

MICROPOLIS
1040/1050 S-100
FLOPPY DISK SUBSYSTEMS
USER'S MANUAL

MODEL NO. 1053-II

DOCUMENT NUMBER 100089-01
REVISION 10 - APRIL 1979

PROPRIETARY NOTICE

Information contained in this manual may not be duplicated in full or in part by any person without prior written consent of Micropolis Corporation. The sole purpose of this manual is to provide the user with adequately detailed documentation to efficiently install and operate the equipment supplied by Micropolis and to write programs using Micropolis Software. The use of this document for all other purposes is prohibited.

MICROPOLIS CORPORATION, 7959 DEERING AVENUE, CANOGA PARK, CALIFORNIA 91304

MICROPOLIS™

COPYRIGHT 1979

FOREWORD

This manual provides operating and programming instructions for the Micropolis 1040/1050 S-100 Series Floppy Disk Sub-systems. The first three chapters provide a detailed description of the physical system integration process from unpacking the system components to defining system configurations. The fourth chapter deals with the Micropolis Diskette Operating System. The fifth chapter deals with Micropolis Disk Extended BASIC. Chapter six indicates disk access techniques independent of BASIC or DOS.

This manual does not deal with maintenance. See 1040/1050 S-100 Series Floppy Disk Sub-systems Maintenance Manual, document number 100090-01-8.

The latest revision of each page has been included with this manual. The individual pages of this manual are subject to replacement under new releases or versions of the software. Such replacement or additional changes are given the date of the change.

The initial release of a Micropolis software product is given the number 1.0, meaning release 1, version 0. Any corrections and amplifications to the software are accumulated and issued under a new version number; thus, the first revision version number is 1.1. When a major group of new features is added, plus the accumulated corrections of earlier versions, a new release number is assigned, such as 2.0.

Please read this manual thoroughly as to installation and operation. Should you require additional assistance in servicing this equipment, please contact your dealer who sold it (or MICROPOLIS in the case of direct purchase).

LIMITED 90 DAY WARRANTY

Micropolis Corporation, 7959 Deering Avenue, Canoga Park, California 91304, telephone (213) 703-1121, warrants the electrical and mechanical parts of its products to be free from defects in design, materials and workmanship for a period of ninety (90) days from date of delivery to the original end user. Should a product prove defective during the warranty period, it will be repaired or replaced free of charge.

Software supplied with the product is warranted to conform to Micropolis' software product description applicable at the time of order. Micropolis' sole obligation with respect to the warranty of its software is to remedy any nonconformance.

In order to validate this warranty, the warranty registration form found in the front of the user's manual must be completed and returned to Micropolis within ten (10) days from date of purchase.

If the product was purchased from a computer store, it must be returned to such store for repair, along with the original shipping carton, if possible.

If, and only if, the product was purchased directly from Micropolis, in order to obtain repairs, the product should be carefully packaged in the original shipping carton and sent (preferably by United Parcel Service) to Micropolis at the above address, attention: Customer Service. Prior to shipment Return Authorization Number should be obtained from Micropolis Customer Service. You should include a note in the package giving your name, address, proof of purchase and delivery date and a brief description of the problem experienced. Micropolis recommends that you insure the package for the full value of the product. The product will be repaired as soon as possible, but, in any event, within thirty (30) days.

This warranty shall be null and void should the product be damaged, subjected to misuse, improper maintenance, negligence, accident or should its serial number or any part thereof be altered, defaced or removed. Further, this warranty will be null and void should the product's design be altered or should repairs be attempted by one not authorized by Micropolis to make repairs.

Micropolis shall not be responsible for any incidental or consequential damages. Some states do not allow the exclusion or limitation of incidental or consequential damages, so this limitation or exclusion may not apply to you.

This warranty limits any implied warranty to ninety (90) days from date of delivery to the original end user. Some states do not allow limitations on how long an implied warranty lasts, so this limitation may not apply to you.

This warranty gives you specific legal rights and you may also have other rights which vary from state to state.

TABLE OF CONTENTS

<u>SECTION I GENERAL INFORMATION</u>	<u>PAGE</u>
1.0 INTRODUCTION	1-1
1.1 IDENTIFICATION PLATE	1-1
1.2 OVERVIEW OF SUBSYSTEMS	1-1
1.2.1 FUNCTION DESCRIPTION	1-1
1.2.2 MODEL VERSIONS	1-2
1.2.3 MEDIA	1-3
1.3 PHYSICAL DESCRIPTION AND DIMENSIONS	1-8
1.3.1 1053/1033 DUAL DISK DRIVE MODULE	1-8
1.3.2 1043/1023 AND 1042/1022 SINGLE DISK DRIVE MODULE	1-8
1.3.3 1041/1021 SINGLE DISK DRIVE MODULE WITHOUT POWER SUPPLY	1-8
1.3.4 1071 CONTROLLER	1-8
1.3.5 INTERFACE CABLES	1-9
1.4 SPECIFICATIONS	1-9
1.4.1 DRIVE PERFORMANCE	1-9
1.4.2 ENVIRONMENTAL	1-10
1.4.3 DRIVE RELIABILITY	1-10
1.5 SUMMARY OF MICROPOLIS PROGRAM DEVELOPMENT SOFTWARE	1-10
1.5.1 ELEMENTS OF MDOS	1-12
1.5.2 ELEMENTS OF MICROPOLIS DISK EXTENDED BASIC	1-13
<u>SECTION II INSTALLATION</u>	
2.0 INTRODUCTION	2-1
2.1 HARDWARE INSTALLATION	2-1
2.1.1 UNPACKING THE EQUIPMENT	2-1
2.1.2 INITIAL CHECKOUT	2-3
2.1.3 CONTROLLER HARDWARE REQUIREMENTS	2-3
2.1.4 CONTROLLER CONFIGURATION	2-3
2.1.4.1 CHANGING THE CONTROLLER BASE ADDRESS	2-4
2.1.4.2 REJUMPERING FOR 3 MHZ OR 4 MHZ OPERATION	2-4
2.1.5 INSTALLING THE CONTROLLER AND INTERFACE CABLE	2-4
2.1.6 DAISY CHAINING MULTIPLE DISK DRIVES	2-9
2.1.7 APPLYING DC POWER - MODEL 1041/1021 ONLY	2-9
2.1.7.1 REGULATED DC	2-11
2.1.7.2 UNREGULATED DC	2-11
2.1.8 CUSTOM MOUNTING OF THE 1041/1021 DRIVES	2-12
2.1.9 LOADING AND UNLOADING	2-14

	<u>PAGE</u>
2.2 SYSTEM SOFTWARE INSTALLATION	2-15
2.2.1 PROGRAM DEVELOPMENT SOFTWARE MEMORY REQUIREMENTS	2-15
2.2.2 SUPPORTED I/O DEVICES	2-15
2.2.3 LOADING THE PDS MDOS SYSTEM INTO MEMORY FROM THE MASTER DISKETTE	2-15
2.2.4 CONFIGURING THE PDS SYSTEMS FOR YOUR TERMINAL	2-19
2.2.4.1 CONFIGURING A STANDARD TERMINAL	2-19
2.2.4.2 CONFIGURING A MODIFIED STANDARD TERMINAL	2-21
2.2.4.3 NON-STANDARD TERMINAL CONFIGURATION	2-22
2.2.4.3.1 THE CONSOLE I/O TABLE	2-22
2.2.4.3.2 LOGICAL CONSOLE I/O	2-24
2.2.4.3.3 PHYSICAL CONSOLE DEVICE INPUT	2-24
2.2.4.3.4 PHYSICAL CONSOLE DEVICE OUTPUT	2-24
2.2.4.3.5 PHYSICAL CONSOLE DEVICE BREAK CHECK ROUTINE	2-24
2.2.4.3.6 PHYSICAL CONSOLE DEVICE INITIALIZE	2-25
2.2.4.3.7 STARTING YOUR SYSTEM	2-25
2.2.5 SYSTEM PRINTER CONFIGURATION	2-25
2.2.5.1 CONFIGURING THE SUPPLIED PRINTER HANDLER	2-26
2.2.5.2 PRINTER INTERFACE EXAMPLE	2-28
2.2.5.3 CONFIGURING SPECIAL PRINTER HANDLERS	2-31
2.2.5.3.1 THE LIST I/O TABLE	2-31
2.2.5.3.2 LOGICAL LIST I/O	2-31
2.2.5.3.3 PHYSICAL LIST DEVICE OUTPUT	2-31
2.2.5.3.4 PHYSICAL LIST DEVICE ATTENTION ROUTINE	2-32
2.2.5.3.5 PHYSICAL LIST DEVICE INITIALIZE	2-32
2.2.6 CREATING YOUR SYSTEM DISKETTE	2-33
2.2.7 CREATING A BASIC ONLY SYSTEM DISKETTE	2-34
2.2.8 MAKING ADDITIONAL COPIES OF YOUR SYSTEM DISKETTE USING A SINGLE DRIVE	2-35
<u>SECTION III NORMAL OPERATION</u>	
3.0 INTRODUCTION	3-1
3.1 BOOTSTRAP PROCEDURE	3-1
3.2 OPERATING HINTS	3-3
3.3 CONCEPT OF BACKUP	3-4
<u>SECTION IV MICROPOLIS DISKETTE OPERATING SYSTEM</u>	
4.0 INTRODUCTION TO MDOS	4-1
4.1 THE MDOS EXECUTIVE	4-2
4.1.1 ENTERING EXECUTIVE COMMANDS	4-2
4.1.2 EXECUTIVE STATEMENT FORMAT	4-2
4.1.3 CANCELING AN OPERATION	4-3
4.1.4 DISPLAY CONTROL	4-4

	<u>PAGE</u>
4.1.5 EXPLICIT EXECUTIVE COMMANDS	4-4
4.1.5.1 THE COMP COMMAND	4-4
4.1.5.2 THE DUMP COMMAND	4-4
4.1.5.3 THE ENTR COMMAND	4-4
4.1.5.4 THE FILL COMMAND	4-5
4.1.5.5 THE MOVE COMMAND	4-5
4.1.5.6 THE SEAR COMMAND	4-5
4.1.5.7 THE SEARN COMMAND	4-6
4.1.5.8 THE CREATE COMMAND	4-6
4.1.5.9 THE DISP COMMAND	4-6
4.1.5.10 THE FILES COMMAND	4-7
4.1.5.11 THE FREE COMMAND	4-7
4.1.5.12 THE SCRATCH COMMAND	4-7
4.1.5.13 THE LOAD COMMAND	4-8
4.1.5.14 THE SAVE COMMAND	4-8,2
4.1.5.15 THE RENAME COMMAND	4-8,2
4.1.5.16 TYPE COMMAND	4-9
4.1.5.17 THE APP COMMAND	4-9
4.1.5.18 THE ASSIGN COMMAND	4-9
4.1.5.19 THE EXEC COMMAND	4-11
4.1.5.20 THE MATH COMMAND	4-11
4.1.5.21 THE PROMPT COMMAND	4-11
4.1.5.22 THE INIT COMMAND	4-12
4.2 MDOS DISK FILE I/O	4-13
4.2.1 TRACK INDEXED FILE STORAGE	4-13
4.2.2 FILE NAMES	4-13
4.2.3 FILE PROTECTION AND TYPE DEFINITION	4-14
4.2.4 FILE AND RECORD STRUCTURE	4-15
4.2.5 FILE ACCESS METHODS	4-16
4.2.6 COMPATIBILITY BETWEEN MDOS AND BASIC FILES	4-17
4.3 MDOS SHARED SUBROUTINES	4-18
4.3.1 CONSOLE AND PRINTER INPUT/OUTPUT SUBROUTINES	4-18
4.3.1.1 @CIN - CONSOLE INPUT	4-18
4.3.1.2 @COUT - CONSOLE OUTPUT	4-18
4.3.1.3 @CBRK - CONSOLE BREAK CHECK	4-19
4.3.1.4 @CDIN - CONSOLE DEVICE INPUT	4-19
4.3.1.5 @CDOUT - CONSOLE DEVICE OUTPUT	4-19
4.3.1.6 @CDBRK - CONSOLE DEVICE BREAK CHECK	4-19
4.3.1.7 @CDINIT - CONSOLE DEVICE INITIALIZATION	4-19
4.3.1.8 @LOUT - LIST OUTPUT	4-19
4.3.1.9 @LATN - LIST ATTENTION	4-20
4.3.1.10 @LDOUT - LIST DEVICE OUTPUT	4-20
4.3.1.11 @LDATN - LIST DEVICE ATTENTION	4-20
4.3.1.12 @LDINIT - LIST DEVICE INITIALIZATION	4-20
4.3.1.13 @CCRLF - CONSOLE LINE FEED CARRIAGE RETURN	4-20
4.3.1.14 @LCRLF - LIST LINE FEED CARRIAGE RETURN	4-20
4.3.1.15 @ASSIGN - ASSIGN	4-20

	<u>PAGE</u>
4.3.1.16 @CILINE - CONSOLE INPUT LINE	4-21
4.3.1.17 @HEXOUT - HEXADECIMAL OUTPUT	4-21
4.3.1.18 @HEXADDOUT - HEXADECIMAL ADDRESS OUTPUT	4-21
4.3.1.19 @HEXOUTSPC - HEXADECIMAL OUTPUT WITH SPACE	4-21
4.3.1.20 @SPACEOUT - SPACE OUT	4-21
4.3.1.21 @ONLINEOUT - NEW LINE OUTPUT	4-22
4.3.1.22 @LINEOUT - LINE OUTPUT	4-22
 4.3.2 TEXT LINE PARSING SUBROUTINES	4-22
4.3.2.1 @PARAM - PARAMETER	4-22
4.3.2.2 @SKIPSPACE - SKIP SPACES	4-23
4.3.2.3 @SCAN - SCAN	4-23
4.3.2.4 @SEAR - SEARCH	4-23
4.3.2.5 @AHEXTBIN - ASCII HEX TO BINARY	4-24
 4.3.3 THE FILE ACCESS ROUTINES	4-24
4.3.3.1 @CREATE - CREATE	4-26
4.3.3.2 @FILESTAT - GET FILE STATUS	4-26
4.3.3.3 @DIRSEARCH - DIRECTORY SEARCH	4-27
4.3.3.4 @OPENFILE - OPEN A FILE	4-27
4.3.3.5 @CLOSEFILE - CLOSE A FILE	4-27
4.3.3.6 @FILEINF - READ FILE INFORMATION	4-27
4.3.3.7 @SINXTRS - SET INDEX POSITION TO RECORD START	4-28
4.3.3.8 @RECORDLEN - READ RECORD LENGTH	4-28
4.3.3.9 @RINXPOS - READ INDEX POSITION	4-28
4.3.3.10 @SINXPOS - SET INDEX POSITION	4-29
4.3.3.11 @INCINX - INCREMENT INDEX POSITION	4-29
4.3.3.12 @RFINXPOS - READ FROM INDEX POSITION	4-29
4.3.3.13 @RFINXPOSI - READ FROM INDEX POSITION AND INCREMENT INDEX	4-30
4.3.3.14 @WTINXPOS - WRITE TO INDEX POSITION	4-30
4.3.3.15 @WTINXPOSI - WRITE TO INDEX POSITION AND INCREMENT INDEX	4-30
4.3.3.16 @LOADDATA - LOAD DATA	4-31
4.3.3.17 @SAVEDATA - SAVE DATA	4-31
4.3.3.18 @DFINXPOSTEOR - DELETE FROM INDEX POSITION TO END OF RECORD	4-31.1
4.3.3.19 @DFINXPOS - DELETE FROM INDEX POSITION TO END OF FILE	4-32
4.3.3.20 @INCRECPoS - INCREMENT RECORD POSITION	4-32
 4.3.4 FILE MANAGEMENT SUBROUTINES	4-32
4.3.4.1 @FREE - FREE	4-32
4.3.4.2 @RENAME - RENAME	4-32
4.3.4.3 @TYPE - FILE TYPE	4-33
4.3.4.4 @SCRATCH - SCRATCH A FILE	4-33

	<u>PAGE</u>
4.3.5 PHYSICAL DISK ACCESS ROUTINES	4-33
4.3.5.1 @GETASEC - GET A SECTOR	4-34
4.3.5.2 @PUTASEC - PUT A SECTOR	4-34
4.3.5.3 @WRITESECTOR - WRITE A SECTOR	4-35
4.3.5.4 @VERIFYSECTOR - VERIFY A SECTOR	4-35
4.3.5.5 @SEEKTRACK - SEEK TO A TRACK	4-35
4.3.5.6 @RESTOREDISK - RESTORE THE READ/WRITE HEAD	4-35
4.3.6 PROCESSOR ORIENTED UTILITY ROUTINES	4-36
4.3.6.1 @HLADDA - ADD A TO HL	4-36
4.3.6.2 @INXM - INCREMENT MEMORY	4-36
4.3.6.3 @LHLINDEXED - LOAD HL INDIRECT INDEXED	4-36
4.3.6.4 @LHLI - LOAD HL INDIRECT	4-37
4.3.6.5 @TRANSDHC - TRANSFER FROM DE TO HL FOR A COUNT OF C	4-37
4.3.6.6 @TRANSDHBC - TRANSFER FROM DE TO HL FOR A COUNT OF BC	4-37
4.3.6.7 @TRANSDHBCR - TRANSFER FROM DE TO HL FOR A COUNT OF BC REVERSE	4-37
4.3.6.8 @TRANSFILENAME - TRANSFER A FILENAME	4-38
4.3.6.9 @FILLZER - FILL ZEROES	4-38
4.3.6.10 @FILLSPC - FILL SPACES	4-38
4.3.6.11 @FILLA - FILL FROM THE A REGISTER	4-38
4.3.6.12 @COMPARE - COMPARE HL TO DE	4-38
4.3.7 EXTENDED 8080 INTEGER ARITHMETIC (16 BITS)	4-39
4.3.7.1 @DEADDHL - BC=DE+HL	4-39
4.3.7.2 @DESUBHL - BC=DE-HL	4-39
4.3.7.3 @DEMULHL - BC=DE*HL	4-39
4.3.7.4 @DEDIVHL - BC=DE/HL	4-40
4.3.7.5 @DEMODHL - BC=DE%HL	4-40
4.3.8 MESSAGE OUTPUT SUBROUTINES	4-40
4.3.8.1 @DISKERROR - DISK ERROR MESSAGES	4-40
4.3.8.2 @CLOSEFILES - CLOSE ALL FILES	4-41
4.3.8.3 @ERRORMES - ERROR MESSAGES	4-41
4.3.8.4 @MESSAGEOUT - MESSAGE OUTPUT	4-41
4.3.9 SYSTEM BUFFERS AND ENTRY POINTS	4-41
4.4 LINEEDIT - THE MDOS LINE EDITOR	4-43
4.4.1 ENTERING LINES TO LINEEDIT	4-43
4.4.2 KEYING IN A NEW TEXT FILE	4-44
4.4.3 ENTERING LINEEDIT COMMANDS	4-44
4.4.4 THE CLEAR COMMAND	4-45
4.4.5 THE NAME COMMAND	4-45
4.4.6 THE FILE COMMAND	4-45
4.4.7 THE AUTO COMMAND	4-45

	<u>PAGE</u>
4.4.8 THE PROMPT COMMAND	4-46
4.4.9 THE LOAD COMMAND	4-46
4.4.10 THE APPEND COMMAND	4-46
4.4.11 THE SAVE COMMAND	4-47
4.4.12 THE RESAVE COMMAND	4-47
4.4.13 THE LIST COMMAND	4-48
4.4.14 THE LISTP COMMAND	4-48
4.4.15 THE PRINT COMMAND	4-49
4.4.16 THE PRINTP COMMAND	4-49
4.4.17 THE TAB COMMAND	4-49
4.4.18 THE DELT COMMAND	4-49
4.4.19 THE RENUM COMMAND	4-49
4.4.20 THE SEARCH COMMAND	4-50
4.4.21 THE SEARCHALL COMMAND	4-50
4.4.22 THE CHANGE COMMAND	4-51
4.4.23 THE CHANGEALL COMMAND	4-52
4.4.24 THE EDIT COMMAND	4-52
4.4.24.1 ADVANCING THE EDIT POINTER	4-52.1
4.4.24.2 CHANGING THE NEXT CHARACTER - C	4-52.1
4.4.24.3 DELETING THE NEXT CHARACTER - D	4-52.1
4.4.24.4 INSERTING CHARACTERS - I	4-52.1
4.4.24.5 LISTING THE LINE IN THE EDIT BUFFER - L	4-52.1
4.4.24.6 SEARCHING TO A SPECIFIED CHARACTER - S	4-53
4.4.24.7 DELETING TO A SPECIFIED CHARACTER - K	4-53
4.4.24.8 QUITTING THE EDIT COMMAND MODE - Q	4-53
4.4.24.9 COMPLETING THE EDIT COMMAND	4-53
4.4.25 THE DOS COMMAND - EXITING FROM LINEEDIT	4-53
4.4.26 LINEEDIT FILE STRUCTURE	4-54
 4.5 ASSM - THE MICROPOLIS 8080/8085 DISK ASSEMBLER	4-55
4.5.1 HOW TO INVOKE ASSM	4-55
4.5.2 LANGUAGE ELEMENTS	4-57
4.5.2.1 LITERALS	4-58
4.5.2.2 SYMBOLIC NAMES	4-58
4.5.2.3 OPERATORS	4-59
4.5.2.4 OPCODE MNEMONICS	4-60
4.5.3 OPERANDS	4-60
4.5.4 ASSEMBLER DIRECTIVES	4-61
4.5.4.1 ORG - ORIGIN	4-61
4.5.4.2 LINK - LINK TO A FILE	4-61
4.5.4.3 END - END OF ASSEMBLY	4-62.1
4.5.4.4 EQU - EQUATE	4-63
4.5.4.5 INP - INPUT	4-63
4.5.4.6 PRT - PRINT	4-63
4.5.4.7 TAB - TAB SETTINGS	4-64

	<u>PAGE</u>
4.5.4.8 NLIST - NO LIST TO PRINTER	4-64
4.5.4.9 LIST - LIST TO PRINTER	4-64
4.5.4.10 FORM - FORM FEED	4-64
4.5.4.11 DB - DEFINE BYTE	4-64
4.5.4.12 DW - DEFINE WORD	4-65
4.5.4.13 DD - DEFINE DATA	4-65
4.5.4.14 DT - DEFINE TEXT	4-65
4.5.4.15 DTZ - DEFINE TEXT TERMINATED WITH ZERO	4-65
4.5.4.16 DTH - DEFINE TEXT TERMINATED WITH BIT 8 HIGH	4-65
4.5.4.17 DS - DEFINE STORAGE	4-66
4.5.4.18 FILL - FILL STORAGE	4-66
4.5.4.19 IFF - IF FALSE	4-66
4.5.4.20 IFT - IFT TRUE	4-66
4.5.4.21 ENDIF - END OF IFF	4-66
 4.5.5 ASSEMBLER ERRORS	 4-67
4.6 SYMSAVE UTILITY	4-68
4.7 FILECOPY UTILITY	4-69
4.8 DISKCOPY UTILITY	4-69
4.9 MDOS ERROR MESSAGES	4-71
4.10 COPYFILE UTILITY FOR SINGLE DISK	4-74
4.11 MICROPOLIS DEBUG	4-75
4.12 DEBUG-GEN UTILITY	4-92
 <u>SECTION V MICROPOLIS DISK EXTENDED BASIC</u>	
5.0 INTRODUCTION	5-1
5.1 ENTERING LINES TO THE BASIC INTERPRETER	5-1
5.2 ENTERING A PROGRAM	5-2
5.3 IMMEDIATELY EXECUTED LINES	5-3
 5.3.1 THE EDIT COMMAND	 5-3
5.3.2 THE RENUM COMMAND	5-4.1
5.3.3 THE MERGE COMMAND	5-4.3
 5.4 DELETE COMMAND	 5-3
5.5 LIST COMMAND	5-4
5.6 SAVE COMMAND	5-4
5.7 LOAD COMMAND	5-5
5.8 DISPLAY COMMAND	5-5
5.9 SCRATCH COMMAND	5-6
5.10 RUN COMMAND	5-6
5.11 INTERRUPTING A RUNNING PROGRAM	5-7
5.12 CONTINUING AN INTERRUPTED PROGRAM	5-7
5.13 PROGRAM TRACING COMMANDS	5-8
5.14 BASIC SYSTEM ERROR HANDLING	5-8
5.15 BASIC CHARACTER SET	5-9
5.16 DATA	5-9
 5.16.1 CONSTANTS	 5-9
5.16.2 VARIABLES	5-10
5.16.3 OUTPUT FORMATS	5-12

	<u>PAGE</u>
5.17 OPERATORS	5-14
5.17.1 NUMERIC OPERATORS	5-14
5.17.2 STRING OPERATORS	5-14
5.17.3 RELATIONAL OPERATORS	5-15
5.17.4 LOGICAL OPERATORS	5-16
5.18 FUNCTIONS	5-17
5.18.1 INTRINSIC FUNCTIONS	5-17
5.18.1.1 NUMERIC FUNCTIONS	
ABS	5-18
ATN	5-18
COS	5-18
EXP	5-18
FIX	5-18
FRAC	5-18
INT	5-18
LN	5-18
LOG	5-18
MAX	5-18
MIN	5-18
MOD	5-18
RND	5-19
SGN	5-19
SIN	5-19
SQR	5-19
TAN	5-19
5.18.1.2 STRING FUNCTIONS	
ASC	5-20
CHAR\$	5-20
FMT	5-20
INDEX	5-21
LEFT\$	5-21
LEN	5-21
MID\$	5-21
MAX	5-21
MIN	5-21
REPEAT\$	5-21
RIGHT\$	5-21
STR\$	5-21
VAL	5-21
VERIFY	5-21
5.18.1.3 SPECIAL FUNCTIONS	
IN	5-22
PEEK	5-22
PGMSIZE	5-22
SPACELEFT	5-22
5.18.2 USER DEFINED FUNCTIONS	5-22

5.19 Expressions	5-33
5.19.1 Evaluation of Expressions	5-33
5.19.2 Numeric Expressions	5-33
5.19.3 String Expressions	5-34
5.19.4 Logical Expressions	5-35
 5.20 BASIC Statements	 5-36
5.20.1 DATA	5-36
5.20.2 DEF FN	5-37
5.20.3 DEF FA	5-37
5.20.4 DIM	5-38
5.20.5 END	5-38
5.20.6 EXEC	5-39
5.20.7 FLOW	5-39
5.20.8 FOR	5-40
5.20.9 GOSUB	5-42
5.20.10 GOTO	5-43
5.20.11 IF..THEN	5-43
5.20.12 INPUT	5-44
5.20.13 LET	5-44
5.20.14 MEMEND	5-45
5.20.15 NEXT	5-45
5.20.16 NOFLOW	5-45
5.20.17 ON..GOTO	5-45
5.20.18 ON..GOSUB	5-46
5.20.19 OUT	5-46
5.20.20 POKE	5-46
5.20.21 PRINT	5-47
5.20.22 READ	5-49
5.20.23 REM	5-49
5.20.24 RESTORE	5-49
5.20.25 RETURN	5-49
5.20.26 SIZES	5-50
5.20.27 STOP	5-50
5.20.28 STRING	5-50
 5.21 BASIC DISK FILE I/O	 5-51
5.21.1 Disk Files	5-51
5.21.2 Disk File Commands	5-52
5.21.2.1 DISPLAY	5-53
5.21.2.2 LOAD	5-53
5.21.2.3 PLOADG	5-53
5.21.2.4 SAVE	5-54
5.21.2.5 SCRATCH	5-54.1
5.21.2.6 CHAIN	5-54.1
5.21.2.7 LINK	5-54.1

5.21.3 Disk I/O Statements	5-54,1
5.21.3.1 OPEN	5-55
5.21.3.2 PUT	5-57
5.21.3.3 GET	5-60
5.21.3.4 CLOSE	5-60
5.21.3.5 ATTRS	5-61
5.21.3.6 EOF	5-61
5.21.3.7 FREESPACE	5-62
5.21.3.8 GETSEEK	5-62
5.21.3.9 PUTSEEK	5-62
5.21.3.10 RENAME	5-63
5.21.4 Disk I/O Functions	5-63
ATTR	5-64
ERR	5-64
ERR\$	5-64
NAME	5-64
RECGET	5-64
RECPUT	5-64
SIZE	5-64
TRACKS	5-64
FREETR	5-64
5.22 BASIC PRINT FILE OUTPUT	5-65
5.22.1 Printer Related Language Features	5-65
5.22.1.1 OPEN	5-65
5.22.1.2 PUT	5-66
5.22.1.3 CLOSE	5-66
5.22.1.4 ENDPAGE	5-67
5.22.1.5 ASSIGN	5-67
5.22.1.6 LISTP	5-69
5.22.1.7 PAGESIZE	5-69
5.22.2 Notes on Printer Related Programming	5-70
5.22.2.1 Separating Print Files and Interactive Messages	5-70
5.22.2.2 Paginating Print Files	5-73
5.22.2.3 Spooling Print Files to Disk for Later Output	5-76
5.22.2.4 Draining File Output to A Null Device	5-76
5.22.2.5 Echoing of Terminal Output to Printer	5-77

<u>SECTION VI DISK SUBSYSTEM THEORY AND DIRECT PROGRAMMING</u>	<u>PAGE</u>
6.0 INTRODUCTION	6-3
6.1 FUNDAMENTALS OF THE FLEXIBLE DISK: MEDIA	6-3
6.2 HARDWARE FUNDAMENTALS	6-7
6.3 CONTROLLER REGISTERS	6-9
6.4 DISK OPERATIONS	6-13
6.5 ERROR HANDLING	6-20
6.6 DISK DRIVER	6-21
APPENDIX A - BASIC ERROR MESSAGES	A-1
APPENDIX B - BASIC UTILITY PROGRAM	B-1
APPENDIX C - ACCESSING DISKCOPY FROM BASIC	C-1
APPENDIX D - SUMMARY OF MDOS ERROR MESSAGES	D-1
APPENDIX E - SYSTEM I/O LISTINGS	E-1
APPENDIX F - MICROPOLIS DISK BOOTSTRAP	F-1
APPENDIX G - FEATURES PROGRAM TO SHORTEN BASIC	G-1

I GENERAL INFORMATION AND SPECIFICATIONS

1.0 INTRODUCTION

This section provides general information, and specifications of Micropolis Floppy Disk storage subsystems Model Numbers 1021 through 1053.

1.1 IDENTIFICATION PLATE

An identification plate is located on the base chassis (bottom of unit). It shows model number, serial number, line voltage and fuse rating. Both model number and serial number should always be quoted in warranty correspondence.

WARNING - When replacing the fuse always use a fuse of the same type and rating. These are:

OPERATING VOLTAGE RANGE VAC RMS	FUSE TYPE	AMPERE RATING	VOLTAGE RATING	LITTELFUSE PART NUMBER	MICROPOLIS PART NUMBER
100 to 125	3AG SLO-BLO	1.0	250	313001	626-0002-8
200 to 240	3AG SLO-BLO	0.5	250	313.500	626-0001-0

WARNING - This equipment is provided with a 3 pin power plug. The plug must be inserted in a 3 pin receptacle with the third pin connected to earth ground.

1.2 OVERVIEW OF SUBSYSTEMS

1.2.1 FUNCTION DESCRIPTION

Each subsystem comprises:

- ° A storage module consisting of an enclosure, drive electronics and one (or two disk drives) depending on model. Power supplies may or may not be provided depending on model number.
- ° A single printed circuit board designed to be physically and electrically compatible with S-100 bus and 8080/Z80 based micro-computers.
- ° A software package together with documentation is provided, which allows the subsystem to be used effectively by both end users and system designers having varying levels of experience.

The subsystems are fully automatic and require no operator intervention during normal operation. Applications include random access mass data storage, data entry, data output and program storage.

1.2.2 MODEL VERSIONS

Each model number is followed by either the notation Mod I or Mod II. These notations indicate whether the system operates at a track density of 48 TPI (35 tracks total) or 100 TPI (77 tracks total). Mod I storage modules have a black disk load actuator and Mod II modules have a blue disk load actuator.

The models described in this document are:

* 1053 Mod II: Complete Metafloppytm dual-disk subsystem with a total of 630 kilobytes of formatted on-line storage. Includes two disk drives, S-100/8080/Z-80 compatible controller (Model 1071), drive enclosure, interface cable A and power supply.

1053 Mod I: Complete Macrofloppytm dual-disk subsystem similar to the 1053 Mod II except with a total of 287 kilobytes of formatted on-line storage.

1043 Mod II: Complete Metafloppytm single-disk subsystem with a total of 315 kilobytes of formatted on-line storage. Includes one disk drive, S-100/8080/Z80 compatible controller (Model 1071), drive enclosure, interface cable A and power supply.

1042 Mod I: Complete Macrofloppytm single-disk subsystem similar to 1043 Mod II except with a total of 143 kilobytes of formatted on-line storage.

1041 Mod II: Complete Metafloppytm single-disk subsystem with 315 kilobytes of on-line storage. Similar to 1043 except drive is enclosed in a protective sleeve which does not include a power supply or regulator/heat sink package, but includes a power cable A for connection to an external regulated power supply. The 1041 Mod II can be used in a desk-top mode, or the rubber feet can be removed and the unit mounted in the customer's chassis.

1041 Mod I: Complete Macrofloppytm single-disk subsystem with a total of 143 kilobytes of formatted on-line storage. Similar to 1041 Mod II except for storage capacity.

- 1033 Mod II: Add-on dual-disk storage module with a total of 630 kilobytes of formatted on-line storage. Includes two disk-drives, drive enclosure and power supply. Attaches to a 1053 Mod II using a Daisy Chain cable.
- 1033 Mod I: Add-on dual-disk storage module similar to the 1033 Mod II except with a total of 287 kilobytes of formatted on-line storage. Attaches to a 1053 Mod I using a Daisy Chain cable.
- 1023 Mod II: Add-on single disk storage module with a total of 315 kilobytes of formatted on-line storage. Includes one disk drive, drive enclosure and power supply. Attaches to a 1043/1053 Mod II using a Daisy Chain cable.
- 1022 Mod I: Add-on single disk storage module similar to the 1023 Mod II except with a total of 287 kilobytes of formatted on-line storage. Attaches to a 1043/1053 Mod I using a Daisy Chain cable.
- 1021 Mod II: Add-on single disk storage module with a total of 315 kilobytes of formatted on-line storage. Similar to 1023 Mod II except drive is enclosed in a protective sleeve which does not include a power supply or regulator/heat sink package. Attaches to a 1043/1053 Mod II using a Daisy Chain cable and includes a power cable for connection to an external regulated power supply.
- 1021 Mod I: Add-on single disk storage module similar to the 1021 Mod II except with a formatted on-line storage of 143 kilobytes. Attaches to a 1043/1053 Mod I using a Daisy Chain cable.
- 1091-01: Regulator kit. Includes heat sink, regulator IC's, cables and mounting hardware. Provides regulators to convert S-100 bus unregulated voltages to the voltages required to operate 1041/1021 drives and includes a power cable B for connection to S-100 bus unregulated voltages via a socket on the subsystem controller.

1.2.3 MEDIA (DISKETTES)

The recording medium used with Micropolis storage subsystems is an industry-standard 5 1/4-inch diskette (Figure 1.1) in its hard-sectored version with 16 sectors, each defined by a sector hole. Thus, it has one index hole and 16 sector holes. Diskettes of this type are available from Micropolis or from many local sources, such as computer stores.

NOTE: Do NOT use diskettes with other than 16 hard sectors, or those which are soft-sectored (no sector holes). They will not work.

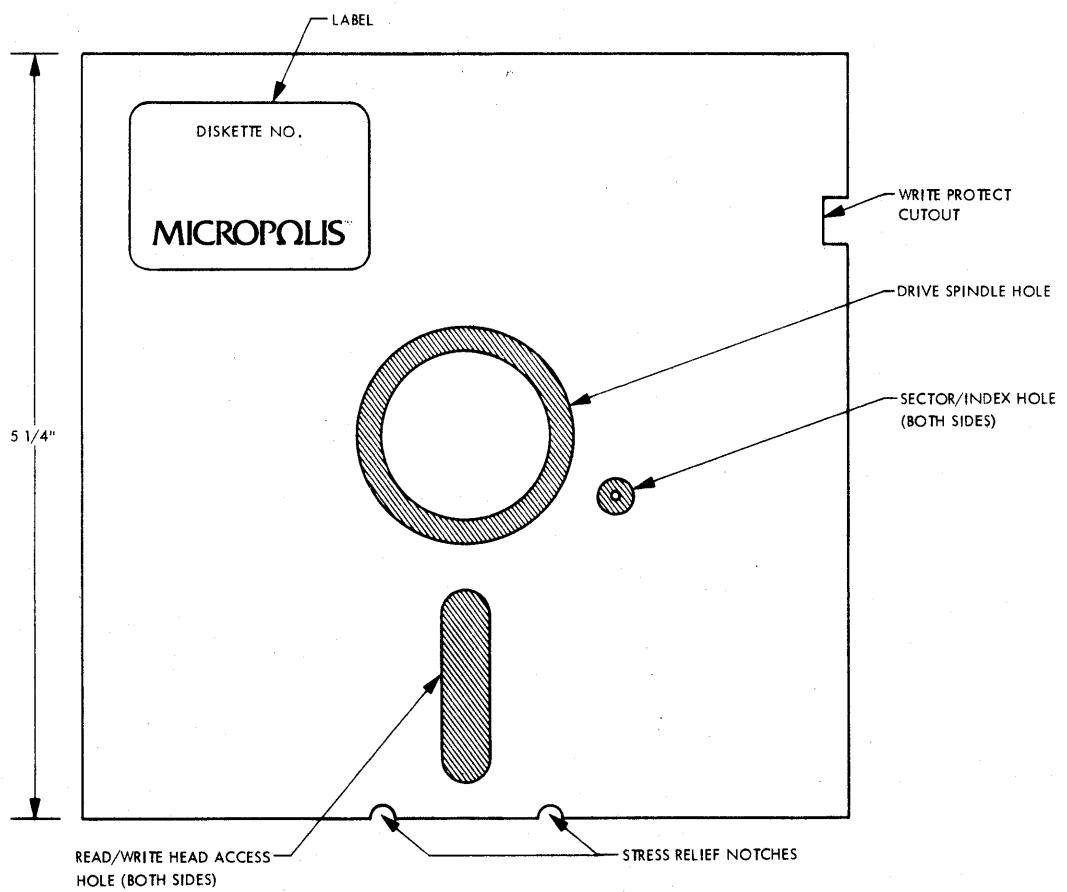


Figure 1.1 5 1/4 inch Diskette

New diskettes must be initialized (formatted) before being used for the first time. See Appendix B for the initialization procedure.

The sub-systems are equipped with a File Protect (Write Protect) feature which protects a suitably treated diskette from inadvertent erasure or over-writing of important files. File Protect tabs are provided with each package of diskettes from Micropolis. Installation of these File Protect tabs is shown in Figure 1.2.

The nature of in-contact recording as used in magnetic tape and floppy disk drives requires that the medium be replaced from time to time. The intervals naturally depend on the kind of usage. Continual loading of the head on a single track will naturally result in its deterioration before that of the remainder of the diskette. Your diskette is protected as far as possible by the smooth characteristics of the Micropolis ceramic head and by the automatic head unload feature which raises the head load pad from the surface of the diskette if no activity has occurred for 5 seconds.

When a diskette is loaded--that is, when a diskette is inserted and the manual load actuator is depressed--it begins and continues to rotate inside the jacket. The user can extend the life of a diskette by unloading the actuator during periods in which the disk is not in use; this raises the head load pad and discontinues rotation.

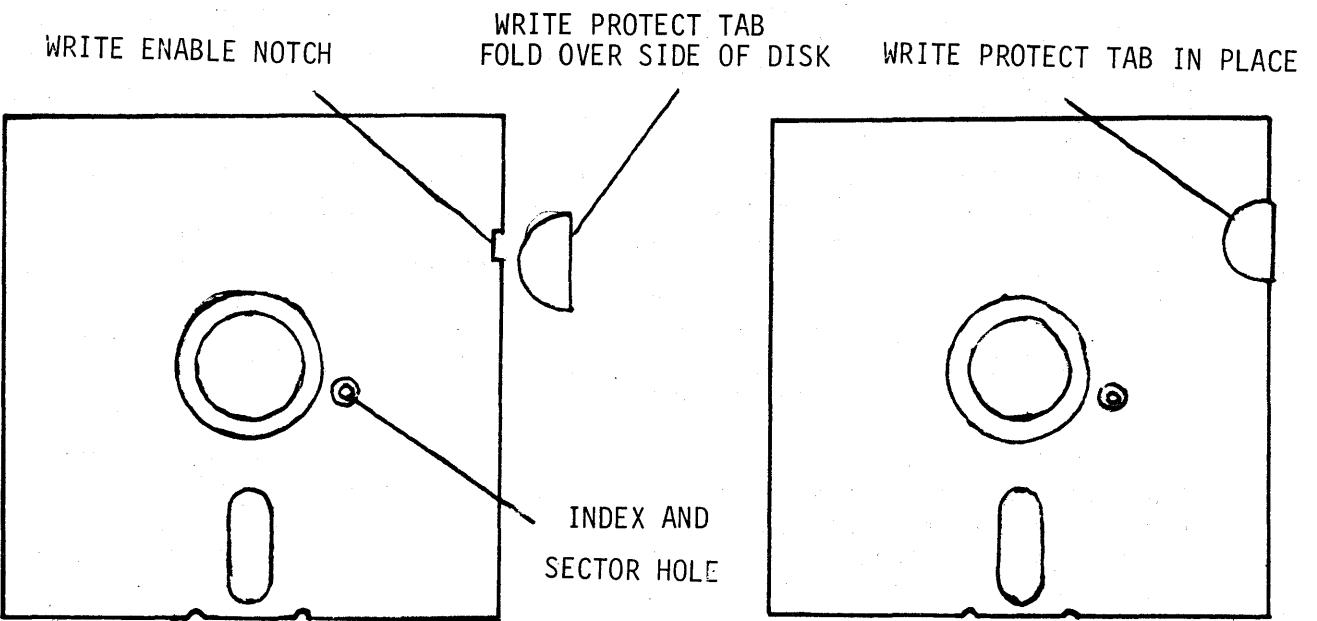


Figure 1.2 How To Mount Write Protect Tab

NOTE: Micropolis 1021 through 1053 Series Systems use standard 5 1/4-inch diskettes with 16 hard sectors ONLY !! Do NOT attempt to use diskettes with other, non-standard number of sectors or diskettes which are soft-sectored. They will not work.

CAUTION: The diskette must be treated with care to ensure good reliability. Figure 1.3 summarizes the DO's and DON'Ts.

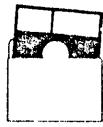
	Protect Proteger Proteger 保護	Proteger Schutzen
	No Non 注意	No Falsch 注意
	Insert Carefully Inserir com cuidado Inserer avec soin Sorgfältig Einsetzen 插入 注意	Insertar Sorgfältig Einsetzen 插入 注意
	Never Jamais 絶対禁止	Nunca Nie 絶対禁止
	10 C - 52 C 50 F - 125 F	
	Never Jamais 絶対禁止	Nunca Nie 絶対禁止

Figure 1.3

1.3 PHYSICAL DESCRIPTION AND DIMENSIONS

1.3.1 1053/1033 DUAL DISK DRIVE MODULE

Height	8.0"	20.3 cm.
Width	9.2"	23.4 cm.
Depth	13.0"	33.0 cm.
Weight	18 lbs.	8.2 Kg.

Input Power requirements: 115/230 VAC, 50/60 Hz.
Standby 60 VA; Operating 78 VA.

1.3.2 1043/1023 and 1042/1022 SINGLE DISK DRIVE MODULE

Height	4.0"	10.2 cm.
Width	5.9"	15.0 cm.
Depth	12.2"	31.0 cm.
Weight	9.0 lbs.	4.1 Kg.

Input Power requirements: 115/230 VAC, 50/60 Hz.
Standby 30 VA; Operating 45 VA.

1.3.3 1041/1021 SINGLE DISK DRIVE MODULE (WITHOUT POWER SUPPLY)

Height	4.0"	10.2 cm.
Width	5.9"	15.0 cm.
Depth	9.6"	24.3 cm.
Weight	5.0 lbs.	2.3 Kg.

Input Power requirements: +5V ±5% regulated .5A
+12V ±5% regulated 1.15A

1.3.4 1071 CONTROLLER

The controller is a single printed circuit board, physically and electrically compatible with S-100 bus and 8080/Z80 microcomputers.

Height (not including the edge connector to the motherboard):

5.0"	12.7 cm.	
Width	10.0"	25.4 cm.

The edge connector for the interface cable is recessed to keep the overall height at 5.0" when the cable is connected.

1.3.5 INTERFACE CABLES

The standard Interface Cable A (1083-01) is 54" (137 cm.) long. It uses 34-wire flat cable with card edge connectors at each end. Pin 1 is indicated by a contrasting wire color along the appropriate edge. This cable is used to connect the controller directly to any single storage module (which can in turn contain one or two disk drives). When two or more storage modules are to be connected to the controller, the appropriate Daisy Chain cable must be used in place of the standard Cable A.

Daisy Chain Type	Model	Total Connectors	Total Storage Modules
B	1083-02	3	2
C	1083-03	4	3
D	1083-04	5	4

The maximum number of storage modules that can be daisy chained to a single controller is four. This can be any combination of single and dual modules with the limitation that the total number of drives that can be daisy chained is four.

1.4 SPECIFICATIONS

1.4.1 DRIVE PERFORMANCE

- * Capacity per drive, Mod II: 315K bytes, formatted
Mod I : 143K bytes, formatted
- * Transfer rate: 250K bits/second
- * Average rotational latency time: 100 milliseconds (ms)
- * Access time - track-to-track: 30 ms
settling time: 10 ms
- * Head load time: 75 ms
- * Head positioner: stepper motor with lead-screw drive
- * Drive motor start time: 1 second
- * Rotational speed: 300 RPM
- * Recording density: 5248 bits per inch (BPI) Mod II
5162 bits per inch (BPI) Mod I
- * Recording mode: MFM
- * Track density, Mod II: 100 tracks per inch (TPI)
Mod I : 48 tracks per inch (TPI)
- * Surfaces used per diskette: 1

1.4.2 ENVIRONMENTAL

Operating temperature: 50°-104°F, 10°-40°C

Relative Humidity: 20%-80% (without condensation)

1.4.3 DRIVE RELIABILITY

MTBF	8000 hrs.
MTTR	0.5 hrs.
Media Life	3×10^6 passes on single track
Head Life	10,000 hrs.
Soft Error Rate	1 in 10^9
Hard Error Rate	1 in 10^{12}
Seek Error Rate	1 in 10^6

1.5 SUMMARY OF MICROPOLIS PROGRAM DEVELOPMENT SOFTWARE

Micropolis Program Development Software (PDS) consists of two systems:

- 1) The Micropolis Diskette Operating System (MDOS)
- 2) Micropolis Disk Extended BASIC

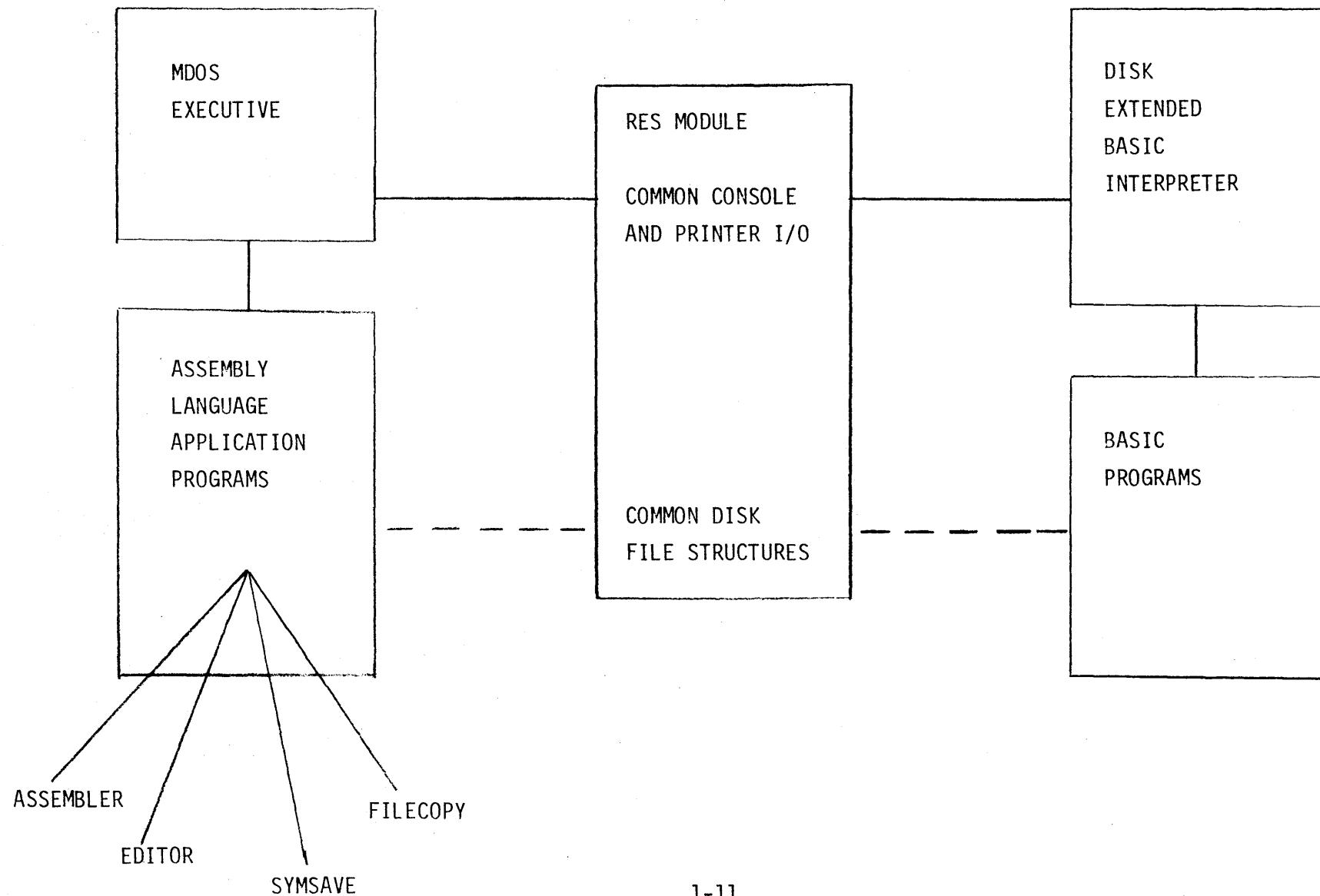
Both PDS systems are included on the PDS MASTER diskette that goes with each Micropolis disk subsystem. Figure 1.4 pictures the relationship between the two PDS systems.

A Program Development Software system is a group of programs that aid the programmer in developing, maintaining, and executing application programs. MDOS and BASIC provide this aid for assembly language programs and BASIC programs, respectively. They are both written in the instruction set of the 8080 microcomputer. They can be run on 8080/8085/Z80 micro-computer systems that utilize the S-100 bus and a Micropolis disk subsystem as the primary file device.

MDOS and BASIC share a common program module called RES. This module contains the system console, system printer, and diskette I/O routines. These routines are always resident in the computer system memory when either MDOS or BASIC is running.

As a consequence of the shared RES module both MDOS and BASIC offer the same console and printer I/O support capabilities and it is only necessary to configure (personalize) the RES module one time for the hardware I/O interfaces of a particular system. Additionally, both MDOS and BASIC utilize the same diskette organization and file structure so that files created under MDOS and files created under BASIC can each be processed by either system. In particular, BASIC can access assembly language functions created by the MDOS assembler provided that the functions meet BASIC's memory requirements and DO NOT call MDOS subroutines; and application programs can be written in assembly language to run under MDOS and process data files created by BASIC.

FIGURE 1.4 MICROPOLIS PROGRAM DEVELOPMENT SOFTWARE (PDS) SYSTEMS



All parts of the MDOS and BASIC systems other than the RES module are completely separate. MDOS consists of the RES module, the MDOS module, and the applications program area which extends into high memory. BASIC consists of the RES module, the BASIC interpreter module, and the BASIC program buffer which extends into high memory. Memory maps of the MDOS and BASIC systems are shown in Chapter II, Figures 2.6 and 2.7.

Also provided is a BASIC UTILITY program that provides for formatting a disk and examining and changing memory.

Control of the computer system is easily transferred from the MDOS system to the BASIC system and vice versa. The MDOS executive responds to the command BASIC. It reads the BASIC interpreter from a specified disk unit, loads it into memory after the RES module and transfers control to BASIC. The BASIC command interpreter responds to the command LINK "MDOS". It reads the MDOS module from a specified disk unit, loads it into memory after the RES module and transfers control to the MDOS executive.

1.5.1 ELEMENTS OF MDOS

The Micropolis Diskette Operating System (MDOS) consists of an executive program, a group of shared subroutines available to user programs, and an assembly language program development package.

The MDOS executive program implements an interactive command language that allows the user to control computer system operations from the system console. It provides commands for memory management, file management, I/O control and program control.

MDOS contains a very large group of subroutines which can be called from a user's application program. These subroutines provide for console and printer character I/O, buffered line I/O, text line parameter parsing, sequential and random file access, file management, physical diskette access, and 16 bit integer arithmetic. There are also a number of processor oriented utility subroutines.

The MDOS application programs that are supplied by Micropolis to support assembly language program development include:

ASSM - a two pass, 8080/8085, disk to disk assembler program.

LINEEDIT - a line number oriented assembly language text editor with character within line editing and global search and change capabilities.

FILECOPY - a utility that copies disk files.

DISKCOPY - a utility that makes a binary copy of an entire diskette.

SYMSAVE - a utility that creates a source file of symbol equate statements from the symbol table left in memory immediately after an assembly.

DEBUG - a utility that facilitates checkout and debugging of 8080/8085 machine language programs.

1.5.2 ELEMENTS OF MICROPOLIS DISK EXTENDED BASIC

Micropolis Disk Extended BASIC is a complete, self-contained software package that provides total support for BASIC programming. When BASIC is loaded you have at hand a powerful set of tools for developing, testing, executing and maintaining BASIC programs.

Program lines may be as long as 250 characters in length and may include multiple statements. The maximum line number is 65529.

BASIC has 12 immediate mode commands, including: SAVE a file, LOAD a file, DISPLAY the file directory, SCRATCH a file, LIST a program, DELETE lines from a program, RUN a program, CNTL/C to interrupt a running program, CONT to continue an interrupted program, CNTL/X to cancel an input line, FLOW and NOFLOW to enable and disable the flow trace debugging aid.

BASIC supports 6 distinct data types, including integers, integer arrays, floating point numbers in the range 1E-61 to 1E62-1, string arrays, floating point arrays, and character strings up to 250 characters long. Integer and floating point arrays may have up to 4 dimensions. String arrays may have up to 3 dimensions plus a length parameter.

A unique SIZES statement enables you to select the precision of numeric variables up to 60 digits for simple arithmetic and 20 digits for transcendental functions. The system defaults to 8 digits for real numbers and 6 for integers.

BASIC supports numeric operators for addition, subtraction, multiplication, division, integer division, and exponentiation. There are relational operators to compare numbers or strings and the logical operators AND, OR, and NOT. String concatenation is also available.

Numeric functions include ABS, ATN, COS, EXP, FIX, FRAC, INT, LN, LOG, MAX, MIN, MOD, RND, SGN, SQR, and TAN.

String functions include ASC, CHAR\$, FMT, INDEX, LEFT\$, LEN, MID\$, MAX, MIN, REPEAT\$, RIGHT\$, STR\$, VAL, VERIFY.

The unique FMT(X,Y\$) function is the key to a powerful formatted output capability. It returns a string which is the value of X formatted per the image defined by format string Y\$.

The DEF FN statement is provided to allow construction of user defined functions. An assembly language function may be linked to using the DEF FA construction.

Standard statements in BASIC include CHAIN, DATA, DEF, DIM, EDIT, END, EXEC, FOR-NEXT-STEP, GOSUB, GOTO, IF-THEN, INPUT, LET, LINK, MEMEND, MERGE, NOFLOW, FLOW, ON-GOTO, ON-GOSUB, OUT, PLOADG, POKE, PRINT, READ, REM, RENUM, RESTORE, RETURN, SIZES, STOP, and STRING.

The CHAIN is a true chain that passes variables from the current program segment to next one loaded from disk.

EXEC is a unique statement that allows a string variable or constant to be executed as if it were a predefined program line.

Data file programming in Micropolis Disk Extended BASIC is simple. Files can be opened simultaneously for both sequential and direct (random) access in both read and write modes. Up to 10 files can be open at one time. A CLEAR option allows a file to be opened for rewrite instead of append. An END option provides an on-endfile-goto capability. An ERROR option provides an on-error-goto capability.

Data is written to and read from files using GET and PUT statements with variable lists that allow a mixture of numeric and string variables.

Files must be CLOSEd after use.

The file I/O structure also extends to printer and console output files to afford a high degree of device independence. Additional options on the OPEN statement facilitate the pagination of output reports.

II INSTALLATION

2.0 INTRODUCTION

This chapter describes how to install your Micropolis disk subsystem hardware in a compatible computer system and how to configure the Micropolis system software for that computer. The computer must be an S-100 bus system using an 8080, 8085, or Z80 processor. A keyboard display console device is required. Figure 2.1 illustrates a typical installation.

2.1 HARDWARE INSTALLATION

The disk subsystem hardware consists of 1 to 4 disk storage modules, an associated interface cable and a controller printed circuit board. Installing the subsystem is accomplished by unpacking and visually inspecting the equipment; configuring the controller as necessary for your particular computer system; installing the controller in the S-100 bus and connecting the storage modules to the controller. A diskette may then be loaded into the disk drive. Hardware installation must be complete before system software configuration can begin.

2.1.1 UNPACKING THE EQUIPMENT

The sub-systems are shipped in a protective container which meets the National Safe Transit Specification (Project 1A, Category 1). The container is designed to minimize the possibility of damage during shipment.

The following procedure describes the recommended method for unpacking the elements of the sub-system.

- 1) Place the shipping container on a flat work surface.
- 2) Cut the sealing tape on the container top carefully; open the top flaps.
- 3) Remove the User's Manual shipping box (12" X 12" X 2") and the controller box (12" X 6" X 1") and set aside.
- 4) For a Dual Disk Module shipment, slide the Disk Module still supported by the 3" foam end pieces carefully out of the container. It will be necessary for the container to be held while this takes place.

For a Single Disk Module shipment, remove the module box from the outer shipping container. Cut open the tape sealing the top of the box, open the top flaps and carefully remove the Disk Module. For 1041/1021 Modules, the interface cable, power cable and optional regulator kit will be packed in the module box.

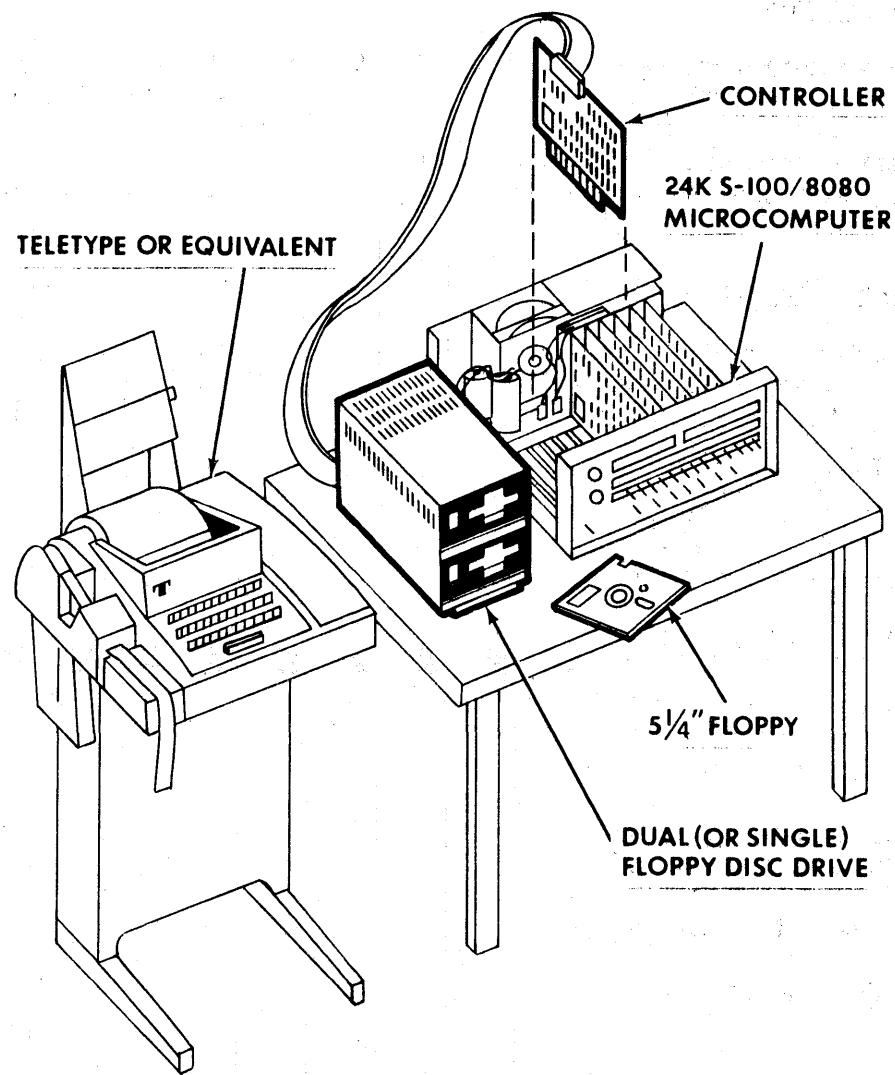


Figure 2.1 Typical Installation

- 5) RETAIN THE PACKING MATERIALS IN CASE IT IS NECESSARY TO RETURN THE EQUIPMENT TO THE SOURCE OR SUPPLIER. DO NOT ATTEMPT TO SHIP THE EQUIPMENT EXCEPT IN THE ORIGINAL PACKING.

2.1.2 INITIAL CHECKOUT

Open the plastic bag enclosing the Disk Module and the controller box and inspect for shipping damage. If shipping damage is evident, call the origin of the shipment: typically, the dealer from whom the equipment was purchased or shipped (or Micropolis in the case of a direct factory sale).

DO NOT RETURN THE DAMAGED EQUIPMENT UNTIL THE SHIPPING COMPANY INSPECTOR HAS REVIEWED THE DAMAGE, SINCE AN INSURANCE CLAIM WILL BE MADE.

Ensure that the model number on the identification plate is as ordered. If a Mod II (high capacity) drive was ordered check that the disk load actuator on the front of the drive is blue; for a Mod I the actuator is black.

2.1.3 CONTROLLER HARDWARE REQUIREMENTS

The disk controller board is accessed as a 1K block of memory using memory-mapped I/O. This addressing scheme leaves the full 256 standard I/O addresses for user devices. The controller is implemented as a "software controller"; most of the work required to access the disk is performed in software. The operation of the primitive read/write and timing loops depends upon instruction timing, which places the following restrictions on the system environment:

- 1) RAM memory must be fast enough to operate without wait states. This implies 450 nsec or less access time with a 2 MHz system clock.
- 2) If dynamic RAM is used, the overhead for refresh must not be more than 1 CPU clock cycle per 32 usecond period. The refresh logic must operate properly with approximately 18 usec/32 usec period spent in wait states. (The controller synchronizes disk transfers by asserting the PRDY line.)
- 3) Interrupts are disabled during disk I/O operations.
- 4) No cycle-stealing DMA devices may be in operation during disk I/O operations.
- 5) The first 512 bytes of the 1K controller address space are allocated to the bootstrap, which is implemented in a 70 nsec ROM. The controller is mapped into the last 512 bytes.

2.1.4 CONTROLLER CONFIGURATION

CHANGED TO C400H

The Micropolis disk controller is normally configured to operate at a base address of F400H with a 2 MHz processor. You must ensure that there is no other memory in your system that conflicts with the 1K space

beginning at F400H. If a conflict exists follow the procedure in 2.1.4.1 to resolve the conflict. If you want to operate with a 3 MHz or 4 MHz processor follow the procedure in 2.1.4.2.

2.1.4.1 CHANGING THE CONTROLLER BASE ADDRESS TO C400H

The controller may be jumpered for a base address at any 1K boundary from C000H to FC00H by performing the following procedure.

- 1) Referring to Figure 2.2, locate the address jumpers W1 through W4. (The controller is shipped with W3 only installed.)
- 2) Referring to Figure 2.3, determine the jumpers required for the desired base address. Install the required jumpers using a short length of insulated wire.
- 3) Solder in the new jumper(s) using a 25-30 watt soldering iron and resin-core solder.

2.1.4.2 REJUMPERING FOR 3 MHZ OR 4 MHZ OPERATION

To operate the disk subsystem at processor speeds greater than 2 MHz, a jumper must be installed on the controller as follows.

- 1) Referring to Figure 2.4 locate the ribbon cable edge connector and the resistors R25, R6 and R7.
- 2) Between R25 and R6 is a jumper location, W9. Install jumper W9 using a small length of insulated wire and solder in place using a 25-30 watt soldering iron and resin-core solder.

A significant throughput advantage may be realized by operating the disk subsystem with a 3 MHz 8085 or 4 MHz Z80 processor. However, two important notes apply to this type of operation.

- 1) System integrity is critical at higher clock rates, particularly 4 MHz. Buss noise in an S-100 buss system which is not specifically designed for 4 MHz operation may reach unacceptable levels when a 4 MHz ZPU is used. To obtain best performance, it is suggested that the user place the Micropolis disk controller as close as possible to the CPU board, preferably the next slot.
- 2) Memory speed is extremely critical. Some "250 nsec" memories may not operate at 4 MHz because of logic delays which degrade the theoretical access time such that the access requirements of M1 cycles are not met. These marginal memory boards may be used if your processor is capable of inserting a wait state in an M1 cycle.

2.1.5 INSTALLING THE CONTROLLER AND INTERFACE CABLE

There are five steps involved in installing the controller and connecting the disk drive(s) to it. Figure 2.1 illustrates a typical installation.

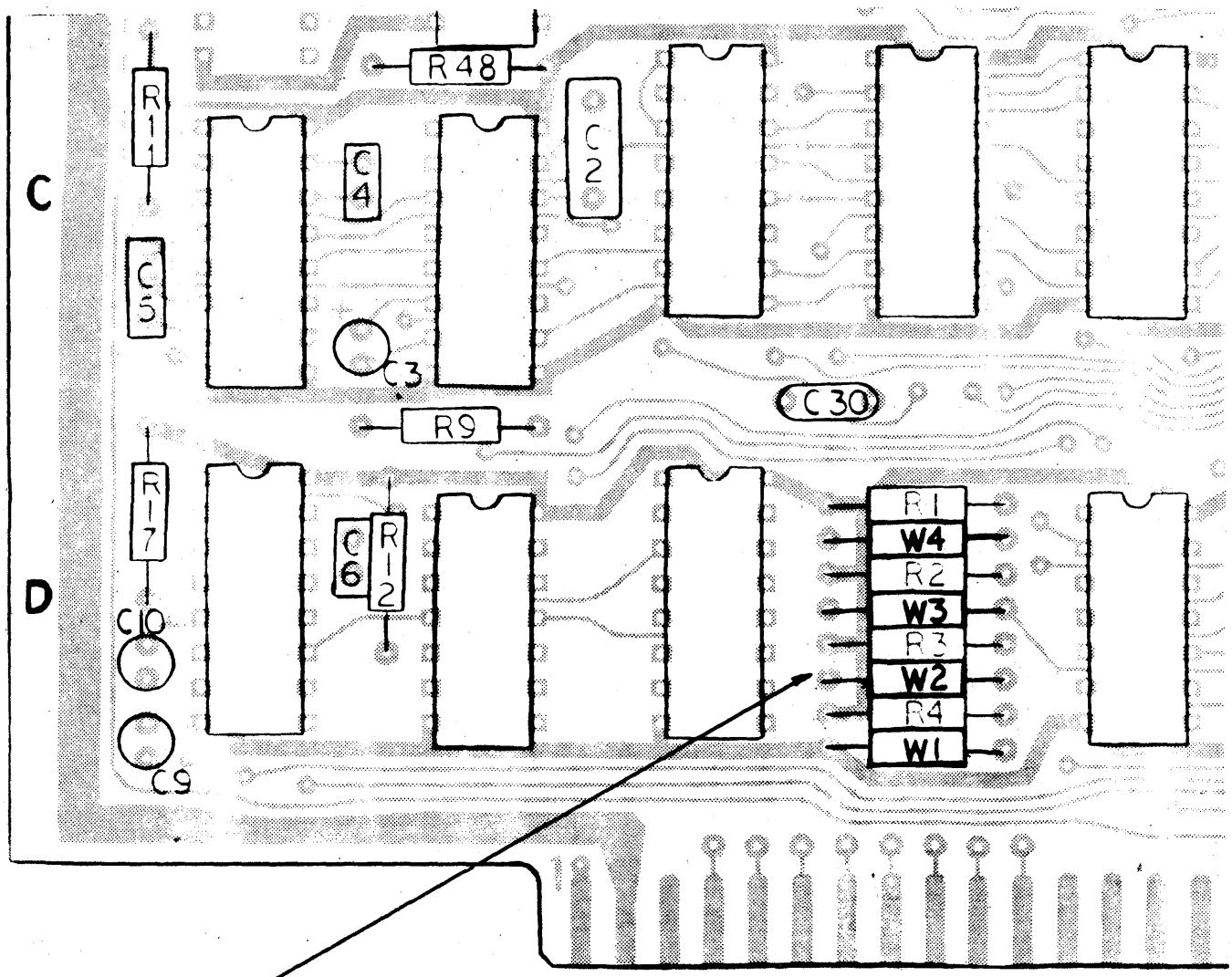


Figure 2.2 Locating The Controller Address Jumpers

VSE
C400

STANDARD
ADDRESS

BASE ADDRESS	ADDRESS BIT JUMPER							JUMPER INSTALL		
	A15 N/A	A14 W1	A13 W2	A12 W3	A11 W4	A10 N/A	A9 N/A	W1	W2	W3
C0 00 - C3FF	1	1	0	0	0	0	0	0	Y	Y
C4 00 - C7FF	1	1	0	0	0	1	0	0	Y	Y
C8 00 - CBFF	1	1	0	0	1	0	0	0	Y	Y
CC 00 - CFFF	1	1	0	0	1	1	0	0	Y	Y
D0 00 - D3FF	1	1	0	1	0	0	0	0	Y	N
D4 00 - D7FF	1	1	0	1	0	1	0	0	Y	N
D8 00 - DBFF	1	1	0	1	1	0	0	0	Y	N
DC 00 - DFFF	1	1	0	1	1	1	0	0	Y	N
E0 00 - E3FF	1	1	1	0	0	0	0	0	N	Y
E4 00 - E7FF	1	1	1	0	0	1	0	0	N	Y
E8 00 - EBFF	1	1	1	0	1	0	0	0	N	Y
EC 00 - EFFF	1	1	1	0	1	1	0	0	N	Y
F0 00 - F3FF	1	1	1	1	0	0	0	0	N	N
F4 00 - F7FF	1	1	1	1	0	1	0	0	N	N
F8 00 - FBFF	1	1	1	1	1	0	0	0	N	N
FC 00 - FFFF	1	1	1	1	1	1	0	0	N	N

As an example, if you wish to use base address E400, install jumpers at W2 and W3.

Figure 2.3 Controller Base Address Jumper Configurations

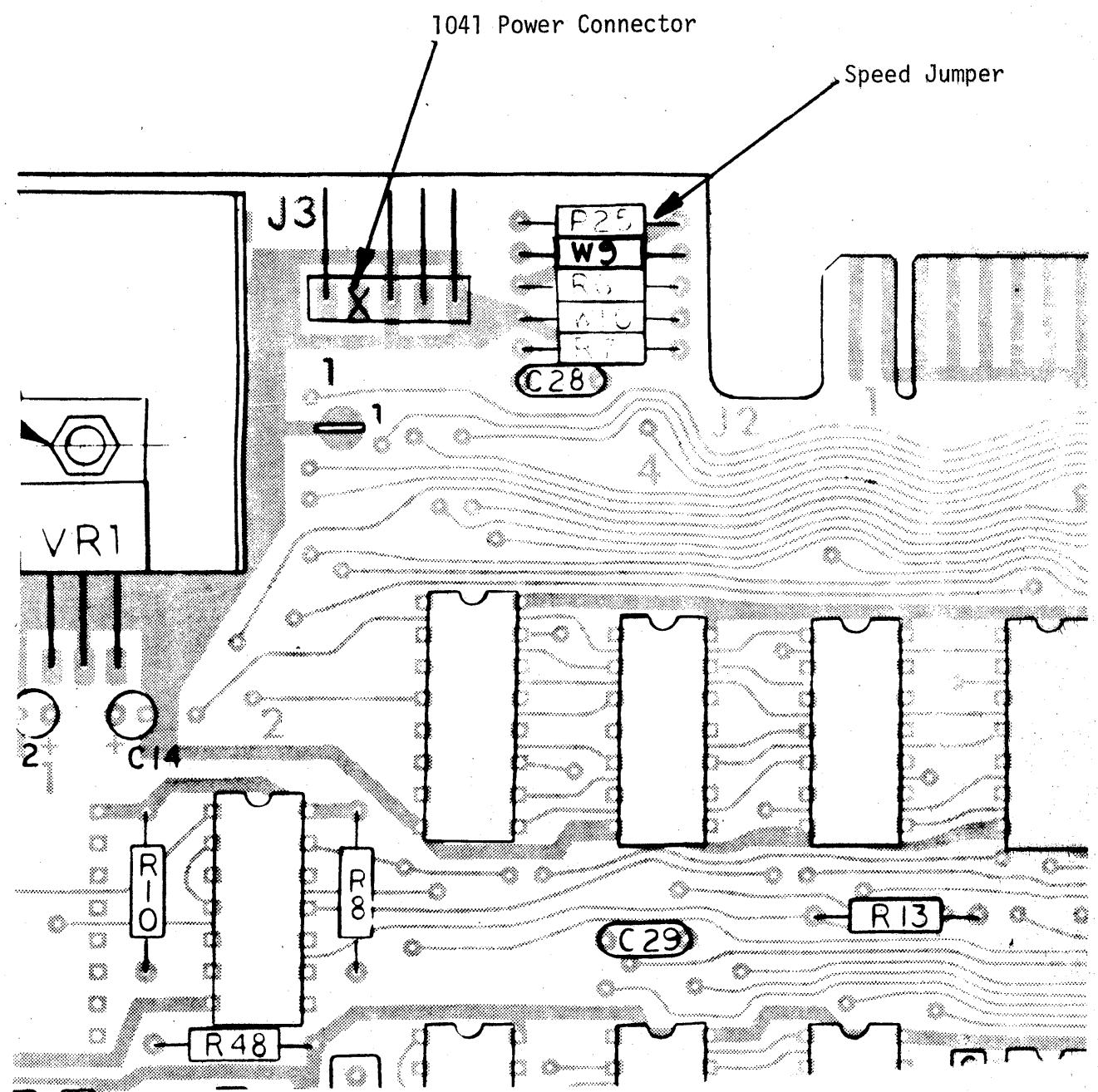


Figure 2.4 Locating the controller processor speed jumper and the 1041 power connector

- 1) Remove the cover from your microcomputer, exposing the printed circuit board assemblies.
- 2) The Micropolis Controller is designed to be inserted at any position on the S-100 bus. Choose a position on the bus which is most convenient for dressing the interface cable from the back or side of the computer chassis.
- 3) Insert the Controller with its component side facing the same direction as the component side of the already installed boards.
- 4) Install the interface connector by inserting it onto the 34-pin etched connector on the top edge of the Controller board. Take care to align the contrasting colored wire in the Interface Cable itself with pin 1 on the Controller board. The connection is keyed and can be inserted one way only, so do not force it hard if there is resistance; instead, remove the connector and recheck the alignment before reinserting.
- 5) Connect the other end of the Interface Cable to the Disk Module.

For the Dual Disk Module, the etched interface connector is located at the rear of the unit. The connector should be installed ensuring that the contrasting cable color indicating pin 1 is located at the top of the flat cable.

For the Single Disk Module except 1041/1021, the Interface Cable is shipped already installed in the module itself. If it becomes necessary to remove the Interface Cable, the following procedure should be followed:

- a) Remove the screws on each side and the single screw at the top-rear of the cover. Slide the cover back about one quarter of an inch to disengage it from the front bezel and raise it carefully to avoid damage to leads.
- b) Unplug the connector from the circuit board and fold it through the slot in the base of the module.
- c) Replace the cover by reversing the process in a) above, taking particular care to avoid trapping the lead wires at the front-left of the module.

