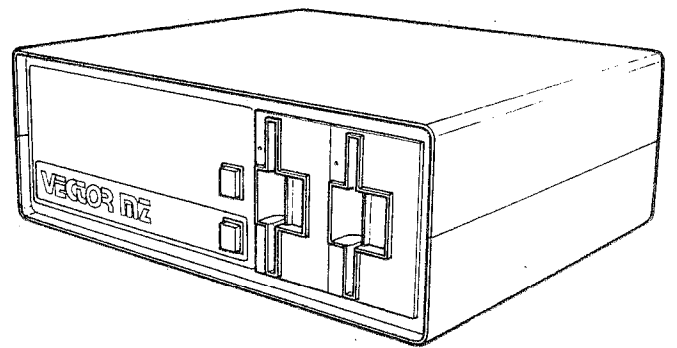
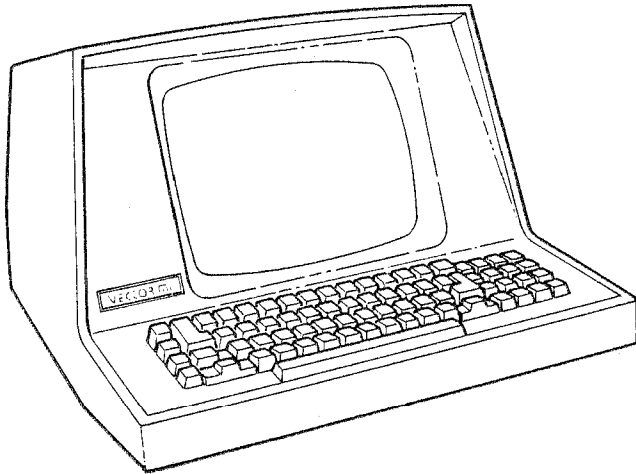


VECTOR GRAPHIC SYSTEMS
CORPORATION

VECTOR GRAPHIC SYSTEMS USING

MDOS



 VECTOR GRAPHIC INC.

To start up a system using MDOS, go directly to first page of Chapter 2.

USER'S GUIDE TO
VECTOR GRAPHIC SYSTEMS
USING MDOS

MDOS System Diskette version 8.4

USER'S GUIDE
Revision A
July 26, 1979

Copyright 1979 Vector Graphic Inc.

Copyright 1979 by Vector Graphic Inc.
All rights reserved.

Disclaimer

Vector Graphic makes no representations or warranties with respect to the contents of this manual itself, whether or not the product it describes is covered by a warranty or repair agreement. Further, Vector Graphic reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Vector Graphic to notify any person of such revision or changes, except when an agreement to the contrary exists.

Revisions

The date of release and revision letter of each page herein appears at the bottom of each page. Changes from the previous revision are marked with a bar in the margin. The revision letter such as A or B changes if the MANUAL has been improved but the PRODUCT itself has not been significantly modified. The date of release and revision letter on the Title Page corresponds to that of the page most recently revised. When the product itself is modified significantly, the product will get a new revision number, as shown on the manual's title page, and the manual will revert to revision A, as if it were treating a brand new product. EACH MANUAL SHOULD ONLY BE USED WITH THE PRODUCT IDENTIFIED ON THE TITLE PAGE.

REPAIR AGREEMENT

The Vector Graphic computer sold hereunder is sold "as is", with all faults and without any warranty, either expressed or implied, including any implied warranty of fitness for intended use or merchantability. However, the above notwithstanding, VECTOR GRAPHIC, INC., will, for a period of ninety (90) days following delivery to customer, repair or replace any Vector Graphic computer that is found to contain defects in materials or workmanship, provided:

1. Such defect in material or workmanship existed at the time the Vector Graphic computer left the VECTOR GRAPHIC, INC., factory;

2. VECTOR GRAPHIC, INC., is given notice of the precise defect claimed within ten (10) days after its discovery;

3. The Vector Graphic computer is promptly returned to VECTOR GRAPHIC, INC., at customer's expense, for examination by VECTOR GRAPHIC, INC., to confirm the alleged defect, and for subsequent repair or replacement if found to be in order.

Repair, replacement or correction of any defects in material or workmanship which are discovered after expiration of the period set forth above will be performed by VECTOR GRAPHIC, INC., at Buyer's expense, provided the Vector Graphic computer is returned, also at Buyer's expense, to VECTOR GRAPHIC, INC., for such repair, replacement or correction. In performing any repair, replacement or correction after expiration of the period set forth above, Buyer will be charged in addition to the cost of parts the then-current VECTOR GRAPHIC, INC., repair rate. At the present time the applicable rate is \$35.00 for the first hour, and \$18.00 per hour for every hour of work required thereafter. Prior to commencing any repair, replacement or correction of defects in material or workmanship discovered after expiration of the period for no-cost-to-Buyer repairs, VECTOR GRAPHIC, INC., will submit to Buyer a written estimate of the expected charges, and VECTOR GRAPHIC, INC., will not commence repair until such time as the written estimate of charges has been returned by Buyer to VECTOR GRAPHIC, INC., signed by duly authorized representative authorizing VECTOR GRAPHIC, INC., to commence with the repair work involved. VECTOR GRAPHIC, INC., shall have no obligation to repair, replace or correct any Vector Graphic computer until the written estimate has been returned with approval to proceed, and VECTOR GRAPHIC, INC., may at its option also require prepayment of the estimated repair charges prior to commencing work.

Repair Agreement void if the enclosed card is not returned to VECTOR GRAPHIC, INC. within ten (10) days of end consumer purchase.

FOREWORD

The User's Guide to Vector Graphic Systems Using MDOS covers a great deal of material, but it is broken into sections in order to encourage you to read and use it.

The following suggestions are provided to help users select those parts which are relevant to them:

- 1) Non-technical end-user operating solely a specific data processing application package, a totally "turn-key" system:

Use Section 2.1 on starting up a new system, Section 2.4 on the care and handling of diskettes, and all of Chapter 3 on normal daily procedures.

- 2) OEM or programmer doing programming, but whose hardware, including peripherals, is set up and running smoothly:

Add to the above material all of Chapter 1, which gives an overview of the system's logical structure, and Section 2.3 which deals further with the MDOS System Diskette. Then, use some or all of Chapters 4, 5, and 6, depending on the level of programming to be done.

- 3) Dealer, OEM, or technically-oriented end-user interested in manipulating the hardware capabilities of the system, interfacing various peripherals to a "naked" system, and also interested in programming:

The entire manual is relevant. Most important in addition to the above is Section 2.2, which describes how to interface peripherals, including what components to buy, and what to do when it all arrives. You may also find the separate technical manuals on system components useful, as well as the notes in Section 2.3 on hardware modifications.

There is of course much grey area between these categories. All users may find some of the more technical material useful.

This manual covers a version of the MDOS System Diskette that includes release 4.0 of MDOS and M.BASIC. Your dealer can provide you with a bulletin distributed by Vector Graphic describing all the differences and incompatibilities with release 3.0 of MDOS and M.BASIC, contained in the version 7 series of the MDOS System Diskette.

This manual does not deal directly with maintenance. However, Appendix Q explains the various memory diagnostic programs which come with your system.

No doubt improvements can be made to our products. We welcome your suggestions, both on the manual and the system. Direct them to:

Vector Graphic Inc.
31364 Via Colinas
Westlake Village, CA 91361

Software Catalog

The following software is currently available from Vector Graphic Inc.

MDOS/MBASIC	MZOS	CP/M	CCA Data Management System
CIS COBOL	EVIOS	UniVid	Extended Monitor
Unassembler	ZSM	Word Management System	

Following is a brief description of each one.

MDOS/MBASIC

MDOS is the powerful Micropolis Disk Operating System, standard with the Vector MZ. This package provides the user with a complete program development system. Included are an extended line editor, a full Z-80 linking assembler, a powerful debugger, and a symbol-saving program. Utilities provided include disk and file copying programs and many others. Interfacing routines for several types of printer are also provided. MBASIC, the Micropolis Extended Basic, is one of the most powerful BASICs in existence. Full record-oriented disk I/O, precision up to 60 digits, true CHAINing, and many other features make MBASIC a most desirable language to program and operate. Taken together, MDOS and MBASIC form an unmatched program development system. (OS)

MZOS

MZOS is our Micropolis/Z-80 Operating System. Although not nearly as powerful as MDOS, MZOS is fully North Star DOS compatible, as of release 3. Although it has many additional features, the general format of MZOS is similar enough to North Star DOS that there will be no difficulty in adapting to it. The real point, though, is that if you have already invested in North Star software, you will be able to protect your investment, since your software will run correctly on MZOS. In addition, you will have expanded capability, since a Micropolis disk has 3.5 times the capacity of a Shugart disk. We even provide a program to make an exact copy from a North Star diskette to a Micropolis diskette, providing that you have a North Star controller and drive. Please note, however, that Vector Graphic is not encouraging infringement of copyright on North Star software. (NB)

CP/M

CP/M is generally accepted as the industry standard operating system. With a powerful executive structure, and unmatched disk handling features, it is a logical choice. Additionally, CP/M has more software available for it from other sources than any other microcomputer operating system, including CBASIC, Fortran, and many others. Included in CP/M are a powerful text editor, assembler, debugger, peripheral interchange program, formatter, and many other utility programs. For business applications, CP/M, along with the collection of software available for it, is undoubtedly the logical choice. (OS)

CCA Data Management System

The CCA system is a breakthrough in data management systems. Useful for anything from mailing lists to inventory to accounting, this system has many features not found on any other. For each file in the system, the user defines all relevant information, including what each field is to be called, the length, and other information. This provides amazing flexibility. Maintenance features include the ability to add, delete, update, inspect, or search records. Sorting can be done on any field, and can be nested up to ten levels deep. Additionally, the sort package uses a special algorithm for the fastest possible sorting, and can sort files that are over a thousand records long. When printing reports, the user specifies exactly what fields are to be printed, and where; both report and label formats are possible. Subtotals and totals can be generated at certain points, and numeric fields may be printed in any format (justified, decimal aligned, for example). A portion of the full file may be printed by specifying a range, which can apply to any field. One particular unique feature is the ability to have computed fields; where one or more fields are computed on the basis of information in other fields. This opens even more additional possibilities for the system. Since file structuring is straightforward, other programs can easily be written to further enhance the system. It should be obvious that this is a package that everyone can use in at least one way. The CCA Data Management System runs under MDOS and MBASIC. (OS)

