input to the named procedure.
Actual Input Parameter(s)	Constants, data units, or expressions that replace the corresponding formal input parameter values during execution of the named

procedure. Actual parameters must agree in type with formal parameters. Multiple parameters are separated by commas. There must be a one-to-one correspondence with the formal parameters defined in the called procedure's declarative statement (see note below).

OUTPUT

Optional. Specifies that the list of actual parameters following are the outputs from the named procedure.

Actual Output Parameter(s)

Data units whose values are replaced by the corresponding formal output parameter values after execution of the named procedure. Actual parameters must agree in type with formal parameters. Multiple parameters are separated by commas. There must be a one-to-one correspondence with the formal parameters defined in the called procedure's declarative statement (see note below).

EXIT

Optional. Specifies the statement name(s) that follow are abnormal exit reentry points.

Statement Name

An identifier that replaces the corresponding formal exit name during execution of the named procedure. Program control is transferred to the named statement if a RETURN (see paragraph 5.3.3) referencing the formal exit name is executed.

NOTE

If the same data unit is specified as both the actual parameter and the corresponding formal parameter, no transfer of values is performed. Alternatively, if an actual parameter is omitted from the calling statement, no transfer of values is performed. In this case, the position of the actual parameter in the parameter list must be maintained with a comma.

Since the passing of parameters involves a passing of actual values, if the designated parameter is a table, subtable, etc., the entire table, subtable, etc., is actually transferred into the procedure receptacle. The procedure must, therefore, provide a receptacle of sufficient size. Any excess beyond the receptacle size is truncated. The CORAD modifier and the INDIRECT table option may be used when a table is specified as a procedure parameter and it is desirable not to have the entire table passed (see paragraph 4.2.2).

NOTE

It is not legal to use status constants as actual input or output parameters for forward reference procedures (procedures which are called before they are formally defined or declared with an EXTREF or LOCREF modifier). Furthermore, a local procedure may not be forward referenced before its local formal parameters are defined or declared, regardless of the parameter types.

Examples

1. AX. | | | | | TESTR INPUT 0 OUTPUT CLAS \$ | | | |

The statement AX will call the procedure TESTR. INPUT specifies that 0 will be passed to TESTR as the actual input parameter value. The actual parameter CLAS will receive the output value from procedure TESTR.

2. RCAC INPUT AP1 AP3 OUTPUT IMAGE
| | | | | EXIT AT3 \$ | | | | |

The procedure RCAC is to be called. AP1 and AP3 are the first and third actual input parameters. The actual parameter IMAGE will contain the output from procedure RCAC. If an abnormal exit is taken from RCAC, reentry to the calling procedure will be at the statement labeled AT3.

procedure. Actual parameters must agree in type with formal parameters. Multiple parameters are separated by commas. There must be a one-to-one correspondence with the formal parameters defined in the called procedure's declarative statement (see note below).

OUTPUT

Optional. Specifies that the list of actual parameters following are the outputs from the named procedure.

Actual Output Parameter(s)

Data units whose values are replaced by the corresponding formal output parameter values after execution of the named procedure. Actual parameters must agree in type with formal parameters. Multiple parameters are separated by commas. There must be a one-to-one correspondence with the formal parameters defined in the called procedure's declarative statement (see note below).

EXIT

Optional. Specifies the statement name(s) that follow are abnormal exit reentry points.

Statement Name

An identifier that replaces the corresponding formal exit name during execution of the named procedure. Program control is transferred to the named statement if a RETURN (see paragraph 5.3.3) referencing the formal exit name is executed.

NOTE

If the same data unit is specified as both the actual parameter and the corresponding formal parameter, no transfer of values is performed. Alternatively, if an actual parameter is omitted from the calling statement, no transfer of values is performed. In this case, the position of the actual parameter in the parameter list must be maintained with a comma.

Since the passing of parameters involves a passing of actual values, if the designated parameter is a table, subtable, etc., the entire table, subtable, etc., is actually transferred into the procedure receptacle. The procedure must, therefore, provide a receptacle of sufficient size. Any excess beyond the receptacle size is truncated. The CORAD modifier and the INDIRECT table option may be used when a table is specified as a procedure parameter and it is desirable not to have the entire table passed (see paragraph 4.2.2).

NOTE

It is not legal to use status constants as actual input or output parameters for forward reference procedures (procedures which are called before they are formally defined or declared with an EXTREF or LOCREF modifier). Furthermore, a local procedure may not be forward referenced before its local formal parameters are defined or declared, regardless of the parameter types.

Examples

1. AX TESTR INPUT 0 OUTPUT CLAS \$

The statement AX will call the procedure TESTR. INPUT specifies that 0 will be passed to TESTR as the actual input parameter value. The actual parameter CLAS will receive the output value from procedure TESTR.

2. RCAC INPUT AP1 AP3 OUTPUT IMAGE
 EXIT AT3

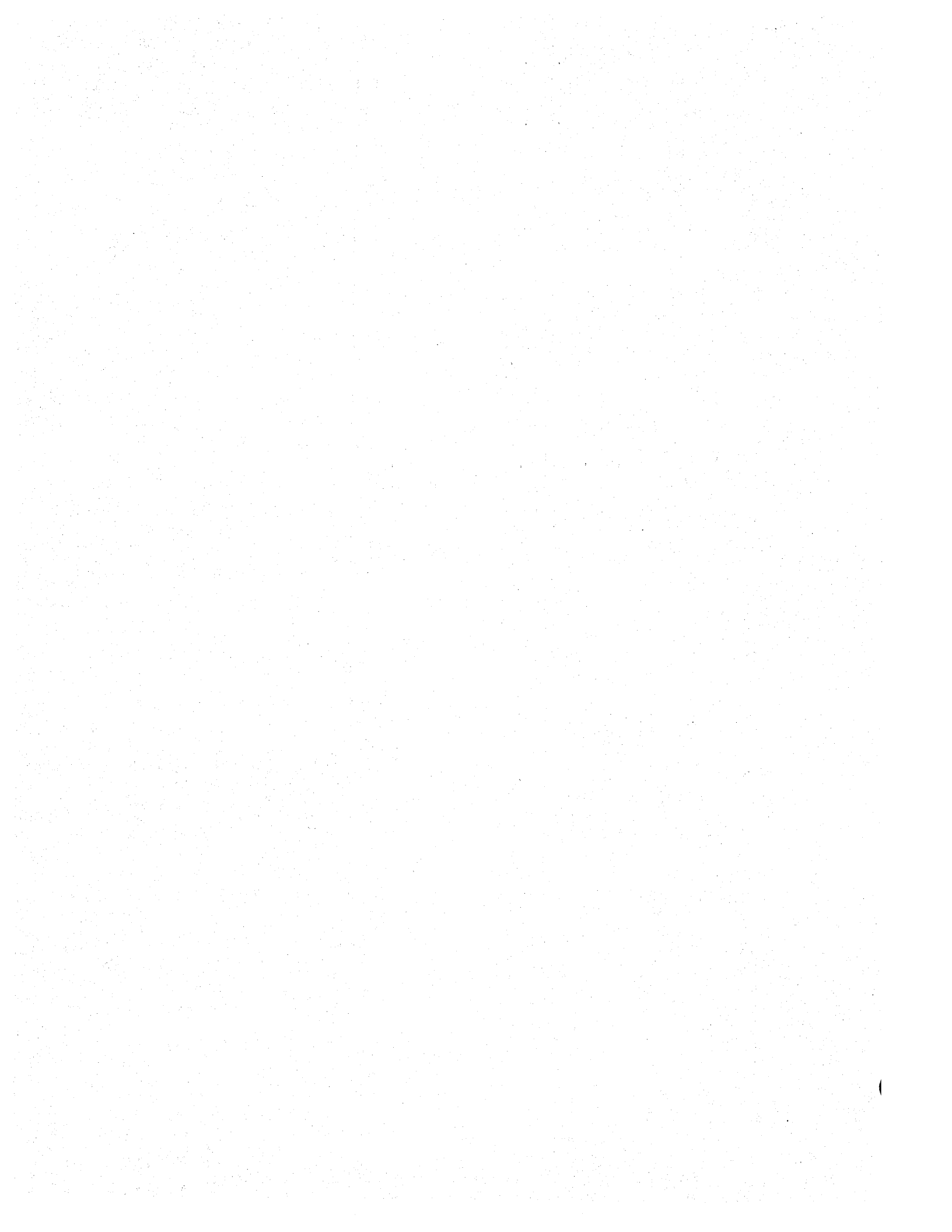
The procedure RCAC is to be called. AP1 and AP3 are the first and third actual input parameters. The actual parameter IMAGE will contain the output from procedure RCAC. If an abnormal exit is taken from RCAC, reentry to the calling procedure will be at the statement labeled AT3.

5.3.2 Function Call

A function call establishes transfer of control to a named function and assigns one or more values to the formal input parameters of the function. A function call may appear in dynamic statements or expressions; it resembles conventional mathematical function notation.

Format

name (actual-input-parameter, . . . , actual-input-parameter)



Explanation

Name	The name of the function to be executed.
Actual input Parameters(s)	Specifies each actual input to the function. There must be at least one specified; multiple parameters are separated by commas.

Example

The function call for the function TAN is TAN(AX, AY). AX and AY are the actual input parameters to be used as the values of the formal input parameters of the function. The function is evaluated and provides the value specified to complete the operation of the SET statement.

NOTE

It is not legal to use status constants as actual input parameters for forward reference functions (functions which are called before they are formally defined or declared with an EXTREF or LOCREF modifier). Furthermore, a local function may not be forward referenced before its local formal parameters are defined or declared, regardless of the parameter types.

5.3.3 Return (RETURN) Statement

The RETURN statement is a transfer of control operation used within a given procedure or function to exit from that procedure or function. There are three types of RETURN statements for procedures: a normal RETURN, an abnormal RETURN, and a conditional RETURN, dependent on a hardware key setting. All procedures should have at least one normal RETURN statement. However, a RETURN statement may be omitted if it immediately precedes the END-PROC declaration. There is only one type of return for a function.

Format

```
RETURN statement-name special-condition $  
RETURN (expression) $
```


2. PROCEDURE READ, OUTPUT, CARD, EXIT, EOF \$
RETURN, EOF \$
RETURN \$
END-PROC READ \$

The RETURN EOF statement results in a return to the alternate reentry statement corresponding to the parameter EOF in the procedure call. In this case, the output parameter value is not transferred.

3. RETURN, STOP \$
or
RETURN, STOP5 \$

These are examples of special condition returns. The first example results in an unconditional halt with the return executed after operator intervention. The second example results in a stop if key 5 is set, with the return executed after operator intervention; otherwise the return is executed.

4. RETURN, STEP9, STOP5 \$

In this example, if key 5 is set, a halt will occur. After operator intervention, the return will be made through STEP9 to the controlling procedure or function. Otherwise, a return through STEP9 is executed without halting.

5. RETURN, (X + Y + Z) \$

In this example of a return from a function, the expression X+Y+Z will be evaluated and returned to the calling procedure as the output value of the function.

5.3.5 Procedure Switch Call

A procedure switch call establishes a procedure link by specifying a transfer of control to one of the procedures named in the procedure switch declaration (see paragraph 4.2). Actual parameters are assigned to formal parameters named by the declaration.

The index value of the procedure switch call determines which procedure within the declaration is to be called. If the value of the switch index is outside the range of allowable values, control may be transferred to the named statement via the INVALID operator. Allowable index values are 0 through n-1, where n is the number of switch points. If the user does not use the INVALID option, it is his responsibility to see that the index is within the range of the switch.

If the procedure switch is defined with formal input and output parameters, actual input and output parameters must be included in the procedure switch call, as in a normal procedure call.

Format

```
name USING index INVALID statement-name INPUT
      actual-parameters OUTPUT actual-parameters $
```

Explanation

Name	The identifier of a previously defined P-SWITCH.
USING	Specifies that the following index indicates the procedure to be called. This parameter is not used in a procedure item switch call.
Index	A data unit, constant, or arithmetic expression whose integer value determines the procedure to be called. This parameter is not used in a procedure item switch call.
INVALID	Optional. Specifies that the procedure linkage should be accomplished only if the switch index is valid.

M-5035

Statement Name	Identifies the statement to which control is transferred if the invalid condition is met.
INPUT	Optional. Specifies that the list of actual parameters following are the inputs to the named procedure switch.
Actual Parameter	A data unit, constant, or expression that replaces the corresponding formal input parameter during execution of a procedure referenced by the procedure switch call. Multiple parameters are separated by commas.
OUTPUT	Optional. Specifies that the list of parameters following are to contain the outputs from the named procedure.
Actual Parameter	A data unit that is replaced by the corresponding formal output parameter after execution of a procedure referenced by the procedure switch call. Multiple parameters are separated by commas.

M-5035
Change 3

5.4 REPLACEMENT STATEMENTS

Replacement statements provide for the transfer of data from one data unit to another, permit performance of algebraic (both Boolean and numeric) manipulations according to a predefined hierarchy, and provide for the simultaneous exchange of values between data units.

5.4.1 Assignment (SET) Statement

Upon execution of an assignment statement, the value of the right term is transferred to one or more specified data units. The four types of assignment statements are: arithmetic, literal, status, Boolean, and multiword.

Format

SET receptacle(s) TO right-term \$

Explanation

SET	Specifies that one or more receptacles follow, to which data from the right term is to be transferred.
Receptacle(s)	A data element that is to receive a new value. Multiple receptacles that are to receive the same value may be specified, and are separated by commas. The receptacle type may be arithmetic, literal, status, or Boolean.
TO	Specifies that the right term follows.
Right Term	A data element or expression. Evaluation of the right term results in a single value. The right term must agree in type (arithmetic, Hollerith, status, or Boolean) with the receptacle.

Examples

1. SET A TO B \$

The value of A is replaced by the value of B.

2. SET A, B TO C \$

The values of A and B are replaced by the value of C.

3. SET A TO B THEN SET C TO D \$

4. SET GUNS TO 'ENGAGE' \$

5.4.1.1. Arithmetic Assignment Statement

The value specified by the right term in an arithmetic statement must be an arithmetic expression, which may include numeric functions and functional modifiers. The receptacle must be an arithmetic data unit. The following rules must be observed.

M-5035

1. When the value designated in the right term is specified to greater precision than that defined by the receptacle, the excess precision is truncated.
2. When the value is specified to less significance than that defined by the receptacle, the most significant bits will be filled with the sign bit if the field is signed. If the field is unsigned, the bits will be filled with zeros.
3. When the value is specified to less precision than that defined by the receptacle, the least significant bits will be set to zeros or ones.
4. When the value has greater significance than that accepted by the receptacle, the most significant bits of the value may be lost, depending on the actual number of bits allocated to the receptacle.
5. The mode of the receptacle defines the replacement as floating- or fixed-point.
6. Any arithmetic expression in the right term involving both a floating- and fixed-point data unit will be evaluated in the floating-point mode.
7. Fixed-point replacement aligns the radix point of the right term with the radix point of the receptacle.
8. The radix point of a receptacle or a right term may be defined by an in-line scaling definition.

Format

```
SET receptacle(s) TO expression SAVING numeric-data-unit  
OVERFLOW statement-name $
```

Explanation

SET Specifies that one or more receptacles follow, to which data from the right term is to be transferred.

Receptacle(s) A data element that is to receive a new value. Multiple receptacles that are to receive the same value may be specified, and are separated by commas. In this case, the assignment operations are performed from right to left.

TO Specifies that the right term follows.

Expression An arithmetic expression.

SAVING Optional. Specifies that the remainder of the last fixed-point division performed in the statement is to be saved.

Numeric Data Unit Any variable or field having an arithmetic data type.

OVERFLOW Optional. Specifies that a check for overflow caused by previous fixed-point (and floating-point, if under MONITOR option) arithmetic operations is requested. If an overflow condition resulted, all overflow indicators are turned off and control is transferred to the abnormal path specified by a statement name.

Statement Name The label of the statement to which control is transferred.

Examples

1. SET A TO (B+C)*D \$

The value of A is replaced by the result of (B+C)*D.

2. SET A TO .5 \$

The value of A is replaced with the constant value .5.

9. SET RESULT TO A*B+C-D/E \$

In this example, the required operations would be executed according to the predefined hierarchy: $((A*B)+C)-(D/E)$.

10. SET ALPHA TO BETA THEN GOTO ENDJOB \$

This compound statement transfers the value from data unit BETA to data unit ALPHA with conversion as required and then transfers to the location specified symbolically by ENDJOB.

5.4.1.2 Literal Assignment Statement

A literal SET statement stores in a literal (Hollerith data type) variable the value specified by the right term. The right term must be a literal expression which includes Hollerith functions, the functional modifiers BIT and CHAR, and Hollerith constants. Variables and fields typed as Hollerith and the functional modifier CHAR are permitted on the left side of the literal SET statement. If the right term is a Hollerith or CHAR-modified data unit, the replacement is executed from left to right. Characters are numbered from the left, starting with 0. Character 0 of the right term replaces character 0 of the receptacle, etc. If the size of the right term is smaller than the size of the receptacle and the right term is not a Hollerith constant, the excess characters of the receptacle are not affected. If the right term is a Hollerith constant and is smaller than the size of the receptacle, the constant is left-justified and blank-filled to the size of the receptacle. If the size of the right term is greater than the receptacle, the rightmost characters are truncated.

Examples

The following declarations will be referenced in the following examples:

TABLE TAT H NONE 10 \$

FIELD CATA H 5 \$

FIELD FATHOMS H 11 \$

END-TABLE TAT \$

VRBL COURSE H 10 \$

1. SET COURSE TO H(VALLEVERDE) \$

The Hollerith variable COURSE is replaced by the string of characters VALLEVERDE.

2. SET TAT(0,CATA) TO TAT(7,CATA) \$

The value of the Hollerith field CATA of item 7 of table TAT replaces field CATA of item 0 of table TAT.

3. SET CHAR(0)(TAT(7),CATA)) TO H(?) \$

The first character of field CATA of item 7 of table TAT is replaced by the Hollerith ?.

4. SET CHAR(0)(COURSE) TO CHAR(1)(TAT(5,CATA)) \$

The second character of field CATA of item 5 of table TAT is placed in the first character position of the variable COURSE.

5. SET BUFFER TO H(DEPTH) CAT TBLE(J,FATHOMS)

CAT H(NO ALERT) \$

Assuming BUFFER and FATHOMS are typed as Hollerith, the variable BUFFER might contain a string of characters similar to the following message, assuming that the value 73 FATHOMS was contained in TABLE (J, FATHOMS).

DEPTH 73 FATHOMS NO ALERT

5.4.1.3 Status Assignment Statement

The status SET statement assigns to a status variable, the value specified by a status constant, a status type data unit, or a status type function.

Example

```

RBL WEATHER S 'FAIR', 'COLD', 'RAINY', 'CLOUDY' $
TABLE TESTER H DENSE 20 $
FIELD CKOUT S 'ACCEPT', 'REPLACE', 'REJECT',
'FIX' $
END-TABLE TESTER $
.
.
.
SET WEATHER TO 'RAINY' $
SET TESTER (KOUNT, CKOUT) TO 'REJECT' $

```

5.4.1.4 Boolean Assignment Statement

A SET statement is classified as Boolean by the presence of a Boolean expression on the right side of the statement. Assignment of the bit string that results on the right side to the receptacle(s) on the left side must adhere to the following rules:

1. If the magnitude (length) of the bit string on the right is greater than the receptacle, the bit string is right-justified in the receptacle and truncation occurs to the excess bits on the left.
2. If the magnitude (length) of the bit string on the right is less than the magnitude of the receptacle, the bit string is right-justified in the receptacle and the unused bits are cleared (set to 0).

Examples

1. SET TEE TO 1 \$

This example assigns the value 1 to the Boolean variable TEE.

2. SET TEE TO BIT(0)(TRAP) \$

The variable TEE is assigned the value of the zero bit location in variable TRAP.

3. SET ALPHA TO BETA GT GAMMA AND DELTA \$

The variable ALPHA is assigned the logical product of the Boolean value of the relational expression BETA GT GAMMA with the Boolean value of the variable DELTA.

4. SET ALPHA TO BETA GT GAMMA AND DELTA AND TBLE(J,K) \$

Assume:

- a. ALPHA is a 16-bit integer variable.
- b. DELTA is a 32-bit integer variable filled with the octal constant 0(307070707).
- c. BETA and GAMMA are floating-point variables.
- d. TBLE (J, K) contains the value 0(25252525252).

Since DELTA and TBLE(J, K) are not Boolean data units, they are treated as bit strings for the purposes of the evaluation of this Boolean expression. In this context, the Boolean result of the relational expression BETA GT GAMMA is an arbitrary string of 0 or 1 bits. If BETA is greater than GAMMA, ALPHA will contain 0(020202). If BETA is not greater than GAMMA, ALPHA will contain 0 (16 bits cleared).

5.4.1.5 Multiword Assignment Statement

This statement assigns a value or values to a multiword data element. There are three types of multiword assignment statements:

1. Table-to-table.
2. Item-to-item.
3. Single word-to-multiword.

5.4.1.5.1 Multiword Table-to-Table Assignment Statement. This statement assigns the values of one table to another table. Since this is a block transfer, care should be taken to ensure that both table structures are similar and that the item size is identical. If the number of words in the receptacle is less than the right term, the excess words are lost. If the number of words in the receptacle is greater than the right term, then the extra words are not affected. This type of statement applies to subtables and like-tables as well as to tables. When vertical tables with major indexes appear in a table-to-table assignment, the major indexes will be used to determine the number of words to be transferred.

Example

The following declarations are referenced in the following multiword table assignment example:

```

TABLE  | | | PAGE H 9 100 $
END-TABLE PAGE $
TABLE  | | | BXNUM H 9 100 $
END-TABLE BXNUM $
.
.
.
SET BXNUM TO PAGE $

```

The words of table BXNUM are replaced by the words of table PAGE.

Examples

The following declarations are referenced in the examples:

```

TABLE | CATT | H | MEDIUM | 4 | $
SUB-TABLE | CATA | 0 | 2 | $
FIELD | CAT1 | H | 5 | $
FIELD | CAT2 | H | 5 | $
END-TABLE | CATT | $
TABLE | WHOLE | A | 3 | 2,3,4 | $
FIELD | W1 | I | 6 | U | 0 | 1A | $
FIELD | W2 | I | 12 | U | 1 | 11 | $
FIELD | W3 | I | 12 | U | 2 | 2A | $
END-TABLE | WHOLE | $
VRBL | LIB | I | 15 | U | P | 13 | $
VRBL | COMAP | H | 8 | P | H (REDKELLY) | $
VRBL | AXFLAG | B | $

```

1. SET WHOLE TO 15 \$

Every word of table WHOLE is set to 15.

2. SET CATT TO 0 \$

The table CATT is cleared to zeros.

3. SET CATT TO LIB \$

Every word of CATT is filled with the value contained in the variable LIB.

4. SET CATA TO WHOLE (1,2,3,W3) \$

Each word of the subtable CATA is assigned the value of the field W3 of item 1,2,3 in table WHOLE.

5. SET WHOLE(0,1,0) TO LIB..2*4 \$

Each of the three words in the item specified by 0,1,0 of table WHOLE is assigned the value of the arithmetic expression LIB..2*4.

6. SET CATT(1) TO COMAP \$

Each word of item 1 of table CATT is filled with all or part of the characters of the Hollerith variable COMAP. For example, since the AN/UYK-7 machine word size accommodates four characters, the characters REDK would be stored in each word of the item.

7. SET CATA TO AXFLAG \$

Each word of subtable CATA is replaced with the true or false (1 or 0) value of the Boolean variable AXFLAG.

5.4.2 Exchange Statement (SWAP)

The exchange statement swaps the values contained in two data units. The exchange statement may be viewed as two assignment statements that are executed simultaneously. The rules regarding data unit lengths and types in the exchange statement are the same as the rules for the assignment statement (see paragraph 5.4.1.1). However, both the left and right terms assume the role of the receptacle.

Format

SWAP data-unit data-unit \$

Explanation

SWAP Specifies the operation SWAP.

Data Unit The identifier of a data unit.

Examples

1. SWAP CRT1, CRT2, \$
2. SWAP ALPHA, ZILK, \$
3. SWAP CAT(X, CATB), CAT(X+1, CATB), \$

5.4.3 Shift (SHIFT) Operation

The SHIFT operation moves the contents of a data location into either the same or a different data location. During the operation, the data is shifted as prescribed.

Format

SHIFT data-unit shift-type shift-count INTO data-unit \$

Explanation

SHIFT	Specifies the operation.
Data Unit	The identifier for a data unit (two words or less in length).
Shift Type	One of the following: CIRC Specifies circular shift. ALG Specifies algebraic shift (sign fill). LOG Specifies logical shift (zero fill).
Shift Count	Specifies the number of positions to shift and the shift direction. This may be a constant, a data unit or an arithmetic expression. A negative shift count denotes a shift to the left, a positive shift count denotes a shift to the right (the sign must be explicitly specified for a left shift).
INTO	Optional. Specifies that the receiving data unit's name follows.

NOTE

The SHIFT statement is intended primarily to provide high-level access to the shift instructions in the AN/UYK-7 repertoire. Consequently several restrictions and limitations are imposed: the data-unit being shifted must not exceed 64 bits in length; partial-word circular shifts are not permitted (i. e., if CIRC is specified, the source data-unit must be defined as exactly 32 or 64 bits, or 4 or 8 characters); when a receptacle data-unit is specified, the normal data-type rules for assignment statements must be followed and the shift operation itself is independent of normal operand alignment or conversion which takes place prior to the assignment; because all left shifts must be performed using AN/UYK-7 circular shift instructions, left algebraic shift operations can result in filling to the right with magnitude bits.

Examples

1. SHIFT INT LOG -2 \$
2. SHIFT ROCK CIRC INCH INTO HOLE \$
3. SHIFT PAD (QBS, RIGHT) LOG -16 INTO
PAD (QBS, LEFT) \$

5.4.4 Pack (PACK) Operation

The PACK operation requests packing of specified data units into a specified data area.

Format

PACK data-unit WITH data-unit data-unit ... data-unit \$

Explanation

- | | |
|-----------|--|
| PACK | Specifies the operation. |
| Data Unit | The identifier of a data unit. |
| WITH | Specifies that the data units which follow are to be packed into the receptacle. |

The effect of the PACK statement is bit string transfers from the source location to the target area, where the strings are stored consecutively and without spacing. If the receptacle is a table or an item, the right side data units must be tables or items. If the receptacle is a variable or field, the right side data units must be variables or fields.

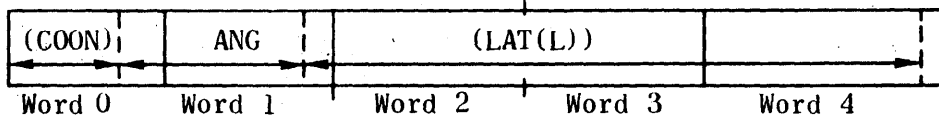
Example

PACK QUAD WITH COON, FIX(L, ANG), LAT(L) \$

Assume:

- a. COON is a half-word variable.
- b. ANG is a 40-bit field.
- c. LAT(L) is a three-word field in item area LAT.

The variable QUAD will receive the following contributions:



presuming that QUAD is a storage area capable of containing the specified bit strings.

5.5 CONTROL STATEMENTS

A control statement alters or affects program flow within a procedure. There are three types of control statements:

1. GOTO statement name
2. GOTO switch name
3. STOP

5.5.1 GOTO Statement Name

This statement performs a transfer to a named statement and may specify a governing special condition imposed by a console hardware switch.

Format

GOTO statement-name special-condition \$

Explanation

GOTO	Specifies a transfer of control.
Statement Name	The name of the statement to which control is to be transferred.
Special Condition	Optional. Specifies a machine-dependent condition of the console key settings. For the AN/UYK-7, these are KEY1, KEY2, KEY3, STOP, STOP5, STOP6 and STOP7.

Examples

1. GOTO UPDATE \$

Execution of this statement transfers program control to the statement labeled UPDATE.

2. GOTO ERRDIAG KEY1 \$

A transfer is made to statement label ERRDIAG if console key 1 is on.

3. GOTO ALTRNATE STOP5 \$

If console STOP5 is on, program execution halts. When restart is accomplished, a transfer is made to statement label ALTRNATE.

5.5.2 GOTO Switch Name

This statement performs a transfer of control to a statement label listed within an index- or item-switch declaration. Transfer may be made conditional as governed by a specified console key setting.

1. Referencing an Index-Switch.

Transfer is based upon the value of the switch index. This value is specified in the GOTO statement. No check is made by the Compiler as to the validity of the index. Therefore, if the index is outside the range of switch index values, program

execution control can be lost. To prevent this, specification of the INVALID operator directs the Compiler to perform a validity check; if the index is outside the range of index values, control will be transferred to a programmer-supplied statement label.

2. Referencing an Item-Switch.

The value of the variable specified in the item-switch is compared against the list of constants defined in the switch declaration. If a match is found, transfer is made to the corresponding statement name. If a match is not found, control is transferred to the next statement or to a statement specified after the INVALID operator.

Format

GOTO switch-name value INVALID statement-name special-condition \$

Explanation

GOTO	Specifies a transfer of control.
Switch Name	The name of the referenced index- or item-switch.
Value	A constant, arithmetic expression, or data unit that provides an index value pointing to a switch point. This parameter is not to be used when referencing an item-switch.
Special Condition	Optional. Specifies the machine dependency conditions to be imposed by console key settings. KEY1, KEY2, KEY3, STOP, STOP5, STOP6, and STOP7 are the CMS-2 identifiers used for the corresponding key settings on the AN/UYK-7.
INVALID	Optional. Specifies a transfer of control to the following named statement if the index is outside the range of the index-switch values or if no match is found in an item-switch.

Statement Name Control is transferred to this statement if the switch test is invalid.

Examples

The following declarations are referenced in the GOTO switch examples:

```

VRBL    I    $
SWITCH SWA SA1, SA2, SA3 $
VRBL    SWX $
SWITCH SWB (SWX) $
   60,    SB1 $
   30,    SB2 $
   20,    SB3 $
END-SWITCH SWB $

```

1. GOTO SWA I \$

Program control is transferred to the switch point of SWA specified by I. For example, if I equals 2, control is transferred to statement SA3.

2. GOTO SWA I INVALID SA4 \$

This is the same as Example 1, except that if I is outside the valid index range, control is transferred to statement SA4. For example, if I equals 4, control is transferred to SA4.

3. GOTO SWB \$

Program control is transferred to the statement corresponding to the value in SWX. For example, if SWX equals 30, control is transferred to statement SB2; however, if SWX equals 35, program control continues to the next sequential statement.

4. GOTO SWB INVALID SB4 \$

This is the same as Example 3, except that if the value of SWX is not found in the switch definition, control is transferred to statement SB4. For example, if SWX equals 40, control is transferred to SB4.

5. GOTO SMA 1 KEY 1 \$

Program control is transferred to the statement labeled SA2 if console key 1 is on.

5.5.3 STOP Statement

The STOP statement temporarily suspends program execution. This statement is legal only in programs being compiled to execute in the executive state.

Format

STOP special-condition \$

Explanation

STOP	Specifies a suspension of program execution.
Special Condition	Optional. Specifies machine dependent conditions of the console key settings.

5.6. DECISION STATEMENTS

The IF operator allows for conditional execution of one or more statements. The condition is based on one of the following four types of decisions:

1. Logical
2. Search
3. Validity
4. Parity

A decision statement may be followed by the ELSE operator which allows for execution of one or more statements if the result of the condition is false.

5.6.1 Logical Decision Statement

The logical decision statement evaluates a specified Boolean condition and reduces the evaluation to a true or false result.

Format

IF Boolean-condition THEN statement(s) \$

Explanation

IF Specifies that an evaluation, resulting in a Boolean true or false condition, is to be made.

Boolean Condition Specifies a Boolean condition, defined in CMS-2 notation by operands (constants, data units, and arithmetic expressions) and relational or Boolean operators.

THEN Specifies that the statement or statements that follow are to be executed only if the result of the Boolean condition is true.

Statement(s) A simple or compound statement, or a block of dynamic statements.

See paragraph 5.8 for an explanation of blocks and the requirements for compounding (nesting) decision statements.

Examples

1. IF A EQ B THEN GOTO S1 \$
SET A TO A + 1 \$

Control is transferred to statement S1 when A equals B. Otherwise, the set statement is executed.

2. IF A+B LT EQ B*C THEN SET C TO D
THEN SET E TO A \$

C is set to D, and E is set to F when A+B is less than or equal to B*C.

3. IF WEATHER EQ 'RAINY' THEN
GOTO EVALWTHR \$

Control is transferred to statement EVALWTHR if the status of WEATHER is 'RAINY'.

4. IF E AND F OR (G GT H AND I) THEN
GOTO S1 \$

If the Boolean variables E and F are both true (1), or if G is greater than H and the Boolean variable I is true, then control is transferred to statement S1.

5. IF BOOL THEN GOTO TRUE \$

If the Boolean variable BOOL is set to the true state (1), program control is transferred to the statement labeled TRUE.

6. IF COMP(BOOL), THEN GOTO FALSE \$

If the Boolean variable BOOL is set to the false state (0), taking the complement will make it true; hence, the true path is taken with program control transfer being made to statement FALSE.

7. IF AGE GT 65 AND MARSTAT EQ 'MARRIED' THEN
SET CODE TO 'A' THEN
GOTO PROCESSA \$

The relational expressions (GT and EQ) are first evaluated and reduced to a true or false condition. These two results are then logically tested (AND) for the final determination of a true or false condition. If evaluated true, the status variable CODE is set and program control is transferred to statement PROCESSA.

5.6.2 Table Search Statement

The table search statement provides the capability of searching a table for data that satisfies specified end conditions. The statement is a combination of a FIND statement and a search decision statement.

The FIND loop is terminated when the value of the table element, specified by the index, first satisfies the condition. If the condition is satisfied, the loop index points to the element that satisfied the condition. The loop is also terminated when the index has reached its final value.

The FIND statement must always be followed immediately by a search decision statement, which may in turn be followed by an ELSE statement (Paragraph 5.6.5).

5.6.2.1 FIND Statement

Format

statement-label. FIND expression VARYING loop-index
initial-value final-value increment \$

Explanation

Statement Label. Optional. The name by which this statement is referenced. This label is required if the table search is resumed.

FIND Specifies a table search.

Expression A relational expression, the first term of which must be a subscripted table reference using the loop index.

VARYING	Optional. Specifies that the operands that follow, control the loop. If not included, the loop index is varied from 0 in increments of 1 within the limits of the table. The loop index, initial value, and final value are allowed only if VARYING is given.
Loop Index	Optional. The name of an index or integer variable to be varied. It must be included whenever VARYING is used and it must be one of the subscripts included in the table reference. If the VARYING clause is omitted, the first subscript in the table reference will be used as the loop index.
Initial Value	Optional. FROM followed by a constant, arithmetic expression, or data unit that specifies the beginning index value of the loop. If omitted, the initial value is 0.
Final Value	One of the following: <ol style="list-style-type: none">1. May be THRU followed by a constant, arithmetic expression or data unit. This value signifies the last pass through the loop.2. May be WITHIN followed by the name of a horizontal or vertical table, subtable, or like-table. The value assigned is the number of items defined for the table or the current value of the major index.

Increment Optional. BY followed by a constant, arithmetic expression, or data unit. If this value is to be a decrement, BY must be followed by a minus sign. If omitted, the increment is 1. When varying within a table by a negative value, the initial value is the number of items defined for the table or the major index, and the final value is 0.

5.6.2.2 Search Decision Statement

The search decision statement must immediately follow a FIND statement. The FOUND/NOTFOUND condition is determined by the results of the FIND search.

Format

```
IF DATA FOUND THEN statement(s) $  
IF DATA NOTFOUND THEN statement(s) $
```

Explanation

IF	Specifies that a condition is to be evaluated as true or false.
DATA	A compiler control word used to clarify the statement.
FOUND	Specifies the condition to evaluate upon satisfaction of the FIND condition.
NOTFOUND	Specifies the condition to evaluate when the search loop is completed.
THEN	Specifies that the statement or statements that follow are to be executed if the FOUND or NOTFOUND condition is true.

Explanation

ELSE Specifies that the statement or statements that follow are to be executed only if the result of the Boolean condition in the corresponding previous decision statement is false.

Statement(s) A simple or compound statement, or a block of dynamic statements.

See paragraph 5.8 for an explanation of blocks and the requirement for compounding (nesting) decision statements.

Examples

1. IF A EQ B THEN SET C TO D \$
ELSE SET C TO 1 \$

C is set to D if A equals B. Otherwise C is set to 1.

2. IF BOOL THEN PROCA \$
ELSE BEGIN \$
SET E TO F \$
SET G TO H \$
END \$
PROCB \$

If the Boolean variable BOOL is true, then procedure PROCA is called. If BOOL is false, then the block of set statements is executed. In either case, PROCB is then called.

5.7 LOOP STATEMENTS

Loop statements direct repeated execution of a specified group of statements (vary block) or perform a table search.

5.7.1 Vary Block Statements

A vary operation is used to execute one or more statements a specified number of times (at least one time). The statements to be executed are bracketed by a VARY statement and an END vary statement. This group of statements is defined as a vary block. The number of passes through the loop is controlled by a loop index. Multiple vary loops on the same level are allowed within the same VARY statement.

Statement(s) A simple or compound statement, or a block of
dynamic statements.

5.6.2.3 Table Search Format

A generalization of the FIND statement in combination with the required search decision statement follows:

1. FIND statement \$
 IF DATA FOUND THEN ... (found sequence, with or without
 RESUME) ... \$
 :
 :
 (search completed sequence, should not RESUME)
 :
 :
2. FIND statement \$
 IF DATA NOTFOUND THEN ... (search completed sequence,
 should not RESUME) ... \$
 :
 :
 (found sequence, with or without RESUME)
 :
 :
 :

The search completed sequence in the first case and the found sequence in the second case may be ELSE statements.

5.6.2.4 Table Search Examples

```
1. FIND PAR(K, PLOTA) EQ 'TARGET' $  
   IF DATA NOTFOUND THEN  
   GOTO AVERAGE $  
   (found sequence)  
AVERAGE. (not found sequence)
```

'TARGET' is compared against field PLOTA in table PAR, starting at item 0 and continuing until an equivalence is found or the complete table has been searched (not found) and an exit made to AVERAGE. If the FIND is satisfied before the complete table has been searched, control is transferred out of the loop and the remaining items are not searched. The variable K is the loop index.

```
2. STEP1. FIND CATD(H, 2) EQ CATC  
   VARYING H THRU 6 $  
   IF DATA NOTFOUND THEN GOTO STEP2 $  
   SET TALLY TO TALLY + 1 $  
   RESUME STEP1 $  
STEP2. (exit sequence)
```

This sequence of statements increments TALLY each time the third word in the first seven items of CATD equals CATC. If no equivalence is found or the varying portion of the FIND is satisfied, control is transferred to STEP2. The RESUME statement continues the search only if an equivalence has occurred. It is not logical to resume a FIND loop on a not-found condition since the search continues without reinitialization. In the example, STEP2 will always be executed whether or not data is found, since completion of the table search will be interpreted as a true NOTFOUND condition and exit will be via THEN GOTO STEP2.

The following three examples illustrate a table search for a specified condition using a FIND statement with a search decision statement of FOUND and NOTFOUND, respectively, in the first two examples, and a VARY block in the third example. All three examples produce the same result: a transfer to ALARM is made if the specified condition is met less than five times during the search; otherwise, processing continues with the next instruction in sequence.

3. SET KOUNT TO 0 \$
AIRREADY. FIND AIRTABL(TX,FLTSTATS)
EQ 'READY' \$
IF DATA NOTFOUND THEN GOTO NEXT \$
SET KOUNT TO KOUNT + 1 \$
RESUME AIRREADY \$
NEXT. IF KOUNT LT 5 THEN GOTO ALARM \$
(next instruction)

4. SET KOUNT TO 0 \$
AIRREADY. FIND AIRTABL(TX,FLTSTATS)
EQ 'READY' \$
IF DATA FOUND THEN
SET KOUNT TO KOUNT + 1 THEN
RESUME AIRREADY \$
IF KOUNT LT 5 THEN GOTO ALARM \$
(next instruction)

```
5. SET KOUNT TO 0 $  
AIRREADY. VARY TX WITHIN AIRTABL $  
IF AIRTABL(TX, FLT, STATS)  
NOT 'READY' THEN  
RESUME AIRREADY $  
SET KOUNT TO KOUNT + 1 $  
END AIRREADY $  
IF KOUNT LT 5 THEN GOTO ALARM  
(next instruction)
```

5.6.3 Validity Decision Statement

The validity decision determines whether a subscripted data reference is valid. For example, if a horizontal table is defined with four items, any reference with the item index larger than 3 would be invalid.

Format

IF table-element VALID/INVALID THEN statement(s) \$

Explanation

IF	Specifies that a decision is to be performed.
Table Element	A subscripted table element where the subscript calculation is checked for validity.
VALID/INVALID	Specifies a condition of a valid or invalid subscript.
THEN	Specifies that the statement or statements that follow are to be executed if the table element reference is VALID or INVALID.
Statement(s)	A simple or compound statement or a block of dynamic statements.

Examples

The following declaration is referenced in the validity decision examples:

```
TABLE CATA H 5 2 $
SUB-TABLE CATA 0 1 $
END-TABLE CATA $
```

```
1. IF CATA(I,0) VALID THEN GOTO S1 $
```

Program control is transferred to statement S1 if I has a value of 0 or 1.

```
2. IF CATA(I,0) INVALID THEN GOTO S1 $
```

Program control is transferred to statement S1 if I is not zero.

5.6.4 Parity Decision Statement

The parity decision statement determines parity by testing the data specified for an odd or even number of 1-bit settings. If the sum of the 1-bits contained within the data unit is an even number, the parity is considered even. If the sum is an odd number, the parity is considered odd.

Format

IF one-word data-unit ODDP THEN statement(s) \$

IF one-word data-unit EVENP THEN statement(s) \$

Explanation

IF Specifies that a condition is to be evaluated as true or false.

One-word data-unit The identifier of a data unit contained in one word.

ODDP/EVENP The decider as to whether the parity condition of the value contained in the data unit is odd or even.

THEN Specifies that the statement or statements that follow are to be executed if the ODDP or EVENP condition is true.

Statement(s) One or more (connected by THEN) dynamic statements.

Example

IF STAT EVENP THEN SET FLAG TO 0 \$ | | | |

The bits set to 1 in the variable STAT are tested to see if their sum is an even number. If so, the variable FLAG is set to 0.

5.6.5 ELSE Statement

Any of the four types of decision statements may be immediately followed by an ELSE statement which allows for execution of one or more statements if the result of the condition is false.

Format

ELSE statement(s) \$

Explanation

ELSE Specifies that the statement or statements that follow are to be executed only if the result of the Boolean condition in the corresponding previous decision statement is false.

Statement(s) A simple or compound statement, or a block of dynamic statements.

Refer to paragraph 5.7 for an explanation of blocks and the requirement for compounding (nesting) decision statements.

Examples

1. IF A EQ B THEN SET C TO D \$
ELSE SET C TO 1 \$

C is set to D if A equals B. Otherwise C is set to 1.

2. IF BOOL THEN PROCA \$
ELSE BEGIN \$
SET E TO F \$
SET G TO H \$
END \$

PROCB \$

If the Boolean variable BOOL is true, then procedure PROCA is called. If BOOL is false, then the block of set statements is executed. In either case, PROCB is then called.

5. 6. 6 Nested Decision Statements

Because the connector THEN serves a special purpose within decision statements, IF and FIND statements cannot directly appear after THEN or ELSE associated with an IF statement. However, the compiler interprets all statements within a statement block (Paragraph 5.7) as the equivalent of a simple statement. This provides the means of nesting decision statements and gives rise to the following rule:

If an IF or FIND statement is to appear following the connector THEN or the operator ELSE associated with another decision statement, the nested IF or FIND statement must appear within a statement block. Furthermore, when a statement block appears within an IF statement, execution of all statements within the block is subject to the condition of the IF statement.

Examples

1. IF A THEN

B1. BEGIN \$

FIND TAB(I) EQ B \$

IF DATA FOUND THEN SET F TO 1 \$

ELSE

SET F TO 0 \$

END B1 \$

ELSE

B2. BEGIN \$

IF B LT 10 THEN SET B TO B+1 \$

ELSE

VARY I WITHIN TAB \$

SET TAB(I) TO 0 \$

END \$

END B2 \$

In this example, the outer decision statement is logically equivalent to the form:

```
IF A THEN block B1 ELSE block B2 $
```

The BEGIN and END statements are necessary due to the nested FIND statement in B1 and the nested IF statement (with its nested VARY loop) in block B2.

```
2.  IF  STATUS EQ 'START' THEN
      SET STATE TO 0 THEN
      VARY I WITHIN ANS $
      SET ANS(I,SUM) TO 0 $
      END THEN
      COMPUTE $
ELSE FINISH $
```

In this example, the VARY block is the equivalent of a simple statement within the compound conditional portion of the decision statement.

5.7 STATEMENT BLOCKS

A statement block is a group of simple or compound statements initiated by a BEGIN, VARY, or FOR statement and terminated by an END statement. A statement block is interpreted by the compiler as the equivalent of a simple statement. A value block (Paragraph 5.7.3.2) is a special block used only in the construction of FOR blocks; a value block is not the equivalent of a simple statement.

5.7.1 BEGIN Block

Format

```
statement-label. BEGIN $  
    :  
dynamic statements  
    :  
END statement-name $
```

Explanation

Statement-label	Optional. The name by which the block is identified.
BEGIN	Specifies the start of a block.
END	Specifies the end of a block.
Statement-name	The name, if any, given to the block as specified in the BEGIN statement.

The BEGIN and END statement brackets serve only to delimit statement blocks; the BEGIN statement does not specify a dynamic processing function as does the VARY or FOR statement (Paragraphs 5.7.2 and 5.7.3). Because the BEGIN statement does not define a loop structure, it cannot be resumed. Statement blocks delimited by BEGIN and END may be nested within, and in the same manner as, VARY blocks and FOR blocks.

Example

```
L1. BEGIN $
      SET A TO B $
      VARY I WITHIN TAB $
        BEGIN $
          SET C TO D**A $
          IF C LT 100 THEN SET TAB(I) TO C $
        END $
      SET A TO A+1 $
    END $
  END L1 $
```

Three statement blocks are defined in the above example: an outer block, L1, delimited by BEGIN and END; a VARY block nested within block L1; and a BEGIN/END block embedded in the VARY block. Note that the two BEGIN statements and their associated END statements serve only to group dynamic statements in a logical manner; removal of these bracketing statements would not alter the processing function of this program segment.

5.7.2 VARY Block

A vary operation is used to execute one or more statements zero or more times. The statements to be executed are bracketed by a VARY statement and an END vary statement. This group of statements is defined as a vary block. The number of passes through the loop is controlled by zero or more loop indices, a WHILE condition, and/or an UNTIL condition.

5.7.2.1 VARY Statement

The VARY statement initializes the loop and specifies controls for the number of passes through the loop (see Figure 5-1).

Format

statement-label. VARY index-clause(s) WHILE clause UNTIL clause \$

Explanation

Statement Label.	Optional. The name by which the vary block is referenced. This label is required if the VARY loop is resumed.
VARY	Specifies the start of a vary block.
Index-clause(s)	Optional. Specifies the data units to be incremented or decremented and possibly tested (Paragraph 5.7.2.1.1). Multiple index-clauses are separated by commas.
WHILE clause	Optional. Specifies a condition which is to be tested <u>before</u> each pass through the loop (Paragraph 5.7.2.1.2).
UNTIL clause	Optional. Specifies a condition which is to be tested <u>after</u> each pass through the loop (Paragraph 5.7.2.1.3).

A program's execution must not depend on incrementing and testing of the indices and the evaluation of the UNTIL clause being performed in any predefined order. The compiler will order the end loop evaluation according to optimal code generation.

Simple and compound statements may also be grouped together into blocks by means of the brackets BEGIN or VARY and END. Statement blocks are particularly useful, and in fact are required, when compounding (nesting) of decision statements is desired.

5.8.1 Statement Blocks

A statement block is a group of simple or compound statements initiated by a VARY or BEGIN statement and terminated by an END statement. A statement block is interpreted by the compiler as the equivalent of a simple statement.

Format

```
statement-label. VARY . . . $  
or  
statement-label. BEGIN $  
:  
dynamic statements  
:  
END statement-name $
```

Explanation

Statement-label	Optional. The name by which the block is identified. This label is required on the VARY statement if the VARY loop is resumed.
VARY . . .	A VARY statement is specified in paragraph 5.7.1.1.
BEGIN	Specifies the start of a block.
END	Specifies the end of a block.
Statement-name	The name, if any, given to the block as specified in the VARY or BEGIN statement.

The BEGIN and END statement brackets serve only to delimit statement blocks; the BEGIN statement does not specify a dynamic processing function as does the VARY statement, described in paragraph 5.7.1. Because the BEGIN statement does not define a loop structure, it cannot be resumed. Statement blocks delimited by BEGIN and END may be nested within, and in the same manner as, VARY blocks.

Example

```
L1. BEGIN $
      SET A TO B $
      VARY I WITHIN TAB $
        BEGIN $
          SET C TO D**A $
          IF C LT 100 THEN SET TAB(I) TO C $
        END $
      SET A TO A+1 $
    END $
  END L1 $
```

Three statement blocks are defined in the above example: an outer block, L1, delimited by BEGIN and END; a VARY block nested within block L1; and a BEGIN/END block embedded in the VARY block. Note that the two BEGIN statements and their associated END statements serve only to group dynamic statements in a logical manner; removal of these bracketing statements would not alter the processing function of this program segment.

5.8.2 Compound Decision Statements

Because the connector THEN serves a special purpose within decision statements, IF statements (and therefore FIND statements) cannot directly appear after THEN or ELSE associated with an IF statement. However the compiler interprets all statements within a statement block (i. e. , all statements from BEGIN or VARY to the corresponding END statement) as the equivalent of a simple statement. This provides a convenient means

of compounding, or nesting, IF statements and gives rise to the following rule.

If an IF or FIND statement is to appear following the connector THEN or the operator ELSE associated with another IF statement, the nested IF or FIND statement must appear within a statement block. (Note that VARY may directly follow THEN or ELSE because the VARY statement defines the start of a block.) Furthermore, when a statement block appears within an IF statement, execution of all statements within the block is subject to the condition of the IF statement.

Examples

```
1.  IF A THEN
      B1. BEGIN $
          FIND TAB(I) EQ B $
          IF DATA FOUND THEN SET F TO 1 $
          ELSE
              SET F TO 0 $
          END B1 $
      ELSE
          B2. BEGIN $
              IF B LT 10 THEN SET B TO B+1 $
              ELSE
                  VARY I WITHIN TAB $
                  SET TAB(I) TO 0 $
                  END $
          END B2 $
```

In this example, the outer decision statement is logically equivalent to the form:

```
IF A THEN block B1 ELSE block B2 $
```

The BEGIN and END statements are necessary due to the nested FIND statement in B1 and the nested IF statement (with its nested VARY loop) in block B2.

```
2.  IF  STATUS EQ 'START' THEN
      SET STATE TO 0 THEN
      VARY I WITHIN ANS $
        SET ANS(I,SUM) TO 0 $
      END THEN
      COMPUTE $
ELSE FINISH $
```

In this example, the VARY block is the equivalent of a simple statement within the compound conditional portion of the decision statement.

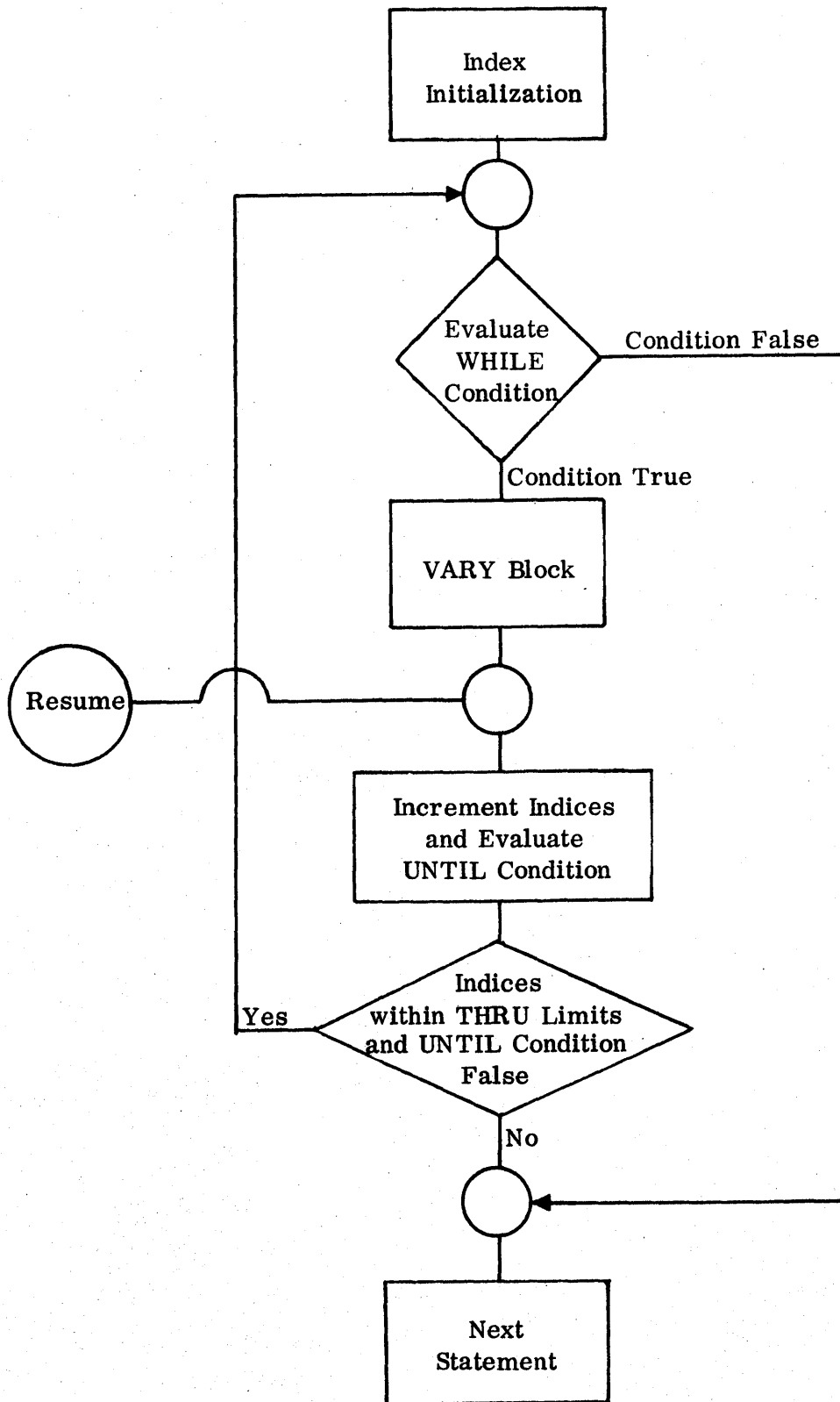


Figure 5-1. VARY Flow

5.7.2.1.1 Index Clause. An index clause specifies a data unit which is to be initialized before the first pass through the vary loop, incremented or decremented after each pass through the loop, and possibly compared against a limit after each pass through the loop in order to terminate the loop.

Format

loop-index initial-value increment final-value

Explanation

Loop-Index	The name of the data unit to be varied.
Initial-Value	Optional. FROM followed by a constant, arithmetic expression, or data unit that specifies the beginning index value of the loop. If omitted, the initial value is 0.
Increment	Optional. BY followed by a constant, arithmetic expression, or data unit. If this value is to be a decrement, BY must be followed by a minus sign. If omitted, the increment is 1. When varying within a table by a negative value, the initial value is the number of items defined for the table or the major index, and the final value is 0.

Final Value

One of the following:

1. May be THRU followed by a constant, arithmetic expression or data unit. This value signifies the last pass through the loop.
2. May be WITHIN followed by the name of a horizontal or vertical table, subtable, or like-table. The value assigned is the number of items defined for the table or the current value of the major index.

The initial-value, increment, and final-value may appear in any order within an index-clause. Refer to Paragraph 5.7.2.4 for examples of index-clauses.

5.7.2.1.2 WHILE Clause. The WHILE clause specifies a condition which is to be tested before each pass. If the WHILE condition is true, the pass will be executed.

Format

WHILE condition

Explanation

WHILE

Specifies a WHILE clause.

condition

A Boolean condition as defined in Paragraph 5.6.1, a validity condition as defined in Paragraph 5.6.3, or a parity condition as defined in Paragraph 5.6.4.

Refer to Paragraph 5.7.2.4 for examples of WHILE clauses.

5.7.2.1.3 UNTIL Clause. The UNTIL clause specifies a condition which is to be tested after each pass through the VARY block. If the UNTIL clause is true, another pass through the VARY block will not be performed.

Format

UNTIL condition

Explanation

UNTIL	Specifies an UNTIL clause.
Condition	A Boolean condition as defined in Paragraph 5.6, a validity condition as defined in Paragraph 5.6.3, or a parity condition as defined in Paragraph 5.6.4.

Refer to Paragraph 5.7.2.4 for examples of UNTIL clauses.

5.7.2.2 Resume (RESUME) Statement

A RESUME statement specifies a transfer to the increment and test steps within a VARY block. This allows partial vary passes through a VARY block and a means to continue the VARY block. A RESUME statement may also be used in conjunction with a FIND statement, since FIND operations permit VARY operands.

Format

RESUME statement-name \$

Explanation

RESUME	Specifies the RESUME operator.
Statement Name	The name given to the VARY or FIND statement to which this RESUME statement applies.

5.7.2.3 End Vary Statement (END)

The END vary statement specifies the conclusion of the vary block. If the loop is not yet completed, any loop indices are incremented and control is transferred to the initial statement within the vary block. If the loop has been completed, control is transferred to the statement following the END vary statement.

Format

END statement-name \$

Explanation

END Specifies the end of a vary block.

Statement Name The name, if any, given to the corresponding VARY statement.

5.7.2.4 Examples of VARY Blocks

1. STEPS. VARY APR FROM 1 THRU 10 \$
SET PAR(AAR, FCX2) TO 'INSERT' \$
END STEPS \$

The data unit APR is varied from starting point 1 (FROM) to the ending point 10 (THRU) by an implied factor of 1. The field FCX2 is set to the value of 'INSERT' in items 1 through 10 of table PAR. When the loop is completed, processing continues with the statement following the END statement.

NOTE

Upon loop termination (after the END statement), the loop index will not necessarily contain the value of the ending point or the ending point plus one.

```
2. STEP 6. VARY DW WITH NEW FC2RT BY 1 $  
          SET FC2RT(DW, 1) TO 0 $  
          END STEP 6 $
```

The data unit DW is varied according to the number of items in table FC2RT or the major index (if any). Each word 1 in all items of table FC2RT will be set to 0.

```
3. ONE. VARY X FROM 39 THRU 0 BY -1 $  
       SET Y TO CATAX1 $  
       TWO. VARY Z THRU CATA $  
           SET NMBR(X, Z) TO Y $  
           SET Y TO Y/2 $  
           END TWO $  
       END ONE $
```

This example illustrates nesting of VARY statements. The vary block TWO is restarted after the initialization of vary block ONE and after each decrement for vary block ONE. The complete loop is terminated after X becomes 0.

When nesting vary blocks, the END vary block statements must be arranged in the reverse order of the vary block definitions.

```
1. AA. VARY ..... $  
   BB. VARY ..... $  
      .  
      .  
      .  
      END BB $  
   CC. VARY ..... $  
      .  
      .  
      .  
      END CC $  
      END AA $
```

More than one complete vary block can be nested within another vary block.

```
5. STEP 7. VARY CAT A THRU 20, CAT B THRU 30  
          BY 4 $  
          SET PAR(CAT A, CTRL) TO CAT B $  
          END STEP 7 $
```

More than one data unit may be varied within a loop. The data unit CATB is varied from the starting point 0 (implied) to the ending point 30 (THRU) by a factor of 4 (BY). The data unit CATA is varied from the starting point 0 (implied) to the ending point 20 by an implied factor of 1. The field CTRL is set to the value of CATB in items 0 through 7 of table PAR.

NOTE

If more than one data unit is to be varied, the data unit that first satisfies its ending point terminates the loop.

M-5035
Change 4

```
6. CARR. VARY K WITHIN CIBR BY -1 $  
    IF CIBR(K,3) NOT 0 THEN  
    RESUME CARR $  
    SET CIBR(K,3) TO RACKS $  
    END CARR $
```

When CIBR(K,3) is nonzero, the statement following the IF is not executed. The transfer to make the next pass through the vary block is accomplished by the RESUME statement.

```
GTER. VARY A FROM 1 THRU 50  
    SET X TO Y + 1 $  
    IF X GT 25 THEN GOTO LATR $  
    END GTER $  
    .  
    .  
    .  
LATR. SET Y TO Y/2 $  
    RESUME GTER $
```

This example illustrates the transfer outside of a vary block. When X is greater than 25, control is transferred to step LATR outside of this vary block. The RESUME statement will transfer control back into the vary block.

```
8.  VARY I FROM I WHILE TAB(I) VALID $  
    SET TAB(I) TO TAB(I)+5 $  
    VARY I FROM I WITHIN TAB $  
    SET TAB(I) TO TAB(I)+5 $  
    END $
```

This example illustrates two methods of incrementing a set of table items not necessarily starting with the first item. The difference between the two methods is the location of the test to determine if another pass is to be performed. The VARY using the WHILE clause will test before each pass, while the VARY using the WITHIN will test after each pass. The WHILE VARY would be used for situations in which no passes through the loop were desired because I does not contain a valid TAB index.

```
9.  VARY I UNTIL SYMBOL NOT H(,) $  
    SCAN $  
    SET NAME(I) TO SYMBOL $  
    SCAN $  
    END $
```

The above example illustrates the use of an UNTIL clause to process a comma separated list, storing each list item in a name table whose index is also initialized and incremented by the VARY block. Assuming SCAN updates the value of the Hollerith variable SYMBOL, a test is made at the end of each pass to determine if another list item is specified.

```
10. VARY I, J THRU 5 WHILE B1 UNTIL B2 $  
    TABPROC INPUT TIN(I) OUTPUT TOUT(1), B1, B2 $  
    END $
```

The above examples illustrate a loop which may be terminated by any of the three loop termination criteria: index, WHILE, or UNTIL. No passes will be made through the block if the Boolean variable B1 is false when first entering the block. TABPROC will set the first six items of table TOUT assuming neither Boolean variable B1 or B2 is set to false due to some condition diagnosed by TABPROC.

```
11. VARY I $  
    :  
    :  
    :  
    END $  
  
VARY $  
    :  
    :  
    :  
    END $
```

The above examples illustrate two methods of programming "infinite" loops. The only difference between the loops is that in the first example, I will maintain a count of how many times the loop has been executed.

5.7.3 FOR Block

A FOR block consists of a FOR statement followed by a set of value blocks and an END statement which terminates the FOR block. Execution of a FOR block will result in the execution of the value block one of whose values is the same as the result of the FOR expression contained in the FOR statement.

Format

```
statement-label. FOR-statement $  
    value-block  
    :  
    value-block  
END statement-name $
```

Explanation

Statement-label	Optional. The name by which the FOR block is identified.
FOR-statement	Refer to Paragraph 5.7.3.1.
Value-block	Refer to Paragraph 5.7.3.2.
END	Specifies the end of the FOR block.
Statement-name	The name, if any, given to the FOR block as specified in the FOR statement.

Examples

Refer to Paragraph 5.7.3.3.

5.7.3.1 FOR Statement

The FOR statement specifies the controlling expression, type, and optional ELSE statement of a FOR block.

Format

```
FOR FOR-expression FOR-type ELSE-statement $
```

Explanation

FOR-expression

Any arithmetic, Boolean, relational, literal, or status expression legal as an assignment expression as defined in Paragraphs 5.4.1.1 through 5.4.1.4.

FOR-type

Optional. The presence of an explicit FOR-type is denoted by a comma following the FOR-expression. The comma is followed by a data type contained in parenthesis. The data type may be any type allowed in a variable declaration with the restriction of a maximum of eight characters for a Hollerith type. If the FOR-expression is a variable, field reference, typed item reference, or function reference and the FOR-type is not explicitly specified, the FOR block will have the type of the FOR-expression data unit. If the FOR-expression is a local or system index and the FOR-type is not explicitly specified, the FOR block will have a type of I 16 U. If the FOR-expression is an item word reference and the FOR-type is not explicitly specified, the FOR block will have a type of I 32 S. If the FOR-expression is not one of the above data units the FOR-type must be explicitly specified. If the FOR-type is explicitly specified, the FOR-expression must agree in type with the FOR-type

according to rules of assignment specified in Paragraphs 5.4.1.1 through 5.4.1.4. Additionally, if the explicit FOR-type is Hollerith, the FOR-type may not specify a greater number of characters than contained in the FOR-expression.

ELSE-statement Optional. Refer to Paragraph 5.6.5.

Execution of a FOR statement will cause the evaluation of the FOR-expression, the required conversion, if any, to the FOR block type, and execution of the value block having the same value as the FOR-expression. If the value of the FOR-expression does not match any of the FOR block values and an ELSE-statement is not specified, the control will be transferred to the statement following the FOR block. If the value of the FOR-expression does not match any of the FOR block values and an ELSE-statement is specified, the ELSE-statement will be executed. After execution of the selected value block or the ELSE-statement, control will be transferred to the statement following the FOR block.

Execution of a FOR statement is subject to the following rules:

1. If the FOR-expression is an arithmetic expression, it will be evaluated with "simulated receptacle" rules defined on Page II-5-4.
2. The result of a FOR-expression will be converted to the FOR block type as if the FOR-expression were being assigned to a variable having the FOR block type.

3. To produce correct code, the FOR block type must express all possible values which may be produced by the FOR-expression. For example, if an arithmetic FOR-expression produces a negative value, the FOR block type must be signed. This rule does not imply that a value block must be specified for each possible value which may be produced by the FOR-expression.
4. To produce optimal code, the FOR block type should express only those values which may be produced by the FOR-expression. For example, if an arithmetic FOR-expression will always produce only the integers between 0 and 7, the FOR block should be typed I 3 U.

5.7.3.2 Value Block

A value block is a group of statements which is executed when the evaluation of the associated FOR-expression results in one of the constant values associated with the value block.

Format

```
Statement-label.      BEGIN value(s) $  
                      :  
                      :  
                      dynamic statements  
                      :  
                      :  
                      END statement name $
```

Explanation

Statement-label Optional. The name by which the value block is identified.

BEGIN	Specifies the start of a value block (when followed by constants).
Value(s)	Constants that are associated with the value block. Multiple values are separated by commas. The constant must agree in type with the FOR block type. If the constant is Hollerith, it will be blank filled to the right to the size of the FOR block type.
Statement-name	The name, if any, given to the value block as specified on the BEGIN statement.

5.7.3.3 FOR Block Examples

```
1.      FOR X $
VA.    BEGIN 0, 7 $
        CASEA INPUT H($) $
        END VA $
VB.    BEGIN 4 $
        CASEB INPUT H(,) $
        END VB $
VC.    BEGIN 1,2,3 $
        SET ERCOD TO 16 $
        END VC $
      END $
```

Assuming X is a local-index, the appropriate value block will be executed if X has the value 0, 1, 2, 3, 4, or 7. If X is 5, 6 or greater than 7, the statement following the FOR block will be executed. If it is known that at the time of execution of the FOR statement, X will never be greater than 7, the FOR statement should be coded:

```
FOR X, (I 3 U) $
```

M-5035
Change 4

```
2.  FBLOCA.      FOR F(X)+2, (IΔ4ΔS)
                        BEGIN -7,-5, 7 $
VBLOC.      VARY UNTIL X GTEQ 0 $
                        SET X TO X+F(X) $
                        END VBLOC $
                        SET MES TO H( ) $
                        END $
                        BEGIN -1, 0, 1 $
FBLOCB.     FOR H(Y)
                        ELSE
                        SET MES TO H(*****) THEN
                        SET X TO 1 $
                        BEGIN H(T1) $
                        SET MSG TO H(AA) $
                        SET X TO 5 $
                        END $
                        BEGIN H(D1),H(T2) $
                        SET MES TO H(??) $
                        SET X TO 7 $
                        END $
                        BEGIN H(D1A) $
                        SET MES TO H(ERROR) $
                        SET X TO 0 $
                        END $
                        END FBLOCB $
                        END $
                        END FBLOCA $
```

The value block containing VBLOC will be executed if the arithmetic expression $F(X)+2$ results in -7, -5, or 7. The value block containing

FBLOCB will be executed if the expressions results in -1, 0, or 1. If none of these values result, the control will be transferred to the statement following block FBLOCA. FBLOCB is a FOR block whose FOR-expression is an H 4 type function. If the function returns the value H(T1 $\Delta\Delta$), H(D1 $\Delta\Delta$), H(T2 $\Delta\Delta$), or H(D1A Δ), X and MES will be set by the appropriate value blocks. Otherwise, X and MES will be set by the execution of the FOR ELSE clause.

```
3.          FOR T(J)
            ELSE BEGIN $
              VARY UNTIL T(J) EQ 'FREE' $
              END $
              SET T(J) TO 'BUSY' $
              END $
            BEGIN 'I01' $
              VARY UNTIL T(J+1) EQ 'FREE' $
              END $
            END $
            BEGIN 'I02' $
              VARY UNTIL T(J+2) EQ 'FREE' $
              END $
            END $
            BEGIN 'FREE' $
              END $
            END $
```

Example 3 demonstrates a possible usage of a FOR block used to control wait loops in a multiprocessing environment where the processors communicate the system status through a table with status typed items. Note that the value block whose value is 'FREE' contains no dynamic statements.

SECTION 6

INPUT/OUTPUT STATEMENTS

The CMS-2 language includes a number of statements that enable the user's program to communicate with peripheral equipment. The operators for these statements are summarized below:

<u>General Operators</u>	<u>Special File Handling Operators</u>
INPUT	FILE
OUTPUT	OPEN
FORMAT	ENDFILE
ENCODE	CLOSE
DECODE	POS
	FIL
	LENGTH
	DEFID
	CHECKID

To use these statements successfully, the following restriction must be observed: The CMS-2 Monitor must be in core during user-program execution. This requirement arises because of Compiler-generated procedure calls to input/output run-time routines, which are designed to link with the Monitor and communicate with its input/output drivers.

6.1 INPUT/OUTPUT OPERATIONS

For many input/output purposes, a user's program requirements can be most simply met with only the INPUT, OUTPUT and FORMAT statements. In these general situations, the CMS-2 operating system provides system-defined names by which the user may reference the hardware device involved with the input or output function. These names and their associated devices, hereafter referred to as standard files, are described in Table 6-1.