For 1041/1021 modules, install the connector on the circuit board edge connector accessible through the opening at the rear of the protective sleeve.

2.1.6 DAISY CHAINING MULTIPLE DISK DRIVES

Up to four disk drives (four Single or two Dual modules or two Singles plus one Dual) may be connected to the Controller. Accessory cables (daisy chain cables) which allow connection of two or three or four modules are available. These consist of lengths of flat cable with connectors on each end and one or more connectors spliced at appropriate points down the cable. The method of installation is identical to that in 2.1.5 above.

Normally, a Single Disk Module shipped as part of a 1043/1042 or 1041 subsystem has line terminators in place and disk address \emptyset (determined by jumper positions on the printed circuit board in the module). Unless otherwise notified, Micropolis ships an "add-on" model of a single disk module (1023/1022/1021) assigned as address "1" without interface line terminators.

Normally, a Dual Disk Module shipped as part of a 1053 subsystem has line terminators in place and disk addresses set at \emptyset and 1. Note that the slide switch located (from the rear of the unit) on the printed circuit board just above the interface connector may be used to reverse the address assignments within the dual module, so that the drive formerly addressed as \emptyset is now 1, and vice versa. This is particularly useful, for example, when a disk on which data files can only be accessed through programs written for disk \emptyset can be mounted on disk drive address 1, and by toggling the switch the necessity for swapping disks or changing software is removed. An "add-on" Dual Disk Module (1033) is assigned addresses "2" and "3" and has no interface line terminators.

In any multiple disk module system implementation, it is mandatory that:

- 1) Only one drive module contain line terminators.
- 2) That module containing the line terminator (usually module \emptyset for Single Disk Modules or module $\emptyset/1$ for Dual Disk Modules) should be physically connected to the last connector on the daisy chain interface cable, i.e. the furthest from the Controller end of the cable.

2.1.7 APPLYING DC POWER (MODEL 1041/1021 ONLY)

Model 1041/1021 modules do not include power supplies thereby requiring the user to supply DC power. The user may provide regulated power directly or may provide unregulated voltages to modules equipped with the optional regulator kit, Model 1091-01.

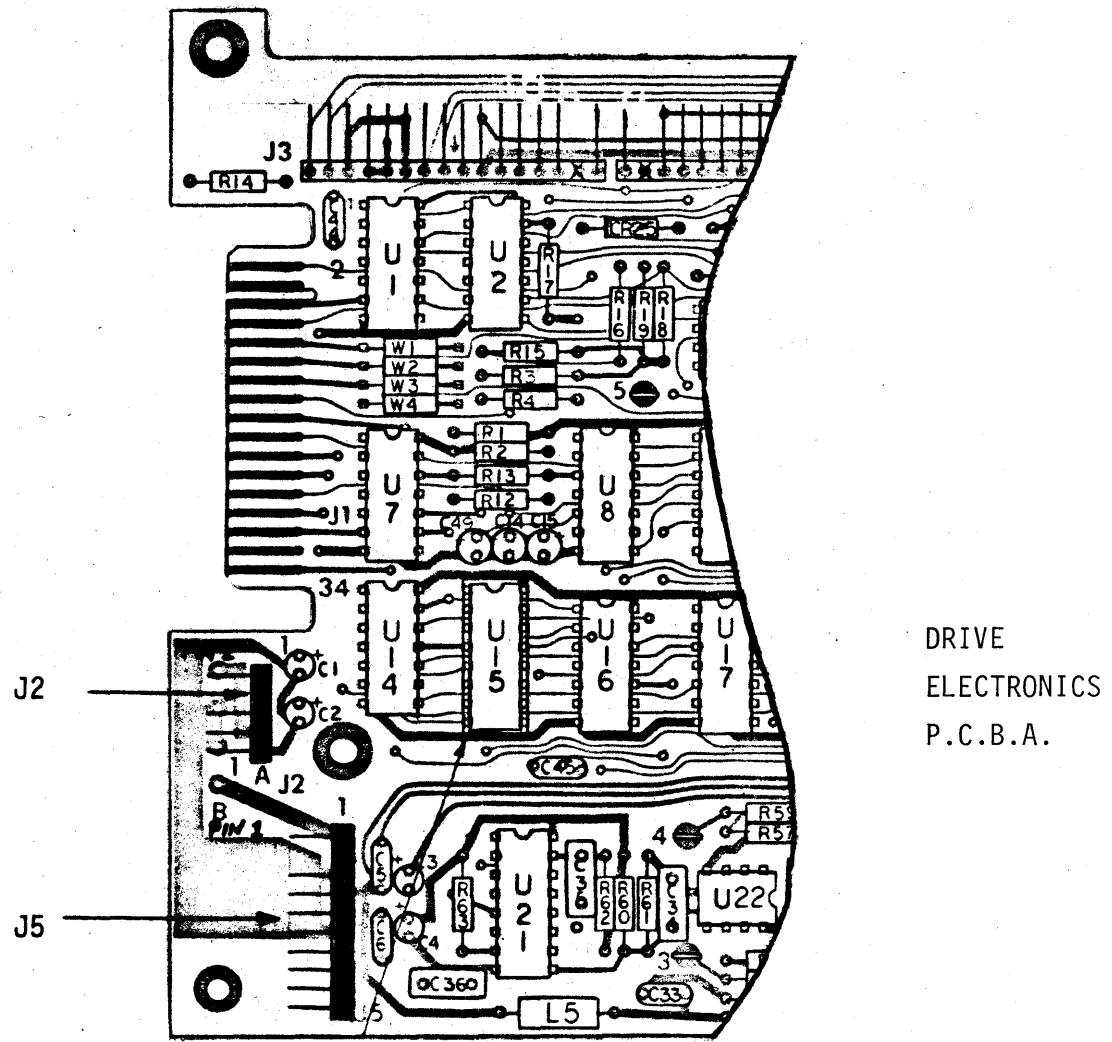


Figure 2.5 Model 1041/1021 Power Connectors

2.1.7.1 REGULATED DC

Regulated DC voltages are applied to J5 of the drive electronics board (refer to Figure 2.5 for location of J5) as follows:

J5 PIN	VOLTAGE	CURRENT REQUIREMENTS	WIRE COLOR*
7	+5VDC ± 5%	.5 AMP	VIOLET
6	+5V RETURN		BLUE
4	+12VDC ± 5%	1.15 AMP	YELLOW
3	+12V RETURN		ORANGE

*Wire color refers to power cable A supplied with 1041/1021 drive.

+5 Return and +12 Return must be connected together at the power supply. The drive chassis must be connected to the computer chassis or directly to earth ground.

2.1.7.2 UNREGULATED DC

Unregulated DC power may be applied to modules equipped with the optional regulator kit Model 1091-1. Each regulator kit provides regulated DC power for one 1041 or 1021 module. Install the kit as follows:

Install the heatsink assembly on the rear of the protective sleeve using the hardware provided. Plug the connector from the heatsink onto J5 of the drive electronics board (see Figure 2.5 for location of J5).

Unregulated DC power is applied to J2 of the drive electronics board (see Figure 2.5 for location of J2) as follows:

J2 PIN	VOLTAGE	CURRENT REQUIREMENTS	WIRE COLOR*
1	+16V UNREGULATED (+15 to +17)	1.15 AMP	BROWN
2	KEY		
3	+16V RETURN		ORANGE
4	+8V UNREGULATED (+8 to +10)	.5 AMP	YELLOW
5	+8 RETURN		GREEN

*Wire color refers to the 4 wire power cable B supplied with the regulator kit.

Unregulated DC may be obtained from an S-100 bus computer by connecting the 4 wire power cable B supplied with the regulator kit between J2 of the drive electronics board and J3 on the Controller B board. A maximum of one drive may be powered by the controller in this manner. It is suggested that multi-drive systems be powered directly by a separate power supply. Dual power supplies providing +5 and +12 regulated are commercially available from several manufacturers.

2.1.8 CUSTOM MOUNTING OF THE 1041/1021 DRIVES

The 1041/1021 disk drive is enclosed in a protective sleeve with four rubber feet installed for desk-top use. The rubber feet may be peeled off for custom mounting, such as in the front panel of a computer main frame.

The following guide lines are recommended.

Refer to Figure 2.5-B.

- a) The drive may be mounted in any orientation except up-side down. If the drive is to be mounted vertically, it should be ordered as such so that the disk eject system can be suitably adjusted.
- b) Use the recommended panel opening and insert the drive through the panel opening from the front so that the drives are restrained from rotating.
- c) On no account should the mounting scheme rely on any restraint to drive motion being applied through the plastic bezel.
- d) When mounting the drive with the width dimension (5.9") vertical, use the outside two screws indicated in Figure 2.5-B on the appropriate side and two spacers to secure the drive to the base chassis.

Spacers should be at least 0.5" outside diameter.

The screws should be of such a length as to not protrude more than 0.2" into the inside of the drive.

The holes in the base chassis should have adequate clearance to take up tolerances. This precludes the use of flat head screws.

- e) When mounting the drive with the width dimension (5.9") horizontal, use brackets made of .060 min. steel mounted to the base chassis to secure the drive in four places using the outside two screws on both sides.

Holes in the brackets should have adequate clearance so that when all screws are tight, stress is not communicated to the drive.

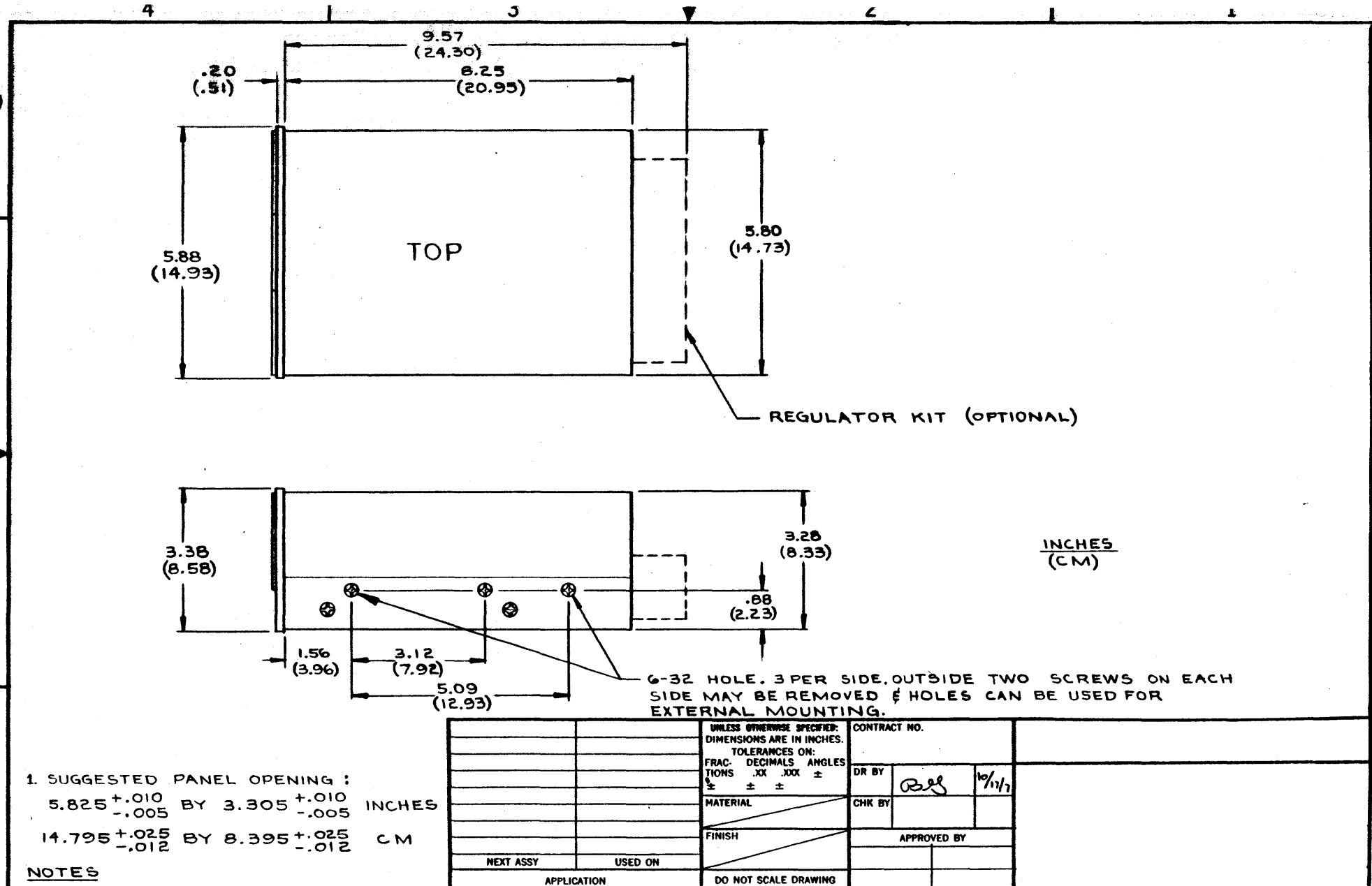


Figure 2.5 - B OUTLINE DRAWING

- f) When mounting the drive vertically, the drive mounting should not allow stress to be placed on the plastic bezel.

2.1.9 DISK LOADING AND UNLOADING

The flexible disk is loaded with the load actuator in the "up" position. Push the disk "home" until an audible click occurs. This means the disk is properly located in the receiver. Load the disk, by pushing down on the load actuator, as far as it will go. Move the actuator firmly and slowly to ensure proper seating of the disk on the locating cone. The actuator remains in the "down" position indicating that the disk is loaded.

If the disk is absent or if it is not properly "home" it is not possible to depress the load actuator. This feature protects the disk from damage if not located properly.

To unload the disk, depress the load actuator as far as it will go, and allow it to rise to the "up" position. In order to eject the disk, place the tip of the forefinger under the load surface of the actuator and tilt the actuator upwards. This action unlatches the interlock and pushes the disk into your hand.

2.2 SYSTEM SOFTWARE INSTALLATION

Each Micropolis disk subsystem includes a MASTER diskette which contains the Program Development Software (PDS) systems. Software installation consists of building a SYSTEM diskette configured for your I/O devices from the unconfigured MASTER diskette.

2.2.1 PROGRAM DEVELOPMENT SOFTWARE MEMORY REQUIREMENTS

The Micropolis Diskette Operating System (MDOS) requires a minimum of 16 kbytes of contiguous RAM starting at $\$000$. Figure 2.6 illustrates these memory requirements.

The Micropolis disk extended BASIC system requires a minimum of 24 kbytes of contiguous RAM, starting at location $\$000$. BASIC automatically sizes RAM memory when it is started. If you have additional RAM which you desire BASIC to use, this memory must be strapped to be contiguous with this first 24K. Figure 2.7 illustrates these memory requirements.

2.2.2 SUPPORTED I/O DEVICES

The PDS MDOS and the PDS BASIC system support the same I/O devices through the common RES module.

- 1) Micropolis flexible disk subsystems
- 2) Terminal - (See Figure 2.8)
- 3) Line Printer

2.2.3 LOADING THE PDS MDOS SYSTEM INTO MEMORY FROM THE MASTER DISKETTE

The first procedure in the sequence leading to a configured SYSTEM disk is to load the PDS MDOS System from the unconfigured MASTER disk into memory and determine that the load was successful.

- 1) Ensure that the Micropolis controller and disk drive are properly connected to your system. Apply power to your system.
- 2) Insert the PDS MASTER diskette into your drive (unit 0 on multiple drive systems) and load the diskette by depressing the actuator lever.
- 3) Activate the bootstrap ROM on the controller. For Altair/Imsai type computers with a front panel, this is done by setting the address switches to the bootstrap address (~~F400H~~ unless the controller base address has been changed), reset, examine, and run. For computers under control of a resident ROM monitor, follow the manufacturers instructions on starting program execution at a given address. Use the address of the bootstrap ROM (~~F400H~~ unless the controller base address has been changed).
C400 H

C400 H

FIGURE 2.6 MDOS MEMORY MAP - Release 4.0

0000H	Not used by Micropolis hardware/software
006AH	Initially used by bootstrap loader. After the system is loaded, this space is used for the system stack.
01A0H	The RES module contains all permanently resident I/O and Disk Control Routines and associated buffers.
1598H	The MDOS module contains the command executive and all user callable routines not in RES.
2B00H	The Applications program area extends from here to the end of contiguous memory.

C400 F400H

F600H

C800 F800H

BOOTSTRAP ROM as supplied by Micropolis
Micropolis Disk Controller

FIGURE 2.7 BASIC SYSTEM MEMORY MAP - Release 4.0

0000H

Not used by Micropolis hardware/software

006AH

Initially used by bootstrap loader. After the system is loaded, this space is used for the system stack.

01A0H

The RES module contains all permanently resident I/O and Disk Control Routines and associated buffers.

1598H

The BASIC Interpreter

5700H

The BASIC current program buffer extends from here to the end of contiguous memory.

C400

F400H

F600H

F800H

BOOTSTRAP ROM as supplied by Micropolis

Micropolis Disk Controller

When the boot program is started, the unit select indicator on the drive will illuminate and the disk head will load with an audible "click". Computer front panel address lights will flash while reading is taking place.

After about 10 seconds, the unit select indicator should go out and the head will unload with an audible "click". If this has not occurred within about 20 seconds then the boot program has been unable to read the system loader into RAM properly. Reset the system and try again. If a retry is unsuccessful, then remove the diskette and re-load it into the drive; the diskette may not have seated properly the first time.

- 4) When the unit select indicator goes out, press stop and observe the address indicators. The halt address should be one of the following.

0397H - Loader error

04CDH - Good Load

To determine if the load was successful in systems without a front panel, or to ascertain the cause of a loader error, examine the contents of location 039AH (Loader termination status). The status code should be one of the following.

47H (ASCII"G") - GOOD LOAD - the RES and MDOS modules are now in RAM.

55H (ASCII"U") - UNRECOVERABLE DISK ERROR - the loader was unable to read the system into memory.

Probable causes:

*Diskette is not seated properly.

*Drive did not step properly - remove and reinsert diskette and retry boot process.

4DH (ASCII"M") - BAD MEMORY - The loader tried to write into memory and was unable to read back the same data. Probable causes are:

*Insufficient contiguous memory - 16K bytes from address 0 are required.

*Memory is write protected.

*Defective memory.

019BH/019CH contain the RAM address at which the error occurred.

If the status code is not one of the above, the memory into which the loader was read may be defective or nonexistent.

2.2.4 CONFIGURING THE PDS SYSTEMS FOR YOUR TERMINAL

The Micropolis disk subsystem and the PDS systems are designed to run in an S-100 bus - 8080 compatible microcomputer. S-100 bus compatibility does not define the device addresses or I/O protocol used in communicating with the various interface boards which may be used to connect a terminal keyboard/printer to your computer. Therefore, it is necessary to customize the terminal input-output routines in the RES module to accommodate your precise equipment configuration.

The MDOS system loaded per Section 2.2.3 contains a special configurator program which is provided to simplify the terminal configuration task for specific "standard" terminal interface boards. These boards are standard in the sense that the port addresses and flag bit assignments conform to what is used by the manufacturers in their stand-alone software. Figure 2.8 is a list of computers and interface boards and "standard" ports and logic. To determine which terminal configuration procedure applies to your equipment refer to Figure 2.8 and follow these steps.

- 1) If your equipment is listed in the DEVICE column and your port addresses and flag bit assignments match the ones listed, then configure your terminal by following the procedure in Section 2.2.4.1.
- 2) If your equipment is listed in the DEVICE column but you have used different port addresses or flag bit assignments, then configure your terminal by following the procedure in Section 2.2.4.2.
- 3) If your equipment is not listed in Figure 2.8, then configure your terminal by following the procedure in Section 2.2.4.3.

2.2.4.1 CONFIGURING A STANDARD TERMINAL

- 1) In Altair/Imsai front panel type systems, set the address switches to 04D1H and examine. Set the program input (sense) switches to the configuration number corresponding to your configuration in Figure 2.8 and press run. This activates the configurator program.

In systems without a front panel, set the desired configuration number into location 04D0H and start program execution at location 04D6H . This activates the configurator program.

- 2) Once started, the configurator program will build the terminal handler corresponding to the configuration number and will start MDOS which should output the sign-on message:

MICROPOLIS MDOS VS X.X - COPYRIGHT 1978

- 3) Continue the SYSTEM disk building process with Section 2.2.5.

FIGURE 2.8 STANDARD TERMINAL CONFIGURATIONS

Config			Port Assignments (HEX)				Flag Bits		Output		Device Type
			Input	Output	Data	Data	Input	Level	Ready	Level	
NBR	Device		Status	Status	In	Out	Ready	Level	Ready	Level	
0	Altair	88-2S10	10	10	11	11	0	HIGH	1	HIGH	SERIAL
1	IMSAI	S102	3	3	2	2	1	HIGH	0	HIGH	SERIAL
2	Altair	SIO A,B,C (not rev 0)	0	0	1	1	0	LOW	7	LOW	SERIAL
3	Altair	SIO A,B,C (rev 0)	0	0	1	1	5	HIGH	1	HIGH	SERIAL
4	PTC	3P+S	0	0	1	1	6	HIGH	7	HIGH	SERIAL
5	IMSAI	MIO	43	43	42	42	1	HIGH	0	HIGH	SERIAL
6	Altair	88-4PIO	10	12	11	13	7	HIGH	7	HIGH	PARALLEL

The above configurations assume a terminal output line width of 72 characters and append 2 nulls to the output stream following a carriage return.

See ASSIGN command in MDOS or BASIC for instructions for changing width or number of nulls.

80H COMPAL - 80 Terminal I/O is performed through the COMPAL monitor

81H PTC SOL - 20 WITH SOLOS 1.3 - Terminal I/O is performed through SOLOS pseudo port 0.

2.2.4.2 CONFIGURING A MODIFIED STANDARD TERMINAL

To modify one of the standard terminal handlers to accommodate different port addresses or flag bit assignments proceed as follows.

- 1) Refer to the listing of the I/O handler and configurator program in Appendix E. The listing is structured as follows:
 - E.1 Logical input/output routines
 - E.2 General terminal handler
 - E.3 Line printer handler
 - E.4 Configurator (starting with label "CNFIG")
 - E.5 Blocks of configuration parameters corresponding to the configurations listed in Figure 2.8, labelled CNFG₀, CNFG₁.....CNFG_n
- 2) Locate the parameter block which corresponds to your I/O board.

The parameter block is organized as follows:

<u>ADDRESS</u>	<u>CONTENTS/DESCRIPTION</u>
CNFG +0	Terminal Input Status Port
+1	Terminal Input Status Port
+2	Terminal Output Status Port
+3	Terminal Data Input Port
+4	Terminal Data Input Port
+5	Terminal Data Output Port
+6	Data Input Ready Flag
+7	Data Input Ready Mask
+8	Data Input Ready Flag
+9	Data Input Ready Mask
+A	Data Output Ready Flag
+B	Data Output Ready Mask
+C	Bytecount For Init Logic = n
+D	
.	
.	Initialize Logic
.	
+E+(n-1)	

- 3) Modify the parameter block for the port addresses and/or flag bit assignments used by your interface card (don't overlook changing the addresses in the initialize code as well).
- 4) The flags and masks are created as follows:
 - a) Data input/output ready flag byte is ANDed with the appropriate status byte to extract the desired status bit. The result is then exclusive - OR'ed with the associated mask byte.

- b) If the status bit is high true, i.e., (1) = condition true, then the mask associated with the flag byte = flag byte.
 - c) If the status bit is low true, i.e., (0) = condition true, then the mask = \emptyset
- 5) In Altair/Imsai front panel type systems, set the address switches to $\emptyset 4D1H$ and examine. Set the program input (sense) switches to the configuration number corresponding to your configuration in Figure 2.8 and press run. This activates the configurator program.

In systems without a front panel, set the desired configuration number into location $\emptyset 4D0H$ and start program execution at location $\emptyset 4D6H$. This activates the configurator program.

- 6) Once started, the configurator program will build the modified terminal handler and will start MDOS which should output the sign-on message:

MICROPOLIS MDOS VS X.X - COPYRIGHT 1978

- 7) Continue the SYSTEM disk building process with Section 2.2.5.

2.2.4.3 NON-STANDARD TERMINAL CONFIGURATION

If your terminal/interface device cannot be found in Figure 2.8, this section describes the I/O requirements of the PDS systems so that you can write your own terminal handler.

When you boot the MASTER diskette a set of generalized I/O handlers are loaded into memory within the RES module. Figure 2.9 is a map of this area.

2.2.4.3.1 THE CONSOLE I/O TABLE

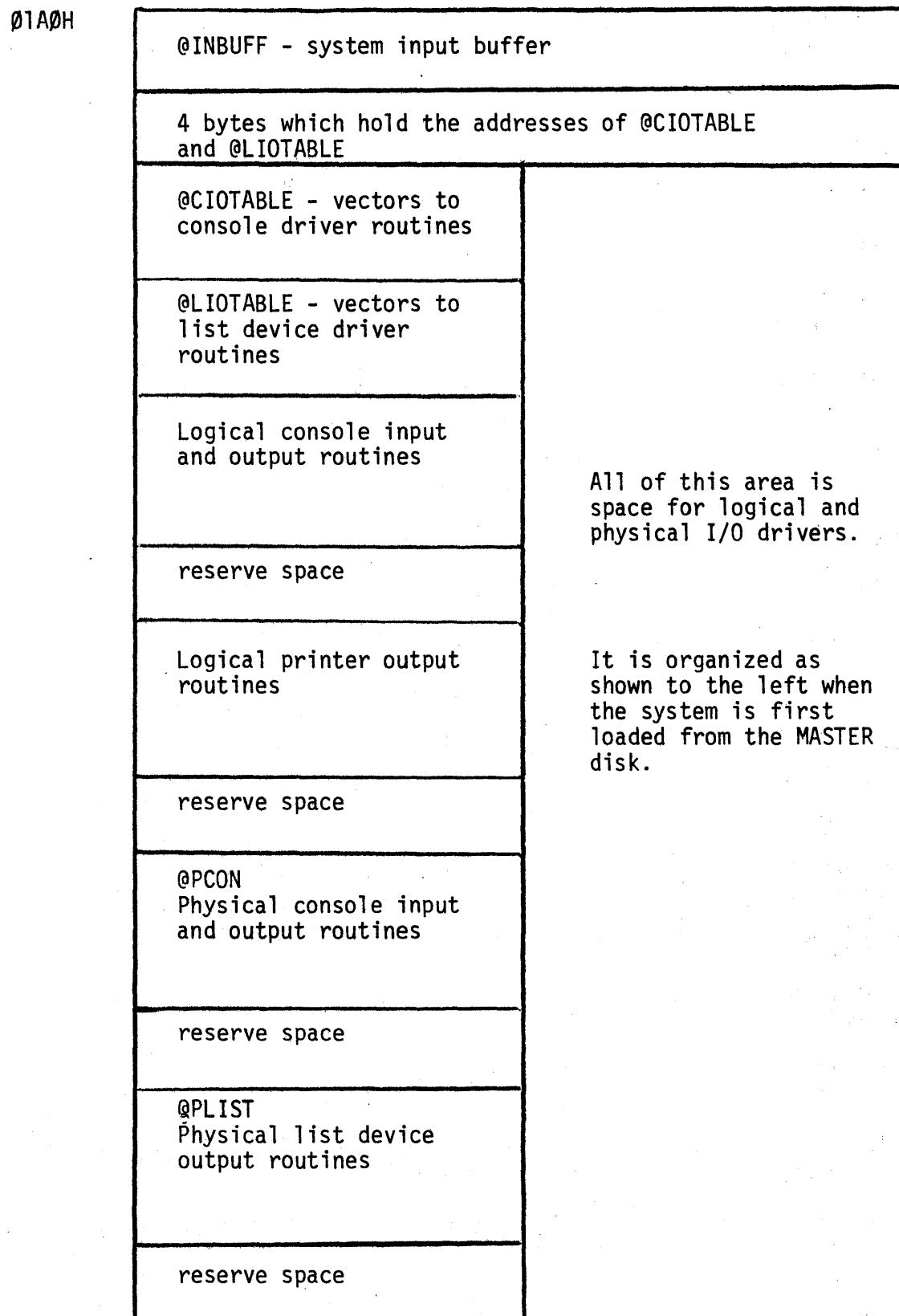
The @CIOTABLE has the following form:

```

ORG @CIOTABLE
DW CIN      ; address of logical console input
DW COUT     ; address of logical console output
DW CBRK     ; address of logical console break check
DW CDIN     ; address of physical console device input
DW CDOUT    ; address of physical console device output
DW CDBRK   ; address of physical console device break check
DW CDINIT   ; address of physical console device initialize
WRAPFLAG  DB 0       ; enable (0) or disable (1) console wrap logic
NULLS     DB 3       ; console null count + 1
WIDTH     DB 3FH     ; console carriage width
DS 1       ; must be provided for internal system use

```

FIGURE 2.9 I/O DRIVER AREA IN RES MODULE



2.2.4.3.2 LOGICAL CONSOLE I/O (CIN, COUT, CBRK)

The logical input, output and check break routines should not have to be changed. They are tailored to support all MDOS and BASIC requirements.

2.2.4.3.3 PHYSICAL CONSOLE DEVICE INPUT (CDIN)

The console physical input routine must have the following characteristics:

- 1) It must return all registers except A & B unchanged.
- 2) It can use the A register (destroy it).
- 3) It must return an ASCII character including the parity bit if any, in the B register.
- 4) It must return the carry flag clear (NC). The other status flags can be in any state.

If the physical character input routine is rewritten, its entry address must be put into the @CIOTABLE at DW CDIN.

2.2.4.3.4 PHYSICAL CONSOLE DEVICE OUTPUT (CDOUT)

The console physical output routine must have the following characteristics:

- 1) It must take an ASCII character in the B register.
- 2) It must return all registers except A unchanged.
- 3) It can use the A register (destroy it).
- 4) It must return the carry flag clear (NC). The other status flags can be in any state.

If the physical character output routine is rewritten, its entry address must be put into the @CIOTABLE at DW CDOUT.

2.2.4.3.5 PHYSICAL CONSOLE DEVICE BREAK CHECK ROUTINE (CDBRK)

The console physical check break routine must have the following characteristics:

- 1) It must check the console input status port to determine if a key has been pressed.
- 2) If no key has been pressed it must return all registers except A unchanged and the zero flag clear (NZ).
- 3) If a key has been pressed return the byte including the parity bit if any, in the B register.

The A register can be used (destroyed). All other registers must be unchanged. The zero flag must be set (Z).

- 4) The status flags other than zero can be in any state.

If the physical check break routine is rewritten, its entry address must be put into the @CIOTABLE at DW CDBRK.

2.2.4.3.6 PHYSICAL CONSOLE DEVICE INITIALIZE (CDINIT)

This routine initializes the input/output interface. Some devices are not programmable and cannot be software initialized, while others like the INTEL 8251, or the Motorola 6850 must be software initialized.

If your equipment needs software initialization, the routine must have the following characteristics:

- 1) It must return all the registers except A unchanged.
- 2) It can use the A register (destroy it).
- 3) It must return the carry flag clear (NC). The other status flags can be in any state.

If your equipment does not need to be software initialized your routine only needs to clear carry (NC) and return.

If you rewrite the initialization routine, you must put its entry address into the @CIOTABLE at DW CDINIT.

2.2.4.3.7 STARTING YOUR SYSTEM

After you have written your driver and made the appropriate patches to the @CIOTABLE, you are ready to start the system. Change the soft halt at location 4CEH and 4CFH to E7H and 04H. Start execution at location 04E7H. The System will sign on with

MICROPOLIS MDOS VS X.X - COPYRIGHT 1978

Proceed to Section 2.2.5 to configure your system for other supported devices.

2.2.5 SYSTEM PRINTER CONFIGURATION

The Program Development system provides line printer support as well as terminal and disk I/O. If your system does not have a printer separate from the terminal, you are not required to build a line printer handler and may proceed to Section 2.2.6 to create your system disk.

PDS as loaded per Section 2.2.3 contains a generalized line printer handler. In many cases this handler can be configured to your equipment by patching the appropriate port addresses and flag bit assignments into the proper locations. To determine if this handler can support your equipment, refer to the listing of the physical line printer handler in Appendix E.3 beginning at the ORG @PLIST. Section 2.2.5.1 is a procedure for configuring this handler, if applicable. Section 2.2.5.2 presents a detailed example of interfacing a TELETYPE Model 40 printer. Section 2.2.5.3 is a procedure for writing your own printer handler, if necessary.

2.2.5.1 CONFIGURING THE SUPPLIED PRINTER HANDLER

The supplied printer handler performs three functions; output of an ASCII character, detection of a printer attention condition, and software initialization of programmable printer interface devices.

Refer to the printer handler in the system I/O HANDLER listing in Appendix E. The handler accesses the printer through three I/O port addresses:

PTDAT -- Printer Data Port -- Character data to be printed will be output to this port.

PTCTL -- Printer Control Port -- READY TO RECEIVE status will be read from this port.

PTSTS -- Printer Status -- PRINTER ATTENTION status will be read from this port. If your printer does not generate attention status then this port will not be used.

Printer attention detection requires two masks: PMSK1 and PMSK2. The handler inputs from port PTSTS and extracts the printer attention bit(s) by ANDing the status with PMSK1. The result is then exclusive OR'ed with PMSK2. The resulting condition code will be zero if printer is operational or non-zero if an attention condition exists.

Example: Assume a printer generates ON-LINE and PAPEROUT status which are connected to bits 7 and 0, respectively, of the status port. PMSK1 will be 081H to extract bits 7 and 0. The printer will be operational if and only if bit 7 = 1 and bit 0 = 0. PMSK2 must be constructed to yield a result of zero for this bit combination. Since Exclusive OR'ing the status which PMSK2 results in complementing each bit of the status for which the corresponding bit in PMSK2 = 1, the mask value required is 080H.

Ready to receive status detection also requires two masks: PMSK3 and PMSK4. The handler inputs from port PTCTL and extracts the ready to receive bit(s) by ANDing the status read with PMSK3. The result is then exclusive OR'ed with PMSK4. The resulting condition code will be zero if the printer is ready to receive or non-zero if the printer is busy. The masks are formed in the same manner as illustrated for printer attention detection.

Configure the printer handler as follows:

- 1) Determine the values of the port addresses and masks as described above for your printer and interface board. Determine the instructions required to initialize your printer/interface board.
- 2) You can make the patches with your running MDOS system or with your front panel switches (or monitor). If you want to use the system to make the changes, refer to the description of the ENTR command under MDOS EXECUTIVE in Chapter 4.
- 3) Change location @LIOTABLE+8 in the listing to the address of LDOUT.

This change is necessary because when the system boots this address is set to CDOUT, so both logical output streams go to the console device, which effectively no ops the printer handler.

- 4) If your printer does not support a printer attention condition skip to Step 8.
- 5) To configure the printer attention routine change location LDATN and LDATN+1 to Ø (NOP). The system boots with an XRA A and a RET in these locations which turns the attention logic off. Placing the two NOP's in the code activates the printer attention logic.
- 6) Change location LDATN+3 to the value of PTSTS (printer status port address) for your printer.
- 7) Change location LDATN+6 to the value of PMSK1 and location LDATN+8 to the value of PMSK2. The printer attention logic is configured.
- 8) To configure the character output routine, change location LDOUT1+1 to the value of PTCTL (printer control port address).
- 9) Change LDOUT2+1 to the value of PTDAT (printer data output port address).
- 10) Change location LDOUT1+4 to the value of PMSK3 and location LDOUT1+6 to the value of PMSK4. The printer character output routine is configured.
- 11) If your interface device requires software initialization, enter the machine code required starting at LDINIT and ending with the code C9H (RET). The code as assembled in the listing initializes an INTEL 8251 USART for two stop & 8 data bits with no parity. To activate this logic change locations LDINIT and LDINIT+1 to Ø.

If your equipment does not need initialization do not make any change to this code.

- 12) The logical printer output routine provides carriage return line feed after a specified number of characters as an option.

This allows lines longer than the carriage to wrap around rather than banging at the end of the carriage. If you want to disable this feature, change location PWRAPFLAG to a 1, otherwise disregard.

- 13) The number of nulls output in conjunction with a carriage return and the width associated with the wrap logic can be set using the ASSIGN command. These values are set at 2 nulls and 72 character width when the system is booted. The ASSIGN command is described in Chapter 4 under MDOS EXECUTIVE COMMANDS and Chapter 5 under BASIC PRINT FILE OUTPUT.

- 14) Some applications and systems programs need to know if the printer hardware is capable of advancing to the top of a page when a form feed is output or if the software needs to handle the top of form by issuing the correct number of line feeds.

A memory location is provided in the RES module which can be set at configuration time to indicate the type of printer you have. This memory location is called FORMFLAG and is located at 4C8H. A FORMFLAG of 0 indicates a printer which does not do a top of form when it receives a form feed. A FORMFLAG of 1 indicates a printer that does a top of form when it receives a form feed. The value of the FORMFLAG is 0 as the system is shipped. This is the configuration that would be used with a Teletype 33 that does not have a hardware top of form feature.

If your printer does a top of form when it receives a form feed (ASCII code 12 decimal) set this location to a 1 by typing:

ENTR 4C8 and a carriage return.
1/ and a carriage return.

The ASSM program, for example, checks the FORMFLAG and outputs a form feed if it is a 1 or line feeds if 0, to advance to the top of the next page.

User applications programs can also use the FORMFLAG to make the software less hardware dependent by providing both form feed logic and multiple line feed logic, which is conditionally executed depending on the sense of the FORMFLAG.

- 15) You have finished configuring the line printer handler. Type EXEC 4E7 and a carriage return, to warmstart the system and initialize the printer.

You can test the printer by typing ASSIGN 2 3 and a carriage return. The printer should echo all characters typed on the console. Type ASSIGN 2 2 and a carriage return and the printer should stop echoing.

16) Proceed to Section 2.2.6 to create a configured SYSTEM disk.

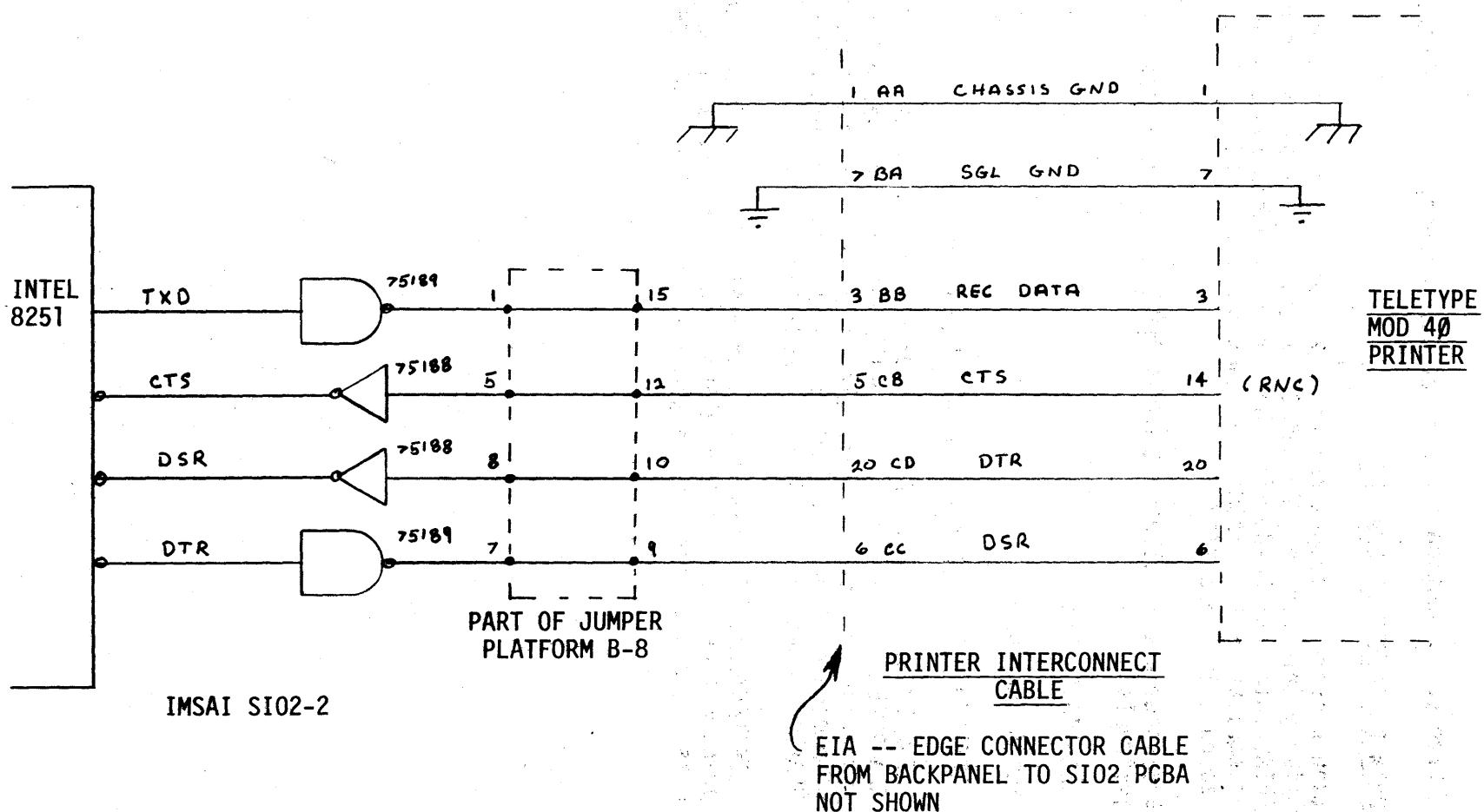
2.2.5.2 PRINTER INTERFACE EXAMPLE

This section presents a comprehensive case study of interfacing a TELETYPE Model 40 line printer to an IMSAI 8080 system. This example assumes an SI02-2 SERIAL INTERFACE BOARD with the terminal connected to port A. The printer is equipped with an ASCII EIA-type interface which interfaces directly to port B of the SI02.

The printer interface is illustrated in Figure 2.10 and consists of the following signals:

- 1) CHASSIS GROUND
- 2) SIGNAL GROUND
- 3) RECEIVED DATA -- Serial data to be printed.
- 4) CLEAR TO SEND -- The printer interface line "REQUEST NEXT CHARACTER" (RNC) is applied to the CTS line to enable the USART device on the serial board. This synchronizes transfers to the printer and allows the TRANSMITTER READY status bit of the USART to function as "READY TO RECEIVE".
- 5) DATA TERMINAL READY -- The printer asserts the DTR line when printer power is on and no alarm conditions such as paper out exist. This status line is jumpered to the USART DATA SET READY

Figure 2.10 Interfacing a Teletype Model 40 Printer with EIA Interface Option to an
IMSAI SI02-2 Port B



(DSR) input line. The state of this line may be read as one of the USART status bits and serves as PRINTER OPERATIONAL for printer attention detection.

- 6) DATA SET READY -- The DATA TERMINAL READY output line from the USART is applied to the DATA SET READY (DSR) interface line. When asserted, DSR turns the printer motor on.

This interface requires the user to fabricate the printer interconnect cable shown.

The SI02-2 is to be jumpered so that the USART status register may be read from port 5 and the USART data register may be written into from port 4.

The status byte read from the USART consists of the following bits:

7	6	5	4	3	2	1	0
D						T	
S	/	/	/	/	/	X	
R	/	/	/	/	/	R	D
							Y

BIT 7 -- DSR = (PRINTER OPERATIONAL)
0 = Printer Attention
1 = Printer Operational

BIT 6-1 -- Don't Care

BIT 0 -- TRANSMITTER READY -- (READY TO RECEIVE) 1 = the USART is ready to receive a character to transmit to the printer.

Since both printer operational and ready to receive are contained in the same status byte, PTSTS = PTCTL = 5.

The printer data port is the USART data register, so PTDAT = 4.

The masks required for attention and ready status are:

PMSK1 = 080H
PMSK2 = 080H
PMSK3 = 1
PMSK4 = 1

Refer to the printer handler in the System I/O HANDLER listing in Appendix E. The handler listed has been assembled for the example given in this section.

Details of the operation of the 8251 USART may be obtained from the INTEL application note AP-16 USING THE 8251 USART.

Since all of the port addresses and other parameters are assembled into the system printer handler, configuring the handler for this example is simply a matter of enabling the handler. However, to illustrate the procedure given in Section 2.2.5.1 the full dialogue is given below. The procedure step numbers are annotated to the right of the listing:

>ENTR 50A	Step 3
>CB 6/	
Skip to Step 8 if Printer Attention is not required	Step 4
>ENTR 6E8	Step 5
>0 0/	
>ENTR 6EB	Step 6
>5/	
>ENTR 6EE	Step 7
>80/	
>ENTR 6F0	
>80/	
>ENTR 6D0	Step 8
>5/	
>ENTR 6DB	Step 9
>4/	
>ENTR 6D3	Step 10
>1/	
>ENTR 6D5	
>1/	
>ENTR 6FE	Step 11
>0 0 3E AA D3 5 3E 40 D3 5 3E CE D3 5	
>3E 17 D3 5 C9/	
>ENTR 510	Step 12
>1/	
>ASSIGN 2 2 48 1	Step 13
>EXEC 4E7	Step 14

MICROPOLIS MDOS VS X.X - COPYRIGHT 1978

2.2.5.3 CONFIGURING SPECIAL PRINTER HANDLERS

If you are unable to patch the generalized print handler for your system, you will have to write your own. A general discussion of the needed routines follows. See Figure 2.9.

2.2.5.3.1 THE LIST I/O TABLE

```
ORG @LIOTABLE
DW 0          ; place holder corresponding to CIN
DW LOUT       ; address of logical list output
DW LATN       ; address of logical list attention check
DW 0          ; place holder corresponding to CDIN
DW LDOUT      ; address of physical list device output
DW LDATN      ; address of physical list device attention check
DW LDINIT     ; address of physical list device initialize
PWRAPFLAG DB 0          ; enable (0) or disable (1) list device wrap logic
PNULLS    DB 3          ; list device null count + 1
PWIDTH   DB 72         ; list device carriage width
DS 1          ; must be provided for internal system use
```

The addresses in the table point to the actual routines. PNULLS AND PWIDTH may be changed at any time in either MDOS or BASIC by using the ASSIGN command.

2.2.5.3.2 LOGICAL LIST I/O (LOUT, LATN)

The logical output routines have been tailored to meet the requirements of MDOS and BASIC. They should not have to be rewritten.

2.2.5.3.3 PHYSICAL LIST DEVICE OUTPUT (LDOUT)

LDOUT is the physical output routine. Most standard interface boards can be accommodated by patching the output port addresses and the ready mask values into the supplied printer handler (see the listing in Appendix E). This generalized printer handler is in place after the system is booted. However, there are some cases where the generalized printer handler cannot be used. A couple of examples might be systems using an old BAUDOT teletype as a printer, or DIABLO which uses a non-standard ETX system. In these cases the physical output routine must do considerably more than just output when the print device is ready. For the BAUDOT teletype the physical output routine must convert from ASCII to BAUDOT before outputting.

The physical output handler must have the following characteristics to interface with the rest of the system:

- 1) The character to be output is passed to the physical output routine in the B register in ASCII.
- 2) The physical output routine can use (destroy) the A register.

- 3) All registers except A must be returned unchanged.
- 4) Some printers can signal when paper is out, the motor is off, or they are out of ribbon. The system supports printers which can signal a PRINTER ATTENTION condition.

If the printer needs attention, the physical output routine should return with the carry flag set (C). If your printer does not support a Printer attention condition, then always return with the carry clear (NC). The other status flags can be returned in any state.

2.2.5.3.4 PHYSICAL LIST DEVICE ATTENTION ROUTINE (LDATN)

LDATN is a routine which checks PRINTER ATTENTION on printers which support this condition.

If your printer does not support printer attention, then this routine can simply clear carry and return.

```
LDATN    XRA    A  
        RET
```

If your printer is capable of signaling printer attention, your LDATN routine must have the following characteristics:

- 1) The LDATN routine can use (destroy) the A register.
- 2) All registers except A must be returned unchanged.
- 3) If the printer needs attention the routine returns with the carry set (C), otherwise the carry is returned clear (NC). The other status flags can be returned in any state.

2.2.5.3.5 PHYSICAL LIST DEVICE INITIALIZE (LDINIT)

LDINIT is a routine which initializes the printer/interface device. Some devices are not programmable and cannot be software initialized in which case the LDINIT routine needs only clear carry and return.

```
LDINIT    XRA    A  
        RET
```

If you need a software initialization sequence, it must have the following characteristics:

- 1) The LDINIT routine can use (destroy) the A register.
- 2) All registers except A must be returned unchanged.
- 3) The carry flag must be returned clear (NC). The other status flags can be returned in any state.

2.2.6 CREATING YOUR SYSTEM DISKETTE

The Program Development system is shipped with a MASTER diskette and a SYSTEM diskette, which is a duplicate of the Master.

This is done as a convenience for people with a single drive system, and provides a simple, fast method for generating your first configured running SYSTEM diskette. However, to generate additional configured systems on a blank diskette requires a more detailed procedure if you have only one drive. This procedure is described in Section 2.2.8.

With a multiple drive system it is simple to make additional copies using either the DISKCOPY utility which makes a duplicate diskette on another drive, or the FILECOPY utility which copies a named file from one drive to another.

After the system has been configured to the input/output requirements of your equipment, you are ready to create your configured SYSTEM diskette.

- 1) Remove the MASTER diskette and keep it in a safe place. The MASTER diskette should never be re-written.
- 2) Insert the nonconfigured SYSTEM diskette in your drive (unit 0 on multiple drive systems).
- 3) Type FILES and a carriage return. A list of all the files on the system diskette will be displayed.
- 4) The first file entry on the diskette is DIR which is the directory. The second file entry is RES, which is the resident portion of the Program Development Software systems.
- 5) Type TYPE "RES" 0 and a carriage return. This changes the file type from read only and permanent to a normal data file. This must be done prior to removing the file from the directory in preparation to saving the new configured version.
- 6) Type SCRATCH "RES" and a carriage return. This removes the file from the directory.
- 7) Type SAVE "RES" 2B1 1598 3 and a carriage return. The unit select light will go on indicating that your configured RES file is being written onto the diskette.
- 8) When the system prompt ">" is printed again, the file has been saved. Type FILES and a carriage return. RES should be the second file entry.

RESTART AT 04E7_H

- 9) Due to the addition of the three commands, EDIT, RENUM and MERGE, the current BASIC is longer than BASICs before version 4.0. If there is no need to shorten BASIC, ignore this step. If you want the SYSTEM diskette to have a shortened version of BASIC, proceed to APPENDIX G, the FEATURES PROGRAM, which describes the procedure for shortening BASIC. When this procedure is completed, you are running a shortened BASIC. Do the following to save the shortened BASIC on the SYSTEM diskette.

In response to BASIC's READY prompt:

- a) Type OPEN 1 "BASIC":ATTRS(1)=8 and a carriage return.
- b) Type SAVE "BASIC" 16R1572, 16R5DFF and a carriage return.
- c) Type ATTRS(1)=16RF:CLOSE 1 and a carriage return.

The System diskette now has a copy of your personalized system. You may want to make a copy of your personalized system at this time as a backup. If you have a single system, go to Section 2.2.8. If you have a multiple drive system, type DISKCOPY and a carriage return. The DISKCOPY program will be brought in from the disk and type instructions for its use.

2.2.7 CREATING A BASIC ONLY SYSTEM DISKETTE

Some users may only want to program in BASIC or may be developing BASIC application program packages for sale. You can create a BASIC only system which will boot up directly to BASIC. The BASIC only system should not use the SYSTEM diskette provided, rather a new blank diskette should be used. This procedure should only be followed after you have configured your system as described in Section 2.2.4 and 2.2.5, and created a configured System disk as in Section 2.2.6.

- 1) Put a blank diskette in disk drive Ø.
- 2) Type INIT Ø and a carriage return.

The system responds ARE YOU SURE? This is done to help prevent accidentally initializing a diskette. The initialization process will destroy anything which was previously on the diskette. If you are sure the diskette you have in drive Ø is to be initialized,

Type Y and a carriage return.

When the prompt ">" is printed again, the diskette is initialized.

- 3) Remove the initialized diskette and put the SYSTEM diskette back into drive Ø.

- 4) Type BASIC and a carriage return.

BASIC will be loaded into memory and sign on with

MICROPOLIS BASIC VS X.X - COPYRIGHT 1978
READY

NOTE: It is possible to optionally remove features from BASIC before creating the BASIC only diskette. See Appendix G for details.

- 5) Remove the SYSTEM diskette and put the initialized diskette back into drive Ø.

- 6) Type SAVE "N:BASIC" 16R2B1, 16R5DFF and a carriage return.

BASIC will be written onto the initialized disk. When this is done the system will respond, READY.

- 7) Type OPEN 1 "BASIC":ATTRS(1)=3:CLOSE 1 and a carriage return.

This will set the attributes of BASIC to permanent and write protected. The diskette is now a valid BASIC only configured system disk.

If you want to copy the BASIC UTILITY program onto the BASIC only diskette, proceed as follows.

- 1) Put the original SYSTEM diskette into drive Ø.

- 2) Type LOAD "UTILITY" and a carriage return.

The UTILITY program will be loaded into BASIC's current program buffer and BASIC will respond, READY.

- 3) Remove the SYSTEM diskette and put the BASIC only diskette in drive Ø.

- 4) Type SAVE "N:UTILITY" and a carriage return.

The UTILITY program will be written on the BASIC disk.

Users with multiple drive systems may also wish to place the DISKCOPY utility on the BASIC disk. This can be done by using the FILECOPY capability in MDOS.

2.2.8 MAKING ADDITIONAL COPIES OF YOUR SYSTEM USING A SINGLE DRIVE

Micropolis provides two diskettes with your drive as described in Section 2.2.3 to simplify the initial system generation procedure, for single drive owners. However, after you have configured your system and created your SYSTEM diskette, it would be a good idea to make a back up copy of your configured SYSTEM diskette - especially if you have a nonstandard system which is harder to personalize.

BEFORE COPYING YOUR CONFIGURED SYSTEM, IT IS RECOMMENDED THAT YOU PUT A WRITE PROTECT TAB ON THE SYSTEM DISKETTE. IF NECESSARY, THIS WRITE PROTECT TAB CAN BE REMOVED AFTER THE COPYING PROCESS IS COMPLETE.

There are two utilities which can be used to make a copy of a configured system diskette:

- 1) The DISKCOPY utility can be used to make an exact duplicate image of a diskette. DISKCOPY can be used on a single drive system, though the procedure is somewhat more difficult than for multiple drives. Refer to chapter 4, section 4.8 for instruction on using DISKCOPY in this manner.
- 2) A special utility called COPYFILE is provided for the single drive owner. COPYFILE is similar to FILECOPY which is designed for multiple drives. COPYFILE makes it simpler for the single drive owner to backup disk files on another diskette. Refer to chapter 4, section 4.10 for instructions on using COPYFILE.

When using COPYFILE to backup your configured systems diskette, the following steps should be followed:

- a) Initialize a blank diskette by typing the command INIT 0 and a carriage return. The system prompts

ARE YOU SURE?

If you are, type a 'Y'. Any other response will cancel the command.

When the system prompts '>', the diskette is initialized.

- b) The RES file must be the first file to be copied on the newly initialized diskette. If any other file is copied before RES, the new diskette will not boot. Type:

COPYFILE "RES"

and a carriage return.

The COPYFILE program leads the user interactively through the copying process.

- c) The second file on the copy diskette should be MDOS. Type:

COPYFILE "MDOS"

and a carriage return.

- d) The rest of the system files can be copied in any order you wish.

III NORMAL OPERATION

3.0 INTRODUCTION

This section describes the day-to-day operating procedure for a user-configured system.

3.1 BOOTSTRAP PROCEDURE

- 1) Ensure that the diskette drive and controller are properly interconnected with your system and that the proper type of memory is configured and installed in your system. Apply power to the diskette drive and system.
- 2) Insert the configured SYSTEM diskette into the drive (drive 0 of dual drives) and load the diskette. On single drives, wait about 5 seconds to ensure the unit is up to speed.
- 3) Activate the bootstrap ROM on the controller. For Altair/Imsai type computers with a front panel, this is done by setting the address switches to the bootstrap address (F400H unless the controller base address has been changed), reset, examine, and run. For computers under control of a resident ROM monitor, follow the manufacturers instructions on starting program execution at a given address. Use the address of the bootstrap ROM (F400H unless the controller base address has been changed).

When the boot program is started, the unit select indicator on the drive will illuminate and the disk head will load with an audible "click".

The address lights on the computer front panel (if you have one) will flash the load process which will take 4 to 7 seconds.

The bootstrap program brings the system loader into RAM and it loads and starts the configured system. When this process is complete, the loaded system will output a sign-on message to your terminal. The MDOS system signs on with

MICROPOLIS MDOS VS. X.X - COPYRIGHT 1978

>

The BASIC system signs on with

MICROPOLIS BASIC VS. X.X - COPYRIGHT 1978
READY

- 4) Approximately 5 seconds after the load is complete, the drive will automatically be de-selected. This will be evidenced by the audible "click" of the head unloading and the unit select indicator will extinguish.

If the system has not signed on within 10 seconds, observe the unit select indicator. If the unit is still selected after about 20-30 seconds, the bootstrap program has not been able to read the loader into memory. Reset your system and remove the SYSTEM diskette. Inspect the diskette for any obvious damage or contamination. Re-load the diskette and retry the bootstrap operation.

If the system has not signed on but the unit select indicator has extinguished, the loader may not have been able to read the system into memory. Stop the system and examine location $\$39AH$ which contains the loader termination status. The status code should be one of the following:

47H (ASCII "G") - THE SYSTEM WAS LOADED WITHOUT AN ERROR - the problem is probably with your terminal or interface (ensure your terminal is on line).

55H (ASCII "U") - UNRECOVERABLE DISK ERROR

The system loader was unable to read the system from the diskette properly. Remove the diskette and inspect for obvious damage or contamination. Re-insert the diskette and retry the boot operation.

4DH (ASCII "M") - MEMORY ERROR - the system loader reads the system into its read buffer sector by sector and moves the data from each sector into the RAM area where it executes. During this process, the loader reads the data back to ensure that the move destination contains operable RAM. If the data read back does not compare, then the loader aborts with an "M" error. $\$19BH/\$19CH$ contains the RAM address at which the error occurred. Try to deposit/examine $\$, FFH, 5AH, A5H$ at the address which caused the error.

- a) If the examine always yields FFH, there is no RAM at that address (or a memory board failure which makes it appear so) or memory is protected.
- b) If the examined data does not match the data deposited at that location, you probably have a defective memory board or the memory is protected.
- c) If memory appears to be OK, retry the boot operation - if it fails again you may have noise or some similar transient memory error problem.

If the status code is not one of the above, the RAM at $\$0A0H - \$3A0H$ into which the loader is read may be defective, protected, or nonexistent.

3.2 OPERATING HINTS

- 1) The Micropolis flexible disk drive subsystem was designed to take every reasonable precaution to protect your diskettes and the data recorded on them. Examples of this care are the door interlock which prevents loading of the diskette until it is properly inserted, and the automatic 5 second deselect feature which relieves the head load pressure from the recording surface when the drive is not in use. Once the diskette is removed from the drive, it is your responsibility to exercise the same care in handling and storing the diskette to ensure its long service life. The following precautions are guidelines for proper handling:
 - a) The exposed recording surface is easily contaminated - do not touch or attempt to clean the surface. Do not smoke, eat or drink while handling the diskette. Whenever the diskette is removed from the drive, return it to its protective envelope.
 - b) The diskette is a thin oxide-coated plastic sheet which may be damaged if handled carelessly. Do not place heavy objects on the diskette; do not expose the diskette to excessive heat or sunlight; do not use rubber bands or paper clips on the diskette; do not bend or fold the diskette.
 - c) Do not write on the diskette labels with an erasable pencil: graphite particles may contaminate the diskette or it may be damaged by the force exerted in writing. A fiber-tip type of pen is recommended. Return the diskette to its envelope before writing on labels.
 - d) Information is recorded on the diskette as magnetized "spots". Exposure of the diskette to magnetic fields or ferromagnetic objects which may become magnetized may result in the loss of information.

If a diskette is damaged or contaminated it should be replaced. If a contaminated diskette is placed in the drive, the receiver and read/write head may become contaminated and ruin other diskettes.

- 2) The auto-deselect will ensure reasonable diskette life. But, as a rule you should unload the diskette whenever it is not going to be accessed for long periods of time. This will give added diskette life and prolong the life of the drive motor.
- 3) All diskettes used with the Micropolis subsystem must be initialized before they can be used. The required initialization can be performed by using the INIT command in the MDOS System or by using the BASIC UTILITY program provided on the MASTER diskette and described in Appendix B.

3.3 THE CONCEPT OF BACKUP

A key concept in the successful operation of any computer system is BACKUP. System failures are not a matter of probability, they are a matter of certainty. Failures may occur because of internal problems such as component failures or defects in media used in storage devices; or because of external sources such as power failure or line transients. Adoption of a sensible back-up scheme can minimize the inconvenience and expense of system failures.

In the context of microcomputers equipped with Micropolis flexible disk storage subsystems, backup means taking steps to ensure that your program and data files are not lost.

Protecting your programs is easy if the convention of master and working copies of programs is adopted as follows:

- 1) A master program diskette exists for the purpose of backup only. It is kept in a safe place and is only used when its contents are copied to a working diskette.
- 2) In day to day operations, programs are loaded and executed from working diskettes.
- 3) Never use the master diskette for program development. Copy its contents onto a working diskette and perform the program editing using the working diskette.
- 4) When editing program files, resave the program file periodically. In the event of a failure, the chance of losing all of a lengthy editing session is reduced.
- 5) When the editing of a program is complete, the diskette containing the source program should be saved as a temporary master. Debugging of the program should be performed using a copy of the temporary master. Subsequent program editing may be performed on the temporary master or a copy of it depending upon how extensive the previous editing was. The key concept is: If the only copy of a program under development is destroyed, it should be possible to recreate the latest version from previous masters and documentation of the changes made to the previous programs. (e.g., marked-up program listings) The extent to which this concept is extended depends upon weighing the inconvenience and time of making backup copies against the possible loss and inconvenience caused by a failure.
- 6) Once the program under development is stable and ready to be phased into operation, the temporary master becomes the new current master diskette. The previous master should be retained as a 'grandfather' backup master, until it is certain that the new program functions properly and there is no need to fall back to the previous program.

Listings of the programs on the master diskette should be saved in a safe place as further security.

Protection of data files is more difficult. The extent to which data files may be protected depends upon the application but the concept is the same. In a properly designed system it should be possible to recreate the current data base from a backup copy and a list of the changes which have occurred since backup copies of the files were made.

A static data base may be protected by procedures similar to those given for program protection.

A dynamic data base, such as the data base used in an interactive order entry or inventory control system, is difficult to protect. A properly designed system should include making frequent backup copies and saving the transactions against the data base in a separate file, preferably on a different device from the device on which the data base resides. If a failure occurs, the data base may be reconstructed except for transactions which may have been processing at the time of the failure. Many books and articles have been written concerning the design and security of data bases - consult them for an in depth discussion of the problems and solutions.

In systems which have only one disk drive, the backup process involves swapping diskettes in and out of the drive. Although the need for backup is independent of the number of drives in the system, in this context the time and inconvenience of the process may appear to overshadow the potential value. Micropolis has attempted to minimize this time and inconvenience by providing file and disk copy utility programs which support the single disk drive environment.

In systems which have two or more disk drives, the Micropolis DISKCOPY program provides the easiest means of making backup copies. The entire contents of a diskette may be copied onto another diskette in a few minutes.

IV MICROPOLIS DISKETTE OPERATING SYSTEM

4.0 INTRODUCTION TO MDOS

Micropolis Program Development Software consists of two systems, Micropolis BASIC which is discussed in Chapter V and the Micropolis Diskette Operating System (MDOS). MDOS consists of an executive program, a group of shared subroutines available to user programs, and an assembly language program development package.

The MDOS executive program implements an interactive command language that allows the user to control computer system operations from the system console. It provides commands for memory management, file management, I/O control and program control.

MDOS contains a very large group of subroutines which can be called from a user's application program. These subroutines provide for console and printer character I/O, buffered line I/O, text line parameter parsing, sequential and random file access, file management, physical diskette access, and 16 bit integer arithmetic. There are also a number of processor oriented utility subroutines.

Six application programs make up the package that supports assembly language program development. LINEEDIT facilitates the creation of source files. ASSM is a two pass 8080/8085 disk to disk assembler. SYMSAVE creates a source file of equate statements from a latent symbol table. FILECOPY is a utility for copying named files. DISKCOPY is a utility for making literal copies of an entire diskette. DEBUG provides facilities to locate and correct program bug's in machine language programs.

4.1 THE MDOS EXECUTIVE

The MDOS executive program implements an interactive command language that allows the operation of the microcomputer system to be controlled from the system console. When MDOS is loaded it signs on with the message

MICROPOLIS MDOS VS. X.X - COPYRIGHT 1978

>

It is then waiting for an executive statement to be entered.

4.1.1 ENTERING EXECUTIVE COMMANDS

Executive statements are entered by typing characters in sequence on the console keyboard. An executive statement is terminated by pressing the RETURN key. During the entry of a statement each character that is typed is echoed by the executive on the console display. Two control features may be used when entering a line.

- 1) Each time the RUBOUT key is pressed the next previously typed character will be deleted from the line. A backarrow is echoed to the terminal display for each character deleted.
- 2) Holding down the control key and typing X (CNTL/X) will cause all of the current line to be cancelled. A carriage return line feed combination is echoed to the terminal display. The executive is positioned to accept entry of a new line.

4.1.2 EXECUTIVE STATEMENT FORMAT

An executive statement has the following form:

[unit:]NAME [<ASCII> "<ASCII>" ... "<ASCII>" <hex> <hex> ... <hex>]

The NAME in an executive statement may be the name of an explicit command or the name of a disk file. MDOS has 23 explicit commands which are discussed in this section. Explicit command names are uppercase only and must not be preceded by any spaces. In addition, executable assembly language programs can be loaded into memory and run by entering their file NAME. This provides an implicit command capability that can be used to extend the executives vocabulary. Implicit command filenames can be up to ten ASCII characters in the code range 21 hex to 7E hex. Imbeded spaces, double quotes, backarrows, and rubouts are not allowed in implicit command filenames.

When an executive statement is entered the executive program searches its table of explicit command names for a match with the NAME that was input. If the NAME is found in the table of command names the statement is executed immediately. If the NAME is not an explicit command name, then the NAME is treated as an implicit command filename which must be

found on disk. Implicit command filenames may be prefixed by an optional unit number. This specifies the disk drive on which the NAMED file is to be found. If no unit number is specified, unit 0 is assumed. If a unit number is specified it must be separated from the first character of the NAME by a colon (:). The executive processes the implicit command filename by searching the directory of the specified disk drive for the file. If the file is found on the disk (and the file type is correct) the executive loads the program file into memory and transfers control, along with any parameters in the executive statement, to the program. If the executive does not find the file on the specified drive an error message is output to the console stream: COMMAND NOT FOUND. If the file is found on the disk but it is not an executable file an error message is output to the console stream: WRONG FILE TYPE. See the section on file type definitions for a detailed discussion of file types.

Executive statements consist of a NAME followed by parameters, as necessary. Parameters can be ASCII or numeric. There can be up to four ASCII parameters and up to four numeric parameters. There must be at least one space between the NAME and any parameters. All parameters must be separated from each other by at least one space. Entry of an executive statement with too many parameters of either type, or without the required spaces between fields will result in a SYNTAX ERROR.

ASCII parameters consist of from 0 to 10 ASCII characters in the code range 20H to 7EH except for 22H which is the double quote and 5FH and 7FH which are interpreted as backspace requests by the logical console input routines. ASCII parameters must be enclosed in double quotation marks. Entry of an executive statement with unbalanced quotation marks or illegal characters in an ASCII parameter will result in a SYNTAX ERROR.

ASCII parameters in executive statements are generally used to specify disk filenames. In this usage a unit number may be prefixed to the ASCII filename within the quotation marks by typing the unit number followed by a colon (:) followed by the filename. This indicates the disk drive unit on which the file is to be found. If no unit is specified, unit 0 is assumed. The digit of the unit specification and the colon are not included in the 10 character length restriction for ASCII parameters. For example, "DATAFILE01" and "1:DATAFILE01" are both valid ASCII parameters in an executive statement.

Numeric parameters in executive statements are unsigned hexadecimal values from 0 to FFFF. They represent such elements as memory addresses, filetypes, and databytes. Entry of a numeric parameter with a value greater than FFFF or with illegal characters will result in a SYNTAX ERROR.

4.1.3 CANCELLING AN OPERATION

All MDOS explicit commands and all application programs supplied by Micropolis can be cancelled in progress by holding down the control key and typing a C (CNTL/C) on the console keyboard. The operation will be terminated as soon as the CNTL/C is recognized and the message CANCELLED will be output to the console. Control is returned to the MDOS executive.

4.1.4 DISPLAY CONTROL

All MDOS explicit commands and all application programs supplied by Micropolis can be temporarily stopped in progress by holding down the control key and typing an S (CNTL/S). The process will pause upon recognition of the CNTL/S. Typing any key other than CNTL/S or CNTL/C will cause the process to resume. This function is very useful in controlling commands and programs that output displays at high speed. For example, the output of a DISP command may be viewed at reading speed by stopping and resuming the output as necessary.

4.1.5 EXPLICIT EXECUTIVE COMMANDS

Command syntax for each of the MDOS explicit commands is illustrated in this section with the aid of the following notation:

[] Option brackets. Any parameters enclosed between brackets are optional.

< > Symbol brackets. This space should be replaced by the item described.

4.1.5.1 THE COMP COMMAND

COMP <start addr. block1> <end addr. block1> <start addr. block2>

The COMP command compares two blocks of memory and displays address locations that do not compare and the data at those locations. Example:

```
>COMP 5000 500F 5010  
5004 01 09 5014
```

The block of memory from 5000 to 500F is compared with the block of memory from 5010 to 501F. One location fails to compare. Location 5004 contains 01 while the corresponding location, 5014, in the second block contains 09.

4.1.5.2 THE DUMP COMMAND

DUMP <start addr.> [<end addr.>]

The DUMP command outputs to the system console a formatted hex display of the contents of a block of memory. Sequential memory locations are shown 16 to a line with the memory address at the left margin. If the optional end address parameter is not entered, only one byte is displayed. Example:

```
>DUMP 5000 5011  
5000 50 C0 27 77 4F 33 4F CD 7D 9E 98 00 6A FD 82 90  
5010 77 2B
```

4.1.5.3 THE ENTR COMMAND

ENTR <start addr.>

The ENTR command allows data to be entered into memory directly from the console device. Example:

```
>ENTR 7000  
>78 89  
6F/
```

Three bytes were entered starting at location 7000 hex. These were 78 at 7000, 89 at 7001, and 6F at location 7002.

Typing in an ENTR command places the executive in a special enter mode. While in the enter mode each line of values that is typed is entered into memory when the RETURN key is pressed. Until the RETURN key is pressed the standard backspacing and CNTL/X tools are available for line correction. The last value on the last line must be followed by a slash (/) to properly terminate the enter mode. Entry of a illegal hex value in any line will also cause termination of the enter mode with the message SYNTAX ERROR.

4.1.5.4 THE FILL COMMAND

```
FILL <start addr.> <end addr.> <byte>
```

The FILL command fills a block of memory with a specified byte.
Example:

```
>FILL 7000 8000 9
```

Each byte of memory in the block from 7000 to 8000 is changed to a 09 by this command.

4.1.5.5 THE MOVE COMMAND

```
MOVE <source addr. start> <source addr. end> <dest. addr. start>
```

The MOVE command copies the source block of memory to the destination block. The source block is not changed. The destination block is changed to be an exact copy of the source block. Example:

```
>MOVE 3000 4000 7000
```

Each byte in the memory block from 3000 to 4000 is copied into the corresponding position in the memory block from 7000 to 8000.

4.1.5.6 THE SEAR COMMAND

```
SEAR <start addr.> <end addr.> <byte>
```

The SEAR command searches a block of memory for all occurrences of the specified byte and displays all locations with a match. Example:

```
>SEAR 3000 3020 9F  
3004 9F  
3018 9F
```

The block of memory from 3000 to 3020 is searched for all occurrences of a 9F. Location 3004 and location 3018 both contain 9F. No other locations in the block contain 9F.

4.1.5.7 THE SEARN COMMAND

SEARN <start addr.> <end addr.> <byte>

The SEARN command searches a block of memory for all non-occurrences of a specified byte and displays all locations that do not match. Example:

```
>SEARN 3000 3010 67  
3002 09 67  
3006 76 67
```

The block of memory from 3000 to 3010 is searched for all non-matches with the mask 67. Location 3002 contained a 9 rather than a 67, and 3006 contained a 76 rather than a 67.

4.1.5.8 THE CREATE COMMAND

CREATE "[unit:]<filename>" [<file type>]

The CREATE command creates a new file in the directory of the diskette in the specified unit and allocates the initial track for the file. If no unit is specified, unit 0 is assumed. The second parameter optionally gives the file a TYPE designation. If no type is specified the type is defaulted to 0.

4.1.5.9 THE DISP COMMAND

DISP "[unit:]<filename>" [<record number>]

The DISP command outputs a formatted hex display of the data contents of a file to the system console. The unit number indicates the disk drive on which the file is to be found. If no unit is specified, unit 0 is assumed. The optional record number indicates on which record in the file the display is to begin. If no record number is specified, record 1 is assumed.

Each record is displayed with a header line that contains the record number, the address in memory where the record is to be loaded, and the number of data bytes in the record. Data lines follow the record header. Each data line has up to sixteen data bytes preceded by the index position in the record of the first data byte on that line.

```
>DISP "1:TEST" 29  
0029 3C00 0022  
00 12 2A BD 76 8F ED 54 41 89 00 00 82 BC CC 76 89  
10 78 88 3B BB 88 54 58 56 90 88 32 31 30 0D 00 00  
20 89 55  
002A 3C80 0003  
00 FF FF FF  
002B 3F00 0009  
00 45 43 4B 4C 31 37 38 0D 00  
002C 2B00 0000  
END-FILE
```

The first line of the display shows the record number 29, the load address 3C00, and the length of the record 22 bytes (all in hex). The header line is followed by three lines which display the data in record 29. Each data line starts with the index position of the first byte in the line. It is followed by two spaces and then the data.

The next header is for record 2A which has a load address of 3C80 and contains 03 bytes of data.

Record 2B has a load address 3F00 and contains 09 bytes of data.

The last header is for record 2C which has a load address of 2B00 and a record length of 0. If the file is an executable object file (like ASSM for example), the address in the zero length sector is the execution address of the file. LOADING stops when the zero length sector is read. If the file is a run type which is being implicitly loaded and run, program control is transferred to the execution address.

4.1.5.10 THE FILES COMMAND

FILES [<unit>]

The FILES command outputs a formatted display of the file information in a diskette directory to the system console. The unit number indicates which disk drive directory is to be displayed. If no unit is specified, unit \emptyset is assumed. Example:

```
>FILES 1
DIR      03  0000
RES      03  0013
MDOS     0F  001C
LINEEDIT 15  000C
ASSM     15  0010
SYMSAVE  15  0003
FILECOPY 15  0003
DISKCOPY  0F  0009
BASIC    0F  004B
```

The files on drive one are displayed on the console. The left column contains the filename, the second column is the file type, and the third column contains the number of sectors the file uses. All numbers are in hex.

4.1.5.11 THE FREE COMMAND

FREE [<unit>]

The FREE command outputs to the system console the number of tracks left unallocated (free) on a diskette. The unit number indicates which disk drive. If no unit is specified, unit \emptyset is assumed. Example:

```
>FREE 1
003B
```

The diskette on drive one has 3B tracks available to be allocated.

4.1.5.12 THE SCRATCH COMMAND

SCRATCH "[unit:]<filename>"

The SCRATCH command removes a named file from the directory of a diskette and returns its allocated tracks to available status. Disk drive \emptyset is assumed if no unit is specified.

Note: Some files cannot be SCRATCHed without first changing the file TYPE (see 4.1.5.9 and 4.2.3).

4.1.5.13 THE LOAD COMMAND

The LOAD command loads (reads) a named file from a diskette into the computers memory and then returns control to the MDOS executive. If no unit number is specified, the file is expected to be found on unit 0.

The LOAD command can be used in conjunction with two categories of files, OBJECT files and DATA files. The specific nature of the load that is performed depends on the category of the specified file to be loaded. The process of LOADING an OBJECT file is described in 4.1.5.13.1. The process of LOADING a DATA file is described in 4.1.5.13.2.