CIS COBOL

With CIS COBOL, the powerful business language of large computers is brought into the realm of the microcomputer. This is a Compact, Interactive, and Standard COBOL. Compact, because it will compile a sizable program easily in 32K; almost anything can be done on the MZ, with 48K. Interactive, because the traditionally batch COBOL language has been extended to allow screen-oriented interaction between the program and the user. This is a major enhancement to the normal language. Standard, because it generally follows the ANSI 74 standard. As a language, COBOL has many good points, including the fact that it is self-documenting and easy to read. CIS COBOL comes with both a language reference manual and operating instructions, and runs under CP/M. For business programmers, this language is a must. (OS)

EVIOS

EVIOS is Vector Graphic's Extended Video Input/Output System, designed for our Flashwriter I board. Since it is memory mapped, the Flashwriter has nearly infinite capability as a terminal; EVIOS is meant to make available many of these features. With this package, many so-called "intelligent" terminals are rivaled. Features like full cursor positioning, either relative or absolute, full control over reverse video, reduced intensity, and graphics, protected fields, creation of bar grids, screen transmission as input, extensive selective screen erasure, and several modes only begin to list the capabilities of this package. In general, there is just no competition for this system. (NB)

UniVid

UniVid is Vector Graphic's Universal Video Driver. While EVIOS is designed to take full advantage of the Flashwriter I board, UniVid is designed for the 24x80 Flashwriter II board. This system is called universal because it is actually three drivers in one. First, it is a simple driver that additionally allows plotting. Second, it is an emulator for the Lear Siegler ADM-3A terminal. Finally, it is an emulator for the Hazeltine 1500 terminal. Each of the emulation modes is to allow software that uses the special functions of one of the terminals to also work on the Mindless Terminal. Even if you don't need to maintain compatibility with another terminal, though, UniVid is still a wise investment. (LL)

Extended Monitor

Standard with Vector Graphic systems, the Extended Monitor is a multi-faceted piece of software. First, it is a system monitor, providing 23 commands for such functions as memory dumping, entering data into memory, testing memory, polling I/O ports, booting the disk, moving blocks of memory, and many, many others. In this way, it is invaluable as a debugging tool. Additionally, it provides all of the console input and output for the system. The Monitor is set up for the particular configuration of the system, and in that way all other software will automatically run on it, without system-dependent modifications. Although standard on Vector Graphic systems, the Extended Monitor is a valuable addition for any system. (RH)

The UnAssembler

For assembly language programmers, the UnAssembler is a necessity. Superceding the Sourcerer in our product line, the UnAssembler features full Z-80 disassembly, and enhanced error-catching features. Such features as both mnemonic and data word formats, production of a complete cross reference table, and an output that is ZSM-compatible make the UnAssembler an invaluable tool. The UnAssembler runs under MDOS. (NB)

ZSM

ZSM is Vector Graphic's powerful assembler. Structured similarly to Micropolis's assembler, ZSM also assembles all Z-80 code, using the 8080 superset (TDL) mnemonics for easy upgrading of 8080 programs. Some features of ZSM are the ability to request information from the user during assembly, the ability to print information during assembly, conditional assembly, and a very useful feature, the ability to link several files together for assembly. This allows otherwise huge or unmanageable files to be broken into as many source modules as desired. ZSM runs under MDOS, and is provided free of charge as part of MDOS with MZ systems or MDOS updates. (NB)

Word Management System

The Word Management System is a complete word processor, matching the most advanced stand-alone systems. Some major features include: a) continuous typing without regard for lines; b) freely changeable tabs; c) no typing of names of documents, especially good for assembly of large documents from several small ones; d) 1-step updating of the disk; e) typing directly to printer; f) character-oriented editing - typing anywhere on the screen; g) delete or move by character, word, line, or any other portion of text, causing automatic realignment of text on the screen; h) full document in working memory; i) scrolling by lines or full screens; j) global search and replace; k) original document on disk is not changed unless told to; l) complete formatting flexibility, including right-justification; m) mid-text (dynamic) format changes; n) variable page length; o) automatic heading, footing, and page numbering, with optional reversal of layout for two-sided printing; p) print-stop for changing type-face or ribbon color; q) continuous or page-by-page printing; r) stop printing in mid-document, even in mid-line; and t) many safety features to avoid erasing memory or disk.

The list processing feature of the system is in fact a small data management system, designed solely for mailing lists or similar lists of up to 1500 members. Each member of the list is assigned "qualifiers", which can be used to identify characteristics of that member. Using this, a subset of the entire list can be sorted, stored on disk for later use, and printed in 1 or 2 columns on mailing labels or continuous form paper. Moreover, by using the merge feature, mailing lists can be combined with text such as form letters, providing a sophisticated mass mailing capability.

The Word Management System contains its own disk operating system, and does not need any other. With the addition of this software, the Vector MZ computer system becomes an office-of-the-future staple. (ED)

TABLE OF CONTENTS

	<u>PAGE</u>
<u>SECTION I GENERAL INFORMATION</u>	
1.0 GENERAL DESCRIPTION OF SYSTEM AND SUBSYSTEMS	1-1
1.0.1 STANDARD HARDWARE AND SOFTWARE	1-1
1.0.2 OPTIONAL COMPONENTS AND SOFTWARE	1-2
1.1 MICROPOLIS DISKETTE SUBSYSTEM SPECIFICATIONS	1-2
1.1.1 PERFORMANCE	1-2
1.1.2 DRIVE RELIABILITY	1-2
1.2 HEXADECIMAL NOTATION	1-3
1.3 OPERATING SYSTEM SOFTWARE	1-3
1.3.1 VECTOR GRAPHIC EXTENDED SYSTEMS MONITOR	1-3
1.3.2 PROGRAM DEVELOPMENT SOFTWARE	1-4
1.3.3 ELEMENTS OF MDOS	1-4
1.3.4 ELEMENTS OF M.BASIC	1-5
1.3.5 OTHER OPERATING SYSTEMS	1-6
1.3.6 RESIDENT PROGRAMS	1-7
FIGURE 1.1 MZ SOFTWARE STRUCTURE USING MDOS	1-8
FIGURE 1.2 MEMORY MAP FOR VECTOR GRAPHIC SYSTEMS	1-9
FIGURE 1.3 MEMORY MAP FOR MDOS AND M.BASIC	1-10
1.4 MDOS SYSTEM DISKETTE	1-11 to 1-13
<u>SECTION II INSTALLATION, CONFIGURING PERIPHERALS, AND USE OF DISKETTES</u>	
2.1 INSTALLATION	2-1
2.2 CONFIGURING THE MZ (for non-turnkey systems)	2-2
2.2.0 MODIFYING THE RES MODULE	2-2
2.2.1 STANDARD CONFIGURATIONS	2-3
2.2.1.1 PRINTER: PARALLEL, CENTRONICS 700 SERIES PROTOCOL CONSOLE: SERIAL VIDEO TERMINAL	2-4
2.2.1.2 PRINTER: SERIAL, DIABLO 1610 PROTOCOL OR TELETYPE PROTOCOL CONSOLE: SERIAL VIDEO TERMINAL	2-4
2.2.1.3 PRINTER: PARALLEL, CENTRONICS 700 SERIES PROTOCOL CONSOLE: PARALLEL ASCII KEYBOARD, SEPARATE VIDEO MONITOR	2-6

	<u>PAGE</u>
2.2.1.4 PRINTER: PARALLEL, CENTRONICS 700 SERIES PROTOCOL CONSOLE: VECTOR GRAPHIC MINDLESS TERMINAL	2-7
2.2.1.5 PRINTER: SERIAL, DIABLO 1610 PROTOCOL OR TELETYPE PROTOCOL CONSOLE: PARALLEL ASCII KEYBOARD, SEPARATE VIDEO MONITOR	2-8
2.2.1.6 PRINTER: SERIAL, DIABLO 1610 PROTOCOL OR TELETYPE PROTOCOL CONSOLE: VECTOR GRAPHIC MINDLESS TERMINAL	2-9
2.2.1.7 SERIAL PRINTING TERMINAL (HAS A KEYBOARD), DIABLO 1610 OR TELETYPE PROTOCOL AND A VIDEO MONITOR	2-10
2.2.1.8 SERIAL PRINTING TERMINAL (HAS A KEYBOARD), DIABLO 1610 OR TELETYPE PROTOCOL AND NO VIDEO	2-12
2.2.2 ADDING A STANDARD PRINTER TO AN EXISTING MZ SYSTEM	2-13
2.2.3 NON-STANDARD CONFIGURATIONS	2-15
2.3 OTHER MODIFICATIONS TO SYSTEM SOFTWARE & HARDWARE	2-15
2.3.1 CHANGING TO 2 MHZ CLOCK RATE	2-15
2.3.2 CONNECTING ADDITIONAL DISK DRIVES	2-15
2.3.3 USING I/O PORTS	2-15
2.3.4 CHANGING MEMORY ADDRESS AND I/O PORT ASSIGNMENTS OF BOARDS	2-16
2.3.5 SHORTENING BASIC	2-16
2.3.6 BASIC-ONLY DISKETTE	2-16
2.3.7 STOPPING ESC FROM RETURNING CONTROL TO THE SYSTEMS MONITOR	2-17
2.3.8 FINALIZING THE PERSONALIZED SYSTEM DISKETTE	2-17
2.4 DISKETTE MEDIA	2-19
2.4.1 DESCRIPTION	2-19
2.4.2 IF YOU HAVE PROBLEMS WITH DISK ERRORS	2-19
2.4.3 HANDLING	2-19
2.4.4 LOADING AND UNLOADING	2-20
2.4.5 RECOVERY TECHNIQUES	2-21
2.4.6 REPLACEMENT AND BACK-UP OF DISKETTES	2-21
2.4.7 INITIALIZING DISKETTES	2-22
2.4.8 WRITE PROTECT FOR DISKETTES	2-22
FIGURE 2.1 5 1/4 INCH DISKETTE	2-23
FIGURE 2.2 HOW TO MOUNT WRITE PROTECT TAB	