TABLE 6-1. CMS-2 OPERATING SYSTEM STANDARD FILES

FILE NAME	FILE DEVICE
READ	Card reader
PRINT	Printer
PUNCH	Card punch
OCM	Operator communication medium

For those cases in which the user wishes to perform input/output operations using devices other than standard devices, special file-handling operators are provided. These operators are discussed in paragraph 6.3 (Nonstandard File Control).

The INPUT and OUTPUT commands are used to transmit data between a hardware device and a user's program. These commands are functionally illustrated in Figure 6-1. Each input or output statement causes the Compiler to generate calls to particular run-time routines. These routines interface with the Monitor to cause immediate transfer of data between a buffer area and the hardware device. For standard files, the run-time routines provide the buffer area.

If formatting (conversion of data from one representation form to another) is required, transformation routines that utilize the specifications of the FORMAT statement as their inputs are provided. These transformations occur in the input/output flow as illustrated in Figure 6-1. Prior to each input or output operation, the buffer area associated with that file name is preset to blanks if the data is to be formatted, and to zeros if the data does not have formatting specified.

For standard input/output operations where no format statement is prescribed, the input card image is interpreted as Hollerith characters and stored with the internal octal representation of the individual character. During output, the word contents are interpreted in 8-bit Hollerith characters and are printed.

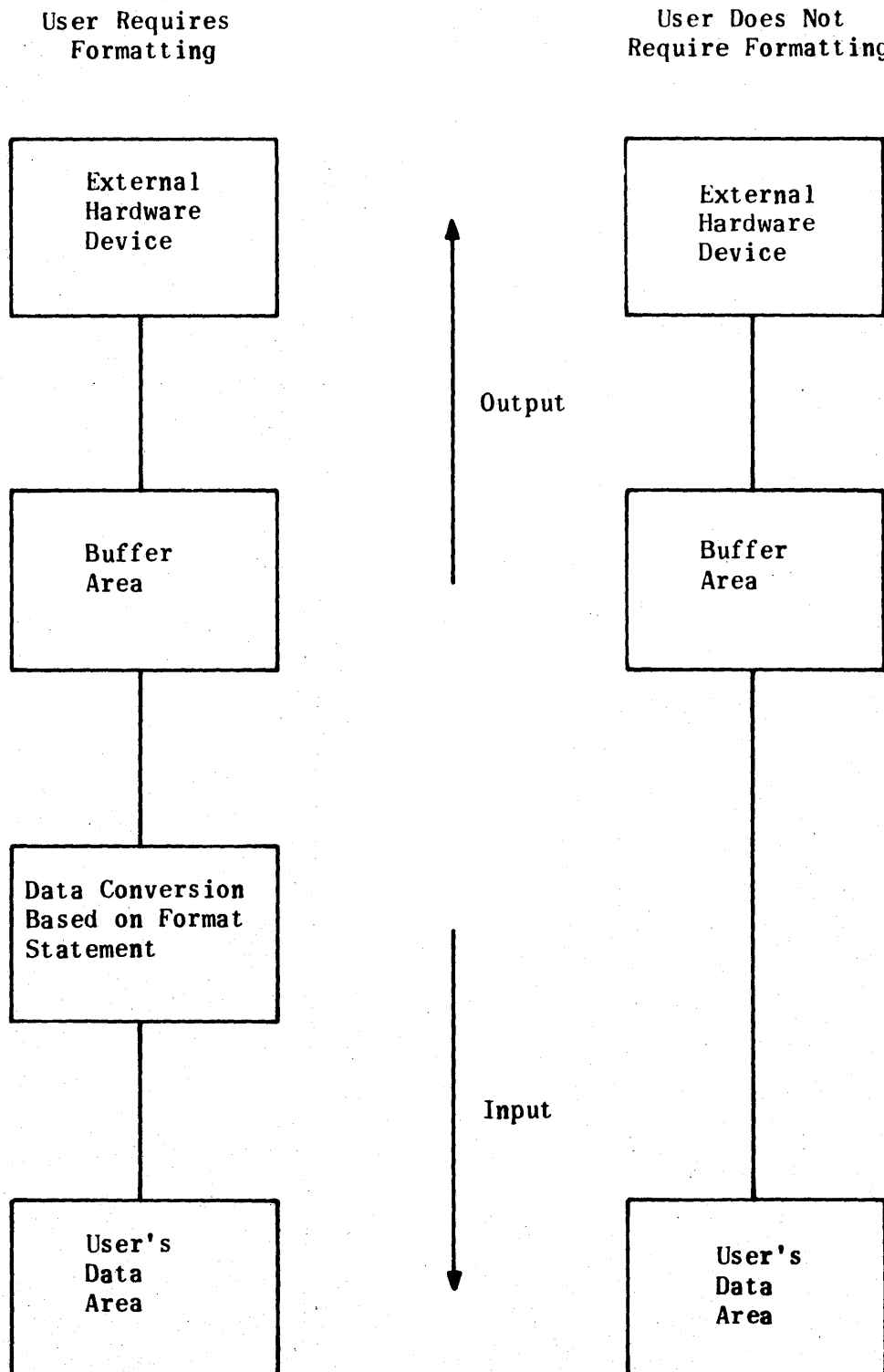


Figure 6-1. Input/Output Data Flow

6.1.1 INPUT Statement

This statement directs an operation to input data from the device associated with the specified file name. When this command is used to transfer data into an entire table by means of whole-table referencing, the data is moved sequentially word-by-word without regard to the defined table structure; however, when all or part of a table is specified by means of item referencing, the data transfer is performed by items. If a FORMAT declaration is referenced, automatic conversion occurs.

Format

INPUT file-name receptacle-data-list format-name \$

Explanation

INPUT	Specifies an INPUT operation.
File Name	The name of a standard file or the name of a non-standard file that has been defined with a FILE declaration statement. Legal standard file names for input are: <ol style="list-style-type: none"> 1. OCM - specifies the operator communication medium. 2. READ - specifies the operating system's input device for punched card images.
Receptacle Data List	One or more of the following CMS-2 data structures. If more than one receptacle is to be specified, the names, separated by commas, are gathered into a list which is enclosed in parentheses. <ol style="list-style-type: none"> 1. The name of a table, subtable, like-table or item-area. This name indicates that the entire structure is to be used. 2. The name of a major index of a table, subtable or like-table. 3. The name of a variable.

4. A specific item reference of a table, subtable, or like-table, which may be indexed by a variable or a numeric parameter. This indicates that every word of this item is to be used.
5. A field or fields of some specific item within a table, subtable, like-table or item-area. Multiple field references can be made to any item specified; these references must be separated by commas.
6. A series of items of a table, like-table or subtable that may be modified by field reference(s). This modification is accomplished by bracketing the starting item number in parentheses followed by three periods followed by the ending item number and bracketed in parentheses, such as:

HIGHT ((A)...(B), M1, M2)

where M1 and M2 are fields within the table.

7. A series of items in an array:

ARRAY ((K,L,M)...(N,P,Q))

Every word in the referenced items is filled.

Format Name

Optional. Refers to the name of a previously defined FORMAT declaration. If a format name is specified, the data units in the receptacle data list must have internal data attributes compatible with the external conversion format type.

Examples

1. TABLE CAB H 9 10 \$
INPUT LBR CAB \$

In this example, one record is read from the device described by the FILE declaration LBR and is transferred to fill each word of the table CAB.

2. TABLE DICT H 3 10 \$
INPUT IMAGE DICT(2) \$

One record is read from the device described by the FILE declaration IMAGE and is transferred, filling each word of item 2 in table DICT.

3. FORMAT FORMA 3I5, 0(3) \$
FILE IMP H 10 R 20 MT2 \$
TABLE MAJOR V (I 32 S) 20 \$
SUB-TABLE MINOR 0 10 \$
END-TABLE MAJOR \$
INPUT IMP MINOR((0)···(3)) FORMA \$

One record is read from the device MT2; assume that it is the Hollerith string:

△ △ 350 △ △ 201 △ △ △ △ 1777 △ △ (20 characters)

When the format FORMA is applied to this string, the results in memory are:

MINOR (0,0) 350₁₀
 MINOR (1,0) 201₁₀
 MINOR (2,0) 1₁₀
 MINOR (3,0) 777₈

6.1.2 OUTPUT Statement

The OUTPUT statement directs an operation to output data to the device associated with the file name. Data transferred from an entire table by an OUTPUT command using whole-table referencing is moved sequentially word-by-word without regard to the defined table structure. However, when all or part of a table is specified using item referencing, the data transfer is performed by items. If a FORMAT declaration is referenced, automatic conversion takes place.

Format

OUTPUT file-name source-data-list format-name \$

Explanation

OUTPUT	Specifies an OUTPUT operation.
File Name	The name of a FILE declaration or a standard file. Legal standard files for OUTPUT are: OCM Operator communication medium. PRINT Systems output for printer listing. PUNCH Systems output for card punching.
Source Data List	A constant or one or more of the operands as described for the INPUT statement. If the FORMAT declaration describes a Hollerith string only, the source data parameter is not required. If more than one data unit is to be acted upon, the names are gathered into a list, separated by commas and enclosed in parentheses.
Format Name	Optional. An operand referring to a previously defined FORMAT declaration. If a format name is specified, the data units in the source data list must have internal data attributes compatible with the external conversion format type.

Examples

1. OUTPUT PRINT DICT TERRY \$

The data to be written on the device identified in the file named PRINT is contained in the variable DICT. The optional operand TERRY, refers to a previously defined FORMAT declaration.

2. OUTPUT DATUM (POS, SPEED, ID) TRUNC \$

The data contained in the tables POS, SPEED, and ID is to be written on the device identified in the file named DATUM. The optional operator TRUNC refers to a previously defined FORMAT declaration.

3. In reference to the data structure of Example 3, paragraph 6.1.1, assume that table MAJOR is the result of the INPUT command.

OUTPUT IMP MINOR(0,0).....(3,0) FORM \$

The values in MINOR (0,0) through MINOR (3,0) are transformed to the Hollerith character string:

△△350△△201△△△△1777△△ (20 characters)

which is then transferred as one record to the hardware device MT2.

6.1.3 FORMAT Declaration

The FORMAT declaration describes the conversion of data between internal and external forms. The external form is usually a Hollerith string containing, in addition to the data, certain spacing and control information. Data existing in this external form is referred to as formatted data and is transmitted to and from an external device in this form.

When the user requires data conversion, he references the FORMAT declarative's name in the INPUT or OUTPUT command. The FORMAT declaration is always referenced by the ENCODE and DECODE statements when they are utilized in a program. FORMAT declarations may appear only in data designs.

Format

FORMAT name $Q_1, Q_2, Q_3, \dots, Q_n$ \$

Explanation

FORMAT Specifies the FORMAT DECLARATION.

Name The identifier to be used to reference this FORMAT declaration.

Q_i A format descriptor. The format descriptors indicate the form and arrangement of data and the types of conversions to be performed.

Numeric conversion types are summarized below.

<u>Internal Form</u>	<u>Format Descriptor</u>	<u>External Form</u>
Fixed-point binary	Iw.d	Fixed-point decimal
Floating-point binary	Fw.d	Fixed-point decimal
Floating-point binary	Ew.d	Floating-point decimal
Fixed-point binary	Ow.d	Fixed-point octal

In the list of format descriptors that follows, w is an unsigned integer representing the maximum width (number of characters) of the field in the external medium. Integer w must not exceed the number of characters in one printer line. Descriptor d is an unsigned integer representing the number of characters in the field that appears to the right of the binary or decimal point. The maximum width of the field w must include space for signs, radix points, and exponent descriptions. Hence, field d must be less than w. The d-field is optional if the internal fixed-point binary value is an integer.

Format Descriptor

Function

Iw.d

Specifies conversion of data between internal fixed-point binary and an external fixed-point decimal Hollerith character string. For a positive value, $w \geq d+2$. For a negative value, $w \geq d+3$.

Fw.d

Specifies conversion of data between internal floating-point binary and external fixed-point decimal. For a positive value, $w \geq d+2$. For a negative value $w \geq d+3$.

Ew.d

Specifies conversion of data between internal floating-point binary and external floating-point decimal.

The acceptable forms of input fields for the E conversion (floating-point) are:

+0. mantissa E

+0. mantissa E + exponent

The mantissa may be of any magnitude; the allowable exponent range depends upon the object machine. The output form for E conversion is:

$$\pm 0.\text{xxx}\dots\text{xxxE}\pm\text{ee}$$

where:

d is the number of digits in the mantissa. w includes all characters. Thus, $w \geq 7$, where the 7 accounts for the specific characters $\pm 0.E\pm\text{ee}$ of the above output form.

Ow.d

Specifies the conversion of data between internal fixed-point binary and external fixed-point octal. For a positive value, w must be ≥ 2 . For a negative value, w must be ≥ 3 .

Other format descriptors that may appear are described below.

Format Descriptor

Function

H (string of characters)

Specifies an alphanumeric field in the form H(ABC), for example, where ABC represents a string. For input, an H (string) specification causes n characters to be skipped in the input record. For output, the string of characters specified within the parentheses is the output image.

Aw

Specifies the first w characters of an alphanumeric data unit in a transfer to or from an input/output buffer. Aw appears in the FORMAT declaration. The related

Format DescriptorFunction

	data unit in the input/output reference can have more than w characters. The remaining characters are ignored. If the data unit has less than w characters, the rightmost characters are truncated on input and trailing spaces are inserted on output.
Lw	Specifies the same as Aw, only here the last w characters of the data unit are used. Everything else said under Aw applies here.
wX	Specifies skip w characters of an input record or insert w spaces in an output record.
Tw	Specifies a position designator for the buffer of a record input/output file. w indicates the character position within the buffer. The count starts with 0 at the start of the buffer. The FORMAT statement can have many T's, one for every data word or constant. T is illegal for files having a length descriptor of S.
n format descriptor	Specifies repetition of a format descriptor n times, such as n Ew.d. Slash specifies end of a record or, when used sequentially, indicates the number of records to be skipped or inserted: n+1 consecutive slashes on input cause n records to be skipped. n+1 consecutive slashes on output cause n blank records to be produced.

Format Descriptor

m(group of descriptors)

Printer carriage
control charactersFormat

Specifies the repetition of a group of format descriptors within the parentheses m times. No other parentheses except for H descriptors are allowed within the group of descriptors.

Appear in an H string. They specify spacing, page eject, etc. These control characters appear as the first Hollerith character in the first word of each record. If the character is not one of the following, it is replaced with a blank.

Carriage-Control
CharacterOperation

blank	Single space and print line.
0	Double space and print line.
-	Triple space and print line.
1	Page eject and print line.
H	Cancel headers, page eject and print line.
A	Cancel headers, page eject, print line and save location and length of line as a major header.
B	Cancel lower-level headers, double space, print line and save

<u>Carriage-Control Character</u>	<u>Operation</u>
	location and length of line as a minor header.
C	Cancel lower-level headers, single space, print line and save location and length of line as a minor header.

If classification, major header, and/or minor header lines are wanted, they will be printed in the following order each time a page eject is necessary: classification, major header with page number, minor header B, minor header C. The page number will occur even if a major header is not used. It is not necessary to specify consecutive levels of minor header information.

It is the programmer's responsibility to ensure that the specified type of format conversion is compatible with the declared data unit mode (for example, F-type conversion should not be applied to integer data units). If the FORMAT descriptors are exhausted and the target list not yet filled, the FORMAT is restarted.

Examples

- Given the external string of characters
350274-0162E+050703 with format

FORMAT CPRE F2.0, F1.1, E9.2, 0A.0, \$

the quantities stored on a read or decode statements are:
35, 27.4, -1.62×10^5 , 703

2. Given the internal quantities
417, -320, 0.536×10^3 and octal 627
with format

```
FORMAT D|R,A,B, H(1), I3.0, F6.2, E10.3, 05.0, $
```

the string of characters resulting from an OUTPUT or ENCODE statement is:

```
1417*****+0.536E+03A627
```

where the asterisks indicate that the value -320 cannot be encoded within an F6.2 format descriptor.

3. Given the internal quantities
27, H(XYZ), 74.51, H(JKLM)
with the format

```
FORMAT H|A,W, F6.2, H(RAG), L2,  
F6.2, H(MODP), A2, $
```

the string of characters resulting from an OUTPUT or ENCODE statement is:

```
27.0ORAGYZ 74.51MODPJK
```

6.2 ENCODE AND DECODE OPERATIONS

The ENCODE and DECODE statements direct run-time routines to transfer data internally from one area of the computer to another, while converting the data from nonformatted to formatted (encoding) or vice versa (decoding). The transformation is specified by a FORMAT declaration.

The ENCODE and DECODE statements are analogous to INPUT and OUTPUT statements that reference a FORMAT declarative, except that no transmission to an I/O device takes place, and the buffer area involved is a data unit specified by the programmer. The rules previously discussed regarding formatted I/O also apply to the decode/encode operations.

When inputting records of different types, a DECODE statement can be used to reformat the data after the record type has been examined. The ENCODE statement can be used to format data that is to be modified before being output. These statements are also valuable for debugging by simulating input/output operations in memory.

The ENCODE statement specifies that the data contained in the data units named in the source list is to be converted as specified in the named format declaration and packed into a character string identified by the formatted data-unit name.

The DECODE statement specifies that the character string identified by the formatted data-unit name is to be converted as specified in the named format declaration and placed in the data units named in the target list.

Format

ENCODE formatted-data-target unformatted-source-list
format-name \$

DECODE formatted-data-source unformatted-target-list
format-name \$

Explanation

ENCODE	Specifies the encoding operation of converting data from unformatted to formatted form.
DECODE	Specifies the decoding operation of converting data from formatted to unformatted form.
Formatted Data Target	The data-unit name that will receive the formatted character string (starting in character position 0). This identifier may refer to a single word or multiword data unit with Hollerith attributes.

- Formatted Data Source** The data-unit name containing the character string (starting in character position 0) to be converted to unformatted data. This identifier may refer to a single word or a multiword data unit with Hollerith attributes.
- Unformatted Source List** One or more data units containing the data to be converted to character string form. If more than one data unit are specified, they are collected together, separated by commas, and enclosed in parentheses. These data units must have internal data type attributes compatible with the external conversion format type.
- Unformatted Target List** One or more data units that will receive the unformatted data after conversion. If more than one data unit is specified, they are collected together, separated by commas, and enclosed in parentheses. These data units must have internal data type attributes compatible with the external conversion format type.
- Format Name** The identifier of the format.

Example

```

LOC-DD $
VRBL A1 I 12 U P 223 $
VRBL A2 H 6 P H(GOBAC) $
VRBL A3 F P 12.14 $
VRBL JOE H 13 $
FORMAT BILL I3, L5, F5.2 $
END-LOC-DD $
PROCEDURE ALPHA $
.
.
ENCODE JOE (A1, A2, A3) BILL $
.
.
END-PROC ALPHA $

```

In this example, the character string JOE is to be packed with the value of VRBLs A1, A2 and A3 as indicated by the format BILL. The result will be

JOE: 223GOBAC12.14

6.3. NONSTANDARD FILE CONTROL

The special file operators presented in the remainder of this section are not to be used with the standard file names listed in Table 6-1.

In addition to the CMS-2 operating system standard file names and their associated devices, each operating system provides other devices for the user. These devices consist principally of magnetic tape and paper tape punch units.

The principal difference in using nonstandard devices versus standard devices is the requirement for the user to describe the physical characteristics of the data to be transferred and to state the name of the device on which input/output operations are to occur. These attributes are collected in a formal declarative called a FILE declaration and identified by a user-supplied file name. This file name has significance only within the user's program. It provides the linkage between the attributes described in the file declarative statement and the run-time routines that interface with the Monitor to achieve satisfactory input/output results.

The following definitions are necessary before proceeding with the discussion of nonstandard file control.

Logical record

A group of adjacent, related data items such as the individual entries in a telephone book for name, address and telephone number. This organization has significance only to the user and is not necessarily a physically definable entity.

Physical record

A group of adjacent, related logical records defined as a unit such as the alphabetic groupings of telephone book entries. On magnetic tape, a physical record is delimited at the beginning and end by inter-record gaps. These gaps are recognizable by the hardware device. A physical record may contain one (unblocked) or more than one (blocked) logical records. Blocking is the technique used to reduce the wasted space of inter-record gaps between unblocked logical records.

Physical file

A group of adjacent, related, physical records defined as a unit such as the complete telephone book for one city. On magnetic tape, a physical file is delimited at the beginning and end by an end-of-file mark (for the first physical file of a tape, its beginning is signified by a beginning-of-tape mark). These marks are recognizable by the hardware handling device. Thus, the physical file comprises one or more physical records.

Peripheral file device	That mechanical device which is the repository for a group of adjacent, not necessarily related, physical files. A file device may contain one or more physical files as space permits.
File-name	A unique, user-assigned name to provide referencing from the dynamic input/output command and control statements to the run-time routines.

6.3.1 FILE Declaration

This declaration is mandatory for all input/output functions involving non-standard file devices.

The FILE declaration defines the environment in which one or more physical files are to be processed. The declaration assigns a file name for dynamic statement referencing, identifies the symbolic name assigned to the actual hardware device, and declares that all data to be processed on the named hardware device is physically organized as described in the declarative statement.

The FILE declaration indicates the hardware device and reserves two areas. The first area contains the record description, the file status variable, device address, space for keeping track of file and record positions, and in case of a file having the length descriptor S, the position pointers. The second area reserved is a buffer, equal in size to the maximum record that may be transmitted. For record input/output the buffer area does not accumulate data for blocking purposes, so that each INPUT or OUTPUT statement causes immediate communication with the device. If the file declaration contains the length descriptor S, data is accumulated in a fixed size stream buffer and output is under control of run-time routines.

The number of words transferred between the buffer and the data area depends upon the number of words specified in the data list of the input or output statement rather than the record length. For example, if 50 words of input are requested by specifying a 50-word table in an input statement and if the input record is only 10 words long, the first 10 words of the input record are

assigned to the first 10 words of the table, and the remainder of the table is set to blanks or 0 depending on the record type. If 50 words of input are requested and the input record is 100 words long, the first 50 words of the input record would be transferred and the remaining 50 words would be lost.

The INPUT and OUTPUT statements, when referencing a FORMAT declarative, imply either an automatic decode from the FILE declaration buffer area or an automatic encode into the FILE declaration buffer area.

The FILE declaration must occur in one of the user's data design areas. A hardware device may be referenced in more than one FILE declaration. Standard hardware units have implied FILE declarations; the user is not required to declare them.

Format

```
FILE file-name type maximum-no.-of-records
      length-descriptor maximum-record-size
      hardware-name states WITHLBL $
```

Explanation

FILE	Specifies a FILE declarative.
File Name	An identifier to be used to reference this information.
Type	One of the following: H - records are all Hollerith. B - records are all binary.
Maximum Number of Records	An integer or tag that specifies the maximum number of records that may be accessed in a physical file. If 0 is specified, any number of records may be accessed.
Length Descriptor	One of the following: R - the size of each record is equal to the maximum length as defined by the record size (rigid length).

V - The size of each record is determined by the amount of data transferred to and from the file buffer and is not to exceed the maximum length as defined by the record size (variable length).

S - The size of each record is rigid. Data is accumulated in the buffer gradually.

Maximum Record Size

An integer or tag that specifies the maximum length of the record (meaning number of words if the type is B and number of characters if the type is H). If 0 is specified, the size of the operand used in the particular input/output statement determines the record length. In this case, no buffer area is set aside, and data is transferred directly between the data area and the input/output device. Zero is illegal with a length descriptor of S.

Hardware Name

The name of any external storage device in a computer's environment. Run-time routines will provide direct interface with peripheral equipment (printer, punch, magnetic tapes, etc.).

Examples of hardware names, which may vary for each installation:

MT2 Magnetic tape input and output units.
TTY Teletype.

States

An optional parameter that defines one or more states, which may be tested between input/output operations on this device. Each state is a mnemonic enclosed in single primes and corresponds to one of the numeric values returned by the Monitor after each I/O operation. (See paragraph 2.3.1.4 Volume I). Testing one or more of these states may be necessary to avoid an input/output abort (see paragraph 6.4 for format to test states).

WITHLBL

Optional. Indicates that this file has or will have a label for identification as its first entry preceding the first record.

Examples

1. FILE LBR H 3200 R 120 MT2 'BUSY',
'NORMAL', 'EOF', 'ERR' \$

The file LBR has a maximum of 3200 Hollerith-type records, whose sizes are rigid (120 characters). The external storage device named is MT2, whose possible states are 'BUSY', 'NORMAL', 'EOF', and 'ERR'.

2. FILE OBJC B 700 N 480 MTH 'ASY', 'WRM',
'EOF', WITHLBL \$

The file OBJC is a binary file containing up to 700 records with the name OBJC as the first record. The record size is variable and may have a maximum of 480 words. If the user tests an 'EOF' hardware state, Monitor hardware state-2 will be tested.

6.3.2 OPEN Statement

Before data can be transmitted between a user's program and an external non-standard device by means of the INPUT or OUTPUT statements, the run-time routines must be informed that any reference using the FILE declaration name is legitimate and expected. The OPEN command instructs the hardware device specified by the FILE declaration to be accessible for input, output or both. Execution of an OPEN statement does not establish or guarantee the physical readiness of the hardware device. In addition, a file that is open may not be reopened until it has first been closed. That is, if a file is to be changed from an input unit to a scratch unit, it must be closed and reopened as a scratch unit.

Format

```
OPEN file-name action $
```

Explanation

OPEN Specifies an OPEN operation.

File Name The identifier of the file.

Action One of the following:

INPUT

OUTPUT

SCRATCH (both input and output allowed)

Example

```
OPEN LBR INPUT $
```

This statement causes the file identified as LBR to be activated and specifies use as INPUT only.

6.3.3 ENDFILE Statement

If a user wishes to group records together, he may form a physical file. A physical file of data is any set of sequential physical records delimited by

an end-of-file mark. These groupings may be used for device positioning; a device may contain any number of these marks.

The ENDFILE statement is used to place an end-of-file mark on those hardware devices to which it is applicable. Writing an end-of-file automatically sets to 0 the record count associated with the file name of the file declarative. There is no change in the physical position of the hardware device as a result of this command.

Format

ENDFILE file-name \$

Explanation

ENDFILE Produces an end-of-file mark on a hardware device.

File Name The name of a previously opened nonstandard file. The parameter OUTPUT or SCRATCH must be included in the OPEN operation for this file name.

Example

```
FILE LBR B 300 B 3000 MT2 'BUSY', 'NRM' $
.
.
.
OPEN LBR SCRATCH $
.
.
.
ENDFILE LBR $
```

The file declaration LBR, specifying hardware device MT2, is opened for both input and output. The ENDFILE operation causes an end-of-

The states may be tested by use of the IF statement containing the file name and this unique mnemonic. The test checks equality between the current status of the file and the value assigned to the unique mnemonic. All runtime I/O operations are completed before returning to user's program.

It should be noted that this test is dependent upon the hardware and Monitor system. The list of states appearing in the FILE declaration must correspond to the numeric status values returned by the Monitor (see Volume I).

Format

IF file-name EQ or NOT state THEN expression \$

Explanation

- | | |
|------------|--|
| IF | Specifies an IF statement. |
| File Name | The name of a FILE declaration. |
| EQ or NOT | Specified relational operators. |
| State | Any mnemonic designated in the FILE declaration. It must be enclosed in single primes. |
| THEN | A CMS-2 connector word. |
| Expression | Any CMS-2 dynamic statement. |

Examples

1. IF LBR EQ 'NRM' THEN GOTO ENJOB \$

The status variable of the FILE LBR is tested.

If the condition indicated by the mnemonic 'NRM' is true, a transfer to ENJOB is executed.

```
2. FILE INPTC H 500 R 120 MTS 'BSY' 'NRIM'
      'EOF' 'ERRS' $
      .
      .
      OPEN INPTC INPUT $
      INPUT INPTC MINOR $
      IF INPTC NOT 'EOF' THEN GOTO ALPHA $
```

The state for 'EOF' (Monitor hardware status code 2) in the file defined by INPTC is tested. If the condition is true, a transfer to ALPHA is executed.

6.5 DEVICE POSITIONING

In order to retrieve data from a device more efficiently, the programmer may separate his data into one or more files or subfiles. The run-time routines will recognize these files by end-of-file marks and maintain position counters for the number of files passed and a counter for the number of records passed within a given file.

The SET statement in conjunction with the functional modifiers POS (record position), FIL (file position) and LENGTH may be used to physically position a device, determine its present position or determine the length of a record.

Positioning requests are not applicable to all devices and incorrect use of the above modifiers will result in an input/output error indication.

6.5.1 Positioning By Files

A unit may be positioned forward or backward by a number of files or subfiles. The unit is always automatically positioned following the end-of-file mark.

Format

```
SET FIL(name) TO signed-integer-constant or data-unit-name $
```

Explanation

SET	Specifies a SET operation.
FIL(name)	Specifies the positioning (FIL) of the unit specified by the name of a FILE declaration.
TO	CMS-2 separator.
Signed Integer Constant or Data-Unit Name	Specifies the number of files the unit is to be positioned. The sign indicates the direction of the movement: + forward and - backward. If a data name is used, it must contain an integer. A data-unit name containing a 0 will cause the device to be positioned to the beginning, If -0 is specified, the unit is locked out from further reference.

Examples

1. SET FIL(NARFLAG) TO LOC1 \$

If the variable LOC1 contains a 4, the device referenced in file declaration NARFLAG will be positioned forward four file marks. The same statement may be used to backspace four files by setting LOC1 to -4. If a request is given to backspace more files than are written on the device, the device is positioned at the beginning. A request to backspace one file mark causes the tape to be positioned at the start of the current file.

```

2. FILE MTF3 300 R 120 MT13 $
   VRBL FILPOS I 15 U $
   .
   .
   OPEN MTF3 READ $
   .
   .
   ALPHA. SET FIL(MTF3) TO FILPOS $
   .
   .
   BETA. SET FIL(MTF3) TO 0 $
   .
   .
   GAMMA. SET FIL(MTF3) TO -0 $

```

Statement ALPHA causes the hardware device MT13 to be positioned forward as many files as indicated in the variable FILPOS. Statement BETA causes the hardware device MT13 to be set to the beginning position 0. By statement GAMMA, the hardware device is set to the beginning, locked out from further reference, and the FILE MTF3 is closed. Each time, the file counter is adjusted.

6.5.2 Positioning by Records

A unit may be positioned forward or backward a number of records within the current file. Attempted record positioning beyond the bounds of the current file will cause the device to be positioned at the beginning or end of the current file. Record positioning should not be used with stream files.

Format

```
SET POS(name) TO signed-integer-constant or data-unit-name $
```

Explanation

SET	Specifies a SET operation.
POS	Specifies the positioning (POS) of the unit named by a FILE declaration.
TO	CMS-2 separator.
Signed Integer Constant or Data-Unit Name	Specifies the number of records to be forward or backward spaced.

Examples

1. SET POS(BOOK) TO SCREEN \$

In this example, the file named BOOK is spaced forward the number of record positions contained in the variable named SCREEN.

2. FILE MAGFILE2 BDR 330 MT4 'BZ' 'NRM' 'EOF' \$
OPEN MAGFILE2 INPUT \$
SET POS(MAGFILE2) TO -3 \$

The hardware device described as MT4 is positioned three records backwards, relative to its current position.

6.6 FILE AND RECORD POSITION DETERMINATION

The record file or subfile position within the current file can be determined with the use of the POS or FIL modifiers.

Format

SET data-unit-name TO FIL(name) or POS(name) \$

Explanation

SET Specifies a SET operation.

Data Unit Name The location where record position is to be stored.

TO CMS-2 separator.

FIL(name) or POS(name) Specifies the file position (FIL) or record position (POS) of the unit specified by the name of a FILE declaration.

Example

```

FILE LBR H 350 V 200 MT2 $
VRBL LBRFIL I 16 U $
VRBL LBRPOS I 16 U $
.
OPEN LBR READ $
.
SET LBRFIL TO FIL(LBR) $
SET LBRPOS TO POS(LBR) $

```

The current file or subfile position of the hardware device MT2 is stored in the data unit LBRFIL; the current record position within this file is stored in LBRPOS.

6.7 RECORD LENGTH DETERMINATION

The length of the last record transmitted by either an INPUT or an OUTPUT statement may be determined by using the functional modifier, LENGTH.

Format

```
SET data-unit-name TO LENGTH(name) $
```

Explanation

SET Specifies a SET operation.

Data-Unit Name Specifies the location where the record length is to be stored.

TO CMS-2 separator.

LENGTH(name) Specifies the length of the previous record on the unit specified by the name of a FILE declaration.

Example

```

FILE MAGFILE B 130 R 1000 MTS 1 $
RBL LENGTH I 16 U $
.
SET LENGTH TO LENGTH(MAGFILE) $

```

8 DEVICE IDENTIFICATION OPERATIONS

It is sometimes desirable to make a check on a device (e.g., a tape unit or disk) to make sure that the right unit is mounted). The following two special statements are used for this purpose:

1. DEFID Statement.
2. CHECKID Statement.

These commands are illegal on standard hardware devices.

.8.1 DEFID Statement

DEFID writes an identifier on an external device. If the tape is not at load point when the DEFID command is given, the statement will be ignored.

Format

```
DEFID file-name STANDARD or (header description) $
```

Explanation

DEFID	Specifies a CMS-2 control word.
File Name	The identifier of a file that references the external device for which an ID block is desired.
STANDARD or Header Description	The description of the identifier record to be written. If it is standard, the header is written in the local convention known to the run-time input/output routine. If a header description is given in parentheses, the contents of the parentheses are used as header. All header records are a standard 30 words long.

Examples

1. DEFID LPR STANDARD \$

A standard header will be written on the device referenced in the file declaration LPR.

2. DEFID LPR (INVENTORY SDIEGO 1 JUL 71) \$

The header "INVENTORY SDIEGO 1 JUL 71" will be written on the device referenced in the file declaration LPR.

6.8.2 CHECKID Statement

CHECKID checks an identifier on an external device. If the tape is not at load point when the CHECKID command is given, the statement will be ignored. Failing the check will result in an input/output error condition.

Format

CHECKID file-name STANDARD or (header description) \$

Explanation

CHECKID	Specifies a CMS-2 control word.
File Name	The identifier of a file which references the external device on which the identifier is to be checked.
STANDARD or Header Description	Describes the wording against which the identifier is to be checked.

Example

CHECKID LPR STANDARD \$

The header record on the device referenced in the file declaration LPR will be checked if it is in standard format.

SECTION 7

COMPILE-TIME SYSTEM FACILITIES

The primary function of the CMS-2 Compiler is the translation of source statements into machine (object) code. To effect this translation, certain conventions regarding the organization of the source statements have been established in previous sections. These conventions pertain to system data designs and system procedures as the organizational elements of a system. In the following sections, the header controls of a system are presented. They contain information specifying listing options, loading controls, system index registers, inclusion of program debug features in the object code or other facilities that effect the compile operation.

A compile-time system comprises system elements and associated header controls (see figure 7-1). This concept establishes a method for obtaining a complete or partial translation of a user's program. The compile-time also signals the initiation and the termination of the translation process. For translation purposes, such a system is regarded as something complete in itself; all information necessary for a successful compile is specified within the system and in accordance with the rules and facilities of the CMS-2 language.

7.1 ACCESSING THE COMPILER

To obtain the services of the Compiler requires use of a uniquely formatted Monitor control card. Furthermore, additional Monitor control cards are required to properly define the user's job processing requirements for the Monitor. These requirements might be one or more of the following: compile; load and execute; build a library; update a library; etc.

The \$CMS-2 Monitor control card activates the CMS-2 Compiler. Following this Monitor control card are one or more compile-time systems (source input to the compiler). Use of the Compiler is terminated by the CMS-2 statement: TERMINATE \$.

The card formats of the Monitor control cards, their functions and their placement in a job deck are discussed in Volume I.

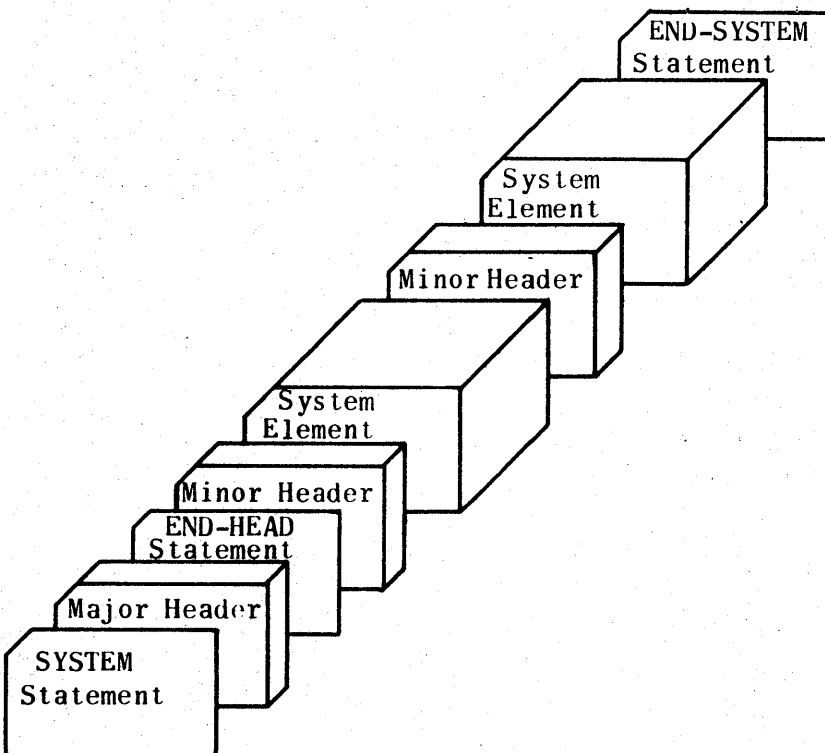


Figure 7-1. Elements of a Compile-Time System

7.2 MAJOR AND MINOR HEADERS

Headers are classified as major or minor depending upon position and range of influence within a compile-time system. A major header consists of those statements immediately following the SYSTEM statement (see paragraph 4.1.1) and bracketed by the statement:

Format

```
END-HEAD name $
```

Explanation

END-HEAD Specifies the end of the header.

Name Optional. The header identifier.

The major header statements contain control information that applies throughout the compile of the entire system. A minor header, on the other hand, is a group of header statements immediately preceding a system element. The parameters within a minor header are in effect only throughout the compilation of the particular system element that the header precedes. Minor headers are optional if sufficient information is included in the major header. Figure 7-1 illustrates the positions of major and minor headers.

Major and minor header may be retrieved from a CMS-2 library. In this case, the header statements are bracketed by HEAD and END-HEAD statements. The format of the HEAD statement follows:

Format

```
name HEAD comments $
```

Explanation

HEAD Specifies a set of header statements to supplement major header controls or establish minor header groups.

Name Optional. The header identifier. If the header group is to be included on a library, the name is required in order to identify the element on the library and for subsequent retrievals.

Below is a list of available header statements, divided into four categories. These statements will be defined and their usage described in following sections.

1. Options header statements:

OPTIONS

2. Allocation header statements:

BASE

TABLEPOOL

DATAPool

LOCDDPOOL

EQUALS

NITEMS

3. Library retrieval header statements:

LIBS

SEL-ELEM

SEL-SYS

SEL-HEAD

SEL-POOL

CORRECT

DEP

4. Miscellaneous header statements:

SYS-INDEX

MEANS

EXCHANGE

DEBUG

CSWITCH-DEL

EXECUTIVE

CMODE

SPILL

CSWITCH-ON

CSWITCH-OFF

CSWITCH

END-CSWITCH

END-CSWITCHS

The OPTIONS, SYS-INDEX, and DEBUG statements are allowable only within a major header. DEP statements are restricted to minor headers. All the remaining statement types may be used in either category of header. The major headers must include the OPTIONS statement. The statements of a major header may occur in any order except for the OPTIONS statement, which should precede all others.

The comment features described in Section 3 are allowable within headers.

7.3 OPTIONS HEADER STATEMENT

Every compile-time system requires a major header containing at least the OPTIONS statement. This statement should immediately follow the SYSTEM statement.

The OPTIONS statement designates the types of output, the output units and the various program listings to be generated by the compiler for a compile-time system.

Format

OPTIONS P₁, P₂, . . . P_n \$

Explanation

OPTIONS	Specifies the OPTIONS header.
P ₁ . . . P _n	Denotes optional parameters of the following types:
SOURCE	Source output and disposition.
OBJECT	Object options, output, and disposition.
LISTING	Listing disposition.
MONITOR	Execution under Monitor.
NONRT	Execution in a non-real-time environment allowing calls to implicit run-time functions.
LEVEL(0)	All error and warning messages are listed. This is the default condition.
LEVEL(1)	Warning messages are not generated.
MODEVRBL	Allows implicit definition of variables.
STRUCTURED	Specifies that the system is written according to CMS-2 structured programming conventions.

Any or all of these parameters may be specified in any order. If no parameters are given or no OPTIONS statement is included in the source input, the Compiler output consists only of syntax error messages. The first three controls listed above may be accompanied by subsidiary information in parentheses following the parameter. This additional information is described in the following paragraphs. The presence or absence of some of this information can affect the duration and comprehensiveness of the compilation. The number of outputs requested may be restricted by the installation's peripheral configuration.

7.3.1 SOURCE Option

This parameter calls for the listing, punching and/or tape output of the (edited) source statements input to the Compiler. The presence of this parameter on the OPTIONS statement indicates a request for source output; the associated parameters indicate the disposition of this source.

Format

SOURCE (LIST, CCOMN, CSRCE, CARDS)

Explanation

SOURCE	Requests output of edited source statements.
LIST	Optional. Indicates that disposition of source output is hardcopy listing. This is the default parameter and is only necessary if a hardcopy listing is desired in addition to one or more of the following parameters.
CCOMN	Optional. Indicates that the disposition of source is the output unit CCOMN.
CSRCE	Optional. Indicates that disposition of source is the output unit CSRCE which is unloaded at the end of the compile time system.
CARDS	Optional. Indicates source output in the form of punched cards.

Any or all of these parameters may be specified in any order. Whenever either CCOMN or CSRCE is requested, any hardcopy listing that results from either the LIST parameter or the OBJECT options will show the source statements numbered as in a Librarian listing. This Compiler output may then be used in lieu of a library listing of the source code.

Examples

1. OPTIONS SOURCE \$
 ^{or}
OPTIONS SOURCE(LIST) \$

Requests a hardcopy listing of the source input.

2. OPTIONS SOURCE(CARDS, LIST) \$

Requests a hardcopy listing and a punched deck of the source input.

3. OPTIONS SOURCE(CCOMN) \$

Requests output of the source input to unit CCOMN (generally for a subsequent library update run). Any hardcopy listing of the source will have statements numbered according to Librarian conventions.

7.3.2 OBJECT Option

This parameter requests the Compiler to proceed all the way through its object generation phases. The subsidiary information with this parameter specifies special generation options, object-code listing forms, and disposition of relocatable binary object-card decks.

NOTE

If OBJECT is not specified on the OPTIONS statement, the Compiler will perform the syntax analysis phase only (producing syntax diagnostics) but will not proceed into the code generation phases. When SOURCE is present without OBJECT, the hardcopy source listing will have syntax error diagnostics interspersed with the source statements.

Format

OBJECT (Special-mode, list-options, CNV, CCOMN, COBJT, CARDS) \$

Explanation

OBJECT Requests object generation phases of the compilation.

Special Mode Optional. One of the following:

CMP Request a compool generation run.

OPT Requests optimization of transient references.

List Options Optional. One or more of the following hardcopy output options:

CR Requests both local and global cross-reference listings, which are alphabetized listings of the identifiers defined within the compile-time system, their assigned locations, and the locations that reference them.

CRG Requests a global cross-reference only.

CRL Requests a local cross-reference only. If CR, CRG, and CRL are omitted, only cross-reference listings of unallocated identifiers are given.

SA Requests symbol analysis listings which provide a list of identifiers categorized according to declarative type. Within each grouping, the identifiers are alphabetized and accompanied by a summary description of their respective attributes.

SM Requests a full symbolic output listing which provides for the complete listing of the source statements intermixed with the octal and mnemonic representations of the generated machine code and their corresponding addresses.

CNV	Optional. Specifies that fixed-to-float and float-to-fixed numeric conversions are to be provided by run-time routines rather than in-line generation.
CCOMN	Optional. Indicates that disposition of relocatable binary object-code is the output unit CCOMN.
COBJT	Optional. Indicates that disposition of relocatable binary object-code is the output unit COBJT, which is unloaded at the end of the compile time system and may be saved.
CARDS	Optional. Indicates relocatable binary object code output in the form of binary punched-card decks.

Any or all of the above primary parameters may be specified in any order.

NOTE

The special mode parameter CMP requests the creation of a compool output. This parameter may be used only with compile-time systems whose elements are system data designs and headers. The compool output is identified by the name of the last system data design and consists of the definitions of all the data design elements decoded into a format internal to the Compiler. A compool may be placed on a library tape and may be retrieved instead of the corresponding source system data designs for subsequent compile-time systems. Using such a compool saves the compilation time normally needed to process the source system data designs.

The special mode parameter OPT requests optimization of transient references. The use of the parameter may result in the elements of a system being no longer independently compilable.

Explanation

LISTING	Requests nonstandard hardcopy listing disposition.
PRINT	Optional. Indicates that disposition of printer listings is the printer. This is the default parameter and is only necessary if printer output is desired in addition to one or more of the following parameters.
CCOMN	Optional. Indicates that disposition of hardcopy listing output is the output unit CCOMN. CCOMN may not be used if specified in OBJECT option.
CLIST	Optional. Indicates that disposition of hardcopy listing output is the output unit CLIST, which is unloaded and may be saved at the end of the compile time system.

Any or all of the above parameters may be specified in any order.

Examples

1. OPTIONS OBJECT(SM), LISTING(CCOMN), \$

Requests the full symbolic hardcopy output listing to be placed on unit CCOMN rather than to be printed.

2. OPTIONS SOURCE(LIST, CARDS),
LISTING(PRINT, CLIST), \$

Requests a hardcopy listing and a punched card deck of the source input. The listing will be printed, as well as placed on the output unit CLIST.

7.3.4 MONITOR Option

The MONITOR parameter allows the compilation of statements that directly or indirectly require access to the CMS-2 operating system (i.e., high-level I/O and DEBUG statements). It also results in all testing of the special console conditions (e.g., KEY1, STOP5) to be simulated by the Monitor. This parameter

should be specified only when the object code produced by the Compiler is to be executed under Monitor control. See Volume I, Section 2 (Monitor) for simulated settings of special console conditions.

7.3.5 NONRT Option

The NONRT parameter indicates that the program is to be executed in a non-real-time environment and allows the generation of calls to implicit run-time functions (exponentiation, BIT/CHAR, and fixed/floating point conversion). The MONITOR option automatically implies the NONRT option. In the absence of NONRT (or MONITOR), all implicit references to these run-time functions will cause source warning messages and/or object error diagnostics.

7.3.6 Two-Level Diagnostics

Error listings produced by the Compiler contain two categories of diagnostic messages: serious errors which affect program execution and warning errors which may not affect program execution. The LEVEL(1) option causes suppression of the listing of errors in the warning category. A LEVEL(0) specification, or no specification of the LEVEL option, causes errors in both categories to be listed. Regardless of level specified or implied, errors in the warning category are not included in the COMPILE ERROR Summary at the end of the compile.

7.3.7 MODEVRBL Option

The MODEVRBL parameter instructs the compiler to create local variable definitions for any undefined data units appearing in dynamic statements where the syntax of the statement allows references to variables. These implicitly defined variables are given the attributes of the MODE VRBL declaration or the compiler's inherent mode (I 16 S) in the absence of a MODE VRBL declaration.

7.3.8 STRUCTURED Option

The STRUCTURED option informs the compiler that the system is written according to the CMS-2 structured programming conventions. The Compiler will issue the warning message "NON-STRUCTURED STATEMENT" for the each statement which violates these conventions. These statements include:

1. Statement switch declarations
2. GOTO statements
3. Procedure declarations containing abnormal exits
4. Procedure call statements containing abnormal exits
5. RETURN statements containing an abnormal exit
6. SET statements containing an OVERFLOW specification
7. Procedure Switch call statements containing an INVALID specification

7.4 ALLOCATION HEADER STATEMENTS

The allocation scheme incorporated in the Compiler generally consists of the assignment of addresses to instructions and data definitions in a sequential manner that reflects the order of the source statements. All identifiers that function as symbolic addresses are assigned locations accordingly. In the case of a data name, the size of the area reserved is determined from its definition in a declarative statement. If the program subsequently specifies a preset value for the data unit, the Compiler generates the preset value originating at the location previously allocated.

The requirements of a particular program or application package often require departure from this standard allocation scheme used by the Compiler. For this purpose, various allocation header statements are provided in the CMS-2 language. Since the effect of these allocation statements on a user program often involves both the Compiler and the Loader, the reader should also refer to the description of the CMS-2 Loader in Volume I, Section 3, in particular, the AC and CS directives.

The words "allocation" or "relative allocation", when used to describe CMS-2 for the AN/UYK-7, will refer to the positioning of an individual data unit or dynamic statement (TABLE, label, etc.) within a basic CMS-2 element (SYS-DD or SYS-PROC). The words "relocatable allocation" will refer to the positioning (offset) of a CMS-2 element from its associated basic register content.

7.4.1 Pooling Statements

Two basic types of pooling statements exist. The first type directs the Compiler to divide a basic element into two separate elements for the purposes of subsequent relocatable allocation by the Loader. Statements in this group are LOCDDPOOL and TABLEPOOL. If it is ever required to treat local data designs and/or tables as relocatable elements at load time, these statements must be

present, with their associated basic elements, SYS-PROC and SYS-DD respectively, during compilation of the source.

The second type of pooling statement includes those pool statements that are associated with the basic CMS-2 elements. They are BASE and DATAPool. It is necessary to include these pool statements with the source only if machine and system-dependent information is included.

In general, pooling statements may occur in both major or minor headers. Pooling statements used in a minor header affect only the SYS-DD or SYS-PROC that immediately follows. (If a pool statement is used that is inappropriate, it is ignored by the Compiler; e.g., LOCDDPOOL within the minor header of a SYS-DD would be ignored.)

If a pooling statement occurs in a major header, it applies to all basic elements not having this pooling statement as a minor header. All four of the above pooling statements may appear in a header, but only one of each type may be specified (i.e., two BASE statements may not appear in the same header).

Each of the pooling statements allows an optional name to be given for the purpose of identifying the pooled element or group of elements. This name defines a compound section and appears in the CS Loader directive generated by the Compiler as part of the binary output. A compound section informs the Loader that relocatable elements are to be grouped together (see Volume I, Section 3). If no name is given on the pooling statements, the Compiler will provide default names. They are as follows:

<u>Pooling Statement</u>	<u>Default Name</u>
BASE	SYSP
DATAPool	SYSDD
LOCDDPOOL	LOCDD
TABLEPOOL	TABLE

Furthermore, if neither the BASE nor the DATAPool statements are provided by the user, the Compiler will use the respective default names for the compound section specifications.

One of the parameters appearing in all of the following pooling statements specifies that the data or instructions in the pool be referenced using a transient base register. This parameter should be used only for extremely large programs.

7.4.1.1 LOCDDPOOL Statement

The LOCDDPOOL statement instructs the Compiler to compile the local data designs with reference to a separate base (i.e., as a relocatable element). If found in a major header, it instructs the Compiler to compile all local data designs in this manner. (A LOCDDPOOL also found in a minor header would be redundant unless it had a different name, requested transient reference, or suggested grouping under a different base register.)

If found in a minor header, only the local data designs of the associated SYS-PROC will be compiled as a relocatable element. All other local data designs will be based relative to the base of the SYS-PROC as encountered within the source.

Format

name LOCDDPOOL (T, identifier) optional-value \$

Explanation

name	An optional name assigned to this element or group of elements. This name defines a compound section and appears in the CS Loader directive generated by the Compiler as part of the binary output.
LOCDDPOOL	Identifies a local data-design pool.
T, Identifier	An optional implementation aid. If not present, normal conventions will be used. If T is present, it specifies that the local data designs covered by this pool statement should be referenced transiently. The identifier, if present, specifies the AN/UYK-7 base register under which

this element may be allocated. This information is passed on to the Loader but may be overridden at load time. The identifier may be an EQUALS tag or a positive integer constant.

Optional Value

The absolute allocation at which it is desired that this element (or group of elements) be located at load-time. The Compiler passes this information to the Loader. The Loader allows this value to be overridden at load time.

Example

```

TRACKM  SYSTEM J DOE 21 APRIL 71 $
.
.
LOCDD  LOCDDPOOL (T, 6) 40000 $
      END-HEAD $
.
.
TRACKA  SYS-PROC $
.
.
      END-SYS-PROC TRACKA $

```

This example represents a program which, because of system design, found itself with a shortage of base registers. Since the programmer realized that he made few references to his own local data designs, he commands the Compiler to use a transient base register whenever any local data is requested in his program. He further commands the Compiler to indicate to the Loader to use S6 as the transient register and to base his local data at address 40000. This can be changed at load time.

7.4.1.2 TABLEPOOL Statement

The TABLEPOOL statement instructs the Compiler to compile all applicable tables with reference to a separate base location. The TABLEPOOL applies only to tables declared in a SYS-DD.

If this pool statement is present in the major header, it is applicable to all tables declared in all SYS-DD's not having a TABLEPOOL statement in their minor header. If this statement is present in the minor header of a SYS-DD, it applies only to tables declared in that SYS-DD.

Format

```
name TABLEPOOL (T, identifier) optional-value $
```

Explanation

Name	An optional name assigned to this element or group of elements. This name defines a compound section and appears in the CS Loader directive generated by the Compiler as part of the binary output.
TABLEPOOL	Specifies pooling of all tables with reference to a separate base location.
T, Identifier	An optional implementation aid. If not present, normal conventions will be used. If T is present, it specifies that the tables declared in the SYS-DD covered by this pool statement should be referenced transiently. The identifier, if present, specifies the AN/UYK-7 base register under which this element may be allocated. This information is passed on to the Loader but may be overridden at load time. The identifier may be an EQUALS tag or a positive integer constant.

Optional Value

The absolute allocation at which it is desired that this element (or group of elements) be located at load time. The Compiler passes this information to the Loader. The Loader allows this value to be overridden at load time.

7.4.1.3 BASE Statement

The BASE statement instructs the Compiler to compile system procedures with reference to a separate base location. This pool statement is considered to be present in the major header by default.

If this pool statement is present in the major header, it is applicable to all SYS-PROC's not covered with a BASE statement in a minor header.

Format

name BASE (T, identifier) optional-value \$

Explanation

Name	An optional name assigned to this element or group of elements. This name defines a compound section and appears in the CS Loader directive generated by the Compiler as part of the binary output.
BASE	Specifies pooling of all instructions with reference to a separate base location.
T, Identifier	An optional implementation aid. If not present, normal conventions will be used. If T is present, it specifies that the generated instructions covered by this pool statement should be referenced transiently. The identifier, if present, specifies the AN/UYK-7 base register under which this element may be allocated. This information is passed on to the Loader but may be overridden at load time. The identifier may be an EQUALS tag or a positive integer constant.

Optional Value

The absolute allocation at which it is desired that this element (or group of elements) be located at load time. The Compiler passes this information to the Loader. The Loader allows this value to be overridden at load time.

7.4.1.4 DATAPOOL Statement

The DATAPOOL statement instructs the Compiler to compile system data designs with reference to a separate base location. This pool statement is considered to be present in the major header by default.

If this pool statement is present in the major header, it is applicable to all SYS-DD's not covered with a DATAPOOL in a minor header.

Format

name DATAPOOL (T, identifier) optional-value \$

Explanation

Name

An optional name assigned to this element or group of elements. This name defines a compound section and appears in the CS Loader directive generated by the Compiler as part of the binary output.

DATAPOOL

Specifies pooling of all data with reference to a separate base location.

T, Identifier

An optional implementation aid. If not present, normal conventions will be used. If T is present, it specifies that the data designs covered by this pool statement should be referenced transiently. The identifier, if present, specifies the AN/UYK-7 base register under which this element may be allocated. This information is passed on to the Loader but may be overridden at load time. The identifier may be an EQUALS tag or a positive integer constant.

Optional Value	The absolute allocation at which it is desired that this element (or group of elements) be located at load time. The Compiler passes this information to the Loader. The Loader allows this value to be overridden at load time.
----------------	--

7.4.2 EQUALS Statement

The EQUALS statement is used for two purposes: the assignment of numeric values to symbols and the specification of relative allocation.

If the EQUALS statement is contained in a major header or system data design, the EQUALS statement applies throughout the system and the value will be substituted wherever the name appears. If the EQUALS statement appears in a minor header or local data design, the value will be substituted only throughout the system procedure or data design which follows the minor header.

When the EQUALS statement is used to assign a numeric value to a symbol used in the following procedures or data designs, the values used in the arithmetic expression must be either constants or values previously defined by an EQUALS statement, or previously specified in a system data design within the system being compiled. Arithmetic expressions appearing in EQUALS statements must be simple, parenthesis-free expressions and are evaluated left to right without precedence consideration. Relative allocation is accomplished with the EQUALS statement where the right-hand side references data units.

Format

name EQUALS expression \$

Explanation

Name	The name of a data unit, or a tag for a numeric constant.
EQUALS	Specifies that an allocation or value assignment follows.
Expression	A simple, parenthesis-free expression representing a relative allocation or numeric value. The basic arithmetic operations of +, -, * and /

are allowed. The operands in this expression may be data unit names, tags defined by previous EQUALS, and constants. Table 10-1 summarizes the rules for legal final results (left side of EQUALS) and legal intermediate results (binary operations within the expression on the right side of the EQUALS).

TABLE 7-1. EQUALS EXPRESSION SUMMARY

ONE OPERAND	OPERATOR	OTHER OPERAND	RESULT (FINAL OR INTERMEDIATE)
Constant (tag)	+ - * /	Constant (tag)	Constant (tag)
Relative (data unit)	+ -	Constant (tag)	Relative (data unit)
Constant (tag)	+	Relative (data unit)	Relative (data unit)
Relative (data unit)	-	Relative (data unit)	Constant (tag)

7.4.2.1 Defining a Tag

When neither the name on the left nor any of the names on the right of an EQUALS statement are the identifiers of data units, the name on the left becomes a tag for the numeric value represented by the expression on the right. This tag may then be used in data declarations and dynamic statements and the appropriate value will be substituted. If the expression on the right involves data units (relative locations) and the result is an integer value, the tag may be used in dynamic statements or other EQUALS statements but not in data declarations.

which are compiled system data designs, in a format internal to the Compiler. The following paragraphs describe the control statements needed to retrieve source elements and compools. Additional control statements are provided for the purpose of correcting source elements during element retrieval. These library retrieval and correction statements may be used either in major or minor headers or in place of system elements of a compile-time system. Library features are described in Volume I, Section 4.

7.5.1 LIBS Statement

Prior to control statements that select elements from a library, the library must be identified by a LIBS statement.

Format

LIBS internal-id (external-id) \$

Explanation

LIBS	The statement identifier.
Internal ID	The name of the library or Compiler output.
External ID	Optional. Some external identification (such as tape reel number) which will be output to the compiling system operator. If not given, the internal ID will be used.

When Compiler-produced outputs are used in library retrieval, the names CCOMN, CSRCE or COBJT must be used as the internal-id. The name used corresponds to the one specified on the OPTIONS statement when the tape was produced by the Compiler. Sources statements may be retrieved from CCOMN or CSRCE; compools from CCOMN or COBJT. If CCOMN is being used as library input and CCOMN has been specified on an OPTIONS statement as an output file, the LIBS statement must include an external-id other than CCOMN to distinguish between the two files.

7.5.2 Retrieval Selection Statements

Elements on a library are identified by name and an optional key. The key is required if the elements selected have been given a key during library preparation. Element keying provides a means of distinguishing between elements of the same name on a library.

The name of a source element for retrieval is the name specified on the HEAD, SYS-DD or SYS-PROC statement. The name of a compool element is the name of the last SYS-DD used to create the compool. The name and the HEAD, END-HEAD, SYS-DD, END-SYS-DD, SYS-PROC and END-SYS-PROC statements define and delimit an element during the retrieval process.

Retrieval of elements from a library is achieved through selection statements which specify the desired elements by name and/or key. There are four types of SEL control statements: SEL-ELEM, SEL-SYS, SEL-HEAD and SEL-POOL.

Formats

SEL-ELEM name (key) , dep-option \$
SEL-SYS (key) \$
SEL-HEAD name (key) , dep-option \$
SEL-POOL name (key) \$

Explanation

SEL-ELEM, SEL-SYS,
SEL-HEAD, SEL-POOL

Statement identifiers.

Name

Identifies the element desired for retrieval.

Key

Required only if the named element has a key on the library. If not required, the parentheses and key are omitted; the key is considered blank.

Dep-option

Optional. Specifies the level of dependent element retrieval. A dep-option may be one of the following:

ALL retrieve all dependent
 elements

ONLY retrieve no dependent
 elements

The SEL-POOL statement must appear in an unnamed major header of the compile-time system such that it precedes all user-defined identifiers except the system name. Retrieval of a compool occurs immediately when requested. Retrieval of elements specified in one or more consecutive select statements commences when one of the following conditions occur:

- a. The Compiler detects the CORRECT statement.
- b. The Compiler encounters a CMS-2 statement other than the LIBS or select statements.
- c. The number of consecutive select statements exceeds 60.

When retrieval is completed for a given set of requests and corrections, the Compiler returns to the standard system input device for the rest of the user's input to the Compiler.

7.5.3 Correcting Elements During Library Retrieval

Source elements may be corrected during the retrieval process; compools may not. The corrections do not modify the library or Compiler output tape itself, but only the elements as they are passed to the Compiler. The name of the element and the card image sequence numbers, as given in the library listing, provide the reference points for making corrections in the form of deletion, insertion or replacement of card images. The Compiler listings produced during a SOURCE output onto CCOMN or CSRCE also provide the same card image sequence numbers.

Corrections decks must be introduced by the statement:

CORRECT \$

which indicates that one or more of the elements (that are to be retrieved as directed by preceding SEL control statements) are to be corrected. Since CORRECT is a CMS-2 statement, it may not start in card columns 1 through 10. The CORRECT card is followed by correction controls (as described in Volume I, Section 4) and CMS-2 statements. The correction deck is terminated by the

Librarian directive /ENDCOR. Within a block of corrections, the order of the corrected elements must be that of the library or Compiler output tape.

7.5.4 DEP Statement

For any given element of a compile-time system, the programmer may declare other elements to be dependent or subordinate to the given element. Such a specification of dependents may appear only in a minor header. An element may have a maximum of 58 dependent elements. Any minor header source element is automatically declared a dependent element of the associated system element.

The DEP statement has no direct effect on the compile process; the information is simply passed through to the source or relocatable output. The dependent element concept has bearing primarily upon the Librarian process and the relocatable loader. In library retrieval, whether of source or relocatable elements, dependent elements are retrieved automatically with the selected element unless otherwise specified by the user. Furthermore, during relocatable loading, all declared dependent elements must be satisfied.

Format

DEP name (key), name (key), ... \$

Explanation

DEP	Declares a dependent element.
Name	The name of another element such as a system data design or system procedure.
Key	An optional key value placed on the element.

7.5.5 Key Specification

Various programmer selected outputs from the Compiler may be incorporated into libraries. Elements on such libraries are identified by name and an optional key. The name of an element output by the Compiler is automatically defined as the name given

on the associated HEAD, SYS-DD or SYS-PROC statement. Key specification provides the programmer with the option of defining library element keys at compile time.

Key specification may be included in the SYSTEM, HEAD, SYS-DD, SYS-PROC and SYS-PROC-REN statements (see Section 4). This section describes the key specification.

Formats

(key)

(key)*S

(key)*O

(key)*C

(key)*L

Explanation

Key	Alphanumeric identifier of not more than four characters.
S	Specifies that the key is to be attached to a source element.
O	The key is to be attached to an object element.
C	The key is to be attached to a compool element.
L	The key is to be attached to a listing element.

More than one key may be specified on any of the applicable declarative statements; each "(key)*element-type" is separated by a comma. Key specification included in the SYSTEM statement applies to all elements of the designated type output for the compile-time system. Key specification included in the HEAD, SYS-DD, SYS-PROC and SYS-PROC-REN statements apply only to outputs associated with that element. If system declared key specifications and an element declared key specification designate the same type of output, the element key is used. Finally, if no output type is attached to the key specification, all output types are keyed; if no key specification is given, all output elements are keyed with blanks.

7.6 MISCELLANEOUS HEADER STATEMENTS

7.6.1 SYS-INDEX Statement

The SYS-INDEX statement may be used only in a major header. This statement assigns a unique identifier to a particular index register. This register is reserved throughout the entire system for use wherever the identifier is referenced.

Format

```
SYS-INDEX n identifier $
```

Explanation

SYS-INDEX	Specifies that a system index is to be declared.
N	An integer specifying a machine index register number from 1 to 5.
Identifier	A unique identifier to which the index register is to be assigned.

Example

```
SYS-INDEX 1 XPOS $
```

The index (B-register) 1 is assigned the name XPOS throughout the system compile.

7.6.2 MEANS Statement

The MEANS statement provides a method of character substitution during the compilation process; no permanent changes are made to the affected source statements.

Format

```
identifier MEANS character-string $
```

Explanation

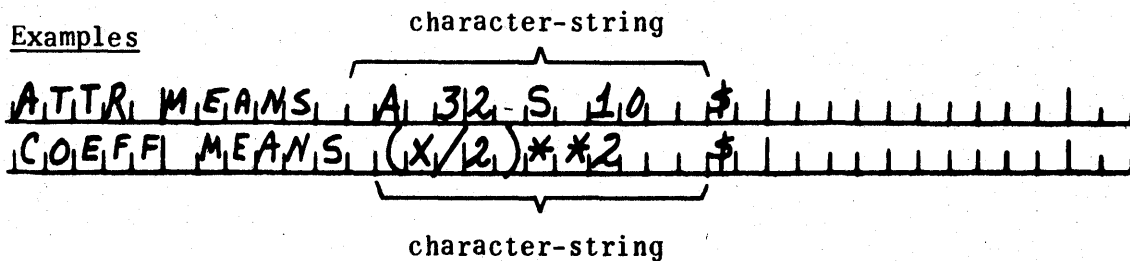
Identifier	Indicates where the substitution is to be made during the compilation. The identifier appears in subsequent statements (but never in another MEANS or EXCHANGE statement).
MEANS	Specifies that a character substitution is to be defined.

Character String

A string of characters that is to be used in place of the identifier. It consists of all characters between the term MEANS and the dollar sign excluding the blank delimiter. It may include other identifiers, constants or CMS-2 symbolic operators. Since the dollar sign terminates the string, it may never appear within the string as a character to be substituted. The maximum number of characters is 132.

If the MEANS is contained in a major header, it applies throughout the system and the character string will be substituted wherever the name appears. If the MEANS appears in a minor header, the string will be substituted only throughout the system procedure or data design that follows the minor header.

Examples



The following statements illustrate the before and after effects of the above character substitutions for the purposes of Compiler interpretation:

VIRBL NAME1 ATTR \$	Before:
VIRBL NAME1 A 32 S 10 \$	After:
SET FLXPT T0 Y**2 H COEFF \$	Before:
SET FLXPT T0 Y**2 H (X/2)**2 \$	After:

7.6.3 EXCHANGE Statement

The EXCHANGE statement provides exactly the same capability as the MEANS statement, except that the specified character substitution appears in the source output and Compiler listings.

Format

identifier EXCHANGE character-string \$

Explanation

Identifier	Indicates where the substitution is to be made during the compilation. The identifier appears in subsequent statements (but never in another MEANS or EXCHANGE statement).
EXCHANGE	Specifies that a character substitution is to be defined.
Character String	A string of characters that is to be used in place of the identifier. It consists of all characters between the term EXCHANGE and the dollar sign excluding the blank delimiter. It may include other identifiers, constants, or CMS-2 symbolic operators. Since the dollar sign terminates the string, it may never appear within the string as a character to be substituted. The maximum number of characters is 132.

7.6.4 DEBUG Statement

Various program checkout statements, as described in Section 8, may be included in the system elements of a compile-time system. These statements are processed by the Compiler. Appropriate calls and parameters for the object-time debug package are generated and included in the object-code output only if the user so requests via the use of the DEBUG control statement in the major header element.

Format

DEBUG parameters \$

Explanation

DEBUG	Requests the Compiler to process those types of program debug statements specified by the parameters.
Parameters	Consist of one or more of the following names separated by commas: DISPLAY, SNAP, RANGE, TRACE, PTRACE, DELETE.

The DISPLAY, SNAP, RANGE, and TRACE parameters permit the compilation of the corresponding types of source debug statements. If one of these parameters is not included in the DEBUG header, the corresponding statements in the system elements are ignored by the Compiler. The PTRACE parameter specifies that code is to be generated during the compilation process to cause a print message to appear during execution before every procedure call. The DELETE parameter specifies that all debug statements not activated by the other parameters are to be deleted from the source output and listings.

7.6.5 CSWITCH Declarations

The CSWITCH feature provides selective compilation of specified sequences of statements within a compile-time system. The CSWITCH selection declaration defines the "on/off" setting. The CSWITCH bracket defines the sequences of statements. The CSWITCH delete declaration instructs the Compiler to remove those sequences which are "off."

7.6.5.1 CSWITCH Selection Declaration

CSWITCH-ON defines the named CSWITCH sequences to be compiled.

CSWITCH-OFF defines the named CSWITCH sequences to be ignored. This declaration is optional; a CSWITCH bracket sequence whose name has not been defined by a CSWITCH selection declaration is considered to be "off."

Format

CSWITCH-ON name-1, name-2, ..., name-n \$

CSWITCH-OFF name-1, name-2, ..., name-n \$

Explanation

CSWITCH-ON Specifies that the listed groups of statements are to be compiled.

CSWITCH-OFF Specifies that the listed groups of statements are not to be compiled.

name-1, ..., name-n Names that identify the selected CSWITCH groups of CMS-2 statements.

A CSWITCH selection declaration may appear anywhere within a compile-time system except within direct code and between a FIND statement and its corresponding IF data statement. The CSWITCH name follows the standard CMS-2 local/global conventions. The "on/off" setting of a CSWITCH name may be reversed at any time during the compile by including the opposite CSWITCH selection declaration. If the setting of a global CSWITCH name is reversed within a system procedure (locally) it is reset to the global setting after the END-SYS-PROC has been processed. If a CSWITCH selection declaration appears within a CSWITCH bracket sequence and reverses the setting of the CSWITCH bracket name, the reversed setting does not affect processing of that CSWITCH bracket sequence.

7.6.5.2 CSWITCH Brackets

The sequence of statements between the CSWITCH bracket declaration and the END-CSWITCH bracket declaration is to be compiled, depending on the "on/off" setting of the CSWITCH name. The CSWITCH bracket may appear anywhere within a compile-time system except within direct code and between a find statement and its corresponding IF data statement.