The LOAD command can NOT be used to load a file in the OVERLAY category. An OVERLAY file is defined as any file with a file type value in the range 0C - 0F hex (see Section 4.2.3). An attempt to LOAD an OVERLAY file results in the message WRONG FILE TYPE. OVERLAY files are not LOADable because they generally imply the replacement of the MDOS module and require immediate execution. Control cannot be returned to the MDOS executive and must be transferred immediately to the newly overlayed program module. If there is a necessity to LOAD an OVERLAY file into a memory area which does not conflict with MDOS, this can be done by changing the file type to an OBJECT type and then using an offset load per Section 4.1.5.13.1.

4.1.5.13.1 THE LOAD COMMAND FOR OBJECT FILES

An OBJECT file is defined as any file with a file type value in the range 08 - 0B hex or 14 - 1B hex. These ranges include ASSM object files, BASIC 'save memory' files, executable system files, and executable user files (see Section 4.2.3).

The format of the LOAD command for OBJECT files is:

```
LOAD "[unit:] <filename>" [<start addr.>]
```

OBJECT files are LOADED by using the address and length information in the header of each record of the file (see Section 4.2.4). This is called a 'scatter load' because it permits records in the file to be loaded into non-contiguous portions of memory depending on the associated addresses. The LOAD is terminated when the first 0 length record in the file is encountered.

If the optional start address is not specified in the LOAD command, then the load of an OBJECT file proceeds according to the following example.

The OBJECT file to be loaded is "TEST".

```
DISP "TEST"
0000 2B00 0005
00 31 32 33 34 35
0001 2C00 0004
00 54 45 53 54
0002 2B00 0000
END-FILE
```

Typing LOAD "TEST" loads two text strings into memory. The string "12345" in record 0 is loaded starting at 2B00 hex for five bytes. The test string "TEST" in record 1 is loaded starting at 2C00 hex for four bytes. The last record contains a zero length sector which terminates the load of an OBJECT type file. For an executable file the zero length sector contains the run address which in this case is 2B00 hex. This file, however, could not be a run file as it stands as there is no executable code.

If the load address of the first record is less than 2B00 hex, the message LOAD ADDRESS ERROR is displayed because file may not be loaded beneath the MDOS application area.

If the optional start-address is specified in the LOAD command, then the first record of the file is loaded starting at the specified address. The load address in the record header of the first record is subtracted from the start-address to produce an offset. When the records following the first record of the file are loaded, the calculated offset is added to the load address in the record header and the record is loaded starting at the calculated address. This is called an 'offset scatter load'.

Using the file TEST in the example above, typing LOAD "TEST" 5000 loads the string "12345" starting at memory location 5000 hex for five bytes. The offset is calculated by subtracting the load address in the header of the first record from the start-address. $5000 - 2B00 = 2500$ hex. The string "TEST" is loaded starting at 5100 hex for four bytes. The load address in the header of the second record, 2C00 has the offset 2500 hex added to it and the result is the offset-load address.

If the optional start-address is less than 2B00 the message LOAD ADDRESS ERROR is displayed.

4.1.5.13.2 THE LOAD COMMAND FOR DATA FILES

Any file which is not an OBJECT file and not an OVERLAY file is treated as a DATA file by the LOAD command. DATA files thereby include file type values in the ranges 0-7, 10-13 hex, and 1C-FF hex. These ranges cover MDOS and BASIC DATA files, ASSM and LINEEDIT source files, BASIC program files and all of the unassigned file types (see Section 4.2.3).

The format of the LOAD command for DATA files is:

```
LOAD "[unit:] <filename>" <start addr.>
```

The start address parameter is mandatory. If a start address is not specified a SYNTAX ERROR message will be displayed. If the start address is less than 2B00 HEX a LOAD ADDRESS ERROR will result. This prevents accidental destruction of the operating system.

Data is loaded starting at the specified address and continuing until the number of records in the file as shown in the directory have been loaded. The data is loaded into memory sequentially and contiguously. Only the number of data bytes in each record are loaded. The LOAD command does not pad records of less than 256 bytes. If a file were loaded at location 3000 and the first record had only 4 data bytes in it, then the first data byte from the next record would be loaded at location 3004. Records with zero length are skipped over. The load address in the sector header (see Section 4.2.4) has no meaning when doing a data LOAD.

4.1.4.14 THE SAVE COMMAND

```
SAVE "[unit:]<filename>" <start addr.> <end addr.> [<file type>]  
[<exec. addr.>]
```

The SAVE command saves (writes) a new file to a diskette from a block of memory. The file is written sequentially from the memory start address through the memory end address into full sequential records. If no unit number is specified, the file is written to unit 0. If a file type is not specified the file type will be zero. If an execution address is not specified, the execution address of the file will be set to the start address of the memory block. Note that the type and execution address parameters are position dependent such that if an execution address is specified then a file type must also be present. Example:

```
>SAVE "1:NEWFILE" 2B00 3700 0 3000
```

A file is created on the diskette in drive one with the name NEWFILE and the memory block from 2B00 to 3700 is written to that file. The file is given a type of 0 and the execution address saved with the file is 3000. If no execution address had been specified then 2B00 would be saved as the execution address.

4.1.5.15 THE RENAME COMMAND

```
RENAME "[unit:]<filename>" "<new name>"
```

The RENAME command changes the name of a diskette file to a specified new name. If no unit number is specified, the file to be renamed is expected to be found on unit 0. Example:

```
>RENAME "1:OLDFILE" "NEWFILE"
```

The file named OLDFILE on the diskette in drive one is changed to NEWFILE on the diskette in drive one. The file type is unchanged by the renaming process.

4.1.5.16 THE TYPE COMMAND

TYPE "[unit:]<filename>" <type>

The TYPE command changes the type designation of a specified file. The type designation is a single hex byte. A definition of file types is given in Section 4.2. Example:

>TYPE "1:PROGRAMX" 15

The type of the file PROGRAMX on disk drive one is changed to a value of 15.

4.1.5.17 THE APP COMMAND

APP [<>ASCII> "<>ASCII>"..."<>ASCII>"] [<hex> <hex>...<hex>]

The APP command transfers program control from the MDOS executive to the start of the MDOS applications area at 2B00 hex. It expects a valid executable program to be in the applications area with its entry point at the beginning. Up to four ASCII parameters and four hex parameters can be passed to the program. For example, if you are doing several assemblies, the assembler need only be read into memory once from diskette as it does not change itself in the process of assembling a program. After it is once in memory the APP command can be used to communicate with the assembler. Example:

>APP "1:SOURCE" "OBJECT" "P"

If the assembler were already in memory, the above example would transfer control and the necessary parameters to the program and the assembler would assemble the source file called SOURCE from drive one; produce an object file on drive zero called OBJECT; and output a paginated listing on the print device.

The APP command functions like the EXEC command in that it PUSHes the address of the operating systems warm start entry point onto the system stack. Therefore if the program in the applications area does not provide its own stack, a RET would return control to the operating system.

4.1.5.18 THE ASSIGN COMMAND

ASSIGN <device #> <logical stream mask> [<width> <null count>]

The ASSIGN command is a dual purpose command which provides the ability to specify the connections of physical output print devices to logical output streams and the values for carriage width and nullcount of the referenced physical device. The physical device number must be 1 or 2. The logical stream mask must be a 0,1,2, or 3. The device width and nullcount must be numeric values in the range 1 to FF hex. The width and nullcount parameters are optional. If width or nullcount are not included, the values corresponding to the referenced physical device

are not changed. If only the device width is included, then the nullcount is left unchanged. However, if a nullcount is specified then the width must be present as a place holder even if it is the same. If the ASSIGN command contains only three parameters the third is always the width.

Logical output stream number one consists of all output generated by system messages, keyboard echoing and the output from any explicit executive command. Logical output stream number two consists of all output generated by LISTP and PRINTP commands in the line editor, and by all listings in the assembler. The logical stream mask can be set to a three to represent both logical output streams one and two, or to a zero indicating that the device is to receive no output.

Physical device number one represents the display element of the keyboard display device that is configured as the system console (see Section 2.2.4.1 on terminal configuration). Physical device number two represents the hard copy print device which is configured as the system printer (see Section 2.2.4.3).

The output of a logical stream is directed to all physical devices which are assigned to it. A physical device may be assigned to one, both, or no logical streams. The ASSIGN command cancels any previous assignment of the specified device.

In its initialized state the terminal is assigned to stream one only, and the printer is assigned to stream two only. This state can be restored by executing:

```
>ASSIGN 1 1  
>ASSIGN 2 2
```

When the console and printer devices are configured, each device has a carriage width and nullcount parameter associated with it. These values may be changed by specifying optional third and fourth parameters in an appropriate ASSIGN command. The width parameter determines the maximum number of characters on each line for the given device. When a line is output that is longer than this value an autowrap feature is activated and a carriage return and line feed is inserted at the appropriate point so that the logical line is continued on the next device line. The width can be changed on a given device by repeating the current assignment with the new width parameter. For example, if the console were currently assigned to stream one with a width of 80 characters (decimal), it could be changed to a width of 72 characters (decimal) as follows:

```
>ASSIGN 1 1 48
```

72 decimal is 48 hex. This width assignment will stay in effect until the width is specifically reassigned, or until the system is rebooted.

The nullcount may have to be changed to accommodate unbuffered character serial devices which may lose characters while the carriage is being returned. The nullcount value is one greater than the actual number of

output nulls (ie. 1 will output no nulls). For example, if the printer were currently assigned to stream two at 132 characters per line and no nulls (nullcount=1), the number of output nulls could be changed to five with the following command:

```
>ASSIGN 2 2 84 6
```

132 decimal is 84, and 6 will result in five nulls being output after a carriage return.

Because the MDOS executive language has been designed to be interactive it depends on the availability of a display device for system messages, keyboard echoing, and display of command results. Therefore an interlock is built into the system to ensure that stream one always has at least one device assigned to it. If an ASSIGN command violates this condition, then physical device one is automatically assigned to stream one as part of the assignment being processed. Additionally if the print device supports a printer attention condition (out of paper, motor off, etc.) the system will force the assignment to an initial state (ASSIGN 1 1, ASSIGN 2 2) if the printer signals that it needs attention. This ensures that the attention message will be output to the console.

4.1.5.19 THE EXEC COMMAND

```
EXEC <address>
```

The EXEC command transfers processor control directly to the specified memory address. It expects a valid program to begin at that address. The address of the operating systems warm start entry point is PUSHed onto the 8080's hardware stack by the EXEC command. Therefore, if the executed program does not set its own stack, a final RET in the program will return to the operating system. This feature allows subroutines to be exercised separate of the rest of a system under development.

4.1.5.20 THE MATH COMMAND

```
MATH <hex number> <hex number>
```

The MATH command performs 16 bit integer math functions on the two specified hex numbers. It displays the sum, difference, product, quotient, and modulus. Example:

```
>MATH 4 5  
0009 FFFF 0014 0000 0004
```

The results are displayed from left to right: 4+5=9 ; 4-5=FFFF ; 4*5=14 ; 4/5=0 (integer division) and a remainder (modulus) of 4.

4.1.5.21 PROMPT "<ASCII>"

The PROMPT command sets the executive prompt string to the value of the ASCII string. The string can be up to ten characters long. Spaces are

not allowed. The prompt is initially > when the system is configured.
Example:

```
>PROMPT "**
**
```

The prompt is changed from > to a **

4.1.5.22 THE INIT COMMAND

INIT <unit>

The INIT command initializes a diskette in the specified drive. The drive unit number must be specified. The INIT command formats the diskette by writing an empty block with the correct track and sector identification on every sector of the diskette and reading each sector to verify the media. It creates a blank directory and places a system loader on the diskette. The INIT command essentially cleans the diskette of any data previously on the diskette and prepares it for new use. Accidental use of the INIT command could destroy the entire content of a diskette. Therefore, the system provides an interlock on this command. After the command is entered, the system prompts ARE YOU SURE?. It waits for a 'Y' or 'N' response to indicate yes or no. An 'N' cancels the command without doing any damage. Example:

```
INIT 1
ARE YOU SURE?
```

The diskette on drive one will be initialized if a 'Y' is typed. All other replies will result in the command being canceled. Control returns to the executive.

4.2 MDOS DISK FILE I/O

MDOS implements a powerful and efficient method for storage and retrieval of files on diskettes compatible with Micropolis disk subsystems. Track 0 of each diskette contains a directory of the files on that diskette. Each directory entry holds the name, protection attributes, type, length and starting location for one file. Track 0 also contains a track map index that lists all unassigned tracks and all tracks assigned to each file in the order of assignment. Files are stored on the remaining tracks of the diskette using a track indexed architecture that allows files to grow or shrink dynamically. Files may be accessed sequentially by byte or record and directly (randomly) by record or byte within record.

4.2.1 TRACK INDEXED FILE STORAGE

The track indexed file storage scheme defines one track as the minimum disk space consumed by a file. The maximum storage assignable to one file is all tracks on the diskette (35 on MOD I subsystems and 77 on MOD II subsystems), except the directory track 0. When MDOS creates a new file it assigns one track to that file. Additional file space is assigned to the file one track at a time as needed. Files are contiguous within a track but not necessarily from track to track. If a file is shortened, unused tracks are returned to available status. When a file is deleted (scratched), all of its assigned tracks are freed for reassignment.

Maintenance of the track map in the track indexed scheme operates as follows. Whenever a file is opened for access MDOS reads the track map from that file's diskette into main memory. Any record in the file may then be accessed with only one disk seek by appropriate reference through the track map. File access operations that cause the file to be extended or shortened by one track also cause the track map to be immediately updated in memory and on disk. When the file is closed its directory entry is rewritten to reflect any changes in the file's size or status.

4.2.2 FILE NAMES

File names consist of from 0 to 10 ASCII characters in the code range 20H to 7EH except for 22H which is the double quote and 5FH and 7FH which are interpreted as backspace requests by the logical console input routines.

A unit number may be prefixed to the filename by typing the unit number followed by a colon (:) followed by the filename. This indicates the disk drive unit on which the file is to be found. If no unit is specified, unit 0 is assumed. The digit of the unit specification and the colon are not included in the 10 character length restriction for ASCII parameters. For example, DATAFILE01 and 1:DATAFILE01 are both valid file names.

If the file name is to be an implicit command in an executive statement there are additional restrictions that apply. The file name may not start with a blank. It may have no imbedded blanks and it may not exist in the MDOS explicit command table.

Files that are to be shared with BASIC must have valid BASIC file names. BASIC file names can be up to 10 characters long and use the ASCII characters from 20 hex through 5A hex except the colon (3A hex). This should be kept in mind when creating file names for MDOS. The BASIC file names are a subset of the MDOS file names and some incompatibility can occur if care is not used.

4.2.3 FILE PROTECTION AND TYPE DEFINITION

MDOS provides two forms of file protection. A file can be write protected or a file can be delete protected. MDOS also allows files to be classified as to unique information content by assigning a type designation. A files' access codes and type designation are combined in one byte of the files' directory entry. The first two least significant bits of the file type byte are bit encoded and specify file access restrictions. The access codes are as follows:

BIT	
1 0	

0 0	A normal read/write file
0 1	A normal read only file
1 0	A permanent read/write file
1 1	A permanent read only file

A normal file can be read, written, and deleted from the diskette by using the SCRATCH command (Section 4.1.2.5). A read only file can be read or SCRATCHed but it cannot be written into. A permanent file can be read or written but it cannot be SCRATCHed. A permanent read only file can be read but it cannot be written into or SCRATCHed. Attempts to SCRATCH a permanent file will result in the message PERM FILE. Attempts to write into a read only file will result in the message READ ONLY FILE. The TYPE command may be used to change the access codes of a file if necessary.

Note that these access code safeguards are software features that will only protect a file as long as the operating system has not been damaged. Diskettes may be physically write protected by placing a write protect tab over the slot in the upper right hand edge of the diskette. This causes the write electronics in Micropolis disk subsystems to be disabled when that diskette is loaded in a disk drive.

The most significant six bits of the file type byte specify the type of file. This allows 64 different classifications of files each having four access codes.

The codes 0 through 7F hex are reserved for present and future system usage and should not be assigned other meanings by the user. The codes from 80 to FF hex are available to the user and are not used by the system.

The executive, the assembler, and the editor check file types when called upon to load, save, or resave a file. If the file type is not correct the function will not take place. A table of file types follows:

TYPE CODE <u>IN HEX</u>	DESCRIPTION
00-03	MDOS & BASIC DATA FILES
04-07	EDITOR/ASSEMBLER SOURCE FILES
08-0B	ASSEMBLER OBJECT & BASIC 'SAVE MEMORY' FILES
0C-0F	EXECUTABLE OVERLAY FILES
10-13	BASIC PROGRAM FILES
14-17	EXECUTABLE SYSTEM FILES
18-1B	EXECUTABLE USER FILES
1C-7F	RESERVED FOR FUTURE EXPANSION
80-FF	AVAILABLE FOR USER DEFINITION

The line editor produces type 4 files. It can load type 4,5,6, and 7 files. The assembler will only assemble type 4,5,6, and 7 files. It produces type 8 files.

Executable system files and user files may be loaded with the load command. Any attempt to load a file below the application program area will result in a LOAD ADDRESS ERROR. Executable overlay files may be loaded below the application program area by typing the file name as an implicit executive command. Any attempt to implicitly load a file that is not an executable file will result in the message WRONG FILE TYPE.

It is not possible to load an overlay file without beginning its execution. However, the entry point of the overlay could contain a jump to the MDOS warmstart address. This would return control to MDOS immediately after the overlay file was loaded, provided that the file did not overlay any functional MDOS code.

4.2.4 FILE AND RECORD STRUCTURE

An MDOS file consists of a group of related records stored on a diskette. The group is given a filename and type designation as described above. These are stored in the file directory on track 0 of the diskette.

Each record of an MDOS file begins with a two byte memory address followed by a two byte length indicator. The remainder of the record consists of 0 to 256 data bytes. The memory address tells MDOS where in memory to load the data from that record. The length indicator tells MDOS how many valid data bytes are in the record. A record needs a minimum block of 4 bytes and a maximum block of 260 bytes to be properly stored.

The records of a MDOS file are stored on the sectors of a diskette, one for one. Micropolis disk subsystems write a physical sector that is 268 bytes long. The first 8 bytes of the sector are used for control purposes strictly by the operating system. The remaining 260 bytes are available for a record. Short records, including 0 length (empty) records are possible. If a particular record has less than 256 data bytes the remainder of the sector is not used. However, the record may be expanded at any time by rewriting the sector to make use of the unused bytes.

The object program file that corresponds to the following assembly language program serves to illustrate the MDOS file and record structure.

ADDR	B1	B2	B3	E	LINE#	LABEL	OPCODE	OPERAND
0000					1000	START	ORG	4000H
4000	21	00	70		2000		LXI	H,7000H
4003					3000	DATA	DS	10H
4013	00				4000	BYTE	DB	0
4014					5000	DATA1	DS	10H
4024	01				6000	BYTE1	DB	1
4025	C3	25	40		7000	BEGIN	JMP	\$
4028					8000		END	BEGIN

The first record of the object file has 4000 hex in the memory address bytes in Intel low/high format. The record length bytes contain 0003, indicating that the record has only three bytes of data. The three data bytes are 21 00 70. This record is written on the disk as one sector. The second record of the object file has a memory address of 4013 and a length of 0001, one byte of data 00. This record is also stored on the disk as one sector. The third record has a memory address of 4024 and a length of 0004, four bytes of data 01 C3 25 40. This record is stored on the disk as one sector. A fourth record is written that has a memory address 4025 and a length of 0000. This empty record marks the end of the object file and its memory address holds the execution address specified in the END statement.

The structure of this object file is standard for all MDOS executable or memory load files. The file is allocated one entire track on the disk. It contains eight data bytes spread across 3 sectors. The 4th and last sector contains no data. Its memory address field holds the file execution address. Given an executable file type, the records of this file could be loaded into memory at 4000, 4013 and 4024 by typing its name to the executive. Direct processor control would transfer to 4025 to begin program execution. This type of file is called a scatter loadable file because it can be loaded non-contiguously into main memory.

Note: The number of records in each MDOS file is included in the directory entry for that file. This determines the end of file for data files. Data files do not require a zero length record to mark their end because there is no execution address for a data file. The special zero length record is used with files that load into a range of memory and may require an associated execution address. For these files the zero length record is included in the record count in the files' directory entry.

4.2.5 FILE ACCESS METHODS

MDOS contains shared subroutines that allow user application programs to access diskette files sequentially by byte or record and directly (randomly) by record and byte within record.

A file may be written sequentially by writing a byte at a time and incrementing the index position. The system buffers the bytes written

until a full 256 byte record is constructed and then writes it to the next sector in the file. The file space is automatically extended as necessary. A file may also be written sequentially by repeatedly writing blocks of data up to 256 bytes in length as one record and then incrementing the record position to the next record. A file written in this manner may have records of varying length up to 256 bytes.

A file may be read sequentially by reading a byte at a time and incrementing the index position until the end of file is reached. If the file contains any short records the unused bytes at the end of the sectors of those records will be automatically skipped by this byte sequential access. A file may also be read sequentially a record at a time by starting at the first record, reading the record length and then reading that number of bytes as a block, incrementing the record position to the next record, and repeating the process until the end of file is reached.

A specific record in a file may be accessed by setting the index position directly to the start of that record. The record may then be read or written either a byte at a time or as a block of bytes. A specific byte in a directly accessed record may be read or written by first setting the index position directly to that byte in the record. These techniques facilitate the spot updating of a file.

4.2.6 COMPATIBILITY BETWEEN MDOS AND BASIC FILES

BASIC file names are a subset of MDOS file names. Therefore all BASIC files can be handled by the MDOS file name parsing logic, but not all MDOS file name can be handled by BASIC. Refer to the Section 4.2.2 on FILE NAMES for a complete discussion.

BASIC data files contain records of from zero to 250 bytes of data. The file and record structure is the same as that used by MDOS as discussed in Section 4.2.4. The two bytes at the start of the record which hold the length of the record can never be greater than 250 if the file is to be used by a BASIC program as a data file. BASIC will output an error message to the console stream and stop the program if the record length is greater than 250. MDOS can create BASIC readable files as follows:

```
1000 * GET DATA TO BE WRITTEN INTO A BASIC COMPATABLE FILE
2000 START      MVI      E,250
3000 GET       CALL     GETDATE
3500          JC      EXIT           ;CLOSE FILE & EXIT
4000          CALL     @WTINXPOSI
5000          DCR     E
6000          JNZ     GET
7000          CALL     @INCRECPOS
8000          JMP     START
```

This partial program illustrates a method for writing 250 byte records. For these records to be meaningful to BASIC, the data must be seven bit ASCII with the proper BASIC string delimiters (refer to the STRING statement in the chapter on BASIC). The subroutine GETDATE is the user's data acquisition routine which returns the carry flag set when the process is done. @WTINXPOSI and @INCRECPOS are MDOS subroutines which are documented in Section 4.3.3. The method shown corresponds to the process for writing a file sequentially by record as described in Section 4.2.5.

4.3 MDOS SHARED SUBROUTINES

MDOS provides the applications development programmer with many useful subroutines that can be accessed directly from an applications program. These subroutines provide for console and printer character I/O, buffered line I/O, text line parameter parsing, sequential and random file access, file management, physical diskette access, and 16 bit integer arithmetic. There are also a number of processor oriented utility subroutines.

When you write an assembly language program, these subroutines can be referenced by name; e.g. CALL @HLADDA. The PDS MASTER diskette contains two files named SYSQ1 and SYSQ2. These are editor compatible source files that contain the names of all of the MDOS shared subroutines equated to their entry addresses. Application programs that reference these routines by name should include the SYSQ1 and SYSQ2 files in their assembly by using the assembler LINK pseudo-op, described in detail in Section 4.5.

The following sections specify what arguments each subroutines expects, what arguments each subroutine returns, and how it functions.

4.3.1 CONSOLE AND PRINTER INPUT/OUTPUT SUBROUTINES

Micropolis Program Development Software packages perform input and output through the following subroutines. These routines link the system with the device handlers described in Chapter II under configuring for supported devices.

The device handler routines start with a vector table whose address is @CIOTABLE for the console, and @LIOTABLE for the printer. The routines in this section enter the drivers by indirectly accessing these tables using @CONSOLEADDR, and @LISTADDR which are buffers that hold pointers to the actual location of @CIOTABLE and @LIOTABLE. By changing the two bytes at locations @CONSOLEADDR or @LISTADDR the user can have special purpose drivers in memory at the same time as the standard drivers.

4.3.1.1 @CIN - CONSOLE INPUT

The @CIN routine waits for input from the system console. It strips parity and changes ASCII codes 5F (backarrow) and 7F (rubout) into 08 (backspace). It returns the input character (7 bit ASCII) in the B register, with the carry flag clear (NC). It preserves the HL, DE, and C registers.

4.3.1.2 @COUT - CONSOLE OUTPUT

The @COUT routines waits until the console stream is ready and then outputs a character. It changes carriage returns into a carriage return followed by the number of nulls associated with the device attached to the console stream. It changes ASCII code 08 hex (backspace) into a 5F (backarrow). If the wrap logic for the device assigned to the console stream is enabled a line feed and a carriage return nulls sequence will be output when the

number of characters on the line equals the width. Refer to the ASSIGN command in the MDOS executive. It expects the character (7 bit ASCII) in the B register. It returns the carry flag set (C) if a printer attention condition occurs, and sets the assignment to ASSIGN 1 1, and ASSIGN 2 2. Refer to the ASSIGN command in the MDOS executive. It preserves the HL, DE, and BC registers.

4.3.1.3 @CBRK - CONSOLE CHECK BREAK

The @CBRK routine checks the console device for the input of a cancel (control C), or a pause (control S). It returns the zero flag set (Z) and the CANCELED message code in the A register if a CONTROL C (03) is input. It preserves the HL, DE, and C registers. On pause (control S) the routine loops, waiting for another character to be input. Entry of any character other than control S will terminate the pause and return to the caller.

4.3.1.4 @CDIN - CONSOLE DEVICE INPUT

The @CDIN routine waits for input from the console device. It returns the character (8 bits including parity) in the B register, with the carry flag clear (NC). It preserves the DE, HL, and C registers.

4.3.1.5 @CDOUT - CONSOLE DEVICE OUTPUT

The @CDOUT routine waits until the console device is ready to receive a byte and then outputs it. It expects the byte for output in the B register. It preserves the DE, HL, and BC registers. It returns the carry flag clear (NC).

4.3.1.6 @CDBRK - CONSOLE DEVICE BREAK CHECK

The @CDBRK routine checks the console input ready status. If an input is ready it gets the input. Otherwise it returns immediately. It returns the zero flag set (Z) and the input character (8 bits including parity) in the B register if there was an input. It preserves the DE, HL, and C registers. If there was no input the @CDBRK routine returns the zero flag clear (NZ), and the B register is unchanged.

4.3.1.7 @CDINIT - CONSOLE DEVICE INITIALIZATION

The @CDINIT routine initializes the console interface device. It preserves the HL, DE, and BC registers. It returns the carry flag clear (NC).

4.3.1.8 @LOUT - LIST OUTPUT

The @LOUT routine waits until the list stream is ready to receive and then outputs a character. It changes carriage returns into a carriage return followed by the number of nulls associated with the device attached to the list stream. It changes ASCII code 08 hex (backspace) into a 5F (backarrow). If the wrap logic for the device assigned to the list stream is enabled a line feed and a carriage return nulls sequence will be output

when the number of characters on the line equals the width. Refer to the ASSIGN command in the MDOS executive. It expects the character (7 bit ASCII) in the B register. It returns the carry flag set (C) if a printer attention condition occurs, and sets the assignment to ASSIGN 1 1, and ASSIGN 2 2. Refer to the ASSIGN command in the MDOS executive. It preserves the HL, DE, and BC registers.

4.3.1.9 @LATN - LIST ATTENTION

The @LATN routine checks the list stream for a printer attention condition. It returns the carry flag set (C) if a printer attention condition occurs, and sets the assignment to ASSIGN 1 1, and ASSIGN 2 2. Refer to the ASSIGN command in the MDOS executive. It preserves the HL, DE, and BC registers.

4.3.1.10 @LDOUT - LIST DEVICE OUTPUT

The @LDOUT routine waits until the list device is ready to receive a byte and then outputs it. It expects the byte for output in the B register. It preserves the DE, HL, and BC registers. It returns the carry flag set (C) if a printer attention occurs.

4.3.1.11 @LDATN - LIST DEVICE ATTENTION

The @LDATN routine checks the list device for a printer attention condition. It returns the carry flag set (C) if a printer attention condition occurs. It preserves the HL, DE, and BC registers.

4.3.1.12 @LDINIT - LIST DEVICE INITIALIZATION

The @LDINIT routine initializes the list device. It preserves the HL, DE, and BC registers. It returns the carry flag clear (NC).

4.3.1.13 @CCRLF - CONSOLE LINE FEED CARRIAGE RETURN

The @CCRLF routine outputs a line feed carriage return and nulls to the console stream. It returns the carry flag set (C) if a printer attention condition occurs, and changes the assignment to ASSIGN 1 1, and ASSIGN 2 2. Refer to the ASSIGN command in the MDOS executive. It preserves the HL, DE, and BC registers.

4.3.1.14 @LCRLF - LIST LINE FEED CARRIAGE RETURN

The @LCRLF routine outputs a line feed carriage return and nulls to the list output stream. It returns the carry flag set (C) if a printer attention condition occurs, and changes the assignment to ASSIGN 1 1, and ASSIGN 2 2. Refer to the ASSIGN command in the MDOS executive. It preserves the HL, DE, and BC registers.

4.3.1.15 @ASSIGN - ASSIGN

The @ASSIGN routine assigns the physical device to specified logical stream(s) and sets the width and nullcount associated with the device. It expects the physical device number in the E register, the logical stream mask in the D

register, the width in the C register, the nullcount (nulls+1) in the B register, and the number of parameters passed in the H register. No registers are preserved. (Refer to the ASSIGN command in the executive for a detailed discussion of physical device assignment to logical output streams).

4.3.1.16 @CILINE - CONSOLE INPUT LINE

The @CILINE routine outputs a specified prompt message to the console and then buffers up to 132 characters of input text from the console device. It provides the standard backspace (rubout) and line cancel (CNTL/X) controls during the line entry process. The text line input is terminated by a carriage return. (Note: The carriage return is not echoed to the console). It expects the address of a string of text to be output as a prompt in the HL registers. The message pointed to must be properly terminated with a byte code of \emptyset through 1F hex or the high order eight bit of the last byte set. It returns the input line in @INBUFF, and the number of input characters including the terminating carriage return in the B register. It preserves the HL, DE, and C registers. Any control characters input during the line entry process are echoed to the console stream but not entered into @INBUFF.

4.3.1.17 @HEXOUT - HEXADECIMAL OUTPUT

The @HEXOUT routine converts an unsigned 8 bit binary value in the A register to a hex number and outputs the number to the console. It returns the carry flag set (C) if a printer attention condition occurs, and changes the assignment to ASSIGN 1 1, and ASSIGN 2 2. Refer to the ASSIGN command in the MDOS executive. It preserves the HL, DE, and C registers.

4.3.1.18 @HEXADDOUT - HEXADECIMAL ADDRESS OUTPUT

The @HEXADDOUT routine converts an unsigned 16 bit binary value in the HL registers to a hex number and outputs the number to the console followed by one space character. It returns the carry flag set (C) if a printer attention condition occurs, and changes the assignment to ASSIGN 1 1, and ASSIGN 2 2. Refer to the ASSIGN command in the MDOS executive. It preserves the HL, DE, and C registers.

4.3.1.19 @HEXOUTSPC - HEXADECIMAL OUTPUT WITH SPACE

The @HEXOUTSPC routine converts an unsigned 8 bit binary value in the A register to a hex number and outputs the number to the console followed by one space character. It returns the carry flag set (C) if a printer attention condition occurs, and changes the assignment to ASSIGN 1 1, and ASSIGN 2 2. Refer to the ASSIGN command in the MDOS executive. It preserves the HL, DE, and C registers.

4.3.1.20 @SPACEOUT - SPACE OUTPUT

The @SPACEOUT routine outputs a space (20 hex) to the console stream. It returns the carry flag set (C) if a printer attention condition occurs, and changes the assignment to ASSIGN 1 1, and ASSIGN 2 2. Refer to the ASSIGN command in the MDOS executive. It preserves the HL, DE, and C registers.

4.3.1.21 @NLINEOUT - NEW LINE OUTPUT

The @NLINEOUT routine outputs a carriage return line feed and a line of text to the console stream. It expects the address of the beginning of the text line in the HL registers. The message pointed to must be properly terminated with a byte code in the range 0 through 1F hex or the high order eighth bit of the last byte set. It returns the carry flag clear (NC) in all cases. It preserves the HL, DE, and C registers.

4.3.1.22 @LINEOUT - LINE OUTPUT

The @LINEOUT routine outputs a line of text to the console stream. It expects the address of the beginning of the text line in the HL registers. The message pointed to must be properly terminated with a byte code in the range 0 through 1F hex or the high order eighth bit of the last byte set. It returns the carry flag clear (NC) in all cases. It preserves the HL, DE, and C registers.

4.3.2 TEXT LINE PARSING SUBROUTINES

The following routines are used by the system to parse input command lines for the MDOS executive. After the command has been entered into the input buffer using @CILINE, the @SCAN routine is used to locate the first space after the command, and @SKIPSPACE skips to the first non-space character. Then the @PARAM routine separates the command parameters into buffers according to their type. @PARAM makes use of @SCAN, @SKIPSPACE, and @AHEXTBIN to do its job. After the parameter types have been separated, the address of the beginning of the input buffer is placed into @MASKADDR and the @SEAR routine searches the MDOS command table for a match. If the command is valid, the @SEAR routine returns with the zero flag clear and @LHLI will get the function from the table, which in this case is an address. Control is passed to the command routine with a PCHL instruction. The command routine can retrieve the parameters from the appropriate buffers with LHLD instructions.

The user can use these routines to parse applications program input lines using similar logic.

4.3.2.1 @PARAM - PARAMETER

The @PARAM routine parses a text line. It separates parameters into ASCII, numeric and unit numbers. It counts the number of occurrences of each parameter type and places the count and each parameter in a separate buffer.

It expects the start address of the text to be parsed in the HL registers.

It returns ASCII parameters in @ASCBUFF0 through @ASCBUFF3.

It returns unit numbers in @DRIVEN0 through @DRIVEN3.

It returns binary (numeric) parameters in @BBUFF0 through @BBUFF3.

It returns the number of ASCII parameters in @NASCPAR.

It returns the number of unit number parameters in @NDRVPAR.

It returns the number of binary parameters in @NBINPAR.

It returns the carry flag clear (NC) and the end of line address in the HL registers if there were no errors.

It preserves the DE and BC registers.

If a parameter is in error the carry flag is set (C), the SYNTAX ERROR code is in the A register, and the location where the error occurred is returned in the HL registers.

4.3.2.2 @SKIPSPACE - SKIP SPACES

The @SKIPSPACE routine skips spaces in a text line.

It expects the text line's start address in the HL register.

It returns the address in the HL registers of the first non-space character.

If the character is a control character the carry flag is set (C).

It preserves the DE and BC registers.

4.3.2.3 @SCAN - SCAN

The @SCAN routine scans a text line for the first occurrence of a specified character.

It expects the text line's starting address in the HL registers and the mask character in the C register.

It returns the address in the HL register where the match occurred and the number of characters passed over in the B register.

The carry flag is set (C) if the mask character was not found prior to a control character.

It preserves the DE and C registers.

4.3.2.4 @SEAR - SEARCH

The @SEAR routine searches a table of argument-function pairs and returns the address of the function associated with the argument. The last character of the argument has the most significant bit set high. For example, an ASCII A is 41 hex. If the most significant bit is set high it is a C1 hex.

The argument is immediately followed by its function. The arguments can be variable length but the functions must all be the same length. The end of the table is marked by a Ø following the last function.

It expects the table's start address in the HL register and the argument masks' starting address in @MASKADDR. The argument mask string must be terminated by a space or control character. It expects the A register to contain the size (number of bytes) of the functions in the table.

It returns the zero flag clear (NZ) and the address of the start of the argument's function in the HL register.

The zero flag is set (Z) if the argument was not in the table. In this case the HL registers contain the end of table address, ie. the address of the Ø after the last function.

It preserves the DE and BC registers.

4.3.2.5 @AHEXTBIN - ASCII HEX TO BINARY

The @AHEXTBIN routine converts a text string of unsigned hexadecimal digits represented in ASCII code into a binary number. The string can be one to four digits in length. It must end with a space or control character.

It expects the string's start address in the DE registers.

It returns a 16 bit binary number in the HL registers.

It returns the number of digits in the number in the B register.

It returns the DE registers pointing to the space or control character that ends the text string.

It preserves the C register.

If the number is greater than four digits long or not a hex value, the routine returns the carry flag clear (NC) and the illegal character's address in the DE registers.

4.3.3 THE FILE ACCESS ROUTINES

The file access subroutines implement the MDOS file access methods described in Section 4.2.5. They allow an open disk file to be accessed sequentially by byte or record and directly (randomly) by record and byte within record.

Before a file can be accessed it must be opened. To open a named file on a specified disk unit the file must be assigned a logical file number and a filebuffer. MDOS supports simultaneously open files numbered from Ø through 7. It makes available two resident filebuffers. Additional filebuffers must be allocated in the memory space of the application program. Each filebuffer requires 288 bytes of memory.

When a file is opened the first record of the file is read into its filebuffer. The record in the file buffer of a file at any given time is called the current record of that file. Associated with the current record of each open file is an update flag. Any access that modifies the content of the current record will cause the update flag to be set. If the update flag is set, any access that leads to the current record being replaced by a new record will first cause the current record with the modified content to be rewritten in place (updated) to the disk file. If the update flag is not set, no update takes place before a new record is read. Invoking a new record resets the update flag.

The current record of each open file has a record length which is written with the record as described in Section 4.2.4. Its value may vary from 0 to 256. A 0 length record indicates an empty record that still occupies one physical sector on the diskette. A 256 byte record is a full record that cannot be extended.

The index position of the current record is a logical pointer that marks the next byte in the record to be accessed. The value of the index position ranges from 0 to 255. However, the index position may never be greater than the length in a particular record. An index position of 0 indicates that the next byte to be accessed is the first byte in a record. An index position of 255 indicates that the next byte to be accessed is the last byte in a full record.

If the index position in the current record is less than the current record length, then it points to a valid byte position within the record. That byte may be read or rewritten. If the index position is equal to the current record length, then it points to the end of record (EOR) position which is the first non valid byte position in a non full record. The EOR position may be written but it may not be read.

Reading from the end of record position updates the current record to disk as necessary and the next record in the file becomes the current record. The index position is set to 0 and the data is read from this position. This allows files containing a mixture of non full records to be read sequentially by byte.

If the end of record position is written to, the length of the current record is increased by one and the position just written becomes a valid byte position. This allows data to be added to the end of a record extending it up to its maximum length of 256 bytes. Note, however, that incrementing the index position when it already has a value of 255 updates the current record to disk as necessary and the next record of the file becomes the current record. The index position will be set to 0.

A new file may be written sequentially by byte by repeatedly writing to the index position and incrementing the index position. This will produce a file of full records with the possible exception of the last record. The system automatically extends the amount of disk space allocated to a file when enough new records are written to require another track.

The current record of each open file also has a record position number associated with it. The record position number specifies which record the current record is in the file. The record position number may be set or incremented. Setting the record position updates the current record to disk as necessary and the specified record from the file is read and becomes the current record. This provides a mechanism for direct (random) access to any record in a file. Incrementing the record position number updates the current record to disk as necessary and the next record in the file is read and becomes the current record. This function can be used to sequentially write a file of short/mixed length records.

When processing of a file is complete, the file must be closed. Closing a file updates the current record to disk as necessary and frees the logical file number and the filebuffer for subsequent reallocation.

4.3.3.1 @CREATE - CREATE

The @CREATE routine creates a file of a specified type on a specified disk unit. The created file has one track allocated to it and one empty (\emptyset length) record written to it. It is left open and ready for access with the index position set to \emptyset and the empty record as the current record.

It expects the file number in the B register and the disk unit number in the C register and the filename in @ASCIIIBUFF.

It expects the file type in the D register and the start address of the file buffer in the HL registers.

If the routine detects an error it returns the carry flag set (C) and the error message code in the A register.

It preserves the HL, DE, and BC registers.

4.3.3.2 @GFILESTAT - GET FILE STATUS

The @GFILESTAT routine checks the open/closed status of a file.

It expects the file number in the B register.

If the file is closed it returns with the zero flag set (Z) and the "FILE NOT OPEN" message code in the A register.

It preserves the HL, DE, and BC registers.

If the routine detects an error it returns the carry flag set (C) and the error message code in the A register.

4.3.3.3 @DIRSEARCH - DIRECTORY SEARCH

The @DIRSEARCH routine reads the directory of a specified disk unit to determine if a specified file exists.

It expects the unit number in the C register and the file name in @ASCIIIBUFF.

It returns the zero flag clear (NZ) and the "FILE NOT FOUND" message code in the A register if the file is not in the directory.

It preserves the HL, DE, and BC registers.

If the routine detects an error it returns the carry flag set (C) and the error message code in the A register.

4.3.3.4 @OPENFILE - OPEN A FILE

The @OPENFILE routine opens a file for processing. It assigns a specified logical file number and filebuffer to the file.

It expects the file name in @ASCIIIBUFF, the file number in the B register, and the drive number in the C register.

It expects the address of the file buffer in the HL registers.

It preserves the HL, DE, and BC registers.

If the routine detects an error it returns the carry flag set (C) and the error message code in the A register.

4.3.3.5 @CLOSEFILE - CLOSE A FILE

The @CLOSEFILE routine updates the current record to disk as necessary and frees the logical file number and the filebuffer for subsequent reallocation.

It expects the file number in the B register.

It preserves the HL, DE, and BC registers.

If the routine detects an error it returns the carry flag set (C) and the error message code in the A register.

4.3.3.6 @RFILEINF - READ FILE INFORMATION

The @RFILEINF routine gets the disk unit number, the number of records in the file, the file type, and the record position number of the current record.

It expects the file number in the B register.

It returns the file type in the B register and the disk unit number in the C register.

It returns the number of records in the file plus one in the DE registers.

It returns the record position number of the current record in the HL registers.

If the routine detects an error it returns the carry flag set (C) and the error message code in the A register.

4.3.3.7 @SINXTRS - SET INDEX POSITION TO RECORD START

The @SINXTRS routine updates the current record to disk as necessary and reads a specified record which becomes the current record. The index position is set to Ø.

It expects the file number in the B register and the record number in the HL registers.

It preserves the HL, DE, BC registers.

If the routine detects an error it returns the carry flag set (C) and the error message code in the A register.

4.3.3.8 @RRECORDLEN - READ RECORD LENGTH

The @RRECORDLEN routine gets the length of the current record in a file.

It expects the file number in the B register.

It returns the length of the record in the HL registers.

It preserves the DE and BC registers.

If the routine detects an error it returns the carry flag set (C) and the error message code in the A register.

4.3.3.9 @RINXPOS - READ INDEX POSITION

The @RINXPOS routine gets the index position of the current record of a file.

It expects the file number in the B register.

It returns the index position in the C register.

It preserves the HL, DE, B registers.

If the routine detects an error it returns the carry flag set (C) and the error message code in the A register.

4.3.3.10 @SINXPOS - SET INDEX POSITION

The @SINXPOS routine sets the index position within the current record in a file.

It expects the file number in the B register and the index position in the C register.

It preserves the HL, DE, BC registers.

If the routine detects an error it returns the carry flag set (C) and the error message code in the A register.

4.3.3.11 @INCINX - INCREMENT INDEX POSITION

The @INCINX routine increments the index position in the current record of a file. If the increment would result in a value greater than the current record length, then the current record is updated to disk as necessary and the next record of the file becomes the current record and the index position is set to \emptyset .

It expects the file number in the B register.

It returns the zero flag set (Z) if the index position is in the same record.

It returns the zero flag clear (NZ) if the index position is in a new record.

It preserves the HL, DE, BC registers.

If the routine detects an error it returns the carry flag set (C) and the error message code in the A register.

4.3.3.12 @RFINXPOS - READ FROM INDEX POSITION

The @RFINXPOS routine reads the data byte pointed to by the index position in the current record of a file. If the index position is at the EOR position the current record is updated to disk as necessary and the next record of the file becomes the current record. The index position is set to \emptyset and the data is read from this position.

It expects the file number in the B register.

It returns the data in the C register.

It returns the zero flag set (Z) if the data is from the same record.

It returns the zero flag clear (NZ) if the data is from a new record.

It preserves the HL, DE, B registers.

If the routine detects an error it returns the carry flag set (C) and the error message code in the A register.

4.3.3.13 @RFINXPOSI - READ FROM INDEX POSITION AND INCREMENT INDEX

The @RFINXPOSI routine reads the data byte pointed to by the index position in the current record of a file and then increments the index position. If the original index position is at the EOR position, the current record is updated to disk as necessary and the next record of the file becomes the current record. The index position is set to \emptyset and the data is read from that position. Then the increment takes place. If the increment would result in a value greater than the current record length, the current record is updated to disk as necessary and the next record from the file becomes the current record. The index position is set to \emptyset in that case.

It expects the file number in B.

It returns the data in the C register.

It returns the zero flag set (Z) if the data is from the same record.

It returns the zero flag clear (NZ) if the data is from a new record.

It preserves the HL, DE, BC registers.

If the routine detects an error it returns the carry flag set (C) and the error message code in the A register.

4.3.3.14 @WTINXPOS - WRITE TO INDEX POSITION

The @WTINXPOS routine writes to the index position in the current record of a file. If the index position is the EOR position the record length is extended by one.

It expects the data in the C register, and the filenumber in the B register.

It preserves the HL, DE, BC registers.

If the routine detects an error it returns the carry flag set (C) and the error message code in the A register.

4.3.3.15 @WTINXPOSI - WRITE TO INDEX POSITION AND INCREMENT INDEX

The @WTINXPOSI routine writes to the index position in the current record and then increments the index position. If the index position is the EOR position the current record length is extended by one. If the increment would result in an index greater than 255, then the current record is updated to disk as necessary and the next record in the file becomes the current record. The index position is set to \emptyset in this case.

It expects the data in the C register, and the filenumber in the B register.

It returns the zero flag set (Z) if the index position remains on the same record as before the write.

It returns the zero flag clear (NZ) if the index position has been incremented to a new record.

It preserves the HL, DE, BC registers.

If the routine detects an error it returns the carry flag set (C) and the error message code in the A register.

4.3.3.16 @LOADDATA - LOAD DATA

The @LOADDATA routine loads a block of data into memory starting from the index position in the current record and continuing for a specified number of bytes. It advances the index position like a repeated sequence of reads and increments.

It expects the file number in the B register.

It expects the start address of the memory block in the HL registers.

It expects the block size in the DE registers.

It returns the zero flag set (Z) if the last byte read is from the same record as the first byte.

It returns the zero flag clear (NZ) if the last byte read is from a new record.

After a call to @LOADDATA the buffer @MEMORYPNTR contains the address of the memory byte immediately after the last memory byte loaded. For example, if 5 bytes are loaded into 4000H through 4004H, then @MEMORYPNTR contains the address 4005H in standard low-high format. This is useful in cases where the number of bytes loaded is less than the number of bytes requested because an end of file is encountered during the @LOADDATA.

It preserves the HL, DE, BC registers.

If the routine detects an error it returns the carry flag set (C) and the error message code in the A register.

4.3.3.17 @SAVEDATA - SAVE DATA

The @SAVEDATA routine writes a block of memory to a file starting at the index position of the current record and continuing for a specified number of bytes. It advances the index position like a repeated sequence of writes and increments.

It expects the file number in the B register.

It expects the start address of the memory block in the HL registers.

It expects the number of bytes in the memory block in the DE registers.

It returns the zero flag set (Z) if the index position remains on the same record as before the write.

It returns the zero flag clear (NZ) if the index position has been incremented to a new record.

After a call to @SAVEDATA the buffer @MEMORYPNTR contains the address of the memory byte immediately after the last memory byte saved. For example, if 5 bytes are saved from 4000H to 4004H then @MEMORYPNTR contains 4005H in standard low-high format. This is useful in cases where a DISK FULL condition causes less bytes to be saved than are requested in the call to @SAVEDATA.

It preserves the HL, DE, BC registers.

If the routine detects an error it returns the carry flag set (C) and the error message code in the A register.

4.3.3.18 @DFINXPOSTEOR - DELETE FROM INDEX POSITION TO END OF RECORD

The @DFINXPOSTEOR routine deletes from the index position to the end of the current record by making the record length equal to the value of the index position.

It expects the file number in the B register.

It preserves the HL, DE, BC registers.

If the routine detects an error it returns the carry flag set (C) and the error message code in the A register.

4.3.3.19 @DFINXPOS - DELETE FROM INDEX POSITION TO END OF FILE

The @DFINXPOS routine deletes from the index position to the end of the file by making the number of records in the file equal to the record position number of the current record and the current record length equal to the value of the index position. Any tracks no longer required by the file due to the deletion are freed for subsequent reallocation to other files.

It expects the file number in the B register.

It preserves the HL, DE, BC registers.

If the routine detects an error it returns the carry flag set (C) and the error message code in the A register.

4.3.3.20 @INCRECPOS - INCREMENT RECORD POSITION

The @INCRECPOS routine updates the current record to disk as necessary, reads in the next record which becomes the current record and sets the index position to 0. If the current record is the last record in the file, the file is automatically extended by one record.

It expects the file number in the B register.

It preserves the HL, DE, BC registers.

If the routine detects an error it returns the carry flag set (C) and the error message code in the A register.

4.3.4 FILE MANAGEMENT SUBROUTINES

In addition to accessing named files on the disk it becomes necessary on occasion to perform housekeeping functions such as removing old files, changing file types and names, and determining the amount of space left on a disk for additional files. These functions are available as executive commands, and are also provided as subroutines that may be used directly by applications programs.

4.3.4.1 @FREE - FREE

The @FREE routine returns the number of tracks left on a diskette that are free and available for allocation to a file.

It expects the unit number in the C register.

It returns the number of free tracks in the HL registers.

If the routine detects an error it returns the carry flag set (C) and the error message code in the A register.

4.3.4.2 @RENAME - RENAME

The @RENAME routine renames a file on a diskette.

It expects the file number in the B register.

It expects the new name in @ASCIIIBUFF.

It preserves the HL, DE, and BC registers.

If the routine detects an error it returns the carry flag set (C) and the error message code in the A register.

4.3.4.3 @TYPE - FILE TYPE

The @TYPE routine changes the type (attributes) of a file. See Section 4.2.3 for type definitions.

It expects the file number in the B register.

It expects the new file type in the C register.

It preserves the HL, DE, and BC registers.

If the routine detects an error it returns the carry flag set (C) and the error message code in the A register.

4.3.4.4 @SCRATCH - SCRATCH A FILE

The @SCRATCH routine deletes a specified file from a specified disk unit.

It expects the unit number in the C register.

It expects the file name in @ASCIIIBUFF.

It preserves the HL, DE, and BC registers.

If the routine detects an error it returns the carry flag set (C) and the error message code in the A register.

4.3.5 PHYSICAL DISK ACCESS ROUTINES

The physical disk access subroutines are the most primitive level of access provided within the MDOS context. They allow a diskette to be treated as a collection of logical blocks independent of the MDOS file system and provide access to a specified logical block on a specified track of a diskette.

Micropolis MOD I disk subsystems write 35 tracks on one side of a diskette. The MOD II subsystems write 77 tracks on one side of a diskette. A track in either subsystem is divided into 16 sectors each of which contains 268 bytes. Tracks numbered 0 through 34 or 76 are written concentrically inward toward the center of the diskette. The physical sectors on a track are numbered from 0 through 15.

Diskettes initialized by and formatted for use with MDOS have the track number written in the first byte and the physical sector number written in the second byte of each sector of a track. These bytes are maintained exclusively by the operating system.

The remaining 266 bytes of a sector are accessible as a logical block by the MDOS physical disk access routines. In order to enhance access time to multiple blocks, MDOS maps logically sequential blocks onto the physical sectors of a track in a staggered pattern as shown.

LOGICAL BLOCKS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
PHYSICAL SECTORS	0	2	4	6	8	10	12	14	1	3	5	7	9	11	13	15

The physical disk access routines automatically access the correct physical sector that corresponds to the logical block that is specified. If it is necessary to access the sectors of a track in true physically sequential order, the application program must use the table above to unmap the sectors. For example, to access sector 0 followed by sector 1 the program would have to specify logical block 1 followed by logical block 9.

Note that the record structure of MDOS files as detailed in Section 4.2.4 must be preserved if the physical disk access routines are used to operate on such records.

4.3.5.1 @GETASEC - GET A SECTOR

The @GETASEC routine gets (reads) a sector from a specified disk unit into a specified memory buffer given the track and logical block numbers.

It expects the unit number in the C register.

It expects the track number in the D register and the logical block number in the E register.

It expects the address in the HL register of the start of a 266 byte buffer.

If the routine detects an error it returns the carry flag set (C) and the error message code in the A register.

4.3.5.2 @PUTASEC - PUT A SECTOR

The @PUTASEC routine puts (writes) from a specified memory buffer to a sector on a specified disk unit given the track and logical block numbers. Before it writes the sector it reads the header information of the target sector-2 to verify that it will be writing on the correct sector. This is called a preread. It requires that the preread sector be readable.

It expects the unit number in the C register.

It expects the track number in the D register and the logical block number in the E register.

It expects the address in the HL register of the beginning of a 266 byte buffer.

If the routine detects an error it returns the carry flag set (C) and the error message code in the A register.

4.3.5.3 @WRITESECTOR - WRITE A SECTOR

The @WRITESECTOR routine writes from a specified memory buffer to a sector on a specified disk unit given the track number and logical block number. It does not do a preread before writing. This allows a sector to be written on an uninitialized track or a track on which the preread sector is unreadable.

It expects the unit number in the C register.

It expects the track number in the D register and the logical block number in the E register.

It expects the address in the HL registers of the beginning of a 266 byte buffer.

If the routine detects an error it returns the carry flag set (C) and the error message code in the A register.

4.3.5.4 @VERIFYSECTOR - VERIFY A SECTOR

The @VERIFYSECTOR routine verifies the validity of the header information and checksum of a sector on a specified disk unit.

It expects the unit number in the C register.

It expects the track number in the D register and the logical block number in the E register.

If the routine detects an error it returns the carry flag set (C) and the error message code in the A register.

4.3.5.5 @SEEKTRACK - SEEK TO A TRACK

The @SEEKTRACK routine moves the read/write head to a specified track on a specified disk unit.

It expects the unit number in the C register.

It expects the track number in the D register.

If the routine detects an error it returns the carry flag set (C) and the error message code in the A register.

4.3.5.6 @RESTOREDISK - RESTORE THE READ/WRITE HEAD

The @RESTOREDISK routine positions the read/write head to track zero of a specified disk unit.

It expects the unit number in the C register.

If the routine detects an error it returns the carry flag set (C) and the error message code in the A register.

4.3.6 PROCESSOR ORIENTED UTILITY ROUTINES

These subroutines effectively extend the instruction set of the 8080 to provide for some commonly required operations.

When parentheses enclose an item in the following subsections, this indicates the contents of the memory location specified by the value within the parentheses. For example, HL=(HL) means that the HL register pair is replaced with the bytes at the address in HL and HL+1. If the HL registers contain the address 4000 hex, and at location 4000 there is a 01, and at location 4001 there is a 02, then the HL register would be replaced by 0201 hex. The low byte goes into L and the high byte into H.

4.3.6.1 @HLADDA - ADD A TO HL

The @HLADDA routine adds the unsigned 8 bit value in the A register to the unsigned 16 bit value in the HL registers.

It expects a value in the HL, and the A registers.

It returns HL=HL+A.

It preserves the DE and BC registers.

4.3.6.2 @INXM - INCREMENT MEMORY

The @INXM routine increments a memory pair pointed to by the HL registers. It is similar to an INR M instruction but it operates on a byte pair (16 bits) in memory.

It expects the address of the memory pair in the HL registers.

It preserves the DE and BC registers and the PSW.

4.3.6.3 @LHLINDEXED - LOAD HL INDIRECT INDEXED

The @LHLINDEXED routine loads the HL registers indirect from the location pointed to by the HL registers indexed by the A register.

It expects the address in the HL registers, and the index in the A register.

It returns HL=(HL+2*A).

It preserves the DE and BC registers.

4.3.6.4 @LHLI - LOAD HL INDIRECT

The @LHLI routine loads the HL registers with the content of the byte pair pointed to by the HL registers.

It expects an address in the HL registers.

It returns $HL = (HL)$.

It preserves the BC and DE registers.

4.3.6.5 @TRANSDHC - TRANSFER FROM DE TO HL FOR A COUNT OF C

The @TRANSDHC routine copies a memory block pointed to by the DE registers to a memory block pointed to by the HL registers for a length in the C register. It begins at the start of each block and working to the end.

It expects the start address of the source block in the DE registers and the start address of the destination block in the HL registers and the number of bytes to copy in the C register.

It returns $(HL+\emptyset\dots+C) = (DE+\emptyset\dots+C)$.

It preserves the B register.

4.3.6.6 @TRANSDHBC - TRANSFER FROM DE TO HL FOR A COUNT OF BC

The @TRANSDHBC routine copies a memory block pointed to by the DE registers to a memory block pointed to by the HL registers for a length in the BC registers. It begins at the start of each block and works to the end.

It expects the start address of the source block in the DE registers and the start address of the destination block in the HL registers and the number of bytes to copy in the BC registers.

It returns $(HL+\emptyset\dots+BC) = (DE+\emptyset\dots+BC)$.

4.3.6.7 @TRANSDHBCR - TRANSFER FROM DE TO HL FOR A COUNT OF BC REVERSE

The @TRANSDHBCR routine copies a memory block pointed to by the DE registers to a memory block pointed to by the HL registers for a length in the BC registers. It begins at the end of each block and working to the beginning.

It expects the start address of the source block in the DE registers and the start address of the destination block in the HL registers and the number of bytes to copy in the BC registers.

It returns $(HL+BC\dots+\emptyset) = (DE+BC\dots+\emptyset)$.

4.3.6.8 @TRANSFILENAME - TRANSFER A FILENAME

The @TRANSFILENAME routine copies a filename from one of the ASCII buffers (@ASCBUFF0 through @ASCBUFF3) to the @ASCIIIBUFF.

It expects the @ASCBUFF number (ie. 0 to 3) in the C register.

It preserves the HL, DE, and BC registers.

4.3.6.9 @FILLZER - FILL ZEROES

The @FILLZER routine fills a block of memory up to 256 bytes in length with zeros.

It expects the start address of the memory block in the HL registers and the number of bytes to fill in the B register.

It preserves the DE and C registers.

4.3.6.10 @FILLSPC - FILL SPACES

The @FILLSPC routine fills a block of memory up to 256 bytes in length with spaces (hex 20).

It expects the start address of the memory block in the HL registers and the number of bytes to fill in the B register.

It preserves the DE and C registers.

4.3.6.11 @FILLA - FILL FROM THE A REGISTER

The @FILLA routine fills a block of memory up to 256 bytes in length with the value specified in the A register.

It expects the start address of the memory block in the HL registers, the number of bytes to fill in the B register, and a fill value in the A register.

It preserves the DE and C registers.

4.3.6.12 @COMPARE - COMPARE HL TO DE

The @COMPARE routine compares the value in the HL registers to the value in the DE registers.

It expects a value in the DE register and the value to compare it to in the HL register. The forms are like an 8080 CMP B instruction where DE is analogous to the A register and HL is analogous to the B register.

It returns the following sense:

DE = HL	zero flag set (Z),	carry flag clear (NC)
DE > HL	zero flag clear (NZ),	carry flag clear (NC)
DE < HL	zero flag clear (NZ),	carry flag set (C)
DE >= HL	zero flag any state,	carry flag clear (NC)

It preserves the HL, DE, and BC registers.

4.3.7 EXTENDED 8080 INTEGER ARITHMETIC (16 BITS)

These routines extend the capability of the 8080 to allow 16 bit unsigned integer addition, subtraction, multiplication, and division (quotient, and modulus).

The result of all of these routines is returned in the BC registers. The HL and DE registers are preserved. With the exception of @DEDIVHL and @DEMODHL (divide and modulus routines), the carry flag is returned set (C) if a carry or borrow occurred. The divide and modulus routines return the carry unchanged.

4.3.7.1 @DEADDHL - BC=DE+HL

The @DEADDHL routine performs 16 bit unsigned integer addition.

It expects the addend in the DE register and the augend in the HL registers.

It returns the sum in the BC registers and the carry clear (NC) unless a carry out of the high order bit occurs.

It preserves the HL and DE registers.

4.3.7.2 @DESUBHL - BC=DE-HL

The @DESUBHL routine performs 16 bit unsigned integer subtraction using twos compliment addition.

It expects the minuend in the DE registers the subtrahend in the HL registers.

It returns the difference in the BC registers as a twos compliment number and the carry clear (NC) unless a borrow into the high order bit occurs.

It preserves the HL and DE registers.

4.3.7.3 @DEMULHL - BC=DE*HL

The @DEMULHL routine performs 16 bit unsigned integer multiplication.

It expects the multiplicand in the DE registers and the multiplier in the HL registers.

It returns the product in the BC registers and the carry clear (NC) unless a carry out of the high order bit occurs.

It preserves the HL and DE registers.

4.3.7.4 @DEDIVHL - BC=DE/HL

The @DEDIVHL routine performs 16 bit unsigned integer division.

It expects the dividend in the DE registers and the divisor in the HL registers.

It returns the integer quotient in the BC registers.

It preserves the HL and DE registers.

4.3.7.5 @DEMODHL - BC=DE%HL

The @DEMODHL routine performs 16 bit unsigned integer division and returns the modulus (remainder) of the operation.

It expects the dividend in the DE registers and the divisor in the HL registers.

It returns the remainder of the division in the BC registers.

It preserves the HL and DE registers.

Example: $5/2=2$ and a remainder of 1. The quotient is the result of @DEDIVHL and the modulus (or remainder) is the result of @DEMODHL.

4.3.8 MESSAGE OUTPUT SUBROUTINES

These routines provide a simple means for outputting standard messages. Some of the routines access the system messages while others allow the user to set up a table of applications messages. The system messages are described in Section 4.8.

4.3.8.1 @DISKERROR - DISK ERROR MESSAGES

The @DISKERROR routine outputs system error messages related to disk operation. The routine closes all open disk files, outputs the appropriate error message to the console stream, and returns control to the MDOS executive which resets the 8080 stack to the MDOS system stack.

It will output the appropriate error messages as detected by FILE MANAGEMENT and PHYSICAL DISK ACCESS routines (Sections 4.3.3 and 4.3.4) when they return a carry set (C) condition and an error message code in the A register.

It expects the error message code in the A register.

It DOES NOT RETURN.

4.3.8.2 @CLOSEFILES - CLOSE ALL FILES

The @CLOSEFILES routine closes all open files using the standard system file close routines. Any errors that are encountered will be reported on the console device.

It always returns the carry flag clear (NC).

It preserves the HL, DE and BC registers.

4.3.8.3 @ERRORMES - ERROR MESSAGES

The @ERRORMES routine performs similarly to @DISKERROR except that it does not close all open files and it does return to the calling routine on exit.

It expects the error message code in the A register.

It preserves the C register.

4.3.8.4 @MESSAGEOUT - MESSAGE OUTPUT

The @MESSAGEOUT routine is a generalized message-table output routine. The user can provide his own applications message table and use this routine to output the messages to the console stream. The table may have variable length messages with imbedded blanks. Each message can be terminated with a control character or a character with the most significant bit set high. The control character will not be output. The character with the eighth bit high will be output after the bit is stripped. For example, an ASCII A is hex 41. C1 hex is an ASCII A with the most significant bit high.

It expects the message table's address in the HL registers.

It expects the message's code in the A register. The code corresponds to the message's location in the table. ie., 0 is the first message, 5 is the sixth etc.

It preserves the C register.

4.3.9 SYSTEM BUFFERS AND ENTRY POINTS

These are miscellaneous entry points and buffers already described in detail in conjunction with other subroutines.

@CONSOLEADDR - Contains the location of @CIOTABLE

@LISTADD - Contains the location of @LIOTABLE

@CIOTABLE - Start address of the console input/output vector table

@LIOTABLE - Start address of the list input/output vector table

@PCON - Start address of physical console driver routines

@PLIST - Start address of physical list driver routines

@WARMSTART - Warm start entry point; initializes console and list devices, and prints the MDOS signon message.

@MDOSEXECUTIVE - Entry point for MDOS executive. Outputs the current MDOS executive prompt and initializes the MDOS stack. This entry does not output the signon message.

@FILEBUFFER0 and @FILEBUFFER1 - @FILEBUFFER0 and @FILEBUFFER1 are 288 byte buffers used by the system for file access. They may be used as applications program file buffers. See the section on FILE ACCESS ROUTINES.

@APROGRAM - Address of the start of the applications area. The APP command transfers program control to this address. All file types except overlay (0C-0F hex) must have load addresses greater than or equal to @APROGRAM or a LOAD ADDRESS ERROR will occur when an attempt is made to load the file.

@MASKADDR - A two byte pointer used by the @SEAR routine. @MASKADDR points to the address of the mask string.

@MDOSRETURN - Applications programs that have not changed the I/O initialization return to this entry point instead of **@WARMSTART**. **@MDOSRETURN** outputs the MDOS signon message and initializes the MDOS stack but does not reinitialize the I/O handlers.

The following buffers are used by the @PARAM routine and are discussed in detail there.

- 1) One byte buffers which holds the number of specified parameters.

@NDRVPAR @NASCPAR @NBINPAR

- 2) Ten byte buffers which holds ASCII parameters.

@ASCBUFF0 @ASCBUFF1
@ASCBUFF2 @ASCBUFF3

- 3) One byte buffers which holds disk unit number parameters.

@DRIVEN0
@DRIVEN1
@DRIVEN2
@DRIVEN3

- 4) Two byte buffers which holds binary parameters.

@BBUFF0 @BBUFF1
@BBUFF2 @BBUFF3

@ASCIIBUFF - @ASCIIBUFF is a ten byte buffer which holds filenames for the @CREATE, @RENAME, @SCRATCH, and @TRANSFILENAME routines.

@INBUFF - @INBUFF is the system input buffer. It is 132 bytes long.

4.4 LINEEDIT - THE MDOS LINE EDITOR

LINEEDIT is an MDOS application program which provides assistance in creating and maintaining assembly language source program files that are compatible with the MDOS 8080/8085 assembler. It may also be used as a limited general text editor.

LINEEDIT is invoked by typing LINEEDIT in response to an MDOS executive prompt or by typing the command LOAD "LINEEDIT" followed by the command APP. It signs on with the message MDOS LINE EDITOR VS. X.X.

The user interacts with LINEEDIT through the system console. Lines entered at the keyboard may be text lines which are stored in the edit buffer or commands for LINEEDIT to execute. The general editing process consists of three parts.

- 1) Placing a text file into the edit buffer by entering it a line at a time from the keyboard or by loading an existing file from disk.
- 2) Modifying the text file in the edit buffer by adding, changing, and deleting lines.
- 3) Storing the file in the edit buffer onto a disk.

How to use LINEEDIT to carry out this process is described in the following sections.

4.4.1 ENTERING LINES TO LINEEDIT

After signing on LINEEDIT waits for a line to be input. A line consists of not more than 132 characters typed in sequence. The entry of a line is terminated by pressing the RETURN key. During the entry of a line each character that is typed is echoed by LINEEDIT on the console display. If more than 132 characters are typed prior to the RETURN, LINEEDIT will stop echoing characters and only honor a valid control function such as the RETURN. Characters which may be entered into a text line are ASCII characters in the code range 20H to 7EH with the exception of the backarrow (5FH). LINEEDIT also uses the MDOS console output system to keep track of the character count as a line is typed and automatically output a carriage return/line feed combination when the count exceeds the width of the display device. This combination is not included in the line count.

Two control features may be used when entering a line.

- 1) Each time the RUBOUT key is pressed the next previously typed character will be deleted from the line. A backarrow is echoed to the terminal display for each character deleted. Neither the deleted characters nor the backarrow are included in the line count.

- 2) Holding down the control key and typing X (CNTL/X) will cause all of the current line to be cancelled. A carriage return/line feed combination is echoed to the terminal display. LINEEDIT is positioned to accept entry of a new line.

4.4.2 KEYING IN A NEW TEXT FILE

LINEEDIT recognizes a line as a text file line by the presence of a leading line number. Each line number must be in the range 0 to 9999. A text file is entered one line at a time using the normal line entry procedure. As each line is entered LINEEDIT stores it in the edit buffer which it maintains in the computer system's main memory. Text lines are stored in the edit buffer in numeric order by line number. The lines in the buffer at any given time constitute the current text file.

To insert a new line in the current text file, type in the new line including the line number. LINEEDIT will automatically place the new line in the program buffer in proper sequence according to its line number.

To replace an existing line in the current text file enter the line number and the new text. The new line will automatically replace the old line that has the same line number in the current text file.

To delete one existing program line in the current text file type the line number and press the return key. The corresponding line will be eliminated from the current text file. Note that multiple lines may also be eliminated by using the DELT command as described in Section 4.4.18.

Consecutive text lines may be entered conveniently by using LINEEDIT's automatic line numbering feature. Prior to typing the first character of a new line, you can cause the 'next' line number to be generated for you by pressing the space bar one time. The 'next' line number will echo to the terminal display and LINEEDIT will then be waiting for the first text character of that line. See Section 4.4.7 on the AUTO command to specify the increment that determines the 'next' line number.

4.4.3 ENTERING LINEEDIT COMMANDS

Whenever a line is typed which does not begin with a line number, LINEEDIT attempts to interpret this line as a command. If the line is not recognizable as a LINEEDIT command, the message COMMAND NOT FOUND will be displayed. LINEEDIT commands are single words or abbreviations followed by parameters if required. All LINEEDIT commands are uppercase only. If the command requires one or more parameters, there must be at least one space between the command word and the first parameter and between each parameter. Parameters may be ASCII or numeric. ASCII parameters must be enclosed in double quotation marks except for within the SEARCH and CHANGE command dialogues. Numeric parameters are entered in decimal. LINEEDIT offers commands to facilitate the management of the editing process.

4.4.4 THE CLEAR COMMAND

The edit buffer may be initialized to an empty state by using the CLEAR command. This command has no parameters. It is entered by typing CLEAR and pressing the return key.

Entering a CLEAR command may result in the message FILE ON DISK NOT UPDATED, PROCEED?. This is a warning that the contents of the current text file has not been stored on disk since it was last altered. When the message appears the current text file is not yet lost. To override this warning type Y and press the return key. The CLEAR command will be processed. Otherwise type N and press the return key. The message CANCELLED will be displayed and LINEEDIT will be waiting for an alternate command.

When the CLEAR command is processed, LINEEDIT will display the message FILE NOT NAMED followed by two hex numbers which indicate that the edit buffer is empty and unnamed.

4.4.5 THE NAME COMMAND

The current text file in the edit buffer may be named or renamed by using the NAME command. NAME "filename" is the general form of this command. The filename may be any valid MDOS filename. No disk drive unit number should be specified since this name is to be associated with the current text file in the edit buffer which is in the main system memory. When the NAME command is executed, LINEEDIT will display the new filename followed by two hex numbers which represent the beginning and ending addresses of the current text file in memory. A text file may be keyed into the edit buffer before it is named. However, it cannot be stored on disk without being named.

4.4.6 THE FILE COMMAND

The name of the current text file and its address limits in memory can be determined by using the FILE command. This command has no parameters. It is entered by typing FILE and pressing the return key. The name of the current text file will be displayed, followed by two hex numbers which are the starting and ending memory addresses of the current text file. If the current text file has not been named, the message FILE NOT NAMED will be displayed in place of the filename.

4.4.7 THE AUTO COMMAND

LINEEDIT's automatic line numbering facility adds a fixed increment to the last entered line number in order to compute the 'next' automatic line number. When LINEEDIT is started this increment value is set at a default of 1. This value may be changed by using the AUTO command. The general form of the command is AUTO number. The increment will be set to the decimal value of number.

4.4.8 THE PROMPT COMMAND

When LINEEDIT is started its prompt message is null. After processing an input line, it simply echoes a carriage return/line feed combination, and waits for a new input with the cursor at the left margin of the terminal display. A prompt character or message can be specified for LINEEDIT by using the PROMPT command. PROMPT "message" is the general form of this command. The message may be from 1 to 10 characters in length and include any characters valid in a text line. It must be enclosed in double quotes as shown. When the PROMPT command is executed, LINEEDIT will immediately display the new prompt at the left of the terminal display and be positioned waiting for a new input line. The LINEEDIT prompt may be restored to its initialized state by typing PROMPT and pressing the return key.

4.4.9 THE LOAD COMMAND

A text file may be loaded into the edit buffer from disk by using the LOAD command. LOAD "unit number:filename" is the general form of the command. The double quotes must be used as shown. The filename must be a valid MDOS filename. The unit number is optional. If it is supplied, it must consist of a single digit from 0 to 3 followed by a colon (:). It designates the disk unit on which the specified file is to be found. If no unit number is specified, unit 0 is assumed.

When a text file is successfully loaded, it replaces the contents of the edit buffer and all text from the previous text file in the buffer is lost. The name of the current text file becomes the name of the disk file that was loaded, not including the unit number.

Entering a LOAD command may result in the message FILE ON DISK NOT UPDATED, PROCEED?. This is a warning that the current text file has not been stored on disk since it was last altered. When the message appears, the current text file is not yet lost. To override this warning type Y and press the return key. The LOAD command will be processed. Otherwise, type N and press the return key. The message CANCELLED will be displayed and LINEEDIT will be waiting for an alternate command.

Entering a LOAD command may result in the message FILE BUFFER OVERFLOW. See Appendix D for an explanation of this condition.

4.4.10 THE APPEND COMMAND

A text file may be loaded from disk and appended to the end of the current text file in the edit buffer by using the APPEND command. APPEND "unit number:filename" is the general form of this command. The double quotes must be used as shown. The filename must be a valid MDOS filename. The unit number is optional. If it is supplied, it must consist of a single digit from 0 to 3 followed by a colon (:). It designates the disk unit on which the specified file is to be found. If no unit number is specified, unit 0 is assumed.

When an APPEND is executed, the text file from disk is concatenated onto the end of the text file which was already in the edit buffer. The text lines of the appended file are not merged into the existing file in order by line number. The appended file may contain line numbers which conflict with the existing file. For these reasons it is important to use the RENUM command immediately after a successful APPEND.

The name of the current text file in the edit buffer is not affected by an APPEND.

Entering an APPEND command may result in the message WRONG FILE TYPE. This is an indication that the requested file has an attribute type different than 4 through 7. These are the only valid source file types acceptable to LINEEDIT and the assembler.

Entering an APPEND command may result in the message FILE BUFFER OVERFLOW. This is an indication that the amount of system memory available for the edit buffer is not enough to hold the additional file which was requested. When this condition occurs, the requested file is not appended but the existing is retained without change.

4.4.11 THE SAVE COMMAND

The current text file in the edit buffer may be stored on disk as a new disk file by using the SAVE command. The general form of this command is SAVE unit number. The unit number is optional. If it is supplied, it must consist of a single digit from 0 to 3. It designates the disk unit on which the current text file is to be stored. If no unit number is specified, unit 0 is assumed.

The name of the current text file in the edit buffer is used to create an entry in the directory of the specified disk and the text file is stored on the disk under that name. If the name already exists on the specified disk a DUPLICATE NAME message will result, and nothing will be written to disk. The edit buffer is unchanged. The file may be SAVED by first changing its NAME to one that doesn't conflict or by using the RESAVE command if appropriate.

A file created by the SAVE command is given the attribute type 4 which marks it as an editor/assembler source file.

4.4.12 THE RESAVE COMMAND

The current text file in the edit buffer may replace an existing file or disk by using the RESAVE command. The general form of this command is RESAVE unit number. The unit number is optional. If it is supplied, it must consist of a single digit from 0 to 3. It designates the disk unit on which the existing file to be replaced is found. If no unit number is specified, unit 0 is assumed.

The directory of the specified disk unit is searched for a filename which matches the name of the current text file in the edit buffer. The current text file is written over that file on the disk. If no match is

found, the message FILE NOT FOUND will be displayed. The current text file can be saved as a new file by using the SAVE command. If the file matched on disk has a type other than 4 through 7, the message WRONG FILE TYPE will be displayed. Text source files must have a source file type.