	<u>PAGE</u>
<u>SECTION III DAY TO DAY OPERATIONS</u>	
3.0 SUMMARY OF NORMAL START UP PROCEDURE	3-1
3.1 SUMMARY OF PROMPTS	3-1
3.2 POWER-ON	3-1
3.3 LOAD MDOS	3-2
3.4 LOAD M.BASIC FROM MDOS	3-2
3.5 OTHER OPERATING SYSTEMS AND LANGUAGES	3-2
3.6 RETURNING TO MDOS FROM M.BASIC	3-3
3.7 RETURNING TO MONITOR FROM ANYPLACE	3-3
3.8 RETURNING TO MDOS (OR M.BASIC) FROM MONITOR IF MDOS (OR M.BASIC) IS ALREADY IN MEMORY	3-4
3.9 RETURNING TO MDOS OR M.BASIC EXECUTIVE FROM A ROUTINE RUNNING UNDER THAT EXECUTIVE	3-4
3.10 VIDEO COMMANDS	3-4
3.10.1 CLEAR SCREEN	3-4
3.10.2 SCROLL SCREEN UP ONE LINE	3-4
3.10.3 BACKSPACE CURSOR	3-4
3.10.4 CONVERT TO REVERSE VIDEO	3-5
3.10.5 TAB CURSOR 8 SPACES TO THE RIGHT	3-5
3.10.6 ELIMINATE CURSOR FROM THE SCREEN	3-5
3.10.7 MOVE CURSOR TO TOP OF SCREEN	3-5
3.10.8 MOVE CURSOR DOWN, UP, LEFT, OR RIGHT	3-5
3.10.9 RETURN CURSOR TO LEFT EDGE OF SCREEN	3-5
3.11 POWER-DOWN	3-5
<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
4.1.5.15 THE RENAME COMMAND	4-8
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 @NLINOUT - 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 @GFILESTAT - 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 @RFILEINF - READ FILE INFORMATION	4-27
4.3.3.7 @SINXTRS - SET INDEX POSITION TO RECORD START	4-28
4.3.3.8 @RRECORDLEN - 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
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>
4.4.8 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 ZSM - Z-80 ASSEMBLER	4-55
4.5.1 HOW TO RUN ZSM	4-55
4.5.2 LANGUAGE ELEMENTS	4-56
4.5.2.1 CONSTANTS	4-57
4.5.2.2 OPERATORS	4-58
4.5.2.3 REGISTERS	4-58
4.5.2.4 PSEUDO-OPS	4-59
4.5.3 ASSEMBLY ERRORS	4-63
4.5.4 INSTRUCTION SET	4-64
4.5.5 TEST FILE FOR ZSM	4-67E

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

SECTION V MICROPOLIS DISK EXTENDED BASIC

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	FLOADG	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>PAGE</u>
<u>SECTION VI DISK SUBSYSTEM THEORY AND DIRECT PROGRAMMING</u>	6-1
FIGURE 6.1 5 1/4 INCH DISKETTE	6-1
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

APPENDICES

- A - BASIC ERROR MESSAGES
- B - BASIC UTILITY PROGRAM
- C - ACCESSING DISKCOPY FROM BASIC
- D - SUMMARY OF MDOS ERROR MESSAGES
- E - RES.I/O SOURCE LISTING
- F - MICROPOLIS DISK BOOTSTRAP
- G - "FEATURES" PROGRAM TO OPTIONALLY SHORTEN BASIC
- H - INTERFACING TO A CENTRONICS PRINTER
- I - TROUBLE SHOOTING IF MDOS DOES NOT LOAD
- J - GAMES AND DISPLAYS ON THE MDOS SYSTEM DISKETTE
- K - CHANGING MICROPOLIS BOOTSTRAP ROM AND DISK I/O ADDRESS
- L - CHANGING CLOCK RATE TO 2 MHZ
- M - WRITING A CONSOLE PHYSICAL I/O ROUTINE
- N - WRITING A PRINTER PHYSICAL I/O ROUTINE
- O - REASSEMBLING AND SAVING THE RES MODULE
- P - MAP OF I/O PORTS
- Q - MEMORY DIAGNOSTICS

I GENERAL INFORMATION

1.0 GENERAL DESCRIPTION OF SYSTEM AND SUBSYSTEMS

Your system is a general purpose microprocessor based computer. It is delivered by Vector Graphic completely assembled and fully tested, including both hardware and operating system software, and including two quad density mini-floppy disk drives.

1.0.1 STANDARD HARDWARE AND SOFTWARE

- 1) Chassis with power supply and 18 slot fully shielded S-100 motherboard;
- 2) 4 MHz Z-80 CPU board;
- 3) Two quad density Micropolis mini floppy disk drives, allowing 1232 256-byte sectors per diskette.
- 4) Disk controller board;
- 5) Bitstreamer I/O board;
- 6) 48K Dynamic RAM board;
- 7) PROM/RAM III board, with space for 12K of EPROM and the ability to program EPROM's (see the PROM/RAM III board manual).
- 8) The Vector Graphic Extended Systems Monitor, on PROM;
- 9) Two copies of the MDOS System Diskette, each containing:
 - a) The Vector Graphic-enhanced Micropolis Disk Operating System - MDOS - a complete floppy diskette operating system, including a Z-80 Assembler, an editor, a debugger, and several other utilities (see Ch.4);
 - b) Micropolis BASIC (see Ch. 5);
 - c) A number of games and video displays (see Appendix J.)

1.0.2 OPTIONAL COMPONENTS AND SOFTWARE

Your MZ can be configured with various optional peripherals. Section 2.2 of this manual lists the configurations of printers and consoles considered "standard" for the MZ, and gives the components such as interface boards and cables needed for each configuration. In addition to the configurations discussed in Section 2.2, the following components can optionally be added to an MZ:

- 1) Additional Bitstreamer I/O board(s), such as the Bitstreamer II having three serial ports, two parallel ports, real-time

- clock, and Z-80 interrupts.
- 2) Additional memory board(s);
 - 3) Other S-100 compatible boards from Vector Graphic or other sources.
 - 4) 2 additional Micropolis mini-floppy disk drives;
 - 5) Other operating system and language software.

Contact your dealer for more information on adding components to the system.

1.1 MICROPOLIS FLOPPY DISKETTE SUBSYSTEM SPECIFICATIONS

1.1.1 PERFORMANCE

Capacity per drive: 315K bytes, formatted
Transfer rate: 250K bits/second
Average rotational latency time: 100 milliseconds
Access time - track-to-track : 30 milliseconds
settling time: 10 milliseconds
Head load time: 75 milliseconds
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)
Recording mode: MFM
Track density: 100 tracks per inch (TPI)
Surfaces used per diskette: 1

1.1.2 DRIVE RELIABILITY

MTBF	8000 hrs.
MTTR	0.5 hrs.
Media life	3 X 10 EXP 6 passes on single track
Head life	10 EXP 4 hrs.
Soft error rate	1 in 10 EXP 9
Hard error rate	1 in 10 EXP 12
Seek error rate	1 in 10 EXP 6

1.2 HEXADECIMAL NOTATION

In this manual as in most microcomputer literature, the base 16 number system is used for all references to memory locations, instruction codes, character codes, and so on. If you are not familiar with it, you will soon find that the hexadecimal system is the most natural way to express these numbers when dealing with a computer that stores data as groups of 8 binary digits (bits) and memory addresses as groups of 16 bits. Hex numbers will be indicated by an upper case H following the digits. Remembering a few key values will make things a great deal easier:

<u>HEX NUMBER</u>	<u>DECIMAL VALUE</u>	<u>JARGON</u>	<u>BINARY BITS</u>
A	10		4
B	11		4
C	12		4
D	13		4
E	14		4
F	15		4
10	16		5
FF	255		8
100	256		9
3FF	1,023		10
400	1,024	1K	11
FFF	4,095		12
1000	4,096	4K	13
4000	16,384	16K	15
8000	32,768	32K	16
FFFF	65,535	64K-1	16

The familiar rules of arithmetic work just the same in hex as in decimal:

$$\begin{array}{r} \text{or} \\ 40) \overline{10} \\ \underline{40} \\ 400 \end{array} \quad \text{HEX (TRIVIAL)}$$
$$\begin{array}{r} 64) \overline{16} \\ \underline{64} \\ 1024 \\ \underline{64} \\ 384 \\ \underline{384} \\ 0 \end{array} \quad \text{DECIMAL (MORE DIFFICULT)}$$

1.3 OPERATING SYSTEM SOFTWARE

1.3.1 VECTOR GRAPHIC EXTENDED SYSTEMS MONITOR

The first program the user comes into contact with after turning on the system is the Vector Graphic Extended Systems Monitor. (Exception: this is not true for MEMORITE systems.) It is entirely stored on non-volatile PROM. Note that this use of the term "Monitor" has a meaning entirely different than the term "monitor", which refers to a piece of hardware, namely a stand-alone video display. (NOTE: in the MEMORITE system, the Extended Systems

Monitor is not encountered unless you press the RESET key; or touch the ESC key while the system is under control of MDOS or another NON-word processing operating system.)

The Monitor consists of two parts: first, the Extended Systems Monitor Executive, which allows the operator, through special commands, to manipulate and display memory data and to jump to some other program; second, a program used to control console I/O.

You know the Extended Systems Monitor Executive is in control of the system when the Monitor prompt (*) appears on the left edge of the screen. The operator is then expected to enter one of the commands available for manipulating or displaying memory or jumping to another program. Most often, the operator will use the command which calls up a full operating system and then transfers control to it, and out of the Monitor.

Regardless of whether executive is in control of the system at any given time, the Monitor console I/O routines, though invisible to the operator, are continually being called on to control the console. (Exception: when MEMORITE or the Word Management System are doing word processing, the Monitor is not used to control the console. Instead, the word processing software in these two systems handles this task.)

Some of the Monitor's features and commands are explained where relevant in this manual. A complete description is included as a separate manual with your system.

1.3.2 PROGRAM DEVELOPMENT SOFTWARE - "PDS"

The operating system found on the MDOS Systems Diskette included with the system is the Micropolis Diskette Operating System (MDOS). MDOS includes an assembly language program development package. Also found on the MDOS Systems Diskette is Micropolis Disk Extended BASIC (often called just M.BASIC). MDOS and M.BASIC together give all the functions a programmer may need for the development of either assembly language or BASIC programs.

1.3.3 ELEMENTS OF MDOS

MDOS consists of an executive program, a group of "shared" subroutines available to user programs as well as being used by MDOS, and various utilities which include assembly language program development tools.

The MDOS executive program 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.

The shared subroutines include those that 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 interger arithmetic. There are also a number of processor oriented utility subroutines.