Format

```
CSWITCH name $  
END-CSWITCH name $  
END-CSWITCHS $
```

Explanation

CSWITCH	Brackets beginning of CSWITCH block.
END-CSWITCH	Brackets end of a CSWITCH block.
END-CSWITCHS	Terminates all CSWITCH sequences.
name	A name of a CSWITCH block. Must correspond to a name in a CSWITCH selection declaration.

CSWITCH brackets may be nested up to a maximum of 10 ("on" or "off") with a last-on first-off sequence. The name following END-CSWITCH terminates that CSWITCH name

M-5035
Change 5

sequence. If the CSWITCH sequence is "off," only CSWITCH warnings are diagnosed; all other syntax checking is suspended until the END-CSWITCH bracket declaration is encountered or the language boundary structure has been violated. A CSWITCH bracket declaration in a data design (local, global or auto), procedure, function or system procedure must have the END-CSWITCH bracket declaration prior to the respective data design bracket (local, global or auto), END-PROC, END-FUNCTION or END-SYS-PROC declaration.

NOTE

A CSWITCH bracket declaration appearing in a header (major or minor) will not be terminated until its corresponding END-CSWITCH bracket declaration, an END-SYSTEM, a TERMINATE, or a monitor control card is encountered. If the END-CSWITCH bracket declaration is not encountered, the remainder of the source will not be compiled when the CSWITCH bracket declaration is "off."

7.6.5.3 CSWITCH Deletion

The CSWITCH delete declaration may appear only in a major or minor header. If in a major header, all sequences of "off" CSWITCH brackets following within the compile-time system are deleted from the listing and source outputs. If in a minor header, all sequences of "off" CSWITCH brackets through the end of the following element are deleted from the listing and the source outputs.

Format

CSWITCH-DEL \$

7.6.5.4 CSWITCH Example

EXMP1 SYSTEM \$

•

•

CSWITCH-ON CSWA1, CSWA2 \$

CSWITCH-OFF CSWA3, CSWA4 \$

END-HEAD \$

SDD1 SYS-DD \$

} Major header.

CSWITCH CSWA2 \$
CSWITCH-ON CSWB2 \$
.
.
END-CSWITCH CSWA2 \$

} CSWITCH CSWA2 is set "on" in the major header; this sequence will compile and CSWB2 will be set "on."

CSWITCH CSWB2 \$
.
.
CSWITCH CSWA3 \$
.
.
END-CSWITCHS \$

} Statements from CSWB2 to CSWA3 will compile. Statements from CSWA3 to END-CSWITCHS will be ignored.

END-SYS-DD SDD1 \$
SPC1 SYS-PROC \$
LOC-DD \$
CSWITCH CSWA1 \$

} Will compile.

END-CSWITCH CSWA1 \$
.
.
CSWITCH CSWA3 \$
.
.
END-LOC-DD \$

} Will not compile. END-LOC-DD will produce END-CSWITCH MISSING diagnostic.

CSWITCH-OFF CSWA2 \$
PROCEDURE PROC1 \$

-
-

CSWITCH CSWA2 \$

-
-

END-CSWITCH CSWA2 \$

-
-

END-PROC PROC1 \$

-
-

END-SYS-PROC SPC1 \$

HED1 HEAD \$

-
-

CSWITCH CSWA3 \$

END-HEAD HED1 \$

SPC2 SYS-PROC \$

-
-

END-SYS-PROC SPC2 \$

HED1 HEAD \$

END-CSWITCH CSWA3 \$

} Will set "off" CSWA2.

} Will not compile.

} Will not compile.

CSWITCH CSWA2 \$
END-HEAD HED1 \$
SPC2 SYS-PROC \$

•
•

CSWITCH CSWA1 \$

•
•

CSWITCH CSWA3 \$

•
•

CSWITCH CSWB2 \$

•
•

END-CSWITCH CSWB2 \$

•
•

END-CSWITCH CSWA3 \$

•
•

END-CSWITCH CSWA1 \$

•
•

END-SYS-PROC SPC2 \$

HED2 HEAD \$

END-CSWITCH CSWA2 \$

END-HEAD HED2 \$

SPC3 SYS-PROC \$

Will not
compile.
CSWB2 is
ignored even
if it is "on."

Will
compile.

Will
compile
since
END-SYS-PROC
has reset
CSWA2
to "on."

CSWITCH CSWA1 \$

-
-

CSWITCH CSWA2 \$

-
-

CSWITCH CSWB2 \$

-
-

END-CSWITCHS \$

-
-

END-SYS-PROC SPC3 \$

END-SYSTEM EXMP1 \$



All will compile. END-
CSWITCHS terminates
the sequence.



7.6.6 EXECUTIVE Statement

The EXECUTIVE statement may appear in a major or minor header and is used to inform the Compiler that the program generation is for use in the interrupt (executive) state of the AN/UYK-7. (The Compiler requires this information when generating control memory references to index registers and accumulators). The Compiler assumes generation for the task state in the absence of this statement.

Format

EXECUTIVE \$

Explanation

EXECUTIVE

Indicates code generation to be executed
in the executive statement

This declaration primarily facilitates the patching of resultant relocatable object code by permitting the use of symbolic addresses (such as statement labels, procedure names or data names) to specify the locations to be patched. This declaration does not alter the normal scope of identifiers during the compilation process.

If SPILL appears in a major header, all local identifiers will be declared as external definitions at output. If in a minor header, only those identifiers in the following element will be declared as external definitions at output.

SECTION 8

DEBUG STATEMENTS

A set of program checkout statements provides the capability for flow analysis and data display while an object program is being executed under control of the operating system. One or more types of program checkout statements may be included in the source input. When the corresponding statement types are enabled, these statements generate calls to debug package routines (see paragraph 7.6.4). Debug package routines may then be selectively activated at program load time (see Volume I, Section 3 for usage of the CMS-2 Loader).

Therefore, the following three conditions must be fulfilled when one or more debug capabilities are desired:

1. The DEBUG header card must be present with the desired debug aid as a parameter. This card instructs the Compiler to generate the code for that aid when encountered. If the debug aid is not included in the header statement, the Compiler will ignore the debug aid and will not generate code for it.
2. The debug aid, as discussed in this section, must be located in the source program deck.
3. The desired debug aid must be included as a parameter on the \$LOAD card (see Volume I, Section 3). This parameter instructs the Loader to set Monitor flags directing execution of the instructions associated with that debug aid. Absence of the debug aid parameter will cause the instructions associated with that missing parameter to be bypassed during program execution.

The various types of program checkout statements and the results of enabling these statements at compile and load time are described in the following paragraphs.

8.1 DISPLAY STATEMENT

The DISPLAY statement allows the contents (image) of specified data units to be output on the system output device in the appropriate format for that data type. Optional value conversion will be made if stated.

Format

```
name DISPLAY image V(w,y), image V(x,y),..., image V(x,y) $
```

Explanation

Name

Optional. An identifier for this statement. If included, this identifier must be followed by a period and is printed with the data units and their contents. This name is not a statement label and therefore may not be referenced.

Image

REGS for machine registers or the identifier of a variable, table, subtable, like-table, item-area, or field. This data-unit reference identifies the image on the printout.

The output format for REGS, table, subtable, like-table, and item-area words is an 11-digit octal number.

The data type for fields and variables is the same as the data type specified for that data unit in its data declaration.

The format of the output is specified below.

<u>Data Type</u>	<u>Format</u>
A	I20.8
I	I20.8
B	The integer 0 or 1
S	The status constant name
F	E20.8
H	Aw (w = number of characters)

V(x,y)

Optional. Specifies the magnitude for conversion for a field or variable. The magnitude must not exceed 15 bits.

Examples

1. DISPLAY M, X, Y

Assuming M is a 4-word table, X is a Hollerith variable, and Y is a floating-point variable, the printout might appear as follows:

```

M  046732115043
    362341023456
    265123245675
    145676343210
X  DOG GONE
Y  0.34244632+07
  
```

2. BETA.DISPLAY TABL(ALPHA, FELD) \$

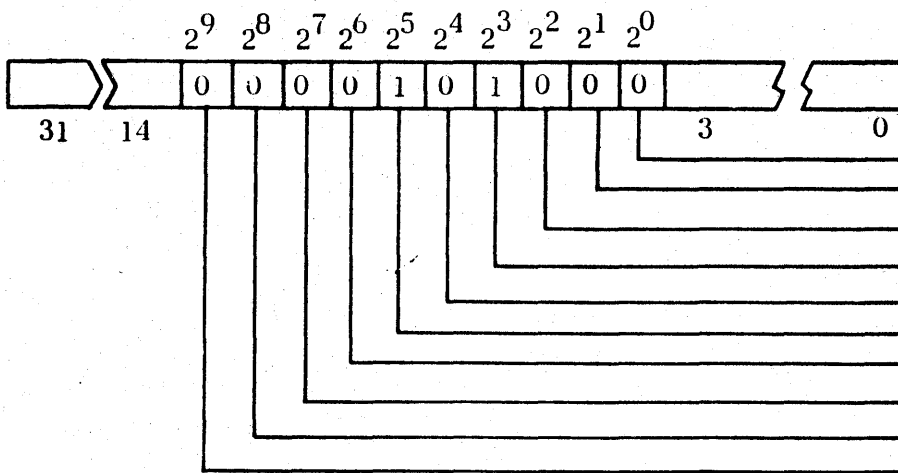
Assuming FELD is an arithmetic field, the printout might appear as follows:

BETA

TABL(ALPHA, FELD) 432.06

3. TABLE NAV V 2 1 \$
FIELD SPEED I 10 4 0 13 \$
END-TABLE NAV \$

KNOTS.DISPLAY NAV(0, SPEED) V(90, 8) \$



Field Bit Position	Magnitude Represented
20	0.15625
21	0.3125
22	0.625
23	1.25
24	2.5
25	5
26	10
27	20
28	40
29	80

Assume that the field SPEED had bit settings as indicated in the illustration above. To execute the display statement with the conversion specification that bit 2^8 of field SPEED represents 40, the output would be:

KNOTS

NAV(0,SPEED) 6.25

where 6.25 is the summation of the binary bit values of 2^5 and 2^3 ($5 + 1.25$) respectively.

8.2 SNAP STATEMENT

The SNAP statement reserves an area image equivalent in size and attributes to the data unit. The first execution of the SNAP statement causes the contents (or converted value) of the data unit to be printed on the system output device and stored in its reserved image area. Subsequent executions of the statement cause a printout only if the contents of the data unit have changed.

Format

name SNAP image V(x,y) \$

Explanation

Name

Optional. An identifier for this statement. If included, this identifier must be followed by a period and is printed with the data unit and its contents. This name is not a statement label and therefore may not be referenced.

Image

An identifier for a table, subtable, like-table, item-area, field, or variable.

The image output format is the same as that described for DISPLAY (see paragraph 8.1).

8.4 TRACE STATEMENT

The TRACE statement generates a line to be printed on the system output device for each executed statement that occurs between the TRACE and END-TRACE statements. This line identifies the flow of execution by the most recent statement label plus an increment of statements relative to this label. The Compiler counts statements whenever a \$ or THEN is encountered. The statement immediately following the TRACE statement should be labeled.

Format

```
TRACE  $  
.  
.  
.  
dynamic statements  
.  
.  
.  
END-TRACE  $
```

Example

```

TRACE $
AA1. SET Z TO X + Y $
     SET Y TO 0 $
     IF X EQ Z THEN GOTO BB1 $
     SET X TO 0 $
BB1. FIND TABL(I, FIELD) EQ X $
     IF DATA NOT FOUND THEN GOTO CC1 $
     SET Y TO Y + 1 $
     RESUME BB1 $
CC1. SET Z TO Y $
     END-TRACE $

```

Assuming that Y is 0 and there are two occurrences of X in field FIELD of table TABL, the printout might appear as follows:

AA1 + 0

AA1 + 1

AA1 + 2

AA1 + 3

BB1 + 0

BB1 + 1

BB1 + 3

BB1 + 4

BB1 + 0

BB1 + 1

BB1 + 3

BB1 + 4

BB1 + 0

BB1 + 1

BB1 + 2

CC1 + 0

If in the above example Y is not 0 and there are no occurrences of X in field FELD of table TABL, the printout might appear as follows:

AA1 + 0

AA1 + 1

AA1 + 2

AA1 + 4

BB1 + 0

BB1 + 1

BB1 + 2

CC1 + 0

8.5 PROCEDURE TRACE (PTRACE)

The PTRACE parameter on the debug header provides a mapping of procedure linkages by printing a message for every CMS-2 procedure call encountered.

Format

PTRACE

Example

DEBUG PTRACE \$

Printed output shows the current procedure name and the called procedure name at each procedure call as follows:

PROCEDURE xxxxxxxxxx CALLING PROCEDURE yyyyyyyyyy

where xxxxxxxxxx is the current procedure being executed and yyyyyyyyyy is the procedure to be executed.

SECTION 9

DIRECT CODE

Direct code statements, or symbolic machine code instructions, are operations which generally result in the generation of a single machine instruction. The CMS-2 Compiler processes as direct code, a subset of the language defined for the CMS-2 Assembler (see Sections 11 and 12). This direct code may appear in both data designs and procedures, but it must be properly bracketed and must follow a specific source card format.

The remainder of this section describes the direct code statement format, the various directives available, and specific processing conventions. In addition, this section describes the differences between the direct code subset that may be embedded in a CMS-2 program for processing by the CMS-2 Compiler and the full assembly language capability available through the CMS-2 Assembler.

9.1 DIRECT CODE STATEMENT FORMAT

When one or more direct code statements appear in a CMS-2 source program, they must be bracketed between two CMS-2 statements provided for that purpose. These statements are described below.

DIRECT	\$	Must immediately precede a sequence of direct code.
CMS-2	\$	Must immediately follow a sequence of direct code.

The format for direct code statements appearing in a CMS-2 program is slightly different from that accepted by the CMS-2 Assembler. The format is consistent, however, with that of CMS-2 source cards in that card columns 1 through 10 are strictly for programmer use and are ignored by the Compiler. The normal direct code format consists of three fields delimited by spaces and commas. Periods indicate the end of a coding line. Direct code statement is limited to one card. The general card format is illustrated below.

Format

CARD-ID label op-code operand

Explanation

- Label** Instruction label. Always starts in column 11 of a coding line. A space in column 11 indicates no label. A label may consist of at least one but not more than eight alphanumeric characters; the first character must be alphabetic. It must not be followed by a period.
- Op Code** Separated from label field by at least one space. May contain a mnemonic function code. A space signifies the end of the op-code field.
- Operand** Contains the elements of the function specified by the op-code. Operand fields are separated by commas. Line termination and comment fields are acceptable and are indicated by a period followed by a space.

9.2 DIRECT CODE STATEMENT REPERTOIRE

The CMS-2 Compiler will accept the full AN/UYK-7 machine instruction repertoire as defined in Appendix G. A more detailed description of these instructions may be found in Section 12.

In addition to processing the symbolic machine language repertoire, the CMS-2 Compiler will accept several Assembler directives and a variety of expressions and constants as operands in the direct code statements.

9.2.1 Direct Code Directives

The following items define the CMS-2 direct code directives that are processed by the CMS-2 Compiler.

1. ABS

Format

label ABS label

Explanation

ABS Translates compile-time location counter value into an object-time absolute address.

2. BYTE

FormatBYTE e_1, e_2 Explanation

BYTE Redefines the embedded character size and number of characters placed in an object word for direct code character strings occurring subsequently within the same CMS-2 element.

e_1 The number of characters to be packed into an object word.

e_2 The size of the character field in bits, not to exceed 16 bits.

3. CHAR

FormatCHAR $C_1, e_1, C_2, e_2 \dots C_n, C_n$ Explanation

C_i The octal code (000 through 377) that is to be redefined.

e_i The redefined value where current character e_i becomes new value e_i .

4. DO

Format

label DO e, direct-constant-entry

Explanation

e An integer defining the number of times the direct constant is to be generated. If a label is specified, it shall apply to the first word of generated data. The direct constant entry must not contain a symbol.

5. FORM

Format

label FORM e_1, e_2, \dots, e_n

Explanation

FORM Describes a special word format specified by the programmer. The word format may include fields of variable length, where the length in bits of each field is user-defined.

e_i The number of bits in a user-defined field. The total number of bits must be equal to or less than 64 and the number of such subfields limited to 16. e_i must be less than 32.

The FORM directive may be implied. The format for utilizing the implicit FORM is illustrated below.

Format

N_1, N_2, \dots, N_n

Only constants are accepted in Form reference sub-fields with the exception of a name appearing in the last sub-field where that sub-field size is defined as 16 bits or greater. When a value appearing in a Form reference sub-field requires more bits than was defined in the Form declaration the leftmost bits of the value will be truncated when packing the resulting constant.

Explanation

N_i Values to be packed into fields of word. The Compiler determines the number of bits required to contain each value by dividing the word into n fields and forming the word accordingly.

6. RES

Format

RES e

Explanation

RES Adds the value of the single expression in the operand field to the current location counter value.

e An expression that must result in a determinable positive value.

9.2.2 Constants

Constants accepted in direct code statements are:

1. Decimal numbers.
2. Octal numbers.
3. Floating-point numbers.
4. Double-word-length octal numbers.
5. Double-word-length decimal numbers.
6. Character strings from one to eight characters in length.
7. Scaled decimal numbers.

Constants may be used in direct constant entry statements and in direct code expressions. The following paragraphs define CMS-2 direct code constants.

9.2.2.1 Decimal Numbers

A decimal number is converted to its binary equivalent and used in its binary form for all computations. The integer may consist of 9 digits if single word and 18 digits if double word specified. The sign of the number is the leftmost bit of the final object word. The first (most significant) digit of the coded decimal number must not be 0. If the decimal number is immediately followed by the letter D, a two-word binary equivalent will result.

Example

```
+52 . PRODUCES OCTAL 0000000064
-16 . PRODUCES OCTAL 3777777757
+512 . PRODUCES OCTAL 0009001009
+65329785D . PRODUCES A TWO-WORD VALUE
              . EQUAL TO 00371155171
              . AND 0000090090
```


Example

+2.5*+6	• PRODUCES	000000000216
	• AND	1142269000
-2.5*+6	• PRODUCES	000000000216
	• AND	26355137777
+10.0	• PRODUCES	00000000009
	• AND	12099000090
-1.0	• PRODUCES	00000000091
	• AND	27777777777
-.3	• PRODUCES	37777777776
	• AND	26310631963

The * + and * - operators are accepted when declaring a floating-point constant.

9.2.2.4 Character Strings

When a + precedes a character string, the Compiler regards the string as a constant; therefore, the number of characters between apostrophes may be from one to eight. One to four characters yield one computer word; five to eight characters yield two computer words. Characters are packed, right-justified, within the generated words with leading binary zeros as required to pad the word. The implied code for character strings in ASCII.

Example

+ 'ACRE'	• PRODUCES	10120651105
+ 'ACREAGE'	• PRODUCES	00020291522
	•	10520293505

Format

N1*/N2

Explanation

N1 Scaled octal number

N2 Decimal number

If N2 is signed negative, N1 shall be converted and shifted right N2 bits.
If N2 is unsigned or positive then N1 shall be converted and shifted left N2 bits.

Example

+05*/16	•	PRODUCES	00000000500
+0500/-6	•	PRODUCES	00000000005
+05*/91	•	PRODUCES	20000000002
-05*/-2	•	PRODUCES	37777777776

9.2.3 Data Expressions

The data expression forms that are accepted by the CMS-2 Compiler are defined below:

1. Constant (as defined in paragraph 9.2.2).
2. Numeric tag (identifier assigned a value by equals statement).
3. Operand₁-operator-operand₂ (operand₂ must be a constant in this form).

The operators allowed within data expressions are:

1. + (addition)
2. - (subtraction)
3. * (multiplication)

4. / (division)

Mixed mode constants are not permitted in data expressions.

9.2.4 Literals

A literal in CMS-2 direct code is defined as a data expression contained within parentheses.

Examples

(56)

('CAT')

One or two object words result from evaluation of a literal. Data expressions allowed in literals are defined in paragraph 9.2.3.

9.2.5 Direct Constant Entries

The CMS-2 Compiler accepts data words declared in direct code that result in one or two generated computer words. Character strings may require more than two words. These direct constant entries shall consist of declared constants as defined in paragraph 9.2.2, character strings, or data expressions.

A + or - sign in the operation field followed by one or more subfields in the operand field signifies that a constant is to be generated. Whenever a + or - sign appears as the first character of the operation field, any number of spaces or no spaces may separate the sign from the first operand. Subfields are separated by commas. In generating constants, the Compiler uses the size of the object computer word. If the operand field contains one subfield, the signed value of the subfield is right-justified in the generated word. If the operand field contains two subfields, two equal-length signed subfields are generated with the values right-justified within each field, and so forth. The first subfield must be signed. Successive subfields may optionally be signed. The absence of a sign implies a positive value. If variants of this implicit equal subdivision of data words are required, the capabilities of the FORM directive may be used to derive the desired format. This is accomplished by referencing the FORM label in the operation field.

Examples

1.

1.	-16384	•	PRODUCES	37777737777
	+8,-4,24,-28	•	PRODUCES	01076612743

If the operand field contains just one subfield immediately followed by a D, or if the constant is a floating-point number, the Compiler generates a double-length constant in two successive computer words. The first generated word of the double-length constant will contain the least-significant bits of the result. The letter D in this context is only meaningful when appended to a numeric constant.

2.

2.	+10.0	•	PRODUCES	0000000001
		•		1200000000
	-16384D	•	PRODUCES	37777737777
		•		37777777777
	+01234567654321D	•	PRODUCES	34567654321
		•		0000000012

Character strings longer than eight characters may be entered as a direct constant entry. No character string in CMS-2 may exceed 132 characters.

9.2.6 Instruction Expressions

The CMS-2 Compiler recognizes and processes simple expressions appearing in the operand field of direct code instructions and in direct code directives. Expression forms allowed are:

- a. Symbol (label, tag, \$).
- b. Symbol \pm constant - If symbol-constant is used and the symbol is externally referenced, see note at end of section 11.3.3.12.

M-5035
Change 2

A symbol defined by a high-level MEANS or EXCHANGE statement may be referenced in direct code. However, the character string to be substituted for the symbol in the direct code is subject to the following restrictions:

- a. The character string may not contain a comma or a period (i.e., it must represent a single term of a single direct code statement).
- b. The character string must be equal in length to the symbol it replaces in direct code.

NOTE

Character substitution for a symbol defined in an EXCHANGE statement will be performed for the purpose of statement interpretation. However, no source card or listing editing is performed.

Example

```
IX MEANS B4 $
PARAM EXCHANGE TAG11 $
.
.
DIRECT $
LB IX, PARAM, K3 .
CMS-2 $
```

The direct code statement would be assembled by the Compiler as:

LB B4, TAG11, K3

9.3 PROCESSING CONVENTIONS

The following conventions apply when processing direct code statements appearing in CMS-2 programs:

- a. Direct code statements, bracketed by DIRECT and CMS-2 may appear within the following source program elements:
 1. Procedures (PROCEDURE).
 2. System data designs (SYS-DD).
 3. Local data designs (LOC-DD).
- b. The rules established for referencing labels within or from outside procedures and data designs apply to direct code segments. The direct code statements assume the same referencing and allocation characteristics as the program element within which the direct code resides.

A direct code statement may be declared to preset a data unit defined in a high-level declaration. The direct code statement must be labeled with the duplicate of the identifier of the high-level declarative and must not precede the high-level declaration. It must be given within the same data design. Any direct code following the preset of a high-level data unit is assumed to be part of that preset unless labeled by an identifier duplicating another high-level definition.

Example

```
EXAMPLE SYS-DD $  
VRBL NAM1 $  
.  
.  
DIRECT $  
.  
NAM1 +625 .  
.  
CMS-2 $  
END-SYS-DD EXAMPLE $
```

The computer word to which the variable NAM1 is allocated will be preset to the decimal number 625 declared in the low-level direct constant entry.

- c. Direct code segments may reference other direct code segments resident in the same SYS-PROC or SYS-DD.
- d. CMS-2 high-level statements may reference direct code statement labels (GOTO or SWITCH only).
- e. CMS-2 high-level statement may not reference data units defined only by direct code statements.
- f. Direct code statements may reference data unit names (symbolic addresses) defined by high-level declaratives. K-designators must be coded explicitly.

- g. Direct code statements may reference high-level statement labels.
- h. The rules of symbols, operator priority, and elements established for direct code statements are independent of those applicable to high-level statements.
- i. The following mnemonics, used to reference computer registers, are recognized by the direct code processor:
1. Accumulators: A0, A1, A2, A3, A4, A5, A6, A7.
 2. Index registers: B0, B1, B2, B3, B4, B5, B6, B7.
 3. Base registers: S0, S1, S2, S3, S4, S5, S6, S7.
 4. Quarter-word memory: Q1, Q2, Q3, Q4.
 5. Half-word memory: L, U.
 6. Whole word memory: W.
 7. K-designators: K0, K1, K2, K3, K4, K5, K6, K7.
- j. Direct code half-word instructions are packed together into one word where possible. Labeled half-word instructions will always be assigned to the upper half of the word. The following tables illustrate the method for packing half-word instructions (HW = half-word instruction, FW = whole-word instruction):

<u>Instruction</u>	<u>Memory Assignment</u>
1. Nonlabeled:	
HW	HW
FW	FW
HW	
HW	
}	HW HW

j. 2. Labeled:

	FW		FW
	HW	}	
	HW		HW HW
	FW		FW
	HW		HW
LABL1	HW		LABL1 HW
	FW		FW
	HW		HW
LABL2	HW		LABL2 HW
LABL3	HW		LABL3 HW

- k. Spaces appearing between elements in operand fields of direct code statements are ignored except where specifically defined as a delimiter, as illustrated below:

MOD	+	RAD		Spaces ignored.	
M	OD	+	R	AD	Illegal because of spaces within element itself.

- l. Labels of direct code in system data designs are treated as global labels by the Compiler.
- m. When referencing procedures declared by high-level statements, the direct code instructions must be the same as the instructions generated by the Compiler for a high-level procedure call.
- n. A direct code statement that contains an explicit coded s-designator (base register) may contain only a constant in the Y-field.

Table 9-1 summarizes the values and symbols that may appear in direct code instruction subfields and in direct constant entries.

Table 9-1. Instruction Sub-field Valid Forms

	a	y	k	b	s	ak	af4	sy	m/b	l	w	p	e	DCE	Lit
1) Octal digit	x		x	x	x					x					
2) Q (1-4)	x		x	x	x					x					
3) "L" (=1)	x		x	x	x					x					
4) "U" (=2)	x		x	x	x					x					
5) "W" (=3)	x		x	x	x					x					
6) A, B, C, D, H, K, S (0 - 7)	x		x	x	x					x					
7) NTAG (=0 - 7)	x		x	x	x					x					
8) MEAN/EXC (2)-6) above)	x		x	x	x					x					
9) Integer ≤ 177						x	x				x	x			
10) NTAG (≤ 177)						x	x				x	x			
11) "ALL" (= 177777)								x							
12) C(0 - 17)								x							
13) Integer ≤ 177777								x							
14) NTAG ≤ 177777								x							
15) Y operand \$ Y								x							
16) Integer ≤ 77										x					
17) \pm Integer or Integer		x									x				

II-9-16

M-5035
Change 2

Table 9-1. (continued)

	a	y	k	b	s	ak	af4	sy	m/b	l	w	p	e	DCE	Lit
18) Integer + Relocatable Identifier		x													
19) Relocatable Identifier + Integer		x													
20) Relocatable Identifier		x													
21) \$+ Integer		x													
22) Literal: (Constant, NTAG, data express.)		x													
23) NTAG		x											x	x	x
24) MEAN/EXC tag		x													
25) Constant													x	x	x
26) Const $\begin{bmatrix} + \\ * \\ / \end{bmatrix}$ Const (no mix)													x	x	x
27) Char String ≤ 132 chars														x	
28) Integer ≤ 37777 (BCW)											x				
≤ 4000 (BCWE)											x				

RTAG - Identifier equated to identifier.
 NTAG - Identifier equated to a value.
 e - data expression
 Identifier - includes RTAG

II-9-20

M-5035
 Change 2

The following items identify additional capabilities available under the CMS-2 Assembler which are not included in the direct code capability of the CMS-2 Compiler.

1. Addressing section declarations in the label field are not allowed (direct code is compiled under the existing high-level allocation environment).
2. Macros and related directives are not allowed (including macro name and statements, paraform usage, and macro reference lines).
3. Library retrieval from direct code is not allowed.
4. Labels of direct code externalized by postfixing an asterisk to the label are not allowed.
5. The CMS-2 Compiler does not process direct code statements containing expressions using parentheses.
6. The following directives are not acceptable as direct code:
 - a. END
 - b. SEGEND
 - c. LINK (can use the high-level EXTREF statement instead)
 - d. Loader directives (*AC, *CS, etc.)
 - e. LLT
 - f. LCR
 - g. LIST, ELIST, and NOLIST
 - h. ODD and EVEN
 - i. PXL
 - j. WRD
 - k. RF\$
 - l. EQU (can use the high-level EQUALS statement instead)
7. SETADR and LIT literal directives are not allowed (the Compiler controls when and where to dump the literals).

SECTION 10
COMPILER OUTPUTS

A variety of hardcopy or listed outputs is available from the CMS-2 Compiler. The method of selecting these various outputs using the OPTIONS statement is described in Section 7. The purpose of Section 10 is to explain the various listing formats, page headers, and column descriptors.

10.1 SOURCE LISTING FORMAT

The source listing provides a record of input to the Compiler. This listing consists of a page header for each listing page containing the element name and number, the date of the compilation and a page number.

If the source listing is requested by a SOURCE option and no OBJECT option, the format is as follows:

<u>Column Heading</u>	<u>Meaning</u>
CARD ID	Columns 1 through 10 of the card image.
SOURCE STATEMENT	Columns 11 through 80 of the card image (the CMS-2 source statement field).
ERROR CONDITION	An error message for any syntax errors detected in the source statement.

If the source listing is requested by a SOURCE option and an OBJECT option (no SM), the body of the listing consists of several columns, the contents of which are explained below.

<u>Column Heading</u>	<u>Meaning</u>
ERR	Used to indicate errors that occurred in the adjacent source statement. A list of these errors, their meanings and identifying numbers that appear in the error column is given in Appendix E.

S	Base register number.
AC	Address counter number.
LOC	The relative memory location (in octal) of the first instruction or data word generated for this statement.
LABEL	The label associated with the source statement or the first 10 characters of the input statement.
STATEMENT	A character field containing the remainder of the CMS-2 source statement.
CID	Four characters of the card ID (columns 1 through 4).
SID	Four characters of the card ID representing the statement number (columns 5 through 8).
CR	The remaining two characters of the card ID.

10.2 SOURCE AND MNEMONIC LISTING FORMAT

The source and mnemonic listing (SM) provides a record of the source input to the Compiler as well as a side-by-side mnemonic and octal representation of the machine instructions generated for the source statements. This listing consists of a page header for each listing page containing the system procedure or system data-design name, the date of the run, the number of the element being compiled, the page number, and the type of element. The body

of the listing consists of several columns, the contents of which are described below.

<u>Column Heading</u>	<u>Meaning</u>
ERR	Used to indicate errors that occurred in the adjacent source statement. A list of these errors, their meaning, and identifying numbers that appear in the error column is presented in Appendix E.
S	Base register number.
AC	Address counter number.
LOC	The relative memory location (in octal) of the data or instruction (13 bits), or the operand of the instruction.
FUNCTION	The first half of the data or instruction containing the operation (upper 16 bits left-adjusted).
X	Flag for external reference (R) or transient reference (T).
LABEL	The label associated with the source statement or the first 10 characters of the input statement.
STATEMENT	A character field containing the remainder of the CMS-2 source statement.
CID	Four characters of the card ID that appeared on the card associated with the statement (columns 1 through 4).

SID	Four characters of the card ID representing the statement number (columns 5 through 8).
CR	The remaining two characters of the card ID. May contain a flag (R) in column 9 if the source statement references a reserved word (see Appendix D).

10.3 LOCAL CROSS-REFERENCE LISTINGS

The local cross-reference (CRL or CR) listing provides a record of each symbol defined in the system element; the listing shows the location of each symbol and all references to that symbol. Also included in the list are global symbols defined in other elements but referenced within the current element. If no CR or CRL is requested, only unallocated identifiers and references are listed. The listing contains a page header at the top of each page. The header shows the name of the system procedure, the date of the run, and page number. The body of the listing consists of several columns, as follows:

<u>Column Heading</u>	<u>Meaning</u>
AC	Address counter number.
S	Base register number.
LOC	Gives the relative location, in octal, of the following label. A location of all sevens denotes an allocation error.
LABEL	The label to which the references apply. The labels are printed alphabetically. The symbol "*****" denotes a generation error in referencing an identifier.

EXT

This column identifies the label as being defined as local or external to the system procedure (blank means local to the system procedure, D means external definition, M means local, implicitly MODE defined in the system procedure, R means external reference, T means transient reference). If not blank, the label will also appear in the global cross-reference listing.

REFERENCES

A set of octal addresses within the system procedure that shows the location of each instruction using the preceding label. The word NONE appears if there were no references to the label. The references are given in the same format (AC S LOC) as the location of the referenced identifier.

10.4 GLOBAL CROSS-REFERENCE LISTING

The global cross-reference (CRG or CR) listing provides a record of each global element defined in the system, showing the name of each system element that referenced it. The global cross-reference appears at the end of the compile.

The listing consists of a header at the top of each page; the header contains the name of the system, the date of the run and the page number.

The body of the listing consists of several columns, as follows:

Column Heading

Meaning

EXT

The label is an external reference.
The definition is presumed to be given in another compile-time system.

LABEL	The label of the element referenced. Labels are printed alphabetically.
DEFINED IN	The name of the system data design or system procedure that contained the element.
REFERENCED BY	The name of each system procedure that referenced the label.

10.5 SYMBOL ANALYSIS FORMAT

The symbol analysis (SA) option provides information as to the usage of the symbols in the system compilation. The analysis utilizes the following categories (the items in the individual categories are printed in alphabetical order):

- a. Files.
- b. Formats.
- c. Tables (including subtables, like-tables, item-areas, fields).
- d. Switches.
- e. Variables.
- f. Procedures-functions.
- g. Index registers declared locally.

A header, printed at the top of each page, gives the heading SYMBOL ANALYSIS and the identifier as either a SYS-DD or SYS-PROC name. Each of the above categories is headed by the category type and bracketed with lines of asterisks. The following are descriptions of the headings for the various categories:

a. Files:

The general heading is FILES DECLARED. The columns are:

<u>Column Heading</u>	<u>Meaning</u>
NAME	The name of the files.

MD	The file mode: H - Hollerith. B - Binary.
TP	The record type: V - Variable length. F - Fixed length. S - Stream.
HRDWR	The hardware code.
MXSZ	The maximum record size.
MXRCD	The maximum number of records.
NSTC	The number of associated status constants.

b. Formats:

The general heading is FORMAT STATEMENTS. There are eight identical heading sets of two headings each as described below:

<u>Column Heading</u>	<u>Meaning</u>
NAME	The name of the format.
XT	If declared, the external identifier, R, for external reference or D for external definition.

c. Tables:

The first segment of the print line identifies the table or associated table. The second segment identifies the attributes of the table or associated table. The third segment identifies the fields and their associated attributes.

<u>Column Heading</u>	<u>Meaning</u>
TABLE-NAME	The name of the prime table being described.
ASSOC NAME	The name of a subtable, like-table of item-area associated with the prime table.
ASSOC TYPE	One of the following: SUB - Subtable. LIKE - Like-table. ITEM - Item-area.
TP	Denotes the type of the prime table: H - Horizontal. V - Vertical. A - Array.
NI TM	If the length of the table is variable (specified by a NITEMS statement), this column will contain NT.
PACK NDIM	Denotes the packing usage on fields for horizontal or vertical tables or denotes the number of dimensions if an array; this may be NULL, MEDIUM or DENSE or an integer from 1 to 7.
ADD MOD	Indicates whether the table is addressed directly (DIR) or addressed indirectly (IND).

WDS/ ITEM	Specifies the number of words per item.
INDEX-NAME	Gives the name of the major index for the table or the associated table.
NO. ITEMS DIMS. SIZE	The number of items if H (horizontal) or V (vertical), or the size of the dimensions if A (array).
EXT	Designates whether the table or associated table is externally defined (D), externally referenced (R), or transiently referenced (T).
START ITEM	The item number at which the associated table starts.
FIELDS	Applies to the remainder of the headings.
NAME	The name of the field.
TP	Defines the field type: F - Floating-point. B - Boolean. H - Hollerith characters. A - Arithmetic fixed-point. I - Integer.
SN	Indicates whether the field type is S for signed, U for unsigned, or blank if neither S nor U apply.
START BIT	Starting bit position in the word.

M-5035

WORD
LOC.

The word number by which the field is addressed.

NO. BITS
OR CHARS

Designates the number of words for multi-word fields, the number of bits for types F, A, or I, the number of characters for type H, or the number of status constants for type S.

FB

Fractional bits.

d. Switches:

The general heading is SWITCHES. The columns are:

Column Heading

Meaning

NAME

The name of the switch.

TYPE

One of the following:

S - Statement switch.

P - Procedure switch.

IT - Item switch.

NO. PTS

Gives the number of switch points.

EX

Defines the external specifications:

R - Externally referenced.

T - Transiently referenced.

D - Externally defined.

SHSW-CVRBL

Indicates the name of a shared switch; if the switch is an item switch (IT), the compared variable is given.

INPUT
PARAMETERS

Three columns of input parameters
if switch type P.

OUTPUT
PARAMETERS

Three columns of output parameters
if switch type P.

e. Variables:

The general heading is VARIABLES. The columns are:

<u>Column Heading</u>	<u>Meaning</u>
NAME	The name of the variable.
TYPE	Gives the variable type: F - Floating-point. B - Boolean. S - Status. I - Integer. A - Arithmetic fixed-point. H - Hollerith.
EX	Defines the external specifications, if any: D - Externally defined. R - Externally referenced. T - Transiently referenced. M - Implicitly and locally defined.
SN	Specifies whether signed (S) or unsigned (U).
FB	Fractional bits.
NO. CHAR. BIT	The number of characters, bits, or status constants depending on type.

f. Procedures-Functions:

The general heading is PROCEDURES-FUNCTIONS. The columns are:

<u>Column Heading</u>	<u>Meaning</u>
NAME	The procedure or function name.
TP	Either P (for procedure) or F (for function).
INPUT PARAMETERS	Four columns listing the input parameters.
OUTPUT PARAMETERS	Four columns listing the output parameters.
EXIT	The names of any abnormal exits.

g. Index Registers Declared Locally:

The general heading is LOC-INDEXES DECLARED. The columns are:

<u>Column Heading</u>	<u>Meaning</u>
NAME	The name of the local index declared for the current system procedure.
REG	The B-register assigned to the above symbolic name, or the letter T if a temporary cell is assigned.
PROCEDURE	The name of the procedure in which the local index is defined.

10.6 COMPILER ERROR SUMMARY

At the end of each compile-time system for which the OBJECT option is used, a summary of errors is listed. The name of each element, its element number within the compile and the number of syntax/generation errors and the number of allocation errors are listed. The syntax/generation errors are individually listed at the front of the compile listing. The allocation errors are flagged

M-5035

in the source (and mnemonic) listing where they occurred and are also listed in the local cross-reference.



SECTION 11

ASSEMBLER

11.1 ASSEMBLER FUNCTIONS

The Assembler accepts symbolic source code in 80-column card image format and translates this coding into an object machine language suitable for loading into the AN/UYK-7 Computer memory via an object-code loader program (see Section 3 of Volume I). The Assembler operates in conjunction with the Monitor to provide programmers with a level of programming assistance not ordinarily associated with an Assembler class of language processors.

The Assembler capabilities include:

- a. Macro directives as well as other directives which enable the programmer to control the assembly process in a positive way via conditional assembly.
- b. A powerful set of directives which enables variable bit-field definitions, character substitution, segmentation, and so forth.
- c. The ability to handle multiple addressing sections (counters) for use in segmenting and assembly-time allocation control.
- d. Printer side-by-side listings of the symbolic source code and also an edited representation of the generated object code.
- e. Optionally selected printer listing of all alphanumeric labels referenced within the source code. The labels are separated into internally referenced labels and those which can be referenced from another program.
- f. Optionally selected printer listing of all alphanumeric labels cross-referenced with their respective addresses.
- g. Evaluation of arithmetic and logical expressions.
- h. Relocatable object machine code output which employs full binary card image format (960 punches per 80-column card).

- i. All Assembler-detected errors in source statements are flagged when encountered.

The \$ASM card commands the Monitor to place the Assembler in memory and initiate its execution.

In operation, the Assembler scans the subsequent symbolic input code twice. The first assembly pass performs a pseudo-generation primarily to record any programmer-defined macro sample code and to define forward referencing. The second assembly pass simultaneously produces the object code and the program listing. When referenced in the program, generation for macro sample code is carried on as a subassembly of the main program. Figures 11-1 and 11-2 show the basic functions performed during the first and second passes, respectively.

11.1.1 Input Language Structure

Inputs to the Assembler consist of programmers prepared symbolic coding statements. The programmer has one basic unit available when constructing symbolic code on the coding sheet. This unit is the operation which may consist of three parts:

1. Label.
2. Statement.
3. Notes.

The label and notes are generally optional attachments to the statement and are always separated from the statement by at least one space. In order to apply notes, the statement must be terminated by a period (.).

Format

LABEL STATEMENT. NOTES

A program written in the CMS-2 Macro Assembler Mnemonic Language consists of action statements. Their structure is discussed in the following paragraphs.

11.1.1.1 Label

Within the programming language, the labels consist of 1 to 8 alphanumeric characters. The first character of any label must be a letter. The letter O should also be used with caution as any character of a label because of the

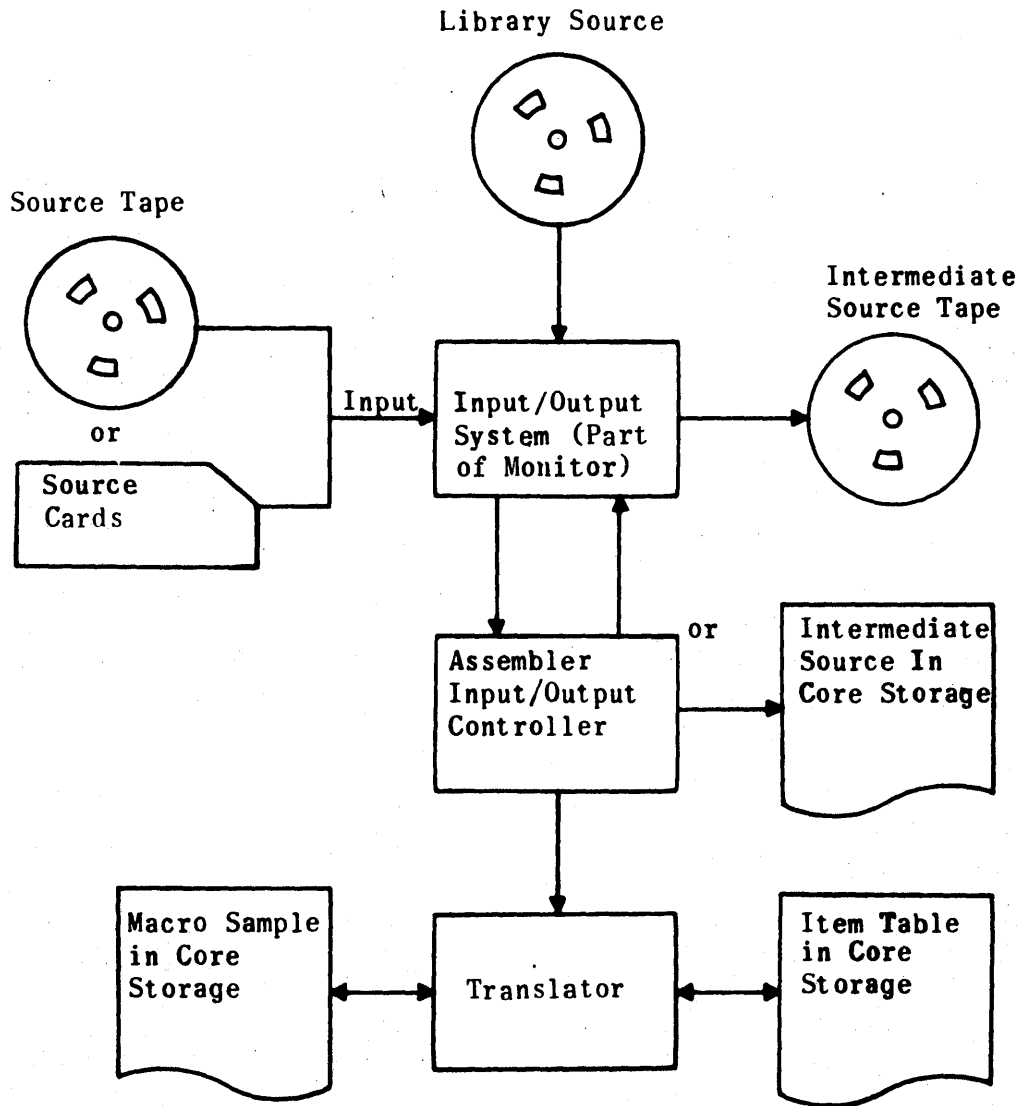


Figure 11-1. Assembler Pass 1 Data Flow

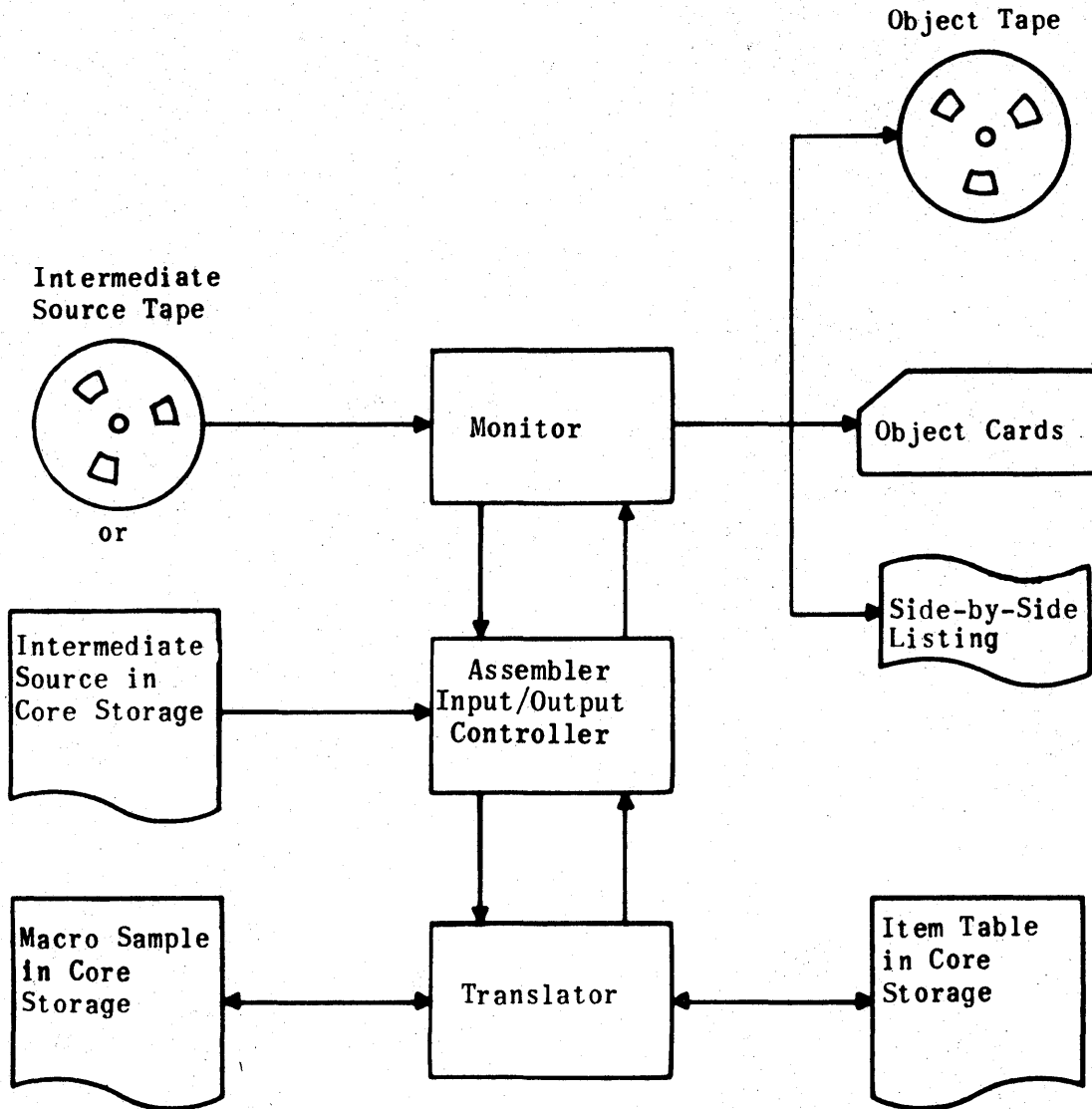


Figure 11-2. Assembler Pass 2 Data Flow

visual difficulty in distinguishing it from zero. A label is used to identify a statement which may be referenced by another.

11.1.1.2 Statements

Except for purely comment statements, programmers are normally concerned with three basic fields of a coding line: 1) label field; 2) operation field; and 3) operand field(s). A line of coding is defined as a logical symbolic statement not necessarily confined to a single physical line; for example, a logical line may extend over several cards.

11.1.1.2.1 Fields. Fields are delimited by at least one space following the last character of the field. There can be no spaces between characters of an element or expression within a field or the Assembler will interpret the space as the end of the field. The label field is always assumed to start in column 1; if there is no label field, its absence is indicated by at least one space starting in column 1.

11.1.1.2.2. Subfields. Any field may consist of one or more subfields separated by a comma and terminated by a space. There can be no spaces between the characters of a subfield or between the subfield and its terminating comma. The Assembler interprets the first space as terminating the current field. To the Assembler, a comma indicates that another subfield follows; therefore the last subfield coded does not terminate with a comma, since a space terminates both the field and the last subfield. Any number of spaces may intervene between a terminating comma and the first character of the next subfield, since the Assembler was alerted that another subfield will follow.

11.1.1.2.3 Omission of Subfields. The first subfield must always be expressed. If the programmer desires to omit this field, he codes a zero followed by a comma. (A coded zero is not legitimate as the first subfield of the label field).

Intermediate subfields may be omitted by coding two successive commas, or comma-space-comma, or comma-zero-comma. The Assembler interprets any of these representations as assigning a zero value to the subfield. Trailing subfields may be omitted by following the last expressed subfield with at least one space. The Assembler assigns a zero value to any missing subfields.

11.1.1.2.4 Statement Continuation. A logical coding line may be interrupted at any point (except between apostrophes) by coding a semicolon (;) as the next character and continuing the line on the next physical line; i.e., next card. The Assembler ignores any characters following the semicolon on the interrupted line and continues its scan starting with column 1 of the next card. The sequence of coding on continuation statements must follow the syntax rules governing fields and sub-fields.

11.1.1.2.5 Statement Termination and Notes. Programmers may include notes as part of their coding lines by following the statements with a period followed by at least one space. The period space combination, except when it appears between apostrophes, causes the Assembler to stop scanning for additional fields or subfields. If a coding line involves a fixed number of fields and subfields and all are encoded on the line, programmers may add notes without the preceding period space combination, since the Assembler ceases scanning when the last required field or subfield is evaluated. A coding line may be nothing but a notes line. In this case, the period space must precede the first character of the notes. Any number of spaces or no spaces may precede the period.

11.1.1.2.6 Blank Card Images. Blank cards (source statements containing no non-space characters) are given a source line number but are otherwise ignored by the Assembler. It is important not to include blank cards within macros since this results in slowing up the expansion of them, whenever called.

11.1.1.2.7 Language Structure Summary. Spaces delimit fields. A comma delimits a subfield except for the last encoded subfield which terminates with a space. A semicolon (except when it appears between apostrophes) denotes line continuation, causing the Assembler to continue its scanning with

externally defined labels to be used as source input in programs assembled at some other time.

11.1.4 Assembly Base Addresses

The Assembler initially sets all initial address counters to zero. The base address of any address counter can be set to some other value by beginning coding for that counter with a SETADR directive. When generating code for the AN/UYK-7 this may have the effect of biasing the base register number on all references to words associated with that address counter. In generating code for non-base register machines, address counter values may have a direct correlation to physical memory addresses.

11.1.5 Conditional Assembly

Sometimes it is desirable to code a program on a modular basis. Certain sections of the program may not be needed at any given time. Whether to omit or include this code may be based on a condition known at assembly time. This leads to what is sometimes called conditional assembly which can be made by two methods:

- a. Any given source statement can be generated dependent on the setting of a condition known at assembly time. This can be done by coding the statement as part of a D0 line, where the D0 count is the result of the condition (either 0 or 1).
- b. An instruction or series of instructions can be effectively overridden or included in object output by following these instructions with a negative reserve (RES) directive, dependent on the result of the condition. If the result of the condition is one, the specified number of previously generated lines of code are overlain by those following the negative reserve line.

Format

```

DO BANKS>2 , LBJ B7,PAX
• THE LBJ B7,PAX IS GENERATED IF
• BANKS HAS AN ASSEMBLY-TIME VALUE
• GREATER THAN 2; OTHERWISE NO WORD
• IS GENERATED.

*ULTRALR(EEFI)SL,SI TESTIPROG
BANKS EQU 2
:
PAX LA 2,DATX,3
:
RES (BANKS<3)*(PAX-$)
• ASSUMING NO CHANGE OF ADDRESS
• COUNTER NUMBER DURING GENERATION
• OF THE ABOVE CODING, ALL OF THE
• INSTRUCTIONS BEGINNING AT PAX
• AND ENDING WITH THE CURRENT
• ADDRESS COUNTER VALUE -1 WILL BE
• OVERLAIN BY THAT GENERATED
• FOLLOWING THE RES LINE.

```

11.1.6 Library Usage

Source library programs can be included within the calling program at the point at which they are called.

11.1.7 Macros

Often programs require repetition of sequences of coding not necessarily identical but similar enough so that repetition of the coding becomes mechanical. A device within the Assembler which generates such sequences is called a macro. The Assembler stores the macro sample code when encountered and generates this coding whenever the procedure is called upon. The Assembler modifies the lines generated in accordance with parameters supplied in the calling line/reference line.

11.1.8 Expressions

An expression is an elementary item or series of elements connected by operators; which, when evaluated, results in a binary value, a floating point number or a memory reference address. If more than one element is included within an expression, they must be separated from one another by operators. An elementary item is an expression containing only one element.

11.1.9 Assembler Generation

The Assembler generates object code in accordance with the capabilities of the AN/UYK-7 Computer instructions.

11.1.9.1 Full-Word

Full words are generated from computer instructions, data words, FORM reference lines (see paragraph 11.3.3.8), and character strings. One data word can produce up to two computer words of object code. A character string can generate a variable number of computer words.

11.1.9.2 Half-Word

If a number of successive half-word (16-bit) instructions are encountered by the Assembler, they are packed two per word. When a half-word is encountered between two full words or is the last of an odd number of successive half-words, it is generated in the upper half-word and the lower half-word contains zeros.

11.1.10 Temporary Storage

Two modes of temporary storage are available: 1) magnetic tape, or 2) computer memory. The standard mode is storage on magnetic tape. Small programs can be assembled using core memory as temporary storage (see paragraph 11.2.1).

11.1.11 Assembler Output

Output from the Assembler consists of relocatable object code. This output is in a format recognized by the Object Code Loader and may also be composed of Loader directives passed on through the Assembler source input language.

11.1.12 Assembly Time Allocation

Allocation of address sections can be achieved at assembly time by including Object Code Loader directives in the source input to the Assembler (see Section 3, Volume I of this document).

11.1.13 Linking

Values corresponding to labels can be made available to some independently processed code by suffixing the labels with an asterisk. Conversely, symbolic program names not defined within the current program can be referred to, and the necessary information is saved to provide a link to the program which defines them. The current program simply declares these names in the operand field of a LINK directive.

11.2 CONTROL CARD

Generally a card containing an asterisk (*) in column 1 is considered by the Assembler to be a control card. One exception to this rule is when column 2 contains a space (blank). This combination is used to achieve assignment of a particular line of coding within a macro to a label found on the macro reference line (refer to paragraph 11.4). Those non-Assembler control cards found between the first card and assembly terminating END card are transmitted to the output code when they are encountered.

11.2.1 Start Assembly (ULTRA)

Each program submitted for assembly must begin with the ULTRA statement.

Format

*ULTRA, source-option, object-option, listing-option name, version

Explanation

Source Option

One of the following (option: R may be used with M/S):

M - Memory will be used as the intermediate device.

S - An ISCM tape will be built which contains the source.
It will be used as the intermediate device.

Blank - A scratch tape will be used for the intermediate
device. This is the default option for source.

R - Source will be resequenced.

Object Option

Any logical combination of the following:

D - Disable object output.

S - An ISCM tape will be built which contains the object
output of the assembly(s). This output will be
saved for the user.

P - Object output will be in the form of binary
punched-card decks.

E - An ISCM tape will be built which contains the object
output of the assembly(s). This option should be
used when the user wants to assemble, load, and
execute his program but does not want the object
output saved after his job has been run. This is
the default option for object and is necessary only
if this option is desired in addition to one of the
other object options.

Listing Option	Any combination of the following: S - An ISCM tape will be built which contains the side-by-side listing output. H - The side-by-side listing will be output to the printer. This is the default for listing and is necessary only if this option is desired in addition to the above listing option.
Name	A one to eight-character name which will be used to build the object output program ID and the library element IDs for ISCM tape output(s).
Version	A one to four-character version which will be used to build the object output program ID and the library element IDs for ISCM tape output(s).

For options which request output on an ISCM tape, the ISCM tape will be given the following internal and external names:

<u>Field</u>	<u>Option</u>	<u>Internal and External Name</u>
Source Option	S	ASOURCE
Object Option	S	AOBJECT
	E	ACOMMON
Listing Option	S	ALIST

No spaces are allowed on the label field between *ULTRA and the three following subfields nor can any be present between the program name and version.

The resequencing option causes the last twelve columns (69-80) to be overlaid with a period-space, four character name, four character number (starting at 0001), and two spaces. The four character name is taken from the first four characters of the element name specified on the *ULTRA card. The final two spaces allow for insert numbers when the assembler is not resequencing.

a programmer codes within the source deck(s) the following statement:

```
*OFO . NOTES
```

*OFO appears in the label field and all other fields are empty.

If no object output is wanted for all the stacked assemblies, a 'D' is coded in the object options subfield of the first *ULTRA card, thus eliminating the need for *OFO cards in each source deck.

11.2.4 Sample Deck Using Control Cards

Figure 11-3 and 11-4 show sample source decks to illustrate the use of Assembler and Monitor control cards.

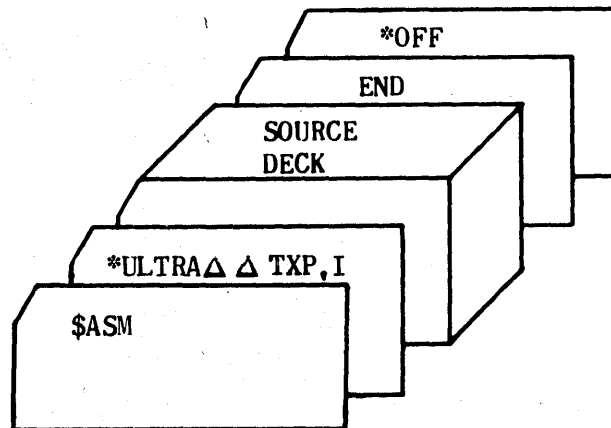


Figure 11-3. Sample Deck Using Control Cards

11.3 SOURCE STATEMENTS

A source statement is a coding line not necessarily confined to a single physical line; for example, a logical line may extend over several cards. Except for purely comment lines, programmers are normally concerned with three basic fields of a coding line: 1) label field, 2) operation field, and 3) operand field(s).

11.3.1 Label Field

The label field must start in column 1 of the source line. The label field of a line of symbolic coding may contain:

- a. An address counter declaration.
- b. A symbolic label.

- c. An address counter declaration followed by a symbolic label.
- d. An asterisk (*) in place of a symbolic label.

If an address counter declaration and a label appear in the label field, the address counter declaration is coded first followed by a comma (,); then the label is coded as in line 2 of the example shown on the next page.

The label field may be preceded by the control character slash (/), which causes the Assembler printer listing to be ejected to the top of the next page. If the slash control character is used, it must be coded in column 1, and the label field must then start in column 2. A space in column 1 (or in column 2 if column 1 contains a slash) implies that the label field is empty.

Examples

\$(4)	DO 4, + 0	COUNTER DECLARATION
\$(0),	LAW RES 8	COUNTER DECL & LABEL
	SB 6, CAT, 3	UNLABELED LINE
MILES	EQU LAW	LABEL ONLY
/ROW	SB 6, CAT+9, 3	PAGE EJECT & LABEL

11.3.1.1 Labels

A label is a means of identifying a symbolic coding line. Normally a label is given the current value of the active address counter. Labels associated with EQU, FORM, GO, DO, MACRO, NAME and LIT have unique interpretations which are explained for these directives under paragraphs 11.3.3 and 11.4.

A label may consist of up to eight alphanumeric characters. The first character must be alphabetic (A through Z). Subsequent characters may be any combination of alphabetic or numeric characters (0 through 9) or \$.

An asterisk (*) may follow a label without intervening spaces. Asterisks so used do not count as a character of the label. If a label outside of a macro is suffixed with an asterisk, the label is externalized (defined as level 0). This label then becomes available outside the program. Refer to paragraph 11.4.4.2 for an explanation of the significance of starred labels within macros.

11.3.1.2 Address Counter Declaration

The first time an address counter is declared, it has the relative value of zero. Subsequent declarations of the same address counter cause the associated generation to continue at the next sequential address, regardless of how many other address counters were declared in between. A declared address counter controls the generated coding until another counter is declared. If no address counter is declared, the entire assembly is under control of address counter zero.

Format

\$ (e)

Explanation

- e The desired address counter 0=31. If an address counter is used in conjunction with the LIT directive, the active address counter is not changed (see paragraph 11.3.3.14).

11.3.1.3 Leading Asterisk (*)

An asterisk (*) may be coded in the label field in place of a symbolic label, within macro definition coding. During macro expansion, this causes the label coded on the macro reference line to take on the value of the active address counter corresponding to the line containing the leading asterisk. The leading asterisk must be followed by a space, and can only be used within a macro definition. The asterisk must only appear once within a particular macro.

11.3.1.4 Half-Word Instruction Labels

The computer instruction repertoire includes a group of half-word (16-bit) instructions as well as full-word (32-bit) instructions. An Assembler-generated object word may, therefore, contain either a single 32-bit instruction or two 16-bit instructions. The programmer codes each half-word instruction as

a single source input statement. Two half-word instructions may not be coded as one source statement.

The Assembler will collect two sequential half-word instructions into a single object word with the first occupying the most significant (upper) half and the second occupying the least significant (lower) half. A single unpaired half-word instruction will be placed in the upper half of the object word and the Assembler will pad the lower half with a no-operation instruction.

Only those half-word instructions which will occupy the upper half of an object word (as described above for the Assembler's pairing convention) may be labeled.

Examples

CODING

• ASSUME ADDRESS COUNTER Q IS ACTIVE WITH
 • A CURRENT VALUE OF 000010 AT THE TIME
 • THE ASSEMBLER ENCOUNTERS THE SEQUENCE:

SIDE - BY - SIDE

000010	1 0 4 3 0 0	000014	LA	4,GOOF,W	.	LINE 1
000011	7 1 4 0 5 0		FLIP	HOR	4,5	. LINE 2
000012	2 4 4 3 0 0	000014	SA	4,GOOF,W	.	LINE 3
000013	7 1 4 5 3 0		FLOP	HAND	4,3	. LINE 4
000013	6 2 4 0 1 0		GOOF	HLC	4,8	. LINE 5

Line 1 is assembled at relative address 000010. Line 2 is assembled into the upper half of relative address 000011; the lower half will be a no-operation. Line 3 is assembled at relative address 000012. Lines 4 and 5 are assembled at relative address 000013 and 000014 respectively, with the logical instruction in the upper half and a no-operation instruction in the lower half of each word. The label FLIP has a value of 000011; label FLOP has a value of 000013. Line 5 violates the half-word labeling convention; therefore, references to GOOF have a value of 000014.

11.3.2 Operation Field

The first non-space (non-blank) character following the label field is assumed to be the start of the operation field except when that character is a period or a semicolon. (A period space signifies line termination; a semicolon signified line continuation.)

The operation field may contain:

- a. A computer-instruction mnemonic function code.
- b. An Assembler directive.
- c. The label of a previously defined FORM directive.
- d. A label already defined as an entry point to macro coding.
- e. + or - followed by a data word. (When the operator is + or -, spaces may separate the sign from its related operand.)
- f. An apostrophe. (When an apostrophe is the first non-blank character following the label field, the remainder of the line through the terminating apostrophe is assumed to be a character string.)

In any event, except as noted in items e) and f) above, a space following any character except a comma signifies the end of the operation field.

Whenever a symbolic line results in generation of a computer word, the value of the controlling address counter is increased by one. An exception to this is the RES directive which causes the counter to be modified by the value derived from the expression in the operand field.

11.3.2.1 Processor Instruction Mnemonics

Processor instruction mnemonic codes consist of up to four alphabetic characters. All half-word instruction mnemonics begin with an H.

11.3.2.2 Input/Output Controller Command Mnemonics

Input/Output Controller (IOC) command mnemonic codes, like those for the processor instructions, consist of up to four letters. All IOC commands are 32 bits in length.

11.3.2.3 Directive Mnemonics

The Assembler provides programmers with a level of programming versatility not ordinarily associated with an assembler-class language processor. This versatility is derived primarily from a group of symbolic assembler directives. Some of these directives allow the programmer to override the imbedded language of the Assembler, to redefine the size and field boundaries of

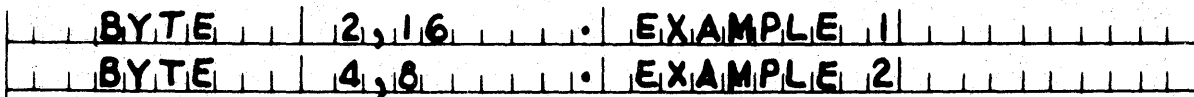
e_2

The size of the character field in bits not to exceed a 16-bit character field. Expression e_2 may be omitted if the character size is eight bits.

The expression $e_1 * e_2$ must be \leq the number of bits in the object word.

If the product of the two expressions exceeds that number of bits, the Assembler sets the expression error flag (E), and ignores the line.

Examples



Assume a thirty-two-bit object word:

The first example causes all character strings encountered thereafter (until the next BYTE directive) to be packed with up to two characters per word, with each character right-justified in a 16-bit field.

The second example alters the generation to four eight-bit characters per word for all subsequent character strings.

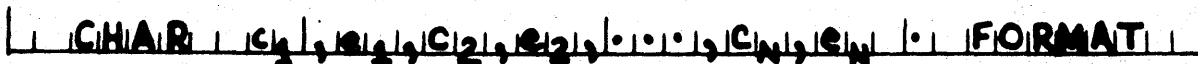
The BYTE directive must be used and expression e_2 encoded if the CHAR (CHARACTER) directive is used to define characters other than eight bits.

The values of e_1 and e_2 must be determinable at the time the Assembler evaluates them.

11.3.3.3 CHAR Directive

The CHAR directive allows the programmer to redefine the Assembler's eight-bit imbedded character set used for generation of characters coded between apostrophes (character strings). If the characters of the redefinition exceed eight bits, the programmer must have previously used the BYTE directive to define the maximum number of characters and the field size to be packed into an object word; otherwise, incorrect generation and/or truncation (T) errors occur.

Format



The CHAR directive is coded with CHAR in the operation field and n pairs of expressions in the operand field. The first expression of each pair defines the octal code (000 through 377) which is to be redefined, and the second expression of each pair is the redefined value. In the absence of a preceding BYTE directive, the Assembler assumes that the redefinition is an eight-bit character set. For all character string generation following a CHAR directive, the Assembler uses the redefined character codes until it encounters another CHAR directive which changes them.

The use of a label on a CHAR line is optional. Values of expressions in the operand list must be determinable when they are evaluated.

Examples

```

CHAR 0101,6,'B',7,'C',8,' ',5
. 'A' NOW PRODUCES 006, 'B' PRODUCES 007.
. 'C' PRODUCES 010, AND ' ' PRODUCES 005;
'ABC' . PRODUCES OCTAL 00601604005
+'B' . PRODUCES OCTAL 00000000007
+'B' ' . PRODUCES OCTAL 00001602405

```

11.3.3.4 DO Directive

The DO directive causes a value or line of coding to be generated a stipulated number of times. DO is written in the operation field. The operand field contains two entries. The first operand entry is an expression defining the DO count (the number of times the second operand is to be done). The second operand entry is any valid symbolic line with or without a label. The two operand entries are separated by space comma. If there is no space following the delimiting comma, the line to be done is presumed to have a label. If one or more spaces follow the delimiting comma, the line to be done is presumed to have no label.

Format

```

LABEL DO e, LINE TO BE DONE . FORMAT 1
LABEL DO e, LABEL LINE TO BE DONE . FORMAT 2

```

There may be a label in the label field of a DO line. Such a label is not equated to the value of the address counter, but is treated as a unique counter with initial value always one. For each iteration of the line to be done, the value of this counter increases by one until the DO count specified by the first operand expression is reached.

To refer to the first word of a group of words generated by a DO statement, use a label which is not a part of the DO statement.

Format

```
X      DO 2 , +0      LINE 1
. ASSUMING NO OTHER DEFINITION FOR X WITHIN
. THE ASSEMBLY, X HAS A FINAL VALUE OF 2.
. IF ONE WISHED TO REFER TO THE ADDRESS
. COUNTER VALUE CORRESPONDING TO THE FIRST
. WORD GENERATED IN LINE 1, HE WOULD CODE:
X      DO 2 , +0
```

Example

```
ITER      DO 65 , +ITER-1
. WILL RESULT IN 65 SUCCESSIVELY GENERATED
. WORDS HAVING THE OCTAL VALUES ZERO
. THROUGH 0100 IN THAT ORDER
```

If the DO count exceeds $2^{16}-1$, an expression (E) error occurs. The Assembler arbitrarily sets the DO count to zero.

The DO directive may be used to perform the line to be done on a conditional basis. In this context the first operand expression (the DO count) results in either zero (false) or one (true). When the result of the condition is zero, no action is taken. When the result of the condition is one, the line to be done is processed once.