4.4.13 THE LIST COMMAND

A formatted display of lines in the current text file can be output to the system console by using the LIST command. The forms of this command are LIST, LIST linenumber1, and LIST linenumber1 linenumber2. The display will begin with linenumber1 or the next highest and continue through linenumber2 or the next lowest. If linenumber1 and linenumber2 are the same, only one line will be displayed. If linenumber2 is less than linenumber1, nothing will be displayed. If linenumber2 is not supplied, the display will begin with linenumber1 or the next highest, and continue through the last line currently in the current text file. If no line numbers are supplied, the entire edit buffer will be displayed.

The LIST command produces a formatted display of the text lines that is oriented to 8080 assembly language source text. The format is defined as four fields each beginning at a specific tab location. The first field begins at the left margin and displays the line number as a 4 digit number. The second field is the label field. It consists of all characters in the text line through the first space or colon (:) that occurs. The third field is the opcode and operands field. The opcode consists of all characters following the label field through the next occurrence of a space. The operand consists of all characters following the opcode through the next occurrence of a space. The fourth field is the comment field. It begins with a semicolon (;) following the space that terminates the operands and continues to the end of the text line.

Refer to the TAB command to change the tab settings which determine the placement of the fields for the LIST format. When using the LIST command with general text editing, it is advisable to set the tabs to 1 1 1. This effectively removes the tabulation effects which are designed for assembly language source text.

4.4.14 THE LISTP COMMAND

A formatted display of lines in the current text file can be output to the system printer by using the LISTP command. The forms of this command are LISTP, LISTP linenumber1, and LISTP linenumber1 linenumber2.

The LISTP command functions the same as the LIST command except that output is directed to the system printer instead of the system console.

4.4.15 THE PRINT COMMAND

A literal (unformatted) display of lines in the current text file can be output to the system console by using the PRINT command. The forms of this command are PRINT, PRINT linenumber1, and PRINT linenumber1 linenumber2. The linenumber specifications in the PRINT command function the same as in the LIST command.

The PRINT command displays text lines as they are stored in the edit buffer but without the line numbers so that general text may be displayed just as it was entered. If an unformatted display of assembly language source text is desired, it can be obtained by setting the tabs to 1 1 1 and using the LIST command.

4.4.16 THE PRINTP COMMAND

A literal (unformatted) display of lines in the current text file can be output to the system printer by using the PRINTP command. The forms of this command are PRINTP, PRINTP linenumber1, and PRINTP linenumber1 linenumber2.

The PRINTP command functions the same as the PRINT command except that output is directed to the system printer instead of the system console.

4.4.17 THE TAB COMMAND

The tab settings that determine the placement of the fields for the LIST and LISTP format may be changed by using the TAB command. TAB number1 number2 number3 is the form of this command. The first number is the column at which the opcode field begins. The second number is the column at which the operand field begins. The third number is the column at which the comment field begins.

The initial and default values of the TAB parameters are 15, 22, 36 decimal. The settings may be reset to these values by typing TAB without any parameters. Missing parameters are set to the default if possible or the value of the preceding parameter if that parameter is greater than the default value for that tab column. If TAB 17 were typed the tab setting would be 17, 22, 36. TAB 25 would set the tabs to 25, 25, 36.

4.4.18 THE DELT COMMAND

A group of consecutive lines may be deleted from the current text file by using the DELT command. The forms of this command are DELT linenumber1, and DELT linenumber1 linenumber2. Lines will be deleted from linenumber1 or the next highest that exists, through linenumber2 or the next lowest that exists. If linenumber2 is less than linenumber1 nothing will be deleted. If they are equal only that line will be deleted. If only linenumber1 is specified then only that line will be deleted. The edit buffer is automatically compressed whenever lines are deleted.

4.4.19 THE RENUM COMMAND

All or part of the lines in the current text file can be renumbered by using the RENUM command. The forms of this command are RENUM, RENUM startingnumber, RENUM startingnumber increment, and RENUM startingnumber increment first-line-to-change. RENUM takes the line number of the first line to change and sets it equal to the starting number. The line number of each line after the first line to change is then set to the value of the preceding new line number plus the increment value. If no first line to change is specified, the first line in the edit buffer is assumed. If no increment value is specified, the value 10 is used. If no starting number is specified, the value 0 is used. Typing RENUM alone will produce a text file numbered from 0 by 10's.

Entering a RENUM command may result in the message LINE NUMBER OVERFLOW. This is an indication that the renumbering attempt lead to a line number greater than 9999. When this occurs the edit buffer is left in a partially renumbered state. Lines up to the overflow point have been renumbered but the ones after that point retain their old value. A RENUM with a smaller increment value should be executed immediately to correct this condition.

4.4.20 THE SEARCH COMMAND

Lines in the current text file that contain a specified string of text can be located and displayed by using the SEARCH command. The forms of this command are SEARCH, SEARCH linenumber1, or SEARCH linenumber1 linenumber2. SEARCH without a linenumber specified will search the whole buffer. SEARCH linenumber1 will search from the line number specified to the end of the buffer. SEARCH linenumber1 linenumber2 will search the buffer starting at the first line specified through the second line specified.

When the SEARCH command is entered, LINEEDIT will respond with the prompt SEARCH MASK ?. A string of up to 132 legal text line characters can be entered. The entry is terminated by pressing the return key. LINEEDIT searches through the lines in the current text file looking for the first occurrence within each line of a substring that matches the specified search mask. It examines every line except those lines that begin with an asterisk (*). Every examined line that contains a match is displayed on the system console. This display is a literal (unformatted) display including the line number. Lines with a leading asterisk (*) are considered comment lines in assembly language source text. Refer to the SEARCHALL command to operate on comment lines.

The SEARCH command also provides a universal match character capability. Each question mark (?) that is entered in the search mask string is treated as a match for any character in that position. For example, the search mask A?I will match all three character substrings that begin with A and end with I. Note that this capability means that question marks (?) included in the text cannot be explicitly searched for.

If no lines in the current text file contain a match to the specified search mask, the message STRING NOT FOUND will be displayed.

4.4.21 THE SEARCHALL COMMAND

All lines in the current text file that contain a specified string of text, including those lines that begin with an asterisk (*) can be located and displayed by using the SEARCHALL command.

The forms of this command are SEARCHALL, SEARCHALL linenumber1, or SEARCHALL linenumber1 linenumber2. SEARCHALL without a linenumber specified will search the whole buffer. SEARCHALL linenumber1 will search from the line number specified to the end of the buffer. SEARCHALL linenumber1 linenumber2 will search the buffer starting at the first line specified through the second line specified. The SEARCHALL command functions the same as the SEARCH command except that all text lines including those that begin with an asterisk (*) are included in the search.

4.4.22 THE CHANGE COMMAND

The first occurrences of a specified string in lines of the current text file can be replaced with a different string of same or different length by using the CHANGE command. The forms of this command are CHANGE, CHANGE linenumber1, or CHANGE linenumber1 linenumber2. CHANGE without a linenumber specified will change all lines in the buffer. CHANGE linenumber1 will change lines from the line number specified to the end of the buffer. CHANGE linenumber1 linenumber2 will change lines in the buffer starting at the first line specified through the second line specified.

CHANGE operates on all lines within the specified range except lines starting with an asterisk (*) or semicolon (;). These lines are considered comment lines in assembly language source text. Refer to the CHANGEALL command to operate on comment lines.

When the CHANGE command is entered, LINEEDIT will respond with the prompt SEARCH MASK ?. A string of up to 132 legal text line characters may be entered. The entry is terminated by pressing the return key. If no lines in the current text file contain a match to the specified search mask, the message STRING NOT FOUND will be displayed. Otherwise, LINEEDIT will then respond with the prompt CHANGE TO ?. Another string of up to 132 legal text string characters can be entered. The entry is terminated by pressing the return key. LINEEDIT searches through lines in the current text file looking for the first occurrence within each line of a substring that matches the specified search mask. It replaces such occurrences with the specified change-to string, adjusting line and buffer length accordingly. Each line as changed is displayed on the console without tabs expanded.

The CHANGE command also respects the universal match character capability as described under the SEARCH command. If the search mask contains one or more question marks (?) these characters positions will match any character in the search process, and the matched substring will then be replaced by the change-to string. Example:

```
LIST
10 S1@LABEL1A
20 S2@LABEL2A
30 @LABEL3
CHANGE
SEARCH MASK ? S?@
CHANGE TO ? @
10 @LABEL1A
20 @LABEL2A
```

The change-to string may also contain question marks (?). This provides the ability to retain specified character positions in the search string while making changes on either or both sides of the retained character. Example:

```
LIST
10 TAGØ1A
20 TAGOFF
30 TAG22A
CHANGE
SEARCH MASK ? TAG??A
CHANGE TO ? LABEL??B
10 LABELØ1B
30 LABEL22B
```

Lines 10 and 30 have been changed while line 20 is unchanged because it did not match the search string. The TAG at the beginning and the A at the end of lines 10 and 30 have been changed. The Ø1 in line 10 and the 22 in line 30 have been retained.

4.4.23 THE CHANGEALL COMMAND

The first occurrences of a specified string in all lines of the current text file, including those lines that begin with an asterisk (*), or semicolon (;) can be replaced with a different string of same or different length by using the CHANGEALL command. The forms of this command are CHANGEALL, CHANGEALL linenumber1, or CHANGEALL linenumber1 linenumber2. When the CHANGEALL command is entered it functions the same as the CHANGE command, except that all text lines including those that begin with an asterisk (*) are included in the search.

4.4.24 THE EDIT COMMAND

The text within a specified line in the current text file can be changed without retyping the entire line by using the EDIT command. EDIT linenumber is the form of this command. If the specified linenumber is not found in the current text file, the message LINE NOT FOUND is displayed. LINEEDIT processes an EDIT command by copying the specified line into a special editing buffer and displaying the line number at the left margin of the console. An invisible edit pointer is set to point to the first character in the text line after the space that terminates the line number. LINEEDIT is now in the EDIT command mode. A separate set of single key commands is available for editing a line in the special edit buffer.

4.4.24.1 ADVANCING THE EDIT POINTER - THE SPACE BAR

The invisible edit pointer in the special editing buffer may be advanced one position by pressing the space bar one time. The character to which the edit pointer is pointing will be displayed on the console. This indicates that the edit pointer has passed over the character. The edit pointer is then advanced so that it is now pointing at the next character in the text line immediately after the one that is displayed. The entire line can be displayed in this manner.

4.4.24.2 CHANGING THE NEXT CHARACTER - C

The character to which the edit pointer is pointing in the edit buffer can be changed by typing a c or C, followed by the new character. The new character is printed on the console and replaces the character in the edit buffer at that position. The edit pointer is advanced to point to the character immediately after the new displayed character.

4.4.24.3 DELETING THE NEXT CHARACTER - D

The character to which the edit pointer is pointing in the edit buffer can be deleted by typing a d or D. The deleted character is printed on the console enclosed in backslashes (\). The edit pointer is left pointing at the character immediately after the deleted character.

4.4.24.4 INSERTING CHARACTERS - I

Characters may be inserted into the line or at the end of the line by typing an i or I followed by the characters to be inserted. The insertion begins immediately before the character pointed to by the edit pointer. Characters are inserted in sequence as typed until the insert mode is terminated by typing an escape (1B hex). The edit pointer remains pointing to the same character that it pointed to when the insertion began. The insert mode may also be terminated by pressing the return key. This also terminates the EDIT command and replaces the line in the current text file with the newly edited version from the special editing buffer.

4.4.24.5 LISTING THE LINE IN THE SPECIAL EDITING BUFFER - L

The remainder of the line in the special edit buffer from the position of the edit pointer to the end of the line may be displayed by typing an l or L. The characters are displayed on the console followed by a carriage return-line feed. The line number is reprinted at the left margin of the console display and the edit pointer is reset to the beginning position. This command is useful to see what the line looks like before editing is completed. It may also be useful to use this command immediately after entering the original EDIT command. This would display the line about to be edited without exiting the editing mode.

4.4.24.6 SEARCHING TO A SPECIFIED CHARACTER - S

The edit pointer may be advanced in the special editing buffer to the first occurrence of a specified character by typing an s or S followed by the character to search for. The characters from the position of the edit pointer up to but not including the searched for character are printed on the console. The edit pointer is left pointing at the first occurrence of the searched for character. If the search argument does not exist in the line then the entire line is printed and the edit pointer is positioned at the end of the line.

4.4.24.7 DELETING TO A SPECIFIED CHARACTER - K

Characters in the special editing buffer from the edit pointer position up to but not including a specified search character can be deleted by typing a k or K followed by the search character. The deleted characters are displayed on the console, enclosed in backslashes (\). If the search argument does not exist in the edit line, then all the characters from the edit pointer to the end of the line are deleted. The edit pointer is left pointing at the search character or at the end of the line.

4.4.24.8 QUITTING THE EDIT COMMAND MODE - Q

The EDIT command may be aborted without changing the line in the current text file by typing a q or Q. The partially edited line in the special editing buffer is abandoned. No changes are made to the line in the current text file. LINEEDIT is ready to accept a new command.

4.4.24.9 COMPLETING THE EDIT COMMAND - THE RETURN KEY

The line in the special editing buffer can replace the line in the current text file at any point by pressing the return key. This terminates the EDIT command in a normal manner.

4.4.25 THE DOS COMMAND - EXITING FROM LINEEDIT

Control of the computer system can be returned from LINEEDIT to the MDOS executive by using the DOS command. This command has no parameters. It is entered by typing DOS and pressing the return key. Control is returned to the MDOS executive which signs on with the message MICROPOLIS MDOS VS. X.X. LINEEDIT remains in the system application program area and the contents of the current text file are not disturbed unless some action taken from the executive destroys these areas. Entering an APP command to the executive would return control to LINEEDIT.

Entering the DOS command may result in the message FILE ON DISK NOT UPDATED, PROCEED?. This is a warning that the current text file has not been stored on disk since it was last altered. When the message appears the current text file is not yet lost. To override this warning type Y and press the return key. The DOS command will be processed. Otherwise type N and press the return key. The message CANCELLED will be displayed and LINEEDIT will be waiting for an alternate command.

4.4.26 LINEEDIT FILE STRUCTURE

The current text file in the LINEEDIT edit buffer has the following format. Each line begins with a byte that contains a count of the number of bytes in the line. The count includes the count byte and the carriage return at the end of the line. The count byte is followed by four bytes that hold the digits of the line number in ASCII. The line number can range from 0000 to 9999 . At least one space (20 hex) follows the line number. The remainder of the line can contain from 0 to 125 characters followed by a carriage return. The shortest line contains 6 bytes. The longest line contains 132 bytes. The characters of the source program appear in the line exactly as they were typed during input. ASSM and LINEEDIT require only one space between elements of an assembly statement. Additional spaces are ignored. Therefore, there is no reason to type in more than the minimum number of spaces when entering a source program. After the carriage return that terminates the last line of the current text file there is a byte that contains a 01 to mark the end of the file.

The current text file is written to a disk file just as it appears in the edit buffer. All records in the disk file with the possible exception of the last one are full records. A text line may span two records. The following logic could be used in an MDOS application program designed to process an editor source file.

1000	START	CALL	@RFINXPOS
2000		DCR	C
3000		JZ	ENDOFFILE
4000		MVI	D, \emptyset
5000		MOV	E,C
6000		LXI	H,BUFFER
7000		CALL	@LOADDATA
8000	*PROCESS THE LINE IN THE BUFFER		
9000		JMP	START

The @RFINXPOS routine gets the line count byte into the C register. If the count is 01 the end of the file has been reached. Otherwise, all program lines have a line length of no less than 6. The line length is moved into the DE registers ($D=\emptyset$) and the buffer address is placed into the HL registers. The @LOADDATA routine starts at the index position and loads the next DE bytes into the buffer which leaves the index position pointing to the line count byte of the next text line. The program can then process the text line and loop back to get the next line.

4.5 ASSM - THE MICROPOLIS 8080/8085 DISK ASSEMBLER

An assembler converts a source program written in an assembly language into an object program which consists of a sequence of binary codes that can be loaded into a computer's main memory and executed. ASSM is an assembler for the 8080/8085 micro-processors. It uses a MICROPOLIS diskette subsystem as peripheral storage for the source and object files during the assembly process. Use of a peripheral storage medium allows the assembly of programs that could not otherwise be assembled because the source and object files could not fit into the micro-processors main memory during the assembly.

ASSM produces an absolute object file that can be scatter loaded into main memory. The object file contains all address references generated by ORG and DS statements. The operating system puts the object code in the proper place. Object files on disk do not have to be contiguous memory images to load correctly.

4.5.1 HOW TO INVOKE ASSM

From the MDOS executive ASSM is invoked by entering the file name ASSM, like an MDOS command, followed by a list of parameters. The format is as follows:

ASSM "<source filename>" "<object filename>" "<options>" [<offset>]

The source file must be a TYPE 04 through 07 file which has been created by the line editor program described in Section 4.4. The object file will be created by ASSM and given a TYPE of 8.

The option field directs the output from the assembler to different places. Options are specified by grouping the following letters together as required:

- E Only assembly errors will be listed. *ERRORS*
- P The assembly listing will be paginated.
- S Only an assembly listing will be produced. No object code will be written to disk or memory. *SOURCE CODE*
- M The object code will be written directly into memory at locations specified in the source unless an offset is specified.
- L The line numbers used during editing will not be written on the assembly listing. *NO LINE NO.*
- T The symbol table created by the assembly will be output at the end of the listing. *SYMB. TABLE*
- C All output from the assembler will go to the console output device. *OUTPUT
TO
CONSOLE*

Option codes are grouped together within the option string. For example, ASSM "1:SGAME" "GAME" "PLT" will assemble the source file called SGAME on disk drive one and create an object file on drive zero by the name GAME. The assembly listing will be output to the list stream and each page will be numbered and titled with a field header at the top of each page. The line numbers used by the editor will not appear on the assembly listing. The symbol table will be added to the end of the assembly listing.

The P option causes the assembler to paginate and title the output listing. If the FORMFLAG location (see 2.2.5.1) contains a zero value, the pagination will be done by outputting linefeeds to advance the paper to the top of the next page. If the FORMFLAG location contains a non-zero value, a single FORMFEED (ASCII 12) will be output. When using the linefeed mode, the assembler will assume the paper is at the top of form when the assembly command is given from MDOS. In the formfeed mode, a formfeed will be output before any printing to resynchronize to the top of form. The FORMFLAG location should be configured when the system printer I/O is configured. See section 2.2.5.1.

The S and M options are mutually exclusive. S indicates that no object code is to be produced while M indicates that the object code is to be placed into memory. The S option is always dominant.

When the options S or M are specified the second ASCII parameter which holds the filename for the object file can be left out by typing "", because these options do not produce a disk file. If the second parameter is present it is ignored. The parameters are positional so the "" must be used if there is no object file and/or there are no options.

Examples: ASSM "STEST" "TEST" "

ASSM "STEST" "" "PS"

The blank parameter is mandatory in both cases.

The optional offset parameter is only used when the object code is to be placed directly into memory using the M option. The offset is added to the actual address where the code would be placed as specified by the programs ORG statement. The code is assembled to run at the ORG address but is placed at a different address temporarily. The object code will not run at the resulting offset address and must be moved to the proper location before being executed. The offset is useful when the program is intended to run at @APROGRAM. The program must first be placed into a memory area that does not conflict with the assembler program or the generated symbol table.

Example:

0500	LINK	'SYSQ1'
1000	LINK	'SYSQ2'
2000	START	ORG @APROGRAM ;2B00 HEX
3000	FILL	10H,0 ;2B00-2B0F FILL 0
4000	BEGIN	JMP \$;SOFT HALT
5000	END	BEGIN ;EXADD TO SOFTHALT

ASSM "STEST" "" "M" 3000

After the assembly memory would look as follows:

5B00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
5B10	C3 10 2B

The code will not run at this location because the soft halt jump is assembled for an ORG of 2B00 hex. A MOVE 5B00 5B12 2B00 command would move the code down to the proper location and an EXEC 2B10 would run the program properly.

If this program were assembled into memory without the offset parameter, it would attempt to overwrite the assembler resulting in a LOAD ADDRESS ERROR. For this reason assembling directly into memory requires great care and should only be done if you are absolutely sure all the code produced during the assembly is outside of the operating system; outside of the assembler program; and does not conflict with the generated symbol table. The symbol tables starts immediately after the assembler program and grows toward high memory. Each label requires one byte for each character of the label plus two bytes for the address or value definition of the label. The approximate size of the symbol table can be evaluated by averaging the label size adding two and multiplying by the number of labels. If the size of the symbol table plus a safety margin is added to 3D00 hex (the start of the symbol table rounded up to the next even page), the resulting address should be a safe area to put the object code when using the M option.

4.5.2 LANGUAGE ELEMENTS

The assembler translates assembly language statements into 8080 machine code. A statement consists of a line number, a label, an opcode, operands, and comments.

The first element is the line number. The assembler ignores the line numbers in the source line. Line numbers are only used by the line editor program and have no meaning to the assembler.

The second element in a statement must be a label or a delimiter to indicate that there is no label in the statement. ASSM accepts two label delimiters, the space, and the colon (:). If a label does not appear, then the first element of the statement must be a space or a colon (:). Additional spaces are ignored and the next non space character is the start of the third element.

The third element in a statement is the operation code mnemonic. The assembler uses the standard 8080/8085 opcode mnemonics developed by the manufacturer.

The fourth element in a statement is the operand field. Some opcodes require no operands while others need one or two. If an operand is not required, the fourth element is automatically considered to be a comment field.

The last element of a statement is the comment field. The comment field is printed on assembly listings but is ignored by the assembler. It's only purpose is for program documentation and clarity. If the comment field is preceded by a semi-colon (;) the comment will be formated on the assembly listing. If the semi-colon is left out, the comment will appear after the operand field just as it was entered. This feature does not effect the assembly in any way. It is only a formating feature.

The statement line looks as follows:

LINE# LABEL OPCODE OPERANDS COMMENTS

All spaces shown are mandatory as delimiters, except after a label where the colon (:) can replace the space as a delimiter. Additional spaces are ignored with the exception that a label must start immediately after the space following the line number.

A line can be designated as a comment only line. This is done by putting an asterisk (*) or a semicolon (;) as the character immediately after the space that follows the line number. If the comment line is formed with an asterisk (*), the line will be listed exactly as entered. If it is formed with a semicolon (;), it will be tabulated to start on the same column as in-line comments.

4.5.2.1 LITERALS

The assembler provides for numeric and ASCII literals. Numeric literals can be decimal, hexadecimal, binary, or octal. The following suffixes designate the appropriate base:

A capital H is used to designate base 16, hexadecimal.

A capital B is used to designate base 2, binary.

A capital Q is used to designate base 8, octal.

Base 10, decimal, can be designated by either a capital D, or no suffix.

All numeric literals must begin with a digit in the range zero through nine regardless of the base. This is done to avoid ambiguity between hexadecimal literals and symbolic names. For example, the hex address F90C must be written as ØF9ØCH.

ASCII literals appear between single quotes ('') and can include any ASCII character from 20 hex to 7E hex except the backarrow (5F hex), and the single quote ('').

4.5.2.2 SYMBOLIC NAMES

Labels are symbolic names. Operands may also be symbolic names instead of literals.

Symbolic names consist of a string of ASCII characters. A symbolic name can be from 1 to 47 characters long. It is made up of ASCII characters from 30 hex to 39 hex and 40 hex to 7E hex, except the backarrow (5F hex).

Symbolic names may not start with the digits 0 through 9. This avoids ambiguity between numeric literals and symbolic names. The following characters are valid within a symbolic name:

ABCDEFGHIJKLMNPQRSTUVWXYZ
abcdefghijklmnoprstuvwxyz
Ø123456789@[]^~[;]\

Symbolic names are defined when they appear as labels of an opcode or a pseudo-opcode. This associates a sixteen bit address or value with the label. Symbolic names that are defined may appear as arguments in operands.

Some symbolic names are already defined by ASSM itself and should not normally be redefined in the source program. ASSM gives the registers of the 8080 the following symbolic names:

Register 7 is A.
Register 0 is B.
Register 1 is C.
Register 2 is D.
Register 3 is E.
Register 4 is H.
Register 5 is L.
Register 6 is M, PSW, and SP.

ASSM gives the value of the program counter the symbolic name \$. It changes as the assembly proceeds by assuming the value of the program counter at the start of each statement line being translated.

For example, the following source program line would produce a jump to itself (also called a soft halt).

```
0100      JMP    $      ;JUMP TO SELF
```

4.5.2.3 OPERATORS

ASSM recognizes 10 operators designated with the characters +, -, *, /, %, &, , #, >, and <. These operators may combine with symbolic names and literals to form complex expressions, as described in Section 4.5.3.

All operators treat their arguments as 16 bit unsigned quantities and generate 16 bit unsigned quantities as their result. The operations are from left to right with no hierachal precedence and no precedence specifier (no parenthesis).

The operator + produces the arithmetic sum of its operands to 16 bits.

The operator - produces the arithmetic difference of its operands when used as a subtraction, or the arithmetic negative of its operand when used as unary minus.

The operator * produces the arithmetic product of its operands.

The operator / produces the arithmetic integer quotient of its operands, discarding any remainder.

The operator % produces the integer remainder obtained by dividing the first operand by the second.

The operator & produces the bit-by-bit logical AND of its operands.

The operator ! produces the bit-by-bit logical OR of its operands.

The operator # produces the bit-by-bit logical EXCLUSIVE-OR of its operands.

The operator **>** produces a 16 bit rotate to the right by the number of bits specified in the second operand. The least significant bit becomes the most significant bit and all other bits are shifted to the right.

Example:

1111000011110101B>3 evaluates to 1011111000011110B

The operator **<** produces a 16 bit rotate to the left by the number of bits specified in the second operand. The most significant bit becomes the least significant bit and all other bits are shifted to the left.

Example:

1111000011110101B<3 evaluates to 1000011110101111B

4.5.2.4 OPCODE MNEMONICS

The standard Intel mnemonics for the 8080 and the 8085 are used without exception. For a detailed discussion of the 8080/8085 opcodes refer to the "INTEL 8080 ASSEMBLY LANGUAGE PROGRAMMING MANUAL". Opcodes must be UPPERCASE only.

4.5.3 OPERANDS

Not all opcodes have operands. If an opcode does not have an operand, the element after the opcode is the comment. Some opcodes have one operand while others have two. Where two operands are required they must be separated by a comma (,). There may be no spaces imbedded within the operands.

An operand may consist of a simple or complex expression. A simple expression is a numeric or ASCII literal, or a symbolic name. A complex expression is a combination of numeric or ASCII literals, symbolic names, and operators. The operand is the evaluation of the expression to 16 bits.

Examples:

1000 REG7	EQU	1
2000 TEST	EQU	1000H
3000 INC	EQU	6
4000 TEST	LXI	REG7+4,TEST*6+INC&7<8 ;COMPLEX EXP

The LXI opcode takes two operands. Both of the operands in the example are complex expressions. The expressions are evaluated to 16 bits and truncated to the size of the operands. In the case of the example the first operand would be truncated to 8 bits because it represents a register (05). The second operand evaluates to 16 bits and is not truncated because it is a 16 bit operand (0600 hex).

4.5.4 ASSEMBLER DIRECTIVES

Assembler directives appear in the source program and provide information needed by the assembler to allocate memory space, initialize values, and format listings. Assembler directives are often called pseudo-operations or pseudo-ops. The assembler directives are mnemonics. They are issued in statements like opcodes. The general pseudo-op statement form is as follows:

LINE# LABEL PSEUDO-OP OPERAND COMMENT

The label is optional in all but two of the assembler directives. These are the EQU and the INP pseudo-ops. For the others the label is used when necessary and has the same form and restrictions as labels with opcodes. The END, INP, and PRT pseudo-ops can optionally have operands. The FORM, LIST, NLIST, and ENDIF never have operands.

Pseudo-op operands have the same form as opcode operands. They can be simple or complex expressions.

Labels and operands are optional with some pseudo-ops. Comments are always optional.

Many of the assembler directives are the same as the INTEL pseudo-ops described in the "INTEL 8080 ASSEMBLY LANGUAGE PROGRAMMING MANUAL". However, some are unique to ASSM. Therefore, all of the pseudo-operations are described in detail.

4.5.4.1 ORG - ORIGIN

The ORG pseudo-op specifies where a program or routine within a program is to be placed in memory by setting the assembler's program counter to the value of the operand. If a program does not have an ORG, then the program is assembled at zero. Symbolic names used as operands in the ORG statement must be defined before the ORG statement is encountered. If a label is present, it is associated with the evaluated operands' address.

4.5.4.2 LINK - LINK TO A FILE

The LINK pseudo-op allows separate program files on disk to be assembled to produce one object file. The LINK operand is a source program file name enclosed between single quotes. When a LINK statement is encountered in a source program, assembly continues from the start of the source file named in the operand field and information is saved to allow ASSM to pick up from where it was in the linking source file when the linked to source file is completed.

```
1000      LXI     H,4000H
2000      LINK    'TEST'          ;ASSM TEST AND
3000      MOV     A,M           ;COME BACK HERE
```

In the above example the assembly would assemble all of the file TEST between the LXI H,4000H and the MOV A,M.

The LINK statement allows the assembly of source programs that are much larger than could possibly fit into memory at one time.

No unit is specified in the LINK operand. The linked to file is located as follows. The disk that has the source file which was given in the MDOS command that invoked the assembly is searched. If the linked to file is on that disk the assembly continues as described above. If the linked to file is not located the search proceeds from unit zero through three. If a unit is not loaded or does not exist in the system it is bypassed and the search continues until the filename is found or all units have been searched.

4.5.4.3 END - END OF ASSEMBLY

The END statement signifies to ASSM that the physical end of a program has been reached. Because ASSM allows multiple disk files to be assembled as a single large program, multiple END statements can occur when files have been LINKed together. Under these conditions the END signals the end of a source file and not the absolute end of the assembly. The END will cause the assembler to terminate its current pass on a source file and proceed to the next source file, or the next pass. When a program consists of multiple source files, the END statement can be absent from all but the last file.

In addition to marking the end of a program, the END pseudo-op also designates the start-of-execution address by its' operand. If the END statement is missing, or the operand is left off, the start of execution address is the physically first ORG of a program. The END statement allows an execution address to be specified that is different from the physical start of the program. For example, a program which is structured to have a data area before the executable code can specify the start of execution address as follows:

```
0500      LINK      'SYSQ1'
1000      LINK      'SYSQ2'
1100      ORG       4000H
1200 INBUF   DS        255      ;INPUT BUFFER
1300 BEGIN   MVI      C,1
1400 LXI      H,INBUF
1500 START   CALL     @CIN
1600 CALL     @COUT
1700 INR      C
1800 JZ       BEGIN
1900 CPI      ØDH
2000 JNZ     START
2100 LXI      H,INBUF
2200 CALL     @ONLINEOUT
2300 JMP      BEGIN
9999      END       BEGIN
```

The name of the object file for this program could be used as an implicit command in the MDOS executive. By typing the name of the file, the executive loads the file and transfers program control to the address specified in the END statement.

4.5.4.4 EQU - EQUATE

The EQU pseudo-op equates a literal value to a symbolic name. This pseudo-op requires a label and an operand.

```
1000 TEN      EQU      10  
2000 TWENTY   EQU      2*TEN
```

When the labels TEN, TWENTY are used within the program they will have the value of 10 and 20 respectively.

4.5.4.5 INP - INPUT

The INP pseudo-op allows the operator to assign a value to a label from the system console during pass one of the assembly. The INP statement requires a label. It can have an optional operand. The operand must be an ASCII literal. If an operand is present, it is output as a prompt to the console stream during pass one followed by a question mark (?). ASSM then waits for an input from the console. If no operand is present the INP statement prompts with a question mark.

The input can be in the form of simple or complex expression including literals and/or symbolic names. Symbolic names must have already been defined before the INP statement is encountered during pass one.

Example:

```
1000 TEST      INP      'INPUT'
```

During pass one of the assembly the prompt would be displayed on the system console and the assembler would wait for an input.

```
INPUT  
?
```

4.5.4.6 PRT - PRINT

The PRT pseudo-op outputs the values of its operands to the console stream during assembly pass two. The operands can be simple or complex literals and/or symbolic names. If no operands are present the PRT outputs a carriage return line feed only.

Example:

```
1000 TEST      EQU      7000H  
2000          PRT      'THIS IS A TEST',TEST
```

During pass two the message THIS IS A TEST 7000 is displayed on the system console.

4.5.4.7 TAB - TAB SETTINGS

The TAB pseudo-op changes the tab settings for the assembly listing at assembly time. The TAB settings are initially 15,22,36. The first column is defined as the column at which labels start. The positions of the opcode, operand and comment fields can be changed with the TAB pseudo-op. The statement expects three operands. The first operand is the opcode field tab, the second operand is the operand field tab, and the third is the comment field tab.

If an operand is set to zero, that tab is set to the initial default value. The operands can be simple or complex expressions.

4.5.4.8 NLIST - NO LIST TO PRINTER

The NLIST pseudo-op suppresses the listing of the assembly to the list stream from the point in the source file at which it is encountered until the next occurrence of a LIST pseudo-op.

Note: If the E option (see section 4.5.1) was specified, the LIST and NLIST pseudo-ops will be ignored. In all cases, however, any assembly lines which contain errors will be output.

4.5.4.9 LIST - LIST TO PRINTER

The LIST pseudo-op is used to start a listing at the point at which it is encountered in the source file after a previous NLIST statement has suppressed the listing.

4.5.4.10 FORM - FORM FEED

The FORM pseudo-op is used to control output pagination when the P option is in effect. The FORM statement has two modes. The first causes the assembler to eject the paper to the top of the next page and continue printing. To use this mode the FORM statement must have no operand. The second mode sets the length (in # of lines) of the output page.

1000	FORM 66	;this sets the form length
1010	FORM	;this ejects the page

The example will cause the assembler to use a page size of 66 lines. This means that 58 lines of program text will be output with 8 lines for page number, header and margin.

4.5.4.11 DB - DEFINE BYTE

The DB pseudo-op defines one or more bytes of memory storage. The DB statement has one or more operands.

```
1000 TEST      DB      1,20H,11B,76Q,TEST+3 ;DEFINE BYTES
```

The DB statement's operands can be simple or complex expressions, with the exception that ASCII literals can only be one byte long per operand.

Example:

```
1000      DB      'T','H','I','S'
```

Is valid while:

```
1000 TEST      DB      'THIS'
```

Is not valid.

4.5.4.12 DW - DEFINE WORD

The DW pseudo-op defines one or more two byte words of memory storage in standard Intel low/high address format. The DW statements can have multiple operands which can be simple or complex expressions.

1000 TEST DW 4000H,5557H

4000H would appear as 00 40 and 5557H would appear as 57 55 in the object file.

4.5.4.13 DD - DEFINE DATA

The DD pseudo-op defines one or more two byte words in high/low format.

1000 TEST DD 4000H,5557H

4000H would appear as 40 00 and 5557H would appear as 55 57 in the object file.

4.5.4.14 DT - DEFINE TEXT

The DT pseudo-op is used to define a line of text enclosed between single quotes. The text string can contain any ASCII characters as described in the section on literals.

1000 TEST DT 'ABC'

The following object code would be produced by this example: 41 42 43.

4.5.4.15 DTZ - DEFINE TEXT TERMINATED WITH ZERO

The DTZ pseudo-op is used to define a line of text. When the string is assembled the ASCII code is terminated by a zero.

1000 TEST DTZ 'ABC'

The following object code would be produced by this example: 41 42 43 00.

4.5.4.16 DTH - DEFINE TEXT TERMINATED WITH BIT 8 HIGH

The DTH pseudo-op is used to define a line of text. When the string is assembled the ASCII code of the last character in the string is ORed with 80 hex.

1000 TEST DTH 'ABC'

The following object code would be produced by this example: 41 42 C3.

4.5.4.17 DS - DEFINE STORAGE

The DS pseudo-op is used to set aside storage space. It requires one operand which can be a simple or complex expression that evaluates to the number of bytes to be set aside as storage (1 to FFFF hex). No code is written into the storage area. The assembler adds the operand to the assembler program counter and continues code production at the resulting address. Because the assembler produces scatter loadable object files, any code in the DS area will not be disturbed when the object file is loaded.

4.5.4.18 FILL - FILL STORAGE

The FILL pseudo-op sets aside storage space and fills it with a specified byte. It requires two operands which can be simple or complex expressions that evaluate to the number of bytes to be filled (1 to FF hex) and the byte to be stored (0 to FF hex).

```
1000 TEST      FILL      0AH,8          ;FILL 10 BYTES WITH 8
```

The above example would set aside ten bytes of storage and fill it with 08's.

4.5.4.19 IFF - IF FALSE

The IFF pseudo-op allows conditional assembly of a block of source code statements. The beginning of the block is marked with an IFF statement and the end of the block is marked by an ENDIF statement. The block is assembled if the operand of the IFF statement evaluates to zero.

```
2000 TEST      IFF      LABEL
```

If LABEL is equal to zero then the code between the IFF and the ENDIF will be assembled, otherwise it will not be assembled.

4.5.4.20 IFT - IF TRUE

The IFT pseudo-op allows conditional assembly of a block of source code statements. The beginning of the block is marked with an IFT statement and the end of the block is marked by the ENDIF statement. The block is assembled if the operand of the IFT statement evaluates to non-zero.

4.5.4.21 ENDIF - END OF IF

The ENDIF pseudo-op ends a conditional assembly block. Conditional assemblies can be nested up to 255 deep. A label associated with an IFF or IFT will always appear in the symbol table. However, a label associated with an ENDIF will appear only if the block is active. Statements inside nested conditional assemblies will be active only if the outer IF is active.

4.5.5 ASSEMBLY ERRORS

The assembler is designed to catch typographical and syntactic errors and flag them on listings. These errors are typically oversights; improper use of labels, opcodes, or operands. The assembler cannot catch programming logic errors. A program with flagged errors may still assemble properly depending on the type of error. This is true of syntax errors in listing format statements like TAB. If the TAB statement is used and the operands are left out a syntax error is printed with the line on the listing. The assembler defaults the tab settings to the initial value and continues. The code will be OK (assuming no other errors) and the listing will have the default tabs. In all but one case the assembler will continue the assembly doing the best with each line it encounters and flagging lines that do not make sense. The one exception is a LINK error when the file named in the operand does not exist. This error outputs a FILE NOT FOUND message to the console stream and the assembly is aborted at the point the error is encountered. Syntax errors in the LINK statement do not abort the assembly. The line is flagged and the assembly continues.

Because this is a two pass assembler, pseudo-ops which are evaluated during pass one must have operands that have already been defined before the statement is encountered. This is true of the following pseudo-ops: EQU, ORG, DS, INP, IFF, IFT, FILL. If the operand is not defined before the pseudo-op is encountered, an undefined symbol error (error code U) is output along with the line in error during pass one. Because the program counter is not properly updated at that point in pass one, a phase error will occur in pass two. That is, code will be placed in the right place but references (addresses) in branch instructions will be wrong. The following example illustrates this case:

```
1000 START      ORG      4000H
2000 STORAGE    DS       LENGTH
3000 LENGTH     EQU      40H
4000           JMP      $
```

In the above example line 2000 has a forward reference to line 3000 which is not defined at this point in pass one. During pass one line 2000 will be flagged as having an undefined symbol and the assembly continues. The code produced during pass two will have a phase error as follows:

```
ADDR  B1 B2 B3
4040  C3 00 40
```

The jump to self is in error because the storage could not be properly defined during pass one due to the forward reference.

A quick reference summary of ASSM error messages is shown below. Refer to appendix D for explanations of these message conditions.

A ARGUMENT ERROR
D DUPLICATE LABEL ERROR
L LABEL ERROR
M MISSING LABEL ERROR
O OPCODE ERROR

R REGISTER ERROR
S SYNTAX ERROR
U UNDEFINED SYMBOL ERROR
V VALUE ERROR

4.6 SYMSAVE UTILITY

The SYMSAVE utility is an applications program that may be used to create an equate batch from a symbol table left in memory immediately after an assembly. This equate batch is stored as an editor source file and can be edited by the line editor and assembled by the assembler. The program is invoked from the MDOS executive by typing SYMSAVE followed by an ASCII filename parameter enclosed in double quotes and an optional ASCII mask string enclosed in double quotes.

[unit:]SYMSAVE "<filename>" ["<mask string>"]

The mask string can be up to ten characters long. It is used to save only those symbols in the symbol table that start with the specified mask string.

Example:

ADDR	B1	B2	B3	E	LINE	LABEL	OPCODE	OPERAND
0000					1000		ORG	4000H
4000	C3	00	40		2000	START	JMP	\$
4003	01				3000	DATA1	DB	01
4004	02				4000	DATA2	DB	02
4005	03				5000	DATA3	DB	03
4006					6000	FINISH	END	START

Immediately after the above program is assembled, the symbol table is still resident in memory. To create a disk file of symbols from the above assembly type:

SYMSAVE "TEST"

The file TEST that SYMSAVE creates is an editor compatible source file which looks as follows:

0001	START	EQU	4000H
0002	DATA1	EQU	4003H
0003	DATA2	EQU	4004H
0004	DATA3	EQU	4005H
0005	FINISH	EQU	4006H

If only the data symbols were required, the mask string parameter can be used as follows:

SYMSAVE "TEST1" "DATA"

The file TEST1 looks as follows:

0001	DATA1	EQU	4003H
0002	DATA2	EQU	4004H
0003	DATA3	EQU	4005H

This file contains only the symbols which start with the string DATA.

A symbol equate file can be used in other programs by using the assembler LINK pseudo-op.

Example:

ADDR	B1	B2	B3	E	LINE	LABEL	OPCODE	OPERAND
0000					1000		LINK	'TEST'
0000					2000		ORG	FINISH
4006	3E	01			3000	BEGIN	MVI	A,DATA1
4008	32	03	40		4000		STA	DATA2
400B	C3	00	40		5000		JMP	START
400E					6000		END	BEGIN

By linking the equate batch file with the new program segment all of the symbols defined in the first program segment can be referenced in the new program segment.

4.7 FILECOPY UTILITY

The FILECOPY utility is an applications program that allows files to be copied from one disk to another or onto the same disk under a different filename. To improve speed in the process of copying a file, it uses all available memory after the end of the program as a buffer. To invoke the program from the MDOS executive type FILECOPY followed by a filename enclosed in double quotes and an optional newfilename enclosed in double quotes or a unit number by itself if the copied file is to have the same name as the original.

[unit:]FILECOPY "<[unit:]filename>" "<[unit:]newfilename>"

or

[unit:]FILECOPY "<[unit:]filename>" <unit number>

FILECOPY exits to the MDOS executive when it is done or if it encounters an error condition. The copied file has the same filetype as the original. Any file can be copied regardless of type or origin. This includes BASIC data and program files. Attempting to copy a file onto the same disk without specifying a newfilename results in a DUPLICATE NAME error.

4.8 DISKCOPY UTILITY

DISKCOPY is a special overlay utility that writes an absolute binary copy of one disk onto another. The utility overlays MDOS or BASIC. It uses all available memory during the copying process. The more memory in a system the faster the copying process. On average it takes about two minutes to copy and verify all 315k bytes of a MOD II disk. To invoke the utility from the MDOS executive, type:

DISKCOPY

A sign-on message is output:

MICROPOLIS DISKCOPY VS X.X - COPYRIGHT 1978
SPECIFY UNIT # FOR ORIGINAL (SOURCE) DISKETTE
?

DISKCOPY waits until the unit number is entered. When a number between 0 and 3 is entered it prompts:

SPECIFY UNIT # FOR DESTINATION DISKETTE
?

and waits until the unit number (0 to 3) is entered. It then prompts:

PUT DISKETTES IN SPECIFIED UNITS
TYPE Y WHEN READY
?

and waits for a Y. A note of CAUTION, we strongly recommend placing a write protect tab on the original (source) diskette. It is possible to put the wrong diskette in the wrong drive or type the wrong unit numbers. If your original does not have a write protect tab and you make an error, the original can be overwritten. The write protect tab provides a physical interlock which disables the write electronics.

When a Y is typed DISKCOPY will start the copying process. During copying, the process can be temporarily halted between read source and write destination cycles by typing a control S. The process is restarted by typing any other key except a control C.

The control C will cancel the entry or copy process and prompt:

CANCELLED
MORE ?

If a Y is typed DISKCOPY starts from the top asking for the unit numbers again. If an N is typed DISKCOPY prompts:

PUT SYSTEM DISKETTE IN UNIT 0
TYPE Y WHEN READY
?

When a Y is typed the disk in unit 0 is rebooted. If it's an MDOS diskette MDOS is booted. If the disk in unit 0 is a BASIC only disk or some other bootable system, it will be booted in and sign on. DISKCOPY is overlayed by the incoming system and is no longer in memory.

When the disk has been copied and verified correctly DISKCOPY outputs:

GOOD COPY
MORE ?

If the copy cannot be completed or does not verify correctly DISKCOPY outputs:

PERM I/O ERROR ON DESTINATION DISKETTE

or

PERM I/O ERROR ON SOURCE DISKETTE

indicating where the error occurred.

It is possible for single drive systems to make use of the DISKCOPY utility to copy from one disk to another. In this case it is imperative that the original diskette be write protected with a write protect tab. The procedure involves specifying the same unit number for both source and destination disks. Immediately after typing a Y in response to the TYPE Y WHEN READY prompt, type a control S. The DISKCOPY program will read as many tracks from the source disk as can be contained in main memory and then pause. When the select indicator light goes out, remove the source diskette and insert the destination diskette. Press the return key and as soon as the select indicator light comes on type a control S again. When the select indicator light goes out again, the data from the source disk has been written to the destination disk and one complete cycle is finished. This process is repeated, swaping the source and destination disks in and out until the entire disk is copied. After the last data is written onto the destination disk, the program goes directly into a verifying process and will not pause until this is over. When the source is placed back into the drive and the return key is pressed the system will prompt: GOOD COPY or output an error message as discussed above. At this point the copy is complete.

4.9 ERROR MESSAGES

This section is a summary of the error messages generated by the MDOS shared subroutines. The shared subroutines return an error code in the A register when an error exit occurs. These codes can be passed to the error message output routines to generate the proper error message.

Example:

A file is created by the following BASIC program:

```
10 DIM A$(248)
20 Z$=CHAR$(13):REM CARRIAGE RET
30 OPEN 1 "N:TEXTFILE":REM NEW FILE
40 INPUT A$:REM GET A LINE OF TEXT FROM CONSOLE
50 IF A$="EXIT" THEN 80:REM END INPUT BY TYPING EXIT
60 PUT 1 A$+Z$:REM CONCATENATE CARR RTN AT END
70 GOTO 40:REM LOOP TILL EXIT
80 CLOSE 1
90 END
```

This BASIC program writes one text line per record. Each line is terminated with a carriage return.

The file can be read by the following assembly language routine. Assume it has been assembled and given the name READ and an executable file type of 15. Typing READ "TEXTFILE" loads and executes the program.

0000	LINK	'SYSQ1'	;MDOS EQUATE BATCH
0010	LINK	'SYSQ2'	;MDOS EQUATE BATCH
0020	ORG	@APROGRAM	;APPLICATIONS AREA
0030	START	CALL @CCRLF	;CARRIAGE RETURN LINEFEED
0040		LDA @NASCPAR	;NUMBER OF ASCII PARAMETERS
0050		ORA A	;IF ZERO
0060		JZ @ERRORMES	;ERROR
0070		MVI C,Ø	;@ASCBUFFØ
0080		CALL @TRANSFILENAME	;MOVE INTO @ASCIIBUFFER
0090		MVI B,Ø	;FILE NUMBER
0100		LDA @DRIVENØ	;UNIT NUMBER
0110		MOV C,A	;INTO C FOR OPEN
0120		LXI H,@FILEBUFFERØ	;USE SYSTEM BUFFER Ø
0130		CALL @OPENFILE	;OPEN THE FILE
0140		JC @DISKERROR	;IF ERROR CODE IN A
0150		CALL @RFILEINF	;CHECK THE FILE TYPE
0160		JC @DISKERROR	;IF ERROR CODE IN A
0170		MOV A,B	;FILE TYPE
0180		ANI ØFCH	;TYPE NOT ATTRIBUTES
0190		ORA A	;BASIC DATA FILES=Ø
0200		MVI A,17	;WRONG FILE TYPE MESSAGE
0210		JNZ @DISKERROR	;ERROR
0220	NEXTCHR	MVI B,Ø	;FILE NUMBER
0230		CALL @RFINXPOSI	;READ FILE BYTE AT A TIME
0240		JC EXIT	;END? OR ERROR?
0250		MOV B,C	;CHARACTER FOR OUTPUT
0260		MOV A,B	;INTO A FOR COMPARE
0270		CPI ØDH	;CARRIAGE RET END OF LINE
0280		CZ @CCRLF	;IF CR DO CR LF
0290		CALL @COUT	;OTHER CHR JUST OUTPUT
0300		JMP NEXTCHR	;LOOP TILL END-FILE
0310	EXIT	CPI 2	;END-FILE?
0320		JZ @CLOSEFILE	;CLOSE AND RETURN TO MDOS
0330		STC	;ERROR
0340		JMP @DISKERROR	;ERROR MESSAGE IN A
0350		END	START

Note the handling of the errors in lines 6Ø, 14Ø, 16Ø, 21Ø, 24Ø, and 31Ø-34Ø.

The error codes are summarized below. See appendix D for definitions of the error messages.

CODE#	MESSAGE
0	SYNTAX ERROR
1	PERM I/O ERR
2	END-FILE
3	DISK FULL
4	FILE NOT FOUND
5	DUPLICATE NAME
6	PARM ERR
7	DRIVE NOT UP
8	PERM FILE
9	WRITE PROTECT
10	FILE NOT OPEN
11	COMMAND NOT FOUND
12	BAD FILE #
13	FILE OPEN
14	READ ONLY FILE
15	BAD RECORD #
16	CANCELLED
17	WRONG FILE TYPE
18	INDEX PAST EOR
19	LOAD ADDRESS ERROR

4.10 COPYFILE UTILITY

The COPYFILE utility is an applications program that allows files to be copied from one disk to another on a system with only one disk drive. The utility uses all the available memory after the end of the COPYFILE program as a buffer. To invoke the program from MDOS type COPYFILE followed by a filename:

[unit:] COPYFILE "<[unit:] filename>"

The COPYFILE program signs on:

INSERT SOURCE DISKETTE INTO DRIVE 0
ARE YOU READY?

The system waits for a capital Y to be typed. Any other input is ignored except a control C which returns control to MDOS. When a Y is typed the COPYFILE program loads as much of the source file into memory as it can and then prompts:

INSERT DESTINATION DISKETTE INTO DRIVE 0
ARE YOU READY?

Take the source diskette out of your drive and put the destination diskette into the drive. When ready type a capital Y. Any other input is ignored except a control C which returns control to MDOS. The COPYFILE program creates a file on the destination disk with the same name and filetype as the source file. It then writes the file from memory onto the destination diskette.

If the file is longer than can be held in memory at one time the COPYFILE program will prompt:

INSERT SOURCE DISKETTE INTO DRIVE 0
ARE YOU READY?

The same procedure as above must be repeated until the whole file has been copied. When the copy is complete the COPYFILE program returns to MDOS which prompts:

>

If the COPYFILE program encounters any errors it displays the proper error message and returns to MDOS.

COPYFILE can copy any type or length file. This includes BASIC data and program files.

4.11 DEBUG - THE PDS 8080/8085 PROGRAM DEBUGGER

Micropolis DEBUG is a utility program which facilitates checkout and debugging of 8080/8085 machine language programs. It provides an environment in which the performance of a program can be monitored by starting and stopping program execution at user-specified points and by examining and/or changing the contents of relevant machine registers and memory locations.

DEBUG and the program to be monitored must co-reside in the main system memory. Before DEBUG can be used an executable version must be obtained that uses a 4K block of memory which does not conflict with the program to be debugged. The process of creating an executable version of DEBUG configured for a specific memory space is described in Section 4.12.

DEBUG is invoked from the MDOS executive by typing the name of a configured DEBUG-XX version as created by the DEBUG-GEN utility (see Section 4.12).

Example:

```
>DEBUG-70
```

```
MICROPOLIS DEBUG VS. X.X - COPYRIGHT 1978  
*
```

DEBUG signs on and displays an asterisk (*) which is the DEBUG Executive prompt. Program execution control and machine state examination and modification are performed by entering appropriate commands to the DEBUG Executive.

The program may be executed one instruction at a time (referred to as "single-stepping") with the machine state displayed after each step. Alternatively, the results of a program segment may be examined by placing a breakpoint at the end of the segment. When execution of the program is started, it will execute in real time until the breakpoint is reached. Control of the computer is then returned to the DEBUG Executive and the user may examine the contents of memory and the machine registers.

4.11.1 THE DEBUG EXECUTIVE

Operation of DEBUG facilities is controlled by the DEBUG Executive. The executive prompts the user for a command with the character '*'.

Executive statements are entered by typing characters in sequence on the console keyboard. An executive statement is terminated by pressing the RETURN key. During the entry of a statement each character that is typed is echoed by the executive on the console display. Two control features may be used when entering a line.

- 1) Each time the RUBOUT key is pressed the next previously typed character will be deleted from the line. A backarrow is echoed to the terminal display for each character deleted.

- 2) Holding down the control key and typing X (CNTL/X) will cause all of the current line to be cancelled. A carriage return line feed combination is echoed to the terminal display. The executive is positioned to accept entry of a new line.

An executive statement has the following form:

NAME [<hex> <hex>...<hex>]

The NAME in an executive statement is the name of one of the DEBUG commands. Command names are uppercase only and must not be preceded by any spaces. If the command name is not recognized by DEBUG a SYNTAX error message is displayed.

Executive statements consist of a NAME followed by up to four numeric parameters. There must be at least one space between the NAME and any parameters. All parameters must be separated from each other by at least one space. Entry of an executive statement with too many parameters or without the required spaces between fields will result in a SYNTAX error.

Numeric parameters in executive statements are unsigned hexadecimal values from 0 to FFFF. They represent such elements as memory addresses and register values. Entry of a numeric parameter with a value greater than FFFF or with illegal characters will result in a SYNTAX error.

4.11.2 DEBUG MEMORY RELATED COMMANDS

The DEBUG memory related commands are similar to those available under the MDOS executive (see Section 4.1) with the exception of the LIST command which is unique to the DEBUG context. The syntax of these commands is illustrated with the aid of the following notation:

[] Option brackets. Any parameters enclosed between brackets are optional.

< > Symbol brackets. This space should be replaced by the item described.

4.11.2.1 THE DUMP COMMAND

DUMP <start addr.> [<end addr.>]

The DUMP command outputs a formatted hex display of the contents of a block of memory. Sequential memory locations are shown 16 to a line with the memory address at the left margin. If the <end addr.> is not entered only one byte is displayed. Example:

```
*DUMP 5000 5011  
5000 50 C0 27 77 4F 33 4F CD 7D 9E 98 00 6A FD 82 90  
5010 77 2B
```

Notice that memory bytes are printed out in groups of four so that addresses inside the line may be more easily computed. The grouping follows the address.

```
*DUMP 5002 501F  
5002 27 77 4F 33 4F CD 7D 9E 98 00 6A FD 82 90  
5010 77 2B 54 56 F4 3E 23 2A 34 87 19 3D 21 2C 2A 2B
```

4.11.2.2 THE ENTR COMMAND

ENTR <start addr.>

The ENTR command allows data to be entered into memory directly from the console device. Example:

```
*ENTR 7000  
*78 89  
6F/
```

Three bytes were entered starting at location 7000 hex. These were 78 at 7000, 89 at 7001, and 6F at location 7002.

Typing in an ENTR command places the executive in a special enter mode. While in the enter mode each line of values that is typed is entered into memory when the RETURN key is pressed. Until the RETURN key is pressed the standard backspacing and CNTL/X tools are available for line correction. The last value on the last line must be followed by a slash (/) to properly terminate the enter mode. Entry of a illegal hex value in any line will also cause termination of the enter mode with the message SYNTAX ERROR.

4.11.2.3 THE FILL COMMAND

FILL <start addr.> <end addr.> <byte>

The FILL command fills a block of memory with a specified byte. Example:

```
*FILL 7000 8000 9
```

Each byte of memory in the block from 7000 to 8000 is changed to a 09 by this command.

4.11.2.4 THE MOVE COMMAND

MOVE <source addr. start> <source addr. end> <dest. addr. start>

The MOVE command copies the source block of memory to the destination block. The source block is not changed. The destination block is changed to be an exact copy of the source block. Example:

```
*MOVE 3000 4000 7000
```

Each byte in the memory block from 3000 to 4000 is copied into the corresponding position in the memory block from 7000 to 8000.

4.11.2.5 THE SEAR COMMAND

SEAR <start addr.> <end addr.> <byte>

The SEAR command searches a block of memory for all occurrences of the specified byte and displays all locations with a match. Example:

```
*SEAR 3000 3020 9F  
3004 9F  
3018 9F
```

The block of memory from 3000 to 3020 is searched for all occurrences of a 9F. Location 3004 and location 3018 both contain 9F. No other locations in the block contain 9F.

4.11.2.6 THE SEARN COMMAND

SEARN <start addr.> <end addr.> <byte>

The SEARN command searches a block of memory for all non-occurrences of a specified byte and displays all locations that do not match. Example:

```
*SEARN 3000 3010 67  
3002 09 67  
3006 76 67
```

The block of memory from 3000 to 3010 is searched for all non-matches with the mask 67. Location 3002 contained a 9 rather than a 67, and 3006 contained a 76 rather than a 67.

4.11.2.7 THE COMP COMMAND

COMP <start addr. block1> <end addr. block1> <start addr. block2>

The COMP command compares two blocks of memory and displays address locations that do not compare and the data at those locations. Example:

```
*COMP 5000 500F 5010  
5004 01 09 5014
```

The block of memory from 5000 to 500F is compared with the block of memory from 5010 to 501F. One location fails to compare. Location 5004 contains 01 while the corresponding location, 5014, in the second block contains 09.

4.11.2.8 THE LIST COMMAND

LIST <start addr.> <end addr.>

The LIST command displays the 8080/8085 mnemonic form of the bytes contained in the specified memory block.

```
*DUMP 3000 3008  
3000 CA 02 37 B7 C3 1A 37 CB
```

```
*LIST 3000 3008
3000 JZ 3702
3003 ORA A
3004 JMP 371A
3008 CB *
```

The memory block from 3000 to 3007 contains three 8080/8085 instructions. The byte following the third instruction is not a valid 8080/8085 instruction. This is indicated by the '*' following its value.

4.11.3 DEBUG MACHINE REGISTER AND FLAG COMMANDS

The DEBUG commands in this category are used in conjunction with DEBUG's program execution control features during the process of monitoring a programs performance. Whenever the program execution is paused and the DEBUG Executive is waiting for a command, it is possible to display and/or alter the state of the 8080/8085 registers and flags as they are relative to the last instruction executed in the program being monitored.

4.11.3.1 THE DISR COMMAND

DISR

The DISR command displays the contents of the processor registers and flags along with the next instruction to be executed. In addition the contents of memory at locations addressed by register pairs (e.g. at the address contained in BC) along with the word on the top of the stack are displayed. Example:

```
*DISR
A FLAGS BC DE HL SP @B @D @H @SP
@0 ZCMEH @0000 @0000 @0000 1234 @0 @0 @0 @0000
@0000 LXI SP,1234
```

The second line of the display indicates the processor state. The columns @B, @D, @H and @SP indicate the contents of memory at the addresses contained in the respective register pairs. The flag values are indicated by the presence or absence of a character in the FLAGS column. The Z character indicates a zero condition, the C character a carry condition, the M character a negative sign condition (in the SIGN flag), the E character an even-parity condition and the H character a half-carry condition. Absence of any character indicates the opposite condition on the same flag.

The third line displays the address and mnemonic of the next instruction to be executed. The address of the instruction corresponds to the current value of the 8080 program counter (PC) register in the context of the program that DEBUG is monitoring. The instruction is the one that will be executed next by a single step operation or when program execution is resumed by using a command such as the CONT or RET commands. Note that the state of the registers and flags as displayed by the DISR command reflects their values BEFORE the next instruction shown on the third line is executed.

4.11.3.2 REGISTER SETTING COMMANDS

REGISTERNAME <hex number>

The register setting commands allow the contents of the 8080/8085 processor registers to be set to a specified value prior to the execution of the next instruction in the program being monitored. The general format of a register setting command is a register name followed by a hex data value.

The following register names may be used:

A B C D E H L
BC DE HL SP PC @SP

The first line shows 8 bit registers and the second line shows 16 bit registers. PC is the program counter. @SP designates the 16 bit word on top of the machine stack.

The following examples would change the program counter value to 60F3, the A register value to 7, and the value at the top of the stack to C172.

*PC 60F3
*A 7
*@SP C172

4.11.3.3 FLAG SETTING COMMANDS

The flag setting commands allow the states of the 8080/8085 processor flags to be set or reset prior to the execution of next instruction in the program being monitored. The commands set the flag state according to the mnemonic form used in assembly language. The commands are:

FZ FNZ FC FNC FP FM FPE FPO FH FNH

The FZ and FNZ commands set the state of the ZERO flag to zero or non-zero. The FC and FNC commands set the state of the CARRY flag to carry or no carry. The FP and FM command set the state of the SIGN flag to positive or minus. The FPE and FPO commands set the state of the PARITY flag to even or odd. The FH and FNH commands set the state of the HALF-CARRY flag to half-carry or no half-carry.

Examples:

*FNZ
*FC

The state of the ZERO flag is set to non zero and the state of the CARRY flag is set to carry.

4.11.4 DEBUG MISCELLANEOUS UTILITY COMMANDS

The two commands in this category are the MATH command which is useful in doing address computations while engaged in a debug session, and the RST command which may be needed to avoid conflict with program usage of the processor restarts.

4.11.4.1 THE MATH COMMAND

MATH <hex number> <hex number>

The MATH command performs a 16 bit integer addition and subtraction on the two specified hex numbers. It displays the sum and difference. The MATH command is useful for length and address calculations. Example:

```
*MATH 4 5  
0009 FFFF
```

4+5 equals 9 and 4-5 equals FFFF.

4.11.4.2 THE RST COMMAND

RST <vector number>

DEBUG normally uses the 'RST 6' restart vector of the 8080 or 8085 processor as its mechanism for implementing breakpoints (see Section 4.11.5.1). Some computers and/or a particular program may already be using 'RST 6' for a different purpose. In this case it is possible to change the RST vector used by DEBUG to one of the other available RST's, 1-5 or 7. Example:

```
*RST 7
```

The RST vector used by DEBUG is changed to RST 7 from its default usage of RST 6.

4.11.5 DEBUG PROGRAM EXECUTION CONTROL

DEBUG offers 3 modes of control to monitor progress through a program; the breakpoint mode, the single step mode, and the trace mode. There is a permanent breakpoint facility normally used in conjunction with the commands SET, DISB, CLR, EXEC and REPT. There is a temporary breakpoint facility used in conjunction with the commands CONT and RET. The single-step mode is controlled with the space bar. The trace mode is a form of continuous single-stepping. Use of these modes and their associated commands are detailed in this section.

4.11.5.1 THE BREAKPOINT MODE

Breakpoints provide a means to stop program execution at a given point. When program execution reaches that point control of the processor is transferred to DEBUG. Once in DEBUG, the results of the program section which was executed may be examined or modified.

In the breakpoint mode DEBUG replaces the instruction at a given address with one of the 'RST' instructions of the 8080/8085 (see 4.11.4.2 the RST command). Then DEBUG replaces the three bytes of code at the corresponding 'RST' vector location with a 'JMP' instruction to a routine inside itself. DEBUG then loads the processor's registers with the stored 'user program register' values and transfers control of the processor to the user's program. When the breakpointed instruction address is executed, the 'RST' that DEBUG had placed at that location causes the processor to 'CALL' the RST vector location which then causes the processor to 'JMP' back to DEBUG. DEBUG then stores the processor's registers in the 'user program registers' and replaces the original contents of both the breakpointed instruction and the RST vector location.

Because of the introduction of an 'RST' instruction into the program, when a breakpoint is encountered, at least one level of stack space must be available so that the return address back into the program can be stored. Therefore, when using the breakpoint mode the user must insure that at least one stack level will be available when the breakpoint is encountered.

Note that breakpoints cannot be used to DEBUG ROMed code because an 'RST' instruction cannot be patched into the code.

When a breakpoint is encountered during program execution, DEBUG will display the contents of the program registers in the following format:

A	FLAGS	BC	DE	HL	SP	0B	0D	0H	0SP
13		0000	0000	0000	01A2	00	00	00	14FE

Refer to the DISR command section for a detailed description of this display.

4.11.5.2 PERMANENT BREAKPOINTS

Permanent breakpoints are set using the SET command. These breakpoints are not cleared when control of the processor is returned to DEBUG. Permanent breakpoints are only cleared by the CLR command. Permanent breakpoints can be used as traps on such things as error routines or executive loops.

Note that permanent breakpoints do not leave a 'RST' instruction in the program code. The existence of a permanent breakpoint tells DEBUG to place a breakpoint in the code only when the program is executing. Thus the original program is intact whenever the DEBUG has control of the processor.

4.11.5.3 THE SET COMMAND

SET <breakpoint #> <address>

The SET command defines a permanent breakpoint. The breakpoint # and the hex address at which the breakpoint will be set are entered with the command. More than one breakpoint # may be set with the same breakpoint address. However, an attempt to SET a breakpoint # which is already set will cause the message SYNTAX ERROR to be printed and the command to be ignored. A maximum of 4 breakpoint #'s may be set at any time. Example:

*SET 1 2354

Permanent breakpoint number 1 was set at location 2354 (hex).

4.11.5.4 THE DISB COMMAND

DISB

The DISB command displays all currently SET breakpoints.

Example:

```
DISB  
Ø1 2354  
Ø3 2365
```

The display indicates that breakpoint number 1 is set at address 2354 (hex) and breakpoint number 3 is set at address 2365 (hex). Breakpoints number 2 and 4 are not SET.

4.11.5.5 THE CLR COMMAND

CLR [<breakpoint #>]

The CLR command clears a SET breakpoint. If the optional breakpoint number is not entered, then all SET breakpoints will be cleared. If a breakpoint number is entered but is not currently SET, the message SYNTAX ERROR will be displayed.

Example:

```
*CLR 1
```

Permanent breakpoint number 1 is cleared.

4.11.5.6 THE EXEC COMMAND

EXEC <starting address>

The EXEC command transfers control of the processor to the user's program. The processor's PC register will be set to the entered starting address and execution will start there. If a breakpoint is encountered, control of the processor will be returned to DEBUG. If no permanent breakpoints are SET at that time, the program will retain control of the processor.

Example:

```
*EXEC 3Ø14
```

```
A FLAGS BC DE HL SP ØB ØD ØH ØSP  
ØØ Z C ØØ12 Ø341 3674 Ø195 ØØ ØØ ØØ 3Ø54  
35Ø7 JMP 3643  
*
```

Program execution was started at location 3Ø14 (hex). A breakpoint was encountered at location 35Ø7 returning control back to DEBUG.

4.11.5.7 THE REPT COMMAND

REPT <breakpoint #> <repeat count>

The REPT command transfers control to the user's program until a permanent breakpoint has been hit a given number of times. The breakpoint number entered specifies the breakpoint address and the entered repeat count specifies the number of times it must be hit before control is transferred back to DEBUG. If any breakpoint other than the one being repeated is encountered, control will be transferred back to DEBUG and the repeat operation is cancelled. If the breakpoint # specified in the REPT command is not set, a SYNTAX error is displayed. Example:

```
*SET 1 3000
*00 E 2000 0000 0000 01A0 00 00 00 0000
    3000 DCR B
*00 1F00 0000 0000 01A0 00 00 00 0000
    3001 JMP 3000
*REPT 1 8
    A FLAGS BC DE HL SP @B @D @H @SP
    00 E 1800 0000 0000 01A0 00 00 00 0000
*
```

The breakpoint at location 3000 (hex) is allowed to be passed over 8 times before control is transferred back to DEBUG and the processor state is displayed.

4.11.5.8 TEMPORARY BREAKPOINTS

Temporary breakpoints are one-shot breakpoints which the user instructs DEBUG to place in the program by using the CONT or RET commands. When control of the processor returns to DEBUG, the breakpoints are cleared. Temporary breakpoints are the type normally used to follow the execution of the program from routine to routine.

4.11.5.9 THE CONT COMMAND

CONT [<break 1> [<break 2> [<break 3> [<break 4>]]]]

The CONT command continues execution of the user's program at the current PC location with up to four temporary specified breakpoints. If no temporary breakpoints are specified, then control will never return to DEBUG unless an already specified permanent breakpoint is encountered. Example:

```
*CONT 356F
    A FLAGS BC DE HL SP @B @D @H @SP
    00 M 0120 0341 3674 0195 00 00 00 3054
    3507 DCR A
*
```

Program execution is resumed at the next instruction indicated by the value of the user program PC register and execution continues until the breakpoint at location 356F (hex) is encountered, which returns control back to DEBUG.

4.11.5.10 THE RET COMMAND

RET

The RET command transfers control of the processor to the user's program with a temporary breakpoint set at the address which is on the top of the stack (@SP). This allows the user to 'RETURN' from a subroutine which was 'CALL'ed by the program.

If a breakpoint other than the 'RET' breakpoint is hit, control will return to the DEBUG and the 'RET' breakpoint will be cleared.

Note. The RET command should only be used after a 'CALL' type instruction has been executed or when the top of the stack contains a known return address. Otherwise a breakpoint might be placed at an address which is not a part of the program. (e.g. the last instruction was a 'PUSH' and therefore the top of the stack contains a data word instead of a return address)

Example:

```
*DISR
A FLAGS BC DE HL SP @B @D @H @SP
@0 Z      0000 0000 0000 0000 00 00 00 0000
2A00 LXI SP,3000
*00 Z      0000 0000 0000 3000 00 00 00 3243
2A03 CALL 2B00
*00 Z      0000 0000 0000 2FFE 00 00 00 2A06
2B00 STC
*RET
A FLAGS BC DE HL SP @B @D @H @SP
00 ZC     0000 0000 0000 3000 00 00 00 3243
```

After the second instruction single-step, the RET command causes a temporary breakpoint to be set at location 2A06 (which is the return address on the top of stack) and program execution is resumed. When the program reaches 2A06 control of the processor is returned to DEBUG and the processor state is displayed.

Exception Note: The following program fragment illustrates a special programming construct with which the RET command can not be used.

```
TEXT    Call MESSAGE
       DTH 'SIGNON'
       RET
MESSAGE XTHL
        CALL @LINEOUT
        INX H
        RET
```

If an RET command is given after the call to MESSAGE has just been executed, the return address on the top of the stack is pointing to location TEXT. DEBUG puts a breakpoint at that location. MESSAGE then outputs the Signon text and returns without encountering the breakpoint because the return address has been modified by the called routine.

4.11.5.11 THE SINGLE STEP MODE

The single-stepping mode of program execution allows a detailed inspection of what the program is doing on an instruction by instruction basis. Each time the space bar is pressed in response to the DEBUG '*' prompt, DEBUG causes the next instruction in the program to be executed and displays the contents of the processor registers.

Example:

```
*DISR
A FLAGS BC DE HL SP @B @D @H @SP
13 0000 0000 0000 01A2 00 00 00 14FE
2A00 STC
*13 C 0000 0000 0000 01A2 00 00 00 14FE
2A01 XRA A
*00 Z E 0000 0000 0000 01A2 00 00 00 14FE
2A02 STA 345F
```

At the '*' prompt the user typed a space which caused DEBUG to single-step an instruction and print the resulting register contents on the same line. In the single-step mode of operation, DEBUG makes a local copy of the instruction to be executed in its own buffers. DEBUG then executes the instruction in its buffers and stores the results. The single-step mode does not need to modify the program in any way which allows programs in ROM may be stepped through without problem.

4.11.5.12 THE TRACE MODE COMMAND

TRACE

The TRACE command operates as a continuous single-stepping command. It is used to provide a trace printout of the user's program. During a TRACE the Control S / Control C functions provide pause and break control.

Example:

```
*TRACE
00 E 1800 0000 0000 01A0 00 00 00 0000
3001 JMP 3000
00 E 1800 0000 0000 01A0 00 00 00 0000
3000 DCR B
00 E 1700 0000 0000 01A0 00 00 00 0000
3001 JMP 3000
00 E 1700 0000 0000 01A0 00 00 00 0000
3000 DCR B
00 1600 0000 0000 01A0 00 00 00 0000
3001 JMP 3000
*
```

The program was put in TRACE mode. The Control C key was pressed and stopped the TRACE after 5 instructions had been executed.

Exception Note: The nature of Micropolis disk subsystems is such that a disk access must not be interrupted during the data transfer process which is accomplished by a program loop. For this reason it is not possible to TRACE successfully through portions of a program that call MDOS disk access routines, because the TRACE command effectively interrupts the program once every instruction.

4.11.6 INITIATING A DEBUG SESSION

Both DEBUG and the program to be monitored must be in memory at the same time. The program is loaded into memory first by using the LOAD command from the MDOS executive. DEBUG is then invoked from the MDOS executive by typing the name of a configured DEBUG version as created by DEBUG-GEN (see Section 4.12). The version invoked should not use any memory space that is required by the program to be monitored. Example:

```
>LOAD "TEST PROGRAM"  
>DEBUG  
MICROPOLIS DEBUG V.S. X.X - COPYRIGHT 1978  
*
```

DEBUG signs on and displays its executive prompt. Monitoring of program execution is now controlled from the DEBUG executive.

If the program to be monitored is one which runs in the MDOS Application area, and which requires one or more ASCII or binary parameters that are normally input as part of an MDOS Executive statement, then the way to initiate program execution control is by SETting a permanent breakpoint at the address of the entry point (first instruction) of the program and then EXECuting the MDOS Executive at the warmstart address which is 4E7H. Example:

```
*SET 1 2B00  
*EXEC 4E7  
MICROPOLIS MDOS V.S. X.X - COPYRIGHT 1978  
>APP "ASCIIPARM" 12  
  
A   FLAGS   BC      DE      HL      SP      @B @D @H @SP  
.....  
2B00 LXI SP, 01A0
```

Permanent breakpoint number 1 is set at the program entry point 2B00 hex and execution is begun at the system warmstart address. The MDOS executive signs on and prompts for a command. The APP command is used to transfer control to the start of the program in the application area and to pass one ASCII and one numeric parameter. The breakpoint is then encountered. DEBUG outputs a register display and waits for additional single-step, breakpoint or other commands.

If the program to be monitored is one which can be executed directly without requiring any parameters from the MDOS executive, then the simplest way to initiate program execution control is to set the PC register to the program entry point address. Set the stack pointer to an appropriate address and then use the CONT command to set a temporary breakpoint at the first desired stop point and transfer control to the program. Example:

```
*PC 3000  
*SP 1A0  
*CONT 3020
```

The program counter is set to 3000 hex and the stack is set at 1A0 hex. A temporary breakpoint is set at 3020 hex and program execution is begun at the PC value, 3000 hex. When the temporary breakpoint is encountered DEBUG will output a register display and wait for a new command.

4.11.7 EXITING DEBUG

The user may exit DEBUG in one of two ways. First, the user may simply transfer control of the processor to the program permanently. This is done by clearing all permanent breakpoints with the CLR command and then using the CONT command without setting any temporary breakpoints. Second, the user may simply return to the MDOS executive. This is done by CLRing all permanent breakpoints and then typing:

```
*EXEC 4E7
```

This warmstarts the MDOS executive and leaves the program without any breakpoints set.

4.11.8 RE-ENTERING DEBUG

If control of the processor has been permanently given to the program, DEBUG may be restarted by executing the first address of the 1K boundary on which DEBUG is running. This 'warmstart' procedure will cause any breakpoints which were set in the program to be replaced by the original instructions.

An example of a situation where a restart of DEBUG would be necessary is as follows. A breakpoint was set in the program and control transferred by a CONT command. However, the program entered a loop which had a bug such that the loop was never exited. This caused the system to lock up. The only way to get control back to DEBUG is by restarting DEBUG.

4.11.9 SAMPLE PROGRAM DEBUGGING SESSION

This section contains a sample debugging session as an example of the use of various DEBUG features. The program being DEBUGged is listed in 4.11.9.1. Assume that the program and DEBUG are on disk unit 0 along with an MDOS system. The actual debugging session is shown in Section 4.11.9.2.