The MDOS utilities are:

ZSM - a two pass, 8080/8085/Z80 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 an exact 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 by the ZSM assembler.

DEBUG - a utility that facilitates checkout and debugging of 8080/8085 machine language programs. It cannot be used if Z80 code which is not part of the 8080 set is used.

1.3.4 ELEMENTS OF M.BASIC

M.BASIC is a complete, self-contained software package that provides total support for BASIC programming. When M.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.

M.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/U to cancel an input line, and FLOW and NOFLOW to enable and disable the flow trace debugging aid.

M.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.

M.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 accessed by 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, 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 M.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.

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.

Also provided is a BASIC Utility program that provides for initializing diskettes, saving M.BASIC on a BASIC-only diskette, and examining and changing RAM memory. In addition, there is a utility called FEATURES which allows you to shorten M.BASIC by eliminating some of the features needed only for program development, but not for running production programs.

1.3.5 OTHER OPERATING SYSTEMS

Other operating systems and higher level languages are available

from Vector Graphic. These will not be discussed here. (See the literature accompanying this manual.) MDOS and M. BASIC meet the needs of the large majority of users.

1.3.6 RESIDENT PROGRAMS

MDOS and M.BASIC share the Extended Systems Monitor. They also share a common program module called RES. This module contains among other routines, the printer and diskette I/O routines, and some of the console I/O routines.

Also shared is the ROM resident Disk Bootstrap program, (which is what the Monitor uses in order to call up MDOS), and the Disk Controller, (which is simply memory space needed to handle the diskette drives.)

These routines are always resident in the computer memory when either MDOS or M.BASIC is running. For interested users, listings will be found in Appendix E for the I/O portion of RES, Appendix F for the Disk Bootstrap program, and the Extended Systems Monitor manual for the Monitor.

In contrast, MDOS and M.BASIC overlay each other; that is, they are assigned the same area of memory; only one can be in memory at any given time. Commands are provided for leaving one and calling up the other.

Fig. 1.1 illustrates the relationships between the various system programs. Programs which are always in memory when MDOS or M.BASIC is used are in the center.

Fig. 1.2 gives the addresses of the various programs and important memory locations in your system. No particular operating system is shown.

Fig. 1.3 gives addresses for MDOS and M. BASIC. Note that this operating system software fits into the unassigned memory area in Fig. 1.2.

FIGURE 1.1 MZ SOFTWARE STRUCTURE USING MDOS

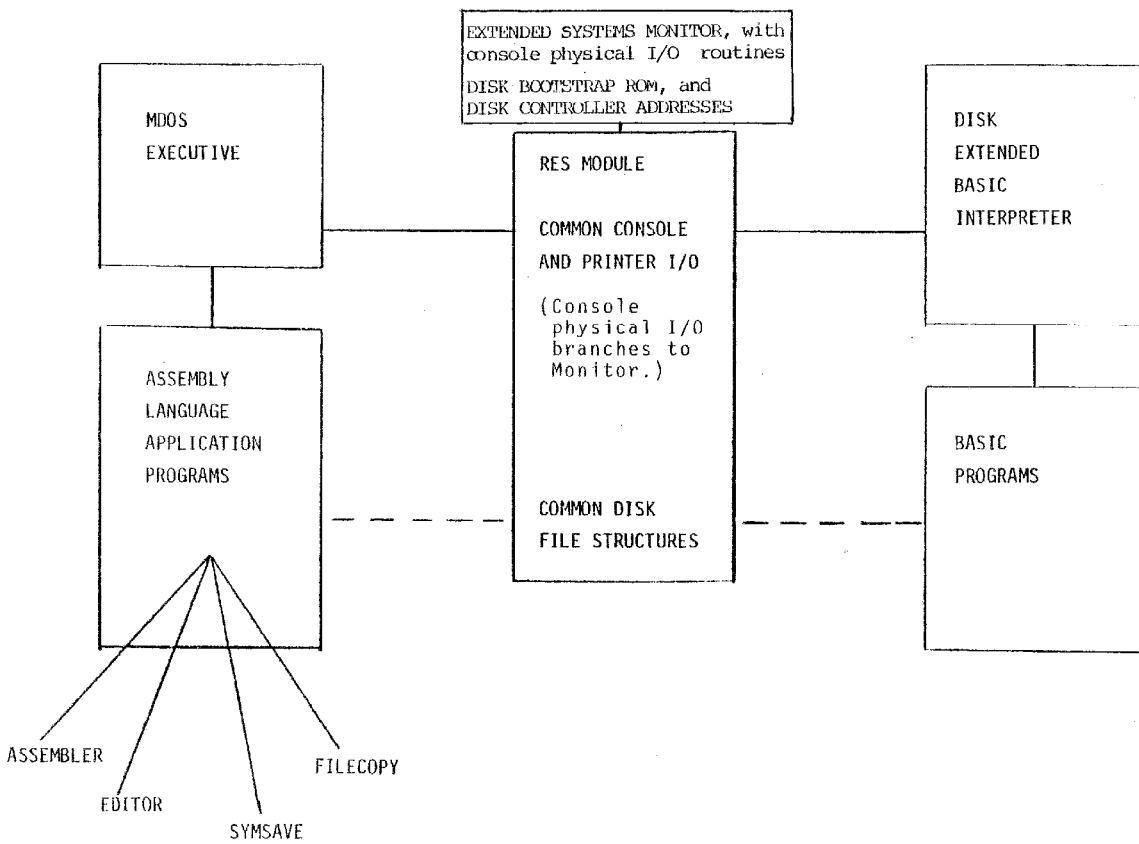


FIG 1.2 MEMORY MAP FOR VECTOR GRAPHIC SYSTEMS

Hex address	Contents
FFFF	8K RAM for user's programs, optional; OR High Resolution Video board, optional; OR Memorite PROM's, optional.
E000	
DF40	PR-2 stack area, not available to user.
DC00	RAM available to user.
DA00	Disk controller - first 3 bytes are addresses used for mem. mapped disk I/O. Remaining are unusable.
D800	Disk Bootstrap ROM.
D000	Flashwriter board video buffer, optional.
CC00	Memorite configuration PROM, optional.
C800	EVIOS PROM, optional.
C400	MZOS PROM, optional.
C000	Extended Systems Monitor, including console I/O routines.
8000	48K RAM, available to user.
0000	

FIG 1.3 MEMORY MAP FOR MDOS AND M. BASIC

Hex address	Contents
BFFF	RAM memory for user's program
2B00 if MDOS 5700 to 5D86 if BASIC	
	Starting point depends on whether MDOS or M. BASIC is being used, and whether BASIC has been shortened.
	MDOS, including all user callable routines not in RES Module; OR M. BASIC Interpreter.
1599	RES MODULE
1598	End of RES Module.
0627	LDOUT - Physical List Output Routine.
0613	LDINIT - Physical List Initialization.
0611	LDATN - Physical List Attention Check Routine.
060F	CDINIT - Physical Console Initialization.
0604	CDBRK - Physical Break Check Routine.
0600	CDOUT - Physical Console Output Routine.
05F8	CDIN - Physical Console Input Routine.
04E7	MDOS or M. BASIC warmstart (entry) location.
02B1	Beginning of RES Module code.
01A0	Beginning of RES Module input buffer.
01A0	MDOS system stack, and used by Boot loader.
006A	
0000	RAM available to user.

1.4 MDOS SYSTEM DISKETTE

This revision of the User's Guide to Vector Graphics Systems Using MDOS corresponds to MDOS System Diskette 8.4, (and minor revisions of it labeled 8.4.1, 8.4.2, etc.) Following is a list of the files on this diskette:

(Under TYPE, "EC" means the file is stored in executable machine language code and it will be executed immediately if you type its name after the MDOS prompt. "AL" means the file is stored in assembly language source code. You must first assemble it using ZSM before it can be executed by the computer. "B" means the file is stored in the M.BASIC language. It will be executed by using the M.BASIC interpreter explained in chapter 5.)

<u>NAME</u>	<u>DESCRIPTION</u>	<u>TYPE</u>
DIR	The disk directory.	
RES	Machine language routines used by both MDOS and M.BASIC. Do not delete it unless you are modifying it.	
MDOS	MDOS executive and disk I/O routines. Do not delete this. See Appendix B to create a BASIC-only diskette.	
BASIC	M.BASIC interpreter and disk I/O. See Chapter 5.	EC
LINEEDIT	Line editor for writing assembly language programs. See Section 4.4.	EC
ZSM	Assembler of Z-80 code prepared in extended 8080 mnemonics. See Section 4.5.	EC
SYMSAVE	Utility which creates a source file of equate statements using the symbol table resulting from an assembly. See Section 4.6. Used occasionally by assembly language programmers.	EC
FILECOPY	Utility for copying a file from one drive to another. See Section 4.7. Used often.	EC
DISKCOPY	Utility for copying a disk from one drive to another. See Section 4.8. Used often.	EC
COPYFILE	Utility for copying a file from one disk to another, using the SAME drive, for systems having only one drive. See Section 4.10.	EC
DEBUG-GEN	Utility used to generate the DEBUG utility residing in a particular portion of memory. See Section 4.11.	EC
FEATURES	Utility used to shorten BASIC. See Appendix G.	EC

NAME	DESCRIPTION	TYPE
SYSQ1, and SYSQ2	Assembly language source code containing the names of all MDOS shared subroutines, equated to their addresses. Used in assembly language programs calling those routines. See Section 4.3. Used from time to time by assembly language programmers.	AL
UTILITY	A utility used to initialize diskettes, create BASIC-only diskettes, and examine memory. See Appendix B.	B
RES.I/O	The source code file of the I/O routines in RES. Used to rewrite the I/O routines if using non-standard peripherals. See Appendices M, N, and O.	AL
DIAB	Routine for interfacing to Diablo-protocol printers if the Bitstreamer board is addressed for ports 0 - 3. Overlays directly over RES in memory. See Section 2.2.2. Not needed after RES is saved on diskette.	EC
DIAB4	Same as DIAB, but Bitstreamer is at 4 - 7.	EC
CENT	Same as DIAB, but for Centronics printers.	EC
CENT4	Same as CENT, but Bitstreamer is at 4 - 7.	EC
DECW	Same as DIAB, but for teletype-protocol printers.	EC
DECW4	Same as DECW, but Bitstreamer is at 4 - 7.	EC
SAVERES	Utility used to save on disk the machine language version of the I/O portion of the RES Module. See Section 2.2.0. Not needed after the RES Module is finalized.	EC
NOESCAPE	Utility which stops the <u>ESC</u> key from causing control to be passed to the Systems Monitor. See Section 2.3.7. Not needed after used once.	EC
MDIAG	Utility used to check the computer's memory. See Appendix Q. Do not delete this.	EC
MAP	Utility which tells what kind of memory (RAM, ROM or nothing) is in the system at each address. See Appendix Q. Useful when servicing a system.	EC
FLASH7	Demonstration of the graphics capability of the Flashwriter II board. See Appendix J. Dealers use often.	EC
PROM	Utility used with the PROM/RAM III board to program EPROM's. See PROM/RAM III manual.	EC
STARTREKG	The Star Trek game. See Appendix J. Dealers use often. Others if they like it.	B
CIVILWAR	Another game. See Appendix J.	B
LUNAR	Another game. See Appendix J.	B
FINANCE	Day-to-day financial calculations. See Appendix J. Used often if you need it.	B

To obtain a list of the files on your diskette, to see what is actually there, turn the machine on, mount the system diskette in drive 0 (right-hand drive), type B after the Monitor prompt (*), type FILES after the MDOS prompt (>), and then press the RETURN key. The interaction looks like this on the screen:

```
*B
Vector MZ MDOS X.XX
>FILES
DIR      03  0000
RES      03  0014
.
.
.
```

The left-hand number refers to the file type, explained in Section 4.2.3. The right-hand number gives the length of the file in sectors. Both numbers are in hexadecimal (base 16).