Example

```
DO BANKS<4, RES CAT-$
. IF BANKS HAS A VALUE GREATER THAN OR
. EQUAL TO 4, NO RESERVE WILL OCCUR
. IF BANKS HAS THE VALUE 0, 1, 2, OR 3,
. THE CURRENT ADDRESS COUNTER WILL BE
. MODIFIED BY THE VALUE OF CAT-$
```

DO directives may be nested. When nested, the innermost DO cycles to completion first, then the next innermost DO, and so forth.

Example

```
D DO 4, D1 DO 3, D+D1
. WOULD PRODUCE TWELVE SUCCESSIVE WORDS
. HAVING THE CONTENTS 2, 3, 4, 3, 4, 5, 4, 5, 6
. 5, 6, 7 RESPECTIVELY
```

The expression defining a DO count must result in a value determinable at the time it is evaluated. The maximum number of DOs which may be nested at any one time is 16.

Since the label of a DO line takes on the value of the DO count, it cannot be used as a reference to the DO line itself. The programmer may, however, immediately precede a DO statement with a line consisting of a label field only. Whenever the Assembler encounters a line consisting of a label field only, the label is equated to the current value of the active address counter, but the counter is not advanced since no generation occurs; hence a reference to such a label immediately preceding a DO line has the effect of referencing the first word generated from the DO statement.

11.3.3.4A EMBED Directive

The EMBED directive adds defined values for certain assembly tags called Embedded Mnemonics. The embedded mnemonics are turned on by placing the following directive prior to an embedded mnemonic reference.

11.3.3.6 EQU Directive

The EQU (EQUate) directive causes a label in the label field to be equated to the single expression in the operand field for all subsequent references to that label. If the programmer wishes to assign a value to a label, he must define the label via the EQU directive prior to any references to the label. The expression in the operand field must result in a determinable value at the time it is evaluated.

Example

LABEL	EQU	FORMAT
ZAP	EQU	A*B+(F*X0777)/2
ZIP	EQU	ZAP
J	EQU	B+64
A14	EQU	016
BS	EQU	5
RH	EQU	2

Except when it appears in Macro coding, a label defined by EQU may not subsequently be redefined or a duplicate (D) error occurs. Within Macro coding sequences, labels may be redefined via EQU lines unless the definition affects an address counter. The expression on an EQU within a Macro will be evaluated twice by the assembler. Thus, the user should be careful in his use of EQUs whose operand expression involves the label of the EQU.

Example

```

A*  MACRO
B*  EQU  0
    END
    A
C*  MACRO
B*  EQU  B+1
    + B
    END
C      .  GENERATES  00000000002
C      .  GENERATES  00000000004
C      .  GENERATES  00000000006

```

M-5035
Change 3

Labels defined by EQU are not considered relocatable memory references unless the value of the operand expression is a relocatable memory reference.

11. 3. 3. 7 EVEN, ODD Directives

The EVEN directive forces the current address counter value to an even number, thus subsequent words start at that number.

The ODD directive forces the current address counter value to an odd number, thus subsequent words start at that number.

are ignored.

Format

```
INSTR FORM 6, 3, 3, 3, 1, 16
INSTR 054, 0, 04, 01, 0, 010002
```

The line above would produce a word as follows: 26020410002

Format

```
INSTR FORM 6, 14, 12, 14, 6, 6, 6
INSTR 027, 01000, 10, 1000, 167, 012, 4
```

The line above would produce the two words as follows, plus an error flag because the expression 167 (247₈) required more than six bits.

26410000012 01750471204

Expressions in the operand field of a FORM directive line must result in a determinable value.

11.3.3.9 LCR Directive

The LCR (List Cross Reference Table) directive instructs the Assembler to print symbolic names appearing within the program together with the program address(es) where the symbolic names have been referenced or used. This optional listing is produced following the normal side-by-side printed program listing. This directive can appear anywhere in the source coding after the *ULTRA header card. Only references to level 0 or level 1 relocatable labels, to external labels, or to undefined labels will be saved in the cross reference table. Labels are alphanumerically sorted on the listing and appear as follows, (see also paragraph 11.9.4.5):

LABEL ADDRESS COUNTER VALUE

Format

```
LCR . FORMAT
```

11.3.3.10 LIBS Directive

The LIBS (Library Select) directive permits the programmer to specify the names of the library tapes or the ISCM tapes that are needed for library retrieval. The LIBS directive statement works in conjunction with the LIB directive statement (see paragraph 11.3.3.11) with the following stipulations:

- a. A LIBS directive must precede any LIB directive(s).
- b. A LIBS directive is effective until another LIBS directive is encountered. The new LIBS directive then replaces the previous one.
- c. An element retrieved from a library should not contain a LIBS directive. If a retrieved element does contain a LIBS directive, that directive is flagged as an error and is ignored.

Format

LIBS internal-name(external-name),internal-name(external-name)....

Explanation

Internal Name	A one to eight-character name identifying the internal library tape label.
External Name	An optional external tape identifier.

11.3.3.11 LIB Directive

The LIB (LIBRARY element) directive permits the programmer to incorporate source library elements into his current assembly at the point where the LIB directive statement is encountered. The LIB directive must be preceded by the LIBS directive (see paragraph 11.3.3.10)

Format

LIB NAME(VER),NAME(VER),.....

Explanation

Name(Ver)	Name and version of a library element to be merged. The version field is optional; but if it is omitted, the library element to be retrieved must not contain a version on the library tape. If a version is
-----------	--

specified but no name is specified, then all elements on the library(s) with that version will be retrieved. If neither a name nor a version is specified, then all source elements on the library(s) will be retrieved.

Library elements may be corrected during retrieval; and if an output ASOURCE tape has been requested for a tape containing Assembler source data (see paragraph 11.2.1), the corrected elements will be written on the ASOURCE tape. To correct a library element, a /CORRECT card must immediately follow the LIB directive (see Volume I, Section 4, paragraph 4.3.7).

The retrieved library elements are assumed to be symbolic and are assembled as parts of the current assembly program. After the last requested element is retrieved from the library and assembled, the Assembler resumes processing the user's source program from the original input medium commencing with the symbolic statement immediately following the LIB directive statement. Use of library calls have the following limitations:

- a. Nested library calls are not allowed; that is, a called element may not itself contain a call for another library element.
- b. A conditional library call may be used in conjunction with the DO directive to conditionally call a library element.

Example

```
| DO A=1, LIB SIN, I |
```

This line would insert library elements SIN and COSIN into the user's source program if A was previously equated to an absolute value of 1. If A was not defined as being equal to 1, the LIB directive line is ignored. Nesting of DO statements in conjunction with the LIB directive is not allowed.

- c. If the called library element is a macro definition, it is the programmer's responsibility to provide a subsequent call line and

parameters for the macro as part of his source program.

- d. If the called library element is a closed subroutine, it is the programmer's responsibility to establish parameters required by the subroutine and transfer control to the subroutine.
- e. A LIB directive used in conjunction with a DO directive is not allowed within a macro definition.
- f. If a macro is to be called from a library tape via the LIB directive, the total macro must have been entered into the library including its END line.
- g. A LIBS directive must precede any LIB directive(s).

11.3.3.12 LINK Directive

This manual has previously discussed external definition of a program label by suffixing the label with an asterisk (refer to paragraph 11.3.1.1). By doing so, the programmer makes the value (usually a relative address counter value) available at load time to some other independently processed code.

Conversely, the programmer may wish to refer in his coding to symbolic names of programs or data not included in his program. These symbolic names are called external references. The primary purpose for externalizing labels is to permit program and data linkages at object program load time. Both externally defined and externally referenced symbols are transmitted to the object program loader by the Assembler.

The LINK directive is the means whereby the programmer declares externally referenced names.

Format

`LINK N1, N2, ..., Nn . FORMAT`

Explanation

N₁ through N_n Symbolic names referenced by the source program but which appear nowhere as labels in the source program.

Examples

```

(077)
. ASSUMING NO PRECEDING LIT LINE, THE
. OCTAL LITERAL 077 WILL GO
. UNDER ADDRESS COUNTER ZERO
*(1) LIT
(077)
. OCTAL LITERAL 077 WILL GO
. UNDER ADDRESS COUNTER ONE
*(1),ZAP LIT
ZAP(077)
(0177)
. OCTAL LITERAL 077 WILL GO
. UNDER ADDRESS COUNTER ONE, AND
. ASSUMING NO PREVIOUS UNLABELED LIT
. LINE, OCTAL LITERAL 0177 WILL
. GO UNDER ADDRESS COUNTER ZERO

```

11.3.3.15 LLT Directive

The LLT (List Label Table) directive instructs the assembler to print symbolic names appearing within the program together with their associated values. The listing is produced immediately following the normal side-by-side printed program listing. It does not include MACRO, NAME, FORM, and LIT line labels or labels which are purely local to a macro.

Example

```

LLT . FORMAT

```

Labels are alphanumerically sorted on the listing.

Format

```

LABEL | m | VALUE | ADDRESS COUNTER.

```

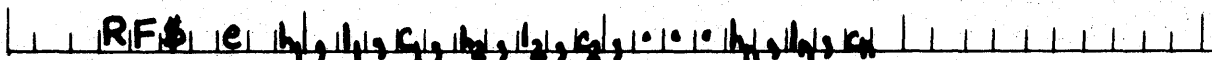

The expression in the operand field must result in a determinable value at the time it is evaluated. The value may be either positive or negative.

A RES line is used to set aside areas for any programmable purpose, or to modify an address counter. The Assembler does not produce any generated words as the result of a RES operation.

11.3.3.18 RF\$ Directive

The RF\$ (Relocation Field) directive is designed to assist users employing the Assembler to generate object programs that differ from that established for normal operating conditions, see Appendix G. The directive permits such users to define relocatable fields within object words and to associate a descriptive modification code with each defined field. The statement must precede any coding which results in object code generation.

Format



Explanation

- | | |
|--|--|
| <code>e</code> | The number of unique relocation fields to be defined. <code>e</code> must be in the range $1 \leq e \leq 16$. |
| <code>h₁-h_n</code> | The higher order bit position of the relocation field. The value of <code>h</code> cannot exceed 31. |
| <code>l₁-l_n</code> | The lower order bit position of the relocation field. |
| <code>c₁-c_n</code> | The modification code to be associated with the relocation field. <code>c</code> must be in the range $1 \leq c \leq 16$. |

A non-relocatable field or word is assumed to carry a modification code of 0. Upon encountering an RF\$ statement, the Assembler overlays its relocatable fields table with those defined by the statement. Each occurrence of the statement causes the overlay; therefore users must declare all desired relocation fields with a single RF\$ statement.

Use of the RF\$ directive, as well as the WRD directive, permits programmers to control not only the size of generated object words but also to define known relocation fields within the generated words, and to associate with each defined field a modification code of their own choosing.

11.3.3.19 SEGEND Directive

The SEGEND (SEGment END) directive provides the programmer with a type of segmentation which differs slightly in concept from the LINK capability. The SEGEND concept may be used when the programmer wishes to assemble a main program followed by one or more sequential overlay segments or when a program is too large to be assembled at one time.

When the Assembler END directive appears at the end of a program, it indicates the end of symbolic input to the Assembler. The Assembler outputs literals, finalizes the assembly, initializes all flags, and clears counters and tables in preparation for a possible new assembly. Nothing pertaining to the just-finished assembly survives this process. However, to assemble successive segments back-to-back, which are actually segmented parts of a single operational program, is made possible by the SEGEND directive. The SEGEND directive in this case replaces the END directive on all segments except the last. SEGEND informs the Assembler that the next immediate assembly is a segment of the current assembly.

Format

SEGEND	.	FORMAT
--------	---	--------

The SEGEND directive causes the Assembler to retain certain information from the just-finished assembly for use by the succeeding assembly.

NOTES

- a. All assembly-level labels which have been suffixed with an asterisk and their associated values are saved in the Assembler's item table so they are available to the next assembly.
- b. Address counter values are not reset to zero; instead they have the highest value attained from the just-finished assembly +1.
- c. Only externally defined symbols are preserved. Externally referenced symbols declared via the LINK directive are not preserved, but must be redefined as required for each segment.

- d. Macros are never preserved from one assembly to the next; they must be redefined for each segment.

11.3.3.20 SETADR Directive

The SETADR (SET ADdRes) directive instructs the Assembler to dump (generate sequentially) literals referred to prior to the occurrence of the SETADR line as defined by the subfield(s) of the operand field.

Format

```
SETADR e, a1, a2, ..., an . FORMAT A  
SETADR e . FORMAT B
```

Explanation

- e Always coded.
- a₁ through a_n (Optional). When present, specify each controlling address counter whose literals are to be dumped.

Literals controlled by the currently active address counter are always dumped and need not, therefore, be included in expressions a₁ through a_n. After literals have been dumped, the value of the currently active address counter is set to the value of expression e (normally the start of the next 8,192 word base).

Examples

```
SETADR 020000 . LINE 1  
SETADR 040000, 3 . LINE 2
```

Line 1 causes the Assembler to dump literals under control of the currently active address counter, then set the counter to 020000.

Line 2 causes the Assembler to dump literals under control of the currently active address counter and those under control of address counter 3, then set the currently active address counter to 040000. The value of address counter 3 must be separately advanced if desired.

Thus the SETADR allows the programmer to ensure that literals and coding referencing them are assembled into the same 8,192 word base group.

The SETADR operation has the effect of erasing dumped literals from the Assembler's item table. Consequently, identical literals may occur under the same address counter but in different SETADR groups.

The Assembler automatically dumps all literals for all address counters when it detects the end of the assembly delimited by the terminating END or SEGEND directive.

11.3.3.21 WRD Directive

The WRD (WoRD) directive is written with WRD in the operation field and a single expression in the operand field which indicates an object computer word size in bits. The value in the operand field may not exceed 32.

Example

WRD	e	FORMAT
WRD	18	EXAMPLE

The WRD directive defines the object computer word size to the Assembler. It is not required when the object word size is 32 bits. Its primary use is in causing the Assembler to generate for other than 32-bit word lengths.

Expression evaluation within the Assembler is carried out modulo, and the object computer word size sign is extended as required for single or double precision values with truncation occurring at the point of output.

Once the WRD directive is encountered, it takes effect for all subsequent generated words until another WRD line is encountered.

11.3.3.22 TAGTBL Directive

The TAGTBL (TAG Table) directive causes all assembly labels other than labels purely local to a macro to be output to the object output. This output will be ignored by the CMS-2 Loader.

Example

```
||TAGTBL|||.NOTICE|||
```

11.4 MACRO STATEMENTS

A macro within the Assembler framework is an arbitrary, programmer-defined guide in generating object code. A macro may be thought of as a sample sub-program whose characteristics are generally defined in the formal parameter references. The actual values for these parameters are supplied when the programmer instructs the Assembler to generate for the sample.

Programs can, of course, be written without using macros; however when repetitive coding sequences occur, it is more efficient and minimizes errors to define the coding sequence just once and let the Assembler produce object words based on the definition. Generation may vary from one call on a macro to the next depending upon the values furnished for interpretation of the macro.

Macro sample programs must always physically precede any call on them. When the assembler encounters a macro directive line, it saves the associated sample through its END line, making use of it only when it is referenced. When a macro is referenced, the Assembler in a sense interrupts its normal sequence and commences generation at the designated entry point in the sample. When this sub-assembly is complete, the Assembler resumes its normal sequence with the next symbolic line.

11.4.1 MACRO and END Directives

Each macro sample begins with a MACRO directive and terminates with an END directive. Both must always be present to delimit the macro sample.

Format

```
A MACRO DEFINITION HAS THE FORMAT
ZEST*  MACRO  e1, e2  .  WHERE e1, AND e2
      . . .  .  ARE OPTIONAL
      . . .  .  . . . REPRESENTS
      . . .  .  MACRO SAMPLE CODE
      END
```

A MACRO directive line must have a label. The label should be suffixed by an asterisk if the call on the macro is via the MACRO line.

Example

```
. THE FOLLOWING MACRO REFERENCE CAUSES
. THE ASSEMBLER TO GENERATE FOR MACRO
. ZEST AT THE POINT WHERE THE CALL APPEARS
ZEST
```

It should be remembered that a macro reference (call) line merely supplies actual values for the macro when any are required. Generation results from matching the actual values of the reference line to they symbolic values within the macro sample coding.

The basic element of a macro reference line is called a subfield. A string of subfields separated by commas is called a field; fields are separated by spaces. Data supplied in this manner are considered to be the actual values to be substituted within the macro sample coding where indicated by the use of symbolic representations. These representations are normally subscripted indexes to elements of the referencing line. These subscripted indexes are called para-forms (parameter reference forms).

A macro directive line may also have an address counter declaration associated with it. The address counter declaration defines the address counter under which the macro is to be expanded. After the macro has been expanded, the address counter is reset to the address counter active at the time of the macro reference (call) line.

Format

EE(1,2), M1 MACRO

11.4.1.1 Paraforms

Paraforms are the means within a macro whereby the programmer references a specific parameter or group of parameters which may be found within a reference line calling on the macro. A part of each paraform is the label of the MACRO directive line. A paraform may consist of the label alone, a label plus one subscript, or the label plus two subscripts. Examples follow:

- a. The most common paraform is label (x,y), where label is the label of the MACRO directive line, x is the field number, and y is the subfield number within field x on the reference line. Thus, ZEST(2,1) indicates that the first subfield of the second field of the reference line is to be substituted wherever ZEST(2,1) appears in the sample coding of macro ZEST. The following example illustrates the use of two fields, assuming a 30-bit object word size.

Example

```

EE*   MACRO
FIU   FORM 7,4,3,4,6,6
      FIU   0152, EE(1,1), EE(1,3),
           EE(1,2), EE(2,2), EE(2,1)
      END
  
```

- WHEN THE ASSEMBLER DETECTS THE EE MACRO LINE, IT RECORDS THE SAMPLE FOR USE WHEN IT IS SUBSEQUENTLY REFERENCED. THE REFERENCE LINE HAS EE IN THE OPERATION FIELD FOLLOWED BY TWO FIELDS OF THREE & TWO SUBFIELDS IN THE OPERAND FIELD.

```

      FIELD 1 FIELD 2
      EE 14,6,1 9,12 REF LINE
  
```

- THE PRECEDING REFERENCE LINE RESULTS IN GENERATION OF OCTAL 6534261411 WHERE 14 REPLACES EE(1,1), 1 REPLACES EE(1,3), 6 REPLACES EE(1,2), AND SO ON.

The writer of the EE macro could have written it to expect one field of five subfields. The resulting generation would have been the same, but the macro sample and the reference line would then have been coded as shown in the next example.

Example

```

EEX      MACRO
FIU      FORM      7, 4, 3, 4, 6, 6
          FIU      0152, EE(1, 1), EE(1, 3), ;
          EE(1, 2), EE(1, 5), ;
          EE(1, 4)
          END
          EE      14, 6, 1, 9, 12      . REF LINE
  
```

- b. A paraform can be written label (x), where x is the field number. The value resulting from such a paraform is the number of subfields found in field x on the macro reference line.

Example

```

• ASSUME THE FOLLOWING LINE IN MACRO FIZ
• DO FIZ(1) = 3, LA 7, 0
• IF THE NUMBER OF SUBFIELDS IN FIELD 1 OF
• THE LINE REFERENCING MACRO FIZ IS 3, THE
• INSTRUCTION LA 7, 0 WOULD BE GENERATED
• SINCE 3 IS SUBSTITUTED FOR FIZ(1) BE-
• CAUSE THE EQUATION 3 = 3 IS TRUE, RESULTING
• IN THE LINE TO BE DONE BEING PROCESSED.
• HAD THE NUMBER OF SUBFIELDS IN FIELD 1
• OF THE REFERENCE LINE BEEN OTHER THAN 3,
• THE EQUATION WOULD HAVE BEEN FALSE AND
• THE LINE TO BE DONE WOULD NOT HAVE BEEN
• PROCESSED
  
```

- c. A paraform may consist only of the macro label without any subscript. The value resulting from such a paraform usage is the number of fields submitted on the macro reference line.

Example

```
• ASSUME THE FOLLOWING LINE IN MARCO FIZ  
  DO FIZ=3 , LA 7,0  
• IF 3 FIELDS ARE PRESENT ON THE LINE  
• REFERENCING FIZ, THE CONDITION IS TRUE  
• AND LA 7,0 WOULD BE GENERATED; OTHER-  
• WISE THE CONDITION IS FALSE, AND  
• LA 7,0 WOULD NOT BE GENERATED.
```

- d. When a paraform involves two subscripts; for example, FITZ (1,1) the programmer may code an asterisk preceding the second subscript; for example, FIX(1,*1). The asterisk preceding in this context has the effect of true (value is one) or false (value is zero) depending on whether or not the corresponding subfield of the reference line is preceded by an asterisk.

Example

```
FIZ* MACRO  
+ 3+(FIZ(1,*1)*0400000++FIZ(1,1))  
END  
FIZ 01000  
• WILL CAUSE FIZ(1,*1) TO HAVE A  
• VALUE OF ZERO SINCE FIZ(1,*1)  
• IS FALSE  
• BUT THE REFERENCE  
  FIZ *01000  
• WILL CAUSE FIZ(1,*1) TO HAVE THE VALUE 1  
• SINCE SUBFIELD 1 OF FIELD 1 IS STARRED  
• AND FIZ(1,*1) IS THEREFORE TRUE
```

NOTE

Paraforms may not be used as labels in the label field of any symbolic line.

Example

```
MF      MACRO  3      .  MAXIMUM OF 3 FIELDS
MFA*   NAME    3
      + MF(1,1)+MF(2,1)+MF(0,0)+MF(3,1)
      GO MFX
MFB*   NAME    6
      + MF(0,1)+MF(0,2)+MF(0,0)+MF(0,3)
MFX*   NAME      .  FORWARD REFERENCE
      END
. SAMPLE REFERENCE LINES
MFA  014 LBL73  3  .  ENTRY VIA MFA
MFB, 013, LBL86, 02 .  ENTRY VIA MFB
```

The count of fields obtained by using the paraform consisting only of the MACRO label includes field zero in the count only when entry is via a NAME line. Entry via a NAME line implies at least one field, field zero.

A NAME line may also be used as a local reference point within a macro.

11.4.2.2 GO Directive

The GO directive transfers Assembler processing to the label in the operand field of the GO line. The label may only be a NAME label or a starred MACRO label. GO provides the mechanism for forward or backward referencing within a macro. By inference, any entry to a macro must have a starred label associated with it (see paragraph 11.4.1 and 11.4.2.1). Externalizing a label is a physical attribute. For assembly purposes, macros are presumed to be nested within the main program; one asterisk overrides this nesting. If a macro is physically nested within another macro, it is one level further removed from the main program. To externalize a label within this innermost macro would therefore require one asterisk to make it available to the outer macro and another asterisk to make it available to the main program.

A special word of caution is necessary when discussing starred labels contained within a macro physically nested inside another macro. A starred label within a macro nested within another macro is not externally defined (or referable)

until the macro containing it has been referenced.

Example

```

      GO LABEL      .  FORMAT
M  MACRO /      .  MAXIMUM / FIELD
AFN* NAME 0//0
PFC* NAME 0//4
      DO M(1,3)=1 , GO SIA
      DO M(1,3)=2 , GO SIB
      + M(0,0)+M(1,1)+0+M(1,2)
      GO ALL
SIA* NAME
      + M(0,0)+M(1,1)+6+M(1,2)
      GO ALL
SIB* NAME
      + M(0,0)+M(1,1)+7+M(1,2)
ALL* NAME
      END

```

A NAME label serving as the destination of a GO must be suffixed by an asterisk if it is a forward reference.

11.4.3 Summary of Macro Usage

A macro describes to the Assembler the format and manner of generation for one or more object words. Input to a macro consists of parameters, which when substituted within the macro, either result in object generation and/or conditionally affect the generation.

A macro begins with a MACRO directive and terminates with an END directive. Entry to a macro may be via the MACRO or NAME lines. MACRO and NAME lines require labels. An asterisk appended to a MACRO or NAME line label defines it as an entry to the macro or a forward reference within the macro.

Macros may be nested within macros. Reference to nested or non-nested macros may be made in a macro only if its definition was encountered prior to the reference.

All directives (except LIBS and LIB) may be used within a macro.

None, one, or two expressions may be encoded in the operand field of a MACRO directive line. The first is the maximum number of fields which may be expected on a reference line, and the second is the precise number of object words which the macro generates. The second expression is never coded if the first is not coded. If the number of fields is variable, the operand field is empty. If the number of words generated by the macro can vary, or if the macro generates half-word instructions or affects the currently active address counter control or contains forward references or external definitions, the second expression must be omitted from the MACRO line operand field. Comments on a MACRO directive line must be preceded by period space. Good programming practice implies that detailed comments be used to describe a macro, including its purpose and the expected order and contents of a line of coding which references it. Such comments should be encoded as pure comment lines outside the macro.

11.4.4 Special Consideration When Coding Macros

11.4.4.1 Comments

Comments on macro reference lines and macro coding lines should always be preceded by a period space, whenever the expected number of fields or subfields may vary. When extensive comments are desirable, it is good programming practice to code several purely comment lines preceding the macro.

11.4.4.2 Labels on a Macro Reference Line

A label may be present on a macro reference line. Under normal conditions this label will be defined equal to the value of the current address counter at the time of the macro call. It is possible to alter the positioning of this label within the macro; that is, it is possible to associate this label with a line within the macro. This is done by coding an asterisk (*) alone in the label field of that particular line in the macro definition. The label of the calling line will be processed exactly as though it had appeared in place of the asterisk, except that it will be defined at the level of the reference line on which it appeared.

Example

```

X*  MACRO  1, 2
    JZ    X(1,1), X(1,2)
*    J    $+3
    END
RAM X    3, ZALENCE
    LB   6, RAM
  
```

In this example, RAM is the address of the J line. If this line has not included the asterisk in the label field, RAM would have been the address of the JZ line.

11.4.4.3 Address Counter Declarations Within a Macro

No change of address counter number is permitted during the expansion of a macro except that governing the generation of in-line constants or literals. Changes of the latter type are made via the LIT directive.

11.4.4.4 Externalizing Labels

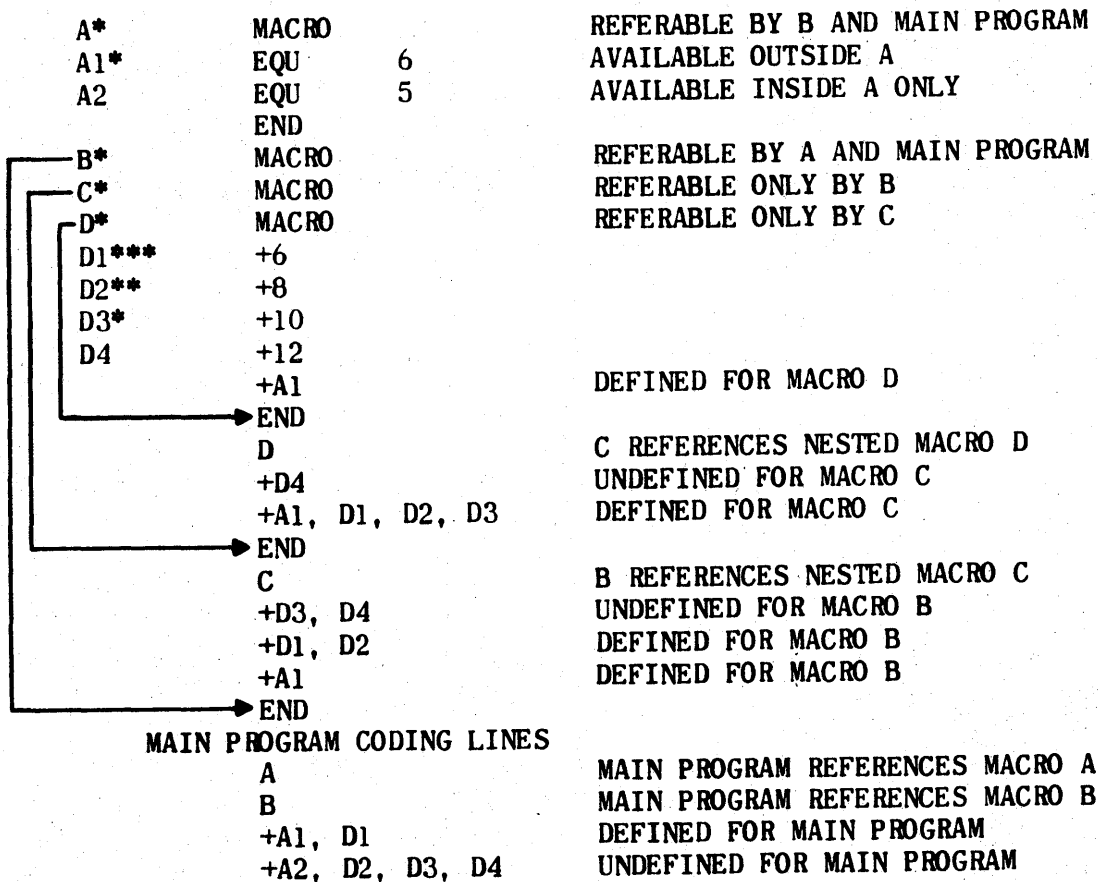
The nesting of macros is permitted up to 29 levels. This nesting can take two forms:

- a. If a macro definition is completely (physically) contained within another macro definition, it is explicitly nested in the larger, and the internal macro is considered to be one level higher than the external macro which contains it. This type of macro may contain other macros. An internal macro may only be referenced after a call has been made on the external macro.
- b. A macro which is called upon by another macro is said to be nested within the calling macro at the time of reference. If a GO statement transfers control to an entrance label of another macro, this is not considered nesting but is a lateral transfer and does not change levels.

The value associated with a label contained in a macro may be defined for use outside of the macro by suffixing the label with an asterisk. A special con-

dition applies to labels contained within macros physically nested within outer macros. One asterisk defines a label for use by the next outer macro. Two asterisks define a label for use by the next two outer macros, and so forth. The skeleton diagram which follows illustrates this method of propagating a label value through the use of suffixing asterisks.

Note that externally defined labels are not defined outside the macro containing them until that macro is referenced. Label D1 remains undefined to the main program until B, C, and D are referenced, in that order. A1 is not defined until A is referenced. Once defined through a reference, A1 and D1 are available, thereafter, throughout the assembly.



11.4.4.5 Macro Reference Lines

Macro reference lines (calls upon macros) are coded following the normal Assembler syntax governing fields and subfields. A macro reference line may be labeled. The label is normally equated to the address of the first word generated from the macro, but may be equated to any line within the macro by coding an asterisk only in the label field of the line. The macro is called by writing the entry label (from the MACRO or a NAME line) in the operation field, followed by as many parameters as may be required in the operand field. Parameters are organized into fields and subfields according to the requirements of the macro. Subfields of the operand field are assumed to be parameters of field 1, field 2, and so forth, in left-to-right order. Any subfield may be preceded by a single asterisk to be used as a conditional value within the referenced macro.

An address counter declaration may be coded on a macro reference line. This declared address counter will supercede any address counter declaration made on the macro directive line. (See also Paragraphs 11.3.1 and 11.5.)

11.4.4.6 Complex Macros

Macro capabilities provide an extremely powerful technique for controlling Assembler generation. Macros can be simple or complex limited only by the programmer's ingenuity. The example which follows of a more complex macro may be used to generate for data word lines assuming a 32-bit object word size.

Example

```

ZAX MACRO 1,1 . . . 1 FIELD, 1 WORD
XB EQU 32/ZA(1) . NO OF PARAMETERS
XC FORM YB, YB, YB, YB, YB, YB, YB, YB .
XC ZA(1,1), ZA(1,2), ZA(1,3), ;
ZA(1,4), ZA(1,5), ZA(1,6), ;
ZA(1,7), ZA(1,8) .
END

```

The following macro example generates an overlapped bit-field object word for an 18-bit object word.

Example

```

M MACRO 1,1 . 1 FIELD, 1 WORD
INX NAME 0
OUTX NAME 2
WRD 18
FN FORM 5,2,2,9
FN 032,M(0,0)++(M(1,2)**04)/04),;
M(1,2)**03,M(1,1)
END
SAMPLE REFERENCES
IN BUFF,7
OUT BUFFA,4

```

11.5 ADDRESS COUNTER DECLARATIONS

The Assembler provides for 32 address counters, and any one may be referenced or used. Grouping of constants and/or instructions under a given address counter may be more convenient or meaningful for segmenting purposes or for collecting coding groups at load time. A specific address counter is declared by coding in the label field:

Format

\$ (e)

Explanation

e The address counter number (0 through 31).

At the start of an assembly, the Assembler assumes address counter 0 is active. All address counters have an initial value of zero. A declared address counter (counter 0 if no other has been declared) remains active until a new address counter is declared and all coding following the declaration is controlled by that counter (assembled relative to zero). Coding resumed under a previously declared counter continues at the next sequential address following the last one used under the counter.

Examples

	LA	7,98		LINE 1
\$ (1)	SA	2,CATCH,3		LINE 2
	SNA	2,CATCH+1,3		LINE 3
\$ (7)	FIL RES	100		LINE 4

11.6 EXPRESSION STATEMENTS

An expression is an element or a series of elements connected by operators which, when evaluated, produce as a final result a binary value, a floating point number, or a memory reference address. The final value may be used as a word(s) of data in memory, as a subfield of an instruction word or data word, or as a parameter for an Assembler directive.

An element is a grouping of characters which is recognizable to the Assembler as an entity and can be replaced by a binary value, a floating point number, or a memory address. An element may be: 1) a symbolic name, (a label); 2) a decimal, octal, or floating point number; 3) the contents of an address counter; or 4) a literal item enclosed by parentheses. A detailed discussion of element follows.

An operator is a special mathematical symbol defining the operation to be performed on the operands immediately preceding and immediately following the operator. An expression need not include any operators if it consists of only one element; however, if more than one element is included within an expression, they must be separated by operators.

Expression evaluation is normally performed in a single-precision mode (single-precision normally being 32 bits, unless altered by the WRD directive). However, if any numerical element of an expression is terminated by the letter D, the remainder of the evaluation shall be performed in a double-precision mode, with a double-precision result generated.

An expression is terminated by either a comma or a space; therefore these two characters cannot be included within an expression. The one exception to this rule is the use of character strings as elements. These are further explained in the following paragraphs.

11.6.1 Labels

An alphanumeric label may be used as an element within an expression. The label must conform to the rules for labels as described in paragraph 11.3.1.1; for example, it must not exceed eight characters; it must consist of alphabetic (A through Z) or numeric (0 through 9) characters, or \$; and the label must begin with an alphabetic character.

When the expression is evaluated, the value allocated to the label is substituted in the expression. If the label is undefined at the time the expression is evaluated, it is assigned a value of 0, and a U error flag appears on the assembly listing adjacent to the line referencing the undefined label.

Example

Z	EQU	024	.	1
BITE2	EQU	Z*2	.	2
	RES	BITE2/Z	.	3
. LINE 1 DEFINES THE LABEL Z				
. LINE 2 DEFINES THE LABEL, BITE2, AND				
. ALSO USES Z AS AN ELEMENT				
. IN THE EXPRESSION				
. LINE 3 USES THE LABEL, BITE2, AS				
. AN ELEMENT OF THE EXPRESSION				

11.6.2 Address Counter

The contents of any address counter used by the program may be referenced in an expression by coding the symbol, \$ or \$(e). \$ signifies that the contents of the current (active) address counter are to be substituted in the expression; \$(e) signifies that the contents of the address counter specified by e are to be substituted. e must be an octal or decimal number, or an expression resulting in a binary value.

Example

	+		B+2		.	1
	+		\$(0)		.	2
	+		CAT		.	3
.	LINE 1 REFERENCES LINE 3					
.	LINE 2 REFERENCES THE					
.	ADDRESS CONTAINED IN COUNTER 0					

11.6.3 Decimal Number

A decimal integer may be used as an element within an expression. The decimal number is converted to its binary equivalent and used in its binary form for all further computations. The integer may consist of any number of decimal digits; however the final binary value represents only the least significant bits of the number as determined by the object word size. The sign of the number is the leftmost bit of the final object word. The first (most significant) digit of the coded decimal number must not be zero (0). If the decimal number is immediately followed by the letter D, a two-word binary equivalent shall be generated.

Example

+52	.	PRODUCES OCTAL	00000000064
-16	.	PRODUCES OCTAL	37777777757
+512	.	PRODUCES OCTAL	00000001000
+65329785D	.	PRODUCES A TWO-WORD VALUE	
	.	EQUAL TO	00371155171
	.	AND	00000000000

11.6.4 Octal Number

An octal integer is specified by preceding the first (most significant) octal digit with a zero (0). Each character of the octal integer must be an octal digit (0 through 7). Rules for evaluation are the same as for decimal numbers. An octal number may also be followed by the letter D to obtain a double-precision result.

Example

+013	•	PRODUCES	00000000013
-0356	•	PRODUCES	37777777421
+0765432101230	•	PRODUCES	36543210123
	•	AND	00000000001

11.6.5 Floating Point Number

A floating point number may be used as an element in an expression. The number must be coded as a decimal mixed number consisting of an integral part and a fractional part, and must include the decimal point. Spaces are not allowed within the number. The number is converted to a 64-bit (regardless of the declared word length) floating point number, formatted in memory as follows:

Least significant 32 bits

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	Mantissa																														

Most significant 32 bits

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Sign of characteristic																Characteristic															

S is the sign of the mantissa.

Example

+2.5*+6	•	PRODUCES	00000000026
	•	AND	11422640000
-2.5*+6	•	PRODUCES	00000000026
	•	AND	26355137777
+10.0	•	PRODUCES	00000000004
	•	AND	12000000000
-1.0	•	PRODUCES	00000000001
	•	AND	27777777777
-.3	•	PRODUCES	37777777776
	•	AND	26310631463

11.6.6 Fixed Point Number

A fixed point number is an expression element containing only an integral part, and hence does not include a decimal point. This type of number is often referred to as a signed pure binary number.

11.7 DATA MODES

Since expression evaluation is essentially an arithmetic process, the elements within an expression (with the exception of character strings) must be able to be identified with a unique numerical quantity. This numerical quantity may exist in one of two formats: 1) a signed pure binary number; or 2) a floating point number. The resultant or final value of the expression is either a binary value or a floating point number. (The final value may be truncated to fit a specified subfield or word size at the time the generated object code is produced in its loadable form.)

All internal expression evaluation is performed in one of these two modes (either binary or floating point). If the expression involves both binary values and floating point values, internal data conversions are performed (for example, binary to floating point, floating point to binary) so the evaluation can be carried out.

11.7.1 Literals

A line of coding without a label field, without leading or trailing spaces, and entirely enclosed in parentheses is called a line item. A line item may only be a symbolic computer instruction, a data word, a macro reference line, or a FORM reference line. The object word(s) (either one or two) generated for the line item is called a literal and is stored in a literal pool. The literal pool address is then substituted for the parenthetical expression in the parent line. The line item must be the only element within the expression being evaluated, and must not be preceded by a + or - sign. If the first subfield of a data word line item is a symbolic name, the name should be preceded by a + or - to avoid possible confusion with a symbolic computer instruction, see paragraph 11.7.2. Six levels of parentheses (for example, five line items nested within line items) are allowed. A double-precision literal constant may be generated by terminating the numeric constant within the parentheses by

the letter D. If the line item is a character string, up to two words can be generated.

Example

DOG	(56)	.	1
	(+CAT, 4)	.	2
. LINE 1 GENERATES A CONSTANT OF 070			
. AND STORES IT IN A LITERAL POOL.			
. AT LINE 1 THE LITERAL POOL ADDRESS			
. OF THE VALUE 070 WILL BE GENERATED.			
NAME	(2048D)	.	PRODUCES A TWO-WORD
. LITERAL POOL CONSTANT OF 00000004000			
. AND 00000000000			
. THE ADDRESS GENERATED FOR THE LITERAL			
. WILL BE THAT OF THE FIRST WORD.			

11.7.2 Data Words

The Assembler recognizes two distinct types of data words: 1) constants, resulting in either one or two generated computer words; and 2) character strings, resulting in one or more generated computer words containing character codes.

11.7.2.1 Constants

A + or - in the operation field followed by one or more subfields in the operand field signifies that a constant is to be generated. Whenever a + or - appears as the first character of the operation field, any number of spaces or no spaces may separate the sign from the first operand. Subfields are separated by commas. In generating constants, the Assembler assumes the size of the object computer word. If the operand field contains one subfield, the signed value of the subfield is right-justified in the generated word. If the operand field contains two subfields, two equal-length signed subfields are generated with the values right-justified within each field, and so forth. The first subfield must be signed. Successive subfields may optionally be signed. The absence of a sign implies a positive value. If variants of this implicit

equal subdivision of data words are required, the capabilities of the FORM directive may be used to derive the desired format.

Example

-16384	.	PRODUCES	37777737777
+'A',-0257	.	PRODUCES	00016177520
+8,-4,21,-28	.	PRODUCES	01076612743

If the operand field contains just one subfield immediately followed by a D, or if the constant is a floating point number, the Assembler generates a double-length constant in two successive computer words. The first generated word of the double-length constant will contain the least significant bits of the result. The letter D in this context is only meaningful when appended to a numeric constant.

Examples

+ 10.0	.	PRODUCES	00000000004
	.		12000000000
-16384D	.	PRODUCES	37777737777
	.		37777777777
+01234567654321D	.	PRODUCES	34567654321
	.		00000000012

11.7.2.2 Character Strings

Strings of characters which can be represented by the Assembler's imbedded character set (with the exception of the apostrophe) may be encoded by enclosing the entire character string between apostrophes. The apostrophe is not included in the allowable characters because it is the control character delimiting the string. A semicolon between apostrophes is treated as a character, not as a line continuation symbol. Similarly, a period space combination between apostrophes is treated as two characters and not as a line termination.

The Assembler's internal character set is the USA Standard Code for Information Interchange, commonly referred to as ASCII code. Unless directed to the contrary by the CHAR and BYTE directives, the Assembler will right-justify each ASCII-

coded character within an eight-bit field, up to a maximum of four characters per generated word.

If the first non-blank character of the operation field is an apostrophe, all subsequent characters up to, but not including the terminating apostrophe are packed left-justified into as many successive computer words as are required to accommodate the string. Any remaining partial word is padded on the right with space characters. If the object word size is not evenly divisible by the declared character length (see paragraphs 11.3.3.2 and 11.3.3.3), the unusable bits will appear as binary zeros in the rightmost bit positions of the generated object word.

If a + precedes a character string, the Assembler regards the string as a constant; therefore, the number of characters between apostrophes may be from one to eight. One to four characters yield one computer word; five to eight characters yield two computer words. Characters are packed right-justified within the generated words with leading binary zeros as required to pad the word. However, if two computer words are generated, the character string is regarded as a double-length constant and the leading character codes will be in the second word.

Examples

+ 'ACRE'	•	PRODUCES	10120651105
+ 'ACREAGE'	•	PRODUCES	10520243505
	•		00020241522
'ACREAGE'	•	PRODUCES	10120651105
	•		10121642440

Refer to paragraph 11.3.3 for descriptions of the BYTE, CHAR, FORM and WRD directives. These directives may be used to deviate from the imbedded object word length and eight-bit character framework.

11.8 OPERATORS

The Assembler provides 14 mathematical operators which define the exact sequence and manner in which elements are to be combined within an expression during evaluation. These operators serve essentially the same purpose as those

encountered within a normal algebraic expression.

Similar to the rules of algebra, which assign a priority to each of its operators (for example, multiplication is performed before addition), the rules for expression evaluation assign a priority to each of the 14 operators recognized by the Assembler. These priorities may be overridden by using parenthetical grouping, in the same way that parenthetical grouping is used in an algebraic expression. The rules for evaluation are discussed in greater detail under Operator Priorities (see paragraph 11.8.2).

Each of the 14 operators falls into one of three classes: 1) arithmetic, 2) logical, or 3) conditional. Arithmetic operators have the highest priority; conditional the lowest.

11.8.1 Symbols

The operator symbols as well as their relative priorities are listed in Table 11-1.

Table 11-2 summarizes the data formats allowed with each operator, and the format of the resultant value. Note that the table is divided into four columns, labeled First Item, Operator, Second Item, and Result. If the entries under First Item and Second Item for a particular operator specify both binary and floating, the corresponding value for that item may be either data format; however, the result is in the format specified under the Result column. The actual evaluation is performed in the mode specified for the result.

TABLE 11-1. OPERATORS AND PRIORITIES OF OPERATORS

Relative Priority	Operator	Meaning
6	*+	$A *+ B$ is equivalent to $A * 10^B$
6	*-	$A * -B$ is equivalent to $A * 10^{-B}$
6	*/	$A * / B$ is equivalent to $A * 2^B$
5	*	Arithmetic product
5	/	Arithmetic quotient
5	//	Covered quotient
4	+	Arithmetic sum
4	-	Arithmetic difference
3	**	Logical product (AND)
2	++	Logical sum (OR)
2	--	Logical difference (EXCLUSIVE OR)
1	=	EQUALS conditional $A=B$ has value 1 if true; 0 if not true
1	>	GREATER THAN conditional $A > B$ has value 1 if true; 0 if not true
1	<	LESS THAN conditional $A < B$ has value 1 if true; 0 if not true

TABLE 11-2. DATA MODES FOR OPERATOR ITEMS

First Item	Operator	Second Item	Result
Binary or floating	< , , >	Binary or floating	Binary (0 or 1)
Binary or floating	++, --, **	Binary or floating	Binary
Binary	+, -, *, /	Binary	Binary
Floating	+, -, *, /	Binary	Floating
Binary	+, -, *, /	Floating	Floating
Floating	+, -, *, /	Floating	Floating
Binary or floating	*+, *-	Binary ^①	Floating
Binary	*	Binary ^①	Binary
Floating	*	Binary ^①	Scaled Binary

① The second item for these operators must be a binary value; if not, the Expression Error (E) flag is set.

11.8.1.1 Arithmetic Operators

This class of operator includes the familiar arithmetic operations, such as addition, multiplication, exponentiation and so forth.

11.8.1.1.1 *+ (Positive Exponentiation). This operator multiplies the element on the left of the operator by 10 raised to the power indicated by the element on the right (for example, A^{*+B} is equivalent to $A \cdot 10^{+B}$). The result is floating point number.

11.8.1.1.2 *- (Negative Exponentiation). This operator is similar to the positive exponential, except that the left-hand element is multiplied by 1/10 raised to the power indicated by the right-hand element (for example, A^{*-B} is equivalent to $A \cdot 10^{-B}$ or $A \cdot (1/10)^{+B}$).

Example

+49.5*-5	. PRODUCES A FLOATING POINT
	. WORD
+321.0*-2	. PRODUCES A FLOATING POINT
	. WORD
-49.5*+5	. PRODUCES A FLOATING POINT
	. WORD

11.8.1.1.3 */ (Binary Exponentiation or Scaling). This operator multiplies the value on the left-hand side of the operator by 2, raised to the power indicated by the element on the right. If the left-hand element is a binary value, the effect of this operator is to shift the value left, if the right-hand element is positive; and to shift the value right, if the right-hand element is negative. If the left-hand element is a floating point value, the result of the */ operation is a scaled binary value. If the right-hand element is a single-precision value, the result will be a single-precision binary value with up to 32 significant bits. If the right-hand element is a double-precision value, the result is a double-precision value with 32 significant bits.

Example

+077*/5	. PRODUCES 00000003740
+040000*/-3	. PRODUCES 00000004000
+4.5*/9	. PRODUCES 00000004400

11.8.1.1.4 * (Arithmetic Product). Multiplication of two values is indicated by separating them with an asterisk (*). The value on the left is multiplied by the value on the right.

Example

	-2*3	. PRODUCES -6
CAT	EQU	4
M	EQU	40
	+CAT*M	. PRODUCES 0240

11.8.1.1.5 / (Arithmetic Quotient). This operator divides the value on the left by the value on the right. Any remainder resulting from division is ignored.

Example

	+6/2	.	PRODUCES	3
	+7/3	.	PRODUCES	2
	-14/2	.	PRODUCES	-7
AREA	EQU	.	4096	
ITEM	EQU	.	512	
	+AREA/ITEM	.	PRODUCES	010

11.8.1.1.6 // (Covered Quotient). The covered quotient operator divides the value on the left by the value on the right, and adds +1 to the resulting quotient if the division resulted in a non-zero remainder.

Example

	+6//2	.	PRODUCES	3
	+7//3	.	PRODUCES	3

11.8.1.1.7 + (Arithmetic Sum) This operator adds the value on the left to the value on the right.

11.8.1.1.8 - (Arithmetic Difference) This operator subtracts the value on the right from the value on the left.

Example

	+6+4	.	PRODUCES	012
	+6-4	.	PRODUCES	2
	+07+077	.	PRODUCES	0106
	+8+3	.	PRODUCES A VALUE EQUAL TO	
		.	THE CONTENTS OF THE CURRENT	
		.	ADDRESS COUNTER + 3	

11.8.1.2 Logical Operators

This class of operator includes the logical product (**), logical sum (++), and logical difference (--).

11.8.1.2.1 ** (Logical Product). The logical product operator forms the bit-by-bit product (AND function) of the two values to the right and left of the operator; for example, if (and only if) the two values both have a 1 in the same bit position, the result contains a 1 in that bit position.

Example

+0374**0642	.	PRODUCES	0240
+54**39	.	PRODUCES	046

11.8.1.2.2 ++ (Logical Sum) The logical sum (INCLUSIVE OR function) sets a 1 in each bit position of the result if either or both of the values preceding and following the operator have a 1 in the corresponding bit position.

Example

+0252++0525	.	PRODUCES	0777
+0400++35	.	PRODUCES	0443

11.8.1.2.3 -- (Logical Difference). The logical difference operator (EXCLUSIVE OR function) sets each bit of the result to a 1 if either, but not both, of the two values on the right and left sides of the operator has the corresponding bit set.

Example

+4--3	.	PRODUCES	7
+4--7	.	PRODUCES	3

11.8.1.3 Conditional Operators

This class of operator always produces a resultant value of 1 or 0, depending on whether or not the condition expressed by the particular operator is satisfied.

11.8.1.3.1 = (Equal). The equal operator results in a binary value of 1 if the value on the left is equal to the value on the right; and a value of 0 if the two values are not equal.

Example

```
+A=1 . PRODUCES A VALUE OF 1 IF  
. A HAS SOMEWHERE BEEN DEFINED AS  
. BEING EQUAL TO 1  
+4*9*(B=3) . PRODUCES 044  
. IF B HAS BEEN DEFINED WITH A VALUE  
. OF 3. OTHERWISE, THE RESULT IS ZERO
```

11.8.1.3.2 > (Greater Than). The greater than operator results in a value of 1 if the quantity on the left is strictly greater than the quantity on the right; otherwise, the resultant value is 0.

Example

```
B EQU 3  
A EQU 2*B  
+A>B . PRODUCES 1  
. IN THIS EXAMPLE  
+5+(BIT>1) . PRODUCES 06  
. IF BIT HAS BEEN DEFINED AS BEING  
. GREATER THAN +1. OTHERWISE THE  
. EXPRESSION PRODUCES 5
```

11.8.1.3.3 < (Less Than). The less than operator compares the value on the left with the value on the right and sets the result equal to a 1 if the left-hand quantity is less than the quantity on the right; if it is not less, the resultant value is 0.

Example

+A<7 . PRODUCES 1 IF A
 . IS LESS THAN 7 AND ZERO IF
 . A IS EQUAL TO OR GREATER THAN 7
 +52*(B<4) . PRODUCES 064 IF
 . B IS LESS THAN 4. IF NOT,
 . THE RESULT IS ZERO

11.8.2 Operator Priorities

In the absence of parentheses, individual operator priorities determine the order in which elements are combined within an expression. Evaluation is performed left to right if two or more operators with the same priority occur within the same expression.

Table 11-2 shows that $*$, $+$, $-$, and $/$ have the highest priority (6), and $=$, $>$, and $<$ the lowest (1). Thus, within a given expression, elements separated by one of the three exponential operators are combined before any one operation is performed. Comparison of values within an expression is done last (such as is done when any one of the three conditional operators is encountered).

Example

+18*2-24/6*2+53 . 1
 . LINE 1 PRODUCES 0121
 . NOTE THAT LINE 1 IS EQUIVALENT
 . TO: +((18*2))-(((24/6)*2))+53
 A EQU 5
 +A*2/3<6*4**7 . 2
 . LINE 2 PRODUCES ZERO
 -A*2/3<6*4**7 . 3
 . LINE 3 PRODUCES 1
 +1+2*3-4++3+6-077 . 4
 . LINE 4 PRODUCES -064
 . AND IS EQUIVALENT TO LINE 5 BELOW
 +(1+(2*3)-4++(3+6-077)) . 5

11.8.3 Parenthetical Grouping

The normal sequence of evaluation within an expression may be altered by using parenthetical grouping; that is, enclosing certain portions of the expression within parentheses to override the normal rules of evaluation. The effect is to evaluate that portion of the expression within the parentheses as though it were the only expression on the line, independent of any elements or operators occurring outside of the parentheses. The value resulting from this evaluation is then substituted as an element in the entire expression, and the evaluation is continued.

Up to five levels of parenthetical grouping are allowed. When encountered, the expression within the innermost set of parentheses is evaluated first.

11.8.4 Relocatability

The final value resulting from the evaluation of an expression may be a memory address reference and, as such, may be potentially relocatable; for example, the memory address may be modified at load time. General rules which govern whether the result of an expression shall be relocatable or not are:

- a. A floating point value is never relocatable.
- b. The value resulting from a logical or conditional operation is not relocatable.
- c. Only the + and - arithmetic operators yield a result that is relocatable. The one exception to this rule is multiplication or division of a relocatable quantity by +1.

Tables 11-3 and 11-4 summarize these rules for relocatability.

TABLE 11-3. RELOCATION OF BINARY ITEMS

First Item	Operator	Second Item	Result
Binary or floating	< , = , >	Binary or floating	Not relocatable
Binary or floating	++,--, **	Binary or floating	Not relocatable
Not relocatable	+, -	Not relocatable	Not relocatable
Relocatable (binary)	+, -	Not relocatable (binary)	Relocatable
Not relocatable (binary)	+, -	Relocatable (binary)	Relocatable
Relocatable	+, -	Relocatable	Not relocatable
Binary or floating	*, /, // ^①	Binary or floating	Not relocatable
Binary or floating	*+, *-, */	Binary	Not relocatable

① Multiplication or division of a relocatable item by +1 results in a relocatable value.

TABLE 11-4. SINGLE AND DOUBLE PRECISION EXPRESSIONS

First Item	Operator	Second Item	Result
Single Double	< , = , >	Single or double Single or double	Single Single
Single Single Double Double	(+, -, ++, --, *, /, //)	Single Double Single Double	Single Double Double Double
Single Single Double Double	*+, *-	Single Double Single Double	Double Double Double Double
Single Single Double Double	*/	Single Double Single Double	Single Double Double Double

11.9 ASSEMBLER OUTPUTS

The Assembler produces only relocatable object output. The location of the object program then becomes a load-time determination. Assembler object code format is described in Volume I, Section 3 as part of the inputs to the Object Code Loader.

Along with the relocatable object code, the Assembler passes to the Loader the number of errors that occurred during the assembly (see Volume 1, paragraph 2.2.3 for loading). The number passed to the Loader does not include the number of T errors or R errors (see paragraphs 11.9.2.5 and 11.9.2.6).

Concurrent with the source language translation and output of the loadable machine code, the Assembler produces a side-by-side assembled program hardcopy listing.

11.9.1 Side-By-Side Listing

This side-by-side listing contains: 1) a sequential decimal source language statement number; 2) each source language statement together with any generated addresses; 3) object code resulting from translation of the statement; 4) Assembler-detected translation errors and warnings included along with the statement in error. By means of Assembler directives, the programmer can suppress all or any part of the side-by-side listing. Another Assembler directive permits the programmer to control the editing of the generated machine code into logically discrete fields.

11.9.2 Error Codes

11.9.2.1 Expression (E)

Expression errors result from illogical expressions such as a decimal digit within an octal number; element type inconsistent with arithmetic operators; expression improper in context, such as a GO line used outside a macro or a DO count in excess of $2^{16}-1$, or unequal number of left and right parentheses.

11.9.2.2 Duplicate (D)

Duplicate errors result from labels defined more than once with different values. A label used in an expression affecting an address counter is not defined prior to its use resulting in a different addressing sequence in the first and second assembly passes.

11.9.2.3 Undefined (U)

An undefined error results from any of the following three conditions:

1. A reference made to a label which was not defined in the program.
2. A reference made to a label that was not externalized properly by a call on a macro.
3. A failure to suffix labels of macro entry points with an adequate number of asterisks.

11.9.2.4 Instruction(I)

An Instruction error results when the Assembler encounters:

- a. A MACRO or EQU directive which has no label.
- b. A SEGEND within a MACRO.
- c. More than one coded subfield in field zero of a MACRO reference line called via a MACRO name.
- d. A nested LIB directive or a LIB directive within a MACRO.
- e. A LIBS directive retrieved from a library.

11.9.2.5 Relocation (R)

A Relocation error results from an arithmetic or logical operation being performed on a relocatable value which destroyed its relocatability.

11.9.2.6 Truncation (T)

A Truncation error occurs when the final value of an expression does not fit in the destined bit field of an object word, resulting in the Assembler truncating the left-most bits of the value in order to make it fit the field.

11.9.2.7 Overflow (O)

The Overflow error occurs when memory available for the Assembler tables is exhausted.

11.9.2.8 Name (N)

A Name error occurs when the Assembler encounters a name which contains more than eight characters.

11.9.2.9 Level (L)

A Level error results from an expression containing a parentheses nested more than five levels or from more than 64 SETADR lines appearing in this assembly or from an incomplete MACRO definition retrieved from a library.

11.9.2.10 Floating Point (F)

A Floating Point error occurs under any of three circumstances:

1. The divisor in a requested floating point divide operation is zero.
2. A floating point operation during evaluation of an expression yielded characteristic underflow. Characteristic underflow occurs whenever the characteristic is less than -32767.
3. A floating point operation during evaluation of an expression resulted in characteristic overflow. Characteristic overflow occurs whenever the characteristic exceeds +32767.

11.9.2.11 Warning (W)

A warning results when a label is used with a half-word instruction which is assigned to the lower half of a computer word.

11.9.3 Generation Formats

Assuming the editing listing option has been selected, a number of generated words, values, or half-words appear. They are:

S FIELD CODED EXPLICITLY. -

000120 14 0 0 0 0 0 00265 AA A0, CLASIV, , , S0

S FIELD NOT CODED (SY COMBINED). -

000205 52 0 0 0 0 000211 LBJ B7, FETCH

FULL-WORD (ON-LINE CONSTANTS OF EQU VALUES). -

001140 37700000377
00000000001 A1 EQU 1

HALF-WORD INSTRUCTIONS. -

000123 71 0 2 0 0	HAN	A0, A0
000123 61 11 0 1	HLCI	010+B1, A0

11.9.4 Listing Of Labels

The Assembler provides programmers via the LLT directive, with an alphanumerically sorted listing of source statement labels and corresponding Assembler-generated addresses or values. Listed labels are those defined at levels 0 and 1. LLT labels do not include MACRO, NAME, FORM, and LIT line labels.

11.9.4.1 Level 0

Level 0 labels are those names which are made available to another program at load time to some other independently processed code.

11.9.4.2 Level 1

Level 1 labels are those names encountered on the main program level.

11.9.4.3 LLT Sample Listing

A sample of the listing produced by the Assembler after processing a LLT directive appears below:

* LIST LABEL TABLE *

LEVEL 0

A0	0000000000	A1	0000000001	A2	0000000002
A3	0000000003	A4	0000000004	A7	0000000007
B1	0000000001	B2	0000000002	B3	0000000003
B4	0000000004	B5	0000000005	B6	0000000006
B7	0000000007	BCOUNT	0000000002	BEG	0000000007
BEGIN1	0000062 00	BUFF	0000150 00	BUFLAG	0000125 00

LEVEL 1

ADD	0000353 00	AGAIN1	0000275 00	AGAIN2	0000402 00
BWAIT	0000347 00	BWAIT0	0000262 00	BWAIT2	0000355 00
CBUFF	0000544 00	CDIV	0000521 00	CHARCK	0000444 00

11.9.4.4 Undefined Labels

Names or symbols encountered during the assembly which are not defined are listed:

UNDEFINED SYMBOLS PRDIR

11.9.4.5 Cross Reference Listing

A cross reference listing can be produced at the end of an assembly if an LCR directive has been previously encountered by the Assembler. This listing consists of labels, address counter values, and address counters where these labels are referenced. A sample listing is shown in figure 11-5.

11.10 ASSEMBLER DIAGNOSTICS AND STATUS

Several categories of errors encountered during the assembly process cause an error message to be output to the standard hardcopy device. These errors fall into one of four classes: 1) Assembly; 2) Assembler Internal; 3) Library call; and 4) Peripheral.

11.10.1 Assembly Errors

If an assembly error occurs during the assembly process, the following status messages will appear on the standard hardcopy devices:

ASSEMBLY ERRORS

XX YY

XX YY

Explanation

XX Any one of the following:
O Overflow
U Undefined
D Duplicate
E Expression
T Truncation
R Relocation
I Instruction
L Level
N Name
F Floating-Point

YY The number of errors of type XX that occurred in the assembly.

LIST CROSS REFERENCE TABLE

L1	000005	00	L1	000004	01	L1	000000	05	L1	000000	07	L1	000000	15
L1	000014	00	L2	000006	00	L2	000007	00	L2	000000	01	L2	000000	05
L2	000001	07	L2	000001	15	L2	000015	00	L3	000010	00	L3	000001	01
L3	000002	05	L3	000002	07	L3	000002	15	L3	000016	00	L4	000011	00
L4	000012	00	L4	000002	01	L4	000003	05	L4	000003	07	L4	000003	15
L4	000017	00	L5	000013	00	L5	000003	01	L5	000004	05	L5	000004	07
L5	000004	15	L5	000020	00									

Figure 11-5. Sample Cross-Reference Listing

ILLOGICAL SOURCE INPUT SEQUENCE

This message occurs when the Assembler has read a sentinel statement. This only occurs when an illogical sequence of source cards is input to the Assembler.

NOT ENOUGH UNASSIGNED TAPES

The Assembler is unable to obtain enough tapes to assemble with the requested output options.

11.10.2 Assembler Internal Errors

11.10.2.1 Core Overflow

The core overflow error occurs when the Assembler item table macro sample storage area, memory intermediate storage area, or literal origin stack overflows. No recovery is possible.

11.10.2.2 Level Overflow

The level overflow error occurs when the number of nested macros exceeds 29. No recovery is possible.

11.10.3 Library Call Errors

The following message is typed if the Assembler is unable to locate a called source library element on the assigned library medium:

```
**** LIB REF ERROR S name (vers)
```

Explanation

S Denotes source library.

Name (vers) The called library element name (version).

After the typeout, the Assembler continues without soliciting a response from the operator.

11.10.4 Peripheral Errors

The following peripheral error indications are received by the Assembler from the Centralized I/O Program or the Standard Input Program, and no recovery

is possible (the MN field in the messages is the name of the device involved in the error):

I/O ERR MN END OF FILE

A tape mark has been read while reading from the input.

I/O ERR MN UNREC ERR

A tape error has occurred and recovery procedure was unsuccessful. The tape function was retried five times without success.

I/O ERR MN ILLEG PROC

An illogical function has been requested (such as a magnetic tape read forward when the tape is positioned at the end of tape, or a pass backward function while the tape is positioned at the beginning of tape).

I/O ERR MN STRG END

End of tape was detected during a magnetic tape function or while using core memory as intermediate storage, or there was not enough core memory available to store the source program. In the second case, MN in the message will be MM.

I/O ERR MN NOT ASSIGNED

An I/O function was requested on a logical unit which had been removed from the current equipment configuration.

11.11 SOURCE DECK ORGANIZATION

Figures 11-6 through 11-10 illustrate example structures of source decks.

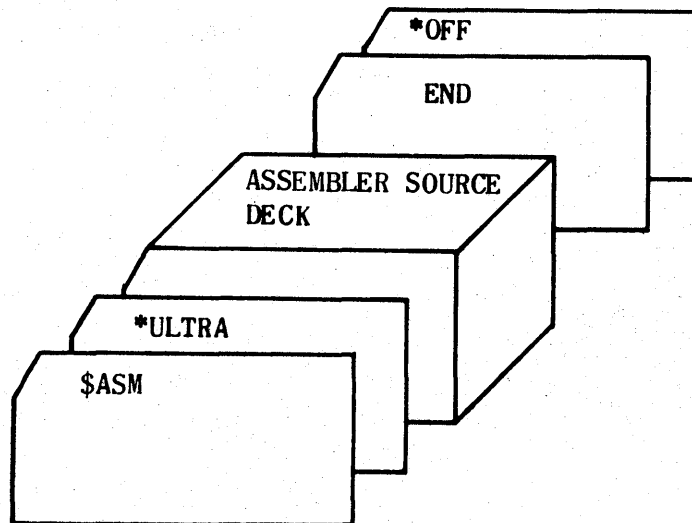


Figure 11-6. Source Deck Organization for a Single Program

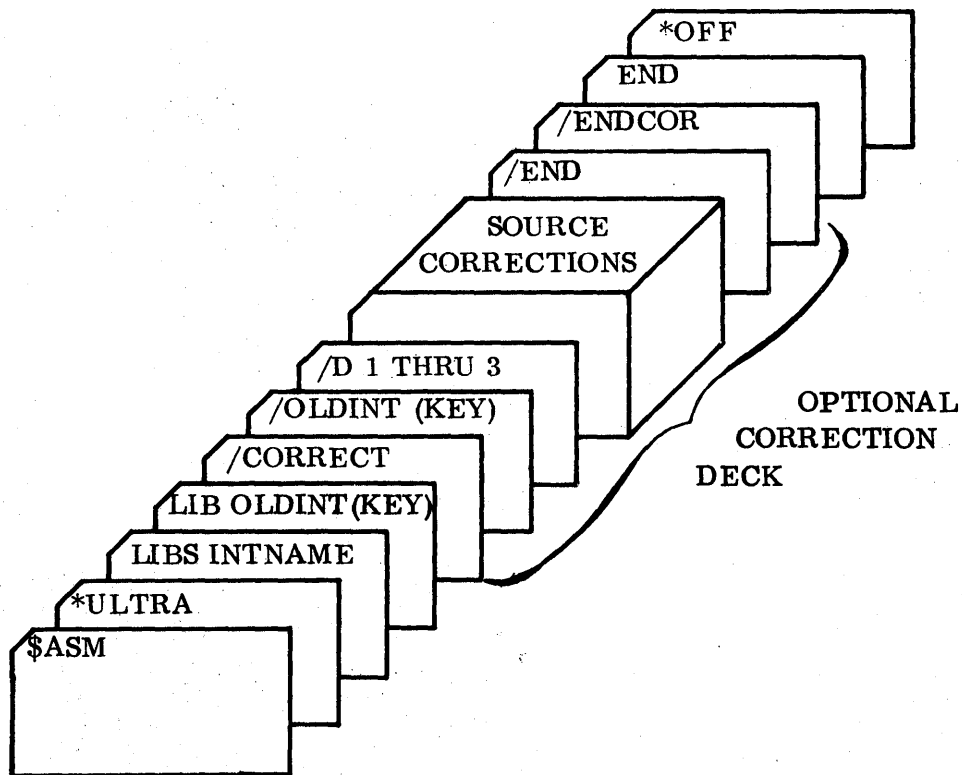


Figure 11-7. Source Deck Organization for Assembling Using Library Input.