4.11.9.1 SAMPLE PROGRAM LISTING

3000	16	00	0000	MVI	D,0
3002	21	80	02	LXI	H,280H
3005	CD	13	30	CALL	SUB
3008	25		0020	DCR	H
3009	C2	05	30	JNZ	LOOP
300C	7D		0040	MOV	A,L
300D	0F		0050	RRC	
300E	6F		0060	MOV	L,A
300F	D2	05	30	JNC	LOOP
3012	C9		0080	RET	
3013	F5		0090		
3014	7C		0100	PUSH	PSW
3015	B5		0110	MOV	A,H
3016	F1		0120	ORA	L
3017	C9		0130	POP	PSW
			0140	RET	

4.11.9.2 DEBUGGING SESSION

The following text is a description of the debugging session listing which follows.

The first three lines show the test program being loaded into memory along with the load and execution of the DEBUG. Once DEBUG is loaded and running it signs on and displays its executive prompt '*'. At that point the PC and SP registers are initialized so that the program can be tested. A permanent breakpoint is set at the final RET instruction so that the program will not return illegally. Then the first three instructions of the program are single-stepped leaving the program inside the subroutine. The subroutine is RETurned from and execution is allowed to proceed to location 300C using the CONT command. Then the TRACE command is used to let execution proceed. The TRACE is cancelled at location 3005. A permanent breakpoint is SET and the REPT command used to allow the inner loop (the CALL, DCR H and JNZ) to execute twice. After two loops control returns to DEBUG. The second breakpoint (the one used for the REPT) is cleared and the program is allowed to execute to the final RET instruction. Having finished testing the program, MDOS is warmstarted.

MICROPOLIS MDOS V.S. 4.0 - COPYRIGHT 1978

>LOAD "TEST" load program into memory
>DEBUG-70 run debug (7000 hex)

MICROPOLIS DEBUG V.S. 4.0 - COPYRIGHT 1978

*SP 1A0 set up a stack
*PC 3000 set up PC

*DISR
 A FLAGS BC DE HL SP @B @D @H @SP
 80 ZC E 0000 0000 0000 01A0 C3 C3 C3 5845
 3000 MVI D,00
 *SET 1 3012 set breakpoint on final RET
 *DISB
 01 3012
 *80 ZC E 0000 0000 0000 01A0 C3 C3 C3 5845 single-step
 3002 LXI H,0280
 *80 ZC E 0000 0000 0280 01A0 C3 C3 11 5845 single-step
 3005 CALL 3013
 *80 ZC E 0000 0000 0280 019E C3 C3 11 3008 single-step
 3013 PUSH PSW
 *RET return from SUB call
 A FLAGS BC DE HL SP @B @D @H @SP
 02 M 0000 0000 0280 01A0 C3 C3 11 5845
 3008 DCR H
 *CONT 300C set temporary break and go
 A FLAGS BC DE HL SP @B @D @H @SP
 01 Z E 0000 0000 0080 01A0 C3 C3 0A 5845
 300C MOV A,L
 *TRACE trace execution
 80 Z E 0000 0000 0080 01A0 C3 C3 0A 5845
 300D RRC
 40 Z E 0000 0000 0080 01A0 C3 C3 0A 5845
 300E MOV L,A
 40 Z E 0000 0000 0040 01A0 C3 C3 0A 5845
 300F JNC 3005
 40 Z E 0000 0000 0040 01A0 C3 C3 0A 5845
 3005 CALL 3013 Control C hit here
 *SET 2 300C set permanent break
 *REPT 2 2 execute inner loop twice
 A FLAGS BC DE HL SP @B @D @H @SP
 20 Z E 0000 0000 0020 01A0 C3 C3 0A 5845
 300C MOV A,L
 *CLR 2 clear breakpoint 2
 *DISB display breakpoints
 01 3012
 *CONT complete program
 A FLAGS BC DE HL SP @B @D @H @SP
 80 ZC E 0000 0000 0080 01A0 C3 C3 0A 5845
 3012 RET
 *CLR clear all breakpoints
 *EXEC 4E7 warmstart MDOS

MICROPOLIS MDOS V.S. 4.0 - COPYRIGHT 1978

4.11.10 USING DEBUG WITH BASIC

DEBUG is designed so that it is independent of the MDOS executive. The only part of PDS on which DEBUG relies is the console and printer I/O logic contained in the RES module. This independence makes it possible to use DEBUG in conjunction with Micropolis BASIC to debug user written machine language routines that BASIC accesses via its DEF FAA construct.

To use DEBUG in this way, its filetype must be changed to an overlay type C, so that it may be accessed with the BASIC LINK statement. This can be done from the MDOS executive by using the TYPE command.

The BASIC program and the machine subroutine should be loaded prior to accessing DEBUG. Also the end of BASIC's memory space must avoid conflict with the machine routine and the particular version of DEBUG being used. When these conditions are met DEBUG can be accessed from the BASIC monitor by using the statement LINK "DEBUG-XX". Example:

MICROPOLIS BASIC V.S. X.X - COPYRIGHT 1978

```
READY
LOAD "BASICPGM"
READY
LIST
10 DEF FAA=16R7010
20 A=FAA (1)
30 PRINT A
40 END
READY
MEMEND 16R7000
READY
LOAD "MROUTINE"
READY
LINK "DEBUG-74"
```

MICROPOLIS DEBUG V.S. X.X - COPYRIGHT 1978

```
*SET 1 7010
*EXEC 4E7
```

MICROPOLIS BASIC V.S. X.X - COPYRIGHT 1978

```
READY
RUN
A FLAGS ....
.....           DEBUG Register display
7010 PUSH H
*
```

From the BASIC monitor the file "BASICPGM" is loaded and listed. It is a program that accesses a machine language routine beginning at address 7010 hex. BASIC's end of memory is set to 7000 hex and the machine routine "MROUTINE" is loaded in above the end of BASIC. A version of DEBUG which starts at 7400 hex is then linked to. In DEBUG a permanent breakpoint is set at 7010 hex, the beginning of the machine routine. Control is then transferred to the system warmstart address 4E7 hex and BASIC signs on again. A RUN command starts execution of the BASIC program, which accesses the machine routine when line 20 is executed. The DEBUG breakpoint is encountered and DEBUG outputs a register display and waits for a command. The machine routine accessed from BASIC may now be stepped through or otherwise debugged as required.

4.12 THE DEBUG-GEN UTILITY

The Micropolis DEBUG program is supplied in a non-configured form embedded within the DEBUG-GEN utility program. Before DEBUG can be used an executable version must be obtained by running the DEBUG-GEN utility.

DEBUG requires 4K of contiguous memory address space which may start on any 1K boundary above the beginning of the MDOS applications area. DEBUG-GEN accepts a memory space specification and creates a version of DEBUG that uses the specified memory space.

From the MDOS executive, DEBUG-GEN is invoked by entering the filename DEBUG-GEN like an executive statement (see Section 4.1.2) or by entering the command LOAD "DEBUG-GEN" followed by the command APP.

The program signs on with the message

DEBUG GENERATION PROGRAM VS. X.X.

and prompts for the memory address at which the DEBUG will run with the message

ENTER PAGE ADDRESS (2B-F0) ?

Type a two digit hexadecimal number that corresponds to the high-order byte of the start address where the DEBUG will run. This address may only be on a 1K boundary. The program will ignore the lowest 2 bits of the response.

DEBUG-GEN creates a type 14 file on disk unit 0 and fills it with the relocated DEBUG system. The file name is "DEBUG-XX" where XX (hex) is the page address entered by the user.

Example:

MICROPOLIS MDOS V.S. 4.0 - COPYRIGHT 1978

>DEBUG-GEN

DEBUG GENERATION PROGRAM V.S. X.X

ENTER PAGE ADDRESS (2B-F0) ? 70

RUN FILE NAMED DEBUG-70

>

In this example a program file named "DEBUG-70" is created on disk unit 0.
This file is a running DEBUG package which will use the memory space from
7000H to 7FFFH.

V MICROPOLIS DISK EXTENDED BASIC

5.0 INTRODUCTION

Micropolis Program Development Software consists of two systems, the Micropolis Diskette Operating System (MDOS) and Micropolis Disk Extended Basic. Both systems are supplied on a MASTER diskette included with each Micropolis disk subsystem. The auto-load bootstrap brings MDOS, which is the first system on the diskette, into memory. Control is transferred from MDOS to BASIC by typing the filename BASIC to the MDOS executive. It is also possible to create a BASIC only diskette so that BASIC may be directly loaded by the bootstrap system. See Chapter II, Section 2. This chapter describes the Micropolis BASIC interpreter and its associated BASIC programming language.

The Micropolis BASIC Interpreter is a special 8080 machine language program supplied on a master diskette included with the disk subsystem. It provides a simple and powerful means for developing, maintaining and executing BASIC programs on 8080 type microcomputer systems. The user interacts with the Interpreter through a terminal which consists of an input keyboard and an output display that may be video or printed hardcopy. Lines entered at the keyboard may be program lines which are stored in the program buffer or commands for immediate execution. A program in the program buffer may be modified in place, stored as a disk file, retrieved from disk and executed under control of the Interpreter. These functions and others are invoked by entering the appropriate immediate commands. Elements of the BASIC Interpreter and its use are described in Sections 5.1 and following.

The original BASIC programming language was developed by John Kemeny and Thomas Kurtz at Dartmouth College, Hanover, New Hampshire; Micropolis Extended Disk BASIC is an elaborated version of that language. BASIC consists of data types, operators, function references and key words which combine to form statements that can be grouped into executable BASIC programs. The details of these language elements and the rules for combining them are described in sections following.

5.1 ENTERING LINES TO THE BASIC INTERPRETER

The BASIC Interpreter is loaded into the main computer memory from MDOS or booted from a BASIC only diskette. At the end of this procedure the message READY is displayed at the terminal. This means that the Interpreter is in control and is waiting for a line to be input.

A line consists of not more than 250 characters typed in sequence. The entry of a line is terminated by depressing the RETURN key. If more than 250 characters are typed prior to the RETURN the Interpreter will ignore the extra characters and respond only to the RETURN, RUBout or CNTL/X keys.

During the entry of a line each character that is typed is echoed by the Interpreter on the terminal display. If the character typed is not part of the BASIC character set (see Section 5.15) it will not be echoed and will not be included in the line entered. The Interpreter also keeps track of the character count as a line is typed and automatically outputs a carriage return / line feed combination to the terminal display when

the count exceeds the width of the display device. This combination is not included in the line count.

Two control features may be used when entering a line.

- 1) Each time the RUBOUT key is depressed the next previously typed character will be deleted from the line. A back arrow is echoed to the terminal display for each character deleted. Neither the deleted characters nor the back arrows are included in the line count.
- 2) Holding down the control key and typing X (CNTL/X) will cause all of the current line to be cancelled. A carriage return line feed combination is echoed to the terminal display; the Interpreter is positioned to accept entry of a new line.

5.2 ENTERING A PROGRAM

The BASIC Interpreter recognizes a line as a program line by the presence of a leading line number. A BASIC program is entered one program line at a time using the normal line entry procedures. The message READY is not displayed after the entry of a program line. This permits consecutive program lines to be entered conveniently. As each program line is entered the Interpreter stores it in a program buffer which it maintains in the computer system's main memory.

Each line of a BASIC program is composed of a line number followed by one or more statements (see Section 5.20) which are separated from each other by a colon (:). The length of a program line may not exceed 250 characters including the digits in the line number. Each line number must be within the range 0 - 65529. Spaces preceding the first digit of a line number are ignored. Spaces embedded in a line number are not legal. All other spaces in a program line are preserved as entered.

Program lines are stored in the program buffer in numeric order by line number. The lines in the buffer at any given time constitute the current program. This program may be modified in three ways.

To insert a new program line, type in the new line including the line number. The interpreter will automatically place the new line in the program buffer in proper sequence.

To modify an existing program line enter the line number and the new statement or statements. The new line will automatically replace the old line in the program buffer that has the same line number.

To delete an existing program line type the line number followed by carriage return. The corresponding line will be eliminated from the program buffer. Note that multiple lines may also be eliminated by using the DELETE command as described in 5.4.

5.3 IMMEDIATELY EXECUTED LINES

Whenever a line is typed in, the Interpreter scans it from left to right until the first non blank character is encountered. If this character is a digit it is assumed to be the first digit of a line number and the line is treated as a program line. (see Section 5.2). If the first non blank character is not a digit then the line is interpreted for immediate execution.

Most normal BASIC statements may be entered for immediate execution. Exceptions are the DEF FN, DEF FA, and DATA statements which are only functional within a program. Multiple statements may be included in an immediate line by separating them with colons (:). BASIC statements are covered in Section 5.20.

Another form of immediate line is the command. Commands are operations which generally make sense only in immediate mode. Most of the commands in BASIC system relate to the program buffer and to the manipulation and execution of BASIC programs. The available commands are described in the following sections.

EDIT, RENUM and MERGE are three commands which function only in the immediate mode. These commands cause a SYNTAX error if they appear in a program.

5.3.1 THE BASIC EDIT COMMAND

EDIT linenumber

A specified line in the BASIC program buffer can be changed without retying the entire line by using the EDIT command. EDIT linenumber is the form of this command. If the specified linenumber is not found in the current program buffer, the message STMT # NOT FOUND is displayed. BASIC processes an EDIT command by copying the specified line into a special editing buffer and setting an invisible pointer to point to the first digit of the linenumber that begins the text line. BASIC is then in the EDIT command mode. A separate set of single key commands is available for editing a line in the special edit buffer. The whole line including the linenumber can be edited.

5.3.1.1 ADVANCING THE BASIC EDIT POINTER - THE SPACE BAR

The invisible edit pointer in the special editing buffer may be advanced one position by pressing the space bar one time. The character to which the edit pointer is pointing will be displayed on the console. This indicates that the edit pointer has passed over the character. The edit pointer is then advanced so that it is now pointing at the next character in the text line immediately after the one that is displayed. The entire line can be displayed in this manner.

5.3.1.2 CHANGING THE NEXT CHARACTER - C

The character to which the edit pointer is pointing in the edit buffer can be changed by typing a c or C, followed by the new character. The new character is printed on the console and replaces the character in the edit buffer at that position. The edit pointer is advanced to point to the character immediately after the new displayed character.

5.3.1.3 DELETING THE NEXT CHARACTER - D

The character to which the edit pointer is pointing in the edit buffer can be deleted by typing a d or D. The deleted character is printed on the console enclosed in backslashes (/). The edit pointer is left pointing at the character immediately after the deleted character.

5.3.1.4 INSERTING CHARACTERS - I

Characters may be inserted into the line or at the end of the line by typing an i or I followed by the characters to be inserted. The insertion begins immediately before the character pointed to by the edit pointer. Characters are inserted in sequence as typed until the insert mode is terminated by typing an escape (1B hex). The edit pointer remains pointing to the same character that it pointed to when the insertion began. The insert mode may also be terminated by pressing the return key. This also terminates the EDIT command and replaces the line in the current text file with the newly edited version from the special editing buffer.

5.3.1.5 LISTING THE LINE IN THE SPECIAL EDITING BUFFER - L

The remainder of the line in the special edit buffer from the position of the edit pointer to the end of the line may be displayed by typing an l or L. The characters are displayed on the console followed by a carriage return-line feed. The edit pointer is reset to the beginning position. This command is useful to see what the line looks like before editing is completed. It may also be helpful to use this command immediately after entering the original EDIT command. This would display the line about to be edited without exiting the editing mode.

5.3.1.6 SEARCHING TO A SPECIFIED CHARACTER - S

The edit pointer may be advanced in the special editing buffer to the first occurrence of a specified character by typing an s or S followed by the character to search for. The characters from the position of the edit pointer up to but not including the searched for character are printed on the console. The edit pointer is left pointing at the first occurrence of the searched for character. If the search argument does not exist in the line then the entire line is printed and the edit pointer is positioned at the end of the line.

5.3.1.7 DELETING TO A SPECIFIED CHARACTER - K

Characters in the special editing buffer from the edit pointer position up to but not including a specified search character can be deleted by typing a k or K followed by the search character. The deleted characters are displayed on the console, enclosed in backslashes (/). If the search argument does not exist in the edit line, then all the characters from the edit pointer to the end of the line are deleted. The edit pointer is left pointing at the search character or at the end of the line.

5.3.1.8 QUITTING THE BASIC EDIT COMMAND MODE - Q

The EDIT command may be aborted without changing the line in the current text file by typing a q or Q. The partially edited line in the special editing buffer is abandoned. No changes are made to the current program buffer. BASIC is ready to accept a new command.

5.3.1.9 COMPLETING THE BASIC EDIT COMMAND - THE RETURN KEY

The line in the special editing buffer can be placed in the current program buffer by pressing the return key at any point while in the BASIC EDIT command mode. If the line number of the line in the special edit buffer matches a line number in the current program buffer, then the edited line replaces the corresponding line in the program buffer and the EDIT mode is completed. If there is no line in the current program buffer with the same line number as the line in the special edit buffer, then the edited line is inserted into the current program buffer in proper line number order. This feature facilitates the copying or repetition of program lines by changing only the line number during the edit.

5.3.2 THE RENUM COMMAND

RENUM

RENUM (starting-number)

RENUM (starting-number, increment)

RENUM (starting-number, increment, first-line-to-change)

Some or all of the lines in the current program buffer can be renumbered by using the RENUM command. This command renames lines in the program, changing line numbers, and line number references that follow branch statements.

These statements are GOTO, GOSUB, ON...GOTO, ON...GOSUB, THEN, RESTORE. The ERROR, END, and ENDPAGE options of the OPEN statement are also affected.

The forms of this command are RENUM, RENUM (starting-number), RENUM (starting-number, increment), and RENUM (starting-number, increment, first-line-to-change). RENUM takes the line number of the first-line-to-change and sets it equal to the starting-number. The line number of each line after the first-line-to-change is then set to the value of the preceding new line number plus the increment value. If no first-line-to-change is specified, the first line in the program buffer is assumed. If no increment value is specified, the value 10 is used. If no starting-number is specified, the value 10 is used. Typing RENUM alone will produce a program numbered from 10 by 10's. Examples:

Assume that the current program buffer contains the following program:

```
9 REM RENUM EXAMPLE PROGRAM
25 INPUT "VALUE";A
30 PRINT "THE SQUARE ROOT OF";A;"IS";SQR(A)
45 GOTO 25
```

The command RENUM (50,30,30) would produce the following:

```
9 REM RENUM EXAMPLE PROGRAM
25 INPUT "VALUE";A
50 PRINT "THE SQUARE ROOT OF";A;"IS";SQR(A)
80 GOTO 25
```

The command RENUM would produce the following:

```
10 REM RENUM EXAMPLE PROGRAM  
20 INPUT "VALUE";A  
30 PRINT "THE SQUARE ROOT OF";A;"IS";SQR(A)  
40 GOTO 20
```

The command RENUM (100) would produce the following:

```
100 REM RENUM EXAMPLE PROGRAM  
110 INPUT "VALUE";A  
120 PRINT "THE SQUARE ROOT OF";A;"IS";SQR(A)  
130 GOTO 110
```

The command RENUM (1000,100) would produce the following:

```
1000 REM RENUM EXAMPLE PROGRAM  
1100 INPUT "VALUE";A  
1200 PRINT "THE SQUARE ROOT OF";A;"IS";SQR(A)  
1300 GOTO 1100
```

Several error conditions are checked before any renumbering is done. This is to safeguard the program against possible damage. As errors are detected error messages are printed along with the lines where the error occurred. No changes are made to the program if any errors are encountered and no renumbering can be successfully carried out until the errors are corrected.

Entering a RENUM command may result in the message NUMBER OUT OF RANGE followed by the line where the error occurred. This is an indication that the renumbering attempt lead to a line number greater than 65529. This can be corrected by entering a RENUM with a smaller increment value that does not cause a line number greater than 65529.

Entering a RENUM command may result in the message MEMORY OVERFLOW. This indicates that renumbering would create a program to long to be run in the memory currently available to BASIC. The program is not renumbered.

Entering a RENUM command may result in the message STMT # NOT FOUND without printing the offending line. This occurs when the specified first-line-to-change does not exist in the program. No change is made. Example; if the program is:

```
10 PRINT "TEST"  
20 GOTO 10
```

The command RENUM (100,10,30) would cause a STMT # NOT FOUND error because there is no line 30 at which to start renumbering.

Entering a RENUM command may result in the message STMT # NOT FOUND followed by the line where the error occurred. This indicates that a branch statement (GOTO,GOSUB, etc.) contained a reference to a line number that does not exist in the program. If this is intentional a stub line should be placed in the program to allow the RENUM to operate. This can be done by typing the line number with a REM statement as a place holder.

Entering a RENUM command may result in the message SYNTAX ERROR. This can be caused by several types of syntactical errors. If the line contains unbalanced quotes or parentheses the SYNTAX ERROR message is displayed, or if renumbering would cause a sequence error in the line numbering (e.g. the lines were numbered 10,20,30,40 and you typed RENUM (10,10,30). This would result in numbers 10,20,10,20 which is not allowed.).

The RENUM command does not change line numbers following LIST, or DELETE. If these statements are used within a program they must be changed manually.

RENUM will not renumber line number references in scientific notation (1E3), or expressions (GOTO 90*8+3). Such references must be changed manually.

If computed GOTO's, GOSUB's or RESTORE's are used in the program they will more than likely be incorrect after renumbering unless extreme care is taken in selecting the renumbering parameters.

Example; if the program is:

```
10 DATA THIS,IS,A,TEST  
20 DATA MORE,TEST,HERE,END  
30 INPUT "WHICH DATA,1 or 2",A  
40 RESTORE (10*A)  
50 READ A$,B$,C$,D$
```

The command RENUM (100,10,30) would renumber the executable part of the program while leaving the DATA statements unchanged.

```
10 DATA THIS,IS,A,TEST  
20 DATA MORE,TEST,HERE,END  
100 INPUT "WHICH DATA,1 OR 2",A  
110 RESTORE (10*A)  
120 READ A$,B$,C$,D$
```

The computed RESTORE on line 110 would still function after the program is renumbered. However, if lines 10 and 20 had been renumbered, then the program would not perform as intended.

The RENUM command can cause a line to expand to a length greater than 250 characters. Such a long line can only be created by RENUM and could not be entered from the keyboard because the input buffer is only 250 characters long. The Basic EDIT command uses the 250 character input buffer during editing. If renumbering causes a line longer than 250 characters and that line is later edited using the Basic EDIT command the line will be truncated at 250 characters by the editor.

5.3.3 THE MERGE COMMAND

MERGE "unit#:filename"

The MERGE command allows existing program files on disk to be incorporated with a program presently in the BASIC program buffer. The form of the command is MERGE "unit#:filename". The unit# is a number from 0 to three followed by a colon. If no unit number is specified, unit zero is assumed.

Lines are merged one at a time from the merge file into the current program buffer, starting with the first line in the merge file. If the line number in the merge file is the same as a line number presently in the program buffer, then the line from the file replaces the line in the buffer. If the line number in the merge file does not match any line number in the program buffer, then the line from the file is inserted in the current program buffer in proper line number order. When all lines from the merge file have been placed in the program buffer the MERGE is complete.

The entire merge file is loaded into memory following the program in the program buffer. Therefore the length of program in the program buffer plus the merge program must be less than the space currently available to BASIC, otherwise a LOAD OVERRUN message is output and the merge does not take place.

The MERGE command also needs some additional buffer space to perform the merge. If there is not enough room the message MEMORY OVERFLOW is output and the merge does not take place.

Large programs are often developed as modules. Each module is written with its test data and debugged separately. The following example shows a three part survey program. Part 1 reads the survey data and tallies the vote. This module is allocated line numbers from 1000 to 2000. The data has been allocated lines 10 to 100 and the printer output module is allocated lines 5000 to 6000.

The program under test uses lines 10-30 as test data, and lines 5000-5010 prints the test results. The program looks as follows in the program buffer:

```
10 REM LIVE DATA SUPPLIED BY OTHER PART OF PROGRAM
20 REM TEST DATA.
30 DATA 1,1,2,2,3,3,4,4,0,1,4,1,99
1000 REM PROCESS SURVEY MODULE.
1010 T=1 :REM INIT TOTAL COUNTER
1020 REM VALID DATA IS 0=NO OPINION,1=YES,2=NO,99=END OF DATA.
1025 READ C
1030 IF C=0 THEN T1=T1+1
1040 IF C=1 THEN T2=T2+1
1050 IF C=3 THEN T3=T3+1
1060 IF C=99 THEN T=T-1:GOTO 5000
1070 IF C<0 OR C>2 AND C<>99 THEN PRINT "ITEM";T;"NOT VALID"
1080 T=T+1
1090 GOTO 1025
5000 REM TEST PRINT OUT ROUTINE
5010 PRINT "NO OPINION=";T1;" YES=";T2;" NO=";T3;" TOTAL=";T
```

This process module with the temporary test data and print logic can be separately tested, debugged and then saved on disk with the command SAVE "PART1".

The real print module can then be developed as follows:

```
DELETE
5000 REM PRINT MODULE
5010 OPEN 1 "*P" ERROR 5200
5020 A$="ZZ9":B$="VZ9"
5030 P1=T1/T:P2=T2/T:P3=T3/T
5040 IF P1+P2+P3<>100 THEN PRINT"PERCENT ERROR":STOP
5050 PUT 1 TAB(60);"NO"
```

```

5060 PUT 1 TAB(10);"RESPONSES";TAB(25);"YES %";TAB(46)"NO %";
5070 PUT 1 TAB(60)"OPINION %"
5080 PUT 1 REPEAT$("=",72)
5090 PUT 1 TAB(12);FMT(T,A$);TAB(25);FMT(T1,A$);TAB(30);FMT(P1,B$);
5100 PUT 1 TAB(45);FMT(T2,A$);TAB(51);FMT(P2,B$);TAB(60);FMT(T3,A$);
5110 PUT 1 TAB(69);FMT(P3,B$)
5120 PUT 1 REPEAT$("-" ,72)
5130 CLOSE 1: STOP
5200 PRINT ERR$:INPUT"CONTINUE",C$:GOTO 5020

```

When the real print module is debugged the command SAVE "PART2" saves it on the disk.

To test the system PART1 and PART2 are combined by typing the commands LOAD "PART1" and a carriage return, and then the command MERGE "PART2" and a carriage return. The combined programs are RUN using the test data. When these parts are debugged they are saved on disk by typing the command SAVE "PROGRAM" and a carriage return.

The data is entered into a separate file as follows:

```

DELETE
10 REM LIVE DATA
20 DATA 1,1,1,2,2,1,0,1,2,1
30 DATA 0,2,2,2,1,2,2,1,1,1
40 DATA 1,1,1,2,2,1,2,1,0,0
50 DATA 99

```

And then saved by typing the command SAVE "DATA" and a carriage return. Several different data files can be produced if needed.

The final program is loaded in two parts by typing the commands: LOAD "PROGRAM" and a carriage return and then MERGE "DATA" and a carriage return. The final program appears as follows:

```

10 REM LIVE DATA
20 DATA 1,1,1,2,2,1,0,1,2,1
30 DATA 0,2,2,2,1,2,2,1,1,1
40 DATA 1,1,1,2,2,1,2,1,0,0
50 DATA 99
1000 REM PROCESS SERVEY MODULE.
1010 T=1 :REM INIT TOTAL COUNTER
1020 REM VALID DATA IS 0=NO OPINION,1=YES,2=NO,99=END OF DATA.
1025 READ C
1030 IF C=0 THEN T1=T1+1
1040 IF C=1 THEN T2=T2+1
1050 IF C=3 THEN T3=T3+1
1060 IF C=99 THEN T=T-1:GOTO 5000
1070 IF C<0 OR C>2 AND C<>99 THEN PRINT "ITEM";T;"NOT VALID"
1080 T=T+1
1090 GOTO 1025

```

```
5000 REM PRINT MODULE
5010 OPEN 1 "*P" ERROR 5200
5020 A$="ZZ9":B$="VZ9"
5030 P1=T1/T:P2=T2/T:P3=T3/T
5040 IF P1+P2+P3<>100 THEN PRINT"PERCENT ERROR":STOP
5050 PUT 1 TAB(60); "NO"
5060 PUT 1 TAB(10); "RESPONSES"; TAB(25); "YES %"; TAB(46) "NO %";
5070 PUT 1 TAB(60) "OPINION %"
5080 PUT 1 REPEAT$("=",72)
5090 PUT 1 TAB(12); FMT(T,A$); TAB(25); FMT(T1,A$); TAB(30); FMT(P1,B$);
5100 PUT 1 TAB(45); FMT(T2,A$); TAB(51); FMT(P2,B$); TAB(60); FMT(T3,A$);
5110 PUT 1 TAB(69); FMT(P3,B$)
5120 PUT 1 REPEAT$("-",72)
5130 CLOSE1: STOP
5200 PRINT ERR$:INPUT"CONTINUE",C$:GOTO 5020
```

5.4 THE DELETE COMMAND

Groups of program lines may be eliminated from the current program buffer by using the DELETE command. There are four forms of this command.

Type DELETE X-Y to eliminate the lines numbered X through Y. Line number Y must be greater than line number X. If either line X or line Y or both are not in the current program buffer a LINE NOT FOUND message will be displayed and nothing will be deleted.

Type DELETE X- to eliminate line X through the last line in the current program buffer. If line X is not in the buffer a LINE NOT FOUND message will be displayed and nothing will be deleted.

Type DELETE -Y to eliminate the first line through line Y in the current program buffer. If line Y is not in the buffer a LINE NOT FOUND message will be displayed and nothing will be deleted.

Type DELETE to eliminate the entire contents of the current program buffer. The buffer will be set to empty and a new program may be entered.

5.5 THE LIST COMMAND

All or part of the program in the current program buffer can be listed on the terminal display device by using the LIST Command. There are four forms of this command.

Type LIST X-Y to display the lines numbered X through Y. Line number Y must be greater than line number X. If either line X or Y are not in the current program buffer the first present line number greater than X or Y will be used instead.

Type LIST X- to display the lines from line X through the last line in the current program buffer. If line X is not in the current program buffer the first present line number greater than X will be used instead.

Type LIST -Y to display the first line through line number Y in the current program buffer. If line Y is not in the current program buffer the first present line number greater than Y will be used instead.

Type LIST to display the entire content of the current program buffer.

5.6 THE SAVE COMMAND

A program in the current program buffer can be stored on disk for later retrieval by using the SAVE command.

SAVE "N: unit number: name of file" is the general form of the command.

The word SAVE and the quotation marks and the name of file must always be present. The name of file may be from 1 to 10 characters long. The characters

which are legal in a file name are the letters A through Z, the digits 0 through 9, and ten special characters including comma (,), dash (-), period (.), slash (/), semi-colon (;), less than (<), equal (=), greater than (>), question mark (?) and at sign (@).

The N: is optional. If it is not included in the command the existing file with the specified name on the specified unit will be overwritten and replaced by the program in the program buffer. If no such file exists the message FILE NOT FOUND will be output. However, if the N: is included in the SAVE command then a new file will be created with the designated name on the designated unit. If N: is used and the file already exists on the specified unit the message DUPLICATE NAME will be output.

The unit number: is also optional. When present it consists of a single digit from 0 to 3 followed by the colon (:). It represents the address of the disk unit on which the specified file is to be replaced or created. If no unit number is specified in the SAVE command, unit 0 is assumed.

5.7 THE LOAD COMMAND

A previously stored program can be retrieved from disk and placed in the current program buffer by using the LOAD command.

LOAD "unit number: name of file" is the general form of the command.

The word LOAD and the quotation marks and the name of file must always be present. The name of file may be from 1 to 10 characters and may use the letters A-Z, the digits 0-9 and the special characters (,), (-), (.), (/), (;), (<), (=), (?), (@), (>).

The unit number: is optional. If it is used it must consist of a single digit from 0 to 3 followed by a colon (:). It designates the address of the disk unit on which the specified file is to be found. If no unit number is specified, unit 0 is assumed.

If the filename specified in a LOAD command is not present on the specified unit the message FILE NOT FOUND will be output. When a program file is successfully loaded it replaces the contents of the current program buffer and all data associated with the last program in the buffer is lost. If the filename specified in the LOAD command is a data file (see section 5.21) which cannot be properly placed in the program buffer, the message NOT A LOAD FILE will be output.

5.8 THE DISPLAY COMMAND

The names of all files which are presently stored on a diskette are recorded in a special file on that diskette. This special file is known as the diskette directory and its name is always DIR. The names currently recorded in a diskette directory can be output to the terminal display by using the DISPLAY command.

DISPLAY "unit number: DIR" is the general form of the command.

The word DISPLAY and the quotation marks and the name DIR must be present. The unit number: is optional. If it is not present unit 0 is assumed. If it is used it must consist of a single digit from 0 to 3 followed by a colon (:). It designates the address of the disk unit whose directory is to be displayed.

The DISPLAY command outputs the filenames five to a line. The first name shown should always be DIR. On disks where it is present the second name shown should always be BASIC.

If the diskette in the specified unit does not contain a valid directory file a PERM I/O ERR message will result because the disk cannot be accessed by the BASIC system.

5.9 THE SCRATCH COMMAND

A file that is stored on disk may be eliminated by using the SCRATCH command.

SCRATCH "unit number: name of file" is the general form of the command.

The word SCRATCH and the quotation marks and the name of file must always be present. The name of file may consist of 1 to 10 characters, including the letters A-Z, the digits 0-9 and the special characters (,), (-), (.), (/), (,), (⟨), (=), (⟩), (?), (@).

The unit number: is optional. If it is used it must consist of a single digit from 0 to 3 followed by a colon (:). It designates the address of the disk unit from which the specified file is to be eliminated. If no unit number is specified, unit 0 will be assumed. If the specified file on the specified unit does not exist the message FILE NOT FOUND will be output.

When a file is SCRATCHed the storage space unused by that file is automatically freed and made available for reallocation.

5.10 THE RUN COMMAND

A BASIC program must be in the current program buffer in order to be executed by the interpreter. This may be accomplished by typing in the program from the input terminal or by using the LOAD command. Once a program is in the current program buffer it may be executed by using the RUN command.

RUN is the form of the command.

When the RUN command is entered, the interpreter resets all disk files to "closed", and frees all memory space previously allocated to variables from the last program run. It then begins execution of the program with the first program line in the buffer and proceeds to execute program lines in

ascending order of line number. This sequence is altered only when particular program statements deliberately change the sequence by transferring control. Each program line is only executed when execution control reaches that line; it is executed each time that this occurs. Execution is halted when an END or STOP statement is encountered or when execution control processes the last line in the current program buffer and it does not alter the control sequence. At this point the interpreter displays the message READY and waits for a line to be entered.

5.11 INTERRUPTING A RUNNING PROGRAM

The execution of a program may be interrupted prior to completion by holding down the CONTROL key and typing C at the input terminal. The interpreter will respond by displaying the message INTERRUPT followed by the message READY.

The interruption generally occurs after the end of whatever program line was being executed when the CONTROL C was entered. In the case of the input statement and whenever characters are being output, the interrupt will occur immediately. Under these circumstances the remainder of the input or output will be lost if a continue is attempted (see section 5.12).

When program execution is interrupted, the value of all program variables remain as last assigned. Any open disk files remain open with file pointers current. Variables may be examined by using immediate PRINT statements and may be altered with immediate assignment statements. These are frequently used aids in debugging programs. However, if the program in the current program buffer is modified (lines deleted, inserted, or changed) then all variable and file information from the interrupted program is lost and the program can no longer be continued.

5.12 CONTINUING AN INTERRUPTED PROGRAM

If an executing program has been interrupted by the CONTROL C procedure and no changes have been made to the current program buffer, then the execution of the program may be continued by using the CONT command.

CONT is the form of the command.

When the CONT command is entered program execution is resumed at the point in the execution control sequence following the last program line executed. If continuation is not possible because no program has been interrupted or because the current program buffer has been altered, the message NOTHING TO RETURN TO will be displayed.

5.13 PROGRAM TRACING COMMANDS

Often, when developing a new program, it is useful to be able to follow the execution on a line by line basis. This capability is provided in the Micropolis BASIC system through the use of the FLOW and NOFLOW commands.

FLOW is the form of the command which enables this program line tracing capability. When the FLOW trace capability is enabled and the RUN command is entered the interpreter displays each program line immediately before it is executed. The FLOW trace remains enabled after the end of a program execution. It must be specifically disabled.

NOFLOW is the form of the command which disables the program line tracing capability.

5.14 BASIC SYSTEM ERROR HANDLING

Whenever the BASIC interpreter attempts to execute an immediate line which has just been entered or the next program line during program execution, it is possible that an error condition may arise. If this occurs the interpreter tries to indicate the problem by displaying an appropriate error message at the terminal.

If the line in error is an immediate line then the error message will be directly followed by the message READY. All or part of the erroneous line may not have been executed.

If the line in error is a program line, the line number and text of the erroneous line are displayed after the error message and before the READY message. All or part of the erroneous program line may not have been executed. Program execution is not continuable after an error.

Appendix A specifies the error messages which may be printed by BASIC and their probable causes.

5.15 THE BASIC CHARACTER SET

BASIC recognizes all printing ASCII characters except the SHIFT 0 (5F HEX) backspace character and the RUB OUT (7F HEX) character. However, lower case symbols may only be used in REM statements and in literal strings. The character set, along with the decimal, hexadecimal and octal values of the corresponding ASCII codes are listed in table 5.1.

5.16 BASIC DATA

BASIC programs operate on two types of data: Numeric and String. Numeric data includes integers and real (floating point) numbers. Character string data items consist of a sequence of characters chosen from the BASIC character set. This includes letters, numbers, special characters and blanks. A data item may be a constant which has an unchanging value, or a variable which may assume different values during the execution of a program. A variable may be either simple or grouped with other variables of like data type into a structure called an array, and referenced as a member of the array.

5.16.1 CONSTANTS

A constant is an unvarying value. It is expressed as its actual value. A constant may be a numeric value, or a character string value.

5.16.1.1 NUMERIC CONSTANTS

Numeric constants may be integers or real numbers.

An integer is a positive or negative whole number which may be defined as a decimal number or in any number base (radix) up to 36. The format of an integer may be:

Integer format: -nn....n Example: -93784

Radix format: -xxRnn....n Example: -16R7B2

Where (-) is an optional sign, xx is the number base, R indicates radix format, and nn....n is the number expressed with the digits 0-9 and the letters A-Z (for radix format). The range of an integer specified in decimal format is 1-5E (2*ISIZE) to 5E (2*ISIZE). See SIZES statement for definition of ISIZE. The maximum value of an integer specified in radix format is 65535. A DIGIT BEYOND RADIX error occurs if a digit or letter is used that is invalid for the radix specified.

A real number is a positive or negative number which includes a decimal point and fractional part or a number expressed in scientific notation. The formats of a real number may be:

Real format: -nn....n.nn... Example: -2.677

Scientific format: -nn...nE-xx Example: 257E-4
-nn...n.nn...E-xx Example: -12.231E14

Where nn...n.nn... represents the number expressed using the digits 0-9 and a decimal point; an optional minus sign (-) denotes a negative number or exponent; E specifies scientific notation and xx represents the exponent expressed with the digits 0-9.

The range of a real number is 1E-61 to (1E62)-1.

BASIC CHARACTER SET IN COLLATING SEQUENCE

CHAR	DECIMAL	HEX	OCTAL	CHAR	DECIMAL	HEX	OCTAL
(space)	32	20	040	@	64	40	100
!	33	21	041	A	65	41	101
"	34	22	042	B	66	42	102
#	35	23	043	C	67	43	103
\$	36	24	044	D	68	44	104
%	37	25	045	E	69	45	105
&	38	26	046	F	70	46	106
.	39	27	047	G	71	47	107
(40	28	050	H	72	48	110
)	41	29	051	I	73	49	111
*	42	2A	052	J	74	4A	112
+	43	2B	053	K	75	4B	113
,	44	2C	054	L	76	4C	114
-	45	2D	055	M	77	4D	115
.	46	2E	056	N	78	4E	116
/	47	2F	057	O	79	4F	117
0	48	30	060	P	80	50	120
1	49	31	061	Q	81	51	121
2	50	32	062	R	82	52	122
3	51	33	063	S	83	53	123
4	52	34	064	T	84	54	124
5	53	35	065	U	85	55	125
6	54	36	066	V	86	56	126
7	55	37	067	W	87	57	127
8	56	38	070	X	88	58	130
9	57	39	071	Y	89	59	131
:	58	3A	072	Z	90	5A	132
:	59	3B	073	[91	5B	133
<	60	3C	074	\	92	5C	134
=	61	3D	075]	93	5D	135
>	62	3E	076	!	94	5E	136
?	63	3F	077	~	95	5F	137

Table 5.1 Standard Collating Sequence

5.16.1.2 STRING CONSTANTS

A character string is a sequence of valid BASIC characters. Entered as a constant, a string must be enclosed in quotes (""). Quotes within a string must be doubled (the constant " is entered as " " " "). The length of a string is the number of characters. The maximum length of all character strings within a program is set by the SIZES statement.

5.16.2 VARIABLES

Variables may be integer, real, or string. The amount of memory used for each of the 3 types can be defined in a SIZES statement before execution of a BASIC program. ISIZE defines the memory space for integers; RSIZE for real variables; and SSIZE for character strings.

5.16.2.1 INTEGER VARIABLES

Integer variables are designated by any letter followed by a percent sign (%).

The range of an integer is from 1-5E(2*ISIZE) to 5E(2*ISIZE). The internal format is 2 BCD digits per byte stored in tens complement. If an attempt is made to store a number that exceeds the range a CONVERSION error occurs.

5.16.2.2 REAL VARIABLES

Real variables are indicated by any letter (not enclosed in quotes) or a letter followed by a digit. The range of a real is 1E-61 to (1E62)-1. The precision or level of accuracy is 2(RSIZE-1) decimal digits.

The Internal Storage Format Is:

Byte 1: 1 bit sign and 7 bit exponent (excess 64)
Byte 2 thru RSIZE: 2 BCD digits per byte.

5.16.2.3 STRING VARIABLES

A string variable is designated by a letter followed by a dollar sign (\$). String variables may have a length of up to 250 characters. The default value of maximum string length is defined by the SSIZE parameter of the SIZES statement. The maximum SIZE of any particular string may be declared in a DIM statement, which supercedes the SIZES statement. If a string which is longer than the maximum length is assigned to a variable, it will be truncated on the right.

The internal format of a string variable is:

Byte 1: Maximum string length
Byte 2: Current string length
Byte 3 thru N: Any character, 1 character per byte
(N= 2+ Maximum string length found in Byte 1)

5.16.2.4 CONVERSIONS

Automatic conversion between integer and real data types is provided which allows mixed-mode arithmetic. A real value is converted to an integer by truncating the fractional part while preserving the sign of the number.

Conversion between string and numeric data types is provided by the STR\$, VAL, FMT, CHAR\$, and ASC functions. See section 5.18.1.2 for description of these functions.

5.16.2.5 ARRAYS

Numeric and character string data may be stored in memory as arrays. An array is a set of variables of one data type (numeric or character) identified by a single variable name. A numeric array is denoted by a single letter or a single letter followed by a percent sign (%) and may have 1 to 4 dimensions. A string array is denoted by a single letter followed by a dollar sign (\$) and may have 1 to 3 dimensions. Both types of array are zero indexed. An array must be declared in a DIM statement which defines the number of dimensions and the index range in each dimension. An array indexing error occurs if an attempt is made to reference an element of an array which has not been defined in a DIM statement.

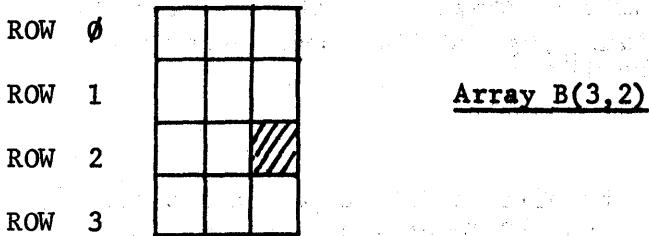
A one dimensional array is a simple linear list in which the elements of the array are stored sequentially in memory. For example, an array A which has a dimension of 4 is stored:

A (0)
A (1)
A (2)
A (3)
A (4)

An element of a one dimensional array is referenced by the array name and by the index of the element within the array, enclosed in parentheses. The 4th element of array A in the above example is A (3). The index may be specified by a constant, as in this example, a numeric variable, or a numeric expression.

A two dimensional array is conceptualized as a table organized by rows and columns. An array B dimensioned as B (3,2) would be represented as:

C	C	C
O	O	O
L	L	L
Ø	1	2



An element of a 2 dimensional array is referenced by the array name and the row and column indices. The shaded element in the above illustration is referred to as B(2,2), where the first index is the row index and the second is the column index.

The elements of a 2 dimensional array are stored sequentially in memory in column major order, that is column by column. The elements of the array B would be stored:

B (Ø,Ø)
B (1,Ø)
B (2,Ø)
B (3,Ø)
B (Ø,1)
B (1,1)
B (2,1)
B (3,1)
B (Ø,2)
B (1,2)
B (2,2)
B (3,2)

As with one-dimensional arrays, the row and column indices may be specified by a constant, a numeric variable or a numeric expression.

3 and 4 dimensional arrays are extensions of the two dimensional concept. An element of one of those arrays is referenced by the array name and the appropriate number of indices.

5.16.3 OUTPUT FORMATS

A numeric data item is converted to a string when it is output to

the terminal. Unless the output format is explicitly specified by use of the FMT function, a numeric value will be output in one of three default formats according to the following rules:

- 1) The negative sign (if present) precedes the number
- 2) A space is output in place of a positive sign
- 3) A space is output following the number.
- 4) A number is either a whole number or a decimal number. A whole number is a number without a fractional part. A decimal number is a number with a whole and a fractional part.
- 5) The output formats are: Whole, Decimal and Scientific.

Whole: (-)xxxxxx \emptyset

Decimal : (-)xxx ... x.xxx \emptyset

Scientific: (-)n.xxxxx E(-)TT \emptyset

(-) = minus sign if negative, blank if positive

x = digit position

n = one non-zero digit

E = signifies exponent

TT = exponent

\emptyset = blank

- 6) The value of an integer variable is output in whole format.

- 7) A constant or the value of a real variable is output as follows:

- a) If the constant or value is a whole number having less than or equal the number of digits specified by RSIZE, then whole format is used.
- b) If the constant or value is a decimal number greater than or equal to .1 and having less than or equal the number of digits specified by RSIZE, then decimal format is used.
- c) Otherwise, scientific format is used.

String data is output without modification.

The maximum output line length is 250 characters. If an attempt is made to output a line longer than the maximum length, e.g., by trying to output 2 strings of 250 characters with the same print statement, the message OUTPUT OVERFLOW is displayed and the line is not printed.

5.17 BASIC OPERATORS

Operators are symbols which specify operations to be performed upon data items. BASIC recognizes 4 classes of operations:

Numeric(arithmetic); String; Relational; and Logical.

5.17.1 Numeric Operators

Numeric operators specify arithmetic operations to be performed upon numeric data items and numeric function references. A numeric data item may be a constant, a simple numeric variable or a numeric array element. Numeric operators are classified as binary operators which perform operations with 2 data items, and unary operators which perform operations upon single data items.

The binary operators are listed below:

<u>Symbol</u>	<u>Operation</u>
\uparrow USE \wedge ()	Exponentiation
/	Division
*	Multiplication
\	Integer Division ($X \backslash Y = \text{Int}(X/Y)$)
-	Subtraction
+	Addition

The unary operators are listed below:

<u>Symbol</u>	<u>Operation</u>
-	Negation
+	No effect

The "+" symbol is recognized as a unary operator to allow constructs such as $A = +7$ and $A = +B$ to be syntactically correct although the "+" has no effect.

5.17.2 String Operators

One operator is recognized for string data items: concatenation. A string data item may be a string constant, string variable or string array element, or a string function reference.

<u>Symbol</u>	<u>Operation</u>
+	Concatenation

The "+" operator yields a string composed of the characters in the string data item to the left of the operator followed by the characters in the string data item to the right of the operator.

EXAMPLE: If A\$ = "ABCD" and B\$ = "EFGH" the operation A\$ + B\$ yields the string "ABCDEFGH".

5.17.3 Relational Operators

Relational operators allow the comparison of the values of numeric or string data items.

The relational operators are listed below:

<u>Symbol</u>	<u>Meaning</u>
<	Less Than
>	Greater Than
=	Equal to
<=	Less than or equal to
>=	Greater than or equal to
<>	Not equal to

A relational operator is used in an expression of the form (Data Item 1 operator Data Item 2) which yields a single value as follows: The values of the two data items are compared. Based upon this comparison if the expression is true, the value "true" (1) is returned. If the expression is false, the value "false" (0) is returned.

EXAMPLE: If A=1 and B=2 then

A < B Yields a value of 1
A=B Yields a value of 0

The data items compared must both be the same data type (numeric or string) or a type error results.

String comparison is performed as follows: Starting from the leftmost character, two strings are compared character-by-character until there is a mis-match or the end of one of the strings is reached. If there is a mis-match, the string containing the character which is higher in the collating sequence is considered "greater" than the other string. If the end of one of the strings is reached without a mis-match and the strings are not of the same length then the longer string is "greater". If the end of one string is reached and the strings are of the same length then the strings are "equal".

5.17.4 Logical Operators

The relational operators as described in section 5.17.3 return a value of "true" or "false". This type of value is referred to as a boolean value and is represented in Micropolis BASIC as an integer. Truth or falsity is determined by converting the integer to a 16 bit binary number. If the least significant bit of the binary number is 0 then the value is false, else the value is true. Logical operators specify operations to be performed with boolean values as described below:

Binary Logical Operators

<u>Operator</u>	<u>Expression</u>	<u>Truth Table</u>		
		<u>VAL 1</u>	<u>VAL 2</u>	<u>RESULT</u>
AND	VAL 1 AND VAL 2	True	True	True
		True	False	False
		False	True	False
		False	False	False

<u>Operator</u>	<u>Expression</u>	<u>Truth Table</u>		
		<u>VAL 1</u>	<u>VAL 2</u>	<u>RESULT</u>
OR	VAL 1 OR VAL 2	True	True	True
		True	False	True
		False	True	True
		False	False	False

Unary Logical Operators

<u>Operator</u>	<u>Expression</u>	<u>Truth Table</u>	
		<u>VAL</u>	<u>RESULT</u>
NOT	NOT VAL	True	False
		False	True

The primary function of the logical operators is to allow the formation of complex expressions which evaluate to a single value of "true" or "false".

EXAMPLE: A <= B AND C = 0

A secondary function is provided by the 16 bit implementation of Boolean values. The logical operators perform the above defined functions across the full 16 bits. This allows you to perform the AND, OR and Complement (NOT) functions in the same manner as the elementary 8080 instructions. The utility of this feature is illustrated in the following example which is a serial I/O handler for an IMSAI SIO board.

```
8000 REM INPUT ROUTINE - RETURNS CHAR IN A
8100 A = IN (3) AND 2: IF A =0 GOTO 8100!: WAIT INPUT READY
8200 A = IN (2) AND 16R7F: RETURN!: MASK PARITY AND RETURN
8300 REM OUTPUT CHARACTER IN A
8400 B= IN (3) AND1: IF B=0 GOTO 8400!: WAIT OUTPUT READY
8500 OUT(2) = A: RETURN!: OUTPUT AND RETURN
```

NOTE: This example will not work for I/O to the terminal device. The BASIC interpreter checks for input from the terminal between execution of BASIC statements and will gobble any character received unless it is a CTL/C.

5.18 BASIC FUNCTIONS

Functions are included in the BASIC language to provide commonly required computations. A function reference consists of the name, followed by its arguments. The arguments are enclosed in parenthesis and separated from each other by commas.

A function returns a single value.

BASIC recognizes two types of functions: Intrinsic functions which are built into BASIC; and user defined functions.

5.18.1 Intrinsic Functions

Intrinsic functions may be classified as numeric, string, special and file. The functions relating to files are discussed in the file I/O section.

5.18.1.1 Numeric Functions

The numeric functions provide most of the commonly used trigonometric and math functions. The math package computes these functions with up to 20 digits of precision, which requires RSIZE to be set less than or equal to 10. Attempting to use the math functions with RSIZE greater than 10 will cause a PRECISION ERROR. The numeric functions are detailed in table 5.2.

Table 5.2 NUMERIC FUNCTIONS

Function Reference	Value
ABS(x)	The absolute value of x, where x is a numeric expression.
ATN(x)	The arctangent of x, where x is a numeric expression. Returns value in the range $-\pi/2$ to $\pi/2$.
COS(x)	The cosine of x, where x is a numeric expression in radians.
EXP(x)	The value of e raised to the power x, where x is a numeric expression.
FIX(x)	The whole number part of x with any fractional part truncated and the sign preserved, where x is a numeric expression.
FRAC(x)	The fractional part of x with the sign preserved, where x is a numeric expression.
INT(x)	The greatest integer not greater than x, where x is a numeric expression.
LN(x)	The logarithm of x to the base e, where x is a numeric expression with a value greater than 0.
LOG(x)	The logarithm of x to base 10, where x is a numeric expression with a value greater than 0.
MAX(x,y)	The greater value, x or y, where both x and y are numeric expressions.
MIN(x,y)	The lesser value, x or y, where both x and y are numeric expressions.
MOD(x,y)	x modulo y which is equal to $x - (y * \text{INT}(x/y))$. Both x and y must be numeric expressions.

Table 5.2 (cont)

Function Reference	Value
RND(x)	Generates a pseudo random number between \emptyset and 1. The argument x is a numeric expression which controls the number generated as follows: If x is non zero, RND generates a number using x as the seed. If $x=\emptyset$, the last random number generated is used as the seed. Repeatedly calling RND with $x=\emptyset$ generates a sequence of pseudo random numbers.
SGN(x)	+1 if the sign of x is positive, -1 if the sign of x is negative, \emptyset if x is \emptyset .
SIN(x)	The sine of x where x is a numeric expression in radians.
SQR(x)	The positive square root of x, where x is a positive numeric expression.
TAN(x)	The tangent of x, where x is a numeric expression in radians.

5.18.1.2 String Functions

String functions are provided to compare strings, manipulate substrings and to convert between numeric and string data types. The string functions are detailed in table 5.3.

Table 5.3. STRING FUNCTIONS

Function Reference	Value
ASC(s\$)	The ASCII code of the first character in string s\$. Returns a numeric value
CHAR\$(x)	Returns the character whose ASCII code is x
FMT(x,y\$)	<p>Returns a string consisting of the value x formatted by the picture contained in string y\$. The argument y\$ can be any expression evaluating to a string. Each character in the string (except a V) represents one character in the result string. The following characters are used to format the digits of a number:</p> <p> 9-- A digit position of the number leading zeroes are output as "0" Z-- A digit position. Leading zeroes are replaced by blanks. V-- Decimal point alignment. If V is not specified, the decimal point is assumed to be at the far right resulting in truncation of the fractional part of the number. \$-- A digit position. If more than 1 \$ appears in the string then the digit position closest to the leading non-zero digit of the number contains a "\$" and the leading zeroes are blanked. *-- A digit position. Leading zeroes are replaced by asterisks. ,-- A comma appearing before the leading digit is replaced with a blank, asterisk or dollar sign according to the context. </p> <p>All other characters are output unchanged. If the number is too large to fit in the format specified, the entire string is filled with question marks (?).</p>

Table 5.3

(continued)

Function Reference	Value
INDEX (x\$, y\$)	The position in string x\$ of the first occurrence of string y\$. If string y\$ is not a substring of x\$, then Ø is returned.
LEFT\$ (x\$, n)	Returns n leftmost characters of x\$.
LEN (x\$)	Returns length of x\$.
MID\$ (x\$, n, y)	Returns y characters from string x\$ starting with character n.
MAX (x\$, y\$)	The greater, string x\$ or string y\$. See the collating sequence in Table 5.1.
MIN (x\$, y\$)	The lesser, string x\$ or string y\$. See the collating sequence in Table 5.1.
REPEAT\$ (x\$, n)	The character string with string x\$ repeated n number of times.
RIGHT\$ (x\$, n)	The n rightmost characters of string x\$.
STR\$ (n)	Converts the number n to a string.
VAL (x\$)	Converts the string x\$ to a number. The contents of x\$ may be numeric digits or a numeric expression. EXAMPLE: If A\$ = "2+2", then VAL (A\$)=4
VERIFY (x\$, y\$)	Verifies that all characters in string x\$ are also in y\$. Returns the position of the first character in x\$ which is not found in y\$. If all characters in x\$ are in y\$ returns Ø.

5.18.1.3 Special Functions

Micropolis BASIC provides several other functions which pertain neither to numbers nor strings. These **special** functions are detailed in Table 5.4.

Table 5.4 **SPECIAL FUNCTIONS**

Function Reference	Value
IN(x)	Inputs a value from I/O port x. The value of x must be greater than 0 and less than 256.
PEEK(x)	Returns the contents of memory location x. The value of x must be greater than 0 and less than 65536.
PGMSIZE	Returns the size of the program currently occupying the program buffer in bytes.
SPACELEFT	Returns the amount of space left in the program buffer in bytes.

5.18.2 User Defined Functions

Micropolis BASIC provides the ability to define two types of functions: BASIC functions and assembly language functions.

5.18.2.1 User Defined BASIC Functions

BASIC allows the user to define functions which consist of BASIC expressions and which are referenced in the same manner as the intrinsic functions. A BASIC function is defined in a DEF statement which has the following form:

DEF FN(letter) (parameter) = expression

Function Name	Optional Parameter	Expression which provides the value of the function
---------------	--------------------	---

The characteristics of a function definition are:

- 1) Function Name--consists of the characters "FN" and one of the letters A-Z yielding up to 26 user-defined BASIC functions.
- 2) Parameter--a function may optionally include a parameter which passes a value to the function when it is referenced. The parameter which appears in the function definition is a "dummy parameter". For example, consider the function defined by:

```
10 DEF FNZ(X) = X↑3+X↑2+A+B
```

The parameter X is a "dummy" in the sense that when the function is referenced, the value passed in the function reference is used in the place of "X". The parameter is only used in the definition to indicate the form of the expression. However, the variables A and B are actual variable names. When the function is referenced, the current values of A and B are used in evaluating the expression.

- 3) Expression--a function may be defined as either a string function or a numeric function by the form of the expression. The expression may be any BASIC expression which yields a single value of the appropriate data type.

A function reference consists of the 3 character function name and the parameter (enclosed in parentheses) if a parameter is included in the function definition. A function reference yields a single value and can be used as a data item in any expression not restricted to constants. A small program using the above defined function is given below as an example:

```
10 DEF FNA(X)=X↑3+X↑2+A+B
20 INPUT A,B,C
30 PRINT FNA(C)
40 GOTO 20
READY
RUN
? 2,3,1
?
? 0,1,2
13
?
INTERRUPT
READY.
```

Below is an example of a string function.

```
5 SIZES(5,4,80)
10 DEF FNB(S$)=REPEAT$(S$,N)
20 INPUT A$,N
30 B$=FNB(A$)+"ISN'T THIS REPETITIVE?"
40 PRINT B$

READY
RUN
? "AGAIN AND ",4
AGAIN AND AGAIN AND AGAIN AND AGAIN AND ISN'T THIS REPETITIVE?

READY
```

See the "DEF FN" statement for more detailed information.

5.18.2.2 Assembly Language Functions

Micropolis BASIC allows the user to define Assembly Language "Functions" which provide linkage to assembly language subroutines. The linkage allows a BASIC program to pass from 1 to 4 arguments to an assembly language subroutine and provides for a result to be passed back to the basic program when the assembly language subroutine returns control.

An Assembly Language Function is defined as follows:

```
DEF FA (letter)= expression
```

The function name consists of the characters "FA" and one of the letters A-Z yielding up to 26 assembly language functions. The expression is a numeric expression which specifies the memory address of the subroutine entry point.

An assembly language function reference consists of the 3 character name followed by a list of arguments enclosed in parentheses.

Examples:

```
100 A = FAA
200 A$ = FAB (B$, C$)
```

Up to 4 arguments may be passed to an Assembly Language Function and 1 result may be passed back as the value of the function reference.

The arguments and result are passed through the following locations which define the subroutine linkage:

<u>LOCATION</u>	<u>LABEL</u>	<u>DESCRIPTION</u>
04BCH	ARG1	Pointer to the first argument
04BEH	ARG2	Pointer to the second argument
04C0H	ARG3	Pointer to the third argument
04C2H	ARG4	Pointer to the fourth argument
04C4H	NARGS	Number of arguments passed
04C5H	RSIZE	Values of RSIZE, ISIZE
04C6H	ISIZE	and SSIZE as described
04C7H	SSIZE	in Section 5.20.26
01A0H	RESULT	250 byte result buffer

When an assembly language subroutine is referenced, the basic interpreter sets the pointers in the linkage table to point to the values of the arguments, indicates the number of arguments passed in NARGS, and calls the subroutine. When the subroutine returns, the interpreter expects to find the value returned by the subroutine, if any, in the result buffer.

The format of the arguments pointed to by ARG1-4 and of the result returned is:

BYTE 0 - Type Indicator
1 - Real
2 - Integer
3 - String
BYTE 1-N - Refer to Section 5.16.2 "Variables" for the internal storage format for each variable type. The length of each variable type is specified by RSIZE, ISIZE and SSIZE.

The general procedure for using assembly language subroutines is as follows:

- 1) Load BASIC from MDOS or directly from a BASIC only SYSTEM DISK.
- 2) Set the memory space used by BASIC using the MEMEND statement to reserve space above BASIC for your subroutine.
- 3) Load the subroutine using the LOAD command. Execution of an object file load within a program is allowed.
- 4) Define the name and entry point of the subroutine with the DEF FA Statement. The subroutine may now be used.

The assembly language program example on the following pages demonstrates most of the principles involved in passing arguments and returning results. It was created by using the assembly language development tools of the MDOS system. The source program was entered with LINEEDIT and then assembled with ASSM to produce an object file named CONCAT which can be loaded by BASIC.

The CONCAT subroutine expects two string arguments to be passed and returns a string which is composed of the second argument concatenated with the first argument. If only one argument is passed, the result string is "argument error". If both arguments are not strings, the string returned is "type error".

Note: This example is not complete - a proper subroutine of this type would have to handle the special cases of null strings and checking to see if the maximum string length has been exceeded, etc.

```

0000 ****
0000 *
0000 *      ASSEMBLY LANGUAGE      *
0000 *      SUBROUTINE LINKAGE    *
0000 *      DEMO 1978            *
0000 *
0000 ****
0000 *
0000 *
0000 *
0000 01A0 RESULT EQU 1A0H
0000 04BC ARG1 EQU 4BCH
0000 04BE ARG2 EQU ARG1+2
0000 04C0 ARG3 EQU ARG1+4
0000 04C2 ARG4 EQU ARG1+6
0000 04C4 NARGS EQU ARG1+8
0000 04C5 RSIZE EQU ARG1+9
0000 04C6 ISIZE EQU ARG1+10
0000 04C7 SSIZE EQU ARG1+11
0000 *
0000 *
0000          ORG 6040H
6040 *
6040 * THIS DEMO ACCEPTS TWO ARGUMENTS
6040 * WHICH ARE STRINGS AND RETURNS
6040 * ARG1 CONCATENATED WITH ARG2.
6040 *
6040 *
6040 3A C4 04 NBRCK LDA NARGS ;CHECK FOR TWO
6043 FE 02 CPI 2 ;ARGUMENTS.
6045 C2 8D 60 JNZ NBRER ;IF NOT TWO - ERROR,
6048 2A BC 04 TYPCK LHLD ARG1 ;ELSE, CHECK TYPE OF
604B 7E MOV A,M ;ARG1. IT MUST
604C FE 03 CPI 3 ;BE A STRING.
604E C2 87 60 JNZ TYPERR ;IF NOT - ERROR,
6051 2A BE 04 LHLD ARG2 ;ELSE, CHECK ARG2
6054 7E MOV A,M ;IT ALSO MUST
6055 FE 03 CPI 3 ;BE A STRING.
6057 C2 87 60 JNZ TYPERR ;IF NOT - ERROR.
605A *
605A * BOTH ARGUMENTS ARE VALID STRINGS
605A *
605A 11 A0 01 LXI D,RESULT ;SETUP RETURN
605D 3E 03 MVI A,3 ;PARAMITER AS A
605F 12 STAX D ;STRING TYPE.
6060 13 INX D ;SKIP OVER
6061 13 INX D ;LENGTH FOR
6062 13 INX D ;NOW
6063 AF XRA A ;ZERO LENGTH
6064 47 MOV B,A ;COUNTER.
6065 2A BC 04 LHLD ARG1 ;MOVE FIRST
6068 CD 79 60 MSTR CALL MOVE ;ARGUMENT TO RESULT
606B 2A BE 04 LHLD ARG2 ;MOVE SECOND
606E CD 79 60 CALL MOVE ;AGRUMENT TO RESULT
6071 78 MOV A,B ;GET LENGTH COUNT
6072 32 A1 01 STA RESULT+1 ;PUT COUNT INTO
6075 32 A2 01 STA RESULT+2 ;RESULT.
6078 C9 RET ;DONE, RETURN TO BASIC

```

6079 *
 6079 * MOVE ARGUMENTS TO RESULT.
 6079 * HL REGISTERS HAS ARGUMENT ADDRESS.
 6079 * DE REGISTERS HAS POSITION IN RESULT.
 6079 * B REGISTER IS COUNT
 6079 *
 6079 23 MOVE INX H ;SKIP TYPE
 607A 23 INX H ;SKIP MAX LENGTH
 607B 4E MOV C,M ;GET LENGTH OF STRING
 607C 23 INX H
 607D 7E MOVE1 MOV A,M ;GET CHARACTER
 607E 12 STAX D ;PUT IT INTO RESULT
 607F 13 INX D ;NEXT
 6080 23 INX H
 6081 04 INR B ;COUNT +1
 6082 0D DCR C ;LENGTH -1
 6083 02 7D 62 JNZ MOVE1 ;LOOP TILL DONE
 6086 C9 RET ;DONE
 6087 *
 6087 *
 6087 21 9E 60 TYPERR LXI H,TYPMSG
 608A C3 90 60 JMP EMSG
 608D *
 608D 21 AB 60 NBRER LXI H,NBRMSG
 6090 11 A0 01 EMSG LXI D,RESULT ;PUT MESSAGE IN RESULT
 6093 3E 03 MVI A,3 ;STRING TYPE
 6095 12 STAX D
 6096 13 INX D
 6097 13 INX D
 6098 13 INX D
 6099 AF XRA A ;ZERO COUNT
 609A 47 MOV B,A
 609B C3 68 60 JMP MSTR ;MOVE TO RESULT
 609E *
 609E * ERROR MESSAGES
 609E *
 609E 00 00 0A TYPMSG DB 0,0,10
 60A1 54 59 50 DT 'TYPE ERROR'
 60A4 45 20 45
 60A7 52 52 4F
 60AA 52
 60AE *
 60AB 00 00 0E NBRMSG DB 0,0,14
 60AE 41 52 47 DT 'ARGUMENT ERROR'
 60B1 55 4D 45
 60B4 4E 54 20
 60B7 45 52 52
 60BA 4F 52
 60BC *
 60BC END NBRCK

Listing of and output from a BASIC program that utilizes
the CONCAT assembly language routine.

```
READY
LIST
10 DIM A$(250),B$(250),C$(250)
20 MEMEND 16R5FFF
30 LOAD "CONCAT"
40 DEF FAA=16R6040
50 INPUT A$
60 INPUT B$
70 C$=FAA(A$,B$)
80 PFINT C$
90 GOTO 50
READY
RUN
? 12345
? 67890
1234567890
? NOW IS THE TIME
? FOR ALL GOOD MEN
NOW IS THE TIMEFOR ALL GOOD MEN
?
INTERRUPT
60 INPUT B$
READY
PRINT FAA(A$)
ARGUMENT ERROR
READY
PRINT FAA(A,B)
TYPE ERROR
READY
PRINT FAA("12345","67890")
1234567890
READY
```

Pages 5-30 through 5-32 left blank intentionally.

5-30

5.19 BASIC EXPRESSIONS

A BASIC expression is a combination of data items and function references connected by operators. An expression specifies an operation or series of operations that yields a single value, which is referred to as the value of the expression. Data items may be constants, simple variables, or array elements. Operators may be arithmetic, string, relational, and logical.

5.19.1 Evaluation of Expressions

BASIC contains a precise set of rules which define the manner in which expressions are evaluated:

- 1) Operator Precedence -- Operators encountered in an expression are performed in the following order:
 - 1) Function references
 - 2) Unary operators
 - 3) Arithmetic & string operators
 - 4) Relational operators
 - 5) Logical operators
- 2) Operators which have the same level of precedence are performed in the order in which they are encountered in scanning the expression from left to right.
- 3) The normal rules of precedence & order of evaluation may be overridden by the use of parentheses to partition an expression into subexpressions. Nesting of subexpressions is limited by the overall complexity of the expression. If an expression is too complex it may cause a STACK OVERFLOW error. In this case, the expression should be broken into two expressions.
- 4) Expressions containing subexpressions are evaluated from the innermost subexpression outward to the next level of parenthesis until all parenthetical expressions have been evaluated. Within a subexpression the rules given for operator precedence and order of evaluation apply.

5.19.2 Numeric Expressions

A numeric expression consists of numeric function references, numeric operators, and numeric data items and evaluates to a numeric result. Operations are performed in the following order:

- 1) Function references
- 2) Unary + and -
- 3) Exponentiation
- 4) Division and Multiplication
- 5) Integer division
- 6) Addition and Subtraction

Parentheses may be used to force evaluation in the exact order desired.

EXAMPLES:

1. $2*3+7*4$

This expression is evaluated as follows: ($V(x)$ indicates the value of x)

- 1) $2*3$ yields 6
- 2) $7*4$ yields 28
- 3) $V(2*3) + V(7*4)$ yields 34

2. $2*(3+7)*4$

This expression is evaluated as follows:

- 1) $3+7$ yields 10
- 2) $2* V(3+7)$ yields 20
- 3) $V(2*V(3+7)) *4$ yields 80

5.19.3 String Expressions

A string expression consists of string function references, string operators, and string data items and evaluates to a string result. Operations are performed in the following order:

- 1) Function references
- 2) Concatenation

EXAMPLE: Let $B\$ = "The number is"$

$B\$+STR\(134)

This expression is evaluated as follows:

- 1) $STR\$(134)$ yields " 134 "
- 2) $V (STR\$(134))$ is concatenated with the current value of $B\$$ which yields "The number is 134 "

5.19.4 Logical Expressions

A logical expression consists of numeric and string expressions combined with relational and logical operators. The value of a logical expression is a Boolean value. Operations are performed as follows:

- 1) Function references are performed.
- 2) The NOT operation is performed.
- 3) Numeric and string expressions are evaluated.
- 4) Relational operations are performed
- 5) The AND operations are performed
- 6) The OR operations are performed
- 7) Parentheses may be used to force evaluation in the exact order desired

EXAMPLE:

$A+2 \leq 3 \text{ AND } B+3 < 5 \text{ OR NOT } (B\$ = "A")$

This expression is evaluated as follows:

- 1) The value of B\$ is compared with "A" (Note: if parentheses had not been used, BASIC would have tried to perform NOT B\$ which would have given an error) Temporary result T1 is set =1 if B\$="A" else is set =0
- 2) T1 is complemented
- 3) A+2 is evaluated
- 4) B+3 is evaluated
- 5) The value of A+2 is compared with 3 and a temporary result T2 is set =0 if $A+2 \geq 3$ or 1 otherwise.
- 6) The value of B+3 is compared with 5 and T3 is set =0 if B+3 is greater than or equal to 5 else is set =1.
- 7) T2 is ANDed with T3 yielding T4
- 8) The value of the expression is obtained by OR'ing T4 with T1

Note: The NOT operator complements the 16 bit representation of Boolean values so the final value of this expression is 65535 if true and 65534 if false.

5.20 BASIC STATEMENTS

BASIC statements specify operations to be performed in a BASIC program, and describe the data and operating environment of the program.

Every BASIC statement consists of a keyword followed by a list of zero or more expressions which specifies the operation to be performed by the statement.

Multiple statements may be included in the same program line separated by the colon (:) (see section 5.2).

The statements included in the BASIC language are listed alphabetically and described in detail in the following pages. Conventions of notation used are:

- 1) { A } Indicates a choice of one of the items enclosed.
- 2) [] Indicates optional items.
- 3) Parentheses () used in definitions must be included as illustrated.

5.20.1 DATA { numeric constant } , { numeric constant } . . .
 { string constant } , { string constant } , . . .

150 DATA 25, "APRIL 1, 1977", 26E-3

The DATA statement is used to define a list of data internal to a BASIC program which may be accessed with the READ statement. When a BASIC program is started, the DATA pointer is initialized to point to the first data item in the first DATA statement in the program. When a READ statement is executed, one value is read from the list for each variable specified and the pointer is advanced to point to the next data item. When the data items in a DATA statement are depleted, the pointer is set to point to the first data item in the next DATA statement encountered in the program such that all the data values contained in DATA statements constitute a contiguous list. The RESTORE statement can be used to re-position the DATA pointer to point to the first data item of any DATA statement within the program.

The DATA statement is non-executable and may therefore appear anywhere within a program.

5.20.2 DEF FN letter [(function parameter name)] = expression

```
10 DEF FNA = X+Y+Z  
100 DEF FNL(A) = (4*3.1415*A)/3  
150 DEF FNR(M$) = REPEAT$(M$,5)
```

The DEF FN statement is used to define a function. The name of the function defined is "FN" followed by one of the letters A-Z. Each function name may be defined only once in a given program.

For example, if the statement 110 DEF FNN= 3.1415*R2 were used in a program. 260 DEF FNN (M\$)=REPEAT(M\$,5) could not be used because the function names are identical. The statement 260 DEF FNM (M\$)=REPEAT(M\$,5) would be legal.

A function parameter is optional. If present, it is a dummy parameter and its name may be any simple variable name. A function will return a numeric or string value depending upon the form of the expression.

A DEF FN statement is non-executable and may appear anywhere in a program.

5.20.3 DEF FA letter = numeric expression

```
90 DEF FAA = 16R7000
```

The DEF FA statement is used to define a function which provides linkage to an assembly language subroutine. The function name consists of the letters "FA" and one of the letters A-Z. The expression contains the starting address of the assembly language subroutine. See section 5.18.2.2 "Assembly Language Functions" for details of linkage and passing arguments.

5.20.4 DIM letter [%] (I1, I2, ... I4)
 DIM letter \$(length)
 DIM letter \$(I1, ... I3,length)

```
10 DIM A (2,4)
20 DIM B%(2,3,4,5)
30 DIM A$(40)
40 DIM A$(2,3,40)
```

The DIM statement is used to define the maximum length of string variables and to define the number of dimensions and index ranges for arrays.

The first form of the DIM statement is used to define a numeric array. The array name consists of one of the letters A-Z. An optional percent sign (%) may follow the letter to denote an integer array. The array may have 1 to 4 dimensions as defined by the number of parameters (I). The value of each I defines the maximum value of the index for that dimension.

The second form is used to set the maximum length of a string variable. The name of the variable is one of the letters A-Z followed by the dollar sign (\$). The length specified must be less than or equal to 250 and overrides the default length specified in the SIZES statement.

The third form is used to define a string array. The array name consists of one of the letters A-Z followed by the dollar sign (\$). A string array may have 1 to 3 dimensions as defined by the number of parameters (I) specified. The value of each I defines the maximum value of the index for that dimension. The last parameter specified in the parameter list is the maximum length of each string element.

Dimension statements are executed dynamically, therefore the parameters may be either constants or expressions.

5.20.5 END
 10000 END

The END statement is optional in BASIC. Execution will terminate when the END statement is executed and may not be continued with the CONT command. It is recommended that an END statement be the last statement of a program to serve as a listing aid. Its presence ensures that the listing is complete.

5.20.6 EXEC string expression

```
100 EXEC A$
```

The EXEC statement is a feature unique to Micropolis BASIC. The EXEC statement causes the string expression to be passed to the BASIC Interpreter and to be executed as a statement. The expression may consist of one or more BASIC statements separated by colons(:). The expression passed is checked for syntax errors and then executed if valid. The following program is given as an example of the power inherent in this statement. The program accepts arithmetic statements from the terminal and prints the results -- effectively operating the terminal as a desk calculator.

LIST

```
10 INPUT A$: EXEC "PRINT "+A$: GOTO 10
READY
RUN
? 2+2
4
? SIN(3.14159/4)
.70710595
?
```

5.20.7 FLOW

```
10 FLOW
```

The FLOW statement turns on the program trace feature which aids in debugging BASIC programs. The program trace will output to the terminal the program line of each statement which is executed. The program line will be output again if the THEN portion of an IF . . . THEN statement is executed. The program trace is turned off by the NOFLOW statement.

5.20.8 FOR numeric = numeric variable expression TO numeric expression [STEP numeric expression]

30 FOR X = 1 TO 30
40 FOR Y = 30 to 0 STEP -1
50 FOR X = A to B

The FOR statement initiates the repeated execution of a set of statements following it. The set begins with the statement immediately following the FOR statement. The set ends with the NEXT statement that contains the same variable as the FOR statement. The numeric variable controls the number of times the set of statements is to be executed and is called the loop variable. The set of statements to be executed is referred to as a FOR . . . NEXT loop.