The list is long and will roll past the edge of the screen. To stop it at any point, depress control-S (CTRL key and S at the same time.) To start it up again, depress the spacebar.

If you have a printer which is up and running with your system, you can print the directory by typing ASSIGN 2,3 (return), before you type FILES. After the directory is printed, type ASSIGN 2,2 (return) to turn the printer off again. "(return)" means press the RETURN key.

II INSTALLATION, CONFIGURING PERIPHERALS, AND USE OF DISKETTES

2.1 INSTALLATION

For turn-key systems (that is, all internal wiring and software modifications have been done prior to delivery), just plug in external cables to the sockets on the rear panel of the mainframe. End users: if sockets are not labeled and choice is not obvious, ask your dealer.

For non-turn-key systems, refer to Section 2.2 for directions on setting up peripherals, interface boards, cables, and interface software. For systems with which a printer will be used, it may be desirable to first set the system up as if there were no printer, test it as explained below, then complete the setting up procedures for the printer. Section 2.2 separates the 2 stages.

When ready to test the system, do as follows:

1. Turn the power key on the front panel and then turn on peripherals. The Monitor prompt * should appear on the screen. (Exception: in MEMORITE systems, depress RESET on the front panel after turning the power on. The Monitor prompt should then appear.)
2. Enter N on the keyboard. This is a memory test which also functions as a test of the console. After a few seconds a hexadecimal number should appear. It indicates the first memory address where no memory hardware is located. In normal systems with 48K of RAM, the number should be C000.
3. Insert and mount the MDOS Personalized System Diskette in drive 0. Drive 0 is the right-hand drive. The left-hand drive is drive 1. Refer to Section 2.4 for how to insert, mount, and in general handle diskettes.
4. Enter B. This causes MDOS to be loaded and take control. This will be indicated by the MDOS sign on message and the MDOS prompt: >.
5. To test a separate printer, if any, first make sure there is paper in the printer. Then, enter ASSIGN 2,3 (return), followed by FILES (return). (The expression (return) always means "press the RETURN key."). A list of the files on the System Diskette will be printed.

When the system is working properly, refer to Chapter 3 for a complete description of normal operating procedures, and to Section 2.4 for instructions on the handling and maintenance of diskettes. Do not neglect either Section 2.4 or Chapter 3 as they contain information which is not effectively acquired by trial and error alone. Section 2.3 describes various modifications which can be made to the hardware.

alone. Section 2.3 describes various modifications which can be made to the hardware and systems software.

2.2 CONFIGURING THE MZ - THIS SECTION FOR NON-TURN-KEY SYSTEMS ONLY

2.2.0 MODIFYING THE RES MODULE

At various points in this chapter (or in related appendices) you will be instructed to carry out procedures which modify the RES Module. The most common of such procedures are the Software Implementation Procedures found in section 2.2.1 under each of the standard configurations. (These Software Implementation Procedures are used only if a printer is implemented.)

To carry out any procedure which modifies the RES Module, turn the system and all peripherals on. In MEMORITE systems, depress the RESET button next. Then insert and mount the Personalized MDOS System Diskette in drive 0. Do not use the Master MDOS System Diskette. This diskette should never be altered and only used for emergency back-up. After the Monitor prompts with *, enter B. This "boots up" MDOS, as indicated by the MDOS sign-on message and MDOS prompt: >. Now proceed with the given procedure.

Note that in all software procedures, "(return)" means "press the RETURN key."

The user may be instructed to enter a command, such as DIAB4 (return). Whenever such a command is entered, the system will respond by displaying the MDOS sign-on message again, or at least the MDOS prompt >.

A step will be found which commands "Save the RES Module on Personalized System Diskette." This is accomplished as follows: Make sure the Personalized MDOS System Diskette is inserted and mounted in drive 0. Then under MDOS type SAVERES (return). The drive should write on the diskette. The RES Module is now saved on the Personalized MDOS System Diskette.

Important: You may want to do several different procedures, each of which terminates with saving the RES Module. You are definitely free to do any group of them at one sitting, and then save the RES Module as described above ONCE at the end of the session, in order to save trouble. Alternately, you may of course save the RES Module after each such procedure, if desired.

Note: SAVERES is a utility which saves on diskette the I/O portion of the RES Module, in machine language form. The block of code which is saved corresponds to the code found in the source listing called RES.I/O, plus a few bytes before and after. In the rare case you have modified the RES Module outside of the I/O portion, then you must use the following alternate steps to save the RES Module: Under MDOS, enter TYPE "RES" Ø (return) SCRATCH "RES" (return) SAVE "RES" 2B1 1598 3 (return).

2.2.1 STANDARD CONFIGURATIONS

At this time, Vector Graphic supplies the interface hardware and software to support several different configurations of main peripheral devices, that is, printers, keyboards, video displays, and terminals. This section is concerned with identifying these standard configurations, and explaining how they are implemented.

If the peripheral device desired is not found among the standard configurations, refer to Section 2.2.3.

The information is collected in the following pages. Each section is concerned with one configuration. Each configuration is a selected group of peripherals. Peripherals are listed as generic types, (upper case lettering). Specific makes are given as examples, (lower case lettering). The user is not limited to these examples, but can use any model that falls within the given generic description.

To use these charts, find the configuration desired. When ordering an MZ or other Vector Graphic computer, order it with the components listed as well as the peripherals desired if supplied by Vector Graphic. (Since all systems are always delivered with one Bitstreamer board and an I/O cable, do not explicitly order these items.)

If no printer is being used, find the desired configuration ignoring the type of printer listed. For this purpose, refer only to those configurations whose headings are NOT preceded by asterisks(*). Then, only order the parts and carry out the steps shown WITHOUT asterisks.

If a printer is being added to an existing system, find the desired configuration, then only order the parts and carry out the steps shown WITH an asterisk (*). To obtain a useful summary of the issues involved with printers, see section 2.2.2

Some systems may already be partially configured at the factory or by intermediaries, so that you need order and set up only the components not already included. For example, "System B" is an MZ with the Vector Graphic Mindless Terminal and Flashwriter II board. All you have to add is a printer. Your choices would be the configurations in Sections 2.2.1.4 and 2.2.1.6 for Centronics or Diablo-type printers respectively. MEMORITE is even simpler than a System B. Just do the Software Implementation procedure in Section 2.2.1.6, using the DIAB4 command.

Flashwriter Board: The charts refer to a "Flashwriter Board." Order a Flashwriter I for 16 x 64 display and Flashwriter II for 80 X 24 display. When ordering an Extended Systems Monitor for use with one of these boards, always state which it is for.

When your system and/or components are delivered, refer again to the chart. Perform the implementation procedures listed in order to implement the desired configuration.

* 2.2.1.1 Printer: PARALLEL, CENTRONICS 700 SERIES PROTOCOL.
Console: SERIAL VIDEO TERMINAL.

Example: Parallel Centronics matrix printer (700 Series), and Hazeltine terminal.

Interface Components Required

1. Option C Extended Systems Monitor, on PROM.
- * 2. Centronics interfacing kit
3. Bitstreamer board and I/O cable (no need to order; included in system automatically.)

Hardware Implementation Procedures

- * 1. Install the Centronics interfacing kit as instructed in Appendix H. Make sure there is an I/O cable connected at one end to J3 on the Bitstreamer board and at the other end installed in one of the cutouts at the rear of the mainframe.
2. Plug the external terminal cable into the socket on the rear of the mainframe which is wired to the 6 pin molex connector on the Bitstreamer board.
- * 3. Plug the printer cable into the socket which is wired to J3 on the Bitstreamer board.

Software Implementation Procedures

- * 1. Under MDOS, enter CENT (return).
- * 2. Save RES Module on Personalized System Diskette.

2.2.1.2 Printer: SERIAL, DIABLO 1610 OR TELETYPE PROTOCOL.
Console: SERIAL VIDEO TERMINAL.

Example: Printer: if Diablo protocol - Diablo 1610 or 1620, Qume Sprint 5, or NEC Spinwriter; if Teletype protocol - Decwriter, Teletype, or TI 810 or 820.
Console: Hazeltine terminal.

Interface Components Required

1. Option C Extended Systems Monitor, on PROM
2. Bitstreamer board and I/O cable (no need to order; included in system automatically.)
- * 3. A second Bitstreamer board
- * 4. A second I/O cable

Hardware Implementation Procedures

- * 1. Jumper one of the Bitstreamer boards so that it is readdressed for ports 4 - 7 rather than the original 0 - 1. Instructions will be found in the Bitstreamer User's

Manual. This board will be used to control the printer.