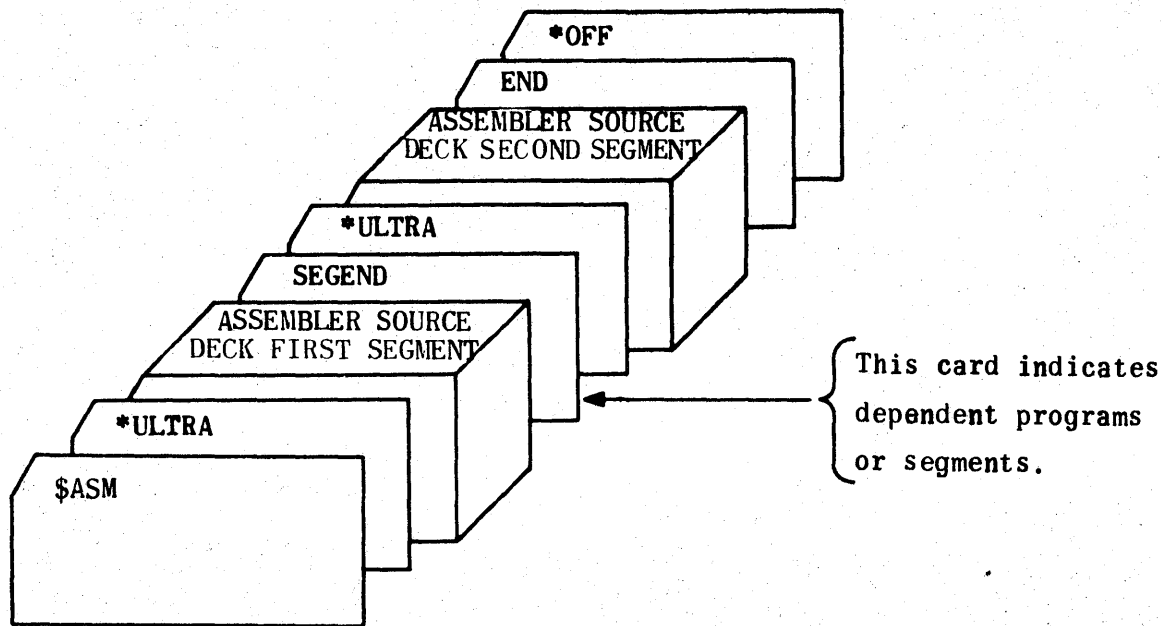


Figure 11-8. Source Deck Organization for Two or More Dependent Programs or Segments

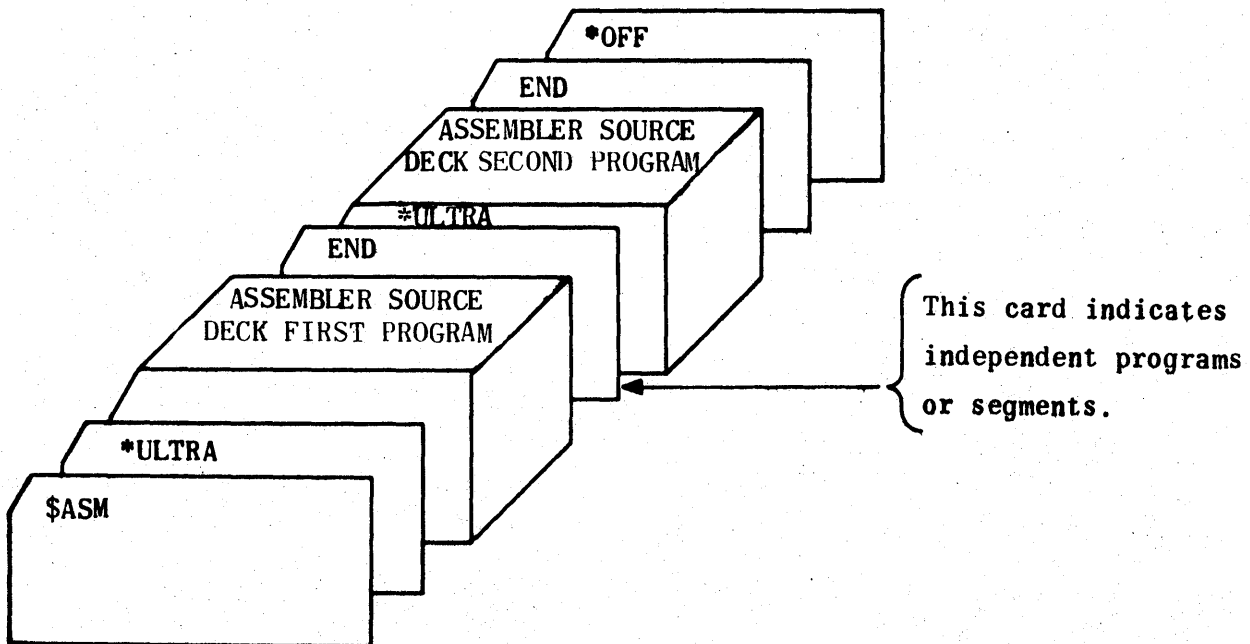


Figure 11-9. Source Deck Organization for Two or More Independent Programs or Segments

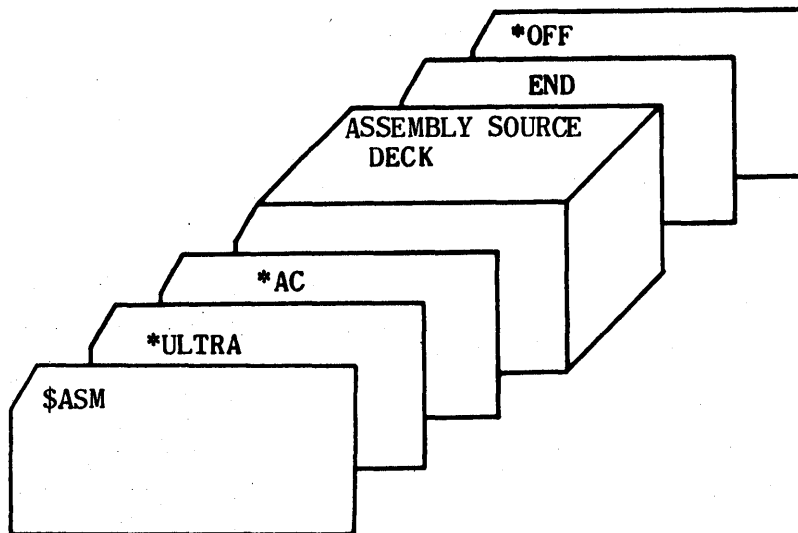


Figure 11-10. Source Deck Assembly Time Allocation

11.12 SPECIAL CONSIDERATIONS

- a. Only the first *ULTRA Assembler control statement has any effect on the mode of intermediate storage or on the name recorded on the object code program ID images. Subsequent *ULTRA statements are passed over by the Assembler.
- b. When coding for the CMS-2 Assembler, it is desirable to keep label lines (lines containing only a label) separate from coded machine instructions. This allows shifting the label line without altering any other line(s).
- c. If a number of successive half-word (16-bit) instructions are encountered (under the same address counter) by the Assembler, they are packed two per word. The first half-word encountered after the address counter has been activated/reactivated is packed into the upper half of the generated word.

- d. Generation of half-word in-line constants or literals is always in the lower half of the word with no packing.
- e. If LLT and LCR directives have been encountered and no reference is made to a relocatable label within the assembly, the label will be flagged with NR.
- f. The assembler call \$ASM,U indicates the ULTRA/32 keypunch code is to be used when interpreting source statements for the assembly. If the U is not present, the keypunch code specified in the \$JOB command is in effect.

Note: ULTRA/32 keypunch code is not interchangeable with the 026/029 keypunch codes. See special character codes for options in Appendix A.

SECTION 12

INSTRUCTION REPERTOIRE

12.1 AN/UYK-7 COMPUTER FUNCTIONS

The UYK-7 is a general purpose, multi-state, multi-processor, multi-I/O processor, stored-program computer. Some of the features of this computer relevant to a programmer are:

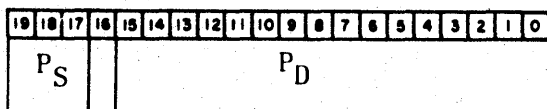
- a) High-speed memory with a cycle time of 1.5 microseconds and a capacity of 16,384 words expandable with more memory banks to 262,144 words.
- b) Memory banks containing 16,384 words each that can be addressed by the arithmetic processor(s), the I/O processor(s), and external devices with proper hardware adaptation.
- c) A portion of memory is non-destructive readout (NDRO) for storage of critical instructions and constants. This storage provides the facility for automatic recovery in case of a system failure or program fault and for automatic initial loading of programs.
- d) A 32-bit word length allowing for storage of one full-word instruction or two packed half-word instructions.
- e) Ability to address a 32-bit whole-word, 16-bit half-word, or 8-bit quarter-word with no difference in execution time.
- f) Use of parallel, one's complement, subtractive arithmetic.
- g) Use of single address instructions with the provision for address modification via seven index registers and eight base registers.
- h) Floating point and double prevision fixed point arithmetic functions.
- i) Memory protection, in segments of up to 65,536 words, under both program and manual control.
- j) An indirect addressing capability.
- k) Any field of a word addressing capability.
- l) Provision for connecting a remote operating console.

- m) A manual/program addressable breakpoint register which may be set to stop or interrupt program operation at any point.
- n) Interrupt and task states each with their own associated registers.
- o) A status register which contains information concerning the current status of a processor.
- p) Interrupt status or definition code capability.
- q) Provision for half-word instructions.
- r) Privileged instructions which can only be executed in the interrupt state.
- s) Eight arithmetic accumulators provided to allow parallel and cumulative computation.
- t) A processor capable of retrieving the current operand and the next instruction in parallel if located in different memory banks.
- u) Five different types of instruction formats.
- v) A processor decremental monitor clock, an IOC incremental realtime clock, and an IOC decremental monitor clock.

12.1.1 Register Format and Usage

12.1.1.1 Program Address Register

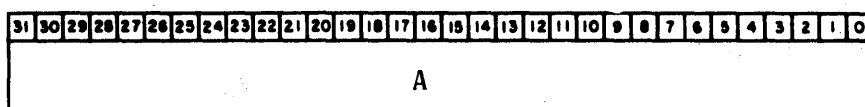
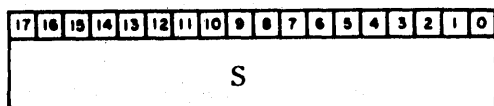
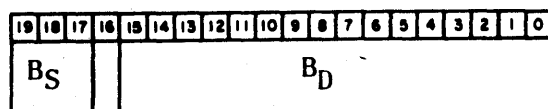
The program address register (P) holds the address of instructions to be executed by the computer. This register holds 19 useable bits. Bits 19 through 17 (P_S) hold a base register designator, 0 through 7, while bits 15 through 0 (P_D) contain a displacement value, relative to the address contained in the designated base register. Bit 16 is not used.



P Register Format

12.1.1.2 Addressable Registers, Control Memory

Table 12-1 gives the control memory address assignments for the central processor.

Accumulator register formatBase register formatIndex register format

Where:

B_S = base register designator.

B_D = displacement value.

When actually used for indexing, only the 16 bits of B_D are used. Bit 16 is not used.

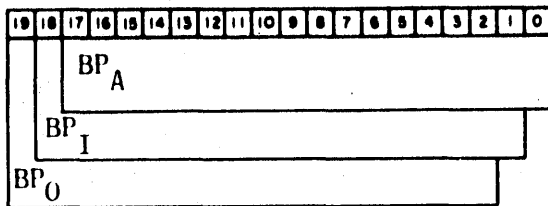
TABLE 12-1. CENTRAL PROCESSOR CONTROL MEMORY ADDRESS ASSIGNMENTS

CMR Address	Register Selected	Register Size
0-7*	Task Accumulators (Registers 0-7)	32 bits each
10*	Unassigned (Addressable)	19 bits
11-17*	Task Index (Registers 1-7)	19 bits each
20-27*	Task Base (Register 0-7, Addressable in Interrupt Mode Only)	18 bits each
30-57*	Unassigned (Not Useable)	
6X**	Breakpoint	20 bits
7X**	Active Status	23 bits
100-107**	Interrupt Accumulators (Registers 0-7)	32 bits
110**	Central Processor Monitor Clock	19 bits (clock only 16 bits)
111-117**	Interrupt Index (Registers 1-7)	19 bits each
120-127**	Interrupt Base (Registers 0-7)	18 bits each
130-137**	Unassigned (Not Useable)	
140-157**	DSW and ICW	20 bits each
160-167**	Storage Protection Registers (Registers 0-7)	21 bits each
170-177**	Segment Identification Registers (Registers 0-7)	21 bits each

* Task mode CMR address.

** Interrupt mode CMR address.

Breakpoint register format



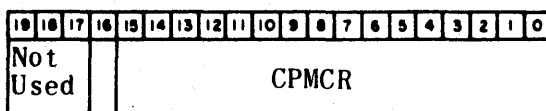
BP_I = when set, compares BP_A with instruction memory address.

BP_O = when set, compares BP_A with operand memory address.

If both BP_O and BP_I are set, both operations are performed.

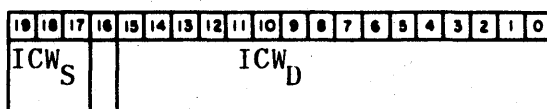
BP_A = an absolute address, up to 18 bits.

Central processor monitor clock register (CPMCR)



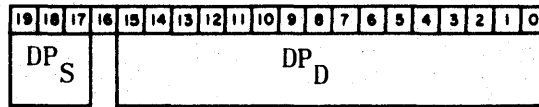
The CPMCR, when activated, is decremented at the rate of 1024 counts per second. A class II interrupt is generated when its value changes from zero to a negative value by hardware decrementation. If bit 15 is set, the clock is deactivated.

Initial condition word register (ICW)

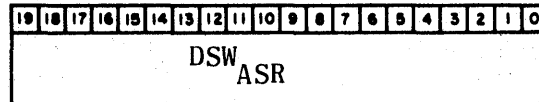


The four ICW registers contain entrance addresses for the four interrupt classes. ICW_S contains a base register designator and bits 15 through 0 (ICW_D) contain a displacement value.

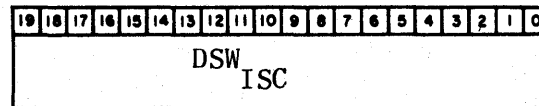
Designator storage word registers (DSW)



DSW (P Register)



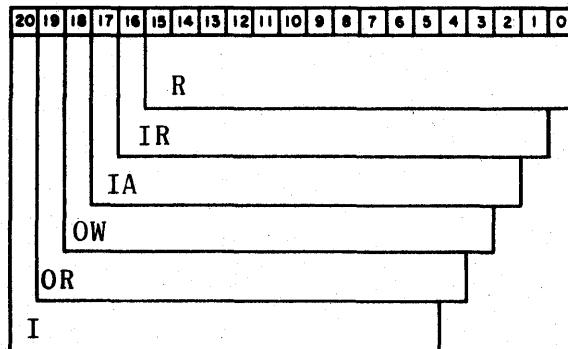
DSW (Active Status Register)



DSW (Interrupt Status Code)

The twelve DSW registers are in sets of three; each set is used for one of the four interrupt classes. During an interrupt sequence, P_S of the P register is stored in DP_S, and P_D of the P register is stored in DP_D. Bits 19 through 0 of the Active Status Register are stored in DSW_{ASR}, while the Interrupt Status Code is stored in DSW_{ISC}.

Storage protection registers (SPR)



The eight SPR registers correspond to the eight task base registers. R is a displacement value, defining a segment of memory with a starting address contained in the corresponding task base register. A final address is this address plus the value in R. This segment is then the only memory area accessible by the corresponding base register. The allowable types of operations within such a segment are defined by bits 20 through 16. The following operation is allowed when these bits are set:

Bit 16 (IR) - Use of interrupt index and base registers in indirect addressing.

Bit 17 (IA) - Use of indirect addressing.

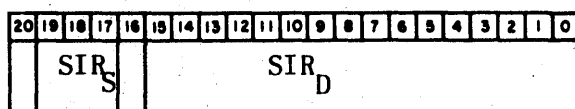
Bit 18 (OW) - Operand writing.

Bit 19 (OR) - Operand reading.

Bit 20 (I) - Instruction execution.

The SPRs are loaded by executing the LOAD BASE and MEMORY PROTECTION instruction (05 4). The lower 21 bits of the contents of the operand address +1 are loaded into an SPR.

Segment identification registers (SIR)



The eight SIR registers correspond to the eight task base registers.

SIR_S : base register designator.

SIR_D : a displacement value.

These registers are loaded with the effective operand address of the LOAD BASE and MEMORY PROTECTION instruction (05 4) which loads the corresponding base register.

s field \rightarrow SIR_S; y + (B_b)₁₅₋₀ \rightarrow SIR_D

12.1.2 Modes of Operation

The AN/UYK-7 Computer can be operated in one of two modes of operation:

1) the executive (or interrupt) state or 2) the task state. Both states can be manually or program initiated. Four bits of the active status register specify the state the computer is in according to Table 12-2.

TABLE 12-2. AN/UYK-7 COMPUTER MODES OF OPERATION

Active Status Register Bits				Processor State
2^{19}	2^{18}	2^{17}	2^{16}	
0	0	0	0	Task state
0	0	0	1	Executive state
0	0	1	0	Interrupt class III state
0	1	0	0	Interrupt class II state
1	0	0	0	Interrupt class I state

12.1.2.1 Interrupt State

In the interrupt state, the computer has the following characteristics:

- a) The computer can reference any memory word which has been locked out.
- b) The computer uses the interrupt set of accumulators, index registers, and base registers.
- c) The computer can come to a stop condition.

12.1.2.2 Task State

In the task state, the computer has the following characteristics:

- a) The computer can reference any memory word which has not been locked out.
- b) The computer uses the task set of accumulators, index registers, and base registers.

- c) The interrupt state set of registers cannot be referenced.
- d) Privileged instructions will not be executed.
- e) The task set of base registers can be modified only under certain conditions.

12.1.2.3 Active Status Register

The active status register (see Table 12-3) is a 23-bit register (one for each processor) showing the current environment relative to that processor at any moment.

TABLE 12-3. ACTIVE STATUS REGISTER

Bit Number	Designator	
22-20	CP identifier	
19	State I	} Under Hardware - see Table 3-2 Control
18	State II	
17	State III	
16	State IV	
15	Upper/lower	
14	Class I lockout	
13	Class II lockout	
12	Class III lockout	
11	Base(s) register selector	
10	Accumulator and index register selector	
9	Memory lockout inhibit	
8	Load base enable	
7	Bootstrap mode	
6-4	Not allocated	
3	Fixed point overflow indicator	
2	} Compare designators	
1		
0		

Upper/lower control - bit 15

The upper/lower control bit is set when an upper half-word instruction has completed execution and is cleared when a whole word or a lower half-word instruction has completed execution.

Class I lockout - bit 14

Locks out class I interrupts when set.

Class II lockout - bit 13

Locks out class II interrupts when set.

Class III lockout - bit 12

Locks out class III interrupts when set.

Accumulator/index register selector - bit 10

This selector is set in the interrupt mode and cleared in the task mode to select which set of A and B registers the active program may access.

Special base register selector - bit 11

This selector is set when entering the interrupt mode and cleared in the task mode to select which set of base registers the active program may access.

Memory lockout inhibit - bit 9

The 1-bit memory lockout inhibit (bit 9) is set (inhibit mode) in the interrupt mode and cleared (memory lockouts used) in the task state.

Load base enable - bit 8

Allows use of Load Base and Memory Protection instructions in task state when set.

Bootstrap mode - bit 7

Set manually to enable access to NDRO memory during bootstrap load. This bit is cleared under program control by execution of the following instructions: Interrupt Return, or Enter Executive State.

Spare bits - bits 4-6

Bits 4 through 6 are programmable spare bits.

Fixed point overflow indicator - bit 3

Displays the status of fixed point overflow (is set if overflow occurred). This bit is tested and cleared under program control by execution of the following instructions: Jump on Overflow, and Jump on No Overflow.

Compare designator - bits 0-2

The compare designator (bits 2,1, 0) display the status of the compare instructions as specified in the descriptions of individual instructions. The status word designation is as follows:

- a) Bit 2 = 0 unequal case
 = 1 equal case
- b) Bit 1 = 0 less than
 = 1 greater than
- c) Bit 0 = 0 within limits
 = 1 outside limits

12.2 AN/UYK-7 INSTRUCTION FORMATS

There are five different types of instruction formats: three are full-word formats and two are half-word formats. Half-word instructions do not have any memory reference parts because these instructions deal mainly with data manipulations between various registers.

The AN/UYK-7 Computer is a self-modifying, one-address computer. Although one reference or address is provided for the execution of an instruction, this reference or address can be modified automatically during a programmed sequence. The references are modified by using the index registers and the base registers which contain previously stored constants. The final operand address is the result of adding together the 18-bit content of the selected base register plus the 16-bit content of the selected index register, and the 13 bits of the immediate operand field of the instruction.

An instruction or data address is coded using octal notation with each octal digit denoting three binary digits. The instructions are read sequentially from memory except after jump instructions or interrupt situations. In these cases, the sequential execution of instructions resumes at another location in memory.

Each of the instructions in the repertoire is assigned to a format class according to the operational characteristics of the instructions. There are three full-word (32-bit) formats (Formats I, II, and III) and two half-word (16-bit) formats (Formats IV-A and IV-B). The paragraphs which follow specify the formats and the type of instructions assigned to each.

12.2.1 Format I Instructions

The instructions of format I have the following word format:

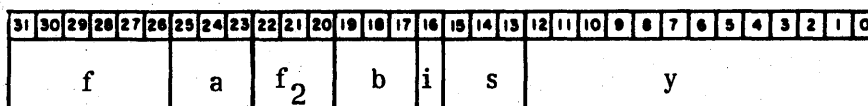
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
f			a			k			b			i			s			y													

These instructions basically require two operands: 1) an arithmetic or index register specified by the 3-bit a-field and 2) an operand address y-designator. The effective operand address y is formed by adding the content of an index register specified by the 3-bit b-field, the 13-bit displacement y-value and the content of a base register specified by the 3-bit s-field. The 3-bit k-field is used to control operand interpretation (see paragraph 12.3.3). The one-bit i-field of the instructions is used to specify indirect addressing. The f-field (6-bits) is the major function code. There are no subfunction codes in this format class.

There is a group of Format I instructions in which the a- and k-designators are interpreted as a combined unit. In these cases, whole-word operands are assigned and ak specifies either a control memory address or a bit position within a computer word.

12.2.2 Format II Instructions

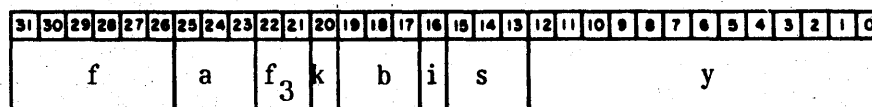
Format II instructions have the following format:



Whole word operands ($k = 3$) are hardware assumed in Format II instructions. Thus, the k-fields are used as a subfunction code labeled f_2 . The other fields of the Format II instructions are used in the same manner as in Format I.

12.2.3 Format III Instructions

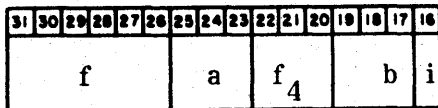
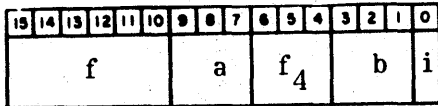
Format III instructions have the following word format:



Again only whole-word operands are specified or assumed in Format III instructions. Thus, the k-field is divided into two parts: 1) a 2-bit subfunction code labeled f_3 and 2) a single bit labeled k for which zero is the only legal entry. The other fields of Format III instructions are used in the same manner as for Format I.

12.2.4 Format IV-A Instructions

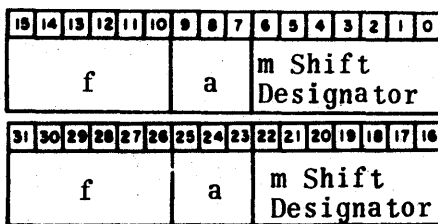
Format IV-A instructions have the following word format:



The instructions of Format IV-A require two registers (index or arithmetic) and are used for register-to-register transfers and arithmetic operations. The 3-bit a-field and the 3-bit b-field specify one of eight accumulators or eight index registers respectively. The 3-bit k-field is used as a subfunction code labeled f₄. Except as specified in the individual instructions, the i-field of the instruction is unused in Format IV-A instructions.

12.2.5 Format IV-B Instructions

Format IV-B instructions have the following word format:



The Format IV-B instructions use the 7-bit m-field to specify a shift count. the 3-bit a-field is used to specify one of the eight accumulators whose data is to be shifted for shift instructions.

The m-field value is interpreted as follows: if the upper bit of the m-field is set and bit 5 of the m-field is cleared, the shift count is contained in the B-register specified by bits 1 through 3; if the upper two bits of the m-field are set, the shift count is given in the A-register specified by bits 1 through 3. If neither of these cases apply (upper-most bit of m equals zero), then the shift count is contained in the lower 6 bits of the m-field (maximum shift permitted is 63 places).

12.2.6 Indirect Word

If $i = 1$, the 20 least significant bit positions of an instruction (b, i, s, and y fields) are replaced with the 20 least significant bit positions of (Y). The 12 higher order bit positions of (Y) are used to specify indirect addressing options. The interpretation of the indirect control word is:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
c		w				p				b		i		s		y															

Explanation

c	Control designator specifying the type of addressing that will occur.
c=10	For indirect addressing only.
c=01 ₂	For single character addressing.
c=11 ₂	For sequential character addressing.
c=00, bit 29=0:	Indirect addressing where bits 17-19 indicate the base register and bits 0-15 the 16-bit displacement.
c=00; bit 29=1:	Indirect addressing where bits 17-19 indicate the index register and bits 0-15 the 16-bit displacement.

Indirect addressing shall continue as long as $i = 1$ with indexing capability at each cascaded level. When $i = 0$, the indirect addressing will terminate and the current instruction will be interpreted in the normal manner.

When $i = 0$ and $c = (01_2, 11_2)$, the remaining ten positions of the indirect control word are interpreted and character addressing will occur. In this case, the p- and w-designators are interpreted as follows:

- p - bit position designator which specifies the least significant bit position of the variable-length character field.
- w - character length designator which specifies the number of bits of the character field based at p.

When a character has been read from memory by character addressing, it is placed in the appropriate arithmetic register, right-justified, and zero-filled. If $c = 11_2$, sequential character addressing is specified. The indirect control word is updated for subsequent addressing of the next character field and then stored back in main memory.

12.2.7 I/O Commands Formats

The IOC will read the command from memory and begin its execution upon receipt of the command address from the central processor. The I/O format is:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
f				k		j			m		c		y																		

Explanation

- f Function code of the command.
- k Partial word designator.
- j Channel number (0-15).
- c Chain flag.
- m Monitor flag.
- y Absolute 18-bit address of the operand (buffer control words, external function words, etc.).

The buffer control words specify the limits of the buffer of data for input/output and the desired mode of transfer (a word at a time, half-word at a time, and so forth). Buffer control words are in two formats: normal mode and ESI mode.

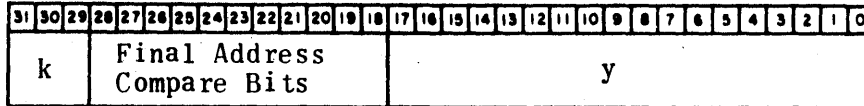
12.2.7.1 Normal Mode

The normal mode format is:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Final Address Compare Bits													y																		

12.2.7.2 ESI Mode

The ESI mode format is:

Explanation

y	Initial 18-bit buffer address.
Final address compare bits	Initial buffer address plus the buffer's length minus 1 truncated to the required number of bits to fit the field.
k	Partial word designator.

12.3 SYMBOLIC CONVENTIONS

For symbols, registers, and terms used in the computer instruction descriptions, see Table 12-14.

12.3.1 f - Function Code Designator

The f-designator always occupies the most significant 6 bits of the instruction. It specifies or determines the type of instruction to be performed. All unused major function codes are illegal instructions and, if executed in the upper half-word, will cause an illegal instruction interrupt.

12.3.2 a - Arithmetic Code Designator

The 3-bit a-designator specifies which one of eight A-registers an instruction will use or reference. The accumulator designation is as follows:

- a) $000_2 \rightarrow A_0$
- b) $001_2 \rightarrow A_1$
- c) $010_2 \rightarrow A_2$
- d) $011_2 \rightarrow A_3$
- e) $100_2 \rightarrow A_4$
- f) $101_2 \rightarrow A_5$

M-5035

g) $110_2 \rightarrow A_6$

h) $111_2 \rightarrow A_7$

NOTE

There are two groups of accumulators.
One group is associated with the task
state and the other with the interrupt
state of computer operation.

TABLE 12-4. INSTRUCTION REPERTOIRE SYMBOL DEFINITIONS

Symbol	Definition
()	Content or the quantity
()'	Complement of the quantity
	Absolute value
:	Compare
x	Multiply
÷	Divide
-	Minus
+	Plus
=	Equal
≠	Not equal
>	Greater than
≥	Greater than or equal
<	Less than
≤	Less than or equal
⊙	Logical AND or logical product defined as:
	$\begin{array}{c cc} & 0 & 1 \\ \hline 0 & 0 & 0 \\ 1 & 0 & 1 \end{array}$
⊕	Inclusive OR or logical sum defined as:
	$\begin{array}{c cc} & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 1 & 1 \end{array}$
⊖	Exclusive OR or logical difference defined as:
	$\begin{array}{c cc} & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 1 & 0 \end{array}$
a	Instruction field designating an accumulator or index register.
A _a	Accumulator designated by the a-field.
A _b	Accumulator designated by the b-field.

TABLE 12-4. INSTRUCTION REPERTOIRE SYMBOL DEFINITIONS (continued)

Symbol	Definition
af_4	Instruction field designating the combined a- and f_4 -fields use to specify a control memory register.
ak	Instruction field designating the combined a- and k-fields used to specify a bit position or a control memory register.
b	Instruction field designating an index or accumulator register.
B_a	16-bit index register designated by the a-field.
B_b	16-bit index register designated by the b-field.
c	Instruction field designating indirect addressing word type or the chain flag in input/output controller commands.
C	Input/output channel.
CA	Capable of indirect word character addressing.
Cj	Input/output channel designated by the j-field.
CD	Hardware compare designator.
CMR	Control memory register.
DSW	Designator storage words.
e	Operand field of the HK pseudo instruction.
EF	External function.
EI	External interrupt.
f	Instruction field designating the major function code.
f_2, f_3, f_4	Instruction fields designating sub-function code.
i	Instruction field designating indirect addressing or interrupt control memory addressing.

TABLE 12-4. INSTRUCTION REPERTOIRE SYMBOL DEFINITIONS (continued)

Symbol	Definition
I/O	Input/output.
IOC	Input/output controller.
j	Instruction field designating an I/O channel.
k	Instruction field partial word designator.
kj	Instruction field designating the combined k- and j- fields used to specify a bit position or control memory register in IOC instructions.
l	Operand subfield of buffer control word pseudo instructions.
m	Instruction field designating a shift count or monitor flag in an IOC command.
n	Used as a subscript indicating a bit position; for example, $(A_a)_n$.
NI	Next instruction.
OD	Overflow designator.
p	Indirect word bit position designator.
P	Program address register, 20 bits.
PI	Privileged instruction executable only when the processor is in the interrupt (executive) state.
RPT	Capable of being executed in the repeat mode.
RTC	Real-time clock.
s	Instruction field designating a base register.
SIR	Segment identification register.
SPR	Stored protection register.

TABLE 12-4. INSTRUCTION REPERTOIRE SYMBOL DEFINITIONS (continued)

Symbol	Definition
sy	Instruction field representing s- and y-fields in combination.
U	U register (program control register).
w	Indirect word character length designator.
y	Instruction operand field designating an address or value.
Y	Effective address formed by $Y + (B_d) + (S_s)$.
<u>Y</u>	Effective operand as qualified by k (and/or p and w when applicable).

12.3.3 k - Operand Interpretation Code Designator

The k-designator determines what part of the word referenced by an instruction is to be used. When $k \neq 0$, bits 15 through 13 specify a base register. If the k-designator specifies an upper or lower half-word, the 16-bit operand will be sign-extended to 32 bits. If the k-designator specifies a quarter-word, the 8-bit quarter-word will be zero extended to 32 bits. If the k-designator specifies the whole word, the full 32-bit word will be used. The general interpretation of the k-designator is specified in Tables 12-5, 12-6, and 12-7.

TABLES 12-5. GENERAL OPERAND INTERPRETATION (MEMORY TO ARITHMETIC)

k Designator	Memory to Arithmetic	
k = 0	$sy+(B_b) \rightarrow A_{15-0}$	Sign extended
k = 1	$(Y_{15-0}) \rightarrow A_{15-0}$	Sign extended
k = 2	$(Y_{31-16}) \rightarrow A_{15-0}$	Sign extended
k = 3	$(Y_{31-0}) \rightarrow A_{31-0}$	
k = 4	$(Y_{7-0}) \rightarrow A_{7-0}$	Zeros extended
k = 5	$(Y_{15-8}) \rightarrow A_{7-0}$	Zeros extended
k = 6	$(Y_{23-16}) \rightarrow A_{7-0}$	Zeros extended
k = 7	$(Y_{31-24}) \rightarrow A_{7-0}$	Zeros extended

TABLE 12-6. GENERAL OPERAND INTERPRETATION (ARITHMETIC TO MEMORY)

k Designator	Arithmetic to Memory	
k = 0	Not used	
k = 1	$(A_{15-0}) \rightarrow Y_{15-0}; Y_{31-16}$	\rightarrow Unchanged
k = 2	$(A_{15-0}) \rightarrow Y_{31-16}; Y_{15-0}$	\rightarrow Unchanged
k = 3	$(A_{31-0}) \rightarrow Y_{31-0}$	
k = 4	$(A_{7-0}) \rightarrow Y_{7-0}; Y_{31-8}$	\rightarrow Unchanged
k = 5	$(A_{7-0}) \rightarrow Y_{15-8}; Y_{31-16}$	\rightarrow Unchanged
	Y_{7-0}	\rightarrow Unchanged
k = 6	$(A_{7-0}) \rightarrow Y_{23-16}; Y_{31-24}$	\rightarrow Unchanged
	Y_{15-0}	\rightarrow Unchanged
k = 7	$(A_{7-0}) \rightarrow Y_{31-24}; Y_{23-0}$	\rightarrow Unchanged

TABLE 12-7. GENERAL OPERAND INTERPRETATION (NORMAL REPLACE INSTRUCTION INTERPRETATION)

Normal Replace Instruction Interpretation	
k = 0	Not used
k = 1	<p>Read: $(Y_{15-0}) \rightarrow A_{15-0}$ (sign extended).</p> <p>Store: Store the lower 16 bits of the operand in bits 15-0 of address Y leaving the upper 16 bits of the contents of address Y unchanged.</p>
k = 2	<p>Read: $(Y_{31-16}) \rightarrow A_{15-0}$ (sign extended).</p> <p>Store: Store the lower 16 bits of the operand in bits 31-16 of address Y leaving the lower 16 bits of the contents of address Y unchanged.</p>
k = 3	<p>Read: $(Y_{31-0}) \rightarrow A_{31-0}$</p> <p>Store: Store the 32 bit operand at address Y.</p>
k = 4	<p>Read: $(Y_{7-0}) \rightarrow A_{7-0}$ (zero extended).</p> <p>Store: Store the lower 8 bits of the operand in bits 7-0 of address Y, leaving the upper 24 bits of the contents of address Y unchanged.</p>
k = 5	<p>Read: $(Y_{15-8}) \rightarrow A_{7-0}$ (zero extended).</p> <p>Store: Store the lower 8 bits of the operand in bits 15-8 of address Y, leaving the remaining bits of the contents of address Y unchanged.</p>
k = 6	<p>Read: $(Y_{23-16}) \rightarrow A_{7-0}$ (zero extended).</p> <p>Store: Store the lower 8 bits of the operand in bits 23-16 of address Y, leaving the remaining bits of the contents of address Y unchanged.</p>
k = 7	<p>Read: $(Y_{31-24}) \rightarrow A_{7-0}$ (zero extended).</p> <p>Store: Store the lower 8 bits of the operand in bits 31-24 of the operand in bits 31-24 of address Y, leaving the lower 24 bits of the contents of address Y unchanged.</p>

12.3.4 b - Index Register Code Designator

The 3-bit b-designator specifies which of the index registers will be used to modify the operand address y-designator. B-registers are generally used for indexing loops in a program. In addition, the B7-register serves as a repeat counter and the B0-register indicates a register which always contains zero. There is a group of index registers for each state of the computer.

12.3.5 i - Indirect Address Code Designator

The i-designator is set by coding an asterisk (*) before the y field. The i-designator of the instruction word controls the use of indirect addressing and variable-length character addressing during execution. If $i = 0$, the instruction will function normally.

12.3.6 s - Base Register Code Designator

The 3-bit s-designator is used to modify the 13-bit operand address y-designator to form $Y = y + B_b + S_s$. The base registers addressing technique is as follows:

a) $000_2 \rightarrow S_0$	(Add (S_0) to $y + (B_b)$)
b) $001_2 \rightarrow S_1$	(Add (S_1) to $y + (B_b)$)
c) $010_2 \rightarrow S_2$	(Add (S_2) to $y + (B_b)$)
d) $011_2 \rightarrow S_3$	(Add (S_3) to $y + (B_b)$)
e) $100_2 \rightarrow S_4$	(Add (S_4) to $y + (B_b)$)
f) $101_2 \rightarrow S_5$	(Add (S_5) to $y + (B_b)$)
g) $110_2 \rightarrow S_6$	(Add (S_6) to $y + (B_b)$)
h) $111_2 \rightarrow S_7$	(Add (S_7) to $y + (B_b)$)

There are two sets of eight base registers, one for each of the task and interrupt states of the processor(s).

12.3.7 y - Operand Code Designator

The operand y-designator is either a 13-bit value (zero extended) if $k \neq 0$, or a 16-bit value (sign extended) when $k = 0$. In the first case, the y-designator is part of the final operand address; in the other case it is a constant.

12.3.8 f₂, f₃, f₄ - Subfunction Code Designators

In an effort to minimize the number of different function codes, various instructions have subfunctions in all or part of the normal k-designator field. In place of the k-field, a whole-word interpretation of the operand is hardware assumed.

12.3.9 m - Shift Counter Field

This is a 7-bit field used in half-word shift instructions to specify the number of data bits in an A-register that will be moved either to the right or to the left.

12.3.10 m - Monitor Interrupt Code Designator

This is a special designator for input/output controller instructions consisting of a 1-bit monitor flag that, if set, will cause the IOC to transmit an interrupt to the processor when a buffer is terminated.

12.3.11 c - Chain Flag Code Designator

This is a special designator for input/output controller instructions consisting of a 1-bit chain flag that, if set, indicates to the IOC that another command follows. When the operation specified by this command terminates, the chain shall remain active.

12.3.12 j - Channel Number

This is a special designator for input/output controller instructions consisting of a 4-bit field specifying which channel (0 through 15) the associated IOC command is to be performed on.

12.4 COMPUTER-INSTRUCTION REPERTOIRE

The assembler recognizes and generates for an inbedded computer-instruction repertoire associated with the AN/UYK-7 Computer System. All mnemonic computer-instruction lines have the general format:

Label	Mnemonic Function	Operand
-------	-------------------	---------

Use of a label is always optional. An asterisk (*) preceding the y field specifies indirect addressing and causes the Assembler to set the i-field of the generated instruction word to a one.

For convenience of programming, the coding sequence of the operand subfield(s) does not necessarily correspond to the field order of the generated instruction.

The material contained in this section presents the instruction repertoire in condensed form for reference purposes only. Programmers requiring more detailed information concerning the hardware operation should consult the appropriate hardware specification document.

Note that the mnemonic function code for all half-word instructions begins with the letter H.

For ease of reference, the instructions are grouped according to their function into eleven major categories as follows: Load and Store, Arithmetic, Jump, Comparison, Logical, Shifts, Control Memory References, Interrupt Handling, Miscellaneous, Extension Mnemonics and Input/Output.

Within each group, instructions are arranged alphabetically by name. Format I and Format II instructions are character addressable and repeatable except as indicated. There is some unavoidable overlap in the above classification. For example, some of the logical type instructions involve arithmetic operations. The list which follows may be consulted for corresponding names, mnemonics, and groups.

<u>MNEMONIC</u>	<u>NAME</u>	<u>GROUP</u>
AA	ADD A	Arithmetic
AB	ADD B	Arithmetic
AEI	ALLOW ENABLE INTERRUPT	Interrupt
AFC	ACTIVATE EXTERNAL FUNCTION CHAIN ON C _j	Input/Output
AIC	ACTIVATE INPUT CHAIN ON C _j	Input/Output
ALP	ADD LOGICAL PRODUCT	Logical
ANA	SUBTRACT A (Add Negative A)	Arithmetic
ANB	SUBTRACT B (Add Negative B)	Arithmetic
AOC	ACTIVATE OUTPUT CHAIN ON C _j	Input/Output
AXC	ACTIVATE EXTERNAL INTER- RUPT CHAIN ON C _j	Input/Output
BC	COMPARE BIT TO ZERO	Comparison
BCW	BUFFER CONTROL WORD	Extension
BCWE	BUFFER CONTROL WORD ESI	Extension
BS	SET BIT	Miscellaneous
BZ	CLEAR BIT	Miscellaneous
C	COMPARE	Comparison
CG	COMPARE GATED	Comparison
CL	COMPARE LIMITS	Comparison
CM	COMPARE MASKED	Comparison
CNT	COUNT ONES	Miscellaneous
CXI	COMPARE INDEX, INCRE- MENTED	Comparison
D	DIVIDE A	Arithmetic
DA	DOUBLE ADD A	Arithmetic
DAN	DOUBLE SUBTRACT A (Double Add Negative A)	Arithmetic
DC	DOUBLE COMPARE	Comparison
DJNZ	DOUBLE JUMP A NOT ZERO	Jump

<u>MNEMONIC</u>	<u>NAME</u>	<u>GROUP</u>
DJZ	DOUBLE JUMP A ZERO	Jump
DL	DOUBLE LOAD A	Load & Store
DS	DOUBLE STORE A	Load & Store
FA	FLOATING POINT ADD	Arithmetic
FAN	FLOATING POINT SUBTRACT (Floating Point Add Negative)	Arithmetic
FANR	FLOATING POINT SUBTRACT WITH ROUND (Floating Point Add Negative with Round)	Arithmetic
FAR	FLOATING POINT ADD WITH ROUND	Arithmetic
FB	INITIATE EXTERNAL FUNC- TION BUFFER ON C _j	Input/Output
FD	FLOATING POINT DIVIDE	Arithmetic
FDR	FLOATING POINT DIVIDE WITH ROUND	Arithmetic
FM	FLOATING POINT MULTIPLY	Arithmetic
FMIR	SET EXTERNAL FUNCTION MONITOR INTERRUPT REQUEST ON C _j	Input/Output
FMR	FLOATING POINT MULTIPLY WITH ROUND	Arithmetic
HA	ADD (SUM)	Arithmetic
HAI	ALLOW CLASS II INTER- RUPTS	Input/Output
HALT	STOP PROCESSOR	Miscellaneous
HAN	SUBTRACT (DIFFERENCE) Add Negative (Differ- ence)	Arithmetic
HAND	AND	Logical
HC	COMPARE, REGISTER	Comparison
HCB	COMPARE B _b WITH B _a	Comparison
HCL	COMPARE LIMITS, REGISTER	Comparison
HCM	COMPARE MASKED, REGISTER	Comparison
HCP	COMPLEMENT A	Arithmetic

<u>MNEMONIC</u>	<u>NAME</u>	<u>GROUP</u>
HD	DIVIDE REGISTER	Arithmetic
HDCP	DOUBLE COMPLEMENT A	Arithmetic
HDLC	DOUBLE SHIFT LEFT CIRCULARLY	Shift
HDRS	DOUBLE SHIFT RIGHT FILL SIGN	Shift
HDRZ	DOUBLE SHIFT RIGHT FILL ZEROS	Shift
HDSF	DOUBLE SCALE FACTOR	Miscellaneous
HK	HALF-WORD CONSTANT	Extension
HLB	LOAD Ba WITH Bb	Load & Store
HLC	SHIFT LEFT CIRCULARY	Shift
HLCI	LOAD INTERRUPT CMR WITH A	Control Memory
HLCT	LOAD TASK CMR WITH A	Control Memory
HM	MULTIPLY REGISTER	Arithmetic
HNO	HALF WORD NO-OPERATION	Extension
HOR	INCLUSIVE OR A (Logical Sum)	Logical
HPI	PREVENT CLASS III INTERRUPTS	Interrupt
HRS	SHIFT RIGHT FILL SIGN	Shift
HRT	SQUARE ROOT	Arithmetic
HRZ	SHIFT RIGHT FILL ZEROS	Shift
HSCI	STORE INTERRUPT CMR IN A	Control Memory
HSCT	STORE TASK CMR IN A	Control Memory
HSF	SCALE FACTOR	Miscellaneous
HSIM	STORE I/O MONITOR CLOCK	Miscellaneous
HSTC	STORE REAL TIME CLOCK	Miscellaneous
HWFI	WAIT FOR INTERRUPT	Interrupt
HXOR	EXCLUSIVE OR A (Logical Difference)	Logical

<u>MNEMONIC</u>	<u>NAME</u>	<u>GROUP</u>
IB	INITIATE INPUT BUFFER ON Cj	Input/Output
IBS	SET BIT	Input/Output
IBZ	CLEAR BIT	Input/Output
ILTC	LOAD REAL-TIME CLOCK	Input/Output
IMIR	SET INPUT MONITOR INTER- RUPT REQUEST ON Cj	Input/Output
JBNZ	INDEX JUMP (Jump B not zero)	Jump
IO	INITIATE INPUT/OUTPUT	Miscellaneous
IPI	INTERPROCESSOR INTERRUPT	Interrupt
ITSF	TEST AND SET FLAG	Input/Output
IW	INDIRECT WORD	Extension
IWB	INDIRECT WORD, SPECIAL INDEX	Extension
IWC	INDIRECT WORD, CHARACTER	Extension
IWCI	INDIRECT WORD, CHARACTER INCREMENT	Extension
IWS	INDIRECT WORD, SPECIAL BASE	Extension
J	JUMP	Jump
JC	JUMP CONDITION SETTING	Jump
JE	JUMP EQUAL	Jump
JEP	JUMP EVEN PARITY	Jump
JG	JUMP GREATER THAN	Jump
JGE	JUMP GREATER THAN OR EQUAL	Jump
JIO	JUMP (INPUT/OUTPUT)	Input/Output
JL	JUMP LOWER	Jump
JLE	JUMP LESS THAN OR EQUAL	Jump
JLT	JUMP LESS THAN	Jump
JN	JUMP A NEGATIVE	Jump
JNE	JUMP NOT EQUAL	Jump
JNF	JUMP NO OVERFLOW	Jump

<u>MNEMONIC</u>	<u>NAME</u>	<u>GROUP</u>
JNW	JUMP NOT WITHIN LIMITS	Jump
JNZ	JUMP A NOT ZERO	Jump
JOF	JUMP OVERFLOW	Jump
JOP	JUMP ODD PARITY	Jump
JP	JUMP A POSITIVE	Jump
JS	JUMP $S_y + B$	Jump
JSC	JUMP STOP CONDITIONAL SETTING	Jump
JW	JUMP WITHIN LIMITS	Jump
JZ	JUMP A ZERO	Jump
LA	LOAD A	Load & Store
LB	LOAD B	Load & Store
LBJ	LOAD B AND JUMP	Jump
LBMP	LOAD BASE AND MEMORY PROTECTION	Load & Store
LCI	LOAD INTERRUPT CMR	Control Memory
LCT	LOAD TASK CMR	Control Memory
LDIF	LOAD DIFFERENCE	Load & Store
LICM	LOAD IOC CONTROL MEMORY	Input/Output
LIM	LOAD, ENABLE IOC MONITOR CLOCK	Miscellaneous
LLP	LOAD LOGICAL PRODUCT	Logical
LLPN	LOAD LOGICAL PRODUCT NEXT	Logical
LM	LOAD MAGNITUDE	Load & Store
LNA	LOAD NEGATIVE	Load & Store
LSUM	LOAD SUM	Load & Store
LXB	LOAD A & INDEX B	Load & Store
M	MULTIPLY A	Arithmetic
MP	MEMORY PROTECTION	Extension
MS	SELECTIVE SUBSTITUTE A	Logical
NLP	SUBTRACT LOGICAL PRODUCT (Add Negative Logical Product)	Logical

<u>MNEMONIC</u>	<u>NAME</u>	<u>GROUP</u>
NOOP	NO OPERATION	Extension
OB	INITIATE OUTPUT BUFFER	Input/Output
OMIR	SET OUTPUT MONITOR INTERRUPT REQUEST	Input/Output
OR	INCLUSIVE OR (SELECTIVE SET)	Logical
PEI	PREVENT ENABLE INTERRUPT	Interrupt
RA	REPLACE ADD	Arithmetic
RALP	REPLACE A + LOGICAL PRODUCT	Logical
RAN	REPLACE SUBTRACT (Replace Add Negative)	Arithmetic
RD	REPLACE DECREMENT	Arithmetic
RI	REPLACE INCREMENT	Arithmetic
RJ	RETURN JUMP	Jump
RJC	RETURN JUMP CONDITIONAL	Jump
RJSC	RETURN JUMP STOP CONDI- TIONAL SETTING	Jump
RLP	REPLACE LOGICAL PRODUCT	Logical
RMS	REPLACE MASK (SEL. SUB- STITUTE)	Logical
RNLP	REPLACE A - LOGICAL PRODUCT (Replace Add Neg- ative Logical Product)	Logical
ROR	REPLACE INCLUSIVE OR (Replace Selective Set)	Logical
RP	REPEAT	Miscellaneous
RSC	REPLACE SELECTIVE CLEAR	Logical
RXOR	REPLACE EXCLUSIVE OR (Replace Selective Com- plement)	Logical
SA	STORE A	Load & Store
SB	STORE B	Load & Store
SC	SELECTIVE CLEAR	Logical
SCI	STORE INTERRUPT CMR	Control Memory

<u>MNEMONIC</u>	<u>NAME</u>	<u>GROUP</u>
SCT	STORE TASK	Control Memory
SDIF	STORE DIFFERENCE	Load & Store
SICM	STORE IOC CONTROL MEMORY	Input/Output
SLP	STORE LOGICAL PRODUCT	Logical
SM	STORE MAGNITUDE	Load & Store
SNA	STORE NEGATIVE	Load & Store
SSUM	STORE SUM	Load & Store
SXB	STORE A AND INDEX B	Load & Store
SZ	STORE ZERO	Extension
TBS	TEST BIT SET	Input/Output
TBZ	TEST BIT CLEARED	Input/Output
TFB	TERMINATE EXTERNAL FUNCTION BUFFER ON Cj	Input/Output
TIB	TERMINATE INPUT BUFFER ON Cj	Input/Output
TOB	TERMINATE OUTPUT BUFFER ON Cj	Input/Output
TSF	TEST AND SET FLAG	Miscellaneous
TXB	TERMINATE EXTERNAL INTERRUPT BUFFER	Input/Output
XB	INITIATE EXTERNAL INTERRUPT BUFFER	Input/Output
XMIR	SET EXTERNAL INTERRUPT MONITOR INTERRUPT REQUEST ON Cj	Input/Output
XOR	EXCLUSIVE OR (SELECTIVE COMPLEMENT)	Logical
XR	EXECUTE REMOTE	Miscellaneous
XRL	EXECUTE REMOTE LOWER	Miscellaneous
XS	ENTER EXECUTIVE STATE	Miscellaneous
ZA	CLEAR A	Extension
ZB	CLEAR B	Extension

12.4.1 Load and Store Instructions

Generally, load instructions load a register or registers with: 1) the contents of memory; 2) the contents of memory plus or minus the contents of an accumulator register; or 3) the contents of memory in absolute magnitude or complemented. In format I instruction, k is the normal read designator. Store instructions store the following in memory: 1) the contents of a register or registers; 2) the sum or difference of consecutively numbered accumulator registers; or 3) an accumulator register in absolute magnitude or complemented.

The following are the formats for load and store instructions:

DOUBLE LOAD A

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
05					a		4		b		i		s		y																

DL a,y,b,s

$$(Y + 1, Y) \rightarrow A_{a+1}, A_a$$

Not character addressable.

Not repeatable.

Load the double length register (formed with the least significant half in A_a and the most significant half in A_{a+1}) with the content of the double length memory word, formed with the least significant half as (Y) and the most significant half as (Y + 1).

DOUBLE STORE A

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
02					a		7		b		i		s		y																

DS a,y,b,s

$$(A_{a+1}, A_a) \rightarrow Y + 1, Y$$

Not character addressable.

Not repeatable.

Store the content of the double length register (formed with the least significant half in A_a and the significant half in A_{a+1}) at the double length memory word (formed with Y as the least significant half and $Y + 1$ as the most significant half).

LOAD A

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
10												a					k			b		i		s		y									

LA a,y,k,b,s

$$\underline{Y} \rightarrow A_a$$

Load A_a with \underline{Y} .

LOAD A AND INDEX B

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
11												a					k			b		i		s		y									

LXB a,y,k,b,s

$$\underline{Y} \rightarrow A_a; (B_b) + 1 \rightarrow B_b$$

Not repeatable.

Load A_a with \underline{Y} and add one to (B_b) .

LOAD B

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
20												a					k			b		i		s		y									

LB a,y,k,b,s

$$\underline{Y} \rightarrow B_a$$

The a sub-field specifies an index register.

Load B_a with \underline{Y} . If B_a is B_0 , the effect is a no-operation.

LOAD BASE AND MEMORY PROTECTION

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
05				a				4		b		i		s		y															

LBMP a,y,b,s

$$\begin{aligned} \underline{Y} &\rightarrow S_a \\ \underline{Y + 1} &\rightarrow SPR_a \\ s &\rightarrow SIR_a \text{ 19-17} \\ y + (B_D)_b &\rightarrow SIR_a \text{ 15-0} \end{aligned}$$

Not character addressable.

Not repeatable.

This instruction loads the task state base register, specified by the a-field, with the lower 18 bits at address Y, and loads the Storage Protection Register specified by the a-field, with the lower 21 bits of address Y + 1. The Segment Identification Register, specified by the a-field, receives the s-field value in bits 19 through 17 and the sum of $y + (B_D)_b$ in bits 15 through 0. This instruction can be executed in the task state under the following conditions:

1. Bit 8 of the Active Status Register must be set.
2. The s-field must equal seven.
3. The a-field must not equal seven.

A violation of these conditions shall cause a privileged instruction interrupt (class II).

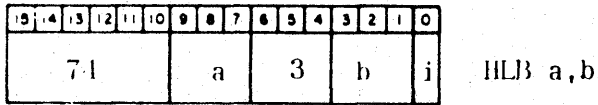
The following condition applies to this instruction in both the task and interrupt states:

The 17-bit relative address quantity, $y + (B_D)_b$ must be an even number.

Its violation shall be an illegal instruction error causing a class II interrupt.

M-5035
Change 1

LOAD B_a WITH B_b

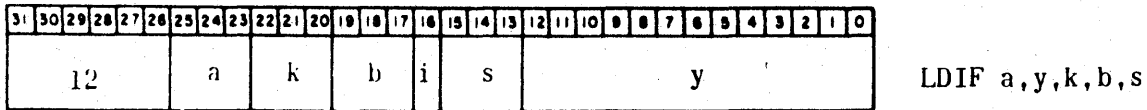


$$(B_b) \rightarrow B_a$$

The a sub-field specifies an index register.

Load B_a with (B_b) . If B_a is B_0 , the effect is a no-operation. If B_b is B_0 , B_a is cleared to zero.

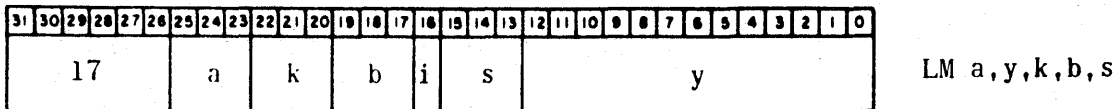
LOAD DIFFERENCE



$$\underline{Y} - (A_a) \rightarrow A_{a+1}; \quad (A_a)_i = (A_a)_f$$

Load A_{a+1} with the difference formed by subtracting (A_a) from \underline{Y} . (A_a) remains unchanged.

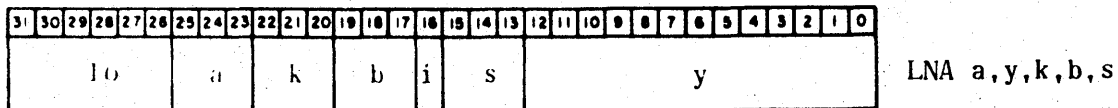
LOAD MAGNITUDE



$$|\underline{Y}| \rightarrow A_a$$

Load A_a with the absolute value of \underline{Y} .

LOAD NEGATIVE



$$\underline{Y}' \rightarrow A_a$$

Load A_a with the logical complement (ones complement) of \underline{Y} .

LOAD SUM

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
15						a	k	b	i	s	y																				

LSUM a,y,k,b,s

$$\underline{Y} + (A_a) \rightarrow A_{a+1}; (A_a)_i = (A_a)_f$$

Load A_{a+1} with the sum of \underline{Y} and (A_a) . (A_a) remains unchanged.

STORE A

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
24						a	k	b	i	s	y																				

SA a,y,k,b,s

$$(A_a) \rightarrow Y$$

Store (A_a) at Y.

STORE A AND INDEX B

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
25						a	k	b	i	s	y																				

SXB a,y,k,b,s

$$(A_a) \rightarrow Y; (B_b) + 1 \rightarrow B_b$$

Not repeatable.

Store (A_a) at Y; add one to (B_b) and store the sum in the B_b register.

STORE B

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
23						a	k	b	i	s	y																				

SB a,y,k,b,s

$$(B_a) \rightarrow Y$$

The a sub-field specifies an index register.

Store (B_a) at Y.

M-5035
Change 1

STORE DIFFERENCE

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
02				a	6	b	i	s	y										SDIF a,y,b,s												

$$(A_{a+1}) - (A_a) \rightarrow Y \text{ and } A_{a+1}; (A_a)_i = (A_a)_f$$

Subtract (A_a) from (A_{a+1}) . The result is noted at Y and A_{a+1} . (A_a) remains unchanged.

STORE MAGNITUDE

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
27				a	k	b	i	s	y										SM a,y,k,b,s												

$$|A_a| \rightarrow Y$$

Store the magnitude of (A_a) at Y.

STORE NEGATIVE

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
26				a	k	b	i	s	y										SNA a,y,k,b,s												

$$(A_a)' \rightarrow Y$$

Take the complement of (A_a) and store the result in Y.

STORE SUM

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
02				a	5	b	i	s	y										SSUM a,y,b,s												

$$(A_a) + (A_{a+1}) \rightarrow Y \text{ and } A_{a+1}; (A_a)_i = (A_a)_f$$

Add (A_a) to (A_{a+1}) . The result is stored in A_{a+1} and Y. (A_a) remains unchanged.

12.1.2 Arithmetic Instructions

These instructions change the contents of a specified register or registers according to the operation specified, except for the replace instructions which also modify the contents of memory. Some instructions in the LOAD AND STORE and LOGICAL sections also involve arithmetic operations.

ADD A

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1-1				a				k				b				i				s				y							

AA a,y,k,b,s

$$(A_a) + \underline{Y} \rightarrow A_a$$

Form the sum of (A_a) and \underline{Y}_b and store the result in A_a .

ADD (SUM)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
71				a				1				b				i			

HA a,b

$$(A_a) + (A_b) \rightarrow A_a; (A_b)_i = (A_b)_f$$

Form the sum of (A_a) and (A_b) and store the result in A_a .

ADD B

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
21				a				k				b				i				s				y							

AB a,y,k,b,s

$(B_a) + \underline{Y} \rightarrow B_a$, where reference is to an index register in sub-field a.
If B_a is B_0 , the effect is a no-operation. k is normal read.

COMPLEMENT A

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
70				a				2				b				i			

HCP a

$$(A_a)' \rightarrow A_a$$

Complement (A_a) and store the result in A_a .

M-5035
Change 1

DIVIDE A

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
41				a				k				b				i				s				y											

D a,y,k,b,s

$$(A_{a+1}, A_a) \div Y \rightarrow A_a; \text{ remainder} \rightarrow A_{a+1}$$

Divide the content of the double length register (formed with the least significant half in A_a and the most significant half in A_{a+1}) by Y. The quotient is stored in A_a and the remainder in A_{a+1} . Divide overflow shall occur if the quotient exceeds 31 data bits and one sign bit.

DIVIDE REGISTER

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
7-1				a				l				b				i			

HD a,b

$$(A_{a+1}, A_a) \div (A_b) \rightarrow A_a; \text{ remainder} \rightarrow A_{a+1}$$

Divide the double length register (formed with (A_a) as the least significant half and (A_{a+1}) as the most significant half) by (A_b) . The quotient is stored in A_a and the remainder in A_{a+1} . $(A_b)_i = (A_b)_f$ if $a \neq b$ and $a+1 \neq b$.

DOUBLE ADD A

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
05				a				l				b				i				s				y											

DA a,y,b,s

Not character addressable.

Not repeatable.

$$(Y+1, Y) + (A_{a+1}, A_a) \rightarrow A_{a+1}, A_a$$

Form the sum of the content of the double length register (formed with the least significant half in A_a and the most significant half in A_{a+1}) and the content of the double length memory word (formed with the least significant half at Y and the most significant half at Y + 1). Store the least significant half of the result in A_a and the most significant half in A_{a+1} .

DOUBLE COMPLEMENT A

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
70					a			3		b		i		H/CP a	

$$(A_{a+1}, A_a)' \rightarrow A_{a+1}, A$$

Complement the double length register (formed with (A_a) as the least significant half, and (A_{a+1}) as the most significant half) and store the most significant half of the result in A_{a+1} and the least significant half in A_a .

DOUBLE SUBTRACT A

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
05					a			2		b		i		s		y											DAN a,y,b,s				

Not character addressable.
Not repeatable.

$$(A_{a+1}, A_a) - (Y+1, Y) \rightarrow A_{a+1}, A_a$$

Subtract the content of the double length memory word (formed with the least significant half at Y and the most significant half at Y + 1) from the content of the double length register (formed with the least significant half in A_a and the most significant half in A_{a+1}). Store the least significant half of the result in A_a and the most significant half in A_{a+1} .

FLOATING POINT ADD

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
06					a			0		b		i		s		y											FA a,y,b,s				

Not character addressable.
Not repeatable.

Compare the characteristic stored at Y with the characteristic located in the lower 16 bits of A_a . The mantissa located at Y + 1 or the mantissa located in A_{a+1} is then shifted, depending on the comparison of the characteristics. (Y + 1) is then added to (A_{a+1}). The normalized shift count is then subtracted from, or added to the final characteristic located in A_a .

FLOATING POINT ADD WITH ROUND

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
06				a				4		b		i		s		y															

FAR a,y,b,s

Not character addressable.

Not repeatable.

Compare the characteristic stored at Y with the characteristic located in the lower 16 bits of A_a . The mantissa stored at Y + 1 or the mantissa located in A_{a+1} is shifted, depending on the comparison of the characteristics. The final mantissa is rounded as required. The normalized shift count is subtracted from, or added to, the final characteristic located in A_a .

FLOATING POINT DIVIDE

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
06				a				3		b		i		s		y															

FD a,y,b,s

Not character addressable.

Not repeatable.

Subtract the characteristic located at Y from the characteristic located in the lower 16 bits of A_a . The mantissa located in A_{a+1} is then divided by the mantissa located at Y + 1. The final mantissa located in A_{a+1} is normalized right one place if necessary and the characteristic adjusted accordingly.

FLOATING POINT DIVIDE WITH ROUND

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
06				a	7	b	i	s	y																						

FDR a,y,b,s

Not character addressable.

Not repeatable.

Subtract the characteristic located at Y from the characteristic located in the lower 16 bits of A_a . The mantissa located in A_{a+1} is then divided by the mantissa located at Y + 1. The mantissa located in A_{a+1} is normalized right one place if necessary and the characteristic adjusted accordingly. The final quotient mantissa, (A_{a+1}), is rounded as required.

FLOATING POINT MULTIPLY

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
06				a	2	b	i	s	y																						

FM a,y,l,s

Not character addressable.

Not repeatable.

Add the characteristic located at Y to the characteristic located in the lower 16 bits of A_a . The mantissa located in A_{a+1} is then multiplied by the mantissa located at Y + 1. The final mantissa located in A_{a+1} is normalized left one place if necessary and the characteristic adjusted accordingly.

FLOATING POINT MULTIPLY WITH ROUND

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
06				a	6	b	i	s	y																						

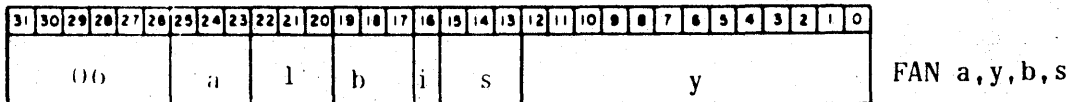
FMR a,y,b,s

Not character addressable.

Not repeatable.

Add the characteristic located at Y to the characteristic located in the lower 16 bits of A_a . The mantissa located in A_{a+1} is then multiplied by the mantissa located at Y + 1. The mantissa located in A_{a+1} is normalized left one place if necessary and the characteristic adjusted accordingly. The final mantissa, (A_{a+1}), is then rounded as required.

FLOATING POINT SUBTRACT

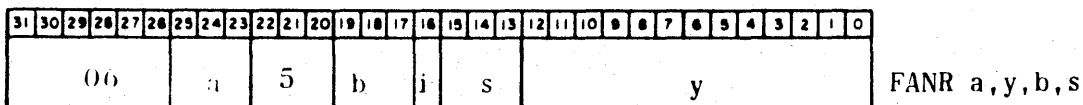


Not character addressable.

Not repeatable.

Compare the characteristic stored at Y with the characteristic located in the lower 16 bits of A_a . The mantissa located at Y + 1 or the mantissa located in A_{a+1} is then shifted depending on the comparison of the characteristics. (Y + 1) is then subtracted from the (A_{a+1}). The normalized shift count is then subtracted from, or added to, the final characteristic located in A_a .

FLOATING POINT SUBTRACT WITH ROUND



Not character addressable.

Not repeatable.

Compare the characteristic stored at Y with the characteristic located in the lower 16 bits of A_a . The mantissa located at Y + 1 or the mantissa located in A_{a+1} is shifted depending on the comparison of the characteristics. (Y + 1) is then subtracted from (A_{a+1}). The final mantissa is rounded as required. The normalized shift count is then subtracted from, or added to, the final characteristic located in A_a .

MULTIPLY A

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
40				a				k				b				i				s				y							

M a, y, k, b, s

$$(A_a) \times \underline{Y} \rightarrow A_{a+1}, A_a$$

Multiply (A_a) by \underline{Y} , store the least significant half of the double length result in A_a , and the most significant half in A_{a+1} .

MULTIPLY REGISTER

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
74				a				0				b				i HM a.b			

$$(A_a) \times (A_b) \rightarrow A_{a+1}, A_a$$

Multiply (A_a) by (A_b) . The least significant half of the double length result is stored in A_a , and the most significant half in A_{a+1} . $(A_b)_i = (A_b)_f$ if $a \neq b$ and $a+1 \neq b$.

REPLACE ADD

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
34				a				k				b				i				s				y							

RA a, y, k, b, s

$$\underline{Y} + (A_a) \rightarrow Y \text{ and } A_{a+1}; (A_a)_i = (A_a)_f$$

Form the sum of \underline{Y} and (A_a) , and store the result at Y and A_{a+1} . (A_a) remains unchanged.

REPLACE INCREMENT

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
35					a			k		b		i		s		y															

RI a,y,k,b,s

$$\underline{Y} + 1 \rightarrow A_a \text{ and } Y$$

Load A_a with \underline{Y} , then add one to (A_a) and store the result at A_a and Y .

REPLACE DECREMENT

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
37					a			k		b		i		s		y															

RD a,y,k,b,s

$$\underline{Y} - 1 \rightarrow A_a \text{ and } Y$$

Load A_a with \underline{Y} , then subtract one from (A_a) and store the result at A_a and Y .

REPLACE SUBTRACT

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
36					a			k		b		i		s		y															

RAN a,y,k,b,s

$$\underline{Y} - (A_a) \rightarrow Y \text{ and } A_{a+1}; (A_a)_i = (A_a)_f$$

Subtract (A_a) from \underline{Y} , then store the result in Y and A_{a+1} . (A_a) remains unchanged.

SQUARE ROOT

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
74					a			b		i									

HRT a,b

$$\sqrt{(A_{a+1}, A_a)} \rightarrow A_b; \text{ residue} \rightarrow A_{b+1}$$

Calculate the square root of the content of the double length register (formed with the most significant half in A_{a+1} and the least significant half in A_a). The result is stored in A_b and the residue is stored in A_{b+1} .

$$(A_{a+1})_j = (A_{a+1})_f \text{ if } a+1 \neq b \text{ and } a+1 \neq b+1$$

$$(A_a)_i = (A_a)_f \text{ if } a \neq b \text{ and } a \neq b+1$$

SUBTRACT (DIFFERENCE)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
71					a		2		b		i		EAN a,b		

$$(A_a) - (A_b) \rightarrow A_a$$

Subtract the (A_b) from (A_a) and store the result in A_a . $(A_b)_i = (A_b)_f$.

SUBTRACT A

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
13					a		k		b		i		s		y												ANA a,y,k,b,s				

$$(A_a) - \underline{Y} \rightarrow A_a$$

Subtract \underline{Y} from (A_a) and store the result in A_a .

SUBTRACT B

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
22					a		k		b		i		s		y												ANB a,y,k,b,s				

$$(B_a) - \underline{Y} \rightarrow B_a \quad a = \text{index register}$$

Subtract \underline{Y} from (B_a) and store the result in B_a . If B_a is B_0 , the effect is a no-operation.

12.4.3 Jump Instructions

There are 28 jump instructions and all are Format III. Zero is the only possible value in the K field of an assembled jump instruction, so the Assembler ignores any value in the k sub-field of a corresponding source line. In the descriptions below, Y is regarded as a relative address formed by adding $(B_b)_{15-0}$ to y zero extended. If the jump is taken, this quantity is transferred to $P_{(15-0)}$ and the s-designator is transferred to $P_{(19-17)}$. If the jump is not taken, the next instruction in sequence is executed.

JUMP

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
53			a			3			0	b			i			s			y												

J y,k,t,s

$Y \rightarrow P$

Jump unconditionally to address Y.

INDEX JUMP

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
52			a			1			0	b			i			s			y												

JBNZ a,y,k,b,s

If $(B_a) \neq 0$, then $(B_a) - 1 \rightarrow B_a$, and jump to Y. If $(B_a) = 0$, take NI.

If $B_a = B_0$, the effect is a no-operation.

DOUBLE JUMP A NOT ZERO

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
50			a			3			0	b			i			s			y												

DJNZ a,y,k,b,s

If $(A_{a+1}, A_a) \neq 0$, jump to Y.

where (A_{a+1}, A_a) is the contents of the double length register formed by A_{a+1} and A_a .

DOUBLE JUMP A ZERO

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
50					a	2	0	b	i	s	y																				

DJZ a,y,k,b,s

If $(A_{a+1}, A_a) = 0$, jump to Y.

where (A_{a+1}, A_a) is the contents of the double length register formed by A_{a+1} and A_a .

JUMP A NEGATIVE

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
51					a	1	0	b	i	s	y																				

JN a,y,k,b,s

If $(A_a) < 0$, jump to Y.

JUMP A NOT ZERO

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
51					a	3	0	b	i	s	y																				

JNZ a,y,k,b,s

If $(A_a) \neq 0$, jump to Y.

JUMP A POSITIVE

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
51					a	0	0	b	i	s	y																				

JP a,y,k,b,s

If $(A_a) \geq 0$, jump to Y.

JUMP A ZERO

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
51					a	2	0	b	i	s	y																				

JZ a,y,k,b,s

If $(A_a) = 0$, jump to Y.

M-5035

JUMP CONDITIONAL SETTING

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
53		a		30		b		i		s		y																			

JC a,y,k,b,s

If the value of the a sub-field, (1, 2, or 3) corresponds to a manual switch which has been selected, then jump to Y.

SPECIAL
a-VALUE

FUNCTION
PERFORMED

1
2
3

Jump if switch 1 is selected
Jump if switch 2 is selected
Jump if switch 3 is selected

JUMP CONDITIONAL STOP SETTING (PI)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
53		a		30		b		i		s		y																			

JSC a,y,k,b,s

If the value of the a sub-field (4,5,6, or 7) corresponds to a manual switch setting, then stop the processor, and transfer Y to P.

SPECIAL
a-VALUE

FUNCTION
PERFORMED

4
5
6
7

Always stop, then jump.
Stop and jump if switch 5 selected.
Stop and jump if switch 6 selected.
Stop and jump if switch 7 selected.

JUMP EVEN PARITY

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
50		a		00		b		i		s		y																			

JEP a,y,k,b,s

If the logical product of (A_a) and (A_{a+1}) contains an even number of binary ones, jump to Y.

JUMP EQUAL

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
53					5					0	b	i	s	y																	

JE y,k,b,s

If CD is set to equal, jump to Y.

JUMP GREATER THAN

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
53					11					0	b	i	s	y																	

JG y,k,b,s

If the CD is set to greater than, jump to Y.

JUMP GREATER THAN OR EQUAL

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
53					15					0	b	i	s	y																	

JGE y,k,b,s

If the CD is set to greater than or equal, jump to Y.

JUMP LESS THAN

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
53					21					0	b	i	s	y																	

JLT y,k,b,s

If CD is set to less than, jump to Y.

JUMP LESS THAN OR EQUAL

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
53					25					0	b	i	s	y																	

JLE y,k,b,s

If CD is set to less than or equal, jump to Y.