The expressions specify the initial value of the loop variable, the terminal value of the loop variable, and the value to be added to the loop variable after each pass through the loop (step). The step parameter is optional; when not specified, a default value of +1 is used.

The statements within the FOR . . . NEXT are executed until the value of the loop variable is stepped outside the range defined by the initial and terminal values.

The STEP value can be negative, as in:

20 FOR I = 100 to 0 STEP -10

This statement would cause the initial value of the loop variable I to be set at 100, subtract 10 from the loop variable each time the loop was completed, and terminate executing the loop when the loop variable contained the value 0.

The statement 15 FOR J = 0 TO 0 would cause the FOR loop to be executed one time. That is, the statements between the FOR J . . . and the NEXT J statements would be executed once before the loop variable of 0 + 1 would be compared to the limit value of 0. At this point the loop variable limit would have been exceeded and program execution would fall through to the next line number.

A set of FOR . . . TO. . .NEXT statements may be nested within one or more sets of FOR. . .TO. . .NEXT statements. For example:

10 FOR K = 1 TO 90
20 FOR L = 1 TO 15
30 PRINT K,L
40 NEXT L
50 NEXT K

When nesting FOR. . .TO. . .NEXT statements it is imperative that the inside loop (in this case the L loop) be completely enclosed within the outer loop.

If the above statements had been entered incorrectly as follows:

```
10 FOR K = 1 TO 90
20 FOR L = 1 TO 15
30 PRINT K,L
40 NEXT K
50 NEXT L
```

The error message "MISSING FOR" would occur when the "NEXT L" statement is encountered.

If a GOTO or IF. . .THEN statement is executed from within a loop, the program execution will continue in a normal manner. BASIC will continue the loop from the current value of the loop variable if the loop is re-entered at some later point.

5.20.9 GOSUB {linenumber}
{line number or numeric expression}

210 GOSUB 1000

The GOSUB statement causes a set of statements to be executed as a subroutine.

When a GOSUB statement is executed, control is transferred to the first statement whose line number is specified in the GOSUB statement. The referenced line number and all statements following it will be executed until a RETURN statement is encountered. Control is then returned to the statement following the GOSUB. Consider the following:

```
150 GOSUB 210: PRINT A + B  
160 END  
210 INPUT X,Z  
220 A = X + 1: B = Z-10  
230 RETURN
```

When line number 150 is executed, control is transferred to line number 210. Line 210 and 220 are executed, then 230, the RETURN statement. The RETURN causes control to be transferred to the statement immediately following the GOSUB. Therefore, the sum of A + B will be printed before the program ends.

GOSUB statements can be nested. That is, a subroutine can contain a GOSUB statement that references another subroutine. Control will be returned to the first subroutine when the RETURN statement of the second is executed. The message STMT # NOT FOUND will be output if a GOSUB statement references a line number that does not exist in the program.

BASIC allows an expression to be used as the line number. If this is done, care must be taken to insure that the value of the expression is a positive real number. The fractional part of the number will be truncated in forming the line number. A NUMBER OUT OF RANGE error will occur if the number is invalid.

5.20.10 GOTO $\left\{ \begin{array}{l} \text{line number} \\ \text{numeric expression} \end{array} \right\}$

100 GOTO 5000
200 GOTO A+B

The GOTO statement causes control to be transferred to the first statement in a specified program line. A GOTO statement may reference any line in a program, including its own line. The line number may be specified as a constant or a numeric expression. Care must be taken to ensure that the expression evaluates to a positive real value. The fractional part of the number will be truncated in forming a line number. If the value is invalid, a NUMBER OUT OF RANGE error will occur. If the line number does exist in the program, a STMT # NOT FOUND will occur.

5.20.11 IF logical expression $\left\{ \begin{array}{l} \left[\begin{array}{l} \text{THEN} \end{array} \right] \text{ STATEMENT} \quad \left[:\text{STATEMENT} \right] \\ \text{THEN} \quad \text{line number} \end{array} \right\}$

10 IF A < B THEN PRINT "*"
20 IF A = 2 GOTO 100
30 IF A = 4 THEN 100
40 IF A = 2 AND C = 3 THEN D = 2: GOTO 1000

The first form of the IF statement provides conditional execution of one or more statements based upon the value of a logical expression.

The statements subject to conditional execution must all reside within the same program line as the IF statement. If the logical expression evaluates to "true", then the statements are executed. If the expression evaluates to "false", then all remaining statements within the line are ignored. The keyword THEN is optional in this form.

The second form of the IF statement provides a conditional program branch based upon the value of a logical expression. If the expression evaluates to "true", control is transferred to the first statement in the specified program line. If the expression evaluates to "false", program execution continues at the next sequential program line. The line number must be specified as a constant. If the line number specified does not exist in the program, a STMT # NOT FOUND error occurs.

5.20.12 INPUT ["promptstring";] variable list

```
10 INPUT A,A$  
20 INPUT "ENTER NUMBERS"; A,B
```

The INPUT statement prompts for data to be entered from the terminal and waits for the user to enter the data. If a prompt string followed by a semicolon (;) is included, the string is output, followed by a question mark (?) before waiting. If a prompt string followed by a comma (,) is included, the string is output and then the question mark is output on the next line before waiting for entry. If no prompt string is included, a question mark is output to the next terminal line before waiting for input.

One value must be entered for each variable in the variable list. Values may be numeric or string constants separated from each other by the current string delimiter. Strings entered do not need to be enclosed in quotes ("") unless they contain the string delimiter. If a string constant is erroneously entered in place of a numeric constant, a TYPE ERROR occurs, followed by the message REENTER FROM BEGINNING. This means that all values in the variable list should be entered again in proper order. The last value entered is delimited by a carriage return. If too few values are entered, INSUFFICIENT INPUT is output to the terminal and the statement waits for more input to satisfy the variable list. If too many values are entered, EXTRA INPUT IGNORED is output to the terminal and the program continues execution.

5.20.13 [LET] variable = expression

```
10 LET A = 5  
20 A$ = "FAT HIPPO"
```

The LET statement causes the expression to be evaluated and assigns the resulting value to the variable. The data type of the expression and the variable must be the same type or a "TYPE ERROR" results. The LET keyword is optional.

5.20.14 MEMEND numeric expression

10 MEMEND 16R7000

The MEMEND statement is used to define the upper limit of the memory space used by BASIC. One of the main applications of this statement is to reserve memory for assembly language subroutines which may be placed above the address specified by the expression.

5.20.15 NEXT numeric variable

10 NEXT X

The NEXT statement terminates the loop initiated by the FOR statement that contains the same variable. While the loop is being executed, each time control reaches the NEXT statement, the loop variable is incremented by the STEP value, or by 1 if a STEP value was not defined.

When loop execution terminates, control passes to the statement following the NEXT statement.

If a NEXT statement is encountered prior to the execution of a FOR statement naming the same loop variable, a MISSING FOR error occurs.

5.20.16 NOFLOW

500 NOFLOW

The NOFLOW statement turns off the program flow trace which may be activated by a FLOW statement.

5.20.17 ON numeric expression GOTO line number list

100 ON K+5 GOTO 200, 300, 400
200 ON J GOTO A+50, 400,B

The ON...GOTO statement causes control to be transferred to the line number whose positional value in the line number list is equal to the expression. If the expression is zero or greater than the number of lines in the list, control is passed to the next statement. If the expression is fractional, the fraction is truncated prior to the GOTO being executed. If the expression is negative a NUMBER OUT OF RANGE error occurs. The line numbers in the line number list may be numeric constants or numeric expressions. If a line number in the list does not exist a STMT # NOT FOUND error occurs.

5.20.18 ON numeric expression GOSUB line number list

```
100 ON X GOSUB 500, 600, 700, 800
200 ON Z+2 GOSUB B,C, 600
```

The ON...GOSUB statement causes execution of the subroutine beginning at the line number whose positional value in the line number list is equal to the value of the numeric expression.

If the expression is zero or greater than the number of lines in the list, control is passed to the next statement. If the expression is fractional, the fraction is truncated prior to the GOSUB being executed. If the expression is negative a NUMBER OUT OF RANGE error occurs.

The line numbers in the line number list may be numeric constants or numeric expressions. If a line number in the list does not exist a STMT # NOT FOUND error occurs.

When a RETURN statement is encountered in the subroutine, control returns to the statement following the ON...GOSUB statement.

5.20.19 OUT (numeric expression 1) = numeric expression 2

```
100 OUT (16R10) = 20
```

The OUT statement causes the value of expression 2 to be output to the I/O port specified by expression 1. Both expressions must be numeric expressions with values in the range 0 to 255 or a NUMBER OUT OF RANGE error occurs.

5.20.20 POKE (numeric expression 1) = numeric expression 2

```
100 POKE (16R6000) = 200
200 POKE (A) = B
```

The POKE statement stores the value specified by expression 2 in the memory location specified by expression 1. Expression 1 must be in the range 0 to 65535 and expression 2 must be in the range 0 to 255. If the value for either expression is outside of the specified range, a NUMBER OUT OF RANGE error occurs. Care must be exercised to ensure that the location POKE'd does not cause BASIC to crash.

5.20.21 PRINT expression {;} [TAB(numeric expression)]. . .

```
100 PRINT A;B;C  
200 PRINT TAB(10); "THE ANSWER IS"; FMT(A,"ZZZ9V.99")
```

The PRINT statement causes the value of the expressions in the expression list to be output to the terminal. Expressions are output in the formats described in section 5.16.3. "Output Formats".

An output line consists of up to 250 characters and is partitioned into 16 character print fields. Print position within an output line is controlled as follows:

- 1) An expression is output starting at the current print position. Each expression must be separated from the next expression by a comma (,) or a semicolon (;).
- 2) If the expression is followed by a semicolon, the print position is set to the next position following the last character output for the expression. If the expression is the last expression of the PRINT statement then output generated by subsequent PRINT statements will start at this position on this line of the output on the terminal.
- 3) If the expression is followed by a comma, the print position will be set to the beginning of the next 16 character print field after outputting the expression. If the expression is the last expression of the PRINT statement then output from subsequent PRINT statements will begin at this position on this line of output on the terminal.
- 4) If the last expression of the PRINT statement is not terminated by a comma or semicolon then the print position is set to the first character of the next line after outputting the value of the expression.
- 5) The print position may be explicitly set by including references to the tab function which operates only in PRINT or PUT statements. TAB moves the print position to the position specified by the value of the tab function parameter. If the position is already beyond the specified value when the print

statement is executed then the specified value is simply ignored.

BASIC contains a parameter which specifies the length of a physical output line on the terminal. If a print line which is longer than the terminal width is output, carriage returns and line feeds will automatically be inserted to wrap the output across as many physical lines as necessary.

5.20.22 READ variable list

10 READ A,B,C\$

The READ statement reads values from the BASIC programs internal data list which is created by including data statements within the program. One value is read from the data list for each variable appearing in the variable list. If there is insufficient data in the data list to satisfy the variable list then RAN OUT OF DATA will be output. If a string value is read for a numeric variable then a TYPE ERROR will occur. Values are read sequentially from the data list unless the pointer which points to the next value to be read is repositioned by use of the RESTORE statement.

5.20.23 REM remark text

10 REM THIS JUNK IS A REMARK AND IS NOT EXECUTED

The REM statement is used to include comment text. The character (!) may also be used to include comments in a program line. The REM statement and any characters following a (!) character in a program line are non-executable and are ignored.

5.20.24 RESTORE [numeric expression]

10 RESTORE

20 RESTORE 25

The RESTORE statement is used to position the data list pointer which allows control of the sequence in which data items are read from the program's internal data list. The pointer will be set to the first data item of the data statement whose line number is specified by the numeric expression. If an expression is not specified, the pointer will be set to the first item in the first data statement appearing in the program.

5.20.25 RETURN

100 RETURN

The RETURN statement transfers control to the statement immediately following the last GOSUB statement executed. If a RETURN statement is encountered prior to the execution of a GOSUB statement the error message NOTHING TO RETURN TO is output to the terminal.

5.20.26 SIZES (numeric numeric numeric [numeric
 constant 1, constant 2, constant 3, [constant 4]])

20 SIZES {5,4,80}

30 SIZES {6,5,40,3000}

The SIZES statement is used to specify the number of bytes of storage to be used for real variables (RSIZE), integer variables (ISIZE) and string variables (SSIZE), and the maximum program size when using chained program segments (see section 5.21.2.6). Constant 1 - constant 3 are positive integer constants. The value of constant 2 specifies ISIZE which must be greater than 1 and less than RSIZE. The value of constant 1 specifies RSIZE which must be greater than ISIZE and less than 40. The value of constant 3 specifies SSIZE which must be greater than 0 and less than 251.

Constant 4 is an optional parameter. If it is present it specifies the maximum number of bytes allocated for program size, after which variable space allocation begins.

If no SIZES statement is executed, the default SIZES are (5,3,40).

The SIZES statement may not be executed if any variables are already allocated. If any of the constraints described are violated, a SIZES ERROR error occurs.

5.20.27 STOP

100 STOP

The STOP statement causes the execution of a BASIC program to cease. The execution may be resumed from the line following the STOP statement with a CONT command.

5.20.28 STRING string expression

10 STRING ";"

The STRING statement defines the current string delimiter used to terminate a string accessed by an INPUT or GET statement. The end of string will be signified by either the end of the record or the first occurrence of the string delimiter. If a STRING statement has not been executed, the default delimiter is the comma (,).

5.21 BASIC DISK FILE I/O

A file is a data structure which may be accessed as a named entity and consists of a collection of data grouped into elementary units called records. The file structure is generally used for storing data on mass storage devices such as a disk. Disk Extended BASIC provides the ability to create and access files stored on the disk. Common maintenance operations such as renaming or deleting a file are included.

5.21.1 Disk Files

Each file stored on a diskette is identified by a file name, which may be from 1 to 10 characters long. The characters may be letters, digits 0-9, or the special characters period (.), slash (/), or hyphen (-).

The minimum amount of space required to store a file is one track. When a "new" file is opened, a complete track is allocated. This track and any other track assigned by the BASIC file system to this file remain unavailable to any other file until released by the user. The maximum number of files that can be stored on a disk is a function of the number of tracks available on the disk. The Mod I disk drive provides 35 tracks per diskette; Mod II provides 77 tracks per diskette. One track per diskette is required for the file directory, so the maximum number of files is either 34 or 76. Conversely, the maximum size of a file is 34 or 76 tracks. Each track consists of 16 sectors of 256 bytes per sector. A file is accessed sector by sector; therefore a "record" is 1 sector.

Actual placement of files is maintained by the BASIC file system. One track is allocated for each "new" file opened. When 16 records have been written to a particular file, another track is allocated. The file appears contiguous to the program, even if it is not stored on contiguous tracks. It is not possible to store one file on more than one disk; that is, a file may not span disks.

Files may be stored in 3 formats: Program, Object and Data.

- 1) Program Files - A program file is a BASIC program which was stored by a SAVE command as described in section 5.6. The data consists of the BASIC program text as it resided in the program buffer with keyword compression. A LOAD command will load the data from a program file into the BASIC program buffer.
- 2) Object Files - An object file is an image of a block of memory which was saved using the memory range option of the SAVE command. A LOAD command will read the data back into the memory locations from which it was saved. This is the format in which assembly language programs may be stored on the disk.

- 3) Data Files - Data files contain data created by and are accessible to BASIC programs by use of the PUT and GET statements. Each execution of a PUT statement stores 1 record in the file. Data within each record is represented as ASCII characters.

Each record is a 250 character string. A data file may not be loaded using the LOAD command. Micropolis BASIC provides the ability to access the records of a data file either sequentially or directly. (commonly referred to as random access)

In addition to the format, a file may also have Write Protect and Permanent attributes.

- 1) Write Protect - A file which is Write Protected cannot be re-written but may be deleted by a SCRATCH command. This is a software Write Protect not related to the physical Write Protect provided by a Write Protect tab installed on a diskette. If a physical Write Protect tab is installed on a diskette, all operations which attempt to modify a file or the directory will yield a WRITE PROTECT error.
- 2) Permanent - A Permanent file may be re-written but may not be deleted by a SCRATCH command.

A file may be both Permanent and Write Protected.

Several keywords are provided to manipulate disk files as described below:

5.21.2 Disk File Commands

Commands are provided to load and save program or object files, delete a file, and to display a list of the files which reside on a diskette. Although commands may appear in a BASIC program, commands will generally be executed in Immediate mode. All disk commands reference the directory of the desired diskette. If the diskette is not loaded or a malfunction exists in the disk drive which causes it to return a not ready status the message "DRIVE NOT UP" will be output to the terminal when a command is executed. If the drive is unable to read or write on the diskette properly then a PERM I/O ERROR will result.

5.21.2.1 DISPLAY string expression

```
DISPLAY "1: DIR"  
DISPLAY A$
```

The DISPLAY command will output the directory of the diskette loaded into the drive specified by the string expression. The value of the string expression must be of the form:

" [unit:] DIR" where unit is the drive

unit address in the range of 0 to 3. If omitted, drive 0 is assumed. If the string is a constant it must be enclosed in quotes (""). If a directory does not exist on the diskette a FILE NOT FOUND error results.

5.21.2.2 LOAD string expression

```
LOAD "2:DEMOPGM"
```

The LOAD command loads a program or object file into memory. The file is specified by the string expression which must evaluate to the following form:

" [unit:] filename" where unit is the

unit address in the range 0 to 3. If omitted, unit 0 is assumed. The file name may be any valid filename. If the string is a constant it must be enclosed in quotes (""). If the desired file does not reside on the diskette a FILE NOT FOUND error results. If the file is a data format file, a NOT A LOAD FILE error results.

5.21.2.3 PLOADG string expression

```
PLOADG "0:NEXTSEG"
```

The PLOADG statement operates like a combined LOAD command and RUN command. It loads the program file named in the string expression into the current program buffer and then transfers control directly to the logic of the RUN command. All variables and file status from the preceding program are reset to the initialize condition and execution begins with the first line of the new program.

The PLOADG statement may be used to cause automatic execution of several program files in sequence. This is accomplished by using a PLOADG statement as the last executed statement of each program in the sequence, such that it names, loads and begins the next program in the sequence. Note, however, that no program variables or open files are retained from one program or segment to the next.

The string expression in the PLOADG statement must evaluate to the following form:

" [unit:] filename"

where unit is the unit address in the range 0 to 3. If omitted, unit 0 is assumed. The file name may be any valid filename. If the string is a constant, it must be enclosed in quotes (""). If the desired file does not reside on the diskette a FILE NOT FOUND error results. If the file is a data format file, a NOT A LOAD FILE error results. If the file is an object file rather than a program file, it will be loaded just as if a LOAD command had been used and the current program will continue executing with the statement after the PLOADG statement.

5.21.2.4 SAVE string expression [memory address range]

```
SAVE "N:1:NEWPRG"  
SAVE "N:LOADER" 16R7000, 16R7DFF
```

The SAVE command stores program format or object format files on the diskette. The file is specified by the string expression which must evaluate to the following form:

" [N:] [unit:]filename"

If the file to be saved does not already exist on the diskette, the "N:" must prefix the unit/file name to cause the creation of a new file in the directory on the diskette. The unit is the drive unit address in the range 0-3. If omitted, unit 0 is assumed. If the string is a constant it must be enclosed in quotes ("").

The filename may be any valid filename.

If the memory range option is not included, the contents of the BASIC program buffer will be stored in the desired file in program format.

If the memory range option is specified it must be of the form:

numeric expression 1, numeric expression 2

The numeric expressions must evaluate to positive real values in the range 0 - 65535. Fractional parts will be truncated. The contents of memory from expression 1 to expression 2 will be stored in the desired file in object format.

If "N:" is not specified for a new file, a FILE NOT FOUND error results. If a file has a Write Protect attribute, it cannot be overwritten and a WRITE PROTECT error will occur if an attempt is made to save it. If a file specified as new already exists a DUPLICATE NAME error occurs.

5.21.2.5 SCRATCH string expression

SCRATCH "1:JUNKFILE"

The SCRATCH command deletes a file from the diskette directory and releases the tracks allocated to the file for use by other files. The file to be scratched is specified by the expression which must evaluate to the form:

"[unit:] filename" where the unit is

the drive unit address in the range 0 - 3. The filename may be any valid filename. If the expression is a constant it must be enclosed in quotes (""). If the unit address is omitted, unit 0 is assumed.

If the specified file does not exist, a FILE NOT FOUND error results. If the file has a permanent file attribute then it cannot be deleted and a PERM FILE error occurs.

5.21.2.6 CHAIN string expression

990 CHAIN "NEXTPART"

The CHAIN statement loads the BASIC program file specified in the string expression into the current program buffer and then transfers execution control to the first line of the newly loaded program segment. This operation is similar to the PLOADG statement with the important exception that the CHAIN statement preserves all allocated variables, user defined assembly language functions, SIZES parameters, and the current string delimiter from the last program segment. These preserved values are passed to the newly loaded program segment which may use them just as if it had assigned them. Note that open file information and user defined BASIC functions are not preserved by the CHAIN statement. If any files are open when a CHAIN is executed they are implicitly closed. This means that the filenumber is disassociated from the filename and made free for reuse; but the directory is not updated and therefore any changes in the length of the file are not recorded. In general, all open files should be properly CLOSEd before executing a CHAIN statement.

The CHAIN statement is a powerful tool which facilitates the construction of programs much larger than available system memory would otherwise permit. It makes it possible to transfer data and control from section to section of a very large program that has been divided into separately loadable segments. To use the CHAIN statement effectively certain rules must be observed.

- 1) The program size of a segment being chained in cannot be greater than the program size of the program currently in the program buffer. If this condition does occur a LOAD OVERRUN error will be reported. A procedure for avoiding this condition is to specify the size of the largest program in a chained program set as the fourth argument of a SIZES statement (see section 5.20.26). This SIZES statement should appear as the first statement of the first executed program of the chained set. The program size of each segment can be determined by LOADING it and using the PGMSIZE function (see section 5.18.1.3). Assuming a set of three program files named SEG1, SEG2, SEG3, the following example illustrates the procedure:

```
LOAD "SEG1"
READY
PRINT PGMSIZE
472
READY
LOAD "SEG2"
PRINT PGMSIZE
526
READY
LOAD "SEG3"
PRINT PGMSIZE
126
READY
```

In this example the largest PGMSIZE is 526. If SEG1 were the first file to be executed and the standard system precisions were desired, then the statement SIZES (5,3,40,526) would be included as the first statement of SEG1.

- 2) All files should be closed before executing a CHAIN statement.
- 3) A CHAIN statement should not normally be executed from within a FOR-NEXT loop. If this is done only the current value of the loop index variable will be preserved across the CHAIN.

- 4) A CHAIN statement should not normally be executed from within a subroutine. If this is done the RETURN information for that subroutine is lost across the CHAIN.
- 5) A program segment which is to be CHAINED should not normally contain a SIZES statement since SIZES statements cannot be executed after any variables have been allocated. The only exception is the case of the SIZES statement used to set the maximum program size. A special internal test allows such a statement to be chained back to as necessary.

5.21.2.7 LINK string expression

LINK "MDOS"

LINK "DISKCOPY"

The LINK command loads the overlay file specified in the string expression into memory and transfers control to the execution address of the overlay. This command is designed primarily for use with Micropolis supplied overlay files such as MDOS and DISKCOPY. These files completely replace BASIC in memory when LINKed to. They take over the control of the computer system and provide their own operating commands and dialogue.

The string expression must evaluate to a valid filename. The file must be an overlay type C through F. If the specified file is not found or the disk unit is not ready, control will return to BASIC where the error will be reported. If an unrecoverable disk error occurs during the LINKing process, the system will execute a soft halt. This is done because BASIC has already been partially destroyed and the new system has not been successfully loaded. The computer must be reset and a new system booted in.

The LINK command can be used to load and transfer control to a machine language program file that runs in high memory above the end of BASIC (see MEMEND statement). It can return to the BASIC interpreter by jumping to the system warmstart address.

5.21.3 DISK I/O STATEMENTS

BASIC statements are provided which allow a BASIC program to create and transfer data to and from data format files, and to perform certain file maintenance functions on any type file such as renaming a file or changing the attributes of a file. The operation of disk I/O statements differs from the disk commands as follows:

- 1) Disk I/O statements refer to files through a program "File Number". An OPEN statement must be executed to associate a file on the diskette with a program file number.
- 2) When all I/O operations on a file are complete, a file must be closed by executing a CLOSE statement. Closing a file consists of updating the directory to reflect all operations which have been performed since the file was opened, and disassociating the file from the program file number. **CAUTION:** A file which has been written to must ALWAYS be closed or data written to the file may be lost.

Prior to any operation which accesses the disk, BASIC ensures that the drive is ready to accept commands. If the diskette is not loaded or a malfunction exists which prevents the drive from performing operations then a DRIVE NOT UP error results. If the disk is unable to perform the specified read/write operation properly, a PERM I/O ERROR results.

A program file number may be in the range 0 to 9. As many as 10 files may be open at once within a program. If an I/O statement attempts to access a file which has not been opened by an OPEN statement then a FILE NOT OPEN error results.

If an I/O statement specifies a file number outside the range 0 to 9 then a NOT A FILE# error occurs.

5.21.3.1 OPEN file number string expression options

```
10 OPEN 1 "N: NEWFILE"
20 OPEN 2 "JOE" END 1000 ERROR 5000
```

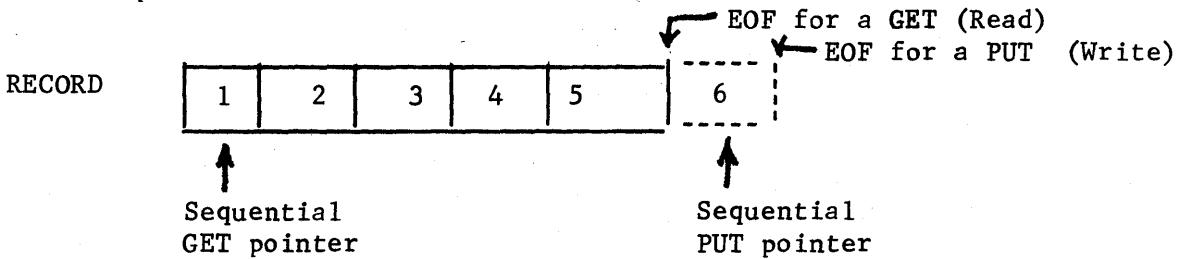
The OPEN statement opens the specified file for access by disk I/O statements. The file is selected by the string expression which must evaluate to the form:

"[N:] [unit:] filename"

If the file to be opened does not exist on the diskette, the characters "N:" must be included in the unit/filename to cause the creation of a new file in the directory. The file created is a data format file. The unit specifies the drive unit address which must be in the range 0-9. The filename may be any valid filename. If the string is a constant, it must be enclosed in quotes (""). If the unit address is omitted, unit 0 is assumed. If the specified file does not exist and is not declared as a new file, a FILE NOT FOUND error occurs. If a file specified as new already exists, a DUPLICATE NAME error occurs.

The filenumber must be a numeric expression with a value of 0 - 9. The filename specified will be associated with this file number until the file is closed and all file I/O directed to the file number will be performed using this file.

Each open file has two associated pointers which point to the next record to be accessed in a sequential PUT or GET statement. When a file is opened, the sequential GET pointer is initialized to point to the first record. The sequential PUT pointer is initialized to point to the record following the last record. The last record in the file is considered the end of the file for GET statements. The last record +1 is considered the end of file for PUT statements. For example a 5 record file would have pointers initialized as follows:



An open file may be read from and written to both sequentially and directly by record.

The open statement includes several options which are listed below:

- 1) CLEAR - The CLEAR option overrides the normal initialization of the sequential GET & PUT pointers. The pointers are initialized so that the file is empty. A subsequent GET will encounter an end-of-file. A PUT will write into record 1. This option is generally used to initialize the pointers for re-writing a file sequentially.
- 2) END numeric expression

The END option specifies the line number to GOTO when the end-of-file is encountered during a read operation. The numeric expression must evaluate to a positive real number which is a valid program line within the program when the fractional part, if any, is truncated. If the line does not exist, a STMT # NOT FOUND error occurs. This option allows the BASIC program to handle an end-file condition without the program being aborted. If the END option is not specified, the normal end-file handling is to abort the program with an 'END-FILE' error.

3) ERROR numeric expression

The ERROR option specifies the line number to GOTO if a disk I/O error occurs. The numeric expression must evaluate to a positive real number which is a valid program line within the program when the fractional part, if any, is truncated. If the line does not exist, a STMT # NOT FOUND error occurs. This option allows a BASIC program to handle disk I/O errors without being aborted. If the error option is not included, a disk I/O error will cause the appropriate error message to be output and abort the program. the ERR function may be used in the error handling program section to determine the type of error.

5.21.3.2 PUT filenumber RECORD record number expression List

```
100  PUT 1 A;B;C  
200  PUT 1 A;A$+";"; B  
300  PUT 1 RECORD 3 A;B;C
```

The PUT statement causes the values of the expressions in the expression list to be written onto a record of the file specified by the filenumber expression. The filenumber must be a numeric expression having a value of the digits 0 - 9 when the fractional part, if any, is truncated.

Each execution of a PUT statement writes one record into the file.

Each disk record is composed of a 250 character string and is, in fact, a print line. Each expression in the expression list is evaluated, converted to a string if the resulting value is numeric, and is placed in the string in exactly the same way that print lines are built. The rules for building the string are as follows:

- 1) The record string is partitioned into 16 character fields. A pointer which is initialized to point to the first character in the string keeps track of the next position in the string to be loaded.
- 2) Expressions are evaluated as they are encountered in scanning the expression list and from left to right, and are converted to strings according to the formats described in section 5.16.3 "Output Formats". The resulting string is loaded into the record string beginning at the pointer position. Each expression must be separated from the next expression by a comma(,) or a semicolon(;).

- 3) If the expression is followed by a comma(,) after the expression has been loaded into the string, the string is padded with enough blanks to position the pointer to the beginning of the next 16 character field.
- 4) If the expression is followed by a semicolon(;), after the expression has been loaded into the string the pointer is set to the character position following the last character of the expression.
- 5) After all expressions have been loaded into the record string, any remaining characters in the string are padded with blanks and the record string is written onto the diskette.

EXAMPLE: If A = 100 and B = -2.5, the statement:

100 PUT 1 A;B

would cause the following record to be written on the disk: (Note: \emptyset denotes a blank)

" \emptyset 100 \emptyset - 2.5 $\emptyset\emptyset$... \emptyset "
 A B 240 Character pad

The Statement

100 PUT 1 A,B

would cause the following record to be written to the disk:

" \emptyset 100 \emptyset $\emptyset\emptyset\emptyset\emptyset\emptyset\emptyset\emptyset\emptyset$ - 2.5 $\emptyset\emptyset$... \emptyset "
 A PAD B 229 Character pad

The expressions in the expression list may be numeric and string in any order subject to the following restrictions: (1) If a string expression follows a numeric expression it must be immediately preceded by the current string delimiter. (2) The last character of a string expression must be the current string delimiter. These restrictions Must Be Strictly Followed or the expression will not be properly read back.

On Input, numeric values are delimited by blanks. The output format of numeric values always follows the value with a blank, so numeric strings built as described will always read back correctly. Strings, however, may contain embedded blanks. The input logic which reads a record from the disk looks for the current string delimiter to denote the end of a string. If a string follows a numeric value, the blank following the numeric field will be included in the string unless the current string delimiter precedes the string.

One solution to this problem is to concatenate the string delimiter on all string variable references, include the string delimiter in all string constants, and precede all string expressions following numeric expressions with the string delimiter.

EXAMPLE:

To write the values of A,B\$,C, E\$ and F\$ on the diskette, the PUT statement would be

```
100 PUT 1 A;"+"B$+";"C;"+"E$+";"F$+";"  
(This example uses the default delimiter, comma (,))
```

If it is desired to change the string delimiter, the following approach could be used to implement the previous example:

```
10 D$ = ";" ! SET STRING DELIMITER  
20 STRING D$
```

•
•
•

```
100 PUT 1 A;D$+B$+D$;C;D$+E$+D$;F$+D$
```

If this approach is used, the string delimiter must be the same when a record is read as when it was written or incorrect results will be obtained.

If the record option is not included, the record is written into the file at the record number specified by the sequential PUT pointer. The pointer is then incremented by 1.

If the record number option is included, the record is written into the record specified by the record number expression. The record number expression must have a value which is a positive real number. The fractional part is truncated. If the record number is greater than the end-of-file as described in 5.21.3.1, a PARM ERROR occurs.

NOTE; Writing a record directly by use of the RECORD option does not affect the sequential put pointer. The pointer will only be moved by a sequential PUT or execution of a PUTSEEK statement.

If an attempt is made to write more than 250 characters into a record, the message OUTPUT OVERFLOW will be output to the terminal and nothing will be written.

5.21.3.3 GET filenumber RECORD record number variable list

```
100 GET 1 A,B,C$  
200 GET 1 RECORD 100 A,B C$
```

The GET statement reads a record from the file specified by the filenumber expression and assigns the values read to the variable list. The filenumber expression must evaluate to one of the digits 0 - 9. The fractional part, if any, is truncated.

If a string is read for numeric variable, a 'TYPE ERROR' results. If too few values exist in the record string to satisfy the variable list, a 'RAN OUT OF DATA' error occurs. If an attempt is made to get a record which is past the last record, an 'END FILE' error occurs.

If the RECORD option is not included, the record read is the record specified by the sequential GET pointer. The sequential GET pointer will then be incremented by 1.

If the RECORD option is included, the record read is the record specified by the recordnumber expression. The expression must evaluate to a positive real number. The fractional part will be truncated.

NOTE: The sequential GET pointer is not affected by a direct GET. The pointer will only be modified by a sequential GET or by execution of a GETSEEK statement.

5.21.3.4 CLOSE filenumber

```
100 CLOSE 1
```

The CLOSE statement causes the file specified by the filenumber expression to be closed for disk I/O. The filenumber expression must evaluate to one of the digits 0 - 9 when the fractional part is truncated.

Closing a file consists of updating the file entry in the diskette directory to reflect all operations which were performed upon the file since it was opened, and disassociating the file from the program filenumber. As a rule, all files which are opened in a program should be closed before the program terminates. All files which have been written into must be closed or the directory will not be updated and data written into the file may be lost. Any files which are left open are implicitly closed by a RUN command or any command that modifies the program buffer, such as a DELETE,

LOAD or line insertion/deletion. Implicit closure does not update the directory.

5.21.3.5 ATTRS (filenumber) = numeric expression

100 ATTRS (2) = 19

The ATTRS statement sets the file attributes of the file referenced by the filenumber to the value of the numeric expression. The file-number expression must evaluate to one of the digits 0-9 when the fractional part is truncated. The numeric expression, when the fractional part is truncated, must evaluate to a valid combination of the attribute values which are described below:

<u>VALUE</u>	<u>ATTRIBUTE</u>
16	Program File
8	Object File
2	Permanent File
1	Write Protect

A file which does not have a Program or Object attribute is assumed to be a Data Format file. Some examples are:

19 = 16+2+1 = Write protected, permanent, program file
9 = 8+1 = Write protected, object file
26 = 16+8+2 = Invalid combination - This would identify a file as being a Permanent Program file and Object file, which is not possible.

A main intent of the ATTRS statement is to allow the user to change the Write Protect and Permanent attributes only. The File Format attributes should not be changed. The current value of the attribute parameter may be accessed by the ATTR function.

5.21.3.6 EOF (filenumber) = expression

150 EOF (9) = 50

The EOF statement sets the file length parameter of the file referenced by the file number to the value of the expression. The filenumber expression must evaluate to one of the digits 0 - 9 when the fractional part is truncated. The expression must evaluate to a positive real number. The fractional part will be truncated. The EOF statement is used to decrease the length of a file. The value of the expression should be set to 1 greater than the last record number. For example if a file contains 100 records and it is desired to delete the last 50 records, the statement

100 EOF (1) = 51

would cause record 50 to be the last accessible record. The following cautions apply to the use of EOF statement:

- 1) The EOF statement does not reset the sequential PUT/GET pointers. If they are set beyond the new EOF an END-FILE error will occur if a PUT or GET is attempted. Reset the pointers to the proper values with the GETSEEK and PUTSEEK statements.
- 2) Do Not Set The EOF Beyond the true length of the file. Any sectors remaining on the last allocated track may be read by a GET and will yield garbage.
- 3) Resetting the EOF does not release the now unused tracks for system use. De-allocate the unused tracks by executing a FREESPACE statement.

5.21.3.7 FREESPACE filenumber

100 FREESPACE 1

The FREESPACE statement de-allocates any tracks allocated to the file referenced by filenumber which are beyond the current end of file. Filenumber expression must evaluate to one of the digits 0 - 9 when the fractional part is truncated. If there are no excess tracks allocated an "END FILE" error results.

5.21.3.8 GETSEEK (filenumber) = numeric expression

50 GETSEEK (1) = 20

The GETSEEK statement sets the sequential GET pointer associated with the filenumber to the value of the numeric expression. The filenumber expression must evaluate to one of the digits 0 - 9 when the fractional part is truncated. The numeric expression must evaluate to a positive real number. The fractional part is truncated. The value must be greater than zero and less than or equal to the last record number or a PARM ERROR or END FILE error will occur when a sequential GET is performed. The current position of the pointer may be accessed by using the RECGET function.

5.21.3.9 PUTSEEK (filenumber) = numeric expression

100 PUTSEEK (2) = 30

The PUTSEEK statement sets the sequential PUT pointer associated with the filenumber to the value of the numeric expression. The filenumber expression must evaluate to one of the digits 0 - 9 when the fractional part is truncated. The numeric expression must

evaluate to a positive real number. The fractional part is truncated. The value must be greater than zero and less than the last record number +2 or a PARM ERROR will occur when a sequential PUT is performed. The current value of the pointer may be accessed by using the RECPUT function.

5.21.3.1Ø RENAME (filenumber) = string expression

1ØØ RENAME (1) = "NEWNAME"

The RENAME statement changes the name of the file referenced by the filenumber to the value of the string expression. The file-number expression must evaluate to one of the digits Ø - 9 when the fractional part is truncated. The string expression must evaluate to a valid file name. The current name can be accessed using the NAME function.

5.21.4 DISK I/O FUNCTIONS

Disk File I/O functions are included within BASIC to provide information about a currently open file. Each function reference includes a file number expression which must evaluate to one of the digits Ø - 9 when the fractional part is truncated. If the specified file number does not have a file currently opened to it a FILE NOT OPEN error occurs. The disk file I/O functions are detailed in table 5.5.

TABLE 5.5 DISK I/O FUNCTIONS

<u>Function Reference</u>	<u>VALUE</u>
ATTR (n)	Returns the attribute parameter associated with file n. See section 5.21.3.5 for a description of the value.
ERR	<p>Returns the error code associated with the last disk error. The error codes are:</p> <ul style="list-style-type: none"> Ø - No Error 1 - Permanent I/O Error 2 - End-File 3 - Disk Full 4 - File Not Found 5 - Duplicate Name 6 - Parameter Error 7 - Drive Not Up 8 - Permanent File 9 - Write Protect <p>12 - Printer Attention</p> <p>The error code is not reset by a successful operation, so is meaningless unless an error occurs.</p>
ERR\$	Returns the error message string associated with the last disk error.
NAME (n)	Returns a string containing the name of the file associated with file number n.
RECGET (n)	Returns the value of the sequential GET pointer associated with file number n.
RECPUT (n)	Returns the value of the sequential PUT pointer associated with file number n.
SIZE (n)	Returns the SIZE (in records) of the file associated with file number n.
TRACKS (n)	Returns the number of disk tracks currently allocated to file number n.
FREETR (n)	Returns the number of disk tracks currently available for allocation (free) on the disk unit associated with file number n.

5.22 BASIC PRINT FILE OUTPUT

Micropolis BASIC provides a set of print file output features for systems which have a hard copy printer device in addition to the standard keyboard-display terminal. This section specifies each of the printer related language features and discusses how to use the available features to solve some common printer programming problems.

5.22.1 Printer Related Language Features

The printer related language features consist of seven statement and option keywords. They achieve a high flexibility of output control by expanding the disk file I/O scheme to include print file and terminal file output and by adding a physical device assignment capability. Following are descriptions of each statement syntax and function.

5.22.1.1 OPEN filenumber string expression option(s)

```
10 OPEN 1 "*P" PAGESIZE 66 ENDPAGE 900  
20 OPEN 2 "*T"  
30 OPEN 7 "*N"
```

The syntax of the OPEN statement in this context is the same as that for disk files as shown in section 5.21.3.1. The statement associates a filenumber with a filename specified in the string expression. The filenumber must be a numeric expression with a value of 0 - 9. The string expression which contains the filename must have one of three specific values which designate a particular output print device.

- 1) Filename *P associates the filenumber being opened with the system printer.
- 2) Filename *T associates the filenumber being opened with the display element of the system terminal.
- 3) Filename *N associates the filenumber being opened with a null output device. The output directed to that file will be discarded or drained.

Any other filename will be interpreted as a disk file name per section 5.21.3.1.

There are two print file options available with the OPEN statement:

- a) PAGESIZE numeric expression

This option allows the programmer to set a limit value for an internal system counter which counts the number of lines output to the associated filenumber. The counter is incremented on each PUT statement to the associated file, unless that PUT statement ends in a comma or semicolon (see section 5.22.1.2). Each time the limit count is reached, the

counter is reset and the system checks for a corresponding ENDPAGE option.

The numeric expression must evaluate to a whole number from 0 - 65535. If a print file is opened without a PAGESIZE option the internal limit value defaults to a value of 66 which is the number of lines per page on standard 11 inch forms.

b) ENDPAGE linenumber

This option specifies a program line number to which the system will perform a GOSUB each time that the limit is reached on the internal lines per page counter. The line-number must be a numeric expression which evaluates to a legal linenumber. That line should be the beginning of a subroutine which programs some appropriate end of page actions and which ends with a RETURN statement. The RETURN will go back to the statement immediately after the PUT statement which triggered the end of page action.

If no ENDPAGE option is specified for a given file the internal lines per page counter is just reset each time the limit is reached and processing continues normally.

5.22.1.2 PUT filenumber expression list

```
15 PUT 0 "TOTAL = "; A1, "ITEM NAME ="; B$  
25 PUT 7 A, B;
```

The PUT statement causes the values of the expressions in the expression list to be assembled into an output record which is then output to the print file device associated with the filenumber. The filenumber must be a numeric expression with a value in the range 0 - 9. The expression list consists of a sequence of constants and/or variables separated by commas or semicolons. The rules by which the output record is assembled are the same as those for PRINT statements as detailed in section 5.20.21. Separate carriage width wraparound control is provided for the printer device. If the expression list ends with a comma or semicolon then no carriage return line feed is output. In this case the internal lines per page counter of the associated file is not incremented. (see section 5.22.1.1 - PAGESIZE option). The TAB and FMT functions may be used in PUT statements.

5.22.1.3 CLOSE filenumber

```
90 CLOSE 6  
99 CLOSE 2
```

The CLOSE statement causes the file specified by the filenumber expression to be closed for output. The filenumber must be in the range 0 - 9. When a print file is closed the associated filenumber is freed for use in a subsequent OPEN to another file.

Any files which are left open are implicitly closed by a RUN command or by any command that modifies the program buffer, such as DELETE, LOAD or line insertion change.

5.22.1.4 ENDPAGE filenumber

25 ENDPAGE 7

28 ENDPAGE R6

The ENDPAGE statement is related to the ENDPAGE option described in section 5.22.1.1. However, it is syntactically and functionally distinct. Its function is to end the current output page of the designated filenumber and thereby position the output device to the beginning of the next logical page. The filenumber must be a numeric expression with a value in the range 0 - 9. When the ENDPAGE statement is executed the current value of the lines per page counter associated with filenumber is subtracted from its limit value. The result determines the number of empty lines which are output to the file device to complete the current logical page. When the ENDPAGE statement is complete the associated lines per page counter is reset to mark the beginning of the next logical page.

5.22.1.5 ASSIGN (physical device number, logical stream indicator, device width, null count)

10 ASSIGN (2,1,80,6)

20 ASSIGN (2,2,132)

30 ASSIGN (1,1)

The ASSIGN statement is a dual purpose statement which provides the ability to specify the connections of physical output print devices to logical output streams and the values for carriage width and nullcount of the referenced physical device. The physical device number must be a numeric expression which evaluates to a 1 or a 2. The logical stream indicator must be a numeric expression which evaluates to a 1, 2 or 3. The device width and nullcount must be numeric expressions with values in the range 1 - 255. They are optional parameters in the ASSIGN statement. If they are not included, the values corresponding to the referenced physical device are not changed. If only the device width is included, then the nullcount is left unchanged. Note however that specifying a nullcount requires that a device width also be specified, i.e., if the statement only contains three arguments, the third will always be treated as a device width.

Logical output stream number 1 consists of all output generated by system messages, keyboard echoing, PRINT statements, LIST commands, and PUT statements when the corresponding filenumber is open to *T. Logical output stream 2 consists of all output generated by LISTP commands and by PUT statements when the corresponding filenumber is open to *P. The logical stream indicator may be set to a value of 3 to represent both logical output streams 1 and 2.

Physical device number 1 represents the display element of the keyboard display device that is configured as the system terminal. (see section 3.3.1 on terminal configuration). Physical device number 2 represents the hard copy print device which is configured as the system printer. (see section 3.3.4).

The output of a logical stream is directed to all physical devices which are assigned to it. A physical device may be assigned to one or both logical streams. Whenever a physical device is ASSIGNED its previous assignment state is effectively cancelled. A list of legal device connections follows:

- ASSIGN (1,1) - connects terminal display to stream 1 only
- ASSIGN (1,2) - connects terminal display to stream 2 only
- ASSIGN (1,3) - connects terminal display to stream 1 and stream 2
- ASSIGN (2,1) - connects printer to stream 1 only
- ASSIGN (2,2) - connects printer to stream 2 only
- ASSIGN (2,3) - connects printer to stream 1 and stream 2

In its initialized state BASIC connects the terminal to stream 1 only and the printer to stream 2 only. This state can be restored by executing an ASSIGN (1,1) followed by an ASSIGN (2,2).

When the terminal and printer devices are configured each device has a carriage width and a nullcount parameter associated with it. These parameters may be altered under program control by specifying optional 3rd and 4th arguments in an appropriate ASSIGN statement. The width parameter determines the maximum number of spaces on each line for the given device. When a line is output that is longer than width the autowrap feature is activated and a carriage return line feed is inserted between character number width and width +1. The autowrap feature may be disabled at configuration time. The width parameter may be changed on a given device by restating the current device assignment with a new width argument. For example, if the terminal were currently assigned to stream 1 with a width of 80, it could be changed to a width of 72 with the statement ASSIGN (1,1,72). Note that any such change remains in effect until a subsequent ASSIGN statement alters it or until the system is re-loaded. The nullcount parameter is one greater than the number of nulls which are output after each carriage return output to a given device. It is important with unbuffered character serial devices which may lose characters while the carriage is being returned. The nullcount parameter for a given device may be dynamically changed by restating the current device assignment and WIDTH with a new nullcount. For example, if the printer were currently assigned to stream 2, 132 columns, no nulls (nullcount = 1), it could be changed to stream 2, 132 columns, 5 nulls by using the statement ASSIGN (2,2,132,6).

Because BASIC is an interactive language it depends on the availability of a display device for system messages and keyboard echoing. An interlock is therefore built in to ensure that stream 1 always has at least one device assigned to it. If an ASSIGN statement is processed the result of which would violate this condition, then physical device 1 is automatically assigned to stream 1 as part of the ASSIGN being processed.

5.22.1.6 LISTP X - Y

LISTP
LISTP 10
LISTP -10
LISTP 10-
LISTP 10-100

The LISTP command causes a listing of the program in the current program buffer to be directed to logical output stream 2 which is normally connected with the system printer. This COMMAND is analogous to the LIST command (see section 5.5) with two exceptions. The LIST command directs its output to logical stream 1 which is normally connected to the system terminal display. The LISTP command outputs a paginated listing with three blank lines at the top and bottom of each page and 60 lines of listing as standard. (see 5.22.1.7).

X and Y must be legal linenumber constants.

LISTP prints the entire program buffer.

LISTP X prints only line X if present or the first line greater than X if no line X exists.

LISTP X- prints all lines starting with X or the first greater than X through the end of the program buffer.

LISTP -Y prints from the beginning of program buffer thru line Y or the first greater than Y.

LISTP X-Y prints from line X or first greater than X through line Y or first greater than Y.

5.22.1.7 PAGESIZE numeric expression

PAGESIZE 42

The PAGESIZE command is related to the LISTP command. It causes the number of lines of listing per page of the LISTP command to be set to the value of the numeric expression in the PAGESIZE statement. This number is the number of actually printed lines not including the 3 blank lines at the top and bottom of each page. For example, to list a program on paper which holds 48 lines per page, the statement PAGESIZE 42 would be the proper value to use. When BASIC is configured the default value for this parameter is 60.

NOTE that the PAGESIZE statement as described here is syntactically and functionally distinct from the PAGESIZE option of the OPEN statement as described in 5.22.1.1

5.22.2 Notes On Printer Related Programming

Used properly and with care the printer related language features in Micropolis BASIC provide for highly flexible and efficient programming of many common print file related functions. This section provides some examples and commentary.

5.22.2.1 Separating Print Files and Interactive Messages

There is a large variety of applications which can be programmed in the following three part structure:

- 1) Output to the terminal display a sequence of prompting messages which lead the user through a process of entering variable data from the terminal keyboard.
- 2) Process the input data through algorithms which create desired output data.
- 3) Output to the printer one or more pages which present the desired output data with proper labelling in an appropriate report format.

This structure requires the ability to separate output which is normally intended for the operators terminal from output which is normally intended for the system printer. In Micropolis BASIC the separation may be accomplished by using PRINT statements for terminal display messages and PUT statements to open print files for system printer output. The technique is illustrated by the following program for building a depreciation schedule chart.

```

100 !    *** DATA INPUT SECTION
110 !
120 PRINT "THIS PROGRAM WILL BUILD A DEPRECIATION SCHEDULE"
130 PRINT "SHOWING YEAR BY YEAR DEPRECIATION OF A FIXED ASSET"
140 PRINT "AT STRAIGHT LINE AND 200% ACCELERATED RATES."
150 PRINT
160 PRINT "PLEASE ENTER ASSET VALUE ";
170 INPUT A
180 PRINT "PLEASE ENTER TERM IN YEARS";
190 INPUT T
200 PRINT "PLEASE ENTER FIRST YEAR OF TERM (EG. 1977)";
210 INPUT Y
220 !
230 !    *** PRINT OUT CHART HEADINGS
240 !
250 OPEN 9 "♦P"
260 PUT 9:PUT 9
270 PUT 9 "DEPRECIATION SCHEDULE FOR $ ";A;" OVER ";T;" YEAR(S)"
280 PUT 9:PUT 9
290 PUT 9 "YEAR", "ST. LN. DEP.", "BALANCE", "200% DEP.", "BALANCE"
300 PUT 9
310 !
320 !    *** COMPUTE AND PRINT EACH LINE
330 !
340 B1=A:B2=A:S=A/T:F$="$ZZZZZZV.99"
350 FOR K=1TOT
360 B1=B1-S
370 D=2♦B2/T
380 B2=B2-D
390 PUT 9 Y,FMT(S,F$),FMT(B1,F$),FMT(D,F$),FMT(B2,F$)
400 Y=Y+1
410 NEXT K
420 CLOSE 9
430 END

```

RUN

THIS PROGRAM WILL BUILD A DEPRECIATION SCHEDULE
SHOWING YEAR BY YEAR DEPRECIATION OF A FIXED ASSET
AT STRAIGHT LINE AND 200% ACCELERATED RATES.

PLEASE ENTER ASSET VALUE ? 100000

PLEASE ENTER TERM IN YEARS? 25

PLEASE ENTER FIRST YEAR OF TERM (EG. 1977) ? 1980

DEPRECIATION SCHEDULE FOR \$ 100000 OVER 25 YEARS (\$)

YEAR	ST. LN. DEP.	BALANCE	200% DEP.	BALANCE
1980	\$ 4000.00	\$ 96000.00	\$ 8000.00	\$ 92000
1981	\$ 4000.00	\$ 92000.00	\$ 7360.00	\$ 84640
1982	\$ 4000.00	\$ 88000.00	\$ 6771.20	\$ 77868
1983	\$ 4000.00	\$ 84000.00	\$ 6229.50	\$ 71639
1984	\$ 4000.00	\$ 80000.00	\$ 5731.14	\$ 65908
1985	\$ 4000.00	\$ 76000.00	\$ 5272.65	\$ 60635
1986	\$ 4000.00	\$ 72000.00	\$ 4850.84	\$ 55784
1987	\$ 4000.00	\$ 68000.00	\$ 4462.77	\$ 51321
1988	\$ 4000.00	\$ 64000.00	\$ 4105.75	\$ 47216
1989	\$ 4000.00	\$ 60000.00	\$ 3777.29	\$ 43438
1990	\$ 4000.00	\$ 56000.00	\$ 3475.10	\$ 39963
1991	\$ 4000.00	\$ 52000.00	\$ 3197.09	\$ 36766
1992	\$ 4000.00	\$ 48000.00	\$ 2941.33	\$ 33825
1993	\$ 4000.00	\$ 44000.00	\$ 2706.02	\$ 31119
1994	\$ 4000.00	\$ 40000.00	\$ 2489.54	\$ 28629
1995	\$ 4000.00	\$ 36000.00	\$ 2290.37	\$ 26339
1996	\$ 4000.00	\$ 32000.00	\$ 2107.14	\$ 24232
1997	\$ 4000.00	\$ 28000.00	\$ 1938.57	\$ 22293
1998	\$ 4000.00	\$ 24000.00	\$ 1783.49	\$ 20510
1999	\$ 4000.00	\$ 20000.00	\$ 1640.81	\$ 18869
2000	\$ 4000.00	\$ 16000.00	\$ 1509.54	\$ 17359
2001	\$ 4000.00	\$ 12000.00	\$ 1388.78	\$ 15971
2002	\$ 4000.00	\$ 8000.00	\$ 1277.68	\$ 14693
2003	\$ 4000.00	\$ 4000.00	\$ 1175.46	\$ 13517
2004	\$ 4000.00	\$.00	\$ 1081.42	\$ 12436

READY

5.22.2.2 Paginating Print Files

When the number of lines in a print file spans several printed pages it is often required to print the file with page numbers, headings and an equal number of lines on each page. The ENDPAGE statement and the PAGESIZE and ENDPAGE options of the OPEN statement provide a useful set of tools for accomplishing this goal. The following example shows the depreciation schedule program of section 5.22.2.1 modified to print on 20 line pages with each page numbered and titled. Note the use of the PAGESIZE and ENDPAGE options in line 320 in conjunction with the page heading subroutine at line 600. NOTE also the use of the ENDPAGE statement in line 510 which ejects the last report page and leaves the printer at the top of the next blank page.

```

100 !    *** DATA INPUT SECTION
110 !
120 PRINT "THIS PROGRAM WILL BUILD A DEPRECIATION SCHEDULE"
130 PRINT "SHOWING YEAR BY YEAR DEPRECIATION OF A FIXED ASSET"
140 PRINT "AT STRAIGHT LINE AND 200% ACCELERATED RATES."
150 PRINT
160 PRINT "PLEASE ENTER ASSET VALUE ";
170 INPUT A
180 PRINT "PLEASE ENTER TERM IN YEARS";
190 INPUT T
200 PRINT "PLEASE ENTER FIRST YEAR OF TERM (EG. 1977)";
210 INPUT Y
300 !
305 !    *** OUTPUT INITIALIZATION
310 !
320 OPEN 9 "♦P" PAGESIZE 20 ENDPAGE 600
330 P=1:GOSUB 600
340 B1=A:B2=A:S=A/T:F$="$ZZZZZZV.99"
400 !
410 !    *** COMPUTE AND PRINT EACH LINE
420 !
440 FOR K=1TOT
450 B1=B1-S
460 D=2♦B2/T
470 B2=B2-D
480 PUT 9 Y,FMT(S,F$),FMT(B1,F$),FMT(D,F$),FMT(B2,F$)
490 Y=Y+1
500 NEXT K
510 ENDPAGE 9:CLOSE 9
520 STOP
600 !
610 !    *** PAGE HEADING SUBROUTINE
620 !
630 PUT 9
640 PUT 9 TAB(72);"PAGE ";P
650 PUT 9
660 PUT 9 "DEPRECIATION SCHEDULE FOR $ ";A;" OVER ";T;" YEAR(S)"
670 PUT 9:PUT 9
675 PUT 9 "YEAR","ST. LN. DEP.", "BALANCE", "200% DEP.", "BALANCE"
677 PUT 9
700 P=P+1
710 RETURN
999 END

```

READY

RUN

THIS PROGRAM WILL BUILD A DEPRECIATION SCHEDULE
SHOWING YEAR BY YEAR DEPRECIATION OF A FIXED ASSET
AT STRAIGHT LINE AND 200% ACCELERATED RATES.

PLEASE ENTER ASSET VALUE ? 100000

PLEASE ENTER TERM IN YEARS? 25

PLEASE ENTER FIRST YEAR OF TERM (EG. 1977)? 1980

PAGE

DEPRECIATION SCHEDULE FOR \$ 100000 OVER 25 YEAR(S)

YEAR	ST. LN. DEP.	BALANCE	200% DEP.	BALANCE
1980	\$ 4000.00	\$ 96000.00	\$ 8000.00	\$ 92000.00
1981	\$ 4000.00	\$ 92000.00	\$ 7360.00	\$ 84640.00
1982	\$ 4000.00	\$ 88000.00	\$ 6771.20	\$ 77868.80
1983	\$ 4000.00	\$ 84000.00	\$ 6229.50	\$ 71639.29
1984	\$ 4000.00	\$ 80000.00	\$ 5731.14	\$ 65908.15
1985	\$ 4000.00	\$ 76000.00	\$ 5272.65	\$ 60635.50
1986	\$ 4000.00	\$ 72000.00	\$ 4850.84	\$ 55784.66
1987	\$ 4000.00	\$ 68000.00	\$ 4462.77	\$ 51321.88
1988	\$ 4000.00	\$ 64000.00	\$ 4105.75	\$ 47216.13
1989	\$ 4000.00	\$ 60000.00	\$ 3777.29	\$ 43438.84
1990	\$ 4000.00	\$ 56000.00	\$ 3475.10	\$ 39963.73
1991	\$ 4000.00	\$ 52000.00	\$ 3197.09	\$ 36766.63

PAGE

DEPRECIATION SCHEDULE FOR \$ 100000 OVER 25 YEAR(S)

YEAR	ST. LN. DEP.	BALANCE	200% DEP.	BALANCE
1992	\$ 4000.00	\$ 48000.00	\$ 2941.33	\$ 38825.30
1993	\$ 4000.00	\$ 44000.00	\$ 2706.02	\$ 31119.28
1994	\$ 4000.00	\$ 40000.00	\$ 2489.54	\$ 28629.73
1995	\$ 4000.00	\$ 36000.00	\$ 2290.37	\$ 26339.36
1996	\$ 4000.00	\$ 32000.00	\$ 2107.14	\$ 24232.21
1997	\$ 4000.00	\$ 28000.00	\$ 1938.57	\$ 22293.63
1998	\$ 4000.00	\$ 24000.00	\$ 1783.49	\$ 20510.14
1999	\$ 4000.00	\$ 20000.00	\$ 1640.81	\$ 18869.33
2000	\$ 4000.00	\$ 16000.00	\$ 1509.54	\$ 17359.78
2001	\$ 4000.00	\$ 12000.00	\$ 1388.78	\$ 15971.00
2002	\$ 4000.00	\$ 8000.00	\$ 1277.68	\$ 14693.32
2003	\$ 4000.00	\$ 4000.00	\$ 1175.46	\$ 13517.85

PAGE

DEPRECIATION SCHEDULE FOR \$ 100000 OVER 25 YEAR(S)

YEAR	ST. LN. DEP.	BALANCE	200% DEP.	BALANCE
2004	\$ 4000.00	\$.00	\$ 1081.42	\$ 12436.42

5.22.2.3 Spooling Print Files To Disk For Later Output

The commonality of the OPEN, CLOSE and PUT statements to both disk and print files makes it possible to alter a print file program so that the output is saved in a disk file instead of sent to the printer. The procedure is to change the filename in the relevant OPEN statement from "*P" to some appropriate disk filename. For example, line 320 in the depreciation program listing might be changed to

```
320 OPEN 9 "N:DEP-REPORT" PAGESIZE 20 ENDPAGE 600
```

A print file that has been spooled to disk in this manner can be printed out at a later time by using the following program:

```
5 INPUT "ENTER PAGE WIDTH OF FILE TO BE PRINTED";A$  
10 DIM A$(A)  
20 STRING CHAR$(16RFFF)  
30 INPUT "ENTER NAME OF FILE TO BE PRINTED";A$  
40 OPEN 1 A$ END 90  
50 OPEN 2 "*P"  
60 GET 1 A$  
70 PUT 2 A$  
80 GOTO 60  
90 CLOSE 1  
100 CLOSE 2  
110 END
```

Note that the string into which each disk record is read must be dimensioned to a length which matches the expected page width of the report (lines 5 and 10). This ensures that the extra blank padding that fills each disk record will not be printed out causing extra blank lines on most printers.

Note also that line 20 changes the system string delimiter to a value that is illegal in normal print files. This ensures that the entire content of each line will be assigned to and printed from A\$ regardless of which characters appear in the print file. If this were not done any commas in the print file would cause erroneous output.

5.22.2.4 Draining File Output To A Null Device

During the program development and test process or in a reduced system hardware environment it is sometimes useful to run a program which outputs one or more files and be able to suppress one or more of the output files while the rest of the program runs normally. In Micropolis BASIC this is easily accomplished by changing the filename in the open statement of each file to be suppressed to a "*N". When the program is run all output to "*N" files will be suppressed or drained away without otherwise affecting program operation. The following program illustrates this idea.

```

10 DIM A$(4,30)
20 FOR J=1 TO 4:A$(J)="" :NEXT J
30 INPUT " FIRST LINE ";A$(1)
40 INPUT "SECOND LINE ";A$(2)

50 INPUT " THIRD LINE ";A$(3)
60 INPUT "FOURTH LINE ";A$(4)
70 B$="LABELS"
80 INPUT "ADD TO DISK FILE (Y/N)";X$
90 IF X$ = "Y" THEN B$="*N"
100 C$="*P"
110 INPUT "PRINT LABEL (Y/N)";X$
120 IF X$= "Y" THEN C$="*N"
130 X$=""
140 OPEN 1 B$
150 PUT 1 A$(1)+X$+A$(2)+X$+A$(3)+X$+A$(4)+A$
160 CLOSE 1
170 OPEN 2 C$
180 FOR J=1 TO 4:PUT 2 A$(J):NEXT J
190 CLOSE 2
200 GOTO 20

```

The file output section attempts to add four lines of input to a label file and then print a copy of the new label entry. If either or both of these functions is refused by the operator during the input section, the program changes the filename variable for the associated OPEN statement to "*N". When the output section executes the refused function output is simply drained, i.e. not output anywhere.

5.22.2.5 Echoing Of Terminal Output To Printer

On systems with a video terminal and printer device it is often desirable to obtain a hard copy audit trail of all system program operation, including all of the prompts and system messages normally directed to the terminal only. This is easily done by using the statement

ASSIGN (2,3).

This statement causes the hard copy printer to be connected to logical output stream 1 which includes all print statements, input dialogue, keyboard echoing, *T files, and system messages; and to logical output stream 2 which includes all *P print files. Thus everything aimed at the terminal thru stream 1 will also go to the printer.

This echo mode remains active until changed. The statement ASSIGN (2,2) will restore the system to normal which is device 1 (terminal) connected to stream 1 and device 2 (printer) connected to stream 2.

(This page left blank deliberately.)

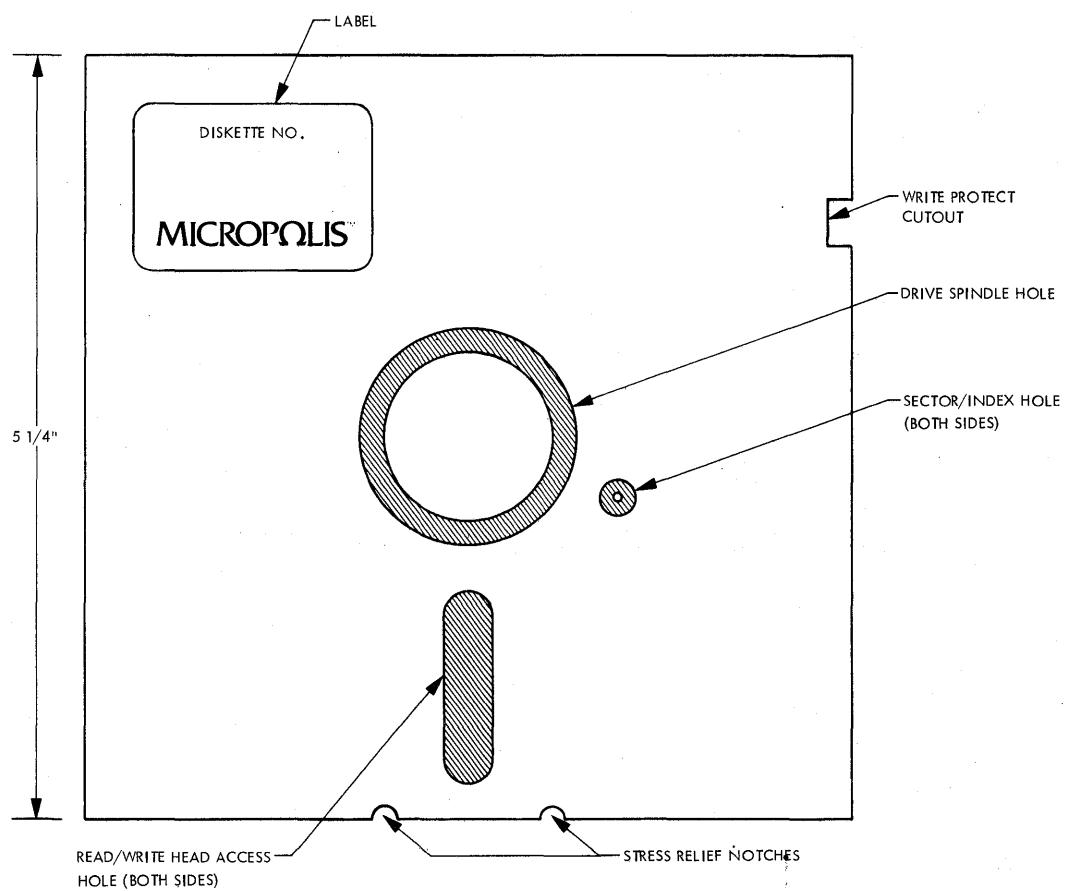


Figure 6.1

VI. DISK SUBSYSTEM THEORY AND DIRECT PROGRAMMING

6.0 INTRODUCTION

This section describes the Micropolis flexible disk subsystem in sufficient detail to enable an experienced 8080 assembly language programmer to implement a disk driver.

6.1 FUNDAMENTALS OF THE FLEXIBLE DISK: MEDIA

6.1.1 Recording Medium

The recording medium used with the Micropolis flexible disk subsystem is illustrated in Figure 6.1. The medium consists of a thin, oxide coated circular disk permanently housed in a protective plastic jacket. The disk rotates freely within the jacket, which is lined with a material that cleans the disk as it rotates. Several holes in the plastic jacket allow a disk drive to access the disk. When a diskette is loaded into a drive, the disk is clamped to a motor-driven spindle through the drive spindle hole. The read/write head and the load pad which presses the disk against the head, access the disk through the read/write head access holes. A photo detector senses sector and index holes through the sector/index hole. A switch in the disk drive senses the Write Protect cutout. If a Write Protect tab is placed over the cutout, the diskette may be read, but may not be written on. If the cutout is open, both read and write operations may be performed.

6.1.2 Disk Data Format

Figure 6.2 illustrates the format of data recorded on the diskette. Data is recorded on the diskette on concentric tracks. The outermost track is Track 0 and the innermost track is 76 in Mod II subsystems and Track 34 in Mod I subsystems. Each track has an unformatted capacity of 6250 bytes. Disk data transfers are performed on a block basis, which would require a 6250 byte RAM buffer in the computer for a full track size block. This buffer size is wasteful of memory, so the actual format used divides a track into blocks of more manageable size called sectors. The format used in the Micropolis flexible disk subsystem divides each track into 16 sectors. The beginning of each sector is indicated by a sector hole punched in the disk. This hole is sensed by a sector/index sensor in the disk drive. An index hole is located halfway between the holes for sector 15 and sector 0 and indicates the next hole is sector 0.

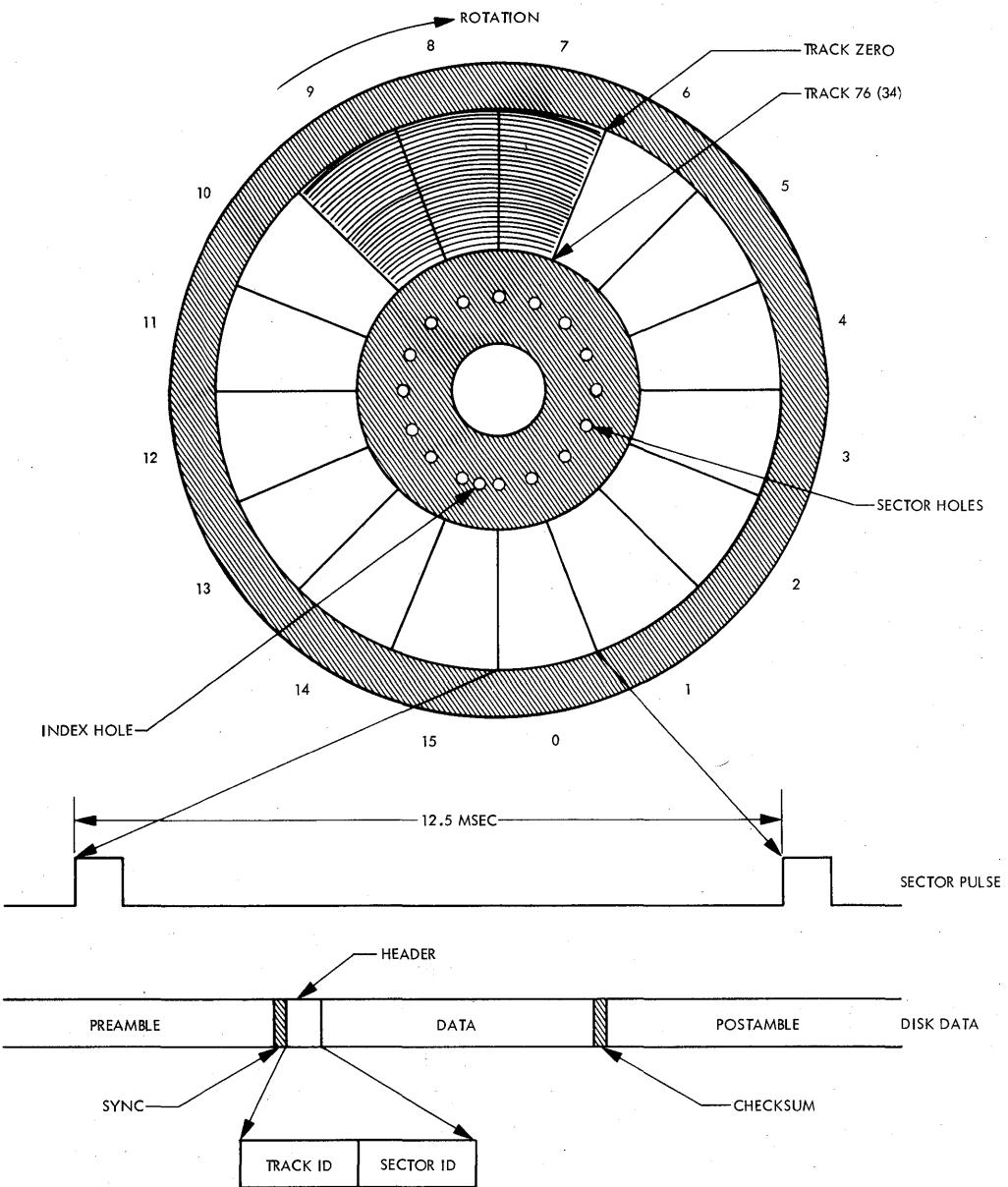


Figure 6.2

Each sector has an unformatted capacity of approximately 390 bytes. However, not all of the available storage space can be used for data. The electronics in the disk drive and the nature of the media and drive mechanism require a certain amount of space be given up to accommodate the electronic characteristics and to allow sufficient tolerance in the recording format to permit interchanging diskettes between different disk drives. Briefly, the factors which must be taken into account are: mechanical tolerance in the physical distance between sector holes punched in the disk; alignment of the sector/index sensor with respect to the read/write head; response of the sector/index sensor and logic; disk speed variation; write clock frequency tolerance; and, acquisition time of the read data decoder.

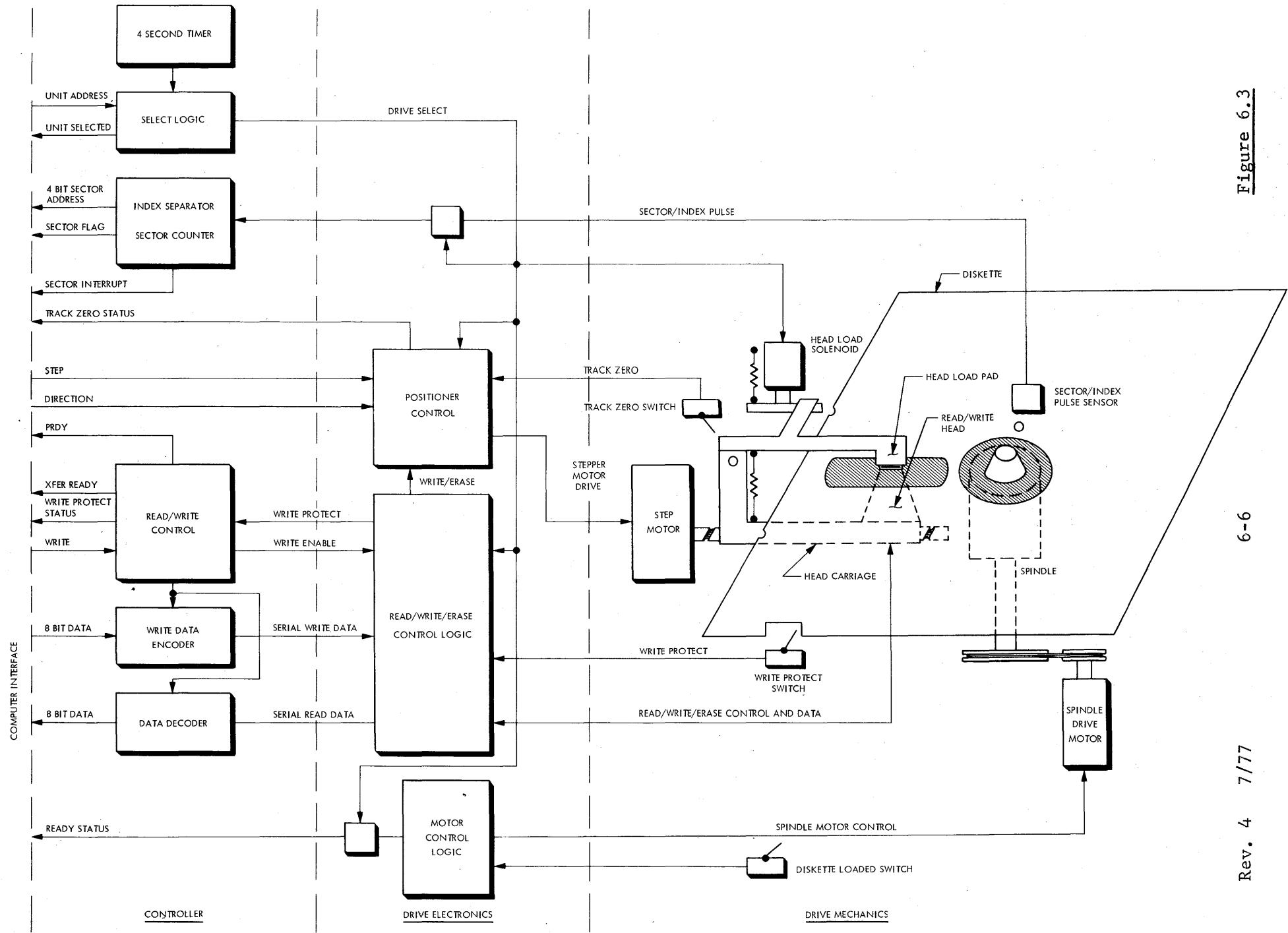
The recommended sector format is illustrated in Figure 6.2. This is the format used in disk files created by the Micropolis Disk Extended BASIC software and is the format required by the disk bootstrap located on the controller board. This format was designed to make the best trade-off between storage capacity and tolerance margins. Although other formats could possibly utilize more storage capacity, they would be incompatible with the bootstrap and a complete discussion of the engineering considerations necessary to design another format is beyond the scope of this section.