- * 2. Make sure that the printer is set for its highest speed, (1200 baud for Diablo 1610 protocol), and that its parity setting is MARK parity. Check the printer manual if necessary. Some printers such as the Diablo require a jumper on internal circuitry to increase from 300 baud to 1200 baud.
- * 3. Make sure that the Bitstreamer board is set for the same speed as the printer. This is set on a dipswitch on the upper left-hand corner of the board. Press the appropriate switch in and upward and make sure all other switches are pressed downward.
- 4. Connect one of the I/O cables to J3 on one of the Bitstreamer boards. Install the 25 pin socket on the other end of the cable in a cutout at the rear of the mainframe.
- * 5. Do step 4 for the second Bitstreamer and I/O cable.
- * 6. Plug the printer cable into the socket connected to the readdressed Bitstreamer.
- 7. Plug the terminal cable into the socket connected to the normal Bitstreamer. **IMPORTANT:** Some terminals will not operate if they are connected to all 25 pins, because some of the pins of J3 on the Bitstreamer have functions other than serial communications. If your terminal does not operate after connecting it to all pins, then connect only the essential ones. Example: the Hazeltine 1400 will function only if a 3-line cable is used, connecting pins 2, 3, and 7. A 25 pin ribbon connector will not work. Other terminals may require additional pins, but again not all 25. Refer to the Bitstreamer board manual if necessary for definitions of each of the pins on the backpanel connector.

Software Implementation Procedures

- * 1. Under MDOS, if printer uses Diablo protocol, enter DIAB4 (return); if printer uses Teletype protocol, enter DECW4 (return).
- * 2. Save RES Module on Personalized System Diskette.

- * 2.2.1.3 Printer: PARALLEL, CENTRONICS 700 SERIES PROTOCOL
Console: PARALLEL ASCII KEYBOARD, SEPARATE VIDEO MONITOR.

Example: Printer: Parallel Centronics matrix printer (Series 700)
Console: Vector Graphic stand-alone parallel keyboard and Hitachi video monitor.

Interface Components Required

1. Option EV Extended Systems Monitor on PROM
2. Flashwriter board
3. I/O cable
4. Video cable, for Flashwriter to rear panel
5. Video monitor to mainframe cable
- * 6. Centronics interface kit
- * 7. Bitstreamer board with I/O cable (no need to order; included in system automatically.)

Hardware Implementation Procedures

- * 1. Jumper the Bitstreamer board so that it is readdressed for ports 4 - 7 rather than the original 0 - 1. Instructions will be found in the Bitstreamer User's Manual.
- * 2. Install the Centronics interfacing kit as instructed in Appendix H. However, do not install the 6 pin molex connector or the serial I/O cable which come in the Centronics interface kit. They are not needed and can be set aside. Make sure that there is a regular I/O cable connected to J3 on the Bitstreamer board and installed at the other end in a cutout at the rear of the mainframe. This socket will be used for the printer cable.
3. Connect the 2 pin socket at one end of the video cable to the 2 left-most pins which will be found rising vertically from the left-hand corner of the Flashwriter board. The socket should be positioned so that the inside wire is connected to pin 1, and the outside "shield" wire is connected to pin 2 (ground). Install the circular socket at the other end of the cable into one of the circular cutouts at the rear of the mainframe.
4. Connect the 24 pin dip plug at one end of the second I/O cable to J1 on the Flashwriter board. Install the 25 pin socket at the other end in one of the cutouts at the rear of the mainframe. This socket will be for the keyboard cable.
- * 5. Plug the printer cable into the appropriate sockets on the rear of the mainframe.
6. Plug the external keyboard and monitor cables into the appropriate sockets on the rear of the mainframe.

Software Installation Procedures

- * 1. Under MDOS, enter CENT4 (return).
 - * 2. Save RES module on Personalized System Diskette.
- * 2.2.1.4 Printer: PARALLEL, CENTRONICS SERIES 700 PROTOCOL.
Console: VECTOR GRAPHIC MINDLESS TERMINAL.

Example: Parallel Centronics matrix printer (Series 700) and Vector Graphic Mindless Terminal.

Interface Components Required

1. Option EV Extended Systems Monitor on PROM
2. Flashwriter board
3. Mindless Terminal 3-part I/O cable
4. External Mindless Terminal cable (or equivalent)
- * 5. Centronics interface kit
- * 6. Bitstreamer board with I/O cable (no need to order; included in system automatically.)

Hardware Implementation Procedures

- * 1. Jumper the Bitstreamer board so that it is readdressed for ports 4 - 7 rather than the original 0 - 1. Instructions will be found in the Bitstreamer User's Manual.
- * 2. Install the Centronics interfacing kit as instructed in Appendix H. However, do not install the 6 pin molex connector or the serial (3 wire) I/O cable which come in the Centronics interface kit. They are not needed and can be set aside. Make sure that there is a regular I/O cable connected to J3 on the Bitstreamer board and installed at the other end in a cutout at the rear of the mainframe. This socket will be used for the printer cable.
- 3. If not already done at the factory, install the Mindless Terminal 3-part I/O cable as instructed in the terminal's documentation. The 3 parts are connected to the power supply, the Flashwriter board video output pins, and the Flashwriter board keyboard input socket (J1). At the other end, the DB25 socket is installed in one of the cutouts at the rear of the mainframe.
- * 4. Plug the printer external cable into the respective socket at the rear of the mainframe.
- 5. Plug the terminal external cable into the respective socket at the rear of the mainframe.

Software Installation Procedures

- * 1. Under MDOS, enter CENT4 (return).

* 2. Save RES module on Personalized System Diskette.

2.2.1.5 Printer: SERIAL, DIABLO 1610 or TELETYPE PROTOCOL
Console: PARALLEL ASCII KEYBOARD, SEPARATE VIDEO MONITOR.

Example: Printer: if Diablo protocol - Diablo 1610 or 1620, Qume Sprint 5, or NEC Spinwriter; if Teletype protocol - Decwriter, Teletype, or TI 810 or 820.

Console: a Vector Graphic stand-alone parallel keyboard and Hitachi video monitor.

Interface Components Required

1. Option EV Extended Systems Monitor on PROM
2. Flashwriter board
3. I/O cable
4. Video cable, Flashwriter to rear panel
5. Video monitor to mainframe cable
- * 6. Bitstreamer board with I/O cable (no need to order; included in system automatically.)

Hardware Implementation Procedures

1. If no printer is being used, remove the Bitstreamer from the mainframe, and do not put it back in. It cannot be in the system (unless readdressed as explained below.)
- * 2. Jumper the Bitstreamer board so that it is readdressed for ports 4 - 7 rather than the original 0 - 1. Instructions will be found in the Bitstreamer User's Manual.
- * 3. Make sure that the printer is set for its highest speed, (1200 baud for Diablo 1610 protocol), and that its parity setting is MARK parity. Check the printer manual if necessary. Some printers such as the Diablo require a jumper on internal circuitry to increase from 300 baud to 1200 baud.
- * 4. Make sure that the Bitstreamer board is set for the same speed as the printer. This is set on a dipswitch on the upper left-hand corner of the board. Press the appropriate switch in and upward and make sure all other switches are pressed downward.
- * 5. Make sure that there is a regular I/O cable connected to J3 on the Bitstreamer board and installed at the other end in a cutout at the rear of the mainframe. This socket will be used for the printer cable.
6. Connect the 2 pin socket at one end of the video cable to the 2 left-most pins which will be found rising vertically from the left-hand corner of the Flashwriter board. Install the circular socket at the other end of the cable

into one of the circular cutouts at the rear of the mainframe.

7. Connect the 24 pin dip plug at one end of the second I/O cable to J1 on the Flashwriter board. Install the 25 pin socket at the other end in one of the cutouts at the rear of the mainframe. This socket will be for the keyboard cable.
- * 8. Plug the printer external cable into the appropriate socket on the rear of the mainframe.
9. Plug the keyboard and monitor external cables in the appropriate sockets on the rear of the mainframe.

Software Installation Procedures

- * 1. Under MDOS, if printer uses Diablo protocol, enter DIAB4 (return); if printer uses Teletype protocol, enter DECW4 (return).
- * 2. Save RES module on Personalized System Diskette

2.2.1.6 Printer: SERIAL, DIABLO 1610 or TELETYPE PROTOCOL. Console: VECTOR GRAPHIC MINDLESS TERMINAL.

Example: Printer: if Diablo protocol - Diablo 1610 or 1620, Qume Sprint 5, or NEC Sprinwriter; if Teletype protocol - Decwriter, Teletype, or TI 810 or 820.

Console: Vector Graphic Mindless Terminal.

Interface Components Required

1. Option EV Extended Systems Monitor on PROM
2. Flashwriter board
3. Mindless Terminal 3-part I/O cable
4. External Mindless Terminal cable (or equivalent)
- * 5. Bitstreamer board with I/O cable (no need to order; included in system automatically.)

Hardware Implementation Procedures

1. If no printer is being used, remove the Bitstreamer from the mainframe. Do not put it back in. It cannot be in the system.
- * 2. Jumper the Bitstreamer board so that it is readdressed for ports 4 - 7 rather than the original 0 - 1. Instructions will be found in the Bitstreamer User's Manual.
- * 2. Make sure that the printer is set for its highest speed, (1200 baud for Diablo 1610 protocol), and that its parity setting is MARK parity. Check the printer manual if

necessary. Some printers such as the Diablo require a jumper on internal circuitry to increase from 300 baud to 1200 baud.

- * 4. Make sure that the Bitstreamer board is set for the same speed as the printer. This is set on a dipswitch on the upper left-hand corner of the board. Press the appropriate switch in and upward and make sure all other switches are pressed downward.
- * 5. Make sure that there is a regular I/O cable connected to J3 on the Bitstreamer board and installed at the other end in a cutout at the rear of the mainframe. This socket will be used for the printer cable.
- 6. If not already done at the factory, install the Mindless Terminal 3-part I/O cable as instructed in the terminal's documentation. The 3 parts are connected to the power supply, the Flashwriter board video output pins, and the Flashwriter board keyboard input socket (J1). At the other end, the DB25 socket is installed in one of the cutouts at the rear of the mainframe.
- * 7. Plug the printer external cable into the respective socket at the rear of the mainframe.
- 8. Plug the terminal external cable into its socket at the rear of the mainframe.

Software Installation Procedures

- * 1. Under MDOS, if printer uses Diablo protocol, enter DIAB4 (return); if printer uses Teletype protocol, enter DECW4 (return).
- * 2. Save RES module on Personalized System Diskette.