JUMP LOWER

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
52					a	3	0	b	i	s	y																				

JL y,k,b,s

$Y_L \rightarrow P;$

Jump to the 16-bit instruction contained in the lower half of the word whose address is Y.

M-5035

JUMP NO OVERFLOW

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
53			00			0	b	i	s	y																					

JNF y,k,b,s

If the OD is not set, jump to Y.

JUMP NOT EQUAL

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
53			01			0	b	i	s	y																					

JNE y,k,b,s

If the CD is set to not equal, jump to Y.

JUMP NOT WITHIN LIMITS

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
53			31			0	b	i	s	y																					

JNW y,k,b,s

If the CD is set to outside limits, jump to Y.

JUMP ODD PARITY

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
50			a	1	0	b	i	s	y																						

JOP a,y,k,b,s

If the logical product of (A_a) and (A_{a+1}) contains an odd number of binary ones, jump to Y.

JUMP OVERFLOW

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
50			04			0	b	i	s	y																					

JOF y,k,b,s

If the OD is set, clear OD, jump to Y.

JUMP SY + B

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
52					a		2		0		b		i		s		y														

JS s,y,k,b

Jump to Y, where Y in this case is defined as the address formed by the 16-bit sy-field and indexed by $(B_b)_{15-0}$. The base register designator, $(B_b)_{17-19}$, is transferred to P_{17-19} . An example of usage of this instruction is as an exit from a subroutine entered through the LOAD B AND JUMP instruction.

JUMP WITHIN LIMITS

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
53					35		0		b		i		s		y																

JW y,k,b,s

If the CD is within limits, jump to Y.

LOAD B AND JUMP

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
52					a		00		b		i		s		y																

LBJ a,y,k,b,s

The a sub-field specifies an index register. Load B_a with the contents of P (address of NI). $P_{19-17} \rightarrow B_a_{19-17}$, $P_{15-0} \rightarrow B_a_{15-0}$.

Jump to Y. If $B_a = B_0$, no address is saved and an unconditional jump to Y occurs.

RETURN JUMP

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
53					02		0		b		i		s		y																

RJ y,k,b,s

P, (address of NI) \rightarrow Y. Jump to Y+1.

RETURN JUMP CONDITIONAL SETTING

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
53				a	2	0	b	i	s	y																					

RJC a,y,k,b,s

If the value of the a sub-field (1,2, or 3) corresponds to a manual switch which has been selected, then execute the Return Jump, (see RETURN JUMP instruction).

<u>SPECIAL</u> <u>a-VALUE</u>	<u>FUNCTION</u> <u>PERFORMED</u>
1	Return jump if switch 1 is selected.
2	Return jump if switch 2 is selected.
3	Return jump if switch 3 is selected.

RETURN JUMP CONDITIONAL STOP SETTING (PI)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
53				a	2	0	b	i	s	y																					

RJSC a,y,k,b,s

If the value of the a sub-field (4,5,6,7) corresponds to a manual switch which has been selected, then stop before executing the Return Jump (see RETURN JUMP instruction).

<u>SPECIAL</u> <u>a-VALUE</u>	<u>FUNCTION</u> <u>PERFORMED</u>
4	Always stop before execution of Return Jump.
5	Stop if switch 5 is selected, Return Jump.
6	Stop if switch 6 is selected, Return Jump.
7	Stop if switch 7 is selected, Return Jump.

12.1.4 Instructions Involving Comparison Operations

All the instructions in this group set the compare designator according to conditions indicated. They are usually followed by one of the jump instructions which act upon the condition of the compare designator.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
44				a				k				b				i				s				y							

C a,y,k,b,s

(A_a): Y; set CD

Compare (A_a) with Y, and set the CD to:

Equal, if (A_a) = Y

Unequal, if (A_a) ≠ Y

Greater than or equal, if (A_a) ≥ Y

Less than, if (A_a) < Y

COMPARE, REGISTER

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
74				a				4				b				i			

HC a,b

(A_a) : (A_b); set CD

Compare (A_a) with (A_b), and set the CD to:

Equal, if (A_a) = (A_b)

Unequal, if (A_a) ≠ (A_b)

Greater than or equal, if (A_a) ≥ (A_b)

Less than, if (A_a) < (A_b)

M-5035
Change 1

COMPARE B_b WITH B_a

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
74					a					7 b i					

HCB a,b

$(B_b) : (B_a)$; set CD

Compare (B_b) with (B_a) , and set the CD to:

Equal, if $(B_a) = (B_b)$

Unequal, if $(B_a) \neq (B_b)$

Greater than or equal if $(B_b) \geq (B_a)$

Less than, if $(B_b) < (B_a)$

COMPARE BIT TO ZERO

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
42					a					k					b i s					y											

BC ak,y,b,s

$(Y)_{ak} : 0$; set CD

Not character addressable.

The ak-designator specifies the bit of (Y) to be compared to zero where $0 \leq ak \leq 31$.

Test the bit specified by the ak-designator; then set the CD to:

Equal, if $(Y)_{ak} = 0$

Not equal, if $(Y)_{ak} \neq 0$

COMPARE GATED

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
47					a					k					b i s					y											

CG a,y,k,b,s

$|\underline{Y} - (A_a)| : (A_{a+1})$; set CD

Compare the absolute value of $\underline{Y} - (A_a)$ with (A_{a+1}) , and set the CD to:

Equal, if $|\underline{Y} - (A_a)| = (A_{a+1})$

Unequal, if $|\underline{Y} - (A_a)| \neq (A_{a+1})$

Greater than or equal, if $|\underline{Y} - (A_a)| \geq (A_{a+1})$

Less than, if $|\underline{Y} - (A_a)| < (A_{a+1})$

COMPARE INDEX, INCREMENT

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
43				a				k				b				i				s				y							

CXI a,y,k,b,s

If $(B_a) \geq \underline{Y}$, $0 \rightarrow B_a$ and set CD outside limits;

If $(B_a) < \underline{Y}$, $(B_a) + 1 \rightarrow B_a$ and set CD within limits.

a - equals B register designator.

Set the CD for outside limits and clears B_a if $(B_a) \geq \underline{Y}$. Otherwise, increment (B_a) by one and set the CD for within limits.

COMPARE LIMITS

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
45				a				k				b				i				s				y							

CL a,y,k,b,s

$(A_a), (A_{a+1}) : \underline{Y}$; set CD

Compare (A_a) and (A_{a+1}) with \underline{Y} . Set the CD to:

Within limits, if $(A_{a+1}) > \underline{Y} \geq (A_a)$

Outside limits, if $\underline{Y} < (A_a)$ or $\underline{Y} \geq (A_{a+1})$

COMPARE LIMITS, REGISTER

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
74				a				5				b				i															

HCL a,b

$(A_{a+1}), (A_a) : (A_b)$; set CD

Compare (A_a) and (A_{a+1}) to (A_b) and set the CD to:

Within limits, if $(A_{a+1}) > (A_b) \geq (A_a)$

Outside limits, if $(A_{a+1}) \leq (A_b)$ or $(A_b) < (A_a)$

M-5035
Change 1

COMPARE MASKED

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
46						a		k		b		i		s		y															

CM a,y,k,b,s

$(A_{a+1}) : (A_a) \odot \underline{Y}$; set CD

Compare the logical product of (A_a) and \underline{Y} with (A_{a+1}) and set the CD to:

Equal, if $(A_{a+1}) = (A_a) \odot \underline{Y}$

Unequal, if $(A_{a+1}) \neq (A_a) \odot \underline{Y}$

Greater than or equal, if $(A_{a+1}) \geq (A_a) \odot \underline{Y}$

Less than, if $(A_{a+1}) < (A_a) \odot \underline{Y}$

COMPARE MASKED, REGISTER

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
74				a		6		b		i					

HCM a,b

$(A_{a+1}) \odot (A_a) : (A_b)$; set CD

Compare the logical product of (A_a) and (A_{a+1}) with (A_b) and set the CD to:

Equal, if $(A_{a+1}) \odot (A_a) = A_b$

Unequal, if $(A_{a+1}) \odot (A_a) \neq A_b$

Greater than or equal, if $(A_{a+1}) \odot (A_a) \geq A_b$

Less than, if $(A_{a+1}) \odot (A_a) < A_b$

DOUBLE COMPARE

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
05				a	3	b	i	s	y																						

DC a,y,b,s

$(A_{a+1}, A_a) : (Y + 1, Y)$; set CD

Not character addressable.

Not repeatable.

Compare the content of the double length register (formed with the least significant half in A_a and the most significant half in A_{a+1}) with the content of the double length memory word (formed with (Y) as the least significant half and $(Y + 1)$ as the most significant half), and set the CD to:

Equal, if $(A_{a+1}, A_a) = (Y + 1, Y)$

Unequal, if $(A_{a+1}, A_a) \neq (Y + 1, Y)$

Greater than or equal, if $(A_{a+1}, A_a) \geq (Y + 1, Y)$

Less than, if $(A_{a+1}, A_a) < (Y + 1, Y)$

12.4.5 Instructions Involving Logical Operations

The following instructions perform a variety of logical operations involving register and memory:

ADD LOGICAL PRODUCT

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
01				a				4		b		i		s		y															

ALP a,y,b,s

$$(A_{a+1}) + \underline{Y} \odot (A_a) \rightarrow A_{a+1}; (A_a)_i = (A_a)_f$$

Add to (A_{a+1}) the logical product of (A_a) and \underline{Y} and store the result A_{a+1} .

AND

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
71		a		5		b		i		HAND a,b					

$$(A_a) \odot (A_b) \rightarrow A_a$$

Form the logical product of (A_a) and (A_b) and store the result in A_a .

$(A_b)_i = (A_b)_f$. Logical product or logical AND is defined by the following:

		A_b	
		0	1
A_a	0	0	0
	1	0	1

EXCLUSIVE OR (SELECTIVE COMPLEMENT A)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
01				a				3		b		i		s		y															

XOR a,y,b,s

$$\underline{Y} \oplus (A_a) \rightarrow A_a$$

Complement the individual bits in A_a corresponding to the ones in \underline{Y} , leaving the remaining bits in A_a unaltered. The result is stored in A_a .

EXCLUSIVE OR A

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
71					a			3		b		i			

HXOR a,b

$$(A_a) \oplus (A_b) \rightarrow A_a$$

Form the logical difference between (A_a) and (A_b) . The result is stored in A_a . Logical difference, exclusive OR, or selective complement is defined by the following:

		A_b	
		0	1
A_a	0	0	1
	1	1	0

INCLUSIVE OR (SELECTIVE SET A)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
01					a			0		b		i		s		y															

OR a,y,b,s

$$\underline{Y} \oplus (A_a) \rightarrow A_a$$

Set the individual bits of A_a corresponding to ones in \underline{Y} , leaving the remaining bits in A_a unaltered. The result is stored in A_a .

M-5035
Change 1

INCLUSIVE OR A

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
71					a	0	b	i							

HOR a, b

$$(A_a) \oplus (A_b) \rightarrow A_a$$

Form the logical sum of (A_a) and (A_b) . The result is stored in A_a .
 $(A_b)_i = (A_b)_f$ if $a \neq b$ and $a+1 \neq b$. Logical sum, inclusive OR, or selective set is defined by the following:

		A_b	
		0	1
A_a	0	0	1
1	1	1	1

LOAD LOGICAL PRODUCT IN A_a

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
01					a	5	b	i	s	y																					

LLP a, y, b, s

$$\underline{Y} \odot (A_a) \rightarrow A_a$$

Form the logical product of (A_a) and \underline{Y} . The result is stored in A_a .

LOAD LOGICAL PRODUCT IN A_{a+1}

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
01					a	7	b	i	s	y																					

LLPN a, y, b, s

$$\underline{Y} \odot (A_a) \rightarrow A_{a+1}; (A_a)_i = (A_a)_f$$

Form the logical product of (A_a) and \underline{Y} . The result is stored in A_{a+1} .
 (A_a) is unchanged.

REPLACE A - LOGICAL PRODUCT

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
03				a	6	b	i	s	y																						

RNLP a,y,b,s

$$(A_{a+1}) - \underline{Y} \odot (A_a) \rightarrow Y \text{ and } A_{a+1}; (A_a)_i = (A_a)_f$$

Subtract from (A_{a+1}) the logical product of \underline{Y} and (A_a) , then store the result in Y and A_{a+1} .

REPLACE A + LOGICAL PRODUCT

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
03				a	4	b	i	s	y																						

RALP a,y,b,s

$$(A_{a+1}) + \underline{Y} \odot (A_a) \rightarrow Y \text{ and } (A_{a+1}); (A_a)_i = (A_a)_f$$

Add to (A_{a+1}) , the logical product of \underline{Y} and (A_a) ; then store the result in Y and A_{a+1} .

REPLACE EXCLUSIVE OR (REPLACE SELECTIVE COMPLEMENT)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
03				a	3	b	i	s	y																						

RXOR a,y,b,s

$$\underline{Y} \oplus (A_a) \rightarrow Y \text{ and } A_a$$

Not character addressable.

Complement the individual bits in A_a corresponding to ones in \underline{Y} , leaving the remaining bits in A_a unaltered. The result is stored in A_a and Y.

REPLACE INCLUSIVE OR (REPLACE SELECTIVE SET)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
03				a	0	b	i	s	y																						

ROR a,y,b,s

$$\underline{Y} \oplus (A_a) \rightarrow Y \text{ and } A_a$$

Set the individual bits in A_a corresponding to ones in \underline{Y} , leaving the remaining bits in A_a unaltered. The result is stored in A_a and Y.

REPLACE LOGICAL PRODUCT

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
03					a	5	b	i	s	y																					

RLP a,y,b,s

$$\underline{Y} \odot (A_a) \rightarrow Y \text{ and } A_{a+1}$$

Load A_{a+1} with the logical product of \underline{Y} and (A_a) ; then store the result at Y .

REPLACE SELECTIVE SUBSTITUTE A

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
03					a	2	b	i	s	y																					

RMS a,y,b,s

$$\text{For } (A_a)_n = 1, (\underline{Y})_n \rightarrow (A_{a+1})_f \rightarrow Y; (A_a)_i = (A_a)_f$$

For each bit, n , which is equal to one in A_a , substitute the n th bit of \underline{Y} for the n th bit of the A_{a+1} . The result in A_{a+1} is stored at Y .

REPLACE SELECTIVE CLEAR

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
03					a	1	b	i	s	y																					

RSC a,y,b,s

$$(A_a) \odot \underline{Y}' \rightarrow Y \text{ and } A_a$$

Clear the individual bits in A_a corresponding to ones in \underline{Y} , leaving the remaining bits in A_a unaltered. The result is stored in A_a and 4.

SELECTIVE CLEAR A

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
01					a	1	b	i	s	y																					

SC a,y,b,s

$$(A_a) \odot \underline{Y}' \rightarrow A_a$$

Form the logical product of (A_a) and the complement of \underline{Y} . The result is stored in A_a . Selective clear is defined by the following:

	A_n		
\underline{Y}_n		0	1
0		0	1
1		0	0

SELECTIVE SUBSTITUTE

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
01				a				2		b		i		s		y															

MS a,y,b,s

$$\text{For } (A_a)_n = 1, \underline{Y}_n \rightarrow (A_{a+1})_n; (A_a)_i = (A_a)_f$$

For each bit, n, which is equal to one in A_a , substitute the nth bit of \underline{Y} for the nth bit of A_{a+1} .

STORE LOGICAL PRODUCT

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
02				a				4		b		i		s		y															

SLP a,y,b,s

$$(A_a) \odot (A_{a+1}) \rightarrow Y; (A_a)_i = (A_a)_f; (A_{a+1})_i = (A_{a+1})_f$$

Store at Y the logical product of (A_a) and (A_{a+1}) . The contents of the A-registers are unchanged.

SUBTRACT LOGICAL PRODUCT

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
01				a				6		b		i		s		y															

NLP a,y,b,s

$$(A_{a+1}) - \underline{Y} \odot (A_a) \rightarrow A_{a+1}; (A_a)_i = (A_a)_f$$

Subtract from (A_{a+1}) the logical product of (A_a) and \underline{Y} . The result is stored in A_{a+1} .

12.4.6 Shift Instructions

DOUBLE SHIFT LEFT CIRCULARLY

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
63						a	m									HDLC a,m
63						a	1	0	0	b			0	HDLC a,b,1	m : normal shift count 1 : shift count in B_b 2 : shift count in A_b	
63						a	1	1	0	b			0	HDLC a,b,2		

Shift the content of the double length register (formed with (A_a) as the least significant half and (A_{a+1}) as the most significant half) to the left circularly, where the shift count is specified by the m-field.

DOUBLE SHIFT RIGHT FILL SIGN

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
67						a	m									HDRS a,m
67						a	1	0	0	b			0	HDRS a,b,1	m : normal shift count 1 : shift count in B_b 2 : shift count in A_b	
67						a	1	1	0	b			0	HDRS a,b,2		

Shift the content of the double length register (formed with (A_a) as the least significant half and (A_{a+1}) as the most significant half) to the right m places, end-off with sign fill on the left. The maximum allowable shift count is $2^6-1 = 63$.

DOUBLE SHIFT RIGHT FILL ZEROS

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
65						a	m									HDRZ a,m
65						a	1	0	0	b			0	HDRZ a,b,1	m : normal shift count 1 : shift count in B_b 2 : shift count in A_b	
65						a	1	1	0	b			0	HDRZ a,b,2		

Shift the content of the double length register (formed with (A_a) as the least significant half and (A_{a+1}) as the most significant half) to the right, end-off and fill with zeros on the left, where the shift count is specified by the m-field. The maximum allowable shift is $2^6-1 = 63$.

SHIFT LEFT CIRCULARLY

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
62						a	m								
62						a	1	0	0	b			0		
62						a	1	1	0	b			0		

HLC a,m

m : normal shift count

HLC a,b,1

1 : shift count in B_b

2 : shift count in A_b

HLC a,b,2

Shift (A_a) left circularly where the shift count is specified by the m-field.

SHIFT RIGHT FILL SIGN

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
66						a	m								
66						a	1	0	0	b			0		
66						a	1	1	0	b			0		

HRS a,m

m : normal shift count

HRS a,b,1

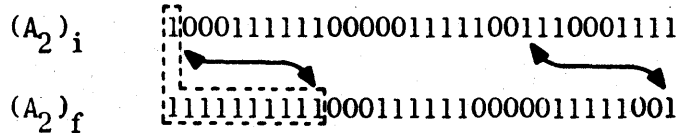
1 : shift count in B_b

2 : shift count in A_b

HRS a,b,2

Shift (A_a) to the right, end-off with sign fill on the left, where the shift count is specified by the m-field. The maximum allowable shift is $2^6 - 1 = 63$.

Example of right shift, sign fill A_2 , $m = 9$.



SHIFT RIGHT FILL ZEROS

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
64						a	m								
64						a	1	0	0	b			0		
64						a	1	1	0	b			0		

HRZ a,m

m : normal shift count

HRZ a,b,1

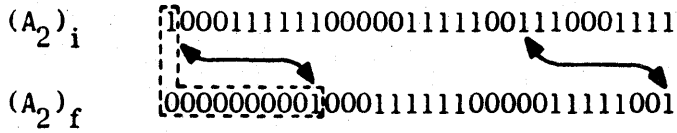
1 : shift count in B_b

HRZ a,b,2

2 : shift count in A_b

Shift (A_2) to the right, end-off and fill with zeros on the left, where the shift count is specified by the m-field. The maximum allowable shift is $2^6 - 1 = 63$.

Example of right shift A_2 , $m = 9$.



12.4.7 Instructions Referencing Control Memory

There are eight instructions referencing control memory; four are Format I full-word instructions, and the other four are half-word instructions, Format IV-A. The instructions which refer to the interrupt set of registers are all privileged instructions (PI) while the other instructions are privileged for certain control memory address reference.

LOAD INTERRUPT CMR (PI)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
55				ak						b	i	s	y																		

LCI ak,y,b,s

Y → CMR

The a-designator and k-designator form a six-bit (central processor) control memory address $(100-177)_8$ of the interrupt mode registers as follows.

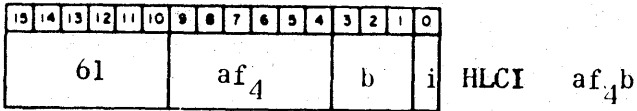
<u>EFFECTIVE ADDRESS</u>		<u>CONTROL MEMORY REGISTER</u>
<u>HARDWARE ASSIGNED</u>	ak-VALUE	
1	00	CMR address 100 (A ₀)
1	01	CMR address 101 (A ₁)
↓	↓	↓
1	26	CMR address 126 (S ₆)
1	27	CMR address 127 (S ₇)
1	30	CMR address 130 (unassigned)
↓	↓	↓
1	75	CMR address 175 (SIR ₅)
1	76	CMR address 176 (SIR ₆)
1	77	CMR address 177 (SIR ₇)

Load the CMR address (specified by the special ak-value) with Y.

NOTE

In the repeat mode, ak + 1 → ak. This instruction is not interruptable in the repeat mode.

LOAD INTERRUPT CMR WITH A (PI)



(A_b) → CMR

b specifies an accumulator register;

i (bit 0 or 16) = 1

The a-designator and k-designator form a six-bit address in (central processor) control memory in conjunction with the i-designator to access control memory locations as follows:

<u>i = 1</u>	<u>af₄ VALUE</u>	<u>CONTROL MEMORY REGISTER</u>
	00	CMR address 100 (A ₀)
	01	CMR address 101 (A ₁)
	↓	↓
	26	CMR address 126 (S ₅)
	27	CMR address 127 (S ₆)
	30	CMR address 130 (unassigned)
	↓	↓
	75	CMR address 175 (SIR ₅)
	76	CMR address 176 (SIR ₆)
	77	CMR address 177 (SIR ₇)

Load the CMR address specified by the af₄-designator with (A_b).

LOAD TASK CMR

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
54						ak						b	i	s	y																

LCT ak,y,b,s

Y → CMR

The a-designator and k-designator form a six-bit (central processor) control memory address (0-77₈) of the task mode registers as follows:

The a-designator and k-designator form a six-bit (central processor) control memory address (0-77₈) of the task mode registers as follows:

ak-VALUE

CONTROL MEMORY REGISTER

00
01
↓
26
27
30
↓
57
6X
7X

CMR address 0 (A₀)
CMR address 1 (A₁)
↓
CMR address 26 (S₆)*
CMR address 27 (S₇)*
CMR address 30 (unassigned)
↓
CMR address 57 (unassigned)
Breakpoint*
Active status

Load the CMR address (specified by the special ak-value) with Y.

NOTE

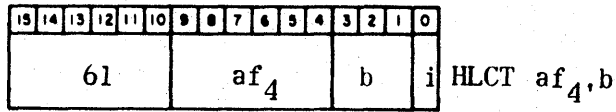
In the repeat mode, ak + 1 → ak. This instruction is not interruptable in the repeat mode.

This instruction is privileged when repeated.

* Addresses 60-77₈ and 20-27₈ are addressable in the interrupt mode only.

M-5035
Change 1

LOAD TASK CMR WITH A



(A_b) → CMR

b specifies an accumulator register

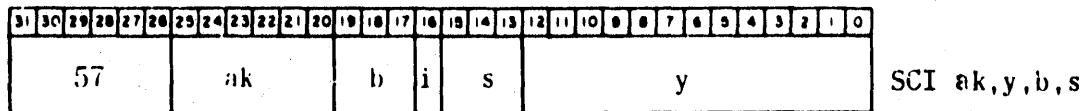
i (bit 0 or 16) = 0

The a-designator and f₄-designator form a six-bit (central processor) control memory in conjunction with the i-designator to access control memory locations as shown below.

<u>i = 0</u>	SPECIAL Af ₄ VALUE	CONTROL MEMORY REGISTER
	00	CMR address 0 (A ₀)
	01	CMR address 1 (A ₁)
	↓	↓
	26	CMR address 26 (S ₆)
	27	CMR address 27 (S ₇)
	30	CMR address 30 (unassigned)
	↓	↓
	56	CMR address 56 (unassigned)
	57	CMR address 57 (unassigned)
	6X	Breakpoint (accessible in interrupt mode only)
	7X	Active status (accessible in interrupt mode only)

Load the CMR address specified by the af₄-designator with (A_b).

STORE INTERRUPT CMR (PI)



(CMR) → Y

The a-designator and k-designator form a six-bit (central processor) control memory address (100 through 177)₈ of the interrupt mode registers as follows:

EFFECTIVE ADDRESS	HARDWARE ASSIGNED	ak-VALUE	CONTROL MEMORY REGISTER
1	↓	00	CMR address 100 (A ₀)
1	↓	10	CMR address 100 (A ₁)
1	↓	26	CMR address 126 (S ₆)
1	↓	27	CMR address 127 (S ₇)
1	↓	30	CMR address 130 (unassigned)
1	↓	75	CMR address 175 (SIR ₅)
1	↓	76	CMR address 176 (SIR ₆)
1	↓	77	CMR address 177 (SIR ₇)

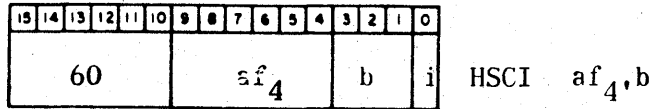
Store the content of the CMR address (specified by the ak-value) at Y.

NOTE

In the repeat mode, ak + 1 → ak. This instruction is not interruptable in the repeat mode.

M-5035
Change 1

STORE INTERRUPT CMR IN A (PI)



(CMR) → A_b

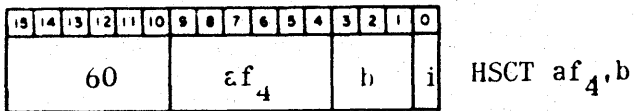
b specifies an accumulator register; ji (bit 0 or 16) = 1

The a-designator and the f₄-designator form a six-bit (central processor) control memory address in conjunction with the i-designator to access control memory locations as follows:

i = 1	af ₄ VALUE	CONTROL MEMORY REGISTER
	00	CMR address 100 (A ₀)
	01	CMR address 101 (A ₁)
	↓	↓
	26	CMR address 126 (S ₆)
	27	CMR address 127 (S ₇)
	30	CMR address 130 (unassigned)
	↓	↓
	75	CMR address 175 (SIR ₅)
	76	CMR address 176 (SIR ₆)
	77	CMR address 177 (SIR ₇)

Load A_b with (CMR_{af₄}).

STORE TASK CMR IN A



(CMR) $\rightarrow A_b$

b specifies an accumulator register;

i (bit 0 or 16) = 0

The a-designator and f_4 -designator form a six-bit (central processor) control memory address in conjunction with the i-designator to access control memory location as follows:

<u>i = 0</u>	<u>af_4 VALUE</u>	<u>CONTROL MEMORY REGISTER</u>
	00	CMR address 0 (A_0)
	01	CMR address 1 (A_1)
	↓	↓
	26	CMR address 26 (S_6)
	27	CMR address 27 (S_7)
	30	CMR address 30 (unassigned)
	↓	↓
	56	CMR address 56 (unassigned)
	57	CMR address 57 (unassigned)
	6X	Breakpoint (accessible in interrupt mode only)
	7X	Active status (accessible in interrupt mode only)

Load A_b with (CMR af_4).

12.4.8 Interrupt Handling Instructions

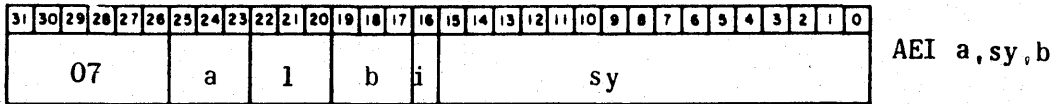
These are all privileged instructions, and none are character addressable or repeatable.

ALLOW CLASS III INTERRUPTS (PI)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
77						0	5	0	0	HAI					

Release the lockout for input/output Class III interrupts. This shall not affect the individual channel interrupts enable/disable logic as set by AEI and PEI instructions.

ALLOW ENABLE INTERRUPT (PI)



The a-designator specifies which IOC will receive the enable interrupt information:

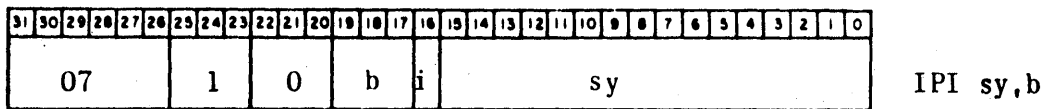
<u>SPECIAL a-VALUE</u>	<u>IOC TO RECEIVE ENABLE INTERRUPT INFORMATION</u>
0	IOC #0 receives enable interrupt
1	IOC #1 receives enable interrupt
2	IOC #2 receives enable interrupt
3	IOC #3 receives enable interrupt
4-7	Not used

The s-designator in conjunction with the y-operand forms a 16-bit sy-field. The 16-bit sy-field enables interrupts on a channel basis in the specified IOC as follows:

<u>SPECIAL sy-VALUE</u>	<u>FUNCTION PERFORMED AT SPECIFIED IOC</u>
sy (bit 0) = 1	Enable channel 0
sy (bit 1) = 1	Enable channel 1
sy (bit 2) = 1	Enable channel 2
sy (bit 3) = 1	Enable channel 3
sy (bit 4) = 1	Enable channel 4
sy (bit 5) = 1	Enable channel 5
sy (bit 6) = 1	Enable channel 6
sy (bit 7) = 1	Enable channel 7
sy (bit 8) = 1	Enable channel 8
sy (bit 9) = 1	Enable channel 9
sy (bit 10) = 1	Enable channel 10
sy (bit 11) = 1	Enable channel 11
sy (bit 12) = 1	Enable channel 12
sy (bit 13) = 1	Enable channel 13
sy (bit 14) = 1	Enable channel 14
sy (bit 15) = 1	Enable channel 15

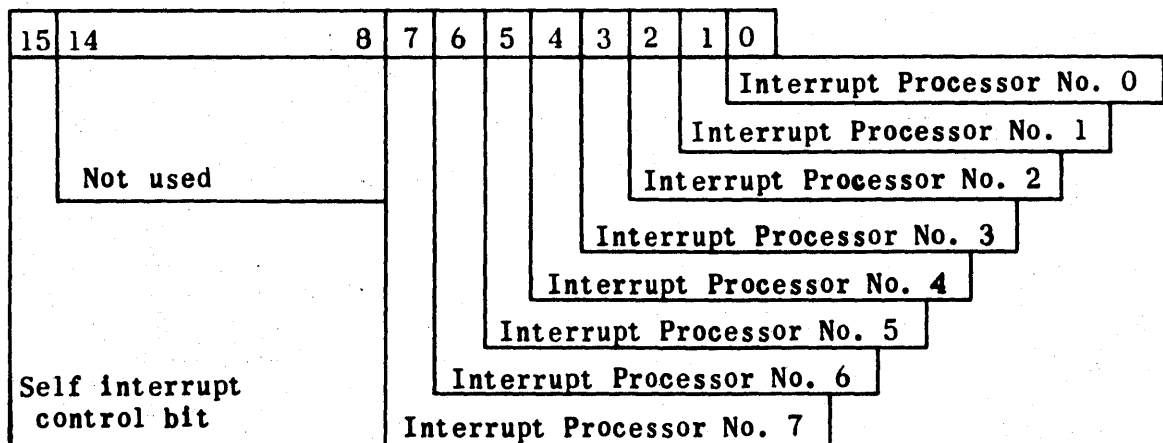
Enable the interrupt request for each IOC channel specified in the 16-bit sy-field.

INTERPROCESSOR INTERRUPT (PI)



A class II interrupt is generated for each processor selected by the sy-field after B_b modification. The processor number is bit-encoded in the lower 18 bits of $sy + (B_b)_{15}$ as follows:

16-bit result of $sy + (B_b)_{15-0}$



This instruction interrupts all processors whose corresponding bit is set as indicated above. If bit 15 is set, and if the processor executing this instruction is selected to be interrupted, then the processor executing the instruction will ignore the interrupt. If bit 15 is clear and the processor executing the instruction is selected to be interrupted, then the interrupt request will occur.

M-5035

INTERRUPT RETURN (PI)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
07				0				5				0				0				0								IR			

Return control to the processor state designated by the DSW corresponding to the state control field in the active status register.

PREVENT CLASS III INTERRUPTS (PI)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
77				0				4				0				0				HPI

Lock out the input/output class III interrupts. The interrupts are held pending until this lockout is removed.

PREVENT ENABLE INTERRUPT (PI)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
07				a				2				b				i				sy								PEI a,sy,b			

The a-designator specifies which IOC will receive the disable interrupt information:

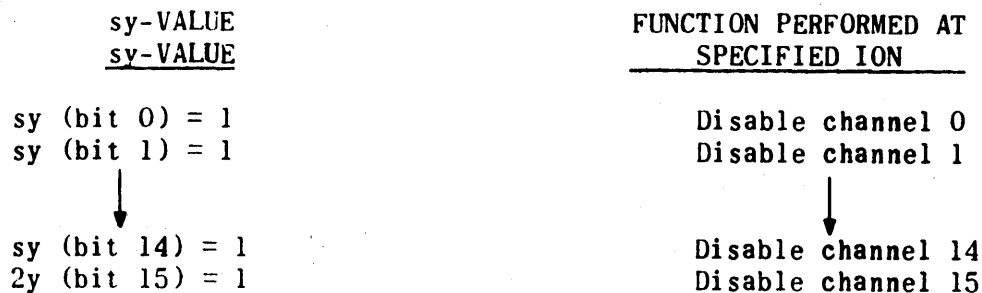
SPECIAL
a-VALUES

- 0
- 1
- 2
- 3
- 4-7

IOC TO RECEIVE DISABLE
INTERRUPT INFORMATION

- IOC #0 receives disable interrupt
- IOC #1 receives disable interrupt
- IOC #2 receives disable interrupt
- IOC #3 receives disable interrupt
- Not used

The s-designator in conjunction with the y-field forms a 16-bit sy-field. The 16-bit sy-field disables interrupts on a channel basis in the specified IOC as follows:



Disable the interrupt request for each IOC channel specified in the 16-bit sy-field.

WAIT FOR INTERRUPT (PI)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
77				0		6		0		1		HWFI			

This instruction causes the computer to stop referencing memory until an interrupt occurs (any class not currently locked out). After an interrupt request is detected, the processor honors the interrupt, saving the address of the instruction following the wait for interrupt. Upon return from the interrupt, normal processing continues.

12.4.9 Miscellaneous Instructions

CLEAR BIT

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
32				ak				b	i	s	y																				

BZ ak,y,b,s

$$0 \rightarrow (Y)_{ak}$$

Not character addressable.

The ak-designator specifies the bit of (Y) to be cleared, where $0 \leq ak \leq 31$.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
02				a	0	b	i	s	y																						

CNT a,y,b,s

Number of "1's" in $\underline{Y} \rightarrow A_a$

Count the bits in \underline{Y} which are set to one and store the count in A_a .

DOUBLE SCALE FACTOR A

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
70				a	1	b	i	HDSF a,b							

Normalize (A_{a+1}, A_a); normalized shift count $\rightarrow A_b$.

Shift the content of the double length register (formed with (A_a) as the least significant half and (A_{a+1}) as the most significant half) to the left circularly until normalized. The required shift count is stored in A_b . If a or $a+1 = b$, the registers shall be normalized with no shift count available. If (A_{a+1}, A_a) $i = 0$ or all 1's, the resultant shift count is 63.

EXECUTE REMOTE

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0?			0			2			b			i			s			y													

XR y,b,s

Execute instruction at Y; (P) is unchanged.

Not character addressable.

Not repeatable.

Execute the whole-word instruction or two half-word instructions which is at Y, without changing (P).

EXECUTE REMOTE LOWER

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
02			0			3			b			i			s			y													

XRL y,b,s

Execute instruction at Y_L; (P) is unchanged.

Not character addressable.

Not repeatable.

Execute the lower half-word instruction at Y, without changing (P).

ENTER EXECUTIVE STATE

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
07			0			0			b			i			sy																

XS sy,b

Interrupt to executive entrance address.

Not character addressable,

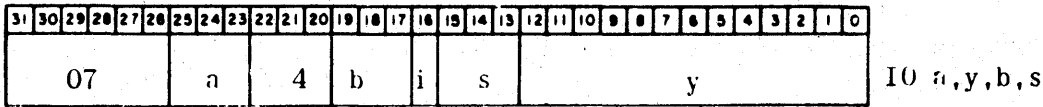
Not repeatable.

The s-designator and the y-designator are combined to form a 16-bit sy-field which, after B modification, is used as follows:

Y_{0-15} form the interrupt code word.

Switch control to the interrupt state and transfer control to the executive entrance address as determined by the initial condition word pointer associated with the Class IV interrupt state.

INITIATE INPUT/OUTPUT (PI)



$$Y \rightarrow IOC_a (Y = y + (B_b) + (S_s))$$

Not character addressable.

Not repeatable.

The a-designator specifies which IOC will receive the processor command as follows:

<u>SPECIAL a-VALUE</u>	<u>IOC to RECEIVE COMMAND ADDRESS</u>
0	IOC #0 receives address
1	IOC #1 receives address
2	IOC #2 receives address
3	IOC #3 receives address
4-7	Not used

Provide the IOC specified by the a-field with the absolute address Y (address of the first command in an input/output program sequence).

IOC MONITOR CLOCK (PI)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
07				a	3	b	i	sy																							

LIM a,sy,b

Y → monitor clock

Not character addressable.

Not repeatable.

The a-designator specifies which IOC monitor clock will be entered with Y:

<u>a-VALUE</u>	<u>IOC MONITOR CLOCK TO BE ENTERED WITH Y</u>
0	IOC #0
1	IOC #1
2	IOC #2
3	IOC #3
4-7	Not used

The s-designator and the y-field form a 16-bit sy-field which, after B modification, is used to load the IOC monitor clock specified by the a-value.

If the 16-bit sy-field (after B modification) is negative, the monitor clock in the specified IOC is disabled; if the 16-bit sy-field (after B modification) is equal to zero, the IOC monitor clock (that was entered with zero) generates a Class III interrupt in an attempt to interrupt any central processor connected to the IOC; otherwise, the IOC monitor clock is counted down in the normal manner at the rate of 1,024 counts per second.

M-5035
Change 1

REPEAT

29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
07				a		6		b		j		sy																RP a, sy, b	

Repeat NI (B_7) times or until test condition is satisfied.

Not character addressable.

Not repeatable.

The a-designator controls the repeated instruction termination conditions as follows:

<u>a-VALUE</u>	<u>NON-COMPARE INSTRUCTIONS</u>	<u>a-VALUE</u>	<u>COMPARE INSTRUCTIONS</u>
0	Terminate if $A \neq 0$	0	Terminate if CD set to \neq
1	Terminate if $A = 0$	1	Terminate if CD set to $=$
2	Terminate if $A \geq 0$	2	Terminate if CD set to $>$
3	Terminate if $A < 0$	3	Terminate if CD set to \geq
4	Do not terminate	4	Terminate if CD set to $<$
5	Terminate if A contains an even number of binary ones.	5	Terminate if CD set to \leq
6	Terminate if A contains an odd number of binary ones.	6	Terminate if CD set to OL
7	Do not terminate	7	Terminate if CD set to WL

The b-designator shall specify the use of S_6 on replace instructions as follows:

If $b \neq 0$, and the repeated instruction is a replace instruction, the operand address of the replace instruction is incremented by (S_6) for the store portion of the replace instruction. If $b = 0$, normal operation occurs.

The s-designator and the y-field form a 16-bit constant (including sign) which specifies the increment or decrement of the operand address after each execution of the repeated instruction. This increment or decrement is added to the B-register of the repeated instruction after each execution.

The instruction shall repeat the next sequential instruction n times, where n is contained in B_7 . If B_7 is zero, the next sequential instruction is skipped.

A, above, refers to the accumulator specified in the repeated instruction. The repeat count is available in B_7 which is decremented by 1 after each execution. The use of B_7 in the instruction being repeated should be avoided.

SCALE FACTOR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
70						a	0	b	i	HSF a,b					

Normalize (A_a); normalized shift count $\rightarrow A_b$

Normalize (A_a), and store the shift count in A_b . (A_a) is shifted left circularly until normalized, the required shift count is then stored in A_b . If $a = b$, the accumulator is normalized with no shift count available. If (A_a) $i = 0$, the resultant shift count is 63.

SET BIT

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
33						ak	b	i	s	y							BS ak,y,b,s														

$1 \rightarrow (Y)_{ak}$

Not character addressable.

The ak-designator specifies the bit of (Y) to be set where $0 \leq ak \leq 31$.

STOP PROCESSOR (PI)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
77						0	6	0	0	HALT					

Terminate processor operation and illuminate the stop indicator light.

M-5035
Change 1

STORE I/O MONITOR CLOCK (PI)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
77				a	0	b	i	HSIM a,b							

(IOC_a monitor clock) → A_b

The a-designator specifies which IOC monitor clock is to be referenced as follows:

SPECIAL a-VALUE	FUNCTION PERFORMED
0	(IOC #0) Mon. Clk → A _b
1	(IOC #1) Mon. Clk → A _b
2	(IOC #2) Mon. Clk → A _b
3	(IOC #3) Mon. Clk → A _b
4-7	Not used

Load A_b with the content of the IOC monitor clock selected by the a-designator.

STORE REAL-TIME CLOCK

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
77				a	1	b	i	HSTC a,b							

(IOC_a real-time clock) → A_b

The a-designator specifies which IOC real-time clock is to be referenced as follows:

SPECIAL a-VALUE	FUNCTION PERFORMED
0	(IOC #0) RTC → A _b
1	(IOC #1) RTC → A _b
2	(IOC #2) RTC → A _b
3	(IOC #3) RTC → A _b
4-7	Not used

Load A_b with the content of the IOC RTC selected by the a-designator.

TEST AND SET FLAG

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
03			0			7			b			i			s			y													

TSF y,b,s

Set the CD; 1 → Y₃₁

Not character addressable.

Set the flag bit in the upper bit of address Y. If this bit was originally cleared, set CD to equal; otherwise, set CD to not equal.

M-5035
Change 1

12.4.10 Extension Mnemonics

Extension mnemonics are recognized by the Assembler, but are not unique hardware instructions themselves; however, they utilize special configurations of hardware instructions or are made available for programmer convenience. These instructions fall into three categories: 1) pseudo hardware instructions; 2) indirect words; and 3) buffer control words.

a) Pseudo Hardware Instructions:

CLEAR A (ZA)

CLEAR B (ZB)

HALF-WORD CONSTANT (HK)

NO OPERATION (Full Word) (NOOP)

NO OPERATION (Half Word) (HNO)

STORE ZEROS (SZ)

b) Indirect Word Instructions:

INDIRECT WORD (IW)

INDIRECT WORD CHARACTER (IWC)

INDIRECT WORD CHARACTER INCREMENT (IWCI)

INDIRECT WORD, SPECIAL BASE (IWS)

INDIRECT WORD, SPECIAL INDEX (IWB)

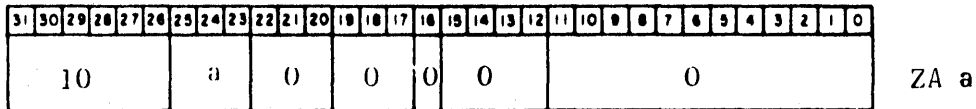
MEMORY PROTECTION (MP) (Not an instruction)

c) Buffer Control Word Instructions:

BUFFER CONTROL WORD (BCW)

BUFFER CONTROL WORD ESI (BCWE)

CLEAR A

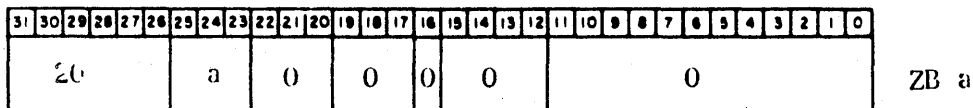


$0 \rightarrow A_a$

a - normal

Load A_a with zeros. All other instruction fields are zeros.

CLEAR B

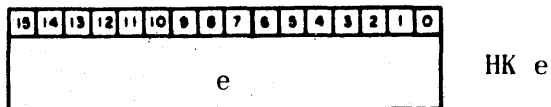


$0 \rightarrow B_a$

a - special (specifies B_a)

Load B_a with zeros. All other instruction fields are zeros.

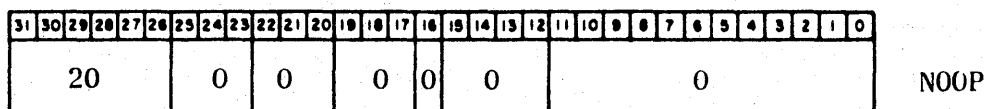
HALF-WORD CONSTANT



This instruction permits programmers to intersperse half-word constants with half-word instructions as desired. By the Assembler's half-word pairing convention, two sequential half-word instructions occupy the upper and lower half of the same generated object word. A single half-word instruction occupies the upper half of the generated word and the Assembler pads the lower half with zeros.

The e-operand may be any expression resulting in a 16-bit value.

NO OPERATION (FULL WORD)



$0 \rightarrow B_0$

a - specifies B_0

M-5035
Change 1

NO OPERATION (HALF-WORD)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
20					0			3		0		0		IINO	

$$(B_0) \rightarrow B_0$$

STORE ZEROS

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
23					0			k		b		i		s		y											SZ y,k,b,s				

$$() \rightarrow Y$$

This is equivalent to the STORE B instruction where B_a is B_0 .

INDIRECT WORD

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1 0		not used										b		i		s		y											IW y,b,s		

This format is for indirect word addressing only. Cascading continues until the i-field is equal to zero. Indexing is available at each cascade level.

INDIRECT WORD, CHARACTER

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0 1		w					p					b		0		s		y											IWC y,w,p,b,s		

- w = number of bits in the character
- p = least significant bit position of the character
- b = normal
- i = zero
- s = normal

This format is for single character addressing only. The character defined by p and w is stored in the arithmetic register (specified by the parent instruction) right-justified and zero-filled on the left. The range of p is $0 \leq p \leq 31$. The range of w is $1 \leq w \leq 31$ only if $p + w$ is ≤ 32 .

INDIRECT WORD, CHARACTER, INCREMENT

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	w					p					b	0	s	y																

IWCI y,w,p,b,s

w = number of bits in the character

p = least significant bit position of the character

b = normal

i = zero

s = normal

This format is for successive character addressing. The character defined by p and w is stored in the arithmetic register (specified by the parent instruction) right-justified and zero-filled on the left. The range of p is $0 \leq p \leq 31$. The range of w is $1 \leq w \leq 31$ only if $p + w \leq 32$.

After the character has been obtained, the indirect control word is modified for subsequent addressing of the next character. This sequential character addressing assumes the next character contiguous, of the same size, and lower in bit position than the present character. If the next character would lie outside the current 32-bit word, the character is assumed to lie in the next sequential memory word with its most significant bit in bit 31 of that word. Modification of the indirect control word occurs accordingly.

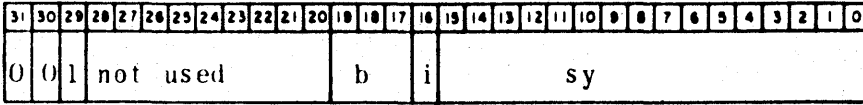
INDIRECT WORD, SPECIAL BASE

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	not used									b	i	sy																	

IWS sy,b

The contents of the b-field is a base register designator. The next address is the sum of the contents of the indicated base register plus the 16-bit D-field.

INDIRECT WORD, SPECIAL INDEX

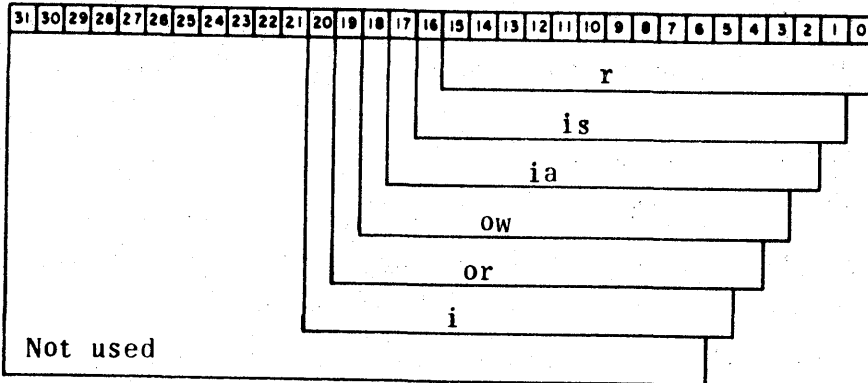


IWB sy,b

The contents of the b-field is an index register designator, where the contents of the index register is in the format of the P-register. The next address is the sum of three quantities:

1. The contents of the base register designated by $(B_b)_{19-17}$
2. $(B_b)_{15-0}$
3. The 16-bit D-field.

MEMORY PROTECTION



MP r,i,or,ow,ia,is

This mnemonic is used to specify the memory protection register word.

The special fields are as follows:

- r - bit 0-15 = maximum allowed displacement
- i - bit 20 = 1 when instructions may be executed
- or - bit 19 = 1 when operands may be read
- ow - bit 18 = 1 when operands may be written
- ia - bit 17 = 1 when indirect references are allowed
- is - bit 16 = 1 when computer will use B and S-registers in indirect addressing

BUFFER CONTROL WORD

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Final Address Compare Bits														y																	

BCW y,1

Description:

1 - length of buffer in whole words where $0 < 1 \leq 16384$

This instruction permits the programmer to code buffer control words symbolically and at the same time inform the assembler of the special addressing characteristics required of buffer control words.

The final address compare bits in bits 31 through 18 of the generated object word are computed by the Assembler or Loader.

BUFFER CONTROL WORD ESI

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
k	Final Address Compare Bits												y																		

BCWE y,1,k

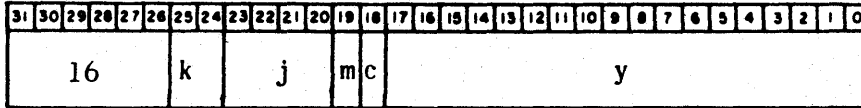
Description:

1 - length of buffer in whole words where $0 < 1 \leq 2048$

This instruction permits the programmer to code ESI buffer control words symbolically and at the same time to inform the assembler of the special addressing characteristics of buffer control words. The final address compare bits in bits 28 through 18 of the generated object word are computed by the assembler or loader.

12.4.11 Input/Output Instructions

ACTIVATE EXTERNAL FUNCTION CHAIN ON C_j

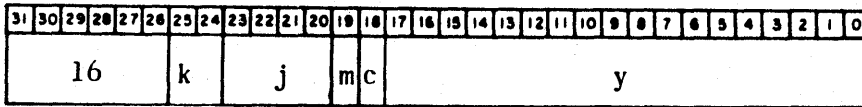


AFC j,y,c

- k - 2
- j - normal
- m - not used
- c - normal

Transfer Y to bits 55 through 38 of the appropriate chain pointer in IOC control memory and set the external function chain for channel j active.

ACTIVATE INPUT CHAIN ON C_j

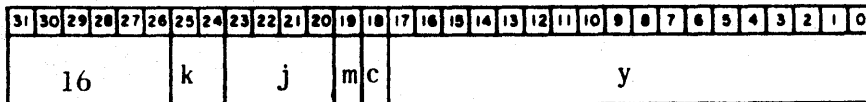


AIC j,y,c

- k - 0
- j - normal
- m - not used
- c - normal

Transfer Y to bits 55 through 38 of the appropriate chain pointer in IOC control memory and set the input chain for channel j active.

ACTIVATE OUTPUT CHAIN ON C_j



AOC j,y,c

- k - 1
- j - normal
- m - not used
- c - normal

Transfer Y to bits 55 through 38 of the appropriate chain pointer in IOC control memory and set the output chain for channel j active.

ACTIVATE EXTERNAL INTERRUPT CHAIN ON C_j

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
16						k	j				m	c	y																		

AXC j,y,c

k - 3

j - normal

m - not used

c - normal

Transfer Y to bits 55 through 38 of the appropriate chain pointer in IOC control memory and set the external interrupt chain for channel j active.

CLEAR BIT

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
26						kj				m	c	y																			

IBZ kj,y,c

$$0 \rightarrow Y_{kj} \quad (0 \leq kj \leq 31)$$

k - special

j - special

m - not used

c - normal

The k- and j-fields are used in combination to designate a bit position where $0 \leq kj \leq 31$.

Clear bit kj of (Y) to zero.

INITIATE EXTERNAL FUNCTION BUFFER ON C_j

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
12				k	j				m	c	y																				

FB j,y,k,c,m

k - special

j - normal

m - m = 1 - with monitor; m = 0 - without monitor

c - c = 1 - activate chain on buffer termination;

c = 0 - do not activate chain on buffer termination

Initiate an external function buffer on channel j as defined by k.

k = 0 - one-word buffer with force; Y is the buffer control word

k = 1 - one-word buffer without force; Y is the buffer control word

k = 2 - n-word buffer; (Y) is the buffer control word

k = 3 - unused

INITIATE EXTERNAL INTERRUPT BUFFER

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
13				k		j		m		c		y																			

XB j,y,k,c,m

k - special

j - normal

m - m = 1 - with monitor; m = 0 - without monitor

c - c = 1 - activate chain on buffer termination;

c = 0 - do not activate chain on buffer termination

Initiate an external interrupt buffer on channel j as defined by the k-designator. (Y) is the buffer control word.

k = 0 - No transfer. The buffer current address is incremented by 1 after each request.

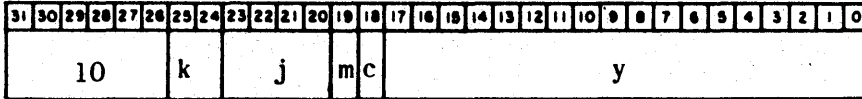
k = 1 - For each request, transfer external interrupt code word bits 7 through 0 to bits 31 through 24, 23 through 16, 15 through 8, and 7 through 0 of memory, in that order. The current buffer address is incremented by 1 after each 4 transfers.

k = 2 - For each request, transfer external interrupt code word bits 15 through 0 to bits 31 through 16 and 15 through 0 of memory, in that order. The current buffer address is incremented by 1 after each 2 transfers.

k = 3 - For each request, transfer all 32 bits of the external interrupt code word to memory. The current buffer address is incremented by 1 after each transfer.

The monitor interrupt flag controls when an external interrupt request is sent to the processor(s) at buffer termination.

INITIATE INPUT BUFFER ON C_j



IB j,k,c,m

k - special

j - normal

m - m = 1 - with monitor; m = 0 - without monitor

c - c = 1 - activate chain on buffer termination;

c = 0 - do not activate chain on buffer termination

Initiate an input buffer on channel j as defined by the k-field (Y) is the buffer control word.

k = 0 - No data transfer. The current buffer address is incremented by one after each request.

k = 1 - For each request, transfer input bits 7 through 0 to bits 31 through 24, 23 through 16, 15 through 8, and 7 through 0 of memory, in that order. The current buffer address is incremented by one after each four transfers.

k = 2 - For each request, transfer input bits 15 through 0 to bits 31 through 16 and 15 through 0 of memory in that order. The current buffer address is incremented by one after each two transfers.

k = 3 - For each request, transfer input bits 31 through 0 to bits 31 through 0 of memory. The current buffer address is incremented by one after each request.

INITIATE OUTPUT BUFFER

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
11				k	j				m	c	y																				

OB j,y,k,c,m

k - special

j - normal

m - m = 1 - with monitor; m = 0 - without monitor;

c - c = 1 - activate chain on buffer termination;

c = 0 - do not activate chain on buffer termination

Initiate an output buffer on channel j as defined by the k-designator.
(Y) is the buffer control word.

k = 0 - Transfer 32 bits of zeros for each request. The current buffer address is incremented by one after each transfer.

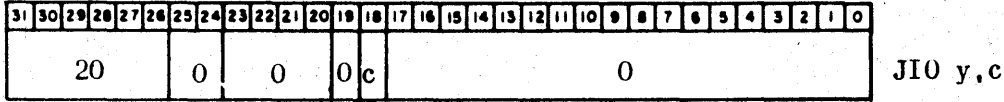
k = 1 - For each request, transfer to output bits 7 through 0 from memory bits 31 through 24, 23 through 16, 15 through 8, and 7 through 0, in that order. The current buffer address is incremented by one after each four transfers.

k = 2 - For each request, transfer to output bits 15 through 0 from memory bits 31 through 16 and 15 through 0, in that order. The current buffer address is incremented by one after each two transfers.

k = 3 - For each request, transfer to output bits 31 through 0 from memory bits 31 through 0. The current buffer address is incremented by one after each transfer.

M-5035

JUMP (INPUT/OUTPUT)



Y → chain pointer

k - not used

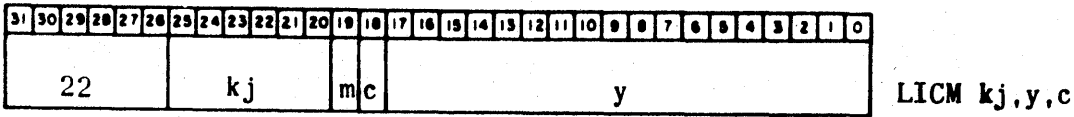
j - not used

m - not used

c - normal

Load the chain pointer with Y.

LOAD IOC CONTROL MEMORY



(Y) → IOC control memory address kj

k - special

j - special

m - not used

c - normal

The k- and j-fields are used in combination to specify an IOC control memory address where $0 \leq kj \leq 63$.

Load the lower 32 bits of the IOC control memory address specified by kj with (Y).

LOAD REAL-TIME CLOCK

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
23					0	0	0	c	y																						

ILTC y,c

(Y) → RTC

k - not used

m - not used

c - normal

Load the real-time clock with (Y).

SET BIT

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
25					kj			m	c	y																					

IBS ki,y,c

 $1 \rightarrow Y_{kj} \quad (0 \leq kj \leq 31)$

k - special

j - special

m - not used

c - normal

The k- and j-fields are used in combination to designate a bit position where $0 \leq kj \leq 31$.

This instruction sets bit kj of (Y) to one.

SET EXTERNAL FUNCTION MONITOR INTERRUPT REQUEST ON Cj

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
15					2	j			m	c	not used																				

FMIR j,c

n - not used

c - normal

Set the external function monitor interrupt request on channel j.

SET EXTERNAL INTERRUPT MONITOR INTERRUPT REQUEST ON Cj

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
15				}	j				m	c	not used																				

XMIR j,c

- k - 3
- j - normal
- m - not used
- c - normal

Set the external interrupt monitor interrupt request on channel j.

SET INPUT MONITOR INTERRUPT REQUEST ON Cj

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
15				0	j				m	c	not used																				

IMIR j,c

- k - 0
- j - normal
- m - not used
- c - normal

Set the input monitor interrupt request on Cj.

SET OUTPUT MONITOR INTERRUPT REQUEST

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
15				1	j				m	c	not used																				

OMIR j,c

- k - 1
- j - normal
- m - not used
- c - normal

Set the output monitor interrupt request on Cj.

STORE IOC CONTROL MEMORY

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
24				kj				m	c	y																					

SICM kj,y,c

(IOC control memory)_{kj} → Y

k - special

j - special

m - not used

c - normal

The k- and j-fields are used in combination to designate an IOC control memory address where $0 \leq kj \leq 63$.

Store the lower 32 bits of the content of the IOC control memory address specified by kj at address Y.

TERMINATE EXTERNAL FUNCTION BUFFER ON Cj

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
14				2		j		m	c	not used																					

TFB j,c,m

k - 2

j - normal

m - m = 1 - do not clear buffer monitor interrupt

m = 0 - clear buffer monitor interrupt

c - normal

y - not used

Terminate the external function buffer on channel j.

TERMINATE EXTERNAL INTERRUPT BUFFER ON Cj

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
14														3			j		m c		not used													

TXB j,c,m

k - 3

j - normal

m - m = 1 - do not clear buffer monitor interrupt

m = 0 - clear buffer monitor interrupt

c - normal

y - not used

Terminate the external interrupt buffer on channel j.

TERMINATE INPUT BUFFER ON Cj

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
14														3			j		m c		not used													

TIB j,c,m

k - 0

j - normal

m - m = 1 - do not clear buffer monitor interrupt

m = 0 - clear buffer monitor interrupt

c - normal

y - not used

Terminate the input buffer on channel j.

TERMINATE OUTPUT BUFFER ON Cj

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
14				1	j	m	c	not used																							

TOB j,c,m

k - 1

j - normal

m - m = 1 - do not clear buffer monitor interrupt

m = 0 - clear buffer monitor interrupt

c - normal

y - not used

Terminate the output buffer on channel j.

TEST AND SET FLAG

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
25				k	j	m	c	y																							

ITSF y,c

$1 \rightarrow Y_{31}$; if $(Y)_{31}$ was originally cleared, skip; else NI.

k - not used

j - not used

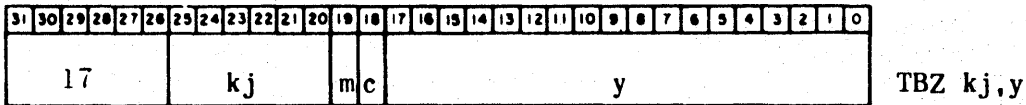
m - not used

c - normal

Set flag bit 31 of (Y) to one. If the flag bit was originally cleared, the chain pointer is indexed by 2. If the flag bit was originally set to one, the chain pointer is indexed by one.

M-5035

TEST BIT CLEARED

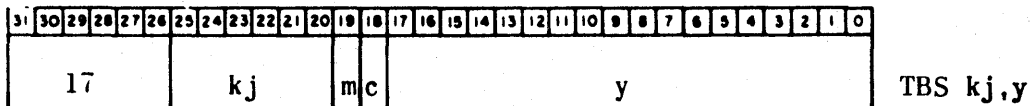


If $(Y)_{kj} = 0$, skip; else NI ($0 \leq kj \leq 31$)

m - 0
c - not used

Test the bit of (Y) specified by the combined kj-field for zero. If $(Y)_{kj}$ is equal to zero, the chain pointer is indexed by 2; otherwise, the chain pointer is indexed by 1 resulting in execution of NI. The valid range is $0 \leq kj \leq 31$.

TEST BIT SET



If $(Y)_{kj} \neq 0$, skip; else NI ($0 \leq kj \leq 31$)

m - 1
c - not used

Test the bit of (Y) specified by the combined kj-field for one. If $(Y)_{kj}$ is equal to one, the chain pointer is indexed by 2; otherwise, the chain pointer is indexed by 1 resulting in execution of NI. The valid range is $0 \leq kj \leq 31$.

APPENDIX A
CHARACTER CODES

ASCII Character	ASCII Code	XS-3 Equivalent-1004M or (1004 REX)	Standard Card Punch
line feed	012	00	0-5-9
carriage return	015	00	12-5-8-9
Space	040	00	blank
!	041	43	12-7-8
"	042	40 (56)	7-8
#	043	35 (37)	3-8
\$	044	42	11-3-8
%	045	61 (55)	0-4-8
&	046	20 (63)	12
'	047	40 (56)	5-8
(050	55 (61)	12-5-8
)	051	77 (75)	11-5-8
*	052	41	11-4-8
+	053	63 (20)	12-6-8
,	054	62	0-3-8
-	055	02	11
.	056	22	12-3-8
/	057	64	0-1
0	060	03	0
1	061	04	1
2	062	05	2
3	063	06	3
4	064	07	4
5	065	10	5
6	066	11	6
7	067	12	7
8	070	13	8
9	071	14	9
:	072	21	2-8
;	073	16	11-6-8
<	074	36	12-4-8
=	075	37 (35)	6-8
>	076	76	0-6-8
?	077	23	0-7-8
@	100	56 (40)	4-8

ASCII Character	ASCII Code	XS-3 Equivalent	Standard Card Punch
A	101	24	12-1
B	102	25	12-2
C	103	26	12-3
D	104	27	12-4
E	105	30	12-5
F	106	31	12-6
G	107	32	12-7
H	110	33	12-8
I	111	34	12-9
J	112	44	11-1
K	113	45	11-2
L	114	46	11-3
M	115	47	11-4
N	116	50	11-5
O	117	51	11-6
P	120	52	11-7
Q	121	53	11-8
R	122	54	11-9
S	123	65	0-2
T	124	66	0-3
U	125	67	0-4
V	126	70	0-5
W	127	71	0-6
X	130	72	0-7
Y	131	73	0-8
Z	132	74	0-9
[133	17	12-2-8
/	134	15	0-2-8
]	135	01	11-2-8
↑	136	00	11-7-8
←	137	00	0-5-8

SPECIAL CHARACTER CODES FOR THE OPTIONS DECLARATION

<u>029</u>	<u>026</u>	<u>U (ULTRA)</u>	<u>Standard Card Punch</u>
046	053	053	12
042	042	047	7-8
075	075	076	6-8
047	047	072	5-8
0100	047	047	4-8
043	075	075	3-8
072	072	053	2-8
077	077	051	0-7-8
076	076	0134	0-6-8
0137	0137	050	0-5-8
045	050	050	0-4-8
054	054	054	0-3-8
0134	0134	012	0-2-8
0136	0136	015	11-7-8
073	073	073	11-6-8
051	051	0135	11-5-8
052	052	052	11-4-8
044	044	044	11-3-8
0135	0135	042	11-2-8
041	041	075	12-7-8
053	053	074	12-6-8
050	050	0133	12-5-8
074	051	051	12-4-8
056	056	056	12-3-8
0133	0133	0100	12-2-8

APPENDIX B

SUMMARY OF SYSTEM STATEMENTS

1.1 NOTATION OF STATEMENTS AND OPERATIONS

Wherever a statement or operation is discussed in this appendix, a uniform system of notation is used to define the structure. This notation is not a part of CMS-2, but is a standardized notation that may be used to describe the syntax (construction) of any programming language. It provides a brief but precise means of explaining the general patterns that the language permits. It does not describe the meaning of the statements or operations; it merely describes structure; that is, it indicates the order in which the operands must appear, the punctuation required, and the options allowed.

The following rules explain this standard notation:

1. A word written in lowercase letters represents the type of entry to be made by the programmer. This word may be hyphenated.

name denotes an entry of a name.

data-unit-name denotes an entry of a data unit name.

2. A word written in uppercase letters or special characters denotes an actual occurrence of that word or character in the language.

name DISPLAY data-unit-name \$ This example denotes the entry of a name followed by the entry of the reserved word DISPLAY followed by an entry of a data unit name.

3. Braces { } are used to denote a choice. The units from which a choice may be made are stacked vertically within the braces. At least one of the units within the braces must occur in the statement.

{ REGS } This example indicates that either REGS
{ data-unit-name } or a data unit name must appear in the statement.

4. Square brackets [] are used to denote options. When one unit is enclosed in brackets, the unit may or may not appear. When more than one unit is enclosed in brackets, any one of the alternative units may

or may not be chosen to appear. In either case, it is possible that no unit may appear. It is generally not possible that more than one unit will appear.

[name]

This example indicates that a name may appear in the statement format. However, this unit is not required.

5. The use of • • • denotes that the type of entry indicated by the word preceding • • • may appear one or more times in succession, where each entry is delimited by the word preceding •. This does not imply that all entries should be identical. It does imply, however, that all entries should be the same type of entry indicated by the word preceding the three dots.