A disk sector consists of the following fields:

- 1) Preamble: The preamble is composed of approximately 40 bytes of zero (0) data bits. The preamble is automatically generated by the disk controller and is necessary to provide tolerance for the mechanical alignment and electrical characteristics of the sector/index sensor. It also provides a field of known data pattern for synchronization of the read data decoder.
- 2) Sync: The sync byte is a byte of 0FFH data which is used in the disk controller to define the beginning of useful data.
- 3) Header: The header is a 2 byte block consisting of the binary track address of the track on which the sector resides (0-76 (34)) and the address of the sector (0-15). The header is used to verify that the proper sector is being accessed in a disk I/O operation.
- 4) Data: The data field consists of 266 bytes of user data.
- 5) Checksum: The checksum is a one byte error detection code which provides error detection in read operations. The checksum is computed as follows: a) The accumulator and carry are initially cleared; b) Each byte of the header and data fields is added to the accumulator with carry. In write operations, the computed checksum is written immediately following the data field. In read operations, the checksum is re-computed from the read data and is compared with the checksum byte which is read. If they do not compare, a read error has occurred.

Figure 6.3

6-6



- 6) Postamble: The rest of a sector from the checksum to the next sector hole is filled with zero data bits. The length of the postamble allows for the mechanical tolerance in the placement of sector holes on the disk and tolerance for disk speed and write clock variations.

6.2 HARDWARE FUNDAMENTALS

Figure 6.3 is a block diagram of the Micropolis flexible disk subsystem. The components of the subsystem may be grouped as: spindle drive control; sector logic; position control logic; read/write logic; select and head load logic.

- 1) Spindle Drive Control: The disk drive spindle motor is controlled by a micro-switch that senses when the diskette is inserted and loaded, or unloaded. When the diskette is loaded, the disk is accelerated to a speed of 300 RPM. After an appropriate delay to allow the speed to stabilize, the drive is ready to accept commands. If the drive is selected by the controller, the drive will indicate this state by asserting ready status.
- 2) Sector Logic: When the disk is rotating, the sector/index hole sensor provides the controller with an electrical pulse corresponding to each hole punched in the disk. The controller separates the sector and index pulses and counts the sector pulses, thereby providing the programmer with the 4 bit address of the sector currently passing under the read/write head. A flag bit in the status register is provided to indicate when the sector address is valid and when a read or write operation may be initiated.
- 3) Position Control Logic: The read/write head is mounted on a carriage which is moved from track to track by a stepper motor-driven lead screw. Positioning is accomplished by specifying the desired direction (in or out) and issuing a step command. Control logic in the drive electronics generates all the signals necessary to cause the motor to move a track in the desired direction. When a drive is first selected, such as at power on, the track position of the drive is indeterminate. Before read or write operations may be performed, the positioner must be recalibrated as follows: when the carriage is positioned at track 0, a microswitch associated with the positioning mechanism is made. The state of this "track 0" switch is provided as a status bit. Recalibration consists of examining the track 0 status and if it is not true, issuing a command to step out. After an appropriate delay to allow the command to be executed, the process is repeated. Once the positioner has been calibrated, the software must keep track of the current position.

- 4) Read/Write Logic: Data is transferred between the computer and the controller on a byte-by-byte basis. For write operations, the controller generates the preamble and then converts 8-bit byte data from the computer to the serial data which is recorded on the disk. When the computer stops supplying data, the controller automatically writes zero data to the rest of the sector until a sector pulse is sensed. For read operations, the controller converts the serial data stream coming from the disk to 8-bit bytes and automatically detects the sync byte to determine when valid data is available.

The controller generates a "transfer ready" status flag which indicates that the controller is ready to accept data in a write operation, or that data is available in a read operation.

The controller is accessed using a technique called "memory-mapped I/O". This means that the controller command, status and data registers are treated as memory addresses and that controller read/write commands are actually memory reference instructions. When the controller data register is accessed in a read or write operation, the controller forces the computer to wait until the controller is ready to transfer data. From the computer's point of view, the controller appears to be slow memory.

The read/write control logic in the drive electronics provides the conversion between the serial digital data at the controller interface and the serial data signals at the read/write head. Whenever the drive is performing a write operation, the positioner control and read logic is disabled and the appropriate signals are generated to drive the read/write and erase heads. The erase head used in flexible disk drives is a "trim" erase head. Old data written on a sector is implicitly erased by being written over by new data. However, any slight track positioning errors could cause sufficient remnant old data to be left in the space between tracks to cause data reliability problems. To eliminate this error source, an erase head which erases the disk a small distance on either side of the newly written data is provided. This erase head is located a small distance behind the read/write head and cleans up the inter-track gap after data is written.

When a write operation is terminated by the occurrence of a sector pulse, the erase head is left on a sufficient amount of time for the last data written to be trimmed. Since the position control and read logic will be inhibited until the write operation is complete (including the erase), a new operation must not be attempted for at least one millisecond after the termination of a write operation.

The drive contains a microswitch which senses the write protect cutout in the diskette jacket. When the write protect tab is installed, the write/erase control logic is inhibited. The state of the write protect switch is available as a status bit.

- 5) Select and Head Load Logic: The controller will support up to 4 disk drive units connected in a "daisy chain" configuration. The drive electronics in each unit are conditioned by the drive select such that only one drive at a time will respond to, or provide, signals on the controller/drive interface. When a drive is not selected, the spring-loaded pressure pad which holds the disk in contact with the read/write head is moved away so that there is no contact and the head is "unloaded". When the drive is selected, a solenoid is energized, which allows the load pad to contact the disk so read or write operations may be performed. The controller contains a 4-second timer which automatically deselects all units if the controller has not been accessed for four seconds.

6.3 CONTROLLER REGISTERS

The disk controller occupies a 1K byte block of memory from F400H to F7FFH. The first half (F400H to F5FFH) is reserved for on-board bootstrap ROM. The controller command, status and data registers start at address F600H and are defined as follows:

- 1) Output Registers

Command Register

F600H or
F601H

	7	6	5	4	3	2	1	0
	COMMAND CODE	/	/	/	/	/	/	MOD

MOD = Command Modifier

The commands available are:

Code	Command	Modifier
001	Select drive	Contains drive unit address (0-3)
010	Set interrupt enable (controls sector pulse interrupt)	01 = enable interrupt 00 = disable interrupt
011	Step 1 track	00 = step out 01 = step in
100	Enable write	Not used
101	Reset controller	Not used

Write Data Register

F602H

If the write data register is referenced when the transfer flag is set during a write operation, the controller expects a data byte to be on the S100 buss data lines. The PRDY line will be held false until the controller has accepted the data, then the PRDY line will be set true for 1 bit time (4 usec). (See the status register description for the definition of the transfer flag.)

2) Input Registers

Sector Register

F600H

7	6	5	4	3	2	1	0
S	I	/	/				
C	N	/	/				
T	T.	/	/				
R.	F	/	/				
F	L	/	/				
L	G.	/	/				
G.							

SECTOR
ADDRESS

Bits

Definition

0-3 Sector Address: Address of the sector currently passing under the read/write head of the selected drive.

4,5

Reserved.

6

Sector Interrupt Flag: Indicates an interrupt request has been generated by a sector pulse. Flag is reset by issuing a reset or an interrupt disable command.

7

Sector Flag: Indicates the sector address is valid and that a read or write operation may be performed. Flag is true for 30 usec at the start of each sector. All data transfers must be initiated within 100 u seconds of the flag going true.

Status Register

F601H

7	6	5	4	3	2	1	0
X	P	R	W	T	S	U	A
F	I	E	P	K	L	N	D
E	N	A	T	Ø	T	I	D
R.	T	D			D	T	R
F	E	Y					
L							
G.							

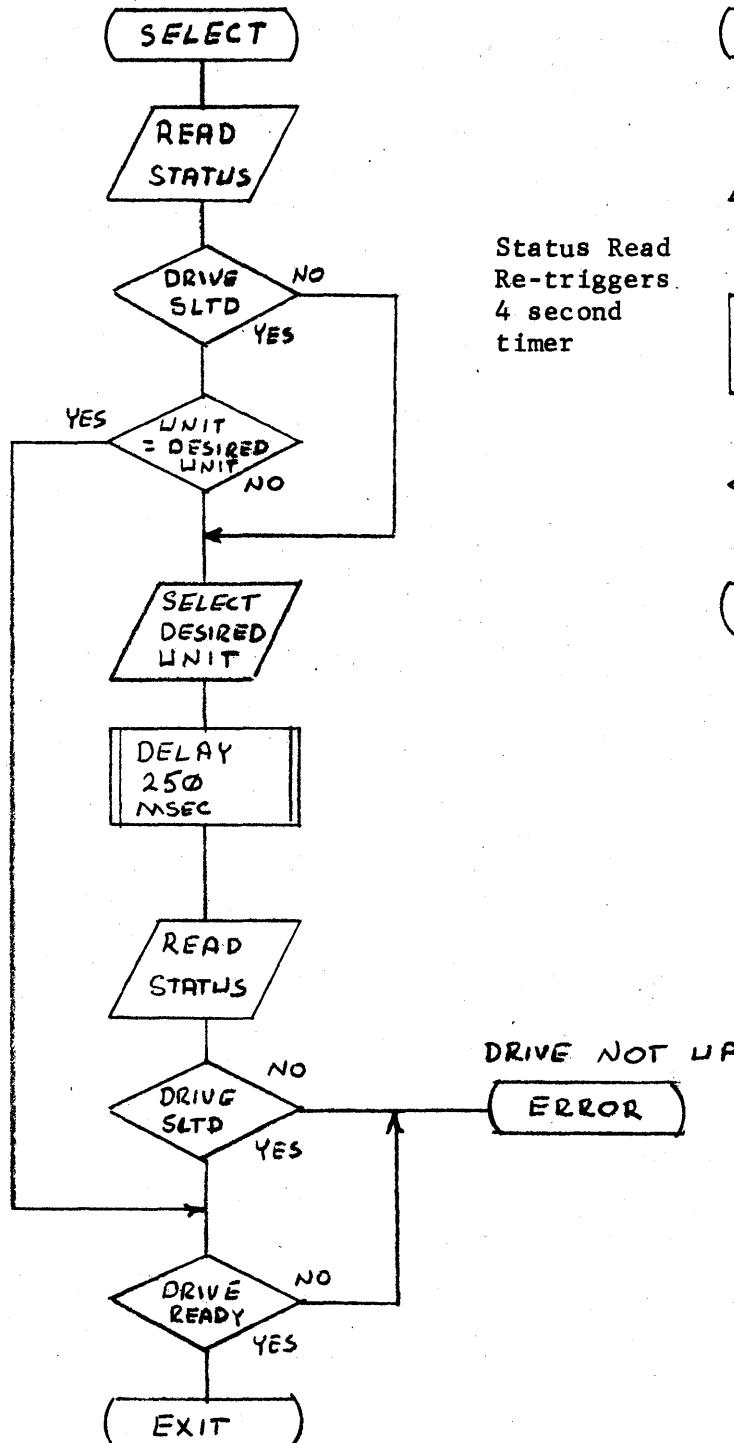
Bits	Definition
0-1	<u>Unit Address:</u> Address of the currently selected drive. Address is valid only if SLTD is true.
2	<u>SLTD:</u> Unit selected. This flag is low true, i.e., <p style="text-align: center;">\emptyset = Selected 1 = Not selected</p> <p>SLTD is true if a drive has been selected and the 4-second timer has not expired. SLTD is low true so that the software may detect when the controller is not installed (non-existent memory references yield $\emptyset FFH$).</p>
3	<u>TK\emptyset:</u> Track \emptyset status from selected drive.
4	<u>WPT:</u> Write protected status from selected drive.
5	<u>READY:</u> Ready status from the selected drive. When true, indicates the drive is ready to perform commands.
6	<u>PINTE:</u> PINTE status from the S100 BUSS.
7	<u>XFER FLAG:</u> Transfer flag. In write operations, indicates that the controller is ready to accept data from the computer. In read operations, indicates the controller has data available to the computer. When the software detects the transfer flag has set, all data transfers are performed by accessing the controller data register, which automatically synchronizes the transfer by use of the PRDY line.

Read Data Register

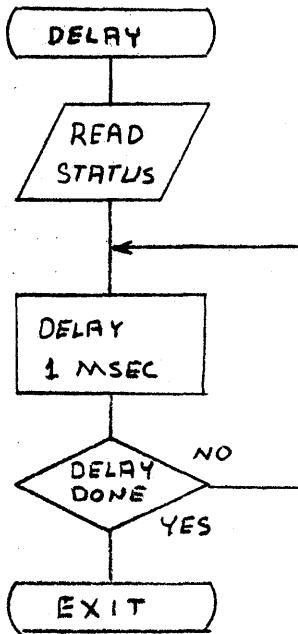
F602H If the read data register is accessed when the transfer flag is set during a read operation, the controller will hold the PRDY line false until a byte of data is available. The controller will then place the data on the S100 BUSS data lines and set PRDY true for 1 bit time (4 usec). The data will only be available for this 1 bit time period.

Figure 6.4

DRIVE SELECT LOGIC



N MILLISECOND TIMER



6.4 DISK OPERATIONS

The following paragraphs describe in detail the steps involved in performing each of the operations required to operate the Micropolis flexible disk drive subsystem.

6.4.1 Select a Drive

A drive must be selected prior to any status read, step or data transfer operation. Selection must be performed for each operation since the 4 second timer may have deselected a unit since it was last accessed. The important considerations in selecting a drive are:

- 1) When the drive is selected, the head will be loaded. A minimum of 75 milliseconds must be allowed for the head to load and settle.
- 2) The sector counter is located in the controller. When a drive is selected, a minimum of 250 milliseconds must be allowed for the sector counter to synchronize to the drive.

Figure 6.4 is a flowchart of the select operation.

NOTE that all delays are generated by a software timing loop subroutine. A read status command is included to re-trigger the 4 second timer every time the delay routine is entered.

6.4.2 Position the Head

A drive must be selected before a step command can be issued to cause the head to move 1 track. One step command of the appropriate direction (in or out) must be issued for each track moved. A minimum delay of 30 milliseconds must be allowed between each step command. (Note a step in moves the head toward the center of the disk and therefore to a higher track number.) Typical logic to implement a 1 track step is illustrated in Figure 6.5.

After the head is positioned to the desired track, an extra delay must be allowed for the head to settle before read/write operations are attempted. The complete process for an N track move is illustrated in Figure 6.6.

6.4.3 Restore to Track Ø

When a drive is first selected, the position of the read/write head is indeterminate. Prior to performing disk data transfers, the positioner must be "recalibrated" which consists of stepping the head out until the track Ø switch is made. If the drive already indicates track Ø status when first selected, the head is stepped in 8 tracks, then out to ensure a good track Ø position. Once calibrated, the software must keep track of the current head position for each drive. The restore logic recommended is illustrated in Figure 6.7.

Figure 6.5

STEP 1 TRACK

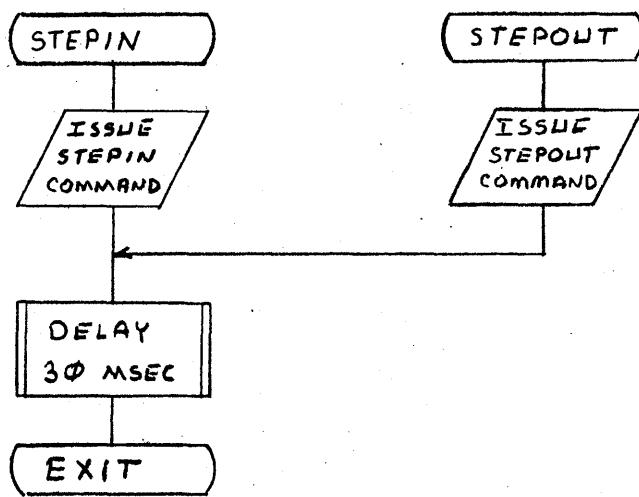


Figure 6.6

POSITION N TRACKS

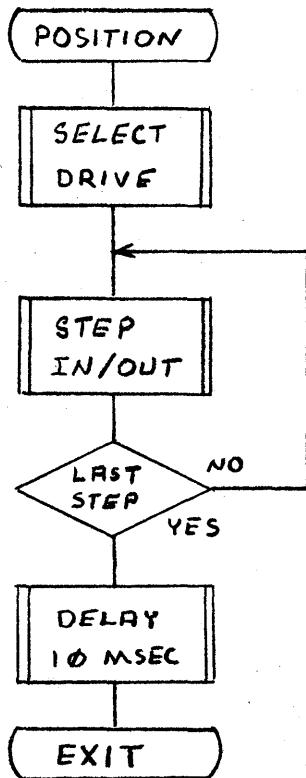
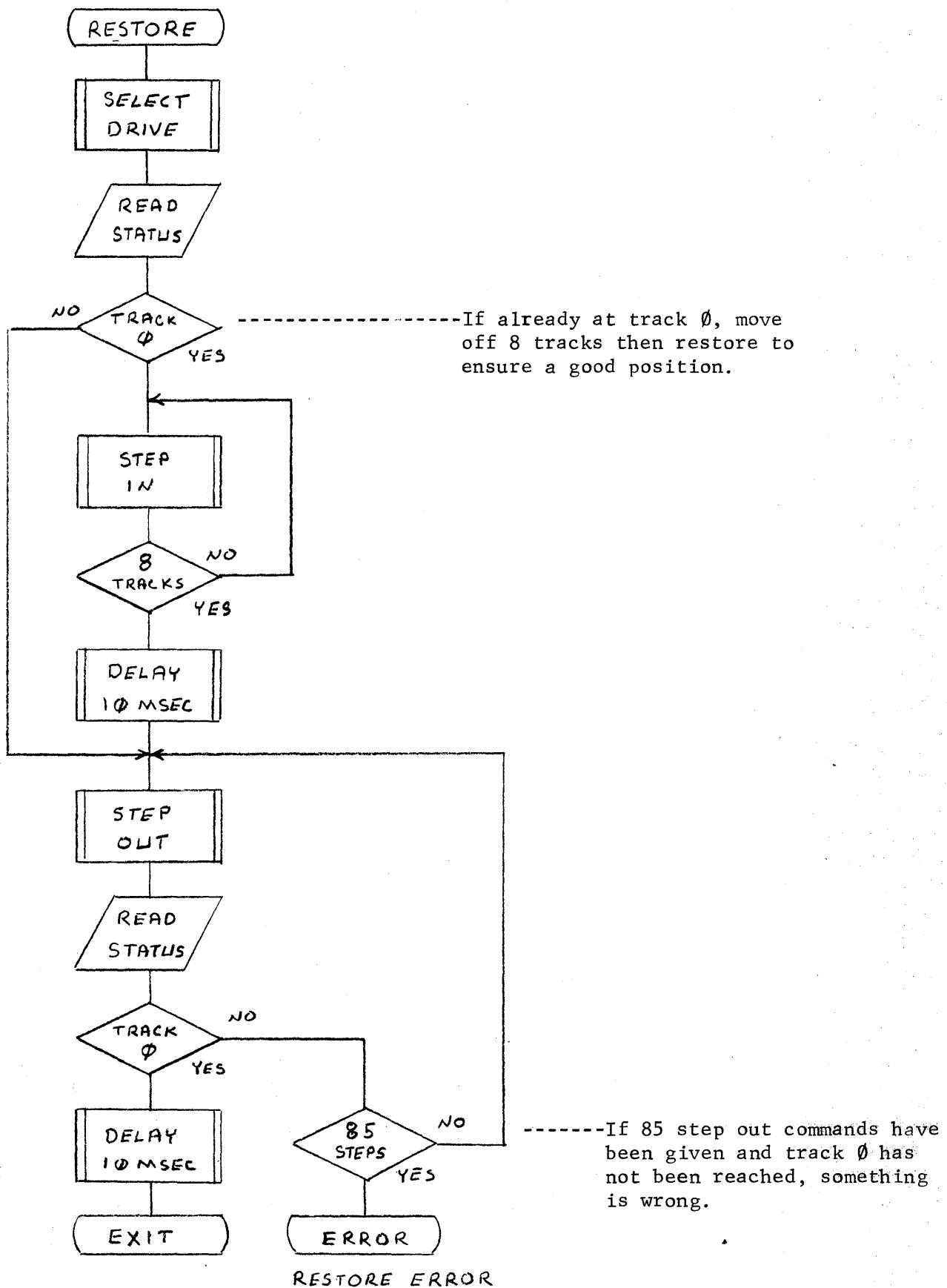


Figure 6.7

RESTORE TO TRACK 0

6.4.4 Write Operation

Figure 6.8 illustrates the logic necessary to perform a sector write operation. The program illustrated requires a 268 byte memory buffer with the first two bytes set to the track and sector address. The sync byte and checksum are generated in the program. The steps involved in writing a sector are:

- 1) Move the data to the write buffer.
- 2) Select the drive.
- 3) Wait for sector flag. When the flag goes true compare the sector address with the desired sector address. When the desired sector is found, issue an enable write command.
- 4) The enable write command causes the controller to generate the preamble. Wait for transfer ready flag to indicate the controller is ready to receive data. The software must then write the sync byte. The timing of the software loop which tests for XFER ready and then outputs the sync byte is extremely critical. The sync byte must be on the S100 buss data lines within 32 usec after XFER ready sets. The following code satisfies the timing requirements:

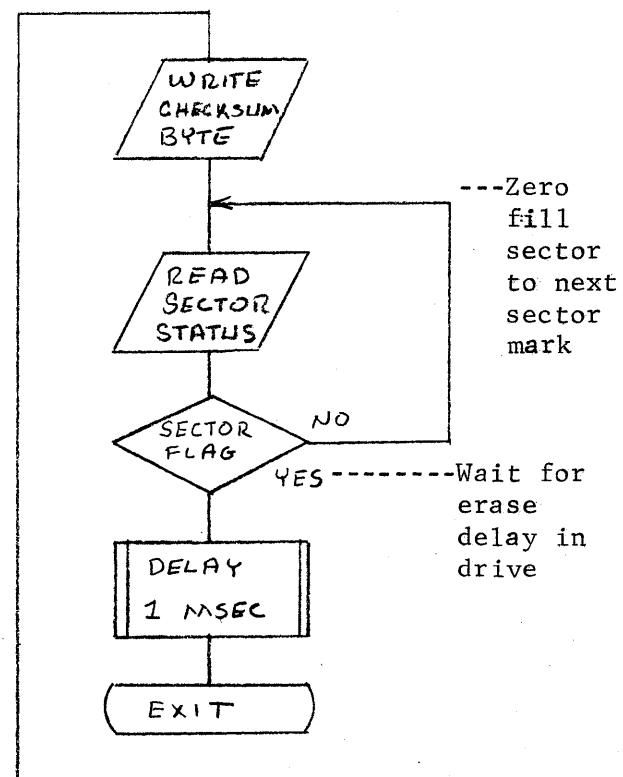
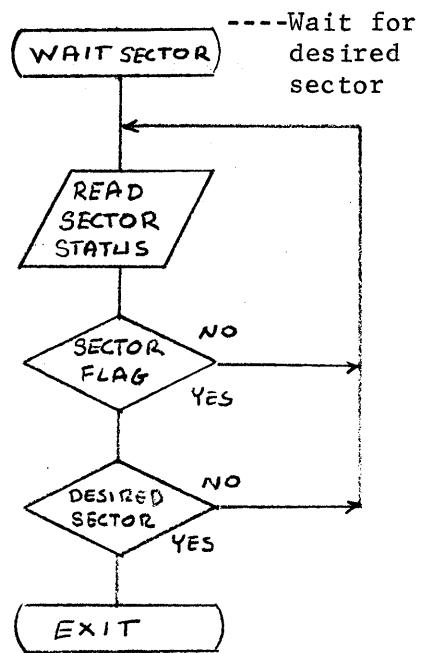
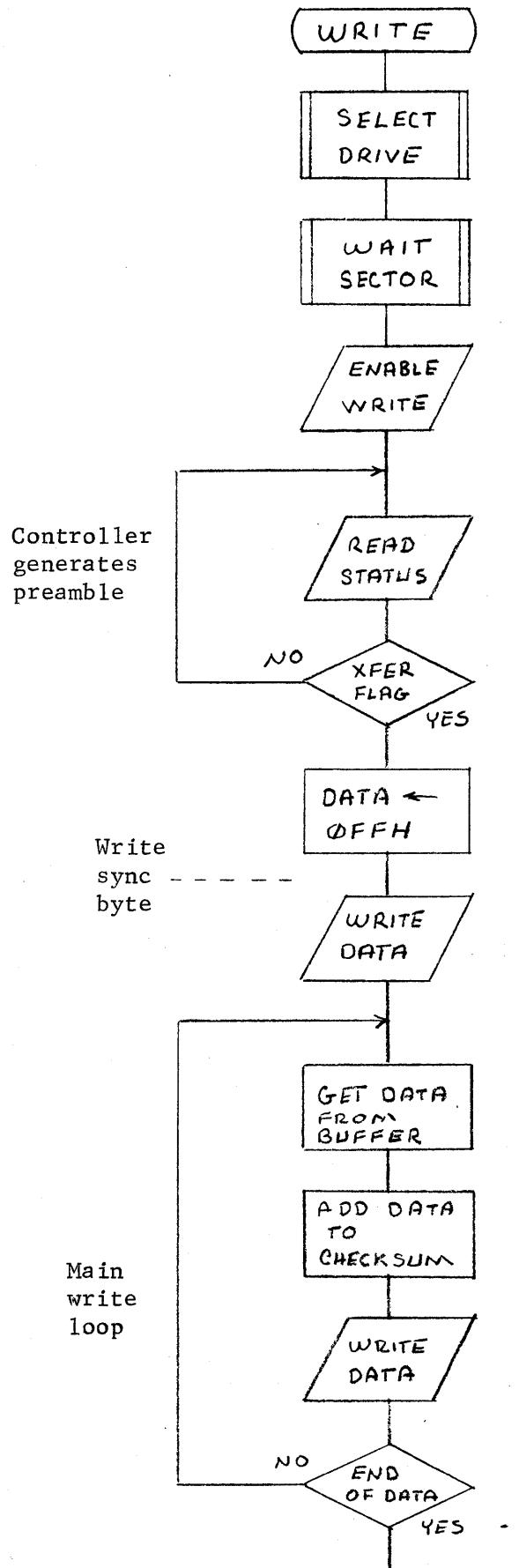
(HL = F601H and A ≠ Ø when this loop is entered)

*Wait for XFER ready flag

```
WAIT ORA M
      JP WAIT
*INSERT SYNC BYTE
      INX H
      MVI M, 0FFH
```

- 5) Each successive data byte must be made available within 32 useconds of the previous byte. When the data register is accessed, the controller will hold PRDY false until it accepts the data and then allow PRDY to go true for 1 bit time. The timing constraints on the write loop are therefore a maximum loop time of 32 useconds and a minimum loop time of 1 bit time (4 useconds). These figures do not include any margin for clock tolerance, so the actual design goals should be about 28 and 6 useconds for a conservative design.
- 6) When the checksum has been written, stop accessing the controller write register. The controller will automatically zero fill the rest of the sector.
- 7) After the checksum is written, the program waits for the next sector flag. At this time the controller terminates the write operation and the erase delay in the drive starts. The 1 millisecond software delay allows sufficient time for the erase delay to expire so that step and read functions are again enabled.

Figure 6.8 SECTOR WRITE



6.4.5 Read Operation

Figure 6.9 illustrates the logic necessary to perform a sector read operation. The program illustrated requires a 268 byte read buffer. The track/sector ID will be read into the first two bytes of the buffer and when the operation is complete, will be compared against the desired track/sector address. The steps involved in reading a sector are:

- 1) Select the drive.
- 2) Wait for the sector flag. When the sector flag is true, compare the sector address with the desired sector.
- 3) When the desired sector is found, wait for the transfer flag to set to indicate disk data is available. Note that no command is necessary to start a read operation, but you must always wait for a sector flag to indicate the start of the read.
- 4) When the transfer flag is set, the sync byte will be available in 25-28 useconds. The sync byte will only be available for 3-4 useconds so the timing of the loop which checks for the transfer ready flag is critical. The following code satisfies the timing requirements:

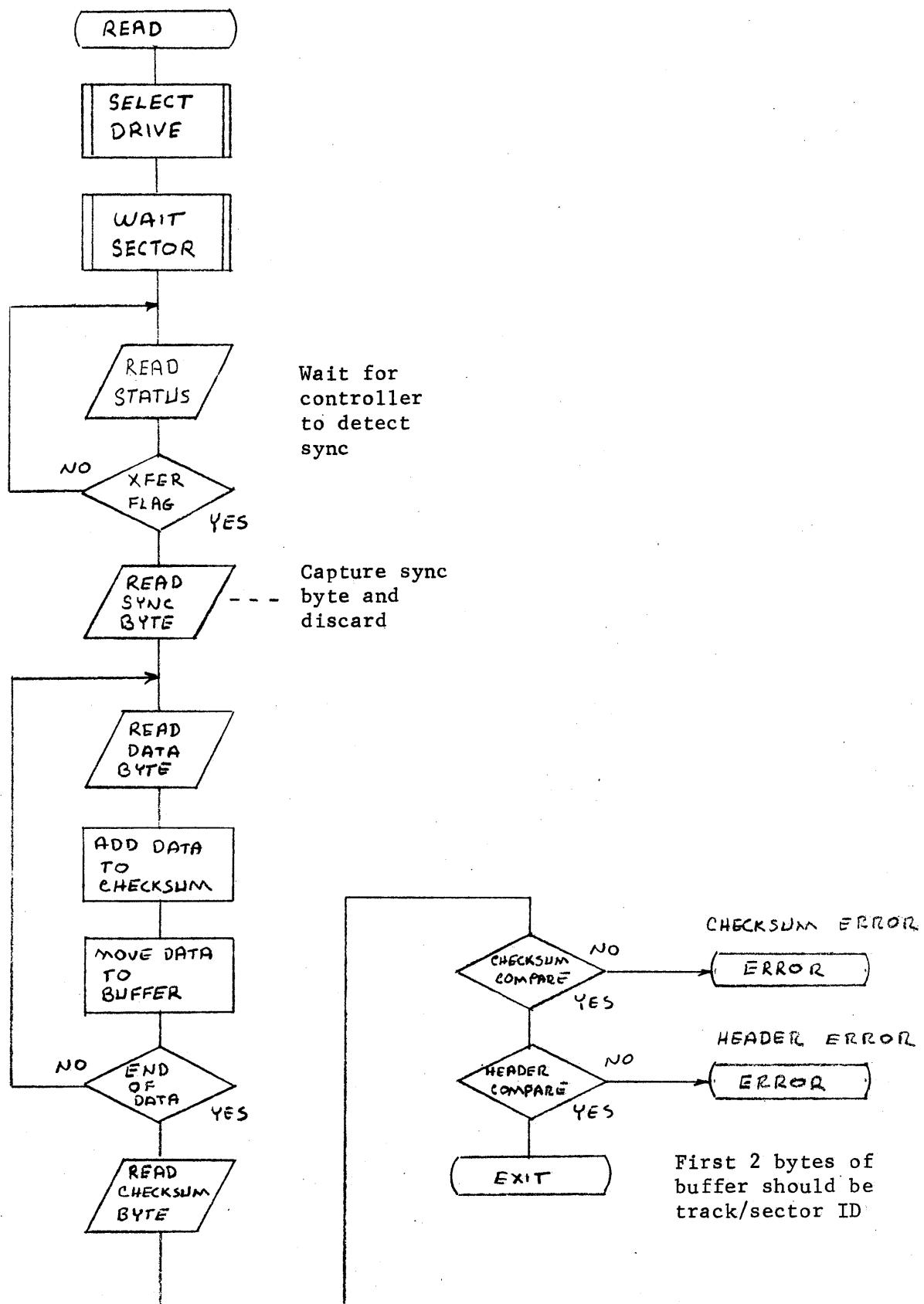
(HL = F601H and A = Ø when this loop is entered)

* Wait for XFER RDY flag

```
WAIT    ORA M
        JP WAIT
*GOBBLE SYNC BYTE
        INX H
        MOV A,M
```

- 5) Each successive data byte will be available within approximately 25 useconds and will be available for about 3 useconds. When the controller data register is accessed, the controller will hold PRDY false until the data is ready, then will place the data on the S100 buss data lines and allow PRDY to go true for 1 bit time. Once the software has read a byte, it must not access the data register again until this bit time has expired. The timing constraints on the read loop are therefore a maximum loop time of 25 useconds and a minimum loop time of 5-6 useconds. These figures reflect a conservative margin to allow for timing variations in the disk read data.
- 6) The last byte to be read from the disk is the checksum. The checksum read should be compared with the re-computed checksum, to determine if a read error has occurred.

Figure 6.9 SECTOR READ



- DATA SHEET
- REV. A
- 7) If no checksum error is detected, the first two bytes read should be compared with the desired track and sector addresses to ensure the correct sector was read.

6.5 ERROR HANDLING

An important consideration which may not be ignored in the design of a flexible disk driver is the handling of errors which occur. Magnetic storage devices in general are subject to errors. The susceptibility of the diskette to damage or contamination due to handling makes error handling particularly important in flexible disk systems. Most errors are of a temporary nature and will be invisible to the system with a properly designed driver.

Most errors can be attributed to one or more of the following sources:

- 1) Transient Electrical Noise
- 2) Media Contamination - Particles of foreign substances may become lodged between the head and the recording surface of the disk and cause data errors.
- 3) Head Positioning - The read write head may be positioned to the wrong track if the specified step rate is exceeded or may be marginally positioned if a drive is misadjusted.
- 4) Disk Centering - Due to the flexible material of which the disk is constructed, or in the event the disk is damaged or distorted due to mis-handling, it is possible that a diskette may be improperly clamped to the spindle in the disk drive.

The following procedures are recommended to perform proper error handling in disk read/write operations:

Read Operations

- 1) Step the positioner to the desired track.
- 2) Perform the read operation as described in Section 6.9.5. If a header or checksum error occurs, re-read the sector up to 5 times.
- 3) If the 5 retries were unsuccessful, step the positioner off one track and then back to the desired track. Repeat Step 2. If still unsuccessful, step the positioner off one track in the other direction and then back. Repeat Step 2.
- 4) Perform the restep procedure given in Step 3 up to 4 times. If still unsuccessful, deselect the unit and wait about 200 milliseconds for the head to unload. Reselect the unit, restore to track 0, and re-seek to the desired track. Repeat Steps 2 and 3.
- 5) Perform the reselect function given in Step 4 up to 3 times. If still unsuccessful, abort the operation with a permanent I/O error.

Write Operation

- 1) Step the positioner to the desired track.
- 2) Read the sector immediately preceding the desired sector. Any errors which occur should be handled in the manner described for normal read operations. This operation ensures the head is properly positioned to the right track and the sector counter is synchronized with the disk.
- 3) Write the desired sector as described in Section 6.4.4.
- 4) Read the sector just written to ensure the data was recorded properly. If an error occurs, repeat Steps 2, 3, and 4 up to 5 times.
- 5) If unsuccessful, perform the restep operation as described for the read operation and repeat Steps 2, 3, and 4.
- 6) If 4 restep operations are unsuccessful, perform the reselect operation as described for the read operation.
- 7) If 3 reselect operations are unsuccessful, abort the operation with a permanent I/O error.

If a permanent I/O error occurs, the disk may be improperly centered, there may be a defect in or damage to the recording surface of the disk, or the disk may have been written on a marginal drive.

The "restep" procedure described takes advantage of the hysteresis present in all positioning systems. Friction in the positioner causes the head position to deviate slightly from the nominal track position. This position will be different when the head is stepped to a track from different directions. In normal operations, this slight position error is well within the tolerance limits for proper operations. However, if errors are encountered in reading a disk which was written on another drive that is marginally aligned, the slight difference may be enough to recover the data.

The "reselect" procedure serves to dislodge any foreign particles and to recalibrate the positioner, should it be positioned to the wrong track.

6.6 DISK DRIVER

As a comprehensive example of all the principles presented in this section, a sample disk driver is presented here. This driver provides the facilities to seek to a track, seek and read a sector, seek and write a sector, and seek and verify a sector. This verify operation is a special case of a sector read but only the header bytes are transferred into the buffer. This allows the use of a single disk buffer to perform write operations, which consist of a header check prior to write, writing the sector, and a read-after-write check.

The power-on recalibration is transparent. The driver maintains a table containing the current track address of each drive connected to the controller. The user's power on initialize software must set the entries in this table to $\text{\$}FFH$. The first time a drive is accessed, the driver will recognize this flag and recalibrate the positioner on the drive before performing the specified operation.

When the driver is called, the HL register must point to a parameter block (referred to as a disk control block) which specifies the operation to be performed. When the driver returns, the condition code will reflect the status of the operation. (See the listing for details.)

The DCB is structured as follows:

ADDRESS	7	6	5	4	3	2	1	0		
DCB + 0	/ / / / / / / /								FN CODE	
DCB + 1	ID	R	/ / / / / / / /							
	F	A	/ / / / / / / /							
	L	W	/ / / / / / / /							
	A	F	/ / / / / / / /							
	G	L	/ / / / / / / /							
		G	/ / / / / / / /							
DCB + 2	S E C T O R		A D D R E S S							
DCB + 3	T R A C K		A D D R E S S							
DCB + 4	B U F F E R		A D D R E S S							
DCB + 5	B U F F E R		A D D R E S S							
			LSB							
			MSB							

The DCB entries are described as follows:

<u>FN CODE</u>	<u>Function code</u>
0	= Seek only
1	= Seek and read sector
2	= Seek and write sector
3	= Seek and verify sector
<u>ID FLAG</u>	<u>Pre-Write Header (ID) Check Flag</u>
0	= Perform check
1	= Inhibit check
<u>RAW FLAG</u>	<u>Read-After-Write Check Flag</u>
0	= Perform check
1	= Inhibit check
<u>UNIT ADDR.</u>	<u>Drive Unit Address</u>
0 - 3	

Sector and Track Address are the address of the sector which is to be written or read and the address of the track upon which the sector resides. The driver will seek as necessary to move the head to the desired track.

The Buffer Address is a 16 bit memory address stored in standard 8080 low/high format. This must be the address of a 268 byte read/write buffer. The first two bytes of the buffer are reserved for the header.

To perform a write operation, move the data to the read/write buffer, set up the DCB, and call the driver.

To perform a read operation, set up the DCB and call the driver. When the operation is complete, the data from the desired sector will be in the read buffer.

* DISK DRIVER FOR MICROPOLIS *
* FLEXIBLE DISK SUBSYSTEM *
* COPYRIGHT MICROPOLIS CORPORATION *
* 8 JUNE 1977 *
* *****

* 1) CALLING SEQUENCE:

* LXI H,UDCB POINT HL TO USER
* CALL DSKIO DCB & PERFORM
* JNZ ERROR OPERATION

* UDCB IS THE USER'S DISK CONTROL
* BLOCK WHICH DEFINES THE OPERATION
* TO BE PERFORMED AND IS STRUCTURED
* AS FOLLOWS:

* UDCB+0 FUNCTION CODE
* 0 SEEK TRACK ONLY
* 1 SEEK AND READ SECTOR
* 2 SEEK AND WRITE SECTOR
* 3 SEEK AND VERIFY SECTOR

* WRITE OPERATIONS CONSIST OF:
* 1) VERIFY THE TRACK/SECTOR ID
* IN THE SECTOR IMMEDIATELY
* PRECEEDING THE DESIRED SECTOR
* 2) PERFORM THE WRITE OPERATION
* 3) THE SECTOR WRITTEN IS THEN
* VERIFIED BY A READ-AFTER-WRITE
* CHECKSUM READ

* NOTE: THE ID CHECK AND READ AFTER
* WRITE CHECKS CAN BE OVERRIDDEN
* BY CONTROL FLAGS IN UDCB+1
* FOR WRITING ON UNFORMATTED DISKS

* UDCB+1 CONTROL FLAGS/UNIT SELECT
* BIT FUNCTION
* 0-1 UNIT ADDRESS
* 6 READ-AFTER-WRITE CHECK
* CONTROL: 0=PERFORM,
* 1=INHIBIT
* 7 PRE-WRITE ID CHECK
* CONTROL: 0=PERFORM,
* 1=INHIBIT

* UDCB+2 SECTOR ADDRESS (0-15)
* UDCB+3 TRACK ADDRESS (0-76)(34)
* UDCB+4&5 BUFFER ADDRESS

* BUFFER ADDRESS IS THE START
* ADDRESS OF THE READ/WRITE
* BUFFER TO BE USED IN
* PERFORMING THE OPERATION.

* ALL OPERATIONS
* REQUIRE A 268 BYTE BUFFER
* ORGANIZED AS FOLLOWS:
* BYTE 0 -- TRACK ID
* BYTE 1 -- SECTOR ID
* BYTE 2-267 -- DATA

* BYTES 0 AND 1 ARE FILLED
* IN AS NECESSARY BY THE
* DRIVER

* 2) THE DISK I/O DRIVER RETURNS WITH
* THE CONDITION CODE SET TO Z IF
* THE OPERATION WAS SUCCESSFUL AND
* NZ IF AN ERROR OCCURRED. THE
* A REGISTER WILL CONTAIN AN ERROR
* CODE AS FOLLOWS:
* 1 -- PERMANENT I/O ERROR - AN
* UNRECOVERABLE DISK ERROR
* OCCURRED
* 2 -- PARAMETER ERROR - ONE OF THE
* PARAMETERS IN THE DCB IS
* INVALID
* 3 -- DRIVE NOT UP - THE SELECTED
* DRIVE IS NOT READY
* 4 -- WRITE PROTECT - THE SELECTED
* DRIVE IS WRITE PROTECTED AND
* A WRITE OPERATION WAS
* SPECIFIED
* 3) INITIALIZATION REQUIREMENTS:

* 1) THE DRIVER CONTAINS A TABLE
* LABELED "TRACK" WHICH CONTAINS
* THE CURRENT TRACK POSITION FOR
* EACH DRIVE CONNECTED TO THE
* CONTROLLER. EACH ENTRY MUST BE
* INITIALIZED TO FFH TO CAUSE THE
* TRACK POSITION OF EACH DRIVE TO
* BE RE-CALIBRATED THE FIRST TIME
* IT IS ACCESSED

* 2) THE PARAMETER LABELED "TRKMX"
* MUST BE SET TO THE HIGHEST
* TRACK ADDRESS WHICH IS 76 FOR
* MOD II SUBSYSTEMS AND 34 FOR
* MOD I SUBSYSTEMS

* 3) THE 16 BIT PARAMETER LABELED
* "DADR" MUST BE SET TO THE ADDRESS
* OF THE DISK CONTROLLER WHICH IS
* THE BOOT PROM ADDRESS+200H

0000 ORG X'400'
*
0400 F3 DSKIO DI

0401 C5	PUSH B	SAVE REGISTERS
0402 D5	PUSH D	
0403 E5	PUSH H	
0404 210000	LXI H,0	SAVE STACK POINTER
0407 39	DAD SP	
0408 220807	SHLD STACK	
040B E1	POP H	GET POINTER TO
040C E5	PUSH H	USER'S DCB
040D 11F506	LXI D,DCB	COPY USER DCB TO
0410 0606	MVI B,DCBLEN	INTERNAL DCB
0412 7E DS010	MOV A,M	
0413 12	STAX D	
0414 23	INX H	
0415 13	INX D	
0416 05	DCR B	
0417 C21204	JNZ DS010	
*		
*	VALIDATE DCB PARAMETERS	
*		
041A 21F506	LXI H,DCB	FUNCTION MUST BE
041D 7E	MOV A,M	3 OR LESS
041E FE04	CPI 4	
0420 D2D205	JNC PARMER	PARAMETER ERROR
0423 23	INX H	
0424 7E	MOV A,M	UNIT ADDRESS MUST
0425 E63F	ANI X'3F'	BE LESS THAN 4
0427 FE04	CPI 4	
0429 D2D205	JNC PARMER	
042C 23	INX H	
042D 7E	MOV A,M	SECTOR MUST BE
042E FE10	CPI 16	15 OR LESS
0430 D2D205	JNC PARMER	
0433 23	INX H	
0434 3AFB06	LDA TRKMX	TRACK MUST BE LESS
0437 96	SUB M	THAN OR EQUAL TO
0438 FAD205	JM PARMER	MAX TRACK
*		
*	ENSURE DRIVE IS OPERATIONAL	
*		
043B CDE405	CALL SLCT	
*		
*	SEEK TO DESIRED TRACK	
*		
043E CDD504	CALL SEEK	
*		
*	GET FUNCTION PARAMETER FROM DCB	
*	AND PERFORM ANY OTHER REQUIRED	
*	FUNCTION	
*		
0441 3AF506	LDA DCBFN	DONE IF FUNCT=
0444 B7	ORA A	SEEK ONLY(0)
0445 CACC04	JZ DS100	DONE
*		
*	PERFORM READ/WRITE FUNCTION	
*		
*	TRY CONTROL FOR READ/WRITE	

* OPERATIONS:
 * A 3 LEVEL RETRY STRUCTURE IS
 * PROVIDED AS FOLLOWS:
 * 1 -- IF AN ERROR OCCURS, UP TO 5
 * RETRYS OF THE OFFENDING OPERATION
 * WILL BE PERFORMED
 * 2 -- IF THE LEVEL 1 RETRYS ARE NOT
 * SUCCESSFUL, THE POSITIONER WILL
 * BE STEPPED OFF TRACK AND BACK
 * AND THE LEVEL 1 RETRYS WILL BE
 * PERFORMED. THE LEVEL 2 RETRYS
 * WILL BE PERFORMED UP TO 4 TIMES
 * 3 -- IF THE LEVEL 2 RETRY
 * PROCEDURE IS NOT SUCCESSFUL, THE
 * UNIT WILL BE DESELECTED TO UNLOAD
 * THE HEAD THEN THE UNIT WILL BE
 * RESELECTED, THE POSITIONER WILL
 * BE RECALIBRATED AND MOVED BACK
 * TO THE DESIRED TRACK AND THE
 * LEVEL 1 AND 2 RETRY PROCEDURES
 * WILL BE PERFORMED. THIS WILL BE
 * DONE UP TO 3 TIMES. IF NOT
 * SUCCESSFUL, A PERMANENT I/O
 * ERROR WILL RESULT

0448 3E03	DS020	MVI A,3	PRESET RETRY
044A 320607		STA L3RTRY	COUNTERS
044D 3E04	DS030	MVI A,4	
044F 320507		STA L2RTRY	
0452 3E05	DS040	MVI A,5	
0454 320407		STA L1RTRY	
*			
* SELECT DESIRED FUNCTION AND			
* PERFORM			
*			
0457 2AF906	DS050	LHLD DCBAD	PRESET BUFFER
045A 220007		SHLD BUFADR	ADDRESS
045D 3AF506		LDA DCBFN	GET FUNCTION
0460 3D		DCR A	
0461 C26A04		JNZ DS060	
*			
* READ SECTOR			
*			
0464 CD8106		CALL READAL	READ SECTOR
0467 C3A204		JMP DS090	CHECK FOR ERROR
046A 3D	DS060	DCR A	
046B C29704		JNZ DS080	
*			
* WRITE SECTOR			
*			
046E 3AF606		LDA DCBUN	IF HEADER CHECK
0471 E680		ANI HCI	INHIBIT SET GO
0473 C28304		JNZ DS070	WRITE
0476 3AF706		LDA DCBSC	BACKSPACE SECTOR
0479 3D		DCR A	COUNT MOD 16
047A E60F		ANI X'0F'	
047C 47		MOV B,A	

047D	CDB106	CALL READCK	DO PRE-WRITE HDR
0480	C2A204	JNZ DS090	CHECK - ABORT ERR
0483	CD2F06	CALL WSECT	GO WRITE
0486	3AF706	LDA DCBSC	DO RAW CHECKSUM
0489	47	MOV B,A	READ CHECK
048A	3AF606	LDA DCBUN	UNLESS INHIBITED
048D	E640	ANI RAFI	
048F	EE40	XRI RAFI	
0491	C4B106	CNZ READCK	
0494	C3A204	JMP DS090	GO CHECK FOR ERR
0497	3D	DS080 DCR A	
0498	C2D205	JNZ PARMER	TRAP-JUST IN CASE
*			
* VERIFY SECTOR			
*			
049B	3AF706	LDA DCBSC	
049E	47	MOV B,A	
049F	CDB106	CALL READCK	DO CHECKSUM READ
*			
* CHECK FOR ERROR			
*			
04A2	CACC04	DS090 JZ DS100	NO ERROR-EXIT
04A5	3A0407	LDA L1RTRY	LEVEL 1 -- RETRY
04A8	3D	DCR A	UP TO 5 TIMES
04A9	320407	STA L1RTRY	
04AC	C25704	JNZ DS050	
*			
* RETRIED 5 TIMES - STEP OFF TRACK			
*			
04AF	CD3605	CALL RESTEP	
04B2	3A0507	LDA L2RTRY	PERFORM UP TO 4
04B5	3D	DCR A	TIMES
04B6	320507	STA L2RTRY	
04B9	C25204	JNZ DS040	
*			
* STEPPED OFF 4 TIMES - DESELECT			
* DRIVE TO UNLOAD HEAD THEN			
* SELECT, RESTORE AND RE-SEEK			
*			
04BC	CD6305	CALL RESLCT	
04BF	3A0607	LDA L3RTRY	PERFORM UP TO 3
04C2	3D	DCR A	TIMES
04C3	320607	STA L3RTRY	
04C6	C24D04	JNZ DS030	
*			
* UNSUCCESSFUL -- ABORT WITH			
* PERMANENT I/O ERROR			
*			
04C9	C3CC05	JMP PERMER	
*			
* END OF OPERATION			
*			
04CC	2A0807	DS100 LHLD STACK	RESTORE STACK PTR
04CF	F9	SPHL	
04D0	E1	POP H	RESTORE REGISTERS
04D1	D1	POP D	

04D2 C1		POP B	
04D3 00	EIADR	NOP	SPACE FOR EI
04D4 C9		RET	
*			
*			
* SEEK TO DESIRED TRACK			
*			
04D5 CDE405	SEEK	CALL SLCT	ENSURE DRIVE SLTD
04D8 E5		PUSH H	AND READY
04D9 CDBD05		CALL LDTRK	POINT HL TO TRACK
04DC 3EFF		MVI A,X'FF'	SEE IF DRIVE HAS
04DE BE		CMP M	BEEN INITIALIZED
04DF C2E504		JNZ SEEK1	YES-CONTINUE
04E2 CD7905		CALL RESTOR	CALIBRATE POSITION
04E5 3AF806	SEEK1	LDA DCBTK	GET TRACK FROM DCB
04E8 4F		MOV C,A	SAVE IN C
04E9 96		SUB M	ALREADY AT TRACK?
04EA CA0405		JZ SEEKR	YES-RETURN
*			
* NOT AT TRACK -- ISSUE THE			
* APPROPRIATE NUMBER OF STEPS TO			
* MOVE TO THE DESIRED TRACK			
*			
04ED FAFA04		JM SEKOUT	
04F0 CD0705	SEKIN	CALL STEPIN	
04F3 3D		DCR A	
04F4 C2F004		JNZ SEKIN	
04F7 C30105		JMP SEEKR1	
04FA CD1D05	SEKOUT	CALL STPOUT	
04FD 3C		INR A	
04FE C2FA04		JNZ SEKOUT	
0501 CD2D05	SEEKR1	CALL SETTLE	WAIT HEAD SETTLE
0504 71	SEEKR	MOV M,C	STORE TRACK
0505 E1		POP H	
0506 C9		RET	
*			
* STEP POSITIONER IN 1 TRACK			
*			
0507 F5	STEPIN	PUSH PSW	
0508 D5		PUSH D	
0509 E5		PUSH H	
050A AF		XRA A	SET DIRECTION FLAG
050B 320707		STA DIRCTN	
050E 2A0207		LHLD DADR	STEP IN ONE TRK
0511 3661		MVI M,STEP+1	
0513 111E00	STP1	LXI D,30	WAIT STEP TIME
0516 CD1706		CALL TIMER	
0519 E1		POP H	
051A D1		POP D	
051B F1		POP PSW	
051C C9		RET	
*			
* STEP POSITIONER OUT 1 TRACK			
*			
051D F5	STPOUT	PUSH PSW	
051E D5		PUSH D	

```

051F E5      PUSH H
0520 3EFF    MVI A,X'FF'   SET DIRECTION FLAG
0522 320707  STA DIRCTN
0525 2A0207  LHLD DADR
0528 3660    MVI M,STEP   STEP OUT ONE TRK
052A C31305  JMP STP1    GO WAIT STEP TIME
*
*
*      WAIT HEAD SETTLE TIME
*
052D D5      SETTLE PUSH D
052E 110A00  LXI D,10     10 MILLISECONDS
0531 CD1706  CALL TIMER
0534 D1      POP D
0535 C9      RET
*
*      STEP OFF TRACK ONE AND BACK TO CORRECT
*      POSSIBLE MARGINAL TRACK POSITION
*      OF DRIVE WHICH WROTE THE DISK
*      IF TRACK 0 SUBSTITUTE RESTOR
*
0536 CDBD05  RESTEP CALL LDTRK   GET CRNT TRK ADDR
0539 7E      MOV A,M     GET CRNT TRK
053A B7      ORA A
053B C24205  JNZ RSTPA
053E CD7905  CALL RESTOR   USE RESTOR IF TK 0
0541 C9      RET
0542 3A0707  RSTPA    LDA DIRCTN
0545 B7      ORA A
0546 C25605  JNZ RSTPB
0549 CD0705  CALL STEPIN
054C CD2D05  CALL SETTLE
054F CD1D05  CALL STPOUT
0552 CD2D05  CALL SETTLE
0555 C9      RET
0556 CD1D05  RSTPB    CALL STPOUT
0559 CD2D05  CALL SETTLE
055C CD0705  CALL STEPIN
055F CD2D05  CALL SETTLE
0562 C9      RET
*
*      RETRY ROUTINE TO RESTORE TO 0 THEN
*      LIFT HEAD, LOWER HEAD AND RESEEK
*
0563 E5      RESLCT PUSH H
0564 2A0207  LHLD DADR
0567 36A0    MVI M,RESET  RESET CONTROLLER
0569 11C800  LXI D,200
056C CD1706  CALL TIMER
056F CDE405  CALL SLCT    RESELECT,LOWR HEAD
0572 E1      POP H
0573 CD7905  CALL RESTOR
0576 C3D504  JMP SEEK    GO RE-SEEK
*
*
*      RESTORE POSITIONER TO TRACK 0
*      POSITIONER MUST BE STEPPED OUT
*      UNTIL THE TRACK 0 SWITCH IS MADE

```

* TO CALIBRATE TRACK POSITION

*
 0579 E5 RESTOR PUSH H
 057A C5 PUSH B
 057B CDBD05 CALL LDTRK POINT HL TO TRACK
 057E 36FF MVI M,X'FF' PRESET TO BAD TRK
 0580 CD8805 CALL RESTRI RESTORE TO TK 0
 0583 3600 MVI M,0 SET TRACK=0
 0585 C1 POP B
 0586 E1 POP H
 0587 C9 RET

*
 * RESTORE TO TK 0
 *
 0588 E5 RESTRI PUSH H
 0589 CDE405 CALL SLCT ENSURE UNIT SLCTD
 058C D5 PUSH D AND READY
 058D C5 PUSH B
 058E 2A0207 LHLD DADR POINT TO STATUS
 0591 23 INX H BYTE
 0592 7E MOV A,M ALREADY AT
 0593 E608 ANI TK0 TRACK 0 ?
 0595 CAA405 JZ REST3 NO - PRESS ON

*
 * ALREADY AT TRACK 0 - STEP
 * IN 8 TIMES THEN RESTORE
 * TO ENSURE GOOD POSITION
 *
 0598 3E08 MVI A,8
 059A CD0705 REST2 CALL STEPIN STEP IN 8
 059D 3D DCR A TRACKS
 059E C29A05 JNZ REST2
 05A1 CD2D05 CALL SETTLE WAIT SETTLE TIME

*
 * STEP OUT UNTIL TRACK 0 SWITCH
 * IS ACTUATED OR UNTIL 85 STEPS
 * HAVE BEEN ISSUED SO THAT WE
 * DONT BANG AGAINST THE STOP
 * FOREVER IF TK0 SWITCH IS
 * BROKEN
 *
 05A4 0E55 REST3 MVI C,85 LOAD MAX STEPCNT
 05A6 7E REST3A MOV A,M TRACK 0?
 05A7 E608 ANI TK0
 05A9 C2B605 JNZ REST4 YES - PRESS ON
 05AC CD1D05 CALL STPOUT STEP OUT ONE TK
 05AF 0D DCR C MAX STEPS ?
 05B0 C2A605 JNZ REST3A NO - TRY AGAIN

*
 * MAXIMUM NUMBER OF STEPS HAVE
 * BEEN ISSUED - ERROR ABORT
 *
 05B3 C3CC05 JMP PERMER

*
 * FOUND TRACK 0 - WAIT
 * SETTLE TIME THEN EXIT
 *

05B6 CD2D05 REST4 CALL SETTLE WAIT HEAD SETTLE
05B9 C1 POP B
05BA D1 POP D
05BB E1 POP H
05BC C9 RET

*

* LOAD ADDRESS OF CURRENT TRACK ON
* CURRENT UNIT INTO HL

*

05BD D5 LDTRK PUSH D
05BE 3AF606 LDA DCBUN
05C1 E603 ANI 03 MASK OUT UNIT
05C3 5F MOV E,A
05C4 1600 MVI D,0
05C6 21FC06 LXI H,TRACK POINT HL INTO
05C9 19 DAD D TRACK TABLE
05CA D1 POP D
05CB C9 RET

*

*

*

*

* ERROR EXITS

*

05CC 3E01 PERMER MVI A,1
05CE B7 ORA A
05CF C3CC04 JMP DS100
05D2 3E02 PARMER MVI A,2
05D4 B7 ORA A
05D5 C3CC04 JMP DS100
05D8 3E03 DRIVER MVI A,3
05DA B7 ORA A
05DB C3CC04 JMP DS100
05DE 3E04 PROTER MVI A,4
05E0 B7 ORA A
05E1 C3CC04 JMP DS100

*

*

*

* REGISTER DEFINITIONS AND *

* FLAG EQUATES FOR MICROPOLIS *

* FLEXIBLE DISK CONTROLLER B *

*

*

*

F400 BPROM EQU X'F400'
F600 DIADR EQU EPROM+X'0200'

*

* DATA REGISTERS

*

F602 WDATA EQU DIADR+X'02'
F602 RDATA EQU WDATA

*

* STATUS REGISTERS

*

```

F600      DSECTR EQU    DIADR
*        0-3     SECTOR COUNT
*        4     SPARE
*        5     SPARE
*        6     SCTR INTERRUPT FLAG
*        7     SECTOR FLAG
*
*        FLAG BITS
*
0040      SIFLG   EQU    X'40'
0080      SFLG    EQU    X'80'
0020      DTMR    EQU    X'20'
*
*
F601      DSTAT   EQU    DIADR+1
*        0-1     UNIT ADDRESS
*        2     UNIT SELECTED (LOW TRUE)
*        3     TRACK 0
*        4     WRITE PROTECT
*        5     DISK READY
*        6     PINTE
*        7     TRANSFER FLAG
*
*        FLAG BITS
*
0080      TFLG    EQU    X'80'
0040      INTE    EQU    X'40'
0020      RDY     EQU    X'20'
0010      WPT     EQU    X'10'
0008      TK0     EQU    X'08'
0004      USLT    EQU    X'04'
*
*
*        COMMAND REGISTER
*
F600      DCMND   EQU    DIADR
*(ALSO WILL RESPOND TO DISK+1)
*
*        0-1     COMMAND MODIFIER
*        5-7     COMMAND
*
*        COMMANDS
*
0020      SLUN    EQU    X'20'      SELECT UNIT
*        MODIFIER CONTAINS UNIT ADDRESS
0040      SINT    EQU    X'40'      SET INTERRUPT
*        MODIFIER =1 ENABLE INTERRUPT
*                  =0 DISABLE INTERRUPT
0060      STEP    EQU    X'60'      STEP CARRIAGE
*        MODIFIER =00 STEP OUT
*                  =01 STEP IN
0080      WTCMD   EQU    X'80'      ENABLE WRITE
*        NO MODIFIER USED
00A0      RESET   EQU    X'A0'      RESET CONTROLLER
*        NO MODIFIER USED
*
*

```

```

*          SCLEN EQU 134      SECTOR LENGTH/2
*
*          SELECT DRIVE SPECIFIED
*          BY UNIT ADDRESS IN DCB
*
05E4 D5      SLCT    PUSH D
05E5 C5      PUSH B
05E6 E5      PUSH H
05E7 2A0207   LHLD DADR   GET CONTROLLER ADR
05EA 3AF606   LDA DCBUN   GET UNIT ADR FROM
05ED E603    ANI X'03'   DCB
05EF 47      MOV B,A    AND SAVE
05F0 23      INX H     POINT TO STATUS
05F1 7E      MOV A,M    AND READ
05F2 4F      MOV C,A    SAVE STATUS
05F3 E607    ANI X'07'   MASK USLD & ADDR
05F5 A8      XRA B     DESIRED UNIT PREV
*
*          NOTE-THIS TEST WILL FAIL IF
*          CONTROLLER IS NOT PLUGGED IN
*
05F6 79      MOV A,C    SELECTED?
05F7 CA0C06   JZ  SL010  YES-CHECK RDY
05FA 78      MOV A,B    GET UNIT ADDRESS
05FB F620    ORI SLUN   BUILD COMMAND
05FD 77      MOV M,A   OUTPUT COMMAND
*
*          WAIT 250 MSEC FOR
05FE 11FA00   LXI D,250  SECTOR CNTR TO
0601 CD1706   CALL TIMER GET IN SYNC
0604 7E      MOV A,M    GET STATUS
0605 E607    ANI X'07'   SELECTED NOW?
0607 A8      XRA B
0608 7E      MOV A,M    GET STATUS AGAIN
0609 C21006   JNZ SL020  ERROR IF NOT SLTD
060C E620    SL010    ANI RDY  ENSURE UNIT IS
060E EE20    XRI RDY   READY
0610 E1      SL020    POP H
0611 C1      POP B
0612 D1      POP D
0613 C8      RZ       RETURN IF OK
*
*          DRIVE NOT UP ERROR
0614 C3D805   JMP DRIVER
*
*
*          1 MILLISECOND TIMER
*          DE=(DELAY) TIME IN MSEC
*
*          A IS DESTROYED
*
0617 C5      TIMER   PUSH B
0618 E5      PUSH H
0619 2A0207   LHLD DADR
061C 7E      MOV A,M    RE-TRIGGER 4
061D 0660    MVI B,96   SECOND TIMER
061F 78      TI010    MOV A,B    COUNT
0620 D601    SUI I     DELAY LOOP=1.008
0622 B7      ORA A     MSEC @500 NSEC

```

0623 C22006	JNZ	TI010+1	
*			
*	IMSEC EXPIRED - DECREMENT DELAY		
*	MULTIPLIER & CHECK FOR DONE		
*			
0626 1B	DCX	D	
0627 7B	MOV	A,E	
0628 B2	ORA	D	
0629 C21F06	JNZ	TI010	
062C E1	POP	H	
062D C1	POP	B	
062E C9	RET		
*			
*	WRITE 1 SECTOR		
*			
*			
062F CDE405 WSECT	CALL	SLCT	ENSURE UNIT SLD
0632 3AF706	LDA	DCBSC	AND READY
0635 47	MOV	B,A	
0636 C5	PUSH	B	
0637 0E86	MVI	C,SCLEN	C <- BYTCT/2
0639 2A0207	LHLD	DADR	GET CONTROLLER ADR
063C E5	PUSH	H	
063D 23	INX	H	READ STATUS
063E 7E	MOV	A,M	ABORT IF
063F E610	ANI	WPT	WRITE PROTECTED
0641 C2DE05	JNZ	PROTER	
0644 2A0007	LHLD	BUFAADR	GET BUFFER ADDR
0647 E5	PUSH	H	
0648 D1	POP	D	MOVE TO DE
0649 3AF806	LDA	DCETK	MOVE TRACK AND
064C 77	MOV	M,A	SECTOR ID TO WRITE
064D 23	INX	H	BUFFER
064E 70	MOV	M,B	
064F 2A0207	LHLD	DADR	GET CONTROLLER ADR
0652 CDE906	CALL	GETSEC	WAIT FOR SECTOR
*			
*	FOUND DESIRED SECTOR-		
*	ENABLE WRITE		
*			
0655 3680	MVI	M,WTCMD	
0657 23	INX	H	
*			
*	WAIT FOR TRANSFER FLAG		
*			
0658 B6	WS010	ORA	M
0659 F25806	JP	WS010	
*			
*	INSERT SYNC BYTE		
*			
065C 23	INX	H	
065D 36FF	MVI	M,X'FF'	
*			
065F AF	XRA	A	CLEAR CARRY
0660 EB	XCHG		
0661 0600	MVI	B,0	AND CHECKSUM
*			

* WRITE HEADER & DATA FIELD
 *
 0663 7E WS020 MOV A,M GET BYTE FROM MEM
 0664 12 STAX D WRITE TO DISK
 0665 88 ADC B ADD TO CKSUM
 0666 47 MOV B,A SAVE CKSUM
 0667 23 INX H NEXT BYTE
 0668 7E MOV A,M -ETC-
 0669 12 STAX D
 066A 88 ADC B
 066B 47 MOV B,A
 066C 23 INX H
 066D 0D DCR C
 066E C26306 JNZ WS020

* END OF DATA - INSERT CHECKSUM
 *
 0671 78 MOV A,B
 0672 12 STAX D

* WAIT END OF SECTOR
 *
 0673 E1 POP H
 0674 AF XRA A
 0675 B6 WS030 ORA M WAIT SCTR FLAG
 0676 F27506 JP WS030
 0679 110100 LXI D,1 WAIT 1 MSEC FOR
 067C CD1706 CALL TIMER ERASE DELAY
 067F C1 POP B
 0680 C9 RET

*
 * READ 1 SECTOR
 * VERIFY CHECKSUM AND HEADER
 *
 * RETURNS Z=OK
 * NZ=ERROR
 *

0681 CDE405 READAL CALL SLCT ENSURE UNIT IS
 * RDY + SLTD
 0684 3AF706 LDA DCBSC GET SECTOR ADDR
 0687 47 MOV B,A FROM DCB
 0688 C5 PUSH B
 0689 0E86 MVI C,SCLEN C <- BYTCT/2
 068B CDD606 CALL WTSYNC WAIT DESIRED
 * SECTOR & STRIP
 * SYNC BYTE

* FOUND DESIRED SECTOR - READ
 *
 068E EB XCHG
 068F 0600 MVI B,0 CLR CHECKSUM

*
 * READ LOOP
 *
 0691 1A RDA10 LDAX D READ FROM DISK
 0692 77 MOV M,A MOVE TO BUFFER

0693 23	INX H	NEXT LOC
0694 88	ADC B	ADD TO CHECKSUM
0695 47	MOV B,A	AND SAVE
0696 1A	LDAX D	NEXT READ
0697 77	MOV M,A	-ETC-
0698 23	INX H	
0699 88	ADC B	
069A 47	MOV B,A	
069B 0D	DCR C	END OF DATA?
069C C29106	JNZ RDA10	NO-LOOP
*		
* END OF DATA-READ CHECKSUM		
*		
069F 1A	LDAX D	
06A0 B8	RDA020 CMP B	COMPARE WITH
06A1 C1	POP B	COMPUTED CHECKSUM
06A2 C0	RNZ	RETURN IF ERROR
*		
* CHECKSUM OK-VERIFY HEADER		
*		
06A3 2A0007	LHLD BUFADR	POINT DE TO READ
06A6 EB	XCHG	BUFFER
06A7 CDBD05	CALL LDTRK	POINT TO CURRENT
06AA 1A	LDAX D	TRACK AND COMPARE
06AB BE	CMP M	WITH TRACK ID READ
06AC C0	RNZ	
06AD 13	INX D	
06AE 1A	LDAX D	COMPARE SECTOR ID
06AF B8	CMP B	WITH DESIRED SCTR
06B0 C9	RET	
*		
* VERIFY SECTOR		
*		
* READ THROUGH SECTOR WITHOUT		
* MOVING DATA INTO MEMORY AND		
* VERIFY TRACK AND SECTOR ID		
* AND CHECKSUM		
*		
* ONLY TRACK AND SECTOR ID ARE READ		
* INTO MEMORY AND CHECKSUM IS		
* VERIFIED		
*		
* SECTOR IS SPECIFIED BY B REG		
*		
* RETURNS Z=OK		
* NZ=ERROR		
*		
06B1 C5	READCK PUSH B	SAVE SECTOR
06B2 CDE405	CALL SLCT	ENSURE SLTD&RDY
06B5 0E85	MVI C,SCLEN-1	C <- BYTCT/2-1
06B7 CDD606	CALL WTSYNC	WAIT SECTOR & STRP
* OFF SYNC BYTE		
06BA 0600	MVI B,0	CLR CHECKSUM
06BC 7E	MOV A,M	READ TRACK ID
06BD 12	STAX D	SAVE IN BUFFR
06BE 88	ADC B	ADD TO CHECKSUM
06BF 47	MOV B,A	AND SAVE

06C0 13	INX D	
06C1 7E	MOV A,M	READ SCTR ID
06C2 12	STAX D	AND SAVE
06C3 88	ADC B	
06C4 47	MOV B,A	
06C5 00	NOP	
*		
* READ THROUGH REMAINDER OF SECTOR		
* TO COMPUTE & VERIFY CHECKSUM		
*		
06C6 7E	RDCK10	MOV A,M READ FROM DISK
06C7 88		ADC B ADD TO CHECKSUM
06C8 47		MOV B,A SAVE CKSUM
06C9 00		NOP
06CA 00		NOP
06CB 7E		MOV A,M -ETC-
06CC 88		ADC B
06CD 47		MOV B,A
06CE 0D		DCR C
06CF C2C606		JNZ RDCK10
*		
* END OF DATA - READ CHECKSUM		
*		
06D2 7E		MOV A,M
06D3 C3A006		JMP RDA020 GO CHECK HDR & CHECKSUM
*		
*		
* WAIT FOR DESIRED SECTOR		
* TO COME AROUND AND STRIP OFF		
* SYNC BYTE FOR READ ROUTINES		
*		
06D6 2A0007	WTSYNC	LHLD BUFADR GET BUFFER ADDRESS
06D9 EB		XCHG
06DA 2A0207		LHLD DADR AND CONTROLLER ADR
06DD CDE906		CALL GETSEC WAIT FOR SECTOR
06E0 23		INX H
06E1 B6	WTS010	ORA M WAIT FOR XFER RDY
06E2 F2E106		JP WTS010 FLAG
06E5 23		INX H OK-READ IN SYNC
06E6 7E		MOV A,M BYTE -- THROW IT
06E7 AF		XRA A AWAY, CLEAR CARRY
06E8 C9		RET AND GO READ
*		
* WAIT FOR DESIRED SECTOR TO COME		
* AROUND		
*		
06E9 7E	GETSEC	MOV A,M WAIT FOR SCTR FLAG
06EA B7		ORA A
06EB F2E906		JP GETSEC
06EE E60F		ANI X'0F'
06F0 A8		XRA B OK - IS THIS THE
06F1 C2E906		JNZ GETSEC ONE WE WANT?
06F4 C9		RET NO-WAIT
PRESS ON		
*		
* RAM STORAGE REQUIRED FOR DRIVER		
*		

```

*
*      INTERNAL DISK CONTROL BLOCK
*
06F5      DCB     EQU    *
06F5      DCBFN   DS     1
06F6      DCBUN   DS     1
06F7      DCBSC   DS     1
06F8      DCBTM   DS     1
06F9      DCBAD   DS     2
0006      DCBLEN  EQU    *-DCB
*
*
0080      HCI     EQU    X'80'      HEADER CHECK INH
0040      RAFI    EQU    X'40'      RAW CHECK INHIBIT
06FB 4C    TRKMX   DC     76       MOD 2
*
*
*      CURRENT TRACK TABLE
*      MUST BE INITIALIZED TO FF
*      AT POWER ON TO CAUSE DISK TO
*      BE RESTORED TO TRACK 0
*      THE FIRST TIME IT IS ACCESSED TO
*      CALIBRATE TRACK POSITION
*
06FC FF    TRACK   DC     X'FF'
06FD FF    DC     X'FF'
06FE FF    DC     X'FF'
06FF FF    DC     X'FF'
*
*
0700      BUFADR DS     2       CURRENT BUFFER ADR
*
*
0702 00F6  DADR    DC     B(DIADR)  DISK CTLR ADDR
*
*      RETRY COUNTERS
0704      L1RTRY  DS     1
0705      L2RTRY  DS     1
0706      L3RTRY  DS     1
*
0707      DIRCTN  DS     1
0708      STACK   DS     2       SAVED SP
*
*
*
070A      END    ***
```

APPENDIX A - BASIC ERROR MESSAGES

ARGUMENT - Argument in a function reference is the wrong data type or missing.

ARRAY INDEXING ERROR - A reference to an array element contains an invalid index. May also be caused if an attempt is made to reference an array element before the array is defined in a DIM statement.

CONVERSION ERROR - Attempt to assign a real value to an integer variable and the converted value is too large.

DIGIT BEYOND RADIX - A number specified in radix format includes a digit which is invalid for the specified radix.

DISK FULL - An attempt was made to allocate another track for a file and no free tracks remain.

DRIVE NOT UP - The desired disk unit does not have a diskette loaded, is not up to speed, or has a malfunction which prevents it from accepting commands.

DUPLICATE NAME - An attempt was made to OPEN a file name which already exists as a new file.

END-FILE - The end-of-file was encountered in a disk file read.

EXTRA INPUT IGNORED - The response to an INPUT statement contained more values than were needed to satisfy the variable list and the extra values were ignored.

FILE ALREADY OPEN - File number specified in an OPEN statement already has a file opened to it.

FILE NOT FOUND - File name specified in a disk I/O command does not exist on the specified diskette.

FILE NOT OPEN - File number specified in a disk I/O statement does not have a file name opened to it.

FILE TYPE ERROR - The attributes of the referenced file are inconsistent with the requirements of the statement or command that referenced it.

ILLEGAL IMMEDIATE - An attempt was made to use a statement as a direct command, but the statement is only valid within a BASIC program.

INPUT OVERFLOW - A program line greater than 250 characters in length was entered - the entire program line is cancelled.

INSUFFICIENT INPUT - The response to an INPUT statement contained insufficient values to satisfy the variable list.

INTERRUPT - Execution of a program was interrupted by entry of a CNTL/C key at the terminal.

INVALID DISK FILE NAME - Disk file name specified is not a valid disk file name.

LOAD OVERRUN - The length of the BASIC program being loaded exceeds the memory space currently available to BASIC.

LOG OF NEG # - Attempt was made to pass a negative or zero value to the LOG or LN function.

MEMORY OVERFLOW - Insufficient memory exists for execution of the program.

MISSING FOR - A NEXT statement was encountered prior to execution of a FOR statement specifying the loop variable.

NOT A FILE # - File number specified in a disk I/O statement is not one of the digits 0 - 9.

NOT A LOAD FILE - Attempt to load a data format disk file.

NOT A RECORD # - The value following the RECORD option in a GET or PUT statement is not a valid record number.

NOTHING TO RETURN TO - A RETURN statement was encountered prior to executing a GOSUB statement.

NUMBER OUT OF RANGE - The value of an expression referenced is illegal.
Refer to the description of the statement in error for the range of valid values.

OVERFLOW - Numeric overflow - Result of an operation is too large to be contained in a variable.

OUTPUT OVERFLOW - A PRINT or PUT statement has attempted to create an output line (record) greater than 250 characters in length. This exceeds the maximum internal buffer capacity. The line (record) is not output.

PARM ERR - Disk I/O Parameter error - usually caused by setting the sequential GET/PUT pointers to an invalid value.

PERM FILE - An attempt was made to SCRATCH a permanent file.