* 2.2.1.7 SERIAL PRINTING TERMINAL (HAS A KEYBOARD), DIABLO 1610 OR TELETYPE PROTOCOL AND A VIDEO MONITOR

Example: Printing terminal: if Diablo protocol - Diablo 1620, Qume Sprint 5 with keyboard, or NEC Sprinwriter with keyboard; if Teletype protocol - Decwriter, Teletype, or TI 810 or 820, with keyboards;
Video monitor: Hitachi.

Interface Components Required

- 1. Option CV Extended Systems Monitor on PROM
- 2. Flashwriter board
- 3. Video cable, Flashwriter to rear panel
- 4. Video Monitor to Mainframe cable
- * 5. Bitstreamer board with I/O cable (no need to order; included in system automatically.)

Hardware Implementation Procedures

- * 1. Make sure that the printer is set for its highest speed, (1200 baud for Diablo 1610 protocol), and that its parity setting is MARK parity. Check the printer manual if necessary. Some printers such as the Diablo require a jumper on internal circuitry to increase from 300 baud to 1200 baud.
- * 2. Make sure that the Bitstreamer board is set for the same speed as the printer. This is set on a dipswitch on the upper left-hand corner of the board. Press the appropriate switch in and upward and make sure all other switches are pressed downward.
- * 3. Make sure that there is a regular I/O cable connected to J3 on the Bitstreamer board and installed at the other end in a cutout at the rear of the mainframe. This socket will be used for the printer cable.
- * 4. Disable the parallel port on the Flashwriter board. To do this, simply remove chip U52 from the board, using a small screw driver to pry it out of its socket. If U52 cannot be easily located, refer to the Flashwriter User's Manual.
- 5. Connect the 2 pin socket at one end of the video cable to the 2 left-most pins which will be found rising vertically from the left-hand corner of the Flashwriter board. The socket should be positioned so that the inside wire is connected to pin 1, and the outside "shield" wire is connected to pin 2 (ground). Install the circular socket at the other end of the cable into one of the circular cutouts at the rear of the mainframe.
- * 6. Plug the printer external cable into the socket on the rear of the mainframe.
- 7. Plug the monitor external cable into the appropriate socket on the rear of the mainframe.

Software Installation Procedures

- * 1. Under MDOS, if printer uses Diablo protocol, enter DIAB (return); if printer used Teletype protocol, enter DECW (return).
- * 2. Save RES module on Personalized Diskette.

* 2.2.1.8 SERIAL PRINTING TERMINAL (HAS KEYBOARD), DIABLO 1610
OR TELETYPE PROTOCOL
NO VIDEO.

Example: Printing terminal: if Diablo protocol - Diablo 1620, Qume Sprint 5 with keyboard, or NEC Sprinwriter with keyboard; if Teletype protocol - Decwriter, Teletype, or TI 810 or 820, with keyboards;

Interface Components Required

1. Option C Extended Systems Monitor on PROM
- * 2. Bitstreamer board with I/O cable (no need to order; included in system automatically.)

Hardware Implementation Procedures

- * 1. Make sure that the printer is set for its highest speed, (1200 baud for Diablo 1610 protocol), and that its parity setting is MARK parity. Check the printer manual if necessary. Some printers such as the Diablo require a jumper on internal circuitry to increase from 300 baud to 1200 baud.
- * 2. Make sure that the Bitstreamer board is set for the same speed as the printer. This is set on a dipswitch on the upper left-hand corner of the board. Press the appropriate switch in and upward and make sure all other switches are pressed downward.
- * 3. Make sure that there is a regular I/O cable connected to J3 on the Bitstreamer board and installed at the other end in a cutout at the rear of the mainframe. This socket will be used for the printer cable.
- * 4. Plug the printer cable into the socket at the rear of the mainframe.

Software Implementation Procedures

- * 1. Under MDOS, if printer uses Diablo protocol, enter DIAB
(return); if printer uses Teletype protocol, enter DECW
(return).
- * 2. Save RES Module on Personalized System Diskette.
- * 3. If printer uses Diablo protocol, then before each session at the computer, as the first step after loading MDOS, enter:

ASSIGN 2,3 (return)
ASSIGN 1,0 (return)

(Do not be concerned that while entering the second line, the printer prints every character twice.)

NOTE: Using the serial Diablo protocol printing terminals at 1200 baud with no video display is limited by the fact that no Extended Systems Monitor commands which cause outputting more than about 40 characters can be used. (This is because serial output from the Extended Systems Monitor does not use the Diablo protocol technique of checking whether the printer can accept the next character. More than 40 characters at 1200 baud will usually cause the printer's buffer to overflow.) MDOS and M.BASIC commands do not cause the same problem, so long as the above mentioned ASSIGN commands are used prior to each session.

One way to solve this problem is to run the printer at 300 baud (Bitstreamer at 300 baud too) and to use the DECW command rather than the DIAB command before saving the RES module on the Personalized System Diskette. In this case, the ASSIGN commands are not needed. The drawback is slower printing.

2.2.2 ADDING A STANDARD PRINTER TO AN EXISTING SYSTEM

The information in this section concerns adding a printer to an existing system, one which already has some kind of video display and keyboard functioning. The logic behind this information is the same as that used in section 2.2.1, except that here it is presented in summary form.

The printers presently considered standard for Vector Graphic systems are:

Centronics Series 700 parallel matrix printers,
Diablo 1610 protocol serial printers, such as Diablo 1610, Qume
Sprint 5, or NEC Sprinwriter, and
Teletype protocol printers, such as Decwriter, Teletype, or TI
810 and 820.

There are many makes and models with protocols similar or identical to the above. Some differences between makes of printers will not make them incompatible with the Vector Graphic computers necessarily, but it is recommended that the user try out with his system any printer not listed above, before purchasing.

Adding a printer involves 3 steps:

- 1) obtain the interface components, as well as the printer,
- 2) do hardware implementation procedures required, and
- 3) do software implementation procedures required.

INTERFACE COMPONENTS REQUIRED

1) Bitstreamer board and I/O cable. Generally, use the one which came with your system. If it is being used to control a serial terminal now, it can be used in addition to control a parallel printer such as a Centronics printer. However, if the present

terminal is serial, and a SERIAL printer such as Diablo, Qume, or Teletype is desired, a second Bitstreamer and I/O cable must be ordered.

2) If a parallel Centronics protocol printer is to be implemented, order a CENTRONICS INTERFACE KIT from Vector Graphic or an authorized dealer.

HARDWARE IMPLEMENTATION

1) If the keyboard and video are controlled by a Flashwriter board, or if both the printer and the video console are serial, then there will be 2 interface boards in the system. When this is the case, the Bitstreamer controlling the printer must be jumpered to respond to port addresses 4 - 7 rather than 0 - 1. Instructions will be found in the Bitstreamer User's Manual.

2) If the printer is a parallel printer using Centronics protocol, make the modifications to the Bitstreamer board and install the Centronics Interface Kit, both as described in Appendix H. Do all the procedures in Appendix H if the keyboard and video are a serial terminal such as Hazeltine. However, if the keyboard and video are controlled by a Flashwriter board, then do not bother to install the 6-pin plug or the serial I/O cable.

3) If printer is serial, make sure it is set at its highest speed (1200 baud if it is Diablo 1610 protocol.) Then make sure the dipswitch on the upper left-hand corner of the Bitstreamer is set at the same rate (chosen switch up, all others down.) Printer must be set for MARK parity.

4) Make sure the the 24 pin dip plug on the I/O cable is inserted in J3 on the Bitstreamer board and that the socket on the other end is installed in one of the cutouts on the mainframe back panel. Then plug the printer cable into that same socket on the back panel.

SOFTWARE IMPLEMENTATION

The RES Module on the MDOS System Diskettes is not configured for any particular printer. However, a large number of versions of the I/O portion of the RES Module are present on the diskettes. The user need only overlay the desired version onto the RES Module stored in memory, and then save the new RES Module onto the Personalized System Diskette. The versions available as of this release are:

CENT and CENT4 for parallel Centronics protocol printers

DIAB and DIAB4 for serial Diablo protocol printers

DECW and DECW4 for serial Teletype protocol printers

In each case, the version with a "4" attached must be used if the Bitstreamer has been readdressed for ports 4 - 7. Otherwise use the version without a "4".

To accomplish the overlay, simply enter the name of the file in upper case letters following the MDOS prompt >. After the overlay is done, indicated by another MDOS sign-on message appearing on the screen, save the RES Module by entering the following commands under MDOS:

```
TYPE "RES" 0 (return)  
SCRATCH "RES" (return)  
SAVE "RES" 2B8 146B 3 (return)
```

If the printer is not one of the above types, then a custom interface routine must be written. See Appendix N.

2.2.3 NON-STANDARD CONFIGURATIONS

Any configuration of peripherals which includes a printer, video unit, keyboard, or terminal different than those used in the standard configurations, is a non-standard configuration.

Hardware: In order to order and implement the interface hardware, use the standard configuration procedures as models as far as is possible.

Software: In many non-standard configurations, it will be necessary to custom write a printer and/or console physical I/O routine. refer to Appendix M for rewriting console I/O and to Appendix N for rewriting printer I/O.

2.3 MODIFYING THE SYSTEM HARDWARE

2.3.1 CHANGING TO 2 MHZ CLOCK RATE

Some non-Vector Graphic S-100 boards operate only at 2 MHz, the rate of the original 8080 clock. Since the Z-80 can operate at both rates, you may desire to run the system at 2 MHz in order to include such boards. Instructions will be found in Appendix L.

2.3.2 CONNECTING ADDITIONAL DISK DRIVES

2 Micropolis disk drives are standard equipment. Additional drives may be added because the Micropolis software can address up to 4 drives. Contact your dealer or Vector Graphic in order to order.

2.4 DISKETTE MEDIA

2.4.1 DESCRIPTION

The recording medium used with the MZ Micropolis diskette subsystem is an industry standard 5 1/4-inch diskette (Fig 2.1) 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 computer stores or from other computer supply sources. DO NOT USE DISKETTES WITH OTHER THAN 16 HARD SECTORS, OR THOSE WHICH ARE SOFT-SECTORED (NO SECTOR HOLES). THEY WILL NOT WORK.

2.4.2 HANDLING

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 mounting 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 unmount 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.

2.4.3 LOADING AND UNLOADING

There are two stages of loading a diskette. First, insert the diskette with label side upward for horizontal drives, or leftward for vertical drives, and with the edge nearest to the read/write head access hole going in first. Insert the diskette all the way, until it clicks into place. At this point the diskette is said to be "inserted" but not yet "mounted". The diskette may be left like this for any length of time without decreasing its life. Power may be turned on or off with the diskette in this condition. It is recommended however that if a diskette will not be used for any length of time it be returned to its envelope or other storage file.