, • data unit name • • • This example indicates that one or more data unit names may occur in succession as entries, separated by commas. Thus, the following would be a legal entry: ALPHA, BETA, GAMMA

B.2 MONITOR STATEMENTS

\$SEQ, ddd [non-numeric-nospace-non-\$ character] [\$ [comments]]

\$JOB [, [user identification], [project identification], [time limit in decimal],

[page limit in decimal], $\left[\begin{array}{c} C \\ S \\ TS \end{array} \right]$, [26] [\$ [comments]]

\$E01 [\$ [comments]]

\$ENDJOB [\$ [comments]]

\$CMS-2 [\$ [comments]]

\$ASM [, U] [\$ [comments]]

\$UTILITY [\$ [comments]]

\$LIBEXEC [\$ [comments]]

\$LOAD [, •

TRACE
PTRACE
SNAP
RANGE
DISPLAY

 • • •] [\$ [comments]]

\$AREG [, •

octal number
decimal number D

 • • •] [\$ [comments]]

\$BREG [, •

octal number
decimal number D

 • • •] [\$ [comments]]

\$KEYSET [, •

ON
OFF

 • • •] [\$ [comments]]

\$CALL,

address section name	[+ { octal number }]
external definition name	[- { decimal number D }]
bound section name	[+ { octal number }]
	[{ decimal number }]

 } [\$ [comments]]

M-5035

\$TYPE, message

\$HALT, message

\$REMARK, message

\$TRA, $\left\{ \begin{array}{l} \left\{ \begin{array}{l} \text{address section name} \\ \text{external definition name} \end{array} \right\} \left[\begin{array}{l} \left\{ \begin{array}{l} + \\ - \end{array} \right\} \left\{ \begin{array}{l} \text{octal number} \\ \text{decimal number D} \end{array} \right\} \end{array} \right] \\ \text{bound section name} \left[\begin{array}{l} + \\ \end{array} \right] \left\{ \begin{array}{l} \text{octal number} \\ \text{decimal number} \end{array} \right\} \end{array} \right\} \left[\$ \text{ [comments]} \right]$

\$DUMP, $\left[\begin{array}{c} I \\ O \\ H \\ Q \\ D \\ C \\ IC \\ OC \\ HC \\ QC \\ DC \\ CC \end{array} \right], \left[\begin{array}{c} W \\ P \\ A \end{array} \right], \bullet$

$\left\{ \begin{array}{l} \left\{ \begin{array}{l} \text{address section name} \\ \text{external definition name} \end{array} \right\} \left[\begin{array}{l} \left\{ \begin{array}{l} + \\ - \end{array} \right\} \left\{ \begin{array}{l} \text{octal number} \\ \text{decimal number D} \end{array} \right\} \end{array} \right] \\ \text{bound section name} \left[\begin{array}{l} + \\ \end{array} \right] \left\{ \begin{array}{l} \text{octal number} \\ \text{decimal number D} \end{array} \right\} \end{array} \right\} \bullet \bullet \bullet \left[\$ \text{ [comments]} \right]$

$$\$SNAP, \left\{ \begin{array}{l} \left\{ \begin{array}{l} \text{address section name} \\ \text{external definition name} \end{array} \right\} \left[\begin{array}{l} \{ + \} \\ \{ - \} \end{array} \right] \left\{ \begin{array}{l} \text{octal number} \\ \text{decimal number D} \end{array} \right\} \\ \text{bound section name} \left[+ \right] \left\{ \begin{array}{l} \text{octal number} \\ \text{decimal number D} \end{array} \right\} \end{array} \right\} , \\
 \left[\begin{array}{l} I \\ O \\ H \\ Q \\ D \\ C \\ IC \\ OC \\ HC \\ QC \\ DC \\ CC \end{array} \right] , \text{ [decimal dumps count]} ,$$

$$\text{[decimal dump frequency]} , \text{ [decimal start dump count]} \\
 \left\{ \begin{array}{l} \left\{ \begin{array}{l} \text{address section name} \\ \text{external definition name} \end{array} \right\} \left[\begin{array}{l} \{ + \} \\ \{ - \} \end{array} \right] \left\{ \begin{array}{l} \text{octal number} \\ \text{decimal number D} \end{array} \right\} \\ \text{bound section name} \left[+ \right] \left\{ \begin{array}{l} \text{octal number} \\ \text{decimal number D} \end{array} \right\} \end{array} \right\} ,$$

$$\left\{ \begin{array}{l} \text{octal number of words} \\ \text{decimal number D of words} \end{array} \right\} \bullet \bullet \bullet [\$ \text{ [comments]}]$$

\$PATCH [\$ [comments]]

Patch Statement:

$$\left\{ \begin{array}{l} \left\{ \begin{array}{l} \text{address section name} \\ \text{external definition name} \end{array} \right\} \left[\begin{array}{l} \{ + \} \\ \{ - \} \end{array} \right] \left\{ \begin{array}{l} \text{octal number} \\ \text{decimal number D} \end{array} \right\} \\ \text{bound section name} \left[+ \right] \left\{ \begin{array}{l} \text{octal number} \\ \text{decimal number D} \end{array} \right\} \end{array} \right\}$$

$$\left\{ \begin{array}{l} I, \bullet \text{ up to twelve octal digits } \bullet \bullet \bullet \\ O, \bullet \text{ up to eleven octal digits } \bullet \bullet \bullet \\ K, \bullet \text{ up to thirteen octal digits } \bullet \bullet \bullet \\ C, \text{ (character string)} \end{array} \right\} [\$ \text{ [comments]}]$$

\$ASG,	SIP,	T1 T2 T3 T4 T5 T6 T7 T8 CR KB PR	[\$ [comments]]
	SOP,	T1 T2 T3 T4 T5 T6 T7 T8 CP PP	
	SHC,	T1 T2 T3 T4 T5 T6 T7 T8 HP KB	
		T1 T2 T3 T4 T5 T6 T7 T8 CR CP HP KB PR PP	
		T1 T2 T3 T4 T5 T6 T7 T8 CR CP HP KB PR PP O	

\$LDUYK-20[\$ [comments]]
\$FORTRAN [\$ [comments]]
\$SYSMAKER [\$ [comments]]

B.3 LOADER STATEMENTS

Table Size Declaration:

TSD, •name = decimal integer ••• [\$ [comment]]

Library Selection:

LIBS, •internal library name [(external library name)]
••• [\$ [comment]]

Element Selction:

SEL-ELEM [element name] [(key)][{ , ALL
, ONLY
, decimal integer }]

[\$[comment]]

Loader Options Select:

LOPTIONS • [FORCE
NOID
NOMAP] ••• [LOBJECT
,LODGO
SAVLODGO]

Combine Elements Selection:

```
[Section name] { DATAPOOL
                  TABLEPOOL
                  BASE
                  LOCDDPOOL } [ (octal digit)]
, • element name • • • [ (octal integer)]
[ $ [comment] ]
```

End Card:

END [\$ [comment]]

B.4 LIBRARIAN STATEMENTS

/LIST

/BUILD internal-library-name

```
/EDIT old-internal-library-name [ (old-external-library-name)]
[, new-internal-library-name]
```

/ENDLIB

/TAPID , • internal-tape-name [(external-tape-name)] • • •

/RELEASE , • internal-tape-name [(external-tape-name)] • • •

/HISTENT notes

```
/ADD name [(key)] [ * { S
                       O
                       L
                       C } ] [ SYSIN
                               internal-tape-name [(external-tape-name)] ]
[NOLIST]
```


M-5035

/DEP [FROM HEADER
 , ● name [(key)] ● ● ●]

/BEGINEL

/ENDEL

/CORRECT

/ENDCOR

/name [(key)] [* { S
 O
 L
 C }]

/END

/I [item-number]

/D [item-number [THRU item-number]]

/NOLIST [HIST
 DIR]

/PRINT , • name [(key)] * $\left\{ \begin{matrix} S \\ O \\ L \\ C \end{matrix} \right\}$ [item-number [THRU item-number]] • • •
 HIST
 DIR
 BOTH
 ALL
 * $\left\{ \begin{matrix} S \\ O \\ L \\ C \end{matrix} \right\}$
 (key) * $\left[\begin{matrix} S \\ O \\ L \\ C \end{matrix} \right]$

/PUNCH , • name [(key)] * $\left[\begin{matrix} S \\ O \\ L \\ C \end{matrix} \right]$ [item-number [THRU item-number]] • • •
 * $\left\{ \begin{matrix} S \\ O \\ L \\ C \end{matrix} \right\}$
 ALL
 (key) * $\left[\begin{matrix} S \\ O \\ L \\ C \end{matrix} \right]$

B.5 PERIPHERAL UTILITIES STATEMENTS

TYPE [[message to operator]] [• [comments]]

HALT [[message to operator]] [• [comments]]

MOUNT [tape name] [, [{ IN }]] [• [comments]]

DISMOUNT [tape name] [• [comments]]

FILSKP [tape name] [, [no. of files]] [• [comments]]

RECSKP [tape name] [, [no. of records]] [• [comments]]

BKFILSKP [tape name] [, [no. of files]] [• [comments]]

BKRECSKP [tape name] [, [no. of records]] [• [comments]]

REWIND [tape name] [• [comments]]

WRITE [tape name], location name [, [no. of words] [, [format]]]
[• [comments]]

WRTFILMK [tape name] [• [comments]]

DUPLICAT [tape name] , [format] , [tape name] , [format] ,
{ no. of files [, [no. of records]] }
{ no. of files] , no. of records } [• [comments]]

REFORMAT [tape name] , [format] , [tape name] , [format] , item size ,
no. of items [, [no. of files] [, [skip words]]] [• [comment]]

CARD TAPE [tape name] [• [comments]]

BOOTWRT [tape name] , location name [, [no. of words] [,
[load base] [, [M]]] [• [comments]]

CONVERT [tape name] [, [tape name] [, [data type] [, [CONTINUE]]]
[• [comments]]

READ [tape name] , location name [, [no. of words][, [format]]]
[• [comments]]

LIST tape name [, [tape format] [,

I
O
H
Q
D
C

 [, [no. of files] [, no. of records]
]]] [• [comments]]

TAPEOUT [tape name] [, [no. of files]] [• [comments]]

COMPARE [tape name] , [format] , [tape name] , [format] ,
{ no. of files [, [no. of records]] }
{ [no. of files] , no. of records } [• [comments]]

B.6 COMPILER STATEMENTS

[name] BASE [([T] [, integer-value])] [integer-value] \$

CMODE \$

CMS-2 \$

CORRECT \$

CSWITCH-ON, • name • • • \$

CSWITCH-OFF, • name • • • \$

CSWITCH, • name • • • \$

CSWITCH-DEL \$

[data-unit] DATA constant [constant] \$

[name] DATAPOOL [([T] [, integer-value])] [integer-value] \$

DEBUG , • $\left. \begin{array}{l} \text{DISPLAY} \\ \text{SNAP} \\ \text{RANGE} \\ \text{TRACE} \\ \text{PTRACE} \\ \text{DELETE} \end{array} \right\}$ • • • \$

DEP, • element-name [(key)] • • • \$

DIRECT \$

END-HEAD [name] \$

END-SYSTEM [name] \$

identifier EQUALS value \$

identifier EXCHANGE character-string \$

EXECUTIVE \$

[name] HEAD [comment] \$

[name] LOCDDPOOL [([T] [, integer-value])] [integer-value] \$

identifier MEANS character-string \$

NITEMS (name) EQUALS integer value \$

OPTIONS , • [SOURCE [({ LIST
CCOMN
CSRCE
CARDS })] ... \$
OBJECT [({ CMP
OPT
CNV
CCOMN
CARDS
COBJT
SA
CR
SM
SMA })]
LISTING [({ PRINT
CCOMN
CLIST })]
MONITOR]

M-5035

SEL-ELEM name [(key)] [, ONLY] \$

SEL-HEAD name [(key)] [, ONLY] \$

SEL-POOL name [(key)] \$

SEL-SYS [(key)] [, ONLY] \$

SPILL \$

SYS-INDEX, • b-register index-name ••• \$

[name] SYSTEM [comment] \$

[name] TABLEPOOL [([T] [, integer-value])] [integer-value] \$

TERMINATE \$

name AUTO-DD [comment] \$

[label] CHECKID file-name { STANDARD
(character-string) } [THEN statement] \$

[label] CLOSE file-name [THEN statement] \$

COMMENT comment \$

CSWITCH name \$

[label] DEFID file-name { STANDARD
(character-string) } [THEN statement] \$

[label] DECODE data-element { (, • image
image • • •) }

format-name [THEN statement] \$

[name] DISPLAY { REGS
, • data-element • • • } [THEN statement] \$

[label] ENCODE data-element { (, • image
image • • •) }

format-name [THEN statement] \$

END-AUTO-DD name \$

END vary-block-name \$

END-CSWITCH name \$

END-CSWITCHS \$

END-FUNCTION function-name \$

END-LOC-DD [name] \$

[label] ENDFILE file-name [THEN statement] \$

$$\left[\left(\begin{array}{l} \text{EXTREF} \\ \text{EXTDEF} \\ \text{TRANSREF} \end{array} \right) \right] \text{ FORMAT name , } \bullet [n][(\left. \begin{array}{l} \text{Ew.d} \\ \text{Iw.d} \\ \text{Fw.d} \\ \text{Ow.d} \\ \text{Iw} \\ \text{Aw} \\ \text{wX} \\ \text{Tw} \\ \text{Hollerith-constant} \end{array} \right)] \dots \$$$

$$\left[\left(\begin{array}{l} \text{EXTREF} \\ \text{EXTDEF} \\ \text{TRANSREF} \\ \text{LOCREF} \end{array} \right) \right] \text{ FUNCTION name (formal-input-parameters) [type] \$}$$

[label] GOTO

$$\left\{ \begin{array}{l} \text{statement-name [special-condition]} \\ \text{switch-name [index] [INVALID statement-name] [special-condition]} \end{array} \right\} \$$$

$$[\text{label}] \text{ IF } \left\{ \begin{array}{l} \text{table-element } \left\{ \begin{array}{l} \text{VALID} \\ \text{INVALID} \end{array} \right\} \\ \text{relational-expression} \\ \text{Boolean-data-element} \\ \text{Boolean-expression} \\ \text{DATA } \left\{ \begin{array}{l} \text{FOUND} \\ \text{NOTFOUND} \end{array} \right\} \\ \text{data-unit } \left\{ \begin{array}{l} \text{EVENP} \\ \text{ODDP} \end{array} \right\} \end{array} \right\} \text{ THEN statement \$}$$

END-PROC procedure-name \$

$$\text{END-SWITCH } \left\{ \begin{array}{l} \text{switch-name-a [, switch-name-b]} \\ \text{procedure-switch-name-a [, procedure-switch-name-b]} \end{array} \right\} \$$$

END-SYS-DD name \$

END-SYS-PROC name \$

END-SYSTEM name \$

END-TABLE table-name \$

END-TRACE \$

FIELD name	[F	[(R)]]
		B		
		H	number-of-characters	
		S	list-of-status-constants	
		A	number-of-bits { S } number-of-fractional-bits	
		I	number-of-bits { S } { U }	

[word-location starting-bit-position] [P constants] [V(x,y)] \$

[({	EXTREF)]	FILE name	{	H	}	maximum-number-of-records	{	R	}
			EXTDEF					B				V	
			TRANSREF					S					

maximum-record-size hardware-name [states] [WITHLBL] \$

[label] FIND expression [VARYING loop-index]

[FROM initial-value] { THRU final-value } [BY [±] increment]] \$
 WITHIN name

NOTE

FROM, THRU, WITHIN, and BY may appear in any order.

[label] INPUT { filename
OCM
READ
PRINT
PUNCH } { receptacle
(, • receptacle • • •) }

[format name] [THEN statement] \$

[({ EXTREF
EXTDEF
TRANSREF })] ITEM-AREA , • name • • • \$

[({ EXTREF
EXTDEF
TRANSREF })] LIKE-TABLE name [number-of-items]

[major-index-name] \$

[name] LOC-DD [comments] \$

LOC-INDEX , • name • • • \$

MODE { variable declaration
field declaration }

[label] OPEN filename { INPUT
OUTPUT
SCRATCH } [THEN statement] \$

data-unit OVERLAY , • data-unit • • • \$

[label] OUTPUT { filename
OCM
PRINT
PUNCH
READ } { data-image
(, • data-image •••) }

[format name] [THEN statement] \$

[label] PACK data-unit WITH , • data-unit •••
[THEN statement] \$

[({ EXTREF
EXTDEF
TRANSREF })] P-SWITCH name [(variable)] [[INPUT
, name
, • formal-parameter •••] [OUTPUT , • formal-parameter •••]] \$

[label] procedure-switch-name [USING index]
[INVALID statement-name] [INPUT , • actual-parameter •••]
[OUTPUT , • actual-parameter •••] [THEN statement] \$

[({ EXTREF
EXTDEF
TRANSREF
LOCREF })] PROCEDURE name [INPUT , • formal-parameter •••]
[OUTPUT , • formal-parameter •••]
[EXIT , • abnormal-exit-name •••] \$

M-5035

[label] procedure-name [INPUT , • actual-parameter • • •]
[OUTPUT , • actual-parameter • • •] [EXIT , • statement-name • • •]
[THEN statement] \$

{ variable-name
field-reference } RANGE upper-value [... lower-value] \$

[label] RESUME vary-block-name \$

[label] RETURN [abnormal exit] [special-condition] \$

[label] SEARCH file-name $\left\{ \begin{array}{l} \text{FW} \\ \text{BW} \end{array} \right\}$ [CONTIN] data-unit NOFIND name
[THEN statement] \$

[label] SET , • data-element • • • TO expression [THEN statement] \$

[label] SHIFT data-unit $\left\{ \begin{array}{l} \text{CIRC} \\ \text{ALG} \\ \text{LOG} \end{array} \right\}$ count [INTO data-unit]
[THEN statement] \$

name SNAP data-element \$

[label] STOP [THEN statement] \$

$$\left[\left(\begin{array}{c} \text{EXTREF} \\ \text{EXTDEF} \\ \text{TRANSREF} \end{array} \right) \right] \text{SUB-TABLE name initial-item-number}$$

maximum-number-of-items [major-index-name] \$

[label] SWAP data-element , data-element [THEN statement] \$

$$\text{SWITCH} \left\{ \begin{array}{l} \text{name} \\ \text{name-a,} \end{array} \left[\begin{array}{l} \text{(variable-name)} \\ \text{, • switch-point • • •} \\ \text{name-b} \end{array} \right] \right\} \$$$

$$\left\{ \begin{array}{l} \text{statement-name} \\ \text{procedure-name} \end{array} \left[\begin{array}{l} \text{, statement-name} \\ \text{, procedure-name} \end{array} \right] \right\} \$$$

constant , statement-name \$

constant , procedure-name \$

name SYS-DD [comment] \$

name SYS-PROC [comment] \$

name SYS-PROC-REN [comment] \$

[({ EXTREF
EXTDEF
TRANSREF })] TABLE name

[{ H { (type)
words-per-item
NONE
V MEDIUM
DENSE } [INDIRECT] number-of-items [major-index-name] \$
A { (type)
words-per-item
NONE
MEDIUM
DENSE } [INDIRECT] , • dimension • • • }]

label VARY loop-index [FROM initial-value]

{ THRU final-value } [BY [+] increment] \$
{ WITHIN name }

NOTE

FROM, THRU, WITHIN and BY may appear in any order.

[({ EXTREF
EXTDEF
TRANSREF })] VRBL { name
(, • name • • •) }

[I number-of-bits { S }
U }
A number-of-bits { S } number-of-fractional-bits
U }
S list-of-status-constants
H number-of-characters
B
F [(R)]]

[P constant] [V(x,y)] \$

M-5035

Equate Directive Card:

label EQU e

EVEN Directive Card:

EVEN

ODD Directive Card:

ODD

FORM Directive Card:

label FORM , • {e} • • •

List Cross Reference Table Directive Card:

LCR

Library Select Directive Card:

LIBS , • {internal-name} [(external-name)] • • •

Library Element Directive Card:

LIB , • [name] [(version)] • • •

LINK Directive Card:

LINK , • {name} • • •

LIST Directive Card:

LIST

ELIST Directive Card:

ELIST

M-5035

Equate Directive Card:

label EQU e

EVEN Directive Card:

EVEN

ODD Directive Card:

ODD

FORM Directive Card:

label FORM , • {e} • • •

List Cross Reference Table Directive Card:

LCR

Library Select Directive Card:

LIBS , • {internal-name} [(external-name)] • • •

Library Element Directive Card:

LIB , • [name] [(version)] • • •

LINK Directive Card:

LINK , • {name} • • •

LIST Directive Card:

LIST

ELIST Directive Card:

ELIST

NOLIST Directive Card:

NOLIST

Literal Directive Card:

$\left[\begin{array}{l} \$ (e) [, label] \\ label \end{array} \right] \text{LIT}$

Punch External Labels Directive Card:

PXL

Reserve Directive Card:

[label] RES e

Relocation Field Directive Card:

RF\$ e , • {h,l,c} • • •

Segment End Directive Card:

SEGEN

Set Address Directive Card:

SETADR e , • [a] • • •

Word Directive Card:

WRD e

M-5035

MACRO Directive Card:

label [*] MACRO [e,[,e₂]]

MACRO Name Directive Card:

label [*] NAME [e]

GO Directive Card:

GO label

APPENDIX C
SUMMARY OF SERVICE ROUTINE CALLING SEQUENCES

Request Packets Reference Words

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
4												B mod. 0	S mod. 0	Y portion of pkt. address																	

IW y,b,s

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0												S mod. 0	Y portion of pkt. address																		

IWS y,s

Standard Input Request Packet

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
function code												status						user code													
data address (in indirect word format)																															
number of characters																															

Standard Output Request Packet

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
																user code															
data address (in indirect word format)																															
number of characters																															

Standard Hardcopy Output Request Packet

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
data address (in indirect word format)																															
number of characters																															

M-5035
Change 3

<u>Service Request Call</u>	<u>Type of Packet</u>
Standard Output Request XS 11 ₈ IW (packet address)	Standard output request packet
Standard Hardcopy Request XS 12 ₈ IW (packet address)	Standard hardcopy request packet
Tape Assignment/Release Request XS 3 IW (packet address)	Tape Assignment/Release packet
Tape Control Selection Request XS 2 IW (packet address)	Tape Control Selection packet
Get Device Name Request XS 1 IW (packet address)	Get Device Name packet
Check Simulated Jump Key 1 Selection Request XS 40 ₈ return if simulated key 1 is "on" return if simulated key 1 is "off"	None
Check Simulated Jump Key 2 Selection Request XS 41 ₈ return if simulated key 2 is "on" return if simulated key 2 is "off"	None

Service Request Call

Type of Packet

Check Simulated Jump Key 3 Selection Request

XS 42₈

None

return if simulated key 3 is "on"

return if simulated key 3 is "off"

Check PTRACE Selection Request

XS 43₈

None

return if not selected

return if selected

Check TRACE Selection Request

XS 44₈

None

return if not selected

return if selected

Check DISPLAY Selection Request

XS 45₈

None

return if not selected

return if selected

Check RANGE Selection Request

XS 46₈

None

return if not selected

return if selected

Check SNAP Selection Request

XS 47₈

None

return if not selected

return if selected

Check Floating Point Error Request

XS 50₈

None

return if floating point error has occurred

return if no floating point error has occurred

Terminate User Program Request

XS 51₈

None

or

XS 52₈

M-5035
Change 3

Service Request Call

Type of Packet

Current Time Request

XS 60₈

None

IW (address where data is wanted) - data occupies one word

Current Data Request

XS 61₈

None

IW (address where data is wanted) - data occupies two words

APPENDIX D

CMS-2 COMPILER RESERVED WORD LIST

The following symbols are compiler reserved words and may not be used as identifiers in a CMS-2 program.

ABS	DECODE	FUNCTION	OCM	SPILL
ALG	DEFID	GOTO	ODDP	STOP
AND	DENSE	GT	ONLY	SWAP
BASE	DEP	GTEQ	OPEN	SWITCH
BEGIN	DIRECT	[†] H	OPTIONS	SYSTEM
BIT	DISPLAY	HEAD	OR	TABLE
BY	ELSE	IF	OUTPUT	THEN
CAT	ENCODE	INDIRECT	OVERFLOW	THRU
CHAR	ENDFILE	INPUT	OVERLAY	TO
CHECKID	END	INTO	PACK	TRACE
CIRC	EQ	INVALID	POS	UNTIL
CLOSE	EQUALS	LENGTH	PRINT	USING
CMODE	EVENP	LIBS	PTRACE	VALID
CNT	EXCHANGE	LOG	PUNCH	VARY
COMMENT	EXEC	LT	RANGE	VARYING
COMP	EXIT	LTEQ	READ	VRBL
CORAD	FIELD	MEANS	REGS	WHILE
CORRECT	FIL	MEDIUM	RESUME	WITH
CSWITCH	FILE	MODE	RETURN	WITHIN
[†] D	FIND	NITEMS	SAVING	XOR
DATA	FOR	NONE	SET	
DATAPool	FORMAT	NOT	SHIFT	
DEBUG	FROM	[†] O	SNAP	

[†] Not allowed as a tabular identifier or function.

M-5035
Change 2

NOTE

Programmers expecting to make use of CMS-2 run-time routines (high-level input/output, debug, and math routines) should avoid the use of identifiers beginning with RT to prevent duplication of identifiers. Any duplication of identifiers, globally defined within the run-time library, would not be detected until load time.

APPENDIX E

COMPILER ERROR MESSAGES AND LIMITS

Source Errors and Warning Messages (Warnings Flagged with *)

The following error messages appear as a result of errors detected by the source syntax analysis phase of a compilation (OPTIONS SOURCE):

- *0 **DEBUG REQUIRES MONITOR**
 A DEBUG statement is only processed if the MONITOR OPTION has been specified.

- 1 **IDENTIFIER GT 8 CHARS**
 Attempt to define an identifier greater than eight characters long.

- 2 **LITERAL GT 132 CHARS**
 Hollerith or literal constant greater than 132 characters.

- 3 **RESERVED WORD USED AS ID**
 Illegal use of reserved word as an identifier.

- 4 **CHARACTER NOT RECOGNIZED**
 Illegal ASCII input character.

- 5 **DECIMAL POINT MISPLACED**
 Erroneous use of period not in a constant or illegal label definition.

- *6 **COMMENT TERMINATED BY \$**
 Double prime comment not completed before end of statement.

- 7 **INCORRECT OCTAL CONSTANT**
 The digits 8 or 9 appear in an octal constant.

- 8 **MISPLACED SEL-POOL**
 Identifier definition appears prior to SEL-POOL statement.

M-5035
Change 3

- 9 ILLEGAL INTEGER TAG
Symbol or constant must be an integer.

- 10 NO STATEMENT TERMINATOR
Missing \$ statement terminator.

- 11 IDENTIFIER MISSING
Missing name in a data unit declaration.

- 12 DUPLICATE IDENTIFIER
Attempt to define a previously defined identifier.

- 13 OUTSIDE TABLE BOUNDS
Subtable not contained within table; field not contained within item;
or multi-word field in a horizontal table.

- 14 NO DESCRIPTIVE OPERATOR
Missing descriptive or separator term.

- 15 ILLEGAL IN MINOR HEADER
Statement is not allowed in minor header (must be placed in major header).

- 16 TOO MANY DIMENSIONS
More than seven dimensions in an array declaration.

- *17 COMMA MISSING
Comma missing in statement.

- *18 OVERLAY PARENT MISMATCH
Data units on right of overlay exceed size of parent data unit.

- 19 DUPLICATED OVERLAY
Data unit appears on right of more than one overlay.

- 20 OVERLAY SEQUENCE ERROR
Data unit on right of overlay appeared on left of previous overlay.
- 21 UNDEFINED IDENTIFIER
Referenced data unit has not been previously defined.
- 22 SCOPE CONFLICT
Definition of a global procedure with local parameters.
- 23 STATEMENT NOT RECOGNIZED
Statement placed in the wrong type of element or within the wrong type of declarative brackets.
- 24 ILLEGAL OPTIONS
CCOMN has been designated the output unit for both LISTING and OBJECT options - options that are produced simultaneously. Only the OBJECT option is produced; the LISTING option on CCOMN is ignored.
- *25 PARENTHESIS MISSING
Parenthesis missing within statement.
- 26 SUB/LIKE-TABLE PROHIBIT
Subtable or like-table declared in an array.
- 27 ILLEGAL OVERLAY DATAUNIT
Illegal data unit appears in overlay.
- 28 ILLEGAL OVERLAY PARENT
Specified data unit may not be used as a overlay parent.
- 29 DUPLICATE RANGE
More than one range statement for the same data unit.
- 30 PRESET NOT ALLOWED
Data unit preset not allowed in AUTO-DD or on an externally referenced data unit.

31 ILLEGAL HARDWARE NAME

Illegal hardware device name specified in file declaration.

32 ILLEGAL FORMAT DESCRIPTOR

Illegal conversion descriptor specified in format statement.

33 MORE THAN 1 LEVEL NESTED

Format descriptors nested (parenthesized) to more than one level.

34 ILLEGAL IDENT REFERENCE

Illegal use of identifier in an expression.

35 ILLEGAL SIZE DESCRIPTOR

Illegal data unit size attribute (e. g., Hollerith over 132 characters, table defined with 0 words per item, too many bits for I- or A-type.)

*36 END BRACKET MISPLACED

Misplaced or missing END- statement.

*37 STATEMENT REQUIRES MONITOR

MONITOR OPTION must be declared for processing of this statement.

*38 CNV REQUIRES NONRT

Processing of a CNV statement requires the NONRT (or MONITOR) OPTION.

39 SYSTEM LIMIT nn EXCEEDED

One of the following compiler limits denoted by nn has been exceeded. The code nn has the following values:

1. Constant conversion limit was exceeded; the value of the constant lies outside the decimal limits (1E57, 1E-38) or the octal limits (1E77, 1E-52).
2. The number of nested subexpressions within the Boolean condition of an IF statement may not exceed ten.
3. The number of libraries requested for retrieval may not exceed ten.

4. The number of operands in a DISPLAY statement has exceeded the compiler limit. The card column indicator in the error output listing points to the operand which first exceeds the limit. This and following operands should be written as a separate DISPLAY statement. The limit may be calculated as follows:
 - a) Allow $3 + n$ words for each operand, where n is the number of words required to contain the operand as a character string;
 - b) the sum of the above may not exceed 94.
5. The maximum number of exit parameters per PROCEDURE declaration is ten.
6. The number of format descriptors exceeds 94 or the number of operands of an input/output list for INPUT, OUTPUT, ENCODE or DECODE statements exceeds 94. (For each operand that is a Hollerith constant, also add in the number of words required to contain the constant value.)
7. A maximum of seven levels of subscripting and function calls per operand is allowed.
8. An item beyond item 255 was specified in a field preset.
9. The length of a dynamic statement is too long for the compiler to process properly. This may be due to the complexity of an expression or an abundance of embedded notes.
10. The maximum number of elements declared dependent of another is 58.
11. A VRBL declaration may define no more than 25 names.
12. The offset of a sibling overlaid data unit relative to its parent data unit must not exceed 65535 words.
13. More than 250 elements.
14. Symbol table overflow -- number of global and local identifiers.
15. Field defined or compiler packed beyond word 255.
16. More than 15 combined nested VARY and BEGIN blocks.

17. More than 10 nested VARYs.
18. COMMENT (or "-type comment) between FIND and IF DATA too long.
19. COMMENT (or "-type comment) between last THEN clause and ELSE clause too long.
- 20 through 29. Not used.
30. A dependent retrieval level greater than 255 was requested. 255 is assumed.
31. A magnitude value greater than 32767 was specified in a magnitude specification. 32767 is assumed.

- *40 OVER 10 NESTED CSWITCH BRACKETS.
Nesting of CSWITCH brackets exceeded.
- 41 ILLEGAL EXTERN MODIFIER
Illegal or misplaced EXTREF, EXTDEF or LOCREF declaration.
- *42 END DECLARATION MISSING
Missing program structure END- declaration.
- 43 HEADER NOT RECOGNIZED
Unrecognizable or illegal statement appearing in a header.
- *44 END HEAD MISSING
END-HEAD statement missing after major header.
- *45 FUNCTION RETURN MISSING
Return statement missing from function.
- 46 ILLEGAL EXIT PARAMETER
Illegal name specified as a formal exit parameter.
- 47 COMPOOL REQUEST IGNORED
Requested COMPOOL generation run not made due to detection of SYS-PROC statement.
- 48 INCOMPATIBLE DATAUNIT
Expression operands do not conform to data unit type restrictions or do not fit context required by operator.
- *50 NO DEF CHECK PERFORMED
No structural compatibility checking between external compiler-packed table definitions.

- 51 FILE TYPE MISSING
Type descriptor missing in file declaration.
- * 52 CMS-2 BRACKET MISSING
CMS-2 statement missing as terminator for a direct code block.
- 53 VALUE SIGNIFICANCE LOST
The most significant bits have been lost during alignment of a numeric constant used as a variable or field preset or a value block value.
- 54 TOO MANY STATUS CONSTANT
More than 12 status constants associated with a data unit.
- 55 DUPLICATE ALLOCATION
Symbol appears on the left of more than one EQUALS statement.
- 56 ILLEGAL ALLOCATION
Attempt to establish EQUALS allocation via a constant (absolute allocation) or illegal EQUALS expression.
- 57 LIBS NOT DEFINED PRIOR
Library selection statement appearing prior to a LIBS statement.
- 58 nnnnnnnn NOT RETRIEVED
The request element, named nnnnnnnn, was not found in any of the declared libraries.
- *59 FIELD LIST MISSING
No fields specified for a compiler packed table.
- 60 WRONG ARGUMENT COUNT
Procedure call parameter list mismatched.

- 61 FORMAT NOT INDICATED
Missing format statement reference in ENCODE/DECODE statement.
- 62 WORD MORE THAN 12 CHARS
Name, identifier, or symbol more than 12 characters.
- 63 MUST BE FORMAT NAME
Syntax requires name to be a format statement reference.
- * 64 WRONG END NAME
Incorrect name on END- statement.
- 65 SYNTAX
Erroneous statement syntax or punctuation.
- 66 COMPILER PROBLEM, SYNTAX
Syntax of statement cannot be analyzed by compiler.
- * 67 INCORRECT END KEYWORD
Wrong END- bracket statement used.
- * 68 NO SYSTEM DECLARATION
Missing SYSTEM declaration as first statement of source input.
- * 69 NO END-SYSTEM
Missing END-SYSTEM statement
- * 70 SYNTAX
Syntax of statement not correct.
- * 71 OPTIONS STMT MISSING
Options statement missing from major header. Syntax diagnostics will be only output.

- * 72 PARAM PROCESSED AS VRBL
 Parameters are not allowed in function definitions.

- 73 PROCEDURE I/O LIST ERROR
 Procedure call parameters do not match procedure definition.

- 74 ILLEGAL KEY TYPE
 Key type not legal for this element.

- 75 DUPLICATE KEY
 KEY previously declared.

- 76 ELEMENT KEY GREATER 4 CHS
 Library element key is greater than four characters.

- * 77 MISPLACED STATEMENT
 A misplaced or extraneous END statement has been encountered at a point
 in the program where all VARY declarations and their END delimiters have
 been paired.

- * 78 VALUE PRECISION LOST
 The least significant bits have been lost during alignment of a numeric constant
 used as a variable or field preset or as a value block value.

- 79 ILLEGAL DECREMENT WITHIN
 Illegal VARY containing explicit FROM and WITHIN parameters with a
 negative BY parameter.

- * 80 32D UNSIGNED DATA UNIT
 Variable is 32 bits unsigned (requiring 2 words).

M-5035
Change 4

- 81 ILLEGAL FORWARD REF
Forward reference PROCEDURE and FUNCTION calls may not have STATUS constants as input or output parameters.
- 82 TOO MANY PARAMS LISTED
More parameters listed in call than specified in definition.
- * 83 TRUNCATED TO INTEGER
Scaled value has been truncated to an integer value where integer is syntactically required.
- 84 MEANS OR EXCHANGE GT 132
Character string in MEANS or EXCHANGE statement greater than 132 characters.
- 85 NESTED MEANS OR EXCHANGE
Referenced MEANS or EXCHANGE name contains another MEANS or EXCHANGE name in its substitution string.
- * 86 NON-STRUCTURED STATEMENT
The current statement violates CMS-2 structured programming conventions.
- * 87 CONSTANT PRECISION LOST
Precision bits of converted constant in the decimal range of 1E-24 to 1E-38 or the octal range of 1E-32 to 1E-52 have been lost.
- 88 ILLEGAL EQUALS
Illegal operator or operands in EQUALS expression.
- * 89 ILLEGAL LEVEL REQUEST
LEVEL OPTION argument must be 0 or 1.
- 90 DUPLICATE REGISTER
PARAMETER registers duplicated.

- 91 VARY VRBL IS THRU VALUE
VARY loop index is the same data unit as the THRU clause data unit;
hence, an illogical VARY statement.
- 92 DEFINITION MISMATCH
External definition or external reference does not match previous external
definition or reference.
- 93 DUPLICATE SYS-INDEX
Definition of an index register as a system index which has already been
declared as a system index.
- * 94 IDENTIFIER EXTERNALIZED
Local identifier definition which has been made global because of a
previous external reference.
- 95 STATUS CONSTANT GT 8 CHAR
More than eight characters specified in a status constant.
- 96 MONITOR CONTROL READ
Missing TERMINATE statement; hence Compiler attempts to read a
Monitor control card.
- 97 TYPE NOT SPECIFIED
A FOR-type was not specified for a FOR-expression which requires an
explicit type specification.
- 98 ERROR LIMIT EXCEEDED
More than 100 syntax errors if options OBJECT requested or more than
1000 syntax errors if options SOURCE requested.

99 DUPLICATE VALUE

The same value was specified more than once for the same FOR block.

100 VALUE MISSING

A value is not present on the BEGIN statement of a value block.

101 VALUE BLOCK MISSING

A BEGIN with associated value is not present following a FOR statement or a value block which is not the last value block of a FOR block.

102 INCOMPATIBLE TYPE

The type of an operand (numeric, Boolean, status, or Hollerith) is not compatible with its associated operator or operator.

103 MISPLACED VALUE BLOCK

A BEGIN with an associated value is present in a context other than immediately following a FOR statement or another value block.

104 CONDITIONAL NOT BLOCKED

A decision statement not enclosed within BEGIN-END brackets is present in the compound statement of another decision statement.

■ *105 UNCOMPLETED CONDITIONAL

The compound statement of a decision statement was not completed at the end of the containing block, procedure, or function.

■ *106 CONSTANT TRUNCATED

The rightmost characters have been truncated during alignment of a Hollerith constant used as a variable or field preset or as a value block value.

107 ILLEGAL REGISTER

A register other than 0 through 7 was specified as a PARAMETER register.

***109 NO END-CSWITCH xxxxxxxx**

Named CSWITCH does not have a corresponding END-CSWITCH bracket declaration.

***110 NO CSWITCH FOR THIS END**

END-CSWITCH bracket declaration does not have corresponding CSWITCH bracket declaration.

Object Errors and Warning Messages (Warnings Flagged with *)

The following additional error messages may appear in a compilation which goes through the object code generation phase (OPTIONS OBJECT):

200 INCOMPATIBLE DATA TYPES

Attempted assignment or comparison of incompatible data unit type.

201 ILLEGAL OPERAND REF

Operand reference illegal in context used in statement.

*202 ABS OF UNSIGNED DATA

Absolute value of unsigned data unit requested.

203 DIRECT CODE SYNTAX ERROR

Illegal or undefined operand, operator, or separator in a direct code statement.

204 SYSTEM LIMIT nn EXCEEDED

One of the following compiler limits denoted by nn has been exceeded. The code nn has the following values:

20. The allocation table for generated labels has overflowed. A maximum of 1000 generated labels per system procedure is allowed. This error may also occur for cases of more than 96 generated labels for a given procedure.
21. Compiler use and allocation of temporary words have exceeded certain limits which, depending upon the distribution of temporary word usage and number of procedures, range from 2460 to 3840 temporary words per system procedure.
22. A maximum of 1536 binary constants can be generated per system procedure.
23. A maximum of 4800 words of Hollerith constants can be generated per system procedure.
24. A maximum of 4000 indirect words can be generated per system procedure.

- 205 **REMAINDER NOT AVAILABLE**
 SAVING remainder specified in statement without fixed-point division.
- 206 **STMT REQUIRES NONRT OPT.**
 Run-Time call will be generated. This requires the NONRT (non-real time)
 option to be present. It is present by default if the MONITOR option is used.
- 207 **EXTERNAL DEF MISMATCH**
 External reference does not match subsequent external definition.
- 208 **UNDEFINED IDENTIFIER**
 Forward reference to an identifier which is not subsequently defined.
- 209 **SYSTEM ERROR**
 Notify CMS-2Y maintenance personnel.
- 210 **COMPILER ERROR**
 Compiler or undetected hardware error.
- 211 **TRANSREF IN P-SWITCH**
 Illegal transient reference to procedure in a P-SWITCH.
- 212 **TOO MANY FRACTION DIGITS**
 Too many fractional digits specified in a direct code constant.
- 213 **NON-NUMERIC CONSTANT**
 Illegal constant or improper punctuation in a direct code statement.
- 214 **TOO MANY CHARACTERS**
 Illegal MEANS or EXCHANGE character substitution in a direct code
 statement.
- 215 **ILLEGAL CHARACTER**
 Illegal ASCII character appearing in a direct code statement.

- 216 UNRESOLVED EQUALS STMT
Reference to an EQUALS tag which is not resolvable at the time of reference.
- 217 ILLEGAL FORM PARAMETER
Illegal parameter in a direct code FORM statement.
- 218 FORM LABEL MISSING
Label missing from direct code FORM statement.
- *219 RIGHT TERM TRUNCATED
Truncation of operand has occurred.
- 220 ILLEGAL SPECIAL COND
Illegal STOP special condition specified on GOTO or RETURN statement.
- 221 COMPILER PROBLEM, SYNTAX
Syntax of statement cannot be analyzed by compiler.
- 222 PARAMETER TRANSFER ERROR
Statement results in alteration of contents currently held in PARAMETER register.

Allocation Errors

The following codes may appear on the output listing to flag allocation errors:

- A Allocation error. Reference to an undefined label or incorrect program allocation.
- C Compiler error. Incorrect instruction generation or undetected hardware error.

Library Retrieval Errors, Messages and Operator Messages

See paragraph 3.4.2.3 and 3.4.2.4 in Volume I.

Compiler Limits

In addition to the various compiler limits given in the explanation of the preceding error messages, the following limits are described here:

Generated Indirect Words

Compiler generated indirect words are locally defined within system procedures. A maximum of 4,000 indirect words can be generated per system procedure. These words are grouped into 100 blocks of 40 words per block. Within each block, all indirect words are unique.

Generated Binary Constants

Compiler generated binary constants are locally defined within system procedures. A maximum of 1,536 binary constants can be generated per system procedure. These constants are grouped into 32 blocks of 48 constants per block. Within each block, all constants are unique.

Generated Hollerith Constants

Compiler generated hollerith constants are locally defined within system procedures. A maximum of 4,800 words of Hollerith constants can be generated per system procedure. These constants are grouped into 32 blocks of 150 constants per block. Within each block, all Hollerith character strings are unique.

Identifiers

The compiler dictionary for user defined identifiers contains a minimum of 10,240 words and a maximum of 32,768 words. The minimum sized dictionary is standard for a 49K AN/UYK-7 configuration. The number of dictionary words per identifier is variable depending upon type of identifier. The normal entry is six to nine words (see Table E-1). The dictionary is divided into a global segment and a local segment. Table overflow occurs whenever the size of the global segment plus the global hash table plus the largest local segment exceeds the size of the dictionary. The compilation is terminated at this point.

TABLE E-1 DICTIONARY ENTRIES

<u>Types of Identifier Dictionary Entries</u>	<u>Number of Words Per Entry</u>
1) abnormal exit name	$n = 5$
2) auto data name	$n = 7$
3) cswitch name	$n = 3$
4) data pool name	$n = 8$
5) display label	$n = 6$
6) equals name	$n = 5 + 2p$
7) dummy loop label	$n = 4$
8) exchange name	$n = 5 + \left\lceil \frac{C+3}{4} \right\rceil$
9) field name	$n = 6 + 2s$
10) file name	$n = 8 + 2s$
11) for block	$n = 3 + 2s$
12) format name	$n = 7$
13) form label	$n = 7$
14) function name	$n = 7 + i$
15) header name	$n = 4$
16) index switch name	$n = 7$
17) item-area name	$n = 8$
18) item switch name	$n = 7$
19) like-table name	$n = 9$
20) local data name	$n = 7$
21) local index name	$n = 7$
22) local pool name	$n = 8$
23) major index name	$n = 8$
24) means name	$n = 5 + \left\lceil \frac{C+3}{4} \right\rceil$
25) nitems name	$n = 5$
26) procedure index switch name	$n = 7 + \max(i, o)$
27) procedure item switch name	$n = 8 + \max(i, o)$
28) procedure name	$n = 8 + \max(i, o)e$
29) program base name	$n = 8$

<u>Type of Identifier Dictionary Entries</u>	<u>Number of Words Per Entry</u>
30) ranged data name	$n = 9$
31) statement name	$n = 7$
32) sub-table name	$n = 9$
33) system data name	$n = 4$
34) system index name	$n = 4$
35) system name	$n = 4$
36) system procedure name	$n = 7$
37) table name	$n = 9 + d + t (1 + 2s)$
38) table pool name	$n = 8$
39) value block	$n = 5$
40) variable name	$n = 7 + f + 2s$

Legend:

c = number of characters; $\left[\frac{C + 3}{4} \right]$ is an integer such that $\left[\frac{C + 3}{4} \right] \leq \frac{C + 3}{4}$

d = zero for horizontal or vertical tables

= number of dimensions for an array

e = number of exit parameters

f = 1 for fixed-point data

= 0 otherwise

i = number of input parameters

n = number of dictionary words per name

o = number of output parameters

p = number of names in the equals expression that are relocatable

s = number of status constants

t = 1 with table typing

= 0 without table typing

Generated Labels

Compiler generated statement labels are locally defined within system procedures. A maximum of 1,000 generated labels per system procedure is allowed. Each procedure is limited to 128 generated labels; the worst case is 96 generated labels for a procedure that follows one with 32 or fewer generated labels. Whenever one of these limits is exceeded, the remaining generated labels assigned to the offending procedure or system procedure will be unallocated.

Generated Temporary Words

Compiler generated temporary words are locally defined within system procedures. A maximum of 32 blocks of temporary words can be generated per system procedure. The maximum number of temporary words per block is 120. The length and starting address of a temporary block is established at the end of a procedure if the number of reserved temporary words is at least 80 or at the end of the system procedure. The number of temporary words per procedure is at most 120 and 40 for the worst case. Depending upon the distribution of temporary words and the number of procedures, the best case maximum number of temporary words per system procedure is 3,840, the worst case is 2,460.

Cross Reference Errors

*****COMPLETE GLOBAL CROSS REFERENCE UNAVAILABLE -- TOO MANY ELEMENTS*****

Whenever the number of elements per compile is 160 or greater, a global cross reference cannot be produced. The local cross reference for each element will still be available and printed. The major header and all system data designs and system procedures are counted as elements; minor headers are not included in the element count since they are considered as part of the succeeding system element.

*****LOCAL CROSS REFERENCE INCOMPLETE*****

This message may appear at the end of a local cross reference listing. It indicates that the table for collecting reference data overflowed and no more references for that element were collected.

*****ERROR ENCOUNTERED DURING CROSS REFERENCE DATA COLLECTION*****

This message may appear at the end of a local cross reference listing. It indicates that an invalid condition was detected during cross reference data collection and was caused by a compiler error.

*****TOO MANY IDENTIFIERS FOR SORTING -- NOT OUTPUT GIVEN*****

This message informs the user that he will not receive his requested cross reference or symbol analysis output because there are too many identifiers for the compiler alphabetized identifier table.

Tape Errors

The user is notified of tape related errors detected during compilation. Error messages are provided on the standard hardcopy device and on the operator communication device.

*****MAG TAPE ERROR UNIT Tn name

This message is printed on the standard hardcopy device for any unrecoverable hardware errors or compiler detected checksum errors. Tn identifies the offending tape unit and the name identifies which compiler output tape or compiler scratch tape encountered the error. The normal range of n in Tn is 1 through 8. The tape names are: compiler outputs, CCOMN, CSRCE, COBJT, CLIST; compiler SCRATCHm (internal compiler scratch numbers m = 1 . . . , 4); and COMPOOL input (for COMPOOL and TO, the SEL-POOL request could not be satisfied; if Tn is a normal tape unit, a bonafide error was detected).

*****COMPILE TERMINATED

This companion message to the one above states that the tape error or the checksum error detected against one of the compiler scratch tapes necessitates termination of the compilation.

M-5035
Change 3

*****OUTPUT TERMINATED

This companion message to the first one above states that the tape error detected against CCOMN, CSRCE, COBJT, CLIST, or COMPOOL necessitates that output to that tape be discontinued. Compilation and optional outputs to other tapes will be continued.

*****WRITE ERROR Tn

This message on the operator communication device informs the operator or user that an unrecoverable hardware error has been detected during a write operation to tape. Tn designates individual tape units T1 through T8.

*****CHECKSUM ERROR Tn

This message on the operator communication device informs the operator or user that an unrecoverable hardware error or a compiler detected checksum error has been detected during a read operation from tape. Tn designates individual tape units T1 through T8. (A TO tape designator may indicate a SEL-POOL request that could not be satisfied. The assigned tape number is not available to the compiler under such circumstances.)

Run-Time Errors

The following error messages are printed by the run time programs during execution of the CMS-2 debug and high-level I/O operations.

During a Debug DISPLAY or SNAP operation, the following error messages are printed in place of the expected results:

** INVALID DATA REFERENCE **

An illegal data unit was referenced. Legal data units are an array, field, item, REGS, sub-table, table, and a word reference.

** INVALID STATUS **

A data unit, with status attributes, has a status value without its corresponding status constant.

**** INVALID USE OF MAGNITUDE ****

A magnitude was specified on a non-numeric data element.

Errors detected by the run-time high-level I/O operations are flagged by a one line location message and a one line description message. The run time abort message will repeat the location message. The location message has the following format:

ERROR IN RTxxxx RTDSL AT yyyyyyy LOGICAL UNIT zzzzzz.

where RTxxxx - Name of run-time routine detecting the error.

yyyyyyy - Core address of the first entry in the Run-Time Data Specification List (RTDSL).

zzzzzz - Hardware device name.

The error description message, with its symptoms and disposition, are as follows:

ALREADY OPEN

A file was OPENed twice without an intervening CLOSE statement - job abort.

BAD READ FOR FORMATTED INPUT

An unrecoverable hardware error has invalidated the input data prior to internal conversion - job abort.

BAD WRITE FOR FORMATTED OUTPUT

An unrecoverable hardware error is detected while outputting a formatted message - job abort.

BUFFER SMALLER THAN DATA UNIT

The formatted buffer or source data unit is smaller than the decoded target unit during a DECODE or formatted INPUT operation - job abort.

The cumulative character position counter of the converted data unit is larger than the literal buffer or target unit on an ENCODE or formatted OUTPUT operation - job abort.

DATA EXHAUST/END OF TAPE

An End of Tape (EOT) mark has been detected on a previous INPUT or OUTPUT operation - job abort.

END ITEM SMALLER THAN START ITEM

A ranged data element specified by the form ((a) ... (b)), has an ending item (b) smaller than the starting item (a) - only the starting entry is processed.

ILLEGAL FORMAT SPECIFICATION

Incompatible structure within the RTDSL's associated with the specified operation - job abort. Data is incompatible with format specification.

ILLEGAL INPUT CHARACTERS

Input characters for an I, E, and F format contains a character other than a "+", "-", "blank", ".", or a number 0 through 9 - flush the target data unit with zeros.

Number of octal characters (for a 0 format) exceed 21 - flush the target data unit with zeros.

LABEL INCORRECT

The first record of a tape file doesn't match the specified label during a CHECKID operation - job abort.

LOST FILE REFERENCE

A file data unit specification wasn't located as the first entry in the parameter list associated with a file operation - job abort.

NOT AT LOAD POINT

A magnetic tape file was not at load point during a CHECKID or DEFID operation - job abort.

NOT OPEN

A file operation was requested before the file was OPENed - job abort.

NOT OPENED AS INPUT OR SCRATCH

A file operation (CHECKID, INPUT, OUTPUT) was requested before the user-defined file was OPENed as an input or scratch file - job abort.

NO AVAILABLE TAPES

No available tape units remaining.

NOT OPENED AS OUTPUT OR SCRATCH

A file operation (DEFID, ENDFIL) was requested before the user-defined file was OPENed as an output or scratch file - job abort.

OUTSIDE TAPE PHYSICAL FILE

An End of Tape (EOT) mark was detected while positioning a file - job abort.

OUTSIDE FILE BOUNDARY

A file operation (INPUT, POS) was requested which either exceeds the number of records declared for that file or an EOT mark was detected - job abort.

OUTPUT NUMBER TOO LARGE

Number of converted characters exceeds the width specified by the format descriptor - flush the formatted target data unit with an ASCII character (*).

PROGRAM TERMINATED BY RUN TIME IO

Job abort message.

UNRECOVERABLE HARDWARE ERROR

The monitor has detected an Unrecoverable Hardware Error (UHE).

For a current unformatted INPUT or OUTPUT statement, processing will continue as normal with one exception. If a multiple read or a multiple write operation occurs as a result of a single INPUT or OUTPUT statement on a stream file, job abort will be invoked.

Job abort will be invoked for all current formatted INPUT and OUTPUT operations, file and record positioning (FIL, POS).

***** USER ERROR DISPLAY FLOATING POINT MAGNITUDE E+21 EXCEEDED**

Display table allows only E+21 characters to be displayed.

***** USER ERROR MAGNITUDE E-22 EXCEEDED--TRUNCATION OCCURRED**

Display table allows only E-22 characters to be displayed.

APPENDIX F

SUMMARY OF ASSEMBLER ERROR CODES

E Error Code

Expression errors result from illogical expressions such as a decimal digit within an octal number; element type inconsistent with arithmetic operators; expression improper in context such as a GO line used outside a MACRO or a DO count in excess of $2^{16}-1$.

D Error Code

Duplicate errors result from labels defined more than once with different values. A label used in an expression affecting an address counter is not defined prior to its use resulting in a different addressing sequence in the first and second assembly passes.

U Error Code

An Undefined error results from a reference made to a label which is nowhere defined in the program. A reference is made to a label which was not externalized properly by a call on a MACRO or by failure to suffix labels of MACRO entry points with an adequate number of asterisks.

I Error Code

An Instruction error results when the Assembler encounters:

- a) A MACRO or EQU directive which has no label.
- b) A SEGEND within a MACRO.
- c) More than one coded subfield in field zero of a MACRO reference line called via a MACRO name.
- d) A nested LIB directive or a LIB directive within a MACRO.
- e) A LIBS directive retrieved from a library.

R Error Code

A Relocation error results from an arithmetic or logical operation being performed on a relocatable value which destroyed its relocatability.

T Error Code

A Truncation error occurs when the final value of an expression does not fit in the destined bit field of an object word. Therefore, the Assembler truncated the left-most bits of the value in order to make it fit the field.

O Error Code

The Overflow error occurs when memory available for the Assembler tables is exhausted.

N Error Code

A Name error occurs when the Assembler encounters a name which contains more than eight characters.

L Error Code

A Level error results from an expression containing a parentheses nest which is more than five deep; more than 64 SETADR lines have been encountered in this assembly.

F Error Code

A Floating Point error occurs under any of the following three circumstances:

- a) The divisor in a requested floating point divide operation is zero.
- b) A floating point operation during evaluation of an expression yielded characteristic underflow. Characteristic underflow occurs whenever the characteristic is less than -32767.
- c) A floating point operation during evaluation of an expression resulted in characteristic overflow. Characteristic overflow occurs whenever the characteristic exceeds +32767.

W Error Code

A warning results when a label is used with a half-word instruction which is assigned to the lower half of a computer word.

Library Retrieval Error Messages and Operator Messages

See paragraphs 3.4.2.3 and 3.4.2.4 in Volume I.

Other Assembler Error Messages

See paragraphs 11.10, 11.10.1, 11.10.2, 11.10.3 and 11.10.4.

APPENDIX G

AN/UYK-7 CONDENSED REPERTOIRE

Processor Instructions

Function Codes	Mnemonic	Instruction	Description	Format	RPT	PI	CA
01 0	OR a,y,b,s	INCLUSIVE OR	$(Y) \oplus (A_a) \rightarrow A_a$	II	X		X
01 1	SC a,y,b,s	SELECTIVE CLEAR	$(Y)' \odot (A_a) \rightarrow A_a$	II	X		X
01 2	MS a,y,b,s	SELECTIVE SUBSTITUTE	$(Y)_n \rightarrow (A_{a+1})_n$ for $(A_a)_n = 1$	II	X		X
01 3	XOR a,y,b,s	EXCLUSIVE OR	$(Y) \oplus (A_a) \rightarrow A_a$	II	X		X
01 4	ALP a,y,b,s	ADD LOGICAL PRODUCT	$(A_{a+1}) + (Y) \odot (A_a) \rightarrow A_{a+1}$	II	X		X
01 5	LLP a,y,b,s	LOAD LOGICAL PRODUCT	$(Y) \odot (A_a) \rightarrow A_a$	II	X		X
01 6	NLP a,y,b,s	SUBTRACT LOGICAL PRODUCT	$(A_{a+1}) - (Y) \odot (A_a) \rightarrow A_{a+1}$	II	X		X
01 7	LLPN a,y,b,s	LOAD LOGICAL PRODUCT NEXT	$(Y) \odot (A_a) \rightarrow A_{a+1}$	II	X		X
02 0	CNT a,y,b,s	COUNT ONES	Count of ones in $\underline{Y} \rightarrow A_a$	II	X		X
02 2	XR y,b,s	EXECUTE REMOTE	$Y \rightarrow U$; (P) unchanged	II			
02 3	XRL y,b,s	EXECUTE REMOTE LOWER	$Y_L \rightarrow U$; (P) unchanged	II			
02 4	SLP a,y,b,s	STORE LOGICAL PRODUCT	$(A_a) \odot (A_{a+1}) \rightarrow Y$	II	X		X
02 5	SSUM a,y,b,s	STORE SUM	$(A_a) + (A_{a+1}) \rightarrow Y$ and A_{a+1} ; $(A_a)_i = (A_a)_f$	II	X		X
02 6	SDIF a,y,b,s	STORE DIFFERENCE	$(A_{a+1}) - (A_a) \rightarrow Y$ and A_{a+1} ; $(A_a)_i = (A_a)_f$	II	X		X
02 7	DS a,y,b,s	DOUBLE STORE A	$(A_{a+1}, A_a) \rightarrow Y + 1, Y$	II			
03 0	ROR a,y,b,s	REPLACE INCLUSIVE OR	$(Y) \oplus (A_a) \rightarrow Y$ and A_a	II	X		X
03 1	RSC a,y,b,s	REPLACE SELECTIVE CLEAR	$(Y)' \odot (A_a) \rightarrow Y$ and A_a	II	X		X
03 2	RMS a,y,b,s	REPLACE SELECTIVE SUBSTITUTE	For $(A_a)_n = 1, \underline{Y}_n \rightarrow Y$ and A_{a+1}	II	X		X
03 3	RXOR a,y,b,s	REPLACE EXCLUSIVE OR	$(Y) \oplus (A_a) \rightarrow Y$ and A_a	II	X		
03 4	RALP a,y,b,s	REPLACE A+ LOGICAL PRODUCT	$(A_{a+1}) + (Y) \odot (A_a) \rightarrow Y$ and A_{a+1}	II	X		X
03 5	RLP a,y,b,s	REPLACE LOGICAL PRODUCT	$(Y) \odot (A_a) \rightarrow Y$ and A_a	II	X		X
03 6	RNLP a,y,b,s	REPLACE A- LOGICAL PRODUCT	$(A_{a+1}) - (Y) \odot (A_a) \rightarrow Y$ and A_{a+1}	II	X		X
03 7	TSF y,b,s	TEST AND SET FLAG	Set CD; $1 \rightarrow Y_{31}$	II	X		

Processor Instructions (Continued)

Function Codes	Mnemonic	Instruction	Description	Format	RPT	PI	CA
05 0	DL a,y,b,s	DOUBLE LOAD A	$(Y + 1, Y) \leftarrow A_{a+1}, A_a$	II			
05 1	DA a,y,b,s	DOUBLE ADD A	$(Y + 1, Y) + (A_{a+1}, A_a) \leftarrow A_{a+1}, A_a$	II			
05 2	DAN a,y,b,s	DOUBLE SUBTRACT	$(A_{a+1}, A_a) - (Y + 1, Y) \leftarrow A_{a+1}, A_a$	II			
05 3	DC a,y,b,s	DOUBLE COMPARE	$(A_{a+1}, A_a) : (Y + 1, Y);$ Set CD	II			
05 4	LBMP a,y,b,s	LOAD BASE AND MEMORY PROTECTION	$Y \leftarrow S_a; Y+1 \leftarrow SPR_a; s \leftarrow SIR_a$ 19-17; $Y + (B_D)_b \leftarrow SIR_a$ 15-0	II			
06 0	FA a,y,b,s	FLOATING POINT ADD	$(A_{a+1}, A_a) + (Y + 1, Y) \leftarrow A_{a+1}, A_a$	II			
06 1	FAN a,y,b,s	FLOATING POINT SUBTRACT	$(A_{a+1}, A_a) - (Y + 1, Y) \leftarrow A_{a+1}, A_a$	II			
06 2	FM a,y,b,s	FLOATING POINT MULTIPLY	$(A_{a+1}, A_a) \times (Y + 1, Y) \leftarrow A_{a+1}, A_a$	II			
06 3	FD a,y,b,s	FLOATING POINT DIVIDE	$(A_{a+1}, A_a) \div (Y + 1, Y) \leftarrow A_{a+1}, A_a$	II			
06 4	FAR a,y,b,s	FLOATING POINT ADD WITH ROUND	$(A_{a+1}, A_a) + (Y + 1, Y) \leftarrow A_{a+1}, A_a$; round result	II			
06 5	FANR a,y,b,s	FLOATING POINT SUBTRACT WITH ROUND	$(A_{a+1}, A_a) - (Y + 1, Y) \leftarrow A_{a+1}, A_a$; round result	II			
06 6	FMR a,y,b,s	FLOATING POINT MULTIPLY WITH ROUND	$(A_{a+1}, A_a) \times (Y + 1, Y) \leftarrow A_{a+1}, A_a$; round result	II			
06 7	FDR a,y,b,s	FLOATING POINT DIVIDE WITH ROUND	$(A_{a+1}, A_a) \div (Y + 1, Y) \leftarrow A_{a+1}, A_a$; round result	II			
07 0	XS sy,b	ENTER EXECUTIVE STATE	Interrupt to Executive Entrance Address	II			
07 0	IPI sy,b (a=1)	INTERPROCESSOR INTERRUPT	Allow selective interrupts to other processors	II			X
07 1	AEI a,sy,b	ALLOW ENABLE INTERRUPT	Enable Interrupt Request for IOC _a Channel Specified	II			X
07 2	PEI a,sy,b	PREVENT ENABLE INTERRUPT	Disable Interrupt Request for IOC _a Channel Specified	II			X
07 3	LIM a,sy,b	LOAD, ENABLE IOC MONITOR CLOCK	$Y \leftarrow IOC_a$ Monitor Clock	II			X
07 4	IO a,y,b,s	INITIATE INPUT/OUTPUT	$ADDR, Y=y + (B_b) + (S_s) \leftarrow IOC$	II			X
07 5	IR	INTERRUPT RETURN	Return to Processor State designated by DSW	II			X
07 6	RP a,sy,b	REPEAT	Repeat NI (B7) times or until test condition is satisfied	II			
10	LA a,y,k,b,s	LOAD A	$Y \leftarrow A_a$	I	X		X

Processor Instructions (Continued)

Function Codes	Mnemonic	Instruction	Description	For- mat	RPT	PI	CA
11	LXB	a,y,k,b,s	LOAD A AND INDEX B	$\underline{Y} \rightarrow A_a; (B_b) + 1 \rightarrow B_b$	I		X
12	LDIF	a,y,k,b,s	LOAD DIFFERENCE (Y-A)	$\underline{Y} - (A_a) \rightarrow A_{a+1}$	I	X	X
13	ANA	a,y,k,b,s	ADD NEGATIVE A	$(A_a) - \underline{Y} \rightarrow A_a$	I	X	X
14	AA	a,y,k,b,s	ADD A	$(A_a) + \underline{Y} \rightarrow A_a$	I	X	X
15	LSUM	a,y,k,b,s	LOAD SUM (Y+A)	$\underline{Y} + (A_a) \rightarrow A_{a+1}$	I	X	X
16	LNA	a,y,k,b,s	LOAD NEGATIVE	$\underline{Y}' \rightarrow A_a$	I	X	X
17	LM	a,y,k,b,s	LOAD MAGNITUDE	$ \underline{Y} \rightarrow A_a$	I	X	X
20	LB	a,y,k,b,s	LOAD B	$\underline{Y} \rightarrow B_a$	I	X	X
21	AB	a,y,k,b,s	ADD B	$(B_a) + \underline{Y} \rightarrow B_a$	I	X	X
22	ANB	a,y,k,b,s	SUBTRACT B	$(B_a) - \underline{Y} \rightarrow B_a$	I	X	X
23	SB	a,y,k,b,s	STORE B	$(B_a) \rightarrow Y$	I	X	X
24	SA	a,y,k,b,s	STORE A	$(A_a) \rightarrow Y$	I	X	X
25	SXB	a,y,k,b,s	STORE A AND INDEX B	$(A_a) \rightarrow Y; (B_b) + 1 \rightarrow B_b$	I		X
26	SNA	a,y,k,b,s	STORE NEGATIVE	$(A_a)' \rightarrow Y$	I	X	X
27	SM	a,y,k,b,s	STORE MAGNITUDE	$ A_a \rightarrow Y$	I	X	X
32	BZ	ak,y,b,s	CLEAR BIT	$0 \rightarrow Y_n; (n = ak)$	I	X	
33	BS	ak,y,b,s	SET BIT	$1 \rightarrow Y_n; (n = ak)$	I	X	
34	RA	a,y,k,b,s	REPLACE ADD	$\underline{Y} + (A_a) \rightarrow Y$ and A_{a+1}	I	X	X
35	RI	a,y,k,b,s	REPLACE INCREMENT	$\underline{Y} + 1 \rightarrow Y$ and A_a	I	X	X
36	RAN	a,y,k,b,s	REPLACE SUBTRACT	$\underline{Y} - (A_a) \rightarrow Y$ and A_{a+1}	I	X	X
37	RD	a,y,k,b,s	REPLACE DECREMENT	$\underline{Y} - 1 \rightarrow Y$ and A_a	I	X	X
40	M	a,y,k,b,s	MULTIPLY A	$\underline{Y} \times (A_a) \rightarrow A_{a+1}, A_a$	I	X	X
41	D	a,y,k,b,s	DIVIDE A	$(A_{a+1}, A_a) \div \underline{Y} \rightarrow A_a;$ Remainder $\rightarrow A_{a+1}$	I	X	X
42	BC	ak,y,b,s	COMPARE BIT TO ZERO	$(Y)_n = 0$; Set CD to equal if $(Y)_n = 0$ and non- equal if $(Y)_n \neq 0, (n=ak)$	I	X	
43	CXI	a,y,k,b,s	COMPARE INDEX INCREMENT	$(B_a) : \underline{Y}$ if $(B_a) \geq \underline{Y},$ $0 \rightarrow B,$ set CD to OL; if $(B_a) < \underline{Y}, (B_a) + 1 \rightarrow B_a,$ set CD to WL	I	X	X
44	C	a,y,k,b,s	COMPARE	$(A_a) : \underline{Y}$; Set CD	I	X	X
45	CL	a,y,k,b,s	COMPARE LIMITS	$(A_a), (A_{a+1}) : \underline{Y}$; Set CD	I	X	X
46	CM	a,y,k,b,s	COMPARE MASKED	$(A_{a+1}) : (A_a) \odot \underline{Y}$; Set CD	I	X	X
47	CG	a,y,k,b,s	COMPARE GATED	$ \underline{Y} - (A_a) : (A_{a+1})$; Set CD	I	X	X
50	0	JEP	a,y,k,b,s	JUMP EVEN PARITY	III		
				If $(A_{a+1}) \odot (A_a)$ even, jump to \underline{Y} ; else NI			

Processor Instructions (Continued)

Function Codes		Mnemonic	Instruction	Description	For- mat	RPT	PI	CA
50	1	JOP	a,y,k,b,s	JUMP ODD PARITY	If $(A_{a+1}) \odot (A_a)$ odd, jump to <u>Y</u> ; else NI			III
50	2	DJZ	a,y,k,b,s	DOUBLE JUMP A ZERO	If $(A_{a+1}, A_a) = 0$ jump to <u>Y</u> ; else NI			III
50	3	DJNZ	a,y,k,b,s	DOUBLE JUMP A NOT ZERO	If $(A_{a+1}, A_a) \neq 0$, jump to <u>Y</u> ; else NI			III
51	0	JP	a,y,k,b,s	JUMP A POSITIVE	If $(A_a) \geq 0$, jump to <u>Y</u> ; else NI			III
51	1	JN	a,y,k,b,s	JUMP A NEGATIVE	If $(A_a) < 0$, jump to <u>Y</u> ; else NI			III
51	2	JZ	a,y,k,b,s	JUMP A ZERO	If $(A_a) = 0$, jump to <u>Y</u> ; else NI			III
51	3	JNZ	a,y,k,b,s	JUMP A NOT ZERO	If $(A_a) \neq 0$, jump to <u>Y</u> ; else NI			III
52	0	LBJ	a,y,k,b,s	LOAD B AND JUMP	$(P) \rightarrow B_a$, jump to <u>Y</u> $(Y \rightarrow P)$			III
52	1	JBNZ	a,y,k,b,s	JUMP B NOT ZERO	If $(B_a) > 0$, $(B_a) - 1$ $\rightarrow B_a$, jump to <u>Y</u> ; else NI			III
52	2	JS	sy,k,b	JUMP SY + b	$Y = (B_b) + sy$			III
52	3	JL	y,k,b,s	JUMP LOWER	Jump to <u>Y</u> Lower			III
53	0	JNF	y,k,b,s	JUMP NO OVERFLOW	If OD not set, jump to <u>Y</u> ; else NI			III
53	01	JNE	y,k,b,s	JUMP NOT EQUAL	If CD set \neq , jump to <u>Y</u> ; else NI			III
53	10	JOF	y,k,b,s	JUMP OVERFLOW	If OD set, jump to <u>Y</u> ; else NI			III
53	11	JE	y,k,b,s	JUMP EQUAL	If CD set $=$, jump to <u>Y</u> ; else NI			III
53	21	JG	y,k,b,s	JUMP GREATER THAN	If CD set $>$, jump to <u>Y</u> ; else NI			III
53	31	JGE	y,k,b,s	JUMP GREATER THAN OR EQUAL	If CD set \geq , jump to <u>Y</u> ; else NI			III
53	41	JLT	y,k,b,s	JUMP LESS THAN	If CD set $<$, jump to <u>Y</u> ; else NI			III
53	51	JLE	y,k,b,s	JUMP LESS THAN OR EQUAL	If CD set \leq , jump to <u>Y</u> ; else NI			III
53	61	JNW	y,k,b,s	JUMP NOT WITHIN LIMITS	If CD set OL, jump to <u>Y</u> ; else NI			III
53	71	JW	y,k,b,s	JUMP WITHIN LIMITS	If CD set WL, jump to <u>Y</u> ; else NI			III
53	2	RJ	y,k,b,s	RETURN JUMP	$(P) \rightarrow Y$, $Y + 1 \rightarrow P$			III

Processor Instructions (Continued)

Function Codes	Mnemonic	Instruction	Description	For- mat	RPT	PI	CA
53 2	RJC a.y,k,b,s	RETURN JUMP CONDITIONAL SETTING	If jump key _a set, (P) $\rightarrow Y, Y + 1 \rightarrow P$; else NI	III			①
53 2	RJSC a.y,k,b,s	RETURN JUMP, STOP CONDITIONAL SETTING	(P) $\rightarrow Y, Y + 1 \rightarrow P$; if stop key _a set, stop; else NI	III			X①
53 3	J y,k,b,s	JUMP	$Y \rightarrow P$	III			
53 3	JC a.y,k,b,s	JUMP CONDITIONAL SETTING	If jump _a set, jump to Y ; else NI	III			①
53 3	JSC a.y,k,b,s	JUMP, STOP CONDITIONAL SETTING	$Y \rightarrow P$; if Stop Key _a set, stop; else NI	III			X①
54	LCT ak,y,b,s	LOAD TASK CMR	$Y \rightarrow \text{CMR}$	I	X		X②
55	LCI ak,y,b,s	LOAD INTERRUPT CMR	$Y \rightarrow \text{Interrupt CMR}$	I	X		X
56	SCT ak,y,b,s	STORE TASK CMR	(CMR) $\rightarrow Y$	I	X		X②
57	SCI ak,y,b,s	STORE INTERRUPT CMR	(Interrupt CMR) $\rightarrow Y$	I	X		X
60	HSCT af ₄ ,b	STORE TASK CMR IN A	(CMR) $\rightarrow A_b$	IV-A			X②
60	HSCI af ₄ ,b	STORE INTERRUPT CMR IN A	(Interrupt CMR) $\rightarrow A_b$	IV-A			X
61	HLCT af ₄ ,b	LOAD TASK CMR WITH A	(A _b) $\rightarrow \text{CMR}$	IV-A			X②
61	HLCI af ₄ ,b	LOAD INTERRUPT CMR WITH A	(A _b) $\rightarrow \text{Interrupt CMR}$	IV-A			X
62	HLC ^③ a,m HLC a,b,1 HLC a,b,2	SHIFT LEFT CIRCULARLY	Shift (A _a) left, end around	IV-B			
63	HDLC ^③ a,m HDLC a,b,1 HDLC a,b,2	DOUBLE SHIFT LEFT CIRCULARLY	Shift (A _{a+1} , A _a) left, end around	IV-B			
64	HRZ ^③ a,m HRZ a,b,1 HRZ a,b,2	SHIFT RIGHT FILL ZEROS	Shift (A _a) right, end off, zero fill on left	IV-B			
65	HDRZ ^③ a,m HDRZ a,b,1 HDRZ a,b,2	DOUBLE SHIFT RIGHT FILL ZEROS	Shift (A _{a+1} , A _a) right, end off, zero fill on left	IV-B			
66	HRS ^③ a,m HRS a,b,1 HRS a,b,2	SHIFT RIGHT FILL SIGN	Shift (A _a) right end off, sign fill on left	IV-B			
67	HDRS ^③ a,m HDRS a,b,1 HDRS a,b,2	DOUBLE SHIFT RIGHT FILL SIGN	Shift (A _{a+1} , A _a) right, end off, sign fill on left	IV-B			
70 0	HSF a,b	SCALE FACTOR	Normalize (A _a); shift count $\rightarrow A_b$	IV-A			
70 1	HDSF a,b	DOUBLE SCALE FACTOR	Normalize (A _{a+1} , A _a); shift count $\rightarrow A_b$	IV-A			
70 2	HCP a	COMPLEMENT A	(A _a)' $\rightarrow A_a$	IV-A			
70 3	HDCP a	DOUBLE COMPLEMENT A	(A _{a+1} , A _a)' $\rightarrow A_{a-1}, A_a$	IV-A			
71 0	HOR a,b	INCLUSIVE OR A	(A _a) \oplus (A _b) $\rightarrow A_a$	IV-A			

① Privileged instruction if $4 \leq a \leq 7$.② Privileged instruction if $60_8 \leq af_4 \leq 77_8$,
 $20_8 \leq af_4 \leq 27_8$

③ Shift instructions have three possible formats

f a,m (m is shift count)

f a,b,1 (shift count in B_b)f a,b,2 (shift count in A_b)

Processor Instructions (Continued)

Function Codes		Mnemonic	Instruction	Description	Format	RPT	PI	CA
71	1	HA	a,b	ADD (SUM)	$(A_a) + (A_b) \rightarrow A_a$			IV-A
71	2	HAN	a,b	SUBTRACT (DIFFERENCE)	$(A_a) - (A_b) \rightarrow A_a$			IV-A
71	3	H XOR	a,b	EXCLUSIVE OR A	$(A_a) \oplus (A_b) \rightarrow A_a$			IV-A
71	5	H AND	a,b	AND A	$(A_a) \odot (A_b) \rightarrow A_a$			IV-A
74	0	HM	a,b	MULTIPLY REGISTER	$(A_a) \times (A_b) \rightarrow A_{a+1}, A_a$			IV-A
74	1	HD	a,b	DIVIDE REGISTER	$(A_{a+1}, A_a) \div (A_b) \rightarrow A_a$ Remainder $\rightarrow A_{a+1}$			IV-A
74	2	HRT	a,b	SQUARE ROOT	$\sqrt{(A_{a+1}, A_a)} \rightarrow A_b$ Residue $\rightarrow A_{b+1}$			IV-A
74	3	HLB	a,b	LOAD B_a WITH B_b	$(B_b) \rightarrow B_a$			IV-A
74	4	HC	a,b	COMPARE, REGISTER	$(A_a):(A_b)$; Set CD			IV-A
74	5	HCL	a,b	COMPARE LIMITS, REGISTER	$(A_{a+1}, A_a):(A_b)$; Set CD			IV-A
74	6	HCM	a,b	COMPARE MASKED, REGISTER	$(A_{a+1}) \odot (A_a):(A_b)$; Set CD			IV-A
74	7	HCB	a,b	COMPARE B_a WITH B_b	$(B_a):(B_b)$; Set CD			IV-A
77	0	HSIM	a,b	STORE I/O MONITOR CLOCK	$(IOC_a \text{ Monitor Clock}) \rightarrow A_b$			IV-A X
77	1	HSTC	a,b	STORE REAL-TIME CLOCK	$(IOC_a \text{ RTC}) \rightarrow A_b$			IV-A
77	4	HPI		PREVENT CLASS III INTERRUPTS	Lock out Class III Interrupts			IV-A X
77	5	HAI		ALLOW CLASS III INTERRUPTS	Allow Class III Interrupts			IV-A X
77	6	HALT		STOP PROCESSOR	Stop Processor Operations			IV-A X
77	6	HWFI	i=1	WAIT FOR INTERRUPT	Stop referencing memory until interrupt occurs			IV-A X

① Privileged instruction if $4 \leq a \leq 7$.