PERM I/O ERROR - A disk I/O error occurred which was not recoverable in the disk I/O retry logic.

PRECISION ERROR - A numeric function or the ↑ operator was referenced with RSIZE greater than 10.

READY - The BASIC interpreter is ready for entry of commands or program lines at the terminal.

RAN OUT OF DATA - A READ statement depleted the data list before satisfying the variable list. A GET statement encountered the end of the current record without satisfying the variable list.

SIZES ERROR - One of the parameters of a SIZES statement is invalid or there are already variables allocated when the statement is encountered.

SQRT OF NEG # - Attempt to pass a negative number to the SQR function.

STACK OVERFLOW - The statement in error contains an expression which is too complex. Break the expression into multiple expressions which are less complex.

STMT # NOT FOUND - The statement in error tried to transfer control to a program line number which does not exist.

SYNTAX - The statement in error is not recognizable or contains an invalid structure such as unequal right and left parentheses.

TYPE ERROR - Attempt to assign a value of the wrong data type to a variable.

WRITE PROTECT - An attempt was made to write on a file with a write protect attribute or the diskette on which the file resides has a write protect tab installed.

UNDERFLOW - Numeric underflow - The result of an operation is too small to be assigned to a variable.

X^Y INDETERMINATE - Attempt to take a fractional power of a negative number or Ø or to raise Ø to a negative or Ø power, which are undefined operations.

ZERO DIVIDE - Attempt to divide by zero which is an undefined operation.

APPENDIX B - BASIC UTILITY PROGRAMS

The PDS MASTER diskette included with each Micropolis disk subsystem contains a BASIC UTILITY program which provides the following functions:

B.1 FORMAT A DISKETTE

A blank diskette must be initialized (formatted) before it can be used with the Micropolis Disk Extended BASIC. Initialization consists of writing track and sector address information in each sector of the data area of the diskette and writing an empty Directory on the Directory track. Once initialized, a diskette may be used as a data diskette, or it may be configured as a system diskette by saving BASIC on it.

Diskettes may be initialized by the following procedure. Read this procedure through completely and carefully before attempting to initialize a diskette. Follow the procedure exactly.

- 1) With BASIC in the computer and running, insert a MASTER diskette or a system diskette with the program "UTILITY" previously saved on it into drive Ø and load the diskette by depressing the actuator.
- 2) Enter the command LOAD "UTILITY" ↴. When the system responds with READY, enter RUN ↴. (↳ denotes Carriage Return.) The Utility program will output its sign-on message and prompt for a function selection as follows:

DISK UTILITY PGM REV 4.X
ENTER KEY TO SELECT DESIRED FUNCTION

F FORMAT DISK
M MEM EXAM/MODIFY
S SAVE BASIC
E EXIT

- 3) Enter F ↴. The Utility program will output the message:

SPECIFY UNIT NUMBER?

- 4) Type the number of the disk unit (Ø-3) that is to be used and press return. The UTILITY program will output the message:

INSERT BLANK DISKETTE IN UNIT X
ARE YOU READY?

Load the diskette you wish to format into the specified unit.

- 5) Enter Y ↴ (for Yes). The Utility program will initialize the diskette in approximately 7Ø seconds and then output:

FUNCTION ?

- 6) At this point, the initialized diskette is ready to be used as a data diskette. If you wish to create a system diskette, enter S ↓. The Utility program will output:

ARE YOU READY ?

Enter Y ↓. A copy of BASIC will then be written on the diskette, in approximately 60 seconds, and then the Utility program will output:

FUNCTION ?

- 7) If you wish to initialize more diskettes, repeat this procedure from Step 3.

B.2 MEMORY EXAMINE/MODIFY

This function provides the means of examining or altering the contents of RAM memory. To examine or modify memory, respond to the FUNCTION ? prompt with M ↓.

The Utility will output:

ENTER ADDRESS ?

Type the hexadecimal representation of the desired memory address followed by a carriage return. The Utility will print a carriage return linefeed, the hex address and the hexadecimal value of the contents of the desired memory location, followed by a question mark (?). Enter one of the following responses:

- 1) If a hexadecimal number from 0 - FF followed by a carriage return is entered, the contents of the memory location just displayed are set to the value entered. The address and contents of the next sequential memory location are then displayed and the Utility prompts for the next response.
- 2) If a carriage return is entered, the address and contents of the next sequential memory location are displayed and the Utility prompts for the next response.
- 3) If a colon (:) followed by a carriage return is entered, the Utility prompts for the entry of a new address to display/modify as described above.
- 4) If an exclamation mark (!) followed by a carriage return is entered, the Utility exits the memory modify/display function and prompts for a new function select.

B.3 SAVE BASIC

This function writes a copy of the BASIC system software currently resident in memory onto a diskette. This function may only be used in conjunction with the disk initialization procedure for creating BASIC system diskettes.

B.4 EXIT

Enter E↓ to exit the utility program.

CAUTION: Each version of BASIC requires its own version of the utility program. Attempting to run utility with the wrong version of BASIC may result in catastrophic errors.

APPENDIX C - ACCESSING DISKCOPY FROM BASIC

DISKCOPY is a special overlay utility that writes an absolute binary copy of one disk onto another. The utility overlays MDOS or BASIC. It uses all available memory during the copying process. The more memory in a system the faster the copying process. On average it takes about two minutes to copy and verify all 315k bytes of a MOD II disk.

NOTE 1: Previous versions of DISKCOPY will not run with BASIC 3.0 and DISKCOPY 3.0 will not run with earlier versions of Micropolis BASIC.

NOTE 2: In multiple drive systems DISKCOPY can be copied onto another disk by using the FILECOPY utility under MDOS (Section 4.7).

The DISKCOPY utility is invoked from BASIC by using the LINK command.

LINK "[unit:]DISKCOPY"

a sign-on message is output:

MICROPOLIS DISKCOPY VS X.X - COPYRIGHT 1978
SPECIFY UNIT # FOR ORIGINAL (SOURCE) DISKETTE
?

DISKCOPY waits until the unit number is entered. When a number between 0 and 3 is entered it prompts:

SPECIFY UNIT # FOR DESTINATION DISKETTE
?

and waits until the unit number (0 to 3) is entered. It then prompts:

PUT DISKETTES IN SPECIFIED UNITS
TYPE Y WHEN READY
?

and waits for a Y. A note of CAUTION, we strongly recommend placing a write protect tab on the original (source) diskette. It is possible to put the wrong diskette in the wrong drive or type the wrong unit numbers. If your original does not have a write protect tab and you make an error, the original can be overwritten. The write protect tab provides a physical interlock which disables the write electronics.

When a Y is typed DISKCOPY will start the copying process. During copying, the process can be temporarily halted between read source and write destination cycles by typing a control S. The process is restarted by typing any other key except a control C.

The control C will cancel the entry or copy process and prompt:

CANCELLED
MORE ?

If a Y is typed DISKCOPY starts from the top asking for the unit numbers again. If an N is typed DISKCOPY prompts:

PUT SYSTEM DISKETTE IN UNIT 0
TYPE Y WHEN READY
?

When a Y is typed the disk in unit 0 is rebooted. If it's an MDOS diskette MDOS is booted. If the disk in unit 0 is a BASIC only disk or some other bootable system, it will be booted in and sign on. DISKCOPY is overlayed by the incoming system and is no longer in memory.

When the disk has been copied and verified correctly DISKCOPY outputs:

GOOD COPY
MORE ?

If the copy cannot be completed or does not verify correctly DISKCOPY outputs:

PERM I/O ERROR ON DESTINATION DISKETTE

or

PERM I/O ERROR ON SOURCE DISKETTE

indicating where the error occurred.

It is possible for single drive systems to make use of the DISKCOPY utility to copy from one disk to another. In this case it is imperative that the original diskette be write protected with a write protect tab. The procedure involves specifying the same unit number for both source and destination disks. Immediately after typing a Y in response to the TYPE Y WHEN READY prompt, type a control S. The DISKCOPY program will read as many tracks from the source disk as can be contained in main memory and then pause. When the select indicator light goes out, remove the source diskette and insert the destination diskette. Press the return key and as soon as the select indicator light comes on type a control S again. When the select indicator light goes out again the data from the source disk has been written to the destination disk and one complete cycle is finished. This process is repeated, swapping the source and destination disks in and out until the entire disk is copied. After the last data is written onto the destination disk, the program goes directly into a verifying process and will not pause until this is over. When the source is placed back into the drive and the return key is pressed the system will prompt: GOOD COPY or output an error message as discussed above. At this point the copy is complete.

APPENDIX D - SUMMARY OF MDOS ERROR MESSAGES

D.1 MDOS EXECUTIVE AND SHARED SUBROUTINES

BAD FILE #

The file number specified is greater than 8.

BAD RECORD #

The record number specified is greater than exists in the specified file.

CANCELLED

A control C was typed at the console, canceling an operation.

COMMAND NOT FOUND

The word typed as a command name, or implicit command (file name) does not exist. The command was spelled incorrectly or the file name was not found on the specified disk.

DISK FULL

An attempt was made to allocate an additional track to a file, and no free tracks exist. The file is closed and the message is output. Some data may have been successfully written to the file before additional track space was needed.

DRIVE NOT UP

The disk unit specified is not loaded.

DUPLICATE NAME

The file name already exists on the unit specified. All files on a disk must have unique names.

END-FILE

The end of the file has been reached during a disk read.

FILE NOT FOUND

The file name specified does not exist on the unit specified.

FILE NOT OPEN

The file with the specified number has not been opened.

INDEX PAST EOR

The index position is beyond the end of the record.

LOAD ADDRESS ERROR

The address specified with a file to be loaded into memory would cause the file to overwrite the operating system.

PARM ERR

A parameter is out of range for a particular command, too big or too small. This is different than a syntax error caused by a parameter beyond the maximum input range.

PERM FILE

The file specified with a SCRATCH command or with the @SCRATCH subroutine has an attribute with bit 1 set high indicating a permanent file.

PERM I/O ERR

A disk I/O error occurred which was not recoverable by the disk I/O retry logic.

READ ONLY FILE

The specified file has an attribute with bit 0 set high. This inhibits rewriting of the file.

SYNTAX ERROR

The syntax of a command is wrong. This may be due to incorrect spelling, or parameters beyond the maximum input ranges; 10 characters for ASCII and four hex digits for numeric.

SYSTEM VERSION ERROR

An attempt was made to run a system program on the wrong version of the system.

WRITE PROTECT

The unit specified with a SAVE command or a subroutine that writes to the disk has a disk in it with a write protect tab in place.

WRONG FILE TYPE

The file type does not correspond to the type of operation that is to be performed.

D.2 EDITOR

FILEBUFFER OVERFLOW

This message occurs whenever there is less than 256 bytes of buffer space remaining in the edit buffer. Input can continue until the buffer is completely full, but the message will be repeated after each carriage return. The file should be written to disk and a new file started. If a file is loaded from disk and is too large to reside in the buffer, this message is output and the load is aborted. No data is loaded. This is most likely to occur in conjunction with the APPEND command. If an APPEND causes an overflow, it is aborted and the files that were in the buffer prior to the command are not changed.

FILE ON DISK NOT UPDATED, PROCEED?

The current working file in the editor buffer has not been saved or resaved to disk. If you want to continue without updating the disk then type a Y in response, otherwise type an N.

FILE NOT NAMED

A name has not been given to the current editor file prior to trying to save it onto a disk.

LINE NOT FOUND

A line number which does not exist in the current text file was specified in an EDIT command.

LINE NUMBER OVERFLOW

The editor command RENUM specified an increment that caused the line number to exceed 9999 decimal. The file is only partially renumbered and care should be taken to do an additional RENUM with a smaller increment to assure that the file is properly numbered prior to doing any editing on the file.

STRING NOT FOUND

The SEARCH MASK specified with a SEARCH or CHANGE command in the editor does not exist in the text.

D.3 ASSEMBLER

A

ARGUMENT ERRORS are flagged with a capital A. They are caused when the operand field contains an invalid character or a three byte opcode has a ASCII literal which is out of range. In the later case the value is truncated to the left. The error is flagged during pass two of the assembly.

D

DUPLICATE LABEL ERRORS are flagged with a capital D. They are the result of the same symbolic name being used more than once as a label. The error is flagged during pass one. The assembler uses the value of the first label during the assembly.

L

LABEL ERRORS are flagged with a capital L. They are caused by labels containing illegal characters. Refer to the section on symbolic names. The error is flagged during pass two of the assembly.

M

MISSING LABEL ERRORS are flagged with a capital M. They are caused when a label is missing from a pseudo-op that requires a label. Only two pseudo-ops require labels. They are the EQU and the INP pseudo-ops. The error is flagged during pass one of the assembly.

O

OPCODE ERRORS are flagged with a capital O. They are caused by illegal or missing opcodes. The error is flagged during pass two of the assembly.

R

REGISTER ERRORS are flagged with a capital R. They are caused when a value greater than 7 or less than 0 is used in the operand file where a register value should occur. The error is flagged during pass two of the assembly.

S

SYNTAX ERRORS are flagged with a capital S. They are caused by missing operands, or improper use of operators. The error is flagged during pass two of the assembly.

U

UNDEFINED SYMBOL ERRORS are flagged with a capital U. They are caused when a symbolic name that has never been defined as a label is used as an operand. Or, the label is used as a forward reference in a DS, EQU, ORG, or INP statement. When the error is the result of a forward reference, it is flagged during pass one of the assembly. Otherwise it is flagged during pass two.

V

VALUE ERRORS are flagged with a capital V. They are caused when the operand of a two byte opcode, or a DB, is beyond the range 0 to FF hex (one byte). The assembler truncates the expression to the left and uses the least significant byte. The error is flagged during pass two of the assembly.

APPENDIX E - SYSTEM I/O LISTINGS

Supplied in this appendix are the assembly listings of the I/O routines and the configuration program.

The I/O routines are broken into three Sections E1, E2 and E3.

Section E1 is the logical I/O routine for the console and list streams. These routines should not normally have to be changed as they are tailored to support BASIC and MDOS system requirements.

Section E2 is the console physical I/O handler which is modified as necessary during the configuration process described in Chapter 2.

Section E3 is the printer physical output handler which must be modified or rewritten as described in Chapter 2.

Section E4 is the configuration logic which is provided for information only and should not be changed.

Section E5 contains the configuration tables which may in some cases have to be changed as described in Chapter 2.

E.1 LOGICAL I/O ROUTINES FOR PDS 4.0

04B8 04 0A	GEIADDR	DW	@GEIADDR		
04BA 04BA	GSYMTABLOC	EQU	\$;USED IN SYMSAVE AND ASSM	
04BA 00 00	ARG0	DW	0		
04BC 00 00	ARG1	DW	0		
04BE 00 00	ARG2	DW	0		
04C0 00 00	ARG3	DW	0		
04C2 00 00	ARG4	DW	0		
04C4 00	NARGS	DB	0		
04C5 05	RSIZE	DB	5		
04C6 03	ISIZE	DB	3		
04C7 28	SSIZE	DB	28H		
04C8 00	GFORMFLAG	DB	0	;Z=LINEFEEDS NZ=FORMFEEDS	
04C9 40 00	GIDWORD	DW	40H	;CURRENTLY ONLY FIRST BYTE	
04CB 02 00	*	FILL	2,0	;ZEROS FILL	
04CD	*	IFT	APPENDIXE		
077B	*	ENDIF			
077B	*				
077B	*				
077B	*				
077B	*				
077B	*				
077B	000D	CR	EQU	0DH	
077B	0218	CNTX	EQU	18H	;CONTROL X
077B	0008	BS	EQU	8	
077B	000A	LF	EQU	10	
077B	005F	BACKARROW	EQU	5FH	
077B	007F	RUBOUT	EQU	7FH	
077B	0003	CNTC	EQU	'C'-64	
077B	0013	CNTS	EQU	'S'-64	
077B	0010	CANCELLED	EQU	16	
077B	*				
077B	*				
077B	*				
077B	0003	TISTAT	EQU	3	
077B	0002	TDIN	EQU	2	
077B	0003	TOSTAT	EQU	TISTAT	
077B	0002	TDOUT	EQU	TDIN	

```

077B      *
077B      * CONSOLE READY FLAG AND MASK ASSIGNMENTS
077B      *
077B 0002 DIFLG      EQU 2
077B 0002 MSK1       EQU DIFLG
077B 0001 DOFLG      EQU 1
077B 0001 MSK2       EQU DOFLG
077B      *
077B      * LIST DEVICE PORT ASSIGNMENTS
077B      *
077B 0005 PTSTS      EQU 5
077B 0005 PTCTL      EQU PTSTS
077B 0004 PTDAT      EQU 4
077B      *
077B      * LIST READY FLAG AND MASK ASSIGNMENTS
077B      *
077B 0080 PMSK1      EQU 80H
077B 0080 PMSK2      EQU PMSK1
077B 0001 PMSK3      EQU 1
077B 0001 PMSK4      EQU PMSK3
077B      *
077B      * VECTORS TO IO TABLES AT @CONSOLEADDR AND @LISTADDR
077B      *
077B          ORG  @CONSOLEADDR
04EC F0 04          DW   @CIOTABLE ; CONSOLE IO TABLE
04EE 02 05          DW   @LIOTABLE ; LIST IO TABLE
04F0      *
04F0      * CONSOLE DEVICE IO TABLE--VECTORS TO IO HANDLER RTNS
04F0      *
04F0          ORG  @CIOTABLE
04F0 14 05          DW   CIN
04F2 2A 05          DW   COUT
04F4 39 05          DW   CBRK
04F6 1B 06          DW   CDIN
04F8 3B 06          DW   CDOU
04FA 54 06          DW   CDBRK
04FC 6B 06          DW   CDINIT
04FE 00          WRAPFLAG  DB   0
04FF 03          DB   3 ;NULL COUNT
0500 3F          WIDTH    DB   3FH
0501 01          CURSOR   DB   1

```

```

0502 *
0502 * LIST DEVICE IO TABLE--VECTORS TO IO HANDLER RTNS
0502 *
0502 ORG QLIOTABLE
0502 DW 0
0504 64 05 DW LOUT
0506 74 05 DW LATN
0508 00 00 DW 0
050A 3B 06 DW CDOUT
050C E8 06 DW LDATN
050E FE 06 DW LDINIT
0510 00 PWRAPFLAG DB 0
0511 03 DB 3 ;NULL COUNT
0512 48 PWIDTH DB 48H
0513 01 PCURSOR DB 1
0514 *
0514 * CONSOLE LOGICAL INPUT ROUTINE
0514 * STRIPS PARITY FROM INPUT BYTE LEAVING 7 BIT ASCII
0514 * CHANGES BACKARROW AND RUBCUT INTO BACKSPACE
0514 * AND IF A CONTROL C RETURNS THE ZERO FLAG SET
0514 * CARRY FLAG ALWAYS RETURNED CLEAR (NC)
0514 *
0514 CD 8D 07 CIN CALL QCDIN ;GET RAW CHR
0517 78 MOV A,B
0518 E6 7F ANI 7FH ;STRIP PARITY
051A 47 MOV B,A
051B FE 03 CPI CNTC
051D C8 RZ
051E FE 5F CPI BACKARROW ;IF BK ARROW
0520 CA 26 05 JZ BSPC ;TURN INTO CNT H
0523 EE 7F XRI RUBOUT
0525 C0 RNZ
0526 06 08 BSPC MVI B,BS
0528 3C INR A ;FORCE TO NOT ZERO
0529 C9 RET
052A *
052A * CONSOLE LOGICAL OUTPUT ROUTINE
052A *

```

052A 2A FE 04	COUT	LHLD	WRAPFLAG	
052D EB		XCHG		; D=NULLS, E=WRAP
052E 2A 00 05		LHLD	WIDTH	; H=CURSOR, L=WIDTH
0531 AF		XRA	A	; O FLAG FOR DEVOUT
0532 CD 7C 05		CALL	DEVOUT	
0535 22 00 05		SHLD	WIDTH	; UPDATE CURSOR
0538 C9		RET		
0539 *				
0539 *	* CHECK CONSOLE READY IF A KEY HAS BEEN PRESSED			
0539 *	* IF NOT RETURN IMMEDIATELY.			
0539 *	* IF A KEY PRESSED GET IT AND IF A CONTROL S WAIT			
0539 *	* UNTIL SOME OTHER KEY IS PRESSED.			
0539 *	* IF A CONTROL C, THEN PUT ERROR CODE IN A REG.			
0539 *	* AND RETURN WITH THE ZERO FLAG SET (Z).			
0539 *	* ANY OTHER CHARACTER RETURN WITH THE ZERO FLAG			
0539 *	* CLEAR (NZ).			
0539 *				
0539 CD 97 07	CBRK	CALL	@CDBRK	
053C C0		RNZ		
053D 78		MOV	A,B	
053E E6 7F		ANI	7FH	; STRIP PARITY
0540 FE 13		CPI	CNTS	; PAUSE
0542 C2 50 05		JNZ	CANC	; IS IT A CNTC
0545 CD 8D 07	PAUSE	CALL	@CDIN	; WAIT FOR INPUT
0548 78		MOV	A,B	; GET CHARACTER
0549 E6 7F		ANI	RUBCUT	; STRIP PARITY
054B FE 13		CPI	CNTS	; IS IT A CNTS
054D CA 45 05		JZ	PAUSE	; YES LOOP
0550 FE 03	CANC	CPI	CNTC	; IS IT A CNTC
0552 3E 10		MVI	A,CANCELLED	; ERROR CODE
0554 C9		RET		
0555 0F 00		FILL	15,0	
0564 *				
0564 *	* LIST DEVICE LOGICAL OUTPUT ROUTINE			
0564 *				
0564 2A 10 05	LOUT	LHLD	PWRAPFLAG	
0567 EB		XCHG		; D=NULLS E=WRAP
0568 2A 12 05		LHLD	PWIDTH	; H=CURSOR, L=WIDTH
056B 3E 01		MVI	A,1	; FLAG FOR DEVOUT
056D CD 7C 05		CALL	DEVOUT	
0570 22 12 05		SHLD	PWIDTH	; UPDATE PCURSOR
0573 C9		RET		

```

0574      *
0574      * LOGICAL LIST ATTENTION CHECK
0574      *
0574 CD EA 07 LATN      CALL    @LDATN      ; PRNT ATTN
0577 D0          RNC
0578 C3 D1 05   JMP     ATT
057B      *
057B      * COMMON DEVOUT ROUTINE FOR COUT AND LOUT
057B      * LF IS OUTPUT WITHOUT ANY CHANGE IN THE COLM POSITION
057B      * CR IS CHANGED TO CR + NULLS
057B      * BS IS CHANGED TO BACKAPROW
057B      * CTLX IS CHANGED TO \ LF CR NULLS
057B      * WRAP, IF REQUIRED, OCCURS WHEN THE WIDTH+1 CHARACTER
057B      * IS PRESENTED FOR OUTPUT
057B      *
057B      * DEVOUT EXPECTS A=0 FOR COUT STREAM OR A=1 FOR LOUT STREAM
057B      * D=NULLS PARAMETER FOR SPECIFIED STREAM
057B      * E=0 FOR WRAP ENABLED OR E=1 FOR DISABLED
057B      * L=WIDTH PARAMETER OF SPECIFIED STREAM
057B      * H=LAST COLM PRINTED ON SPECIFIED STREAM
057B      * B=CHARACTER TO BE OUTPUT
057B      *
057B      * DEVOUT PRESERVES D, E, H
057B      *
057B      * DEVOUT RETURNS L=NEW LAST COLM OR L=0 IF CR WAS LAST OUT
057B      *
057B      * DEVOUT RETURNS CARRY CLEAR (NC) IF OUTPUT WAS SUCCESSFUL
057B      * IF A PRINTER ATTENTION OCCURS IT RETURNS CARRY SET (C)
057B      * AND THE DEVICE ASSIGNMENTS ARE FORCED TO 1,1 AND 2,2.
057B      *
057B 00      DFLAG      DB      0      ;0 FOR COUT, 1 FOR LOUT
057C      *
057C 32 7B 05  DEVOUT      STA     DFLAG      ;SAVE WHICH STREAM FLAG
057F 48          MOV     C,B      ;SAVE MAIN CHAR IN C
0580 78          MOV     A,B
0581 FE 0D          CPI     CR      ;IF CR THEN
0583 CA BE 05          JZ      DEV032     ; OUTPUT CR + NULLS & COLM=0
0586 FE 0A          CPI     LF      ;IF LF THEN
0588 CA CC 05          JZ      DEV050     ; OUTPUT LF & COLM UNCHANGED

```

058B FE 18		CPI	CNTX	;IF CONTROL X THEN
058D C2 99 05		JNZ	DEV010	; SUBSTITUTE BACKSLASH
0590 0E 5C		MVI	C,\'	; WRAP IF REQUIRED, OUTPUT \
0592 CD A1 05		CALL	DEV020	; ATTENTION ERROR EXIT
0595 D8		RC		; OUTPUT LF,CR+NULLS,COLM=0
0596 C3 B6 05		JMP	DEV030	;IF BACKSPACE THEN
0599 FE 08	DEV010	CPI	BS	; SUBSTITUTE BACKARROW
059B C2 A1 05		JNZ	DEV020	;PATCH PLACEHOLDER
059E 0E 5F		MVI	C,BACKARROW	; TEST WRAP ENABLED
05A0 00		NOP		;IF NOT, JUST OUTPUT CHAR
05A1 7B	DEV020	MOV	A,E	;TEST COLM=WIDTH
05A2 B7		ORA	A	;IF SC DC LF CR NULLS
05A3 C2 CC 05		JNZ	DEV050	;ATTENTION ERROR EXIT
05A6 7C		MOV	A,H	;OUTPUT IT
05A7 BD		CMP	L	;ATTENTION ERROR EXIT
05A8 C2 AF 05		JNZ	DEV025	;INCREMENT COLM
05AB CD B6 05		CALL	DEV030	;ENSURE CARRY CLEAR
05AE D8		RC		;NORMAL EXIT
05AF CD CC 05	DEV025	CALL	DEV050	
05B2 D8		RC		
05B3 24		INR	H	
05B4 B7		ORA	A	
05B5 C9		RET		
05B6 *				
05B6 06 0A	DEV030	MVI	B,LF	
05B8 CD CD 05		CALL	DEV055	;OUTPUT LF
05BB D8		RC		;ATTENTION ERROR EXIT
05BC 06 0D		MVI	B,CR	;SET CR FOR OUTPUT
05BE 5A	DEV032	MOV	E,D	;SET NULL COUNTER
05BF CD CD 05	DEV035	CALL	DEV055	;OUTPUT SET CHAR
05C2 D8		RC		;ATTENTION ERROR EXIT
05C3 06 00		MVI	B,0	;SET NULL CHAR FOR OUTPUT
05C5 1D		DCR	E	
05C6 C2 BF 05		JNZ	DEV035	;LOOP UNTIL NULLS COMPLETE
05C9 AF		XRA	A	;ENSURE CARRY CLEAR
05CA 67		MOV	H,A	;AND FORCE COLM=0
05CB C9		RET		

05CC	*		
05CC 41	DEV050	MOV B,C	;RECOVER MAIN OUTPUT CHAR
05CD CD DA 05	DEV055	CALL DEV062	;OUTPUT CHAR FROM B
05I0 D0		RNC	;SUCCESSFUL EXIT
05D1 F5	ATT	PUSH H	;SAVE WIDTH
05D2 21 01 02		LXI H,201H	;FORCE ASSIGNMENT STATE
05D5 22 EA 04		SHLD @D1PORT	;PRESERVING CARRY
05D8 E1		POP H	;RESTORE WIDTH
05D9 C9		RET	;ATTENTION ERROR RETURN
05IA	*		
05DA 3A 7B 05	DEV060	LDA DFLAG	;0 FOR COUT
05DD B7		ORA A	;OR
05DE CA 92 07		JZ @CDOUT	
05E1 C3 E5 07		JMP @LDOUT	;1 FOR LOUT
05E4 37 00		FILL @PCON-\$,0	

E.2 CONSOLE PHYSICAL I/O HANDLERS

061B	*		
061B	*		
061B	* PHYSICAL DRIVERS START HERE		
061B	*		
061B	* FIRST THE CONSOLE DRIVERS		
061B	*		
061B		ORG @PCON	
061B DB 03	CDIN	IN TISTAT	;GET STATUS
061D 00		NOP	
061F E6 02		ANI DIFLG	
0620 FF 02		XRI MSK1	
0622 CA 2C 06		JZ IN010	;IF READY GET CHR

```

0625      *
0625      *SPACE HERE FOR NON-STANDARD BREAK CHK
0625      *PUT CALL INPALCE OF NO OPS
0625      *
0625 00          NOP
0626 00          NOP
0627 00          NOP
0628 C2 1B 06    JNZ   CDIN      ;NOT READY SO WAIT
062B C9          RET
062C      *
062C DB 02        IN010     IN     TDIN      ;GET CHR FROM DATA PORT
062E 00          NOP
062F 47          MOV    B,A       ;CHR INTO B
0630 C9          RET
0631 0A 00        FILL   10,0    ;DONE
063B      *
063B DB 23        CDO1T     IN     TOSTAT    ;OUTPUT STAT READY PORT
063D 00          NCP
063E E6 01        ANI    DOFLG     ;READY FLAG
0640 EE 01        XRI    MSK2
0642 C2 3B 06    JNZ   CDO1T    ;LOOP
0645 78          MOV    A,B       ;INTO A FOR OUT
0646 D3 02        OUT020     OUT    TDOUT    ;OUTPUT IT
0648 00          NOP
0649 C9          RET
064A 0A 00        FILL   10,0    ;DONE
0654      *
0654      * CHECK BREAK
0654      * IF NO KEY RET NZ
0654      * IF KEY GET IT AND PUT INTO B
0654      * RETURN ZERO
0654      *
0654 DB 03        CDBRK     IN     TISTAT    ;READY STATUS
0656 00          NOP
0657 E6 02        ANI    DIFLG
0659 EE 02        XRI    MSK1
065B C0          RNZ
065C DB 02        CDBRK0    IN     TDIN      ;NO KEY IS WAITING
065E 00          NOP
065F 47          MOV    B,A       ;CHR IN B AND RZ
0660 C9          RET
0661 0A 00        FILL   10,0

```

```

066B      *
066B      * SPACE FOR AN INITIALIZATION FOR THE
066B      * CONSOLE DEVICE. LIKE A TTY USING A USART ETC
066B      *
066B 3E AA  CDINIT      MVI    A,0AAH      ;DUMMY
066D D3 03          OUT    TISTAT
066F 3E 40          MVI    A,40H      ;RESET THE 8251
0671 D3 03          OUT    TISTAT
0673 3E CE          MVI    A,0CEH      ;SETUP EQUIP
0675 D3 03          OUT    TISTAT
0677 3E 17          MVI    A,17H      ;TRUN IT ON
0679 D3 03          OUT    TISTAT
067B C9              RET
067C 4F 00          FILL   @PLIST-$,0

```

E.3 PRINTER PHYSICAL I/O DRIVERS

```

06CB      *
06CB      * PHYSICAL HANDLER FOR THE LIST DEVICE
06CB      *
06CB      * FOR THE IMSAI 2S10-2 PORT B
06CB      *
06CB          ORG    @PLIST
06CB      *
06CB      * LIST OUTPUT CHECKS FOR PRINTER ATTENTION
06CB      * IF PRINTER ATTENTION CARRY SET (C) ON RETURN
06CB      *
06CB  CD EA 07  LDOUT      CALL   @LDATN      ;PRINTER ATTENTION
06CE D8          RC      PMSK3      ;YES RETURN CARRY SET
06CF DB 05  LDOUT1      IN     PTCTL      ;PRINTER READY
06D1 00          NOP
06D2 E6 01          ANI    PMSK4      ;READY FLAG
06D4 EE 01          XRI    PMSK4
06D6 C2 CB 06          JNZ    LDOUT      ;LOOP
06D9 78          MCV    A,B      ;CHR INTO A FOR OUT
06DA D3 04  LDOUT2      OUT    PTDAT      ;OUTPUT CHR
06DC 00          NOP
06D1 C9          RET
06DE 0A 00          FILL   10,0      ;DONE

```

06E8	*			
06E8	*	PRINTER ATTENTION CHECK		
06E8	*	CARRY SET=ATTENTION		
06F8	*			
06F8 AF	LDATN	XRA	A	;NO OP TO ACTIVATE
06E9 C9		RET		;IF SUPPORTED
06EA DB 05		IN	PTSTS	;ATTENTION STATUS
06EC 00		NOP		
06ED E6 80		ANI	PMSK1	;ATT FLAG
06FF EE 80		XRI	PMSK2	
06F1 C8		RZ		;OK
06F2 37		STC		;SET CARRY FOR ATT
06F3 C9		RET		;DONE
06F4 0A 00		FILL	10,0	
06FE	*			
06FE	*	INITIALIZE THE LIST DEVICE. LIKE A USART ETC		
06FE	*			
06FE AF	LDINIT	XRA	A	;NO OP
06FF C9		RET		;TO ACTIVATE
0700 3E AA		MVI	A,0AAH	;DUMMY
0702 D3 05	LINIT1	OUT	PTCTL	
0704 3E 40		MVI	A,40H	;RESET
0706 D3 05	LINIT2	OUT	PTCTL	
0708 3E CE		MVI	A,0CEH	;SETUP EQUIP
070A D3 05	LINIT3	OUT	PTCTL	
070C 3E 17		MVI	A,17H	;TURN ON
070E D3 05	LINIT4	OUT	PTCTL	
0710 C9		RET		
0711 6A 00		FILL	0PLIST+0B0H-\$	
077B	*			

E.4 CONFIGURATION LOGIC

```

2A2F      *
2A2F      * CONFIG FOR MDOS
2A2F      * CONFIG RESIDES AT THE APP AREA
2A2F      * WHEN THE SYSTEM IS BOOTTED DOWN.
2A2F      * THE USER SETS THE DESIRED CONFIG ON THE
2A2F      * PROGRAM INPUT SWITCHES AND JMP TO CONFIG.
2A2F      * CONFIG INITIALIZES THE TERMINAL HANDLER
2A2F      * AND MOVES THE APPROPRIATE INITIALIZE CODE
2A2F      * INTO PLACE. CHANGES SOFTHALT TO A JMP TO MDOSSTART
2A2F      *
2A2F          ORG     @APROGRAM ;CONFIG BEGINS
2B00      *
2B00 31 A0 01 CNFIG      LXI     SP,@STACK
2B03 3A D0 04           LDA     CNBR      ;CHK FOR VALID CONFIG #
2B06 B7             ORA     A
2B07 FA 48 2B           JM      CN040     ;SPECIAL CONFIGS
2B0A      *
2B0A      *IF BIT7 LOW THEN USE GENERAL HANDLER
2B0A      *ENSURE # IS VALID
2B0A      *
2B0A FE 07             CPI     NUMBER    ;NUMB OF CONFIGS
2B0C DA 12 2B           JC      CN010     ;OK
2B0F C3 0F 2B           JMP     $         ; INVALID CODE TRAP
2B12      *
2B12 21 8A 2B           CN010
2B15 87             ADD     A
2B16 5F             MOV     E,A      ;SET UP DE TO ADD TO HL
2B17 16 20           MVI     D,0      ;DE=A*2
2B19 19             DAD     D
2B1A 5E             MCV     E,M      ;ADDRESS FROM
2B1B 23             INX     H        ;CONFIG TABLE
2B1C 56             MOV     D,M
2B1D 21 72 2B           LXI     H,GHTBL ;TABLE OF LOCATIONS
2B20 0E 0C           MVI     C.NUMITEMS ;TO BE CONFIGED

```

2B22 1A	CN020	LDAX	D	;GET VALUE FROM TABLE
2B23 13		INX	D	;NEXT POSIT IN TABLE
2B24 D5		PUSH	D	;SAVE THIS ADDR
2B25 5E		MOV	E,M	;LOCATION IN HANDLER
2B26 23		INX	H	;TO BE CONFIGURED
2B27 56		MOV	D,M	;INTO DE
2B28 23		INX	H	
2B29 12		STAX	D	;PUT CONFIG INTO HANDLER
2B2A D1		POP	D	;GET ADDR BACK
2B2B 0D		DCR	C	;NUMBER OF LOCATIONS-1
2B2C C2 22 2B	*	JNZ	CN020	;LOOP TILL DONE
2B2F	*	* HANDLER HAS NOW BEEN CONFIGURED FROM TABLE		
2B2F	*	* MOVE INITIALIZATION CODE INTO CINIT		
2B2F	*			
2B2F EB		XCHG		;ADDR OF CINIT LENG
2B30 4E		MOV	C,M	;LENG INTO C
2B31 23		INX	H	
2B32 11 6B 26		LXI	D,CDINIT	;INIT RTN IN HANDLER
2B35 CD 3F 2B		CALL	CN030	;MOVE HL TO DE FOR C
2B38	*			
2B38 21 99 15	STARTUP	LXI	H,3MDOSSTART	;CHANGE SOLF HALT
2B3B 22 CE 04		SHLD	@SOFTHALT+1	;TO START MDOS
2B3E E9		PCHL		;RESTART MDOS
2B3F	*			
2B3F	*	* SIMPLE MOVE CODE - MOVE FROM HL TO DE FOR A LENGTH C		
2B3F	*			
2B3F 7E	CN030	MOV	A,M	
2B40 12		STAX	D	
2B41 23		INX	H	
2B42 13		INX	D	
2B43 0D		DCR	C	
2B44 C2 3F 2B		JNZ	CN030	
2B47 C9		RET		

2B48 *
2B48 *SPECIAL CONFIGURATIONS
2B48 *
2B48 E6 7F CN040 ANI 7FH ;STRIP OFF SPECIAL CODE
2B4A FE 02 CPI NUMSPEC ;NUMB SPECIAL CONFIGS
2B4C D2 40 2B JNC \$;ERROR TRAP SOFT HALT
2B4F 21 5C 2B LXI H,SCTBL ;INDEX INTO TABLE
2B52 87 ADD A ;A*2
2B53 5F MOV E,A
2B54 16 00 MVI D,0
2B56 19 DAD D ;HL=HL+2*A
2B57 5E MOV E,M ;GET ADDR
2B58 23 INX H ;OF SPECIAL
2B59 56 MOV D,M ;CONFIG FROM TABLE
2B5A EB XCHG
2B5B E9 PCHL ;GO TO SPECIAL CONFIG
2B5C *

E.5 CONFIGURATION TABLES

```
2B5C      *
2B5C      * TABLES ****
2B5C      *
2B5C      * TABLE OF SUPPORTED SPECIAL CONFIGURATIONS
2B5C      *
2B5C 39 20    SCTBL      DW      COMPAL
2B5E 8A 20    END
2B60 2B60    ENDSPECIAL EQU     $
2B60 12 00    FILL      18,0      ;EXTRA SPACE
2B72      *
2B72      * PATCH LOCATIONS IN THE RESIDENT HANDLER
2B72      *
2B72 1C 06    GHTBL      DW      CDIN+1      ;TISTAT
2B74 3C 06    DW      CDOUT+1      ;TOSTAT
2B76 55 06    DW      CDBRK+1      ;TISTAT
2B78 2D 06    DW      IN010+1      ;TDIN
2B7A 5D 06    DW      CDBRK0+1      ;TDIN
2B7C 47 06    DW      OUT020+1      ;TDOOUT
2B7E 1F 06    DW      CDIN+4      ;DIFLG
2B80 21 06    DW      CDIN+6      ;MSK1
2B82 58 06    DW      CDBRK+4      ;DIFLG
2B84 5A 06    DW      CDBRK+6      ;MSK1
2B86 3F 06    DW      CDCUT+4      ;DOFLG
2B88 41 06    DW      CDOUT+6      ;MSK2
2B8A      *
2B8A      *
2B8A      *TABLE OF THE SUPPORTED STANDARD CONFIGURATIONS
2B8A      *
2B8A      *
2B8A AC 2B    CNTBL      DW      CNFG0      ;ALTAIR 88-SIO
2B8C C2 2B    DW      CNFG1      ;IMSAI S102
2B8E E0 2B    DW      CNFG2      ;ALTAIR SIO A,B,C
2B90 EE 2B    DW      CNFG3      ;ALTAIR SIO A,B,C (REV 0)
2B92 FC 2B    DW      CNFG4      ;PTC 3P+S
2B94 2A 2C    DW      CNFG5      ;IMSAI MIO
2B96 1B 2C    DW      CNFG6      ;ALTAIR 88-4PIO
2B98      *
2B98 14 00    ENDCTBL   FILL      20,0
```

```

2BAC *
2BAC *SERIAL INTERFACES
2BAC *
2BAC ****
2BAC *CONFIGURATION 0-- ALTAIR 88-2SIO
2BAC *OR OTHER SIO USING MOTOROLA
2BAC *6850 UART
2BAC ****
2BAC *
2BAC 10 10 10 CNFG0 DB 16,16,16,17,17,17,1,1,1,1,2,2
2BAF 11 11 11
2BB2 01 01 01
2BB5 01 02 02
2BB8 09
2BB9 3E 03
2BBB D3 10
2BBD 3E 11
2BBF D3 10
2BC1 C9
2BC2 *
2BC2 *
2BC2 *
2BC2 ****
2BC2 *CONFIGURATION 1-- IMSAI SI02
2BC2 *OR OTHER SIO USING THE INTEL 8251 USART
2BC2 ****
2BC2 *
2BC2 *
2BC2 03 03 03 CNFG1 DB 3,3,3,2,2,2,2,2,2,1,1
2BC5 02 02 02
2BC8 02 02 02
2BCB 02 01 01
2BCE 11
2BCF 3E AA
2BD1 D3 03
2BD3 3E 40
2BD5 D3 03
2BD7 3E CE
2BD9 D3 03
2BDB 3E 17
2BDD D3 03
2BDF C9
      DB    17 ;INIT LENGTH
      MVI   A,0AAH ;DUMMY
      OUT   3
      MVI   A,40H ;RESET
      OUT   3
      MVI   A,0CEH
      OUT   3
      MVI   A,17H ;TURN ON
      OUT   3
      RET

```

```

2BE0      *
2BE0      *
2BE0      *
2BE0      ****
2BE0      *CONFIGURATION 2-- ALTAIR SIO A,B,C
2BE0      *(NOT REV 0) OR OTHER UART TYPE SERIAL
2BE0      *I/O BOARD NOT REQUIRING INITIALIZATION
2BE0      ****
2BE0      *
2BE0      *
2BE0      *
2BE0      *
2BE0      00 00 00 CNFG2      DB      0,0,0,1,1,1,1,0,1,0,80H,0
2BE3      01 01 01
2BE6      01 00 01
2BE9      00 80 00
2BEC      01
2BED      C9      DB      1
2BEE      RET      ;DONE
2BEE      *
2BEE      *
2BEE      *
2BEE      ****
2BEE      *CONFIGURATION 3-- ALTAIR SIO A,B,C (REV 0)
2BEE      ****
2BEE      *
2BEE      *
2BEE      *
2BEE      *
2BEE      00 00 00 CNFG3      DB      0,0,0,1,1,1,20H,20H,20H,20H,2,2
2BF1      01 01 01
2BF4      20 20 20
2BF7      20 02 02
2BFA      01
2BFB      C9      DB      1
2BFB      RET      ;DONE

```

2BFC
2BFC *
2BFC *
2BFC *****
2BFC *CONFIGURATION 4-- PROCESSOR TECHNOLOGY 3P+S
2BFC *SERIAL I/O
2BFC *****
2BFC *
2BFC *
2BFC *
2BFC 00 00 00 CNFG4 DB 0,0,0,1,1,1,40H,40H,40H,40H,80H,80H
2BFF 01 01 01
2C02 40 40 40
2C05 40 80 80
2C08 01 DB 1
2C09 C9 RET ;DONE
2C0A *
2C0A *
2C0A *
2C0A *****
2C0A *CONFIGURATION 5-- IMSAI MIO SERIAL I/O
2C0A *****
2C0A *
2C0A *
2C0A *
2C0A 43 43 43 CNFG5 DB 43H,43H,43H,42H,42H,42H,2,2,2,2,1,1
2C0D 42 42 42
2C10 02 02 02
2C13 02 01 01
2C16 04 DB 4 ;INIT LENGTH
2C17 AF XRA A
2C18 D3 43 OUT 43H
2C1A C9 RET ;DONE

```

2C1B *
2C1B *
2C1B *
2C1B * PARALEL INTERFACES
2C1B *
2C1B *****
2C1B *CONFIGURATION 6-- ALTAIR 88-4PIO
2C1B *OR OTHER PIO USING THE MOTOTOLA 6820 PIA
2C1B *****
2C1B *
2C1B *
2C1B *
2C1B 10 10 10 CNFG6 DB 16,16,16,17,17,19,80H,80H,80H,80H,3,3
2C1E 11 11 13
2C2I 80 80 80
2C24 80 03 03
2C27 11 DB 17 ;INIT LENGTH
2C28 AF XRA A ;SELCT DATA DIRECTION
2C29 D3 10 OUT 16 ;REGESTER AND SET
2C2B D3 11 OUT 17 ;PORT A=INK
2C2I D3 12 OUT 18 ;PORT B=CUT
2C2F 2F CMA
2C32 D3 13 OUT 19 ;CA2/CB2=OUTPUT
2C32 3E 24 MVI A,24H ;LEVEL HANDSHAKE
2C34 D3 10 OUT 16
2C36 D3 12 OUT 18
2C38 C9 RFT ;DONE

```

2C39 *

2C39 *****

2C39 *SPECIAL CONFIGURATION 0 -- COMPAL 80

2C39 *TERMINAL I/O IS PERFORMED THROUGH THE COMPAL

2C39 *MONITOR

2C39 *****

2C39 *

2C39 2A B8 04 COMPAL LHLD @EIADDR ;ENABLE INTERRUPT AFTER

2C3C 36 FB MVI M,0FBH ;DISK ACCESS

2C3E 21 6E 20 LXI H,CI ;CHR IN RTN IN MONITOR

2C41 11 1B 26 LXI D,CDIN

2C44 0E 06 MVI C,CO-CI ;CI LENGTH

2C46 CD 3F 2B CALL CN030 ;MOVE IT INTO PLACE

2C49 21 74 2C LXI H,CO ;COMPAL CHR OUT

2C4C 11 3B 06 LXI D,CDBRK

2C4F 0E 06 MVI C,CCDBRK-CO ;CO LENGTH

2C51 CD 3F 2B CALL CN030

2C54 21 7A 2C LXI H,CCDBRK

2C57 11 54 06 LXI D,CDBRK

2C5A 0E 10 MVI C,CEND-CCDBRK

2C5C CD 3F 2B CALL CN030

2C5F 21 AF C9 LXI H,0C9AFH ;XRA A RET

2C62 22 6B 06 SHLD CDINIT ;DISABLE CDINIT

2C65 21 01 01 LXI H,101H ;WRAP OFF NULLS 0

2C68 22 FE 04 SHLD WRAPFLAG ;WRAP AND NULLS

2C6B C3 38 2B JMP STARTUP

2C6E *

2C6E *

2C6E CD AD E0 CI CALL 0E0ADE ;CHR IN COMPAL MONITOR

2C71 47 MOV B,A

2C72 B7 ORA A ;CLEAR CARRY

2C73 C9 RET

2C74 *

2C74 78 CO MOV A,B

2C75 CD C2 F0 CALL 0E0C2H ;CHR OUR COMPAL MONITOR

2C78 AF XRA A ;CLEAR CARRY

2C79 C9 RET

2C7A *

2C7A FB CCDBRK EI

2C7B 3A FD ED LDA 0EDFDH ;CHR FROM INTERRUPT KEYBRD

2C7E 47 MCV B,A

2C7F B7 ORA A

2C80 CA 62 06 JZ CDBRK+14

2C83 AF		XRA	A
2C84 32 FD ED		STA	0EDFDH
2C87 C9		RET	
2C88 3C		INR	A
2C89 C9		RET	
2C8A *			
2C8A 2C8A	CEND	EQU	\$
2C8A *			
2C8A *****			
2C8A *		SPECIAL CONFIGURATION 1 --	
2C8A *		PROCESSOR TECHNOLOGY SOL-20	
2C8A *		WITH SOLOS 1.3	
2C8A *****			
2C8A *			
2C8A 3E 3F	SOL	MVI	A,63 ;WIDTH
2C8C 32 00 05		STA	WIDTH
2C8F 0E 0A		MVI	C,SOLOUT-SOLIN ;INPUT LEN
2C91 11 1B 06		LXI	D,CDIN
2C94 21 C4 2C		LXI	H,SOLIN
2C97 CD 3F 2B		CALL	CN030 ;MOVE SOLIN TO CDIN
2C9A 0E 06		MVI	C,SOLCDBRK-SOLOUT
2C9C 11 3B 06		LXI	D,CDOU
2C9F 21 CE 2C		LXI	H,SOLOUT
2CA2 CD 3F 2B		CALL	CN030
2CA5 0E 0D		MVI	C,SOLINIT-SOLCDBRK
2CA7 11 54 06		LXI	D,CDBRK
2CAA 21 D4 2C		LXI	H,SOLCDBPK
2CAD CD 3F 2B		CALL	CN030
2CB0 0E 05		MVI	C,SCLEND-SOLINIT
2CB2 11 6B 06		LXI	D,CDINIT
2CB5 21 E1 2C		LXI	H,SOLINIT
2CB8 CD 3F 2B		CALL	CN030
2CBB 21 01 01		LXI	H,101H ;WRAPOFF, NULLS 0
2CBE 22 FE 04		SHLD	WRAPFLAG
2CC1 C3 38 2B		JMP	STARTUP
2CC4 *			
2CC4 *			
2CC4 AF	SOLIN	XRA	A
2CC5 CD 22 C0		CALL	0C022H ;PSUDO PORT 0
2CC8 CA 1C 06		JZ	CDIN+1
2CCB 47		MOV	B,A
2CCC B7		ORA	A
2CCD C9		RET	

2CC E	*			
2CC E AF	SOLOUT	XRA	A	
2CC F CD 1C C0		CALL	0C01CH	;PSUDO PORT 0
2CD 2 B7		ORA	A	
2CD 3 C9		RET		
2CD 4	*			
2CD 4 AF	SOLCDBRK	XRA	A	
2CD 5 CD 22 C0		CALL	0C022H	
2CD 8 CA 5F 06		JZ	CDBRK+10	
2CD B 47		MOV	B,A	
2CD C AF		XRA	A	;ZERO=READY
2CD D C9		RET		
2CD E AF		XRA	A	;CDBRK+10
2CD F 3C		INR	A	
2CE 0 C9		RET		;NO ZERO CARRY CLEAR
2CE 1	*			
2CE 1 3E 0B	SOLINIT	MVI	A,2BH	
2CE 3 C3 3B 06		JMP	CDOUT	;CLEAR SCREEN SET CURSOR
2CF 6 2CE 6	SOLEND	EQU	\$	
2CE 6	*			
2CE 6	*			
2CE 6	*			
2CE 6 000C	NUMITEMS	EQU	CNTBL-GHTBL/2	
2CE 6 0007	NUMBER	EQU	ENDCNTBL-CNTBL/2	
2CE 6 0002	NUMSPEC	EQU	ENDSPECIAL-SCTBL/2	

APPENDIX F - MICROPOLIS DISK BOOTSTRAP

The Micropolis Disk Bootstrap Program resides in PROM on the controller B board, occupying the first 512 bytes of the controller address space. The bootstrap is involved by starting program execution at the base address of the controller. An address-independent relocator determines the controller base address and moves the bootstrap code from PROM to low RAM system memory where it is executed. The Bootstrap Program selects drive unit 0 and reads the contents of sector 0 of track 0 (the System Loader Program) into memory. Sector 0 must be formatted as described in Section 6.1.2 and must be organized as follows:

Byte 0	Track ID
Byte 1	Sector ID
Byte 2-11	(Ignored)
Byte 12-265	System Loader Program
Byte 266-267	Load Address

Sector 0 is read into RAM at the system loader origin specified by bytes 266 and 267. After a successful read, the bootstrap transfers control to load address +12. The DE register pair will contain the controller base address.

The Bootstrap Program requires approximately 1K of RAM memory from address 90H.

*
* MICROPOLIS DISK BOOTSTRAP
*
* VERSION 2 -- RELOCATABLE
* BOOTSTRAP - OPERATES WITH
* CONTROLLER STRAPPED FOR ANY
* LOCATION FROM C000H-FC00H
*
* PROM PART NUMBERS:
* HIGH 800003-01-4C
* LOW 800003-02-2C
*
* RELEASE 1.0
* COPYRIGHT MICROPOLIS CORPORATION
* OCTOBER 11 1977
*

*
* REGISTER DEFINITIONS AND *
* FLAG EQUATES FOR MICROPOLIS *
* FLEXIBLE DISK CONTROLLER B *

*
*
* F400 BPROM EQU X'F400'
* DEFINITIONS GIVEN FOR STANDARD
* ADDRESS OF F400H -- CONTROLLER
* MAY ACTUALLY BE STRAPPED FOR
* ANY 1K BOUNDARY FROM C000H -FC00H
*
* F600 DISK EQU BPROM+X'0200'
*
* DATA REGISTERS
*
* F602 WDATA EQU DISK+X'02'
* F602 RDATA EQU WDATA
*
* STATUS REGISTERS
*
* F600 DSECTR EQU DISK
* 0-3 SECTOR COUNT
* 4 SPARE
* 5 SPARE
* 6 SCTR INTERRUPT FLAG
* 7 SECTOR FLAG
*
* FLAG BITS
*
* 0040 SIFLG EQU X'40'
* 0080 SFLG EQU X'80'
* 0020 DTMR EQU X'20'
*
* F601 DSTAT EQU DISK+1

```

*      0-1      UNIT ADDRESS
*      2      UNIT SELECTED (LOW TRUE)
*      3      TRACK 0
*      4      WRITE PROTECT
*      5      DISK READY
*      6      PINT
*      7      TRANSFER FLAG
*
*      FLAG BITS
*
0080    TFLG    EQU  X'80'
2040    INTE    EQU  X'40'
2020    RDY     EQU  X'20'
0010    WPT     EQU  X'10'
0008    TK0     EQU  X'08'
2004    USLT    EQU  X'04'
*
*
*      COMMAND REGISTER
*
F600    DCMND   EQU  DISK
*(ALSO WILL RESPOND TO DISK+1)
*
*      0-1      COMMAND MODIFIER
*      5-7      COMMAND
*
*      COMMANDS
*
0020    SLUN    EQU  X'20'      SELECT UNIT
*      MODIFIER CONTAINS UNIT ADDRESS
0040    SINT    EQU  X'40'      SET INTERRUPT
*      MODIFIER =1  ENABLE INTERRUPT
*      =0  DISABLE INTERRUPT
0060    STEP    EQU  X'60'      STEP CARRIAGE
*      MODIFIER =00  STEP OUT
*      =01  STEP IN
0080    WRITE   EQU  X'80'      ENABLE WRITE
*      NO MODIFIER USED
00A0    RESET   EQU  X'A0'      RESET CONTROLLER
*      NO MODIFIER USED
*
*
*      DISK PARAMETERS
*
000F    SDLY    EQU  15      STEP+SETTLE TIME
*      DIVIDED BY 2.6775
0086    BYTCT   EQU  134     BY TCT/2
*
*
*****  

*      PROM-RESIDENT BOOTSTRAP  

*  

*****  

*      BOOTSTRAP REQUIRES AT LEAST 1K  

*      OF RAM MEMORY FROM 90H

```

*
* RELOCATES FROM PROM INTO RAM THEN
* BOOTSTRAP LOADS SECTOR ZERO OF
* TRACK ZERO INTO RAM AND STARTS
* THE PROGRAM LOADED
*

* SECTOR ZERO IS ORGANIZED AS
* FOLLOWS:

* BYTES 0-1 HEADER
* BYTES 2-265 USER PROGRAM
* BYTES 266-267 RAM ADDRESS

* BOOTSTRAP WILL READ SECTOR ZERO
* INTO RAM STARTING AT THE
* ADDRESS SPECIFIED BY BYTES
* 266 & 267 AND WILL START
* THE PROGRAM AT RAM ADDRESS +12

00A0 CTOORG EQU X'A0' CONTROLLER BASE
* ADDRESS SAVED HERE

0000 ORG CTOORG-X'35' CTOORG+2-RCLLEN)

* RELOCATOR -- MOVES BOOTSTRAP INTO
* RAM AND STARTS BOOTSTRAP

006F F3 RELOC DI
006C 21A200 LXI H,CTORG+2 STUFF A RETURN IN
006F F9 SPHL RAM AND CALL IT TO
0070 36C9 MVI M,X'C9' DETERMINE ADDRESS
0072 CDA200 CALL CTOORG+2 OF CONTROLLER
0075 EB XCHG SAVE RAM ADDR
0076 2AA000 LHLD CTOORG GET ADDRESS WHICH
0079 2E00 MVI L,0 WAS PUSHED ON STAC
007B E5 PUSH H MSB IS CTLR ADDR
007C 011D00 LXI B,BTDSP1 BUILD MOVE LOOP
007F 09 DAD B ADDRESS
0080 E5 PUSH H STUFF ON STACK
0081 F1 POP H ADJUST SP
0082 2E1A MVI C,BTDSP2 BUMP HL TO START
0084 09 DAD B OF BOOT CODE
0085 06BD MVI B,BTLEN
0087 EB XCHG
0088 3B RE010 DCX SP ADJUST SP TO POINT
0089 3B DCX SP TO RE010 ON STACK
008A 1A LDAX D MOVE BYTE FROM
008B 77 MOV M,A PROM TO RAM

*
* COMPARE MEMORY WITH A REG --
* IF DIFFERENT THEN DESTINATION
* RAM IS BAD OR IS PROM --
* RELOCATOR WILL LOOP IN MOVE
* LOOP UNTIL SUCCESSFUL

008C BE CMP M GOOD MOVE?

008D	C0	RNZ	NO-LOOP
008E	23	INX H	
008F	13	INX D	
0090	05	DCR B	DONE?
0091	C0	RNZ	NO-LOOP
0092	E1	POP H	YES-CLEAN UP STACK
0093	2AA000	LHLD CTOORG	BUILD CONTROLLER
0096	110002	LXI D,X'200'	ADDRESS FROM BASE
0099	19	DAD D	
009A	22A200	SHLD DADR	AND SAVE
009D	36A0	MVI M,RESET	RESET CONTROLLER
009F	C3D400	JMP SL010	AND GO START BOOT
*			
001D	BTDSP1	EQU RE010-RELOC	
001A	BTDSP2	EQU *-RE010	
0037	RLCLEN	EQU *-RELOC	
*			
00A2	BOOT	EQU *	
00A2	DADR	DS 2	
00A4	LDRST	DS 2	
*			
*			
*			
* READ 1 SECTOR			
*			
*			
* B=SECTOR			
* C=BYTECOUNT /2			
* DE=READ BUFFER			
*			
* A,HL ARE DESTROYED			
*			
* RETURNS Z=OK			
* NZ=ERROR			
*			
*			
* WAIT FOR DESIRED SECTOR			
*			
00A6	2AA200	RDSEC LHLD DADR	
00A9	7E	MOV A,M	WAIT SCTR FLAG
00AA	E680	ANI SFLG	
00AC	CAA900	JZ RDSEC+3	
00AF	7E	MOV A,M	OK-IS THIS THE
00B0	E60F	ANI X'0F'	DESIRED SCTR?
00B2	A8	XRA B	
00B3	C2A900	JNZ RDSEC+3	NO-WAIT
*			
* FOUND DESIRED SECTOR GO READ			
*			
00B6	23	INX H	
*			
*			
00B7	B6	RD005 ORA M	WAIT FOR TRANSFER
* FLAG			
00B8	F2B700	JP RD005	
*			
* TRANSFER FLAG SET-STRIP			

* SYNC BYTE

00BB 23	INX H	
00BC 7E	MOV A,M	READ SYNC BYTE
00BD AF	XRA A	CLEAR CARRY
00BE EB	XCHG	
00BF 0600	MVI B,0	AND CHECKSUM
00C1 00	NOP	
00C2 00	NOP	

* READ LOOP

00C3 1A	RD010 LDAX D	READ FROM DISK
00C4 77	MOV M,A	MOVE TO BUFFER
00C5 23	INX H	NEXT LOC
00C6 88	ADC B	ADD TO CHECKSUM
00C7 47	MOV B,A	AND SAVE
00C8 1A	LDAX D	NEXT READ
00C9 77	MOV M,A	-ETC-
00CA 23	INX H	
00CB 88	ADC B	
00CC 47	MOV B,A	
00CD 0D	DCR C	END OF DATA?
00CE C2C300	JNZ RD010	NO-LOOP

* END OF DATA-READ CHECKSUM

00D1 1A	LDAX D	
00D2 B8	CMP B	COMPARE WITH
00D3 C9	RET	COMPUTED CHECKSUM

* SELECT DRIVE 0

00D4 2AA200	SL010 LHLD DADR	SELECT DRIVE
00D7 3620	MVI M,SLUN	
00D9 23	INX H	
00DA 7E	MOV A,M	
00DB 2B	DCX H	
00DC E624	ANI RDY+USLT	CHECK SLTD & RDY
00DF EE20	XRI RDY	WAIT UNTIL OK
00E0 C2D400	JNZ SL010	TO PROCEED

*		WAIT 250 MSEC
00E3 0E5E	MVI C,94	FOR SECTOR CNTR
00E5 CD4901	CALL TIMER	TO SYNC
00E8 23	SL020 INX H	
00E9 7E	MOV A,M	READ STATUS AGAIN
00EA 2B	DCX H	
00EB E624	ANI RDY+USLT	TO ENSURE STILL
00ED EE20	XRI RDY	OK TO PROCEED
00EF C2D400	JNZ SL010	NO-TRY AGAIN

*
* RESTORE DRIVE TO TRACK 0
*

00F2 23	CZERO	INX H	READ STATUS
00F3 7E		MOV A,M	
00F4 E608		ANI TK0	TRACK 0?
00F6 2B		DCX H	
00F7 CA0701		JZ CZ030	NO-PRESS ON

*
* IF ALREADY AT TRACK ZERO
* STEP IN THEN BACK OUT
* TO ENSURE A GOOD POSITION
*

00FA 0608		MVI B,8	STEP IN 8 TKS
00FC 3661	CZ010	MVI M,STEP+1	STEP IN
00FE 0E0F		MVI C,SDLY	DELAY SEEK +
0100 CD4901		CALL TIMER	SETTLE TIME
0103 05	CZ020	DCR B	
0104 C2FC00		JNZ CZ010	LOOP UNTIL IN
*			
0107 23	CZ032	INX H	READ STATUS
0108 7E		MOV A,M	TRACK 0?
0109 E608		ANI TK0	
010B 2B		DCX H	
010C C21901		JNZ RSZERO	YES-PRESS ON
010F 3660		MVI M,STEP	NO-STEP OUT
0111 0E0F		MVI C,SDLY	DELAY
0113 CD4901		CALL TIMER	THEN TEST AGAIN
0116 C30701		JMP CZ030	

*
* READ THROUGH SECTOR ZERO
* ONE TIME TO FIND RAM ADDRESS
* THEN READ PROGRAM IN & START
*

0119 215F01	RSZERO	LXI H,BTBUF	
011C CD3701		CALL RZERO	READ SCTR ZERO-
011F C2D400		JNZ SL010	RESEEK IF HDR BAD
0122 2A6902		LHLD BTBUF+266	GET PGM ADDRESS
0125 22A400		SHLD LDRST	GO LOAD PGM
0128 CD3701		CALL RZERO	
012B C2D400		JNZ SL010	RESEEK IF HDR BAD
012E 2AA400		LHLD LDRST	COMPUTE START
0131 110C00		LXI D,12	ADDRESS AND GO
0134 19		DAD D	START PROGRAM
0135 D1		POP D	(CTLR CRG STILL
0136 E9		PCHL	ON STACK)
*			

0137 E5	RZERO	PUSH H	SAVE RAM ADDRESS
2138 EB		XCHG	DE<-ADDRESS
0139 018600		LXI B,BYTCT	
013C CDA600		CALL RDSEC	READ IN SECTOR 0
013F E1		POP H	
0140 C23701		JNZ RZERO	RETRY IF CKSUM ERR
0143 E5		PUSH H	
0144 7E		MOV A,M	CHECK HEADER
0145 23		INX H	
0146 B6		CRA M	

0147 E1		POP H	
0148 C9		RET	
*			
*			
*			
* DELAY TIMER			
* HL=DISK CONTROLLER ADDRESS			
* C=DELAY TIME IN MSEC/2.678			
*			
* A,C ARE DESTROYED			
*			
0149 7E	TIMER	MOV A,M	READ STATUS TO
014A E620		ANI DTMR	RE-TRIGGER
014C 79		MOV A,C	4 SECOND TIMER
014D C25101		JNZ *+4	
0150 07		RLC	
0151 4F		MOV C,A	
0152 3EFF	TI010	MVI A,X'FF'	
0154 D601		SUI 1	DELAY LOOP=2.678
0156 B7		CRA A	MSEC @500 NSEC
0157 C25401		JNZ TI010+2	
*			
* DELAY EXPIRED - DECREMENT DELAY			
* MULTIPLIER & CHECK FOR DONE			
*			
015A 0D		DCR C	
015B C25201		JNZ TI010	
015E C9		RET	
*			
*			
00BD	BTLEN	EQU **-BOOT	
*			
* TEMP READ BUFFER FOR FIRST			
* READ OF SECTOR 0			
*			
015F	BTBUF	DS 300	
*			
028B		ORG RELOC+253	
0168 C3A600	RDLNK	JMP RESEC	
*			
016B		END BPROM	

APPENDIX G - FEATURES PROGRAM TO OPTIONALLY SHORTEN BASIC

BASIC contains features which are very useful during program development but unnecessary when running debugged production programs. It is possible to selectively delete some or all of these features. When these features are removed the program buffer is enlarged. If all these features are removed, the program buffer starts at the same place as for versions of BASIC prior to version 4.0 (5700 hex).

A special assembly language program called FEATURES is supplied to selectively remove features from BASIC. The procedure is as follows:

- 1) Load BASIC from MDOS or by booting a BASIC only diskette.
- 2) Type LINK "FEATURES" and a carriage return.

The FEATURES program contains interlocks which prevent it from running if loaded from MDOS or if loaded under the wrong version of the system. The message BASIC NOT LOADED is displayed if FEATURES is run under MDOS and control returns to MDOS. The message SYSTEM VERSION ERROR is displayed and the system soft halts if FEATURES is run under a system with a version number other than that of the FEATURES program. Example: if the system you are running is PDS 3.0, an attempt to run FEATURES 4.0 will result in the message SYSTEM VERSION ERROR and the system will soft halt. To continue you must reboot.

- 3) If there are no inconsistencies, the FEATURES program signs on:

BASIC V.S. 4.0 FEATURES PROGRAM

ENTER NUMBER OF DESIRED FUNCTION (CONTROL-C TO EXIT)

1-REMOVE MERGE
2-REMOVE RENUM AND MERGE
3-REMOVE EDIT, RENUM AND MERGE

?

- 4) Select the desired function and enter its number following the question mark. To exit the FEATURES program hold the control key down and simultaneously press the letter C. The FEATURES program will only respond to legal function numbers and the CONTROL-C. Entries other than these will reprompt ENTER NUMBER OF DESIRED FUNCTION (CONTROL-C TO EXIT) followed by the menu.
- 5) When the selected features are removed the program displays FUNCTION COMPLETE and warmstarts BASIC which signs on:

MICROPOLIS BASIC V.S. X.X - COPYRIGHT 1978

NOTE: Running the FEATURES program on a copy of BASIC which has already been shortened will not replace removed features. For example, if you have created a BASIC with EDIT, RENUM and MERGE removed, rerunning the FEATURES

program and selecting option 1 will not replace the EDIT AND RENUM commands. It will, however, set the beginning of the program buffer as if the commands were there thus loosing the user program memory space.

The shortened BASIC created by the FEATURES program may be saved on the systems disk as described in Section 2.2.6 CREATING YOUR SYSTEM DISKETTE; or saved as a BASIC only disk as described in Section 2.2.7 CREATING A BASIC ONLY SYSTEM DISKETTE; or used immediately as it is inserted in the system memory. In this case it is lost on power off or when any other system (MDOS DISKCOPY) is loaded.

MICROPOLIS SOFTWARE INFORMATION BULLETIN

S.I.B. # 012

Release Date 1-2-79

SUBJECT: Serious Error in the CHAIN Statement Logic of Micropolis BASIC 4.0

The BASIC interpreter of Micropolis PDS 4.0 has a bug in the logic of the CHAIN statement that can cause unpredictable results leading to a major system crash.

ALL SYSTEM USERS SHOULD PERFORM THIS PATCH.

To patch BASIC 4.0 for correct operation, perform the following steps:

1. Boot load MDOS from a configured SYSTEM diskette.
2. Type BASIC and press the RETURN key. This causes the BASIC interpreter to be loaded into memory and control is transferred to the BASIC monitor, which signs on and prompts READY.
3. Type POKE(16R1512) = 16RC3 and press the RETURN key.
4. Type POKE(16R1513) = 16R72 and press the RETURN key.
5. Type POKE(16R1514) = 16R15 and press the RETURN key.
6. Type POKE(16R36A1) = 16R5E:POKE(16R36A2) = 16R23 and press the RETURN key.
7. Type POKE(16R36A3) = 16R56 and press the RETURN key,

The BASIC interpreter is now patched correctly in memory.

8. Type OPEN 1 "BASIC" and press the RETURN key.
9. Type ATTRS(1) = 8 and press the RETURN key. This removes the file protect attribute from the BASIC disk file.
10. Type SAVE "BASIC" 16R1512,16R5DFF and press the RETURN key.
11. Type ATTRS(1) = 16RF and press the RETURN key.
12. Type CLOSE 1 and press the RETURN key.

The corrected BASIC is now rewritten on the SYSTEM diskette.

MICROPOLIS™

MICROPOLIS SOFTWARE INFORMATION BULLETIN

S.I.B. # 013

Release Date REV. B -- 2/12/79

SUBJECT: Implementation of Centronics 703 and 779 Printers -- Micropolis PDS 4.0

The Micropolis PDS 4.0 system issues the linefeed before the carriage return character during output to the console and printer devices. When implementing the Centronics printers, Models 703 and 779 fail to work properly. The linefeed character causes the printer to prematurely clear its line buffer, resulting in no printed output.

The following patch for MDOS and BASIC will reverse the order of the linefeed/carriage return sequence to avert this difficulty.

To patch the PDS 4.0 system for correct operation, perform the following steps:

- 1) Boot load MDOS from a configured system diskette.
- 2) Type ENTR 5B7 and press the RETURN key.
- 3) Type D/ and press the RETURN key.
- 4) Type ENTR 5BD and press the RETURN key.
- 5) Type A/ and press the RETURN key.
- 6) Type ENTR 80F and press the RETURN key.
- 7) Type D/ and press the RETURN key.
- 8) Type ENTR 814 and press the RETURN key.
- 9) Type A/ and press the RETURN key.
- 10) Type ENTR 81C and press the RETURN key.
- 11) Type D/ and press the RETURN key.
- 12) Type ENTR 821 and press the RETURN key.
- 13) Type A/ and press the RETURN key.

The RES module is now correctly patched in system memory.

MICROPOLIS™

- 14) Type TYPE "RES" Ø and press the RETURN key. This removes any protect attributes from the filetype of the RES file.
- 15) Type SCRATCH "RES" and press the RETURN key. This deletes the old RES file from the system diskette.
- 16) Type SAVE "RES" 2B1 1598 3 and press the RETURN key. This writes the corrected RES module onto the system diskette.
- 17) Type LOAD "LINEEDIT" and press the RETURN key.
- 18) Type ENTR 359E and press the RETURN key.
- 19) Type D/ and press the RETURN key.
- 20) Type ENTR 35A3 and press the RETURN key.
- 21) Type A/ and press the RETURN key.
- 22) Type TYPE "LINEEDIT" Ø and press the RETURN key.
- 23) Type SCRATCH "LINEEDIT" and press the RETURN key.
- 24) Type SAVE "LINEEDIT" 2BØØ 38ØØ 17 and press the RETURN key.
- 25) Type LOAD "ASSM" and press the RETURN key.
- 26) Type ENTR 3921 and press the RETURN key.
- 27) Type D/ and press the RETURN key.
- 28) Type ENTR 3926 and press the RETURN key.
- 29) Type A/ and press the RETURN key.
- 30) Type TYPE "ASSM" Ø and press the RETURN key.
- 31) Type SCRATCH "ASSM" and press the RETURN key.
- 32) Type SAVE "ASSM" 2BØØ 3DØØ 17 and press the RETURN key.
- 33) Type BASIC and press the RETURN key. This causes the BASIC interpreter to be loaded into memory and control is transferred to the BASIC monitor, which signs on and prompts READY.
- 34) Type POKE(16R1512) = 16RC3 and press the RETURN key.

- 35) Type POKE(16R1513) = 16R72 and press the RETURN key.
- 36) Type POKE(16R1514) = 16R15 and press the RETURN key.
- 37) Type POKE(16R4061) = 16R0D:POKE(16R4066) = 16R0A and press the CARRIAGE RETURN key. BASIC is now correctly patched in memory.
- 38) Type OPEN 1 "BASIC" and press the RETURN key.
- 39) Type ATTRS(1) = 8 and press the RETURN key.
- 40) Type SAVE "BASIC" 16R1512, 16R5DFF and press the RETURN key.
- 41) Type ATTRS(1) = 16RF and press the RETURN key.
- 42) Type CLOSE 1 and press the RETURN key. The corrected BASIC is now written on the system diskette.

MICROPOLIS SOFTWARE INFORMATION BULLETIN

S.I.B. # 014

Release Date 1-2-79

SUBJECT: Serious Error in the String Concatenation Logic of Micropolis BASIC 4.0

The BASIC interpreter of Micropolis PDS 4.0 has a bug in the string concatenation logic which allows a string to overflow beyond 250 characters and cause a major system crash.

ALL USERS SHOULD PERFORM THIS PATCH.

To patch BASIC 4.0 for correct operation, perform the following steps:

- 1) Boot load MDOS from a configured SYSTEM diskette.
- 2) Type BASIC and press the RETURN key. This causes the BASIC interpreter to be loaded into memory and control is transferred to the BASIC monitor, which signs on and prompts READY.
- 3) Type POKE(16R1512) = 16RC3:POKE(16R1513) = 16R72 and press the RETURN key.
- 4) Type POKE(16R1514) = 16R15:POKE(16R1567) = 16RDA and press the RETURN key.
- 5) Type POKE(16R1568) = 16R60:POKE(16R1569) = 16R4F and press the RETURN key.
- 6) Type POKE(16R156A) = 16RFE:POKE(16R156B) = 16RFB and press the RETURN key.
- 7) Type POKE(16R156C) = 16RDA:POKE(16R156D) = 16R62 and press the RETURN key.
- 8) Type POKE(16R156E) = 16R4F:POKE(16R156F) = 16RC3 and press the RETURN key.
- 9) Type POKE(16R1570) = 16R60:POKE(16R1571) = 16R4F and press the RETURN key.
- 10) Type POKE(16R4F5D) = 16RC3:POKE(16R4F5E) = 16R67 and press the RETURN key.
- 11) Type POKE(16R4F5F) = 16R15:POKE(16R4F61) = 16RFA and press the RETURN key.

MICROPOLIS™

- 12) Type OPEN 1 "BASIC" and press the RETURN key.
- 13) Type ATTRS(1) = 8 and press the RETURN key. This removes the file protect attribute from the BASIC disk file.
- 14) Type SAVE " BASIC" 16R1512, 16R5DFF and press the RETURN key.
- 15) Type ATTRS(1) = 16RF and press the RETURN key.
- 16) Type CLOSE 1 and press the RETURN key.

The corrected BASIC is now rewritten on the SYSTEM diskette.

MICROPOLIS SOFTWARE INFORMATION BULLETIN

S.I.B. # 015

Release Date 1-22-79

SUBJECT: SOL-20 Printer Output via SOL's Printer Port -- PDS 4.0

SOL-20 printer output via SOL's serial port can be accomplished by implementing the following patches:

- 1) Boot load MDOS from a configured system diskette.
- 2) Type ENTR 50A and press the RETURN key.
- 3) Type CB 06/ and press the RETURN key.
- 4) Type ENTR 6CB and press the RETURN key.
- 5) Type CD 4A C0 37 3F C9/ and press the RETURN key.

The system is now properly patched in memory.

- 6) Type TYPE "RES" Ø and press the RETURN key. This removes any protect attributes from the filetype of RES file.
- 7) Type SCRATCH "RES" and press the RETURN key. This deletes the old RES file from the system diskette.
- 8) Type SAVE "RES" 2B1 159B 3 and press the RETURN key. This writes the patched RES file onto the system diskette.

MICROPOLIS™