② Privileged instruction if $60_8 \leq af_4 \leq 77_8$, $20_8 \leq af_4 \leq 27_8$

③ Shift instructions have three possible formats:

- f a,m (m is shift count)
- f a,b,1 (shift count in B_b)
- f a,b,2 (shift count in A_b)

Extension Instructions

Function	Memonic	Instruction	Description	Format	RPT	PI	CA
10	ZA	a	CLEAR A	0 - A ₀	I		X
20	ZB	a	CLEAR B	0 - B ₀	I		X
20	NOOP		NO OPERATION (Full Word)	0 - B ₀	I		X
23	SZ	y,k,b,s	STORE ZEROS	0 - <u>Y</u>	I		X
74	3	HNO	NO OPERATION (Half Word)	(B ₀) - B ₀	IV-A		
c = 2	IW	y,b,s	INDIRECT WORD				
c = 1	IWC	y,w,p,b,s	INDIRECT WORD CHARACTER				
c = 3	IWCI	y,w,p,b,s	INDIRECT WORD CHARACTER INCREMENT				
c = 0, bit 29=0	IWS	sy,b	INDIRECT WORD, SPECIAL BASE				
c = 0, bit 29=1	IWB	sy,b	INDIRECT WORD, SPECIAL INDEX				
	MP	r,i,or,ow,ia,ir					
	HK	e	HALF-WORD CONSTANT				
	BCW	y,l	BUFFER CONTROL WORD				
	BCWE	y,l,k	BUFFER CONTROL WORD ESI				

Input/Output Controller Instructions

Function	Mnemonic	Instruction	Description
10	IB j.y.k.c.m	INITIATE INPUT BUFFER ON Cj	Initiate input buffer on Cj
11	OB j.y.k.c.m	INITIATE OUTPUT BUFFER ON Cj	Initiate output buffer on Cj
12	FB j.y.k.c.m	INITIATE EF BUFFER ON Cj	Initiate EF buffer on Cj
13	XB j.y.k.c.m	INITIATE EI BUFFER ON Cj	Initiate EI buffer on Cj
14 (k = 0)	TIB j.c.m	TERMINATE INPUT BUFFER ON Cj	Terminate output buffer on Cj
14 (k = 1)	TOB j.c.m	TERMINATE OUTPUT BUFFER ON Cj	Terminate output buffer on Cj
14 (k = 2)	TFB j.c.m	TERMINATE EF BUFFER ON Cj	Terminate EF buffer on Cj
14 (k = 3)	TXB j.c.m	TERMINATE EI BUFFER ON Cj	Terminate EI buffer on Cj
15 (k = 0)	IMIR j.c	SET INPUT MONITOR INTERRUPT REQUEST ON Cj	Set input monitor interrupt request on Cj
15 (k = 1)	OMIR j.c	SET OUTPUT MONITOR INTERRUPT REQUEST ON Cj	Set output monitor interrupt request on Cj
15 (k = 2)	FMIR j.c	SET EF MONITOR INTERRUPT REQUEST ON Cj	Set EF monitor interrupt request on Cj
15 (k = 3)	XMIR j.c	SET EI MONITOR INTERRUPT REQUEST ON Cj	Set ESI monitor interrupt request on Cj
16 (k = 0)	AIC j.y.c	ACTIVATE INPUT CHAIN ON Cj	Activate input chain on Cj
16 (k = 1)	AOC j.y.c	ACTIVATE OUTPUT CHAIN ON Cj	Activate output chain on Cj
16 (k = 2)	AFC j.y.c	ACTIVATE EF CHAIN ON Cj	Activate EF chain on Cj
16 (k = 3)	AXC j.y.c	ACTIVATE EI CHAIN ON Cj	Activate EI chain on Cj
17 (m = 0)	TBZ kj.y	TEST BIT CLEARED	If $(Y)_{kj} = 0$, skip; else NI
17 (m = 1)	TBS kj.y	TEST BIT SET	If $(Y)_{kj} \neq 0$, skip; else NI
20	JIO y.c	JUMP (INPUT/OUTPUT)	$Y \rightarrow$ Chain Pointer
22	LICM kj.y.c	LOAD IOC CONTROL MEMORY	$(Y) \rightarrow$ IOC Control Memory address kj
23	ILTC y.c	LOAD REAL-TIME CLOCK	$(Y) \rightarrow$ RTC
24	SICM kj.y.c	STORE IOC CONTROL MEMORY	$(\text{IOC Control Memory})_{kj} \rightarrow Y$
25	IBS kj.y.c	SET BIT	$1 \rightarrow Y_{kj}$
26	IBZ kj.y.c	CLEAR BIT	$0 \rightarrow Y_{kj}$
27	ITSF y.c	TEST AND SET FLAG	$1 \rightarrow Y_{31}$; if $(Y)_{31}$ was originally cleared, skip; else NI

APPENDIX II

CMS-2 SYSTEM TAPE DUPLICATION

Additional copies of the CMS-2 system tape may be made by the following:

Format

```
$JOB  
$SYSMaker  
ENDSYSBD      input name, output name  
$ENDJOB
```

Explanation

input name One to eight character name of the system tape to be copied. To copy the system tape that is currently running, code the name CMS2SYST or leave this field blank.

output name One to eight character name of the tape on which the copy is to be written. If this field is not specified, the copy will be written on a tape named NEWCØPTP.

Example

```
$JOB  
$SYSMaker  
ENDSYSBD IN,OUT  
$ENDJOB
```

This would duplicate the system tape IN on to tape OUT

APPENDIX I SYSTEM MODIFICATION

I.1 INTRODUCTION

There are two basic methods of modifying the data on a system tape using CMS-2. The first method uses the system tape generation control cards (described in paragraph 1.2) to completely reconstruct a system, including recompiling some of the components and adding new components. The second method uses system debugging aid cards (SPATCH described in paragraph I.2.8) to change individual instructions in specified component records each time the system loader loads the record from tape into memory.

I.1.1 System Tape Organization

Figure 1-1 illustrates the organization of a system tape generated by CMS-2. The first record on the tape contains the resident portion of the Monitor that the bootstrap routine loads into memory during system initiation.

This first record also contains the resident system tape directory which identifies the physical position (by record number) of each segment directory. Each of these segment directories identifies which records make up the segment and where on the system tape (again by record number) each required record is located. The record data in memory is also called the compound address section with assigned base registers.

The second record contains the segment directory for segment 2 in component 1. The following records contain the data for segment 2. The reader should note that the second tape record is the directory for segment 2 of component 1 because this record is an exception to a basic organization assumption that the first segment directory on tape for a component is for segment 1 of the component. The first record on the tape is not assigned a segment number because it never leaves memory; it is considered a part of component 1 (the Monitor).

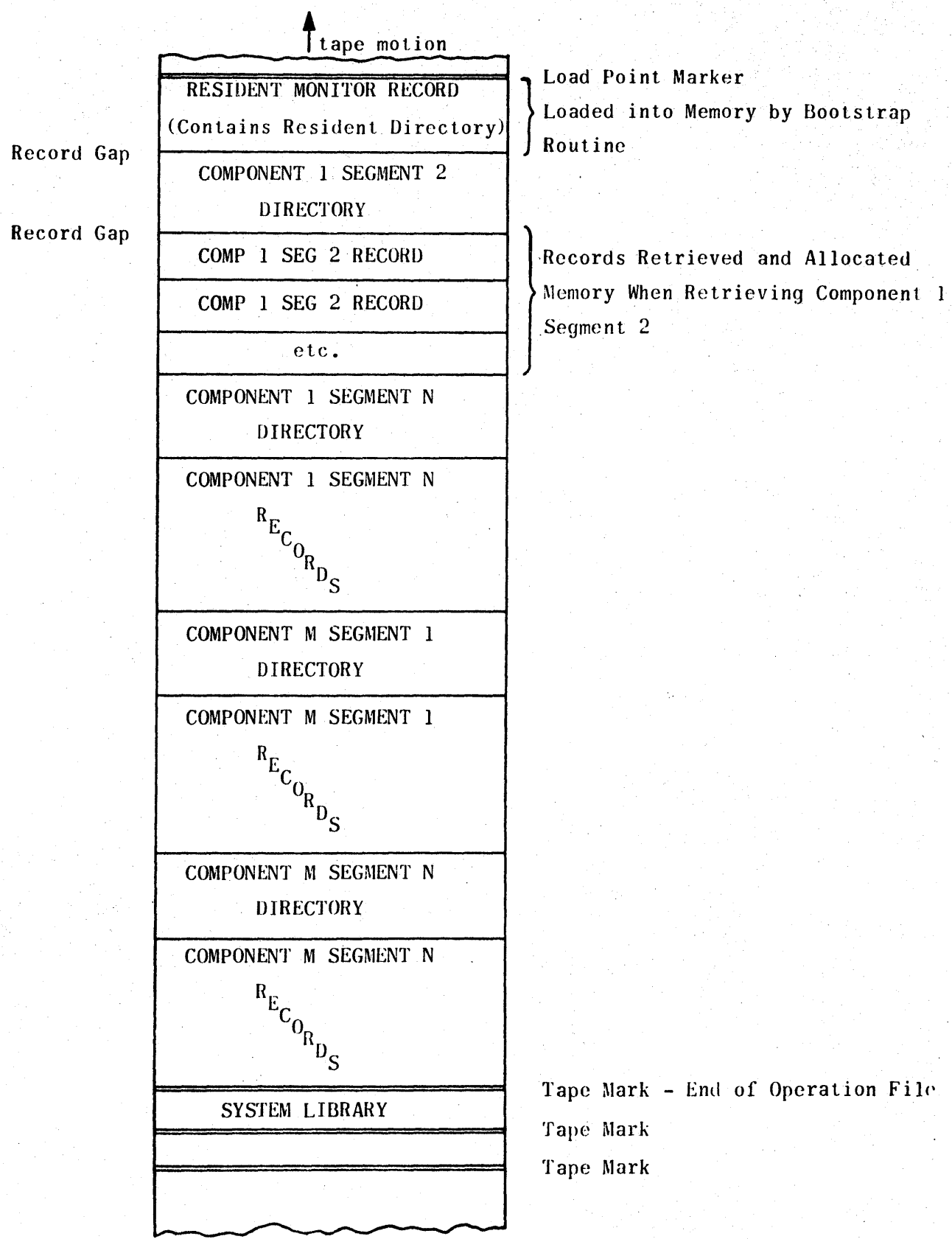


Figure I-1. System Tape Organization

Except for component 1, the first segment in a component must be segment 1. There is no rule governing the physical arrangement of the other segments in a component. They may be arranged in any order (normally to reduce tape movement during operation). Likewise, there is no rule governing the physical order of the components on the tape (except that the first component must be component 1).

A tape mark separates the operational file (containing the records for the segments in the system components) from the system library containing assembler procedures, compiler math routines, and so forth.

Two tape marks appear at the end of the system library to indicate the end of recorded data.

I.1.2 System Directories

There are basically two system directories; the resident directory and the segment directory. The resident directory is loaded into memory during the bootstrap load. This directory contains a list of the components on the tape, and the location of each segment directory for each component. There is a segment directory on the tape for each segment in the system. The segment directory lists each record that must be in memory before the segment can function properly. This list includes the physical location of the required records on tape.

I.1.2.1 Resident Directory

The resident directory has two basic parts: first, an introduction and list of components, and second, a list in physical component/segment order identifying the location of corresponding segment directories on the system tape.

The format of the resident directory is:

31	16	0
Number of Components		
Tape Version		
System Tape Construction Date		
Counter		
Component Number	Location of Seg List	
Component Number	Location of Seg List	

Introduction and
Component List

Component Number	Location of Seg List	
Number of Segments		
Segment Number	Directory Position	
Directory Length		
Directory Checksum		
Segment Number	Directory Position	
Directory Length		
Directory Checksum		

List of Segments
in One Component

Segment Number	Directory Position	
Directory Length		
Directory Checksum		
Number of Segments		
Segment Number	Directory Position	
Directory Length		
Directory Checksum		
etc.		

List of Segments
in Next Component

Unless otherwise specified, all numbers are right-justified and all character strings are left-justified.

Explanation

Number of Components	The number of system components currently on the tape. This value indicates the length of the component list.
Tape Version	The 16-character system version or identifier specified on the first control card when the tape was generated.
System Tape Construction Date	The eight-character date in the format MM/DD/YY supplied when system tape generation routine generated the system. MM = number of the month; DD = date; YY = last two digits of the year.
Counter	Set to zero on BUILD and REPLACE functions. It is incremented by one on MOVE, DELETE, and PATCH functions. If multiple functions are specified, the first designated function determines change in counter value.
Component Number	An assigned number that is associated with process an independent processor. (Loader, Librarian, Sysmaker, etc.)
Location of Segment List	This is a pointer to the segment list of the component. The segment list is for the component that is in the same word as the pointer.
Number of Segments	The number of segments that make up the corresponding component. This value indicates the length of the corresponding segment list (3 words required for each segment and 1 word for the number of segments).
Segment Number	As assigned number corresponding to a phase or pass of a given component. It is the collection of records of a component that are in core and executing at one time.

Directory Position The tape record number of the segment directory corresponding to the segment number in the same word.

Directory Length The length of that segment directory.

Directory Checksum The checksum (formed by executing full length fixed point adds disregarding overflow) of the segment directory data.

NOTE

At the end of each job, the CMS-2 Monitor prints the Tape Version, System Tape Construction Date, and Counter on the SYSTEM MEDIUM line.

I.1.2.2 Segment Directory

Each segment directory contains one word indicating the number of records that make up the segment plus a seven-word packet for each record. The packet contains the data used by the system loader to place the corresponding record in memory upon command.

The format of the segment directory is:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Number of Records in Segment																															
Component Number																Record Type								No. Linked Rec							
Record Name																															
Record Position																Initl Base Reg								Consec Base Reg							
Memory Protection Code																Number of Words															
Record Checksum																															
Not Used																								Cont flg							
Component Number																Record Type								No Linked Rec							
Record Name																															
Record Position																Initl Base Reg								Consec Base Reg							
Memory Protection Code																Number of Words															
Record Checksum																															
Not Used																								Cont Flg							

First Seven-Word Record Packet

Second Seven-Word Record Packet

etc.

Unless otherwise specified, all numbers are right-justified and all character strings are left-justified.

Explanation

Number of Records in Segment	The number of records that make up the corresponding segment. This value indicates the number of record packets in the directory.
Component Number	The assigned number of the component that the record is a part of.
Record Type	One of the following codes that identifies the type of record: 1 - fixed length data record. 2 - fixed length instruction record. 3 - dynamic variable length data record. 4 - dynamic variable length instruction record. The Monitor permits dynamic record length adjustment requests (described in Systems Programmer's Manual paragraph 3.4.2) to reference only type 3 or 4 records.
No. Linked Records	The number of tape blocks that are linked together to form a compound section or record.
Record Name	The up to eight-character name assigned to the record. The system routines reference the record using the assigned name. The name corresponds directly to the compound section name used on the object-code loader directives.

Record Position

The relative position of associated record on tape (the first record after the load point is record 0) counting the segment directories as records.

Initl Base Reg

The number of the base register that the system load routine is to assign to the first 8K positions of the record.

Consec Base Reg

Indicates whether or not the system load routine is to assign consecutive base registers for each 8K of memory as follows:

- 1 - No; assign only one base register.
- 2 - Yes

Memory Protect Code

An octal code that specifies what type of memory protection the System Load Routine is to set up for the record. It is a five-bit field. The operation is allowed if the bit is set.

BIT NUMBER	5	4	3	2	1
TYPE	I	OR	OW	IA	IR

IR = 1 - Allows the use of interrupt index and base registers in indirect addressing.

IA = 1 - Allows indirect addressing.

OW = 1 - Allows operand writing.

OR = 1 - Allows operand reading.

I = 1 - Allows instruction execution.

Number of Words

The number of memory words required to hold the record data. For dynamic records (variable lengths - types 3 and 4) this value indicates the initial length of the record. If this value is 0, there is no data on tape; however, the system load routine places the record name in the active record list and assigns memory space as the routine receives requests to expand the record length (see **Systems Programmer's Manual**, paragraph 3.4.2).

Record Checksum

The checksum (formed by executing full-length, fixed-point adds disregarding overflow) of the specified record data.

Cont Flg

Two bits (bits 0 and 1) indicating whether or not the record is the control record for the associated segment as follows:

- 1 - indicates a control record.
- 2 - indicates non-control record.

When activating a segment, the Monitor passes control to the base address of the indicated control record. If more than one control record exists, control is given to the first one encountered in the segment directory.

I.2 SYSTEM TAPE GENERATION

System tape generation uses two routines; the system tape generation **SYSMaker** routine (to process special cards required to prepare the resident directory and segment directories) and the object-code loader (to bind the routines). The **SYSMaker** can build a system tape using only the special cards and loader

input. However, in normal operation, the SYSMAKER builds a new system tape by starting with data on an existing system tape, then changing and adding data, as commanded, to generate the new tape.

The Monitor loads and transfers control to the SYSMAKER after processing a \$SYSMAKER card (same format as the other component call cards; for example, the \$SYSMAKER starts in column 1 with the rest of the card blank or having comments).

The SYSMAKER performs the following five main functions:

1. Build or insert function which adds a new segment or a new component, and new directory information.
2. Replace function which adds a new version and deletes the old version of one segment at a time.
3. Move function which changes the relative positions of a component or segment.
4. Delete function which eliminates a component or segment.
5. Patch function which inserts system patches.

The routine scans the special control cards for one of the above functions. The first four functions must be declared in such an order that the current component upon which the operation is being performed has not been already copied by a previous function. The patch function must be a separate \$SYSMAKER call.

I.2.1 Build or Insert Function Cards

The build or insert function cards command the SYSMAKER to add a new component or a new segment to a component. The routine retrieves the resident directory and the corresponding segment directory from the input system tape (if applicable). The definitions on the build cards are used to construct (or reconstruct) the resident directory and segment directories. The routine copies the data from the input tape (up to the position when the defined segment or component is supposed to go) onto the scratch tape.

The SYSMAKER then calls the Loader to bind all address sections. The lengths and other information passed from the Loader are placed into the segment directory tables. These bound sections (records) are placed on the scratch tape with the segment directory record first and the remaining records in the relative position declared with as many control records as are declared per segment. At this point, more segment builds, replaces, moves or deletions may be initiated. Upon the reading an ENDSYSBD card, the scratch tape is file marked, and a new system tape is built containing the new resident directory information. The system library will be copied from the input system tape onto the output tape. For the case of an initial build, a null file with two tape marks will follow the operational library.

The following four cards are associated with the build or insert function:

1. Initiate build.
2. Segment definition.
3. Record definition.
4. Initiate Loader.

I.2.1.1 Initiate Build Card

The initiate build card commands the SYSMAKER to activate the build or insert function. One of these cards must be present for each component the programmer wishes to add to the system. The initiate build card defines the component the programmer wishes either to add to the system or to modify. Subsequent cards define individual segments in the component and records in the segments.

Format

BUILD output tape, comp no., asg name, comp position, input tape,
 mon loc., resdir offset, asg 2 name

Explanation

Output Tape

The up to eight-character name of the output system tape the SYSMAKER is to use in the console messages. If this field is blank, the routine assumes the name NEWSYSTP.

Comp No.

The number assigned to the component when it appears on the output tape. If this number is associated with a component on the input tape, the routine assumes that specified segments will be added to the component.

Asg Name

A name or version up to four characters which the routine assigns to the new tape. This name appears in the second word of the resident directory.

Comp Position

The number of the component on the input tape that is to immediately precede the component being processed. If this field is blank, the routine will place the component being processed either at the end of the output tape if the component is not on the input tape, or at the same relative position on the output as the component appears on the input tape.

Input Tape

The up to eight-character name of the input tape the SYSMAKER is to use in the console messages. If this field is blank, the routine will use the current system tape. A zero in this field specifies no input tape.

Mon Loc

The absolute octal address of the start of the Monitor bootstrap load area. The current Monitor is always loaded into the upper area of memory. This value then specifies the bottom of the resident Monitor load area and the top address available to the non-resident routines. If this field is blank, the routine assumes address 0.

Resdir Offset

The offset from the beginning of the boot block (first jump cell) to the first address of the resident directory. If this field is blank, the input system tape offset will be assumed.

Asg 2 Name

A name or version up to eight characters which the routine assigns to the new tape. This name appears in the fourth and fifth words of the resident directory. The two assigned names are output upon completion of each job in the accounting summary.

NOTE

The SYSMaker uses the output tape, assign name, input tape, resdir offset, and Monitor octal load data on the first card processed after the \$SYSMaker card. The routine ignores these fields on subsequent cards; therefore, these fields may be left blank.

Example

BUILD | SYS036,8,SWAN,4,SYS03A,111000

On tape SYS036, prepare to insert a component with the number 8. Use tape SYS034 as the input system to be modified. Insert component 8 immediately behind component 4. The resident monitor starts at address 111,000₈.

I.2.1.2 Segment Definition Card

The segment definition card indicates the start of a segment record list for a specified segment within the component defined by an initiate build card.

Format

SEG segment number, segment position

Explanation

Segment Number

The assigned segment number the segment being processed will have when it appears on the output tape.

Segment Position

The number of the segment within the component (specified by the initial build card) that is to immediately precede the segment being processed on tape. If this field is blank, the SYSMAKER places the segment at the end of the component or, if on the input tape, in the same relative position. This field may not be used when "comp position" is used on the Build card.

I.2.1.3 Record Definition Card

There are two record definition cards. These cards contain the information required to construct the record packet in the segment directory. The two cards have the same format except for the card identifier. REC indicates a record definition card and CREC indicates a control record definition card.

Format

REC

or record name, type, link base reg, mem protect code, length

CREC

Explanation

Record Name

The up to eight-character record name assigned to the record being processed. The specified data appears as the record name in the corresponding record packet in the segment directory.

Type

A code that indicates the type of record as follows:

- 1 - fixed length data record.
- 2 - fixed length instruction record.
- 3 - variable length dynamic data record.
- 4 - variable length dynamic instruction record.

Link Base Regs

A code that indicates whether or not the system Loader is to assign consecutive base registers for each 8K portion of the record as follows:

- 1 - No (only one).
- 2 - Yes (default if not specified).

Mem Protect Code

An octal code that specifies what types of memory protection the Loader is to set up for the record. It is a five-bit field. The operation is allowed if the bit is set.

Bit Number	5	4	3	2	1
Type	I	OR	OW	IA	IR

IR = 1 - Allows the use of interrupt index and base registers in indirect addressing.

IA = 1 - Allows indirect addressing.

- OW = 1 - Allows operand writing.
- OR = 1 - Allows operand reading.
- I = 1 - Allows instruction execution.

NOTE

If the memory protection code is not specified (default mode), the SYSMAKER inserts a code of 36 for type 2 or 4 records (instruction records) or a code of 16 for type 1 or 3 records (data records).

Length

A 1 indicates that the corresponding record has no initial length. The SYSMAKER puts no data on tape. The record name appears in the segment directory; and when the segment requests the Monitor to expand the record, the record is given space in memory (if dynamic).

I.2.1.4 Loader Initiate Card

The loader initiate card commands the SYSMAKER to transfer control to the Object Code Loader (described in Volume I, Section 3 of the User's Reference Manual). The cards following the Loader initiate card are inputs to the Object Code Loader until an END card appears. Then the Loader returns control to the SYSMAKER.

Format

LOAD counter release code

Explanation

Counter Release Code

A code that informs the SYSMAKER whether or not the SYSMAKER is to use previously derived data during the current processing and whether or not the SYSMAKER is to save the data (such as bound data or linkage tables) as follows:

- Blank - Compute new tables as required and discard when finished.

- 1 - Save load information for next loader command.
- 2 - Use load information generated by previous loader processing and save current load information for next loader command.
- 3 - Use load information generated by previous loader processing and discard when finished.

NOTE

When a 2 or 3 is used, no loader directives or END card follow.

I.2.1.5 Example Build or Insert Requests

```

$SYSMAKER |
| BUILD SYS037,8,SWAN,5,SYS036,111000 |
| SEG 1 |
| CREC PRESSET,2,2,36 |
| REC CONT,1,1,36 |
| REC DYNAM,3,2,14 |
| LOAD |
  
```

Loader directives

·
·
·

May have an object deck

·
·
·

```

| END |
| SEG 2 |
| REC INBUF,3,2,14,0 |
| CREC CARDS CAN,2 |
| LOAD |
  
```



```

$SYSMAKER
BUILD SYSØ38,8,SWAN, ,SYSØ37,1111ØØØ
SEG 4
CREC JACK,2
REC JOHN,2
REC JIM,2
REC INPUT,3,2
LOAD
  
```

Loader directives

·
·
·

May have an object deck

```

END
ENDSYSBD
  
```

The last card indicates the end of input to the SYSMAKER. The above example effectively adds a new segment (number 4) to component 8 on tape SYSØ37 and places the new system on tape SYSØ38.

I.2.2 Replace Function

The replace function is the process which replaces old segments with new ones. The new records of the segment retain all old directory information except length and possibly the initial base register. The old directory information is read into a table, after which the scratch tape is correctly positioned by copying everything up to the old segment. The Loader is called and returns the new records and their length information. The segment directory is updated and written on the scratch tape. Then the new records are written on the scratch tape, and the old segment is automatically deleted. Upon continued scanning, if an ENDSYSBD card is found, the scratch tape is file marked, and the new system tape is made with the new resident directory information.

This function uses two cards: the initiate replace card and the initiate load card. Paragraphs I.2.1.4 describes the format of the initiate load card. The format of the initiate replace card appears below.

Format

REPLACE output tape, comp no., seg no., asg name, input tape, mon loc,
 resdir offset, asg 2 name.

Explanation

Output Tape	The up to eight character name of the output system tape that the SYMAKER is to use in the console messages. If this field is blank, the routine assumes the name NEWSYSTP.
Comp No.	The number assigned to the component that contains the data to be replaced.
Seg No.	The number assigned to the segment that the SYMAKER is to replace with new data.
Asg Name	An up to four character name or version the routine assigns to the new tape. This name appears in the second word of the resident directory.
Input Tape	The up to eight character name of the input tape the SYMAKER is to use in the console messages. If this field is blank, the routine will use the current system tape.
Mon Loc	The absolute octal address of the start of the Monitor bootstrap load area. The current Monitor is always booted into the upper area of memory. This value then specifies the bottom of the resident Monitor load area and the top address available to the non-resident routines. If this field is blank, the routine assumes address 0.

M-5035
Change 2

The last card indicates the end of inputs to the SYSMAKER. The above example replaces segment 4 of component 8 on tape SYSØ38 with the data processed by the Loader. When complete, the routine writes a new system on tape SYSØ39.

I.2.3 Move Function

The move function changes the position of the specified component or segment on the system tape. First, it reads in the old resident directory information. The moved item is placed in the delete table. Then the copy routine is called which positions the scratch with the old components copied on it. The defined item is then written onto the scratch tape, along with the correct tape position record numbers being placed in the segment directory.

A move initiate card starts the move function. This card can command the SYSMAKER to change either the relative positions of the components in the system or the relative position of the segments within various components. However, the routine cannot change the relative positions of segments within components that it has moved while changing the relative positions of components. Conversely, the routine cannot move components that are affected by commands to alter the relative positions of internal segments.

Format

MOVE output tape,comp no.,comp pos,seg no.,seg pos,asg name,input tape,mon loc

Explanation

Output Tape

The up to eight character name of the output system tape the SYSMAKER is to use in console messages. If this field is blank, the routine assumes the name NEWSYSTP.

Comp No.

The number assigned to the component that the SYSMAKER either is to move to a new relative position on the system tape or is to change the relative positions of its internal segments.

Comp Pos

The number assigned to the component that is to immediately precede the component specified by the Comp No. field on the system tape. If this field is blank, the SYSMAKER assumes that the card changes the relative positions of segments within the component specified by the Comp No. field. If this field contains data, the Seg No. and Seg Pos field must contain blanks.

Seg No.

The number assigned to the segment (of the component specified by the Comp No. field) that the SYSMAKER is to move to a new relative position on the system tape within the component. This field must be blank when the routine is to move the component containing the segment to a new relative position. If this field contains data, the Comp Pos field must be blank.

Seg Pos

The number assigned to the segment (within the component specified by the Comp No. field) that is to immediately precede the segment specified by the Seg No. field on the system tape. This field must be blank when the routine is to move the component containing the segments to a new relative position. If this field contains data, the Comp Pos field must be blank.

Asg Name

An up to four character name or version the routine assigns to the new tape. This name appears in the second word of the resident directory.

Input Tape

The up to eight character name of the input tape that the SYSMAKER is to use in the console messages. If this field is blank, the routine will use the current system tape.

Mon Loc

The absolute octal address of the start of the Monitor bootstrap load area. The current Monitor is always booted into the upper area of memory. This value then specifies the bottom of the resident Monitor load area and the top address available to the non-resident routines. If this field is blank, the routine assumes address 0.

NOTE

The SYSMAKER uses the output tape, assign name, input tape, and Monitor octal load data on the first card processed after the \$SYSMAKER card. The routine ignores these fields on subsequent cards; therefore, these fields may be left blank.

Examples

```
MOVE | SYS040,5,8,,SWAN, SYS039, | | 2000 | | |
```

Move component number 5 to a position just behind component number 8. Notice that fields Seg No. and Seg Pos are blank on this card as they must be for a move component command.

```
MOVE | SYS041,6,,3,5,SWAN, SYS040, | | 2000 | | |
```

Move segment 3 of component 6 to a position just behind segment number 5. Notice that field Comp Pos is blank on this card as it must be for a move segment command.

I.2.4 Delete Function

The delete function prevents specified components or segments from appearing on the new system tape; thus, deleting them from the new system. The SYSMAKER, after processing an initiate delete card, removes all directory references to the specified component or segment. An initiate delete commands the routine either to remove an entire component or to remove one segment of a component.

Format

DELETE output tape, comp no., seg no., asg name, input tape, mon loc,
resdir offset, asg 2 name.

Explanation

Output Tape

The up to eight character name of the output system tape the SYSMAKER is to use in the console messages. If this field is blank, the routine assumes the name NEWSYSTP.

Comp No.

The number assigned to the component that the SYSMAKER is to delete from the new system or the number of the component containing the segment that the SYSMAKER is to delete from the new system.

Seg No.

The number of the segment within the component specified by the Comp No. field the SYSMAKER is to delete from the new system. If this field is blank, the routine deletes the entire component specified by the Comp No. field.

Asg Name

An up to four character name or version the routine assigns to the new tape. This name appears in the second word of the resident directory.

Input Tape

The up to eight character name of the input tape the SYSMAKER is to use in the console messages. If this field is blank, the routine will use the current system tape.

Mon Loc

The absolute octal address of the start of the Monitor bootstrap load area. The current Monitor is always booted into the upper area of memory. This value then specifies the bottom of the resident Monitor load area and the top address available to the non-resident routines. If this field is blank, the routine assumes address 0.

Resdir Offset

The offset from the beginning of the boot block (first jump cell) to the first address of the resident directory. If this field is blank, the input system tape offset will be assumed.

Asg 2 Name

A name or version, up to eight characters, which the routine assigns to the new tape. This name appears in the fourth and fifth words of the resident directory. The two assigned names are output upon completion of each job in the accounting summary.

NOTE

The SYSMAKER uses the output tape, assign name, input tape, resdir offset, and Monitor octal load point on the first card processed after the \$SYSMAKER card. The routine ignores these fields on subsequent cards; therefore, these fields may be left blank.

I.2.5 End of Input

The SYSMAKER continues to build a scratch tape containing all of the changes to the old system required to produce the new system until the routine processes an end of input card. The routine then copies the rest of the input tape onto the scratch and uses the accumulated change data to produce the new system.

Format

ENDSYSBD

I.2.6 System Tape Patching

The System Tape Patching routine reads system patch statement cards (see paragraph I.2.8), processes system patch cards, and outputs a new system tape with all patches inserted. After the SYSMAKER reads the initiate patch card control is given to the System Tape Patching routine. This routine is independent of the other four SYSMAKER functions and cannot be used concurrently with them.

Upon initiation of this routine, patch cards are read until a \$ card is read. The patch information is stored during the reading phase. In the next phase, the System Tape Patching routine copies from the input tape, inserts the patches, and copies onto the scratch tape. In the final phase, this routine inserts the new resident directory into the Monitor while copying from the scratch tape to the output tape.

The following control card initiates this routine:

Format

PATCH input tape, output tape

Explanation

Input Tape

Up to eight-character name of the input tape the SYSMAKER is to use in console messages. If this field is left blank, the routine assumes the current system tape.

M-5035
Change 2

Output Tape

Up to eight-character name of the system output tape the SYSMAKER is to use in console messages. If this field is blank, the routine assumes the name NEWPATTP.

Asg Name

An up to eight character name or version the routine assigns to the new tape. This name appears in the second and third words of the resident directory.

Resdir Offset

The offset from the beginning of the boot block (first jump cell) to the first address of the resident directory. If this field is blank, the input system tape offset will be assumed.

Asg 2 Name

A name or version up to eight characters which the routine assigns to the new tape. This name appears in the fourth and fifth words of the resident directory. The two assigned names are output upon completion of each job in the accounting summary.

Mon Loc

The absolute octal address of the start of the Monitor bootstrap load area. The current Monitor is always booted into the upper area of memory. This value then specifies the bottom of the resident Monitor load area and the top address available to non-resident routines. If this field is blank, the routine assumes address 0.

I.2.7 Copying System Tape

The SYSMAKER duplicates a system tape when the \$SYSMAKER control is followed by an ENDSYSBD card. The operation reads from an input system tape and writes on an output tape.

Format

ENDSYSBD Input Tape, Output Tape

Explanation

Input Tape The up to eight character name of the input tape the SYSMAKER is to use in the console messages. If this field is blank, the routine will use the current system tape.

Output Tape The up to eight character name of the new copy of the system tape that the SYSMAKER is to use in the console messages. If this field is blank, the routine assumes the name NEWCOPTP.

The SYSMAKER requests a scratch tape during the initialization phase and releases the scratch tape during the copy function. The operator may assign the output tape to the previously assigned scratch tape. Different output names must be declared for multiple copies in one job. A word-by-word comparison of the input tape and output tape can be made via the peripheral utilities compare function.

I.2.8 System Patch

The system patch control card calls the system patch statement processor to process system patch statements that follow the specify system program correction(s). A patch statement must precede the load of the program being corrected.

Format

\$SPATCH

Patch statement cards following the system patch control card contain the actual data. The statement processor modifies the specified component each time it is loaded into memory. Commas are used to separate all parameters. Spaces may be used to separate functional parts of a patch. Any number of consecutive spaces may surround the comma. No more than one starting location may be specified on one system patch statement.

Format

component, location, type, patch, patch...

Each successive patch affects successive memory locations.

Explanation

Component

A decimal number specifying one of the system components on the system tape.

Location

A location specified by a system record name (compound section name in a component of up to eight characters) followed by an optional octal or decimal increment, where a decimal value is followed by a D. This parameter specifies a starting location for a half-word or more of patch information.

Type

The patch statement type is as follows:

I - Instruction

K - I/O Controller command

O - Octal (a signed number consisting of up to eleven octal digits).

C - Variable length Hollerith character string converted to ASCII. If the length of this character string is not a multiple of four characters, the last patched word will be filled on the right with space codes.

Patch

Information in a format determined by the patch type.

Patch type formats include the following:

1. Central Processor Instruction (I)

The patch processor interprets the function code (FF) to determine the proper format. The patch processor considers all function codes legal.

Non-assigned function codes 00 and 04 are treated as format II instructions, and non-assigned function codes 72, 73, and 75 are treated as format IVA instructions.

As shown below, all full length instructions consist of octal digits. The first six digits are interpreted as FFAKBI, FFAF₂BI, or FFAF₃BI, as applicable. The lower digits form SYYYYY. If there are less than twelve digits, the left six digits form the upper half of the word. The remaining digits form the S, Y, and SY fields, right-justified.

Half-word instructions consist of six digits for each half-word. If only six digits are specified, the patch processor places the instruction in the upper half and all zeros in the lower half (a no-op).

The following list shows hardware formats and the corresponding coding sequence:

<u>Hardware Format</u>	<u>Coding Sequence</u>
I	FFAKBISYYYY
II	FFAF ₂ BISYYYY
III	FFAF ₃ KBISYYYY
	F ₃ K is coded as one octal digit.
IVA	FFAF ₄ BI
IVB	FFAF ₅ MM (F ₅ = 0)
	FFAF ₅ RBI
	F ₅ R = 4
	Shift count in B _b .
	F ₅ R = 6
	Shift count in A _b .

2. I/O Controller Commands (K)

I/O Controller Commands are coded in one of two formats:

<u>Hardware Format</u>	<u>Calling Sequence</u>
I (FF = 10-16)	FFKJMC YYYYYY
II (FF = 17-20, 22-27)	FFKJMC YYYYYY

Any function code (FF) not in the range 10 to 20, 22 to 27 shall be interpreted as a format I IOC command.

3. Octal (O)

The patch statement processor accepts a leading plus or minus sign followed by as many as eleven octal digits. A valid octal number beginning with an octal digit is stored as coded. A leading minus sign causes the number following it to be complemented (sevens complement) before storage. If fewer than eleven octal digits are coded, the resultant value is right-justified within the word (before complementing, if preceded by a minus sign).

4. Character (C)

The patch statement processor accepts character strings containing graphic ASCII characters coded as follows:

(C C ...Cn)

Where the desired variable length character string is enclosed in parentheses. If a right parenthesis is desired as part of the string, it is necessary to code two for each right parenthesis wanted. The character string terminates with an odd number of consecutive right parentheses. If the character string does not result in a number of characters which is an even multiple of four, the last word patched is filled on the right with ASCII space codes.

Examples (system patch cards)

```
$SPATCH
1, CARDSCAN+50, 1, 10300000075
```

Change the instruction located in word 50_8 of record CARDSCAN in component 1 to the instruction indicated (load A3 with a value of 75_8).

```
$SPATCH
1, CARDSCAN+500, C, (ILLEGAL OP-CODE AT )
```

Change the character string starting at word 500_8 of record CARDSCAN in component 1 to the string indicated. This string occupies five words.

```
$SPATCH
1, RALPH+776, 1, 44400025
```

Change the instruction located in word 776 of record RALPH in component 1 to the instruction indicated (compare (A4) with a value of 25_8). Because the lower digits describe a constant, all six digits are not required.

```
$SPATCH
2, INSERT+543, 1, 712130 704200, 745200
```

Change the instructions located in words 543 and 544 of record INSERT in component 2 to the instructions indicated (half add (A2) + (A3) and complement (A4) instructions in word 543 with a square root (A5) instruction followed by all zeros in word 544). The patch processor ignores the space between the two half-word instructions listed for word 543.

I.2.9 User Segment Addition and Execution

User segments can be added to the utilities component of the system tape and subsequently executed. A typical SYSMAKER run to add a utility segment might appear as follows:

```
$JOB
$SYSMAKER
BUILD NEWTAPE, 3, XXLYY,,OLDTAPE,115000
SEG 2
REC DATAREC, U, 2, 36
CREC INSTREC, 1, 2, 36
LOAD
DATAREC (1), SYSDD1, SYSDD2
INSTREC (4), SYSPROC1, SYSPROC2
LIBS CCOMN(REELNUM)
SEL-ELEM
END
ENDSYSBD
$ENDJOB
```

The user's program is contained in records DATAREC and INSTREC. The five statements after LOAD are loader commands discussed in Section 3. The first two statements after LOAD are combine elements commands for the loader.

To execute the added segment, the following sequence of operations may be used:

```
$JOB
$UTILITY
EXECUTE 2
$ENDJOB
```

Segment 2 is accessed in the execute with no return mode. Therefore, further utility commands may not follow the EXECUTE command. The user added segment is operating in the system mode, not the user mode, with all system facilities (such as those XS calls not available to a user program) available for use.

CMS-2 KEYWORD INDEX

ABS, II-9-3
ABS Directive, II-11-21
ABS Modifier, II-5-10
Accessing the Compiler, II-7-1
AC Directive, I-3-12
ACKN, I-2-30, I-2-31
Active State Register, II-12-9
Address Counter, II-11-55
Address Counter Declaration,
II-11-53
Allocated Map Listing, I-3-36
Allocation Header Statements,
II-7-124
AN/UYK-20 Loader, I-3A-1
A Register, II-12-3
Arithmetic Assignment State-
ment, II-5-27
Arithmetic Expressions, II-5-1
Arithmetic Operators, II-11-64
Assembler, I-1-8, II-1-8, II-11-1
Assembler Diagnostics, II-11-76
Assembler Outputs, II-11-72
Assembly Errors, II-11-76
AUTO-DD Declarative, II-4-1,
II-4-7
BASE, II-7-4, II-7-17
BASE Directive, I-3A-3
BC Directive, I-3-18
BEGIN Block, II-5-60
BIT Modifier, II-5-10
BKFILSKP, I-5-6
BKRECSKP, I-5-7
Blocks, II-5-59
Boolean Assignment Statement,
II-5-33
Boolean Expressions, II-5-6
BOOTWRT, I-5-14
BP Register, II-12-5
B Register, II-12-3
BYTE, II-9-3
BYTE Directive, II-11-21
CARD-ID, II-9-2
CARDTAPE, I-5-12
Card-to-Tape, I-5-22
CHAR, II-9-3
CHAR Directive, II-11-22
CHAR Modifier, II-5-12
Character Strings, II-9-8,
II-11-60

CMS-2 KEYWORD INDEX (contd)

CHECKID, II-6-35
CLOSE, II-6-26
CMODE, II-7-4, II-7-32
CNT Modifier, II-5-13
Combine Elements, I-3-9
Commands Available, I-6-9
COMMENTS, II-11-7
COMPARE, I-5-19
Compare Tape, I-5-24
Compiler, I-1-7, II-1-7
Compiler Error Summary, II-10-12
Compiler Outputs, II-10-1
Compile-Time System Facilities,
II-7-1
Complex Macros, II-11-52
Compound Decision Statements,
II-5-1
Computer Instruction Repertoire,
II-12-26
Conditional Operators, II-11-67
Console Message Output, I-1-9,
I-2-63
Constants, II-9-6, II-11-59
Control Declaratives, II-4-62
 DATA, II-4-61
 MODE, II-4-63
Control Statements, II-5-41
CONVERT, I-5-15
CORAD Modifier, II-5-14
CORRECT, II-7-4, II-7-25
CPMCR Register, II-12-5
Cross Reference Listing, II-11-76
CS Directive, I-3-14, I-3A-2
CSWITCH, II-7-4
CSWITCH Brackets, II-7-30A
CSWITCH Declarations, II-7-30
CSWITCH-DEL, II-7-30B
CSWITCH Deletion, II-7-30B
CSWITCH-OFF, II-7-4, II-7-30
CSWITCH-ON, II-7-4, II-7-30
Data Declarations, II-4-14
 END-TABLE, II-4-35
 FIELD, II-4-25
 ITEM-AREA, II-4-30
 LIKE-TABLE, II-4-34
 SUB-TABLE, II-4-31
 TABLE, II-4-19
 VRBL, II-4-15
DATA Declarative, II-4-61
Data Expressions, II-9-9
Data Modes, II-11-58
DATAPOOL, II-7-4, II-7-18
Data Referencing, II-4-44
Data Words, II-11-59

CMS-2 KEYWORD INDEX (contd)

DEBUG, II-7-4, II-7-29
Debugging Aid Cards, I-2-7,
I-2-18, I-2-60
 Memory Dump Card (\$DUMP),
 I-2-18
 Patch Card (\$PATCH),
 I-2-26
 Snap Card (\$SNAP), I-2-21
Debug Statements, II-8-1
 DISPLAY, II-8-2
 PTRACE, II-8-11
 RANGE, II-8-7
 SNAP, II-8-5
 TRACE, II-8-9
Decimal Numbers, II-9-6,
II-11-56
Decision Statements, II-5-45,
II-5-50, II-5-54, II-5-55,
II-5-58
Declarative Statements, II-2-2
DECODE Statement, II-6-15
DEFID, II-6-33
DENSE, II-4-39
DEP, II-7-4, II-7-25
Device Identification Opera-
tions, II-6-33, II-6-35
 CHECKID, II-6-35
 DEFID, II-6-33
Device Positioning, II-6-28
Device State Checking, II-6-26
Direct Code, II-9-1
 ABS, II-9-3
 BYTE, II-9-3
 CARD-ID, II-9-2
 CHAR, II-9-3
 DO, II-9-4
 FORM, II-9-4
 RES, II-9-4
Direct Constant Entries, II-9-11
Directives, II-11-8, II-11-21
 ABS, II-11-21
 BYTE, II-11-21
 CHAR, II-11-22
 DO, II-11-23
 ELIST, II-11-32
 EMBED, II-11-25
 END, II-11-26
 EQU, II-11-26A
 EVEN, II-11-26B
 FORM, II-11-27
 LCR, II-11-28
 LIB, II-11-29
 LIBS, II-11-29
 LINK, II-11-31
 LIST, II-11-32
 LIT, II-11-32
 LLT, II-11-34
 NOLIST, II-11-32
 ODD, II-11-26B
 PXL, II-11-34
 RES, II-11-35
 RF\$, II-11-35
 SEGEND, II-11-36
 SETADR, II-11-37
 WRD, II-11-38
DISMOUNT, I-5-5
Dismount Tape Message, I-5-22
DISPLAY, I-2-61

CMS-2 KEYWORD INDEX (contd)

DISPLAY Statement, II-8-2
DO, II-9-4
DO Directive, II-11-23
Double Procedure Switch Declarative, II-4-56
DP Register II-12-6
DSW Register ASR, II-12-6
DSW Register ISC, II-12-6
DUPLICATE, I-5-10
Duplicate (D), II-11-72
Duplicate Tape, I-5-22
Dynamic Statements, II-2-2, II-5-1
ELSE Statement, II-5-56
ELIST Directive, II-11-32
EMBED Directive, II-11-25
ENCODE Statement, II-6-15
END, I-3-11, II-11-39
END-AUTO-DD Declarative, II-4-1, II-4-8
END-CSWITCH, II-7-4
END-CSWITCHS, II-7-4
END Directive, II-11-26
ENDFILE, II-6-24
END-FUNCTION Declarative, II-4-1, II-4-11
END-HEAD, II-7-3
END-HEAD Declarative, II-4-1, II-4-3
END-LOC-DD Declarative, II-4-1, II-4-7
END-PROC Declarative, II-4-1, II-4-8
END Statement, II-5-60, II-5-67
END-SYS-DD Declarative, II-4-1, II-4-4
END-SYS-PROC Declarative, II-4-1, II-4-13
END-SYSTEM Declarative, II-4-1, II-4-14
END-TABLE Declarative, II-4-35
END VARY Statement, II-5-67
EP Directive, I-3-17
EQUALS, II-7-4, II-7-19
EQU Directive, II-11-26A
Error Codes, II-11-72
ESI Mode, II-12-17
EVEN Directive, II-11-26B
EXCHANGE, II-7-4, II-7-28
EXEC Function, II-5-22
EXECUTIVE, II-7-4, II-7-31
Expression (E), II-11-72
Expressions, II-5-2, II-11-11

CMS-2 KEYWORD INDEX (contd)

Expression Statements, II-11-54
EXTDEF Linkage, II-4-65
Externalizing Labels, II-11-50
EXTREF Linkage, II-4-66
Field, II-4-47, II-11-5
FIELD Declarative, II-4-25
FILE, II-6-20
File and Record Position
Determination, II-6-31
File Search Operation, II-6-35
 SEARCH, II-6-36
FIL Modifier, II-5-14
FILSKP, I-5-6
FIND Statement, II-5-48
Fixed Point Number, II-11-58
Floating Point (F), II-11-74
Floating-Point Numbers,
II-9-7, II-11-57
FOR Block, II-5-72, II-5-77
FOR Statement, II-5-73
FORCE, I-3-6A
FORM, II-9-4
Format I, II-12-12
Format II, II-12-13
Format III, II-12-13
Format IV-A, II-12-14
Format IV-B, II-12-14
FORMAT Statement, II-6-8
FORM Directive, II-11-27
Full-word, II-11-11
Function Call, II-5-18
 EXEC, II-5-22
 RETURN, II-5-19
FUNCTION Declarative, II-4-1,
II-4-11
Functional Modifiers, II-5-9
 ABS, II-5-10
 BIT, II-5-10
 CHAR, II-5-12
 CNT, II-5-13
 CORAD, II-5-14
 FIL, II-5-14
 LENGTH, II-5-15
 POS, II-5-15
Generation Formats, II-11-74
Global Cross-Reference Listings,
II-10-5
GO, II-11-47
GOTO Statement Name, II-5-41
GOTO Switch Name Statement,
II-5-42
Half-word, II-11-11
HALT, I-5-3
Hardcopy Output, I-1-9, I-2-37

CMS-2 KEYWORD INDEX (contd)

HEAD, II-7-3

HEAD Declarative, II-4-1,
II-4-2

ICW Register, II-12-5

ID Directive, I-3-17

Index Clause II-5-64

Index Procedure Switch De-
clarative, II-4-55

Index Switch Declarative,
II-4-50

Indirect Word, II-12-15

Initial A-Register Values
(\$AREG), I-2-16

Initial B-Register Values
(\$BREG), I-2-16

Initial Condition Cards,
I-2-16

Initial Hardware Setup,
I-6-1

Initiate Execution Cards,
I-2-17

 Jump Key Set Card
 (\$KEYSET), I-2-17

 Transfer Control Card
 (\$TRA), I-2-18

 User Program Call Card
 (\$CALL), I-2-17

Input/Output Statements,
II-6-1

INPUT Statement, II-6-4

Instruction Expressions, II-9-13

Instruction Formats, II-12-12

Instruction (I), II-11-73

Instruction Repertoire, II-12-1

Interrupt, I-2-7

Interrupt State, II-12-8

I/O Command Formats, II-12-16

I/O Device Card, I-2-12

Item, II-4-45

Item-Area, II-4-49

ITEM-AREA Declarative, II-4-30

Item Procedure Switch Declara-
tive, II-4-58

Item Switch Declarative, II-4-53

Item-to-Item Statement, II-5-36

Job Definition Cards, I-2-9

 End of Input Card (\$EOI)
 I-2-12

 End of Job Card (\$ENDJOB),
 I-2-12

 I/O Control Cards (\$ASG),
 I/O Device Card, I-2-12

 Job Limits Card (\$JOB),
 I-2-10

 Sequence Card (\$SEQ), I-2-9

LABEL, II-11-2

Label Field, II-11-15

Labels, II-11-16, II-11-55

CMS-2 KEYWORD INDEX (contd)

LCR Directive, II-11-28
LENGTH Modifier, II-5-15
Level 0, II-11-75
Level 1, II-11-75
Level (L), II-11-74
LIB Directive, II-11-29
Librarian, I-1-7, I-4-1,
II-1-7
Librarian Control Cards,
I-4-5
 Add Element (/ADD),
 I-4-10
 Begin Element (/BEGINEL),
 I-4-19
 Build New Library
 (/BUILD), I-4-6
 Change Element Name
 (/CHANGE), I-4-17
 Copy Elements (/COPY),
 I-4-12
 Declare Dependent
 Elements (/DEP),
 I-4-19
 Delete Elements (/DEL),
 I-4-15
 Delete Items (/D),
 I-4-22
 Edit (/EDIT), I-4-7
 End Corrections
 (/ENDCOR), I-4-20
 End Element (/ENDEL),
 I-4-19
 End Element Corrections
 (/END), I-4-21
 History Entry (/HISTENT),
 I-4-9
 Insert Items (/I),
 I-4-21
 Librarian Terminate,
 (/ENDLIB), I-4-7
List Job (/LIST), I-4-6
Start Corrections (/CORRECT),
 I-4-20
Start Element Corrections
 (/ELname), I-4-20
Tape Release (/RELEASE),
 I-4-8
Tape Select (/TAPID),
 I-4-8
Librarian Error Messages, I-4-30
Librarian Operator Messages,
I-4-32
LIBS, I-3-7, II-7-4, II-7-23
LIBS Directive, II-11-29
LIKE-TABLE Declarative, II-4-34
LINK Directive, II-11-31
Linking, II-11-12
List Options, II-7-8
LIST Directive, II-11-32
LISTING, II-7-5, II-7-10
Listing of Labels, II-11-75
LIT Directive, II-11-32
Literal Assignment Statement,
II-5-31
Literal Expressions, II-5-8
Literals, II-9-11, II-11-58
LLT Directive, II-11-34
LLT Sample Listing, II-11-75
Loader Control Cards, I-3-2,
I-3-6

CMS-2 KEYWORD INDEX (contd)

- Combine Elements, I-3-9
- Element Select (SEL-ELEM), I-3-8
- End (END), I-3-11
- Library Select (LIBS), I-3-7
- Table Size Declaration (TSD), I-3-7
- Loader Diagnostic Messages, I-3-32
- Loader Directives, I-3-12
 - Address Counter Directive (AC), I-3-12
 - Binary Code Directive (BC), I-3-18
 - Compound Section Directive (CS), I-3-14
 - End of Element Directive (EP), I-3-17
 - Library Reference Directive (LR), I-3-16
 - Program Element Identification Directive (ID), I-3-17
 - User Correction Directive (UC), I-3-26
 - Loader Options Select, I-3-6A
- LOBJECT, I-3-6B
- Local Cross-Reference Listings, II-10-4
- Local Indexes, II-4-59
- LOC-DD Declarative, II-4-1, II-4-6
- LOCDDPOOL, II-7-4, II-7-14
- LODGO, I-3-6B
- LOCREF Operator, II-4-67
- Logical Operators, II-11-67
- Logical Statement, II-5-46
- Loop Statements, II-5-50A
- LOPTIONS, I-3-6A
- LR Directive, I-3-16
- MACRO, II-11-39
- Macros, II-11-10
- Macro Reference Lines, II-11-52
- Macro Statements, II-11-39
- Major and Minor Headers, II-7-3
 - BASE, II-7-4, II-7-17
 - CMODE, II-7-4, II-7-32
 - CORRECT, II-7-4, II-7-25
 - CSWITCH, II-7-4, II-7-30
 - DATAPPOOL, II-7-4, II-7-18
 - DEBUG, II-7-4, II-7-29
 - DEP, II-7-4, II-7-25
 - EQUALS, II-7-4, II-7-19
 - END-HEAD, II-7-3
 - EXCHANGE, II-7-4, II-7-28
 - EXECUTIVE, II-7-4, II-7-31
 - HEAD, II-7-3
 - LIBS, II-7-4, II-7-23
 - LISTING, II-7-5, II-7-10
 - LOCDDPOOL, II-7-4, II-7-14
 - MEANS, II-7-4, II-7-27
 - MONITOR, II-7-5, II-7-11
 - NITEMS, II-7-4, II-7-22
 - OBJECT, II-7-5, II-7-7
 - OPTIONS, II-7-4, II-7-5
 - SEL-ELEM, II-7-4, II-7-24
 - SEL-HEAD, II-7-4, II-7-24
 - SEL-POOL, II-7-4, II-7-24
 - SEL-SYS, II-7-4, II-7-24
 - SOURCE, II-7-5, II-7-6
 - SPIII, II-7-4, II-7-32
 - SYS-INDEX, II-7-4, II-7-27
 - TABLEPOOL, II-7-4, II-7-16

CMS-2 KEYWORD INDEX (contd)

MEANS, II-7-4, II-7-27
MEDIUM, II-4-38
Miscellaneous Header Statements,
II-7-27
Mnemonics, II-11-20
Mnemonic Listing Format,
II-4-63
MODE Declarative, II-4-63
MODEVRBL Option, II-7-12
Modes of Operation II-12-8
Monitor, I-1-6, I-2-1, II-1-6,
II-7-5, II-7-11
Monitor Control Cards, I-2-2,
I-2-5, I-2-8
Monitor I/O, I-2-5, I-2-6
Monitor Loader, I-2-3
MOUNT, I-5-4
Mount Tape Message, I-5-21
Multiword Assignment State-
ment, II-5-35
NAME, II-11-45
Name (N), II-11-73
Nested Decision Statements,
II-5-58
NITEMS, II-7-4, II-7-22
NOID, I-3-6A
Nolist Directive, II-11-32
NOMAP, I-3-6A
NONE, II-4-38
NONRT Option, II-7-12
Nonstandard File Control, II-6-18
CLOSE, II-6-26
ENDFILE, II-6-24
FILE, II-6-20
OPEN, II-6-24
Normal Mode, II-12-16
OBJECT, II-7-5, II-7-7
Object Code Loader, I-1-6, I-3-1,
II-1-6
ODD Directive, II-11-26
OFF, II-11-14B
OFO, II-11-14B
Octal Numbers, II-9-7, II-11-56
OPEN, II-6-24
Operand Field, II-11-44
Operation, I-6-1
Operation Field, II-11-19
Operator Communication, I-3-36
Operator Communication Cards,
I-2-30
Enter Executive State (XS),
I-2-32

CMS-2 KEYWORD INDEX (contd)

Hardcopy Message (\$REMARK),
I-2-31

Operator Message Cards, I-5-2

 Dismount Tape (DISMOUNT)
 I-5-5

 Mount Tape (MOUNT), I-5-4

 Operator Acknowledge
 (HALT), I-5-3

 Operator Information
 (TYPE), I-5-3

Operator Priorities, II-11-69

Operators, II-11-61

OPTIONS, II-7-4

OUTPUT Statement, II-6-7

Overflow (O), II-11-73

OVERLAY, II-4-40

Packing Rules, II-4-37

 DENSE, II-4-39

 MEDIUM, II-4-38

 NONE, II-4-38

 OVERLAY, II-4-40

PACK Statement, II-5-40

Paraforms, II-11-40

PARAMETER, II-4-18A

Parenthetical Grouping,
II-11-70

Parity Decision Statement,
II-5-55

PAUSE, I-2-30, I-2-31

Performing the Bootstrap Load,
I-6-3

Peripheral Utilities, I-1-7,
I-5-1

Pooling Statements, II-7-12A

Positioning by Files, II-6-28

Positioning by Records, II-6-30

Position Tape Cards, I-5-5

 Rewind (REWIND), I-5-7

 Skip Backward Specified
 Number of Files
 (BKFILSKP), I-5-6

 Skip Backward Specified
 Number of Records
 (BKRECSKP), I-5-7

 Skip Forward Specified
 Number of Files (FILSKP),
 I-5-6

 Skip Forward Specified
 Number of Records
 (RECSKP), I-5-6

POS Modifier, II-5-15

P Register, II-12-2

Procedure Call, II-5-16

PROCEDURE Declarative, II-4-1,
II-4-8

Procedure Linking, II-5-15

Procedure Switch Call, II-5-23

Processing Conventions, II-9-15

Program Structure Declaratives,
II-4-1

 AUTO-DD, II-4-1, II-4-7

 END-AUTO-DD, II-4-1, II-4-8

CMS-2 KEYWORD INDEX (contd)

END-FUNCTION, II-4-1,
II-4-11
END-HEAD, II-4-1, II-4-3
END-LOC-DD, II-4-1, II-4-7
END-PROC, II-4-1, II-4-8
END-SYS-DD, II-4-1, II-4-4
END-SYS-PROC, II-4-1,
II-4-13
END-SYSTEM, II-4-1, II-4-14
FUNCTION, II-4-1, II-4-11
HEAD, II-4-1, II-4-2
LOC-DD, II-4-1, II-4-6
PROCEDURE, II-4-1, II-4-8
SYS-DD, II-4-1, II-4-4
SYS-PROC, II-4-1, II-4-5
SYS-PROC-REN, II-4-1,
II-4-6
SYSTEM, II-4-1, II-4-2

Pseudo-operations (MACROS),
I-1-8

P-SWITCH Declarative, II-4-55

PTRACE, I-2-61

PTRACE Statement II-8-11

PXL Directive, II-11-34

RANGE, I-2-61

RANGE Statement, II-8-7

READ, I-5-17

READ/Compare Tape Cards, I-5-17
 Compare Tapes (COMPARE),
 I-5-19
 Read into Memory (READ),
 I-5-17

 READ to Cards or Printer
 (TAPEOUT), I-5-18

Read Formatted Tape Record,
I-5-23

Record Length Determination,
II-6-32

RECSKP, I-5-6

REFORMAT, I-5-11

Reformat Tape I-5-22

Relational Expressions, II-5-5

Relocatability, II-11-70

Relocation (R), II-11-73

Replacement Statements, II-5-26

RES, II-9-4

RES Directive, II-11-35

RESUME Statement, II-5-66

RETURN Function, II-5-19

REWIND, I-5-7

RF\$ Directive, II-11-35

SAVL0DGO, I-3-6B

Scaled Decimal Numbers, II-9-9

Search Decision Statement, II-5-50

SEGEND Directive, II-11-36

Segmentation, II-11-8

SEL-ELEM, I-3-8, II-7-4, II-7-24

SEL-HEAD, II-7-4, II-7-24

SEL-POOL, II-7-4, II-7-24

SEL-SYS, II-7-4, II-7-24

SETADR Directive, II-11-37

CMS-2 KEYWORD INDEX (contd)

SET Statement, II-5-26
SHIFT Statement, II-5-39
Side-by-Side Listing, II-11-72
Simple Statement, II-5-1
Single Word-to-Multiword Statement, II-5-36
SIR Registers, II-12-7
SNAP, I-2-61
SNAP Statement, II-8-5
SOURCE, II-7-5, II-7-6
Source Deck Organization, II-11-79
Source Listing Format, II-10-1
Special Considerations, II-11-82
SPILL, II-7-4, II-7-32
SPR Registers, II-12-6
S Register, II-12-3
Standard Input, I-1-9, I-2-33
Standard Output, I-1-9, I-2-35
 DISPLAY, I-2-61
 PTRACE, I-2-61
 RANGE, I-2-61
 SNAP, I-2-61
 Tape Assignment/Release, I-2-33, I-2-49
 TRACE, I-2-61
 Type Message (\$TYPE), I-2-30
 Type Message PAUSE (ACKN) (\$HALT), I-2-30, I-2-31
Statements, II-11-5
Statement Blocks, II-5-59
Status Assignment Statement, II-5-32
STOP Statement, II-5-45
Subfields, II-11-5
SUB-TABLE Declarative, II-4-31
SWAP Statement, II-5-38
SWITCH Declarative, II-4-50
Switch Referencing, II-4-59
Symbol Analysis Format, II-10-6
Symbol Definitions, II-12-19
Symbolic Conventions, II-12-17
Symbols, II-11-62
SYS-DD Declarative, II-4-1, II-4-4
SYS-INDEX, II-7-4, II-7-27
SYSMAKER, I-2-15
SYS-PROC Declarative, II-4-1, II-4-5
SYS-PROC-REN Declarative, II-4-1, II-4-6
System (CMS-2), I-1-3, I-2-3, II-1-3
System Component Call Cards, I-2-14
 Call Assembler (\$ASM, U), I-2-15

CMS-2 KEYWORD INDEX (contd)

Call CMS-2 Compiler
(\$CMS-2), I-2-15
Call Librarian
(\$LIBEXEC), I-2-15
Call Peripheral Utility
Routines (\$UTILITY),
I-2-15
Call Relocatable Object
Code Loader (\$LOAD),
I-2-15
SYSTEM Declarative, II-4-1,
II-4-2
System Linkage, II-4-64
 EXTDEF, II-4-65
 EXTREF, II-4-66
 TRANSREF, II-4-66
System Messages, I-6-14
System Tape Generator, I-1-8,
II-1-8
Table, II-4-44
TABLE Declarative, II-4-19
TABLEPOOL, II-7-4, II-7-16
Table Search Statement,
II-5-48
Table-to-Table Statement,
II-5-35
TAPEOUT, I-5-18
Tape-to-Card or Printer,
I-5-23
Task State, II-12-8
TRA Directive, I-3A-4
TRACE, I-2-61
TRACE Statement, II-8-9
Transfer Declaratives, II-4-50
 Double Procedure Switch,
 II-4-56
 Index Procedure Switch,
 II-4-55
 Index Switch, II-4-50
 Item Procedure Switch,
 II-4-58
 Item Switch, II-4-53
 P-SWITCH, II-4-55
 SWITCH, II-4-50
TRANSREF Linkage, II-4-66
Truncation (T), II-11-73
TSD, I-3-7
Two-Level Diagnostics, II-7-12
TYPE, I-5-3
Type Message and Await Response,
I-5-21
Type Operator Messages, I-5-21
UC Directive, I-3-26
ULTRA, II-11-12
Undefined Labels, II-11-75
Undefined (U), II-11-73
UNTIL Clause, II-5-66
User I/O Chains, I-2-48
User Program Execution Cards,
I-2-15
User Routine Execution, I-5-27

CMS-2 KEYWORD INDEX (contd)

Utility Control Cards, I-5-2	XS, I-2-32
Validity Statement, II-5-48	\$AREG, I-2-16
Value Block, II-5-76	\$ASG, I-2-12
VARY Block, II-5-61, II-5-67	\$ASM, U, I-2-15
VARY Statement, II-5-62	\$BREG, I-2-16
VRBL Declarative, II-4-15	\$CALL, I-2-17
Warning, II-11-75	\$CMS-2, I-2-15
WHILE Clause, II-5-65	\$DUMP, I-2-18
Whole Table, II-4-45	\$ENDJOB, I-2-12
WRD Directive, II-11-38	\$EOI, I-2-12
WRITE, I-5-8	\$FORTRAN, I-2-15
Write Formatted Tape Record, I-5-22	\$HALT, I-2-30, I-2-31
Write on Tape Cards, I-5-8	\$JOB, I-2-10
Convert to System Data Card (CONVERT), I-5-15	\$KEYSET, I-2-17
Duplicate Tape (DUPLICAT), I-5-10	\$LDUYK-20, I-2-15, I-3A-1
Reformat Tape (REFORMAT), I-5-11	\$LIBEXEC, I-2-15
Write End of File Mark (WRTFILMK), I-5-10	\$LOAD, I-2-15
Write from Memory (WRITE), I-5-8	\$PATCH, I-2-26
Write in Bootstrap For- mat (BOOTWRT), I-5-14, I-5-23	\$REMARK, I-2-31
Write Tape from Cards (CARDTAPE), I-5-12	\$SEQ, I-2-19
	\$SNAP, I-2-21
	\$SYSMAKER, I-2-15
WRTFILMK, I-5-10	\$TRA, I-2-18

CMS-2 KEYWORD INDEX (contd)

\$TYPE, I-2-30
\$UTILITY, I-2-15
/ADD, I-4-10
/BEGINEL, I-4-19
/BUILD, I-4-6
/CHANGE, I-4-17
/COPY, I-4-12
/CORRECT, I-4-20
/D, I-4-22
/DEL, I-4-15
/DEP, I-4-19
/EDIT, I-4-7
/EL name, I-4-20
/END, I-4-21
/ENDCOR, I-4-20
/ENDEL, I-4-19
/ENDLIB, I-4-7
/HISTENT, I-4-9
/I, I-4-21
/LIST, I-4-6
/RELEASE, I-4-8
/TAPID, I-4-8