Second, the diskette is "mounted" by depressing the manual load actuator on the disk drive slowly but firmly until it stays in the mounted position. The drive will begin to turn and rotate the diskette inside its jacket. If the load actuator cannot be fully depressed, this indicates that the diskette was not inserted completely or properly.

Power should NOT be turned on or off when a diskette is in the mounted position. The consequence is from time to time the loss of data on the diskette.

Once the diskette is mounted, it is accessible by software for writing or reading. When a read or write operation is initiated, you will hear an audible click from the drive unit and the red light on the unit will glow, indicating that unit has been selected. After the operation is complete, the unit will remain selected for 5 seconds. At the end of 5 seconds, the unit will be automatically deselected: the red light will go out, and there will be another click as the head load pad is raised off the surface of the diskette. This automatic deselect feature is important in lengthening the life-span of diskettes.

To dismount the diskette, press the load actuator down as far as it will go, then release pressure. It will then open to the unmounted position. This discontinues rotation of the diskette within its jacket. In order to do your part as user in prolonging the life of the diskette, observe the following rule: UNLOAD THE DISKETTE DURING PERIODS IN WHICH IT IS NOT IN USE. This reduces wear of the diskette against its jacket. Note that the diskette may be left inserted, so long as it is unmounted, without shortening its life.

To remove the diskette, press the load activator upward (or leftward in vertical drives). The diskette will be popped out (de-inserted) and can now be removed.

2.4.4 REPLACEMENT AND BACK-UP OF DISKETTES

The nature of floppy diskette drives is that the read-write head is in contact with the diskette surface whenever the unit is selected, resulting in gradual deterioration of the surface. Continual loading of the head on a single track will naturally result in its deterioration before the rest of the diskette. The rotation of the diskette within its jacket is an additional source of wear.

Backup: The BEST defense against loss of diskette-based data is maintaining a back-up diskette for each diskette you use. In the business world, this is considered dogma. Data is most often lost due to damage to diskettes from accidental mis-handling; normal wear is much less often a problem. The standard rule of thumb is as follows: copy a front-line diskette on to its back-up whenever you cannot afford to lose the information stored since you last backed it up. This goes for programs as well as data. If you are operating business programs such as inventory or accounts receivable, maintain a regular back-up schedule, once a week or once bi-weekly. In addition, your programs if possible should be written so that an internal file of entries is maintained, and a printout of entries made each day is produced. Then, if data is lost before it can be copied on to the back-up, it is fairly easy to re-enter it, using the back-up diskette as the starting point. In business particularly, back-up diskettes and printouts of daily entries should be stored in a safe place.

Replacement: In addition to being backed up, frequently used diskettes must be replaced from time to time. The intervals are entirely dependent on the kind of usage. There are no accurate predictions for diskette life-span, but 2000 to 3000 hours of rotation is a reasonable estimate. A good suggestion therefore is to replace such diskettes every 6 months. Data diskettes used infrequently may never require replacement.

Failure of a diskette will be indicated by the inability of the system to read a file which it normally has been able to read. MDOS will report "PERM I/O ERROR". With proper care, this should not occur.

Replacing a diskette simply means copying it onto a new previously unused diskette. The old diskette can be used for temporary storage, or disposed of.

To copy diskettes use the Diskcopy Utility, see Section 4.3.

2.4 DISKETTE MEDIA

2.4.1 DESCRIPTION

Use an industry standard 5 1/4-inch diskette (Fig 2.1) with 16 "hard" sectors. There will be 16 sector holes and 1 additional index hole around the edge of the center hole. Get them from computer stores or from other computer supply sources. DO NOT USE DISKETTES WITH OTHER THAN 16 HARD SECTORS, OR THOSE WHICH ARE SOFT-SECTORED (NO SECTOR HOLES). THEY WILL NOT WORK.

Without relation to price, some brands of diskettes do not work well in the Micropolis high-density drives. Use one of the following brands: Scotch, Dysan, or Maxell. Other brands will not be reliable.

Individual diskettes may sometimes not work. Besides manufacturing defects, we have occasionally found batches of diskettes with the wrong number or sectors, and sometimes diskettes are manufactured with 2 diskettes inside the jacket. Diskettes which do not work or do not work reliably should be replaced immediately.

2.4.2 IF YOU HAVE PROBLEMS WITH DISK ERRORS

By a disk error, we are referring to errors reported on the screen as "PERM I/O ERROR", indicating something wrong with the diskette or drive. (The message is different in different operating systems. Another uses "CRC ERROR".) If your system generates such errors often with different diskettes, take the following measures in the order given:

- a) Make sure the cover to the mainframe is on. It is a shield.
- b) Switch to another of the suggested brands of diskettes.
- c) If the errors persist, contact your dealer or service representative.

2.4.3 HANDLING

Diskettes are easily damaged and contaminated. Please obey the following rules without exception:

- a) Do not touch or attempt to clean the inner surface.
- b) Do not smoke, eat, or drink while handling the diskette.
- c) Do not place heavy objects on the diskette.
- d) Do not expose the diskette to excessive heat or sunlight.

- e) Do not use rubber bands or paper clips on the diskette.
- f) Do not bend or fold the diskette.
- g) Do not write on a diskette with a pencil. A fiber-tipped pen is recommended. Return the diskette to its envelope before writing on it.
- h) Do not expose the diskette to magnetic fields.
- i) After use, always return a diskette to its protective envelope or other protective system such as plastic notebook pages designed for diskettes.
- j) Store diskettes in a vertical position, thus reducing rubbing.
- k) If a diskette is damaged or contaminated, replace it. If a contaminated diskette is placed in the drive, the receiver and read/write head may become contaminated and ruin other diskettes.
- l) Unmount the diskette if it will not be accessed for a half hour or more. If the interval is very long, remove it from the drive and return it to its storage envelope.

2.4.4 LOADING AND UNLOADING

There are two stages of loading a diskette. First, insert the diskette with label side leftward, with the edge nearest the exposed area pointing inward. Insert the diskette until it clicks into place. You should not have to push so hard that the diskette bends. The diskette is now "inserted" but not yet "mounted". Although not good practice for long periods, you may leave the diskette like this any length of time, and even turn power on or off.

Second, to "mount" the diskette, push the door of the drive until you feel increased resistance about half-way closed, then SLOW DOWN, and push SLOWLY but surely until it stays in the mounted position. The drive will begin to turn and rotate the diskette inside its jacket. If you cannot fully close the door, the diskette is not inserted properly.

Do NOT turn power on or off while a diskette is in the mounted position. This will sometimes damage the diskette. However, if you accidentally do this, go ahead and use the diskette because it is probably undamaged.

Once the diskette is mounted, it is accessible by software for writing or reading. When the computer accesses the diskette, you will hear a click from the drive and its red light will glow. After the operation is complete, the drive will remain on for 5 seconds. You can be entering new material at the keyboard during this time. At the end of 5 seconds, the red light will go out, and there will be another click as the head load pad is raised off the surface of

the diskette. This automatic deselect feature is important in lengthening the life-span of diskettes.

To dismount the diskette, press the door further open as far as it will go, then let it close. It will then release to the unmounted position. This stops the rotation of the diskette. UNMOUNT THE DISKETTE DURING PERIODS IN WHICH IT IS NOT IN USE. This reduces wear of the diskette against its jacket. You may leave it inserted withouth shortening lifespan.

To remove a diskette, press the door leftward. The diskette will pop out.

2.4.5 RECOVERY TECHNIQUES

If you repatedly get PERM I/O erros using one particular diskette, then it is probably defective. This will sometimes happen with a new diskette when you are initializing it or copying another diskette to it. After several attempts, discard it or return it if possible. Whenever you repeat a disk operation after an error, always unload and reload the diskette, because it may be seated incorrectly.

If an old diskette repeatedly gives errors, first repeat the operation several times, unloading and reloading the diskette each time. If there is still a problem, check the center hole. If it is wrinkled, straighten it out with your fingers and then try again. If you still get errors, try copying the diskette to another diskette using the DISKCOPY utility in MDOS. If the error still occurs, try switching source and destination drives. Some combination of drives and repositioning of diskettes within drives will almost always result in a successful copy. If you cannot copy a diskette at all, then copy it file by file to another initialized diskette using the MDOS COPYFILE utility. There will probably be one file which does not copy, but if you are lucky, they will all be good.

2.4.6 REPLACEMENT AND BACK-UP OF DISKETTES

As with any magnetic storage medium, the recording gradually deterioreates over time. Even if a diskette is not damaged, it will begin producing errors after sufficient use.

Backup: The BEST defense against loss of diskette-based data is maintaining a back-up diskette for each diskette you use. In the business world, this is considered dogma. Copy a diskette on to its back-up whenever you cannot afford to lose the information stored since you last backed it up. This goes for programs as well as data. If you are operating business programs such as inventory or accounts receivable, maintain a regular back-up schedule, once a week or once bi-weekly. In addition, a transaction journal - that is a printed copy of entries made each day into the system - is an excellent idea to build into business software as a last resort back-up.

Replacement: In addition to being backed up, replace frequently used diskettes by copying to a fresh diskette every 6 months. A good suggestion is to use the back-up diskette, which is fairly fresh, as the new front-line diskette, and to create a fresh back-up. Do not wait until a frequently used diskette fails, before you replace it with the back-up.

To copy diskettes, use the DISKCOPY utility. See Section 4.3

2.4.7 INITIALIZING DISKETTES

Previously unused diskettes must be initialized (also called "formatted") before use. There are two routines in the Micropolis software that can do this. Use either the INIT command in MDOS (see 4.1.5.22) or the F command in the BASIC UTILITY program operating under M.BASIC. (see Appendix B). Their results are identical. DO NOT INITIALIZE THE MDOS SYSTEM DISKETTES PROVIDED WITH THE SYSTEM, OR ANY OTHER DISKETTE CONTAINING DESIRED INFORMATION. THIS DESTROYS THEIR CONTENTS.

2.4.8 WRITE PROTECT FOR DISKETTES

Write protect tabs come in boxes of new diskettes. If you attach a tab over the write protect cutout on a diskette as shown in Fig. 2.2 the disk drive will not allow you to erase or change any information on the diskette. The tab may be removed later.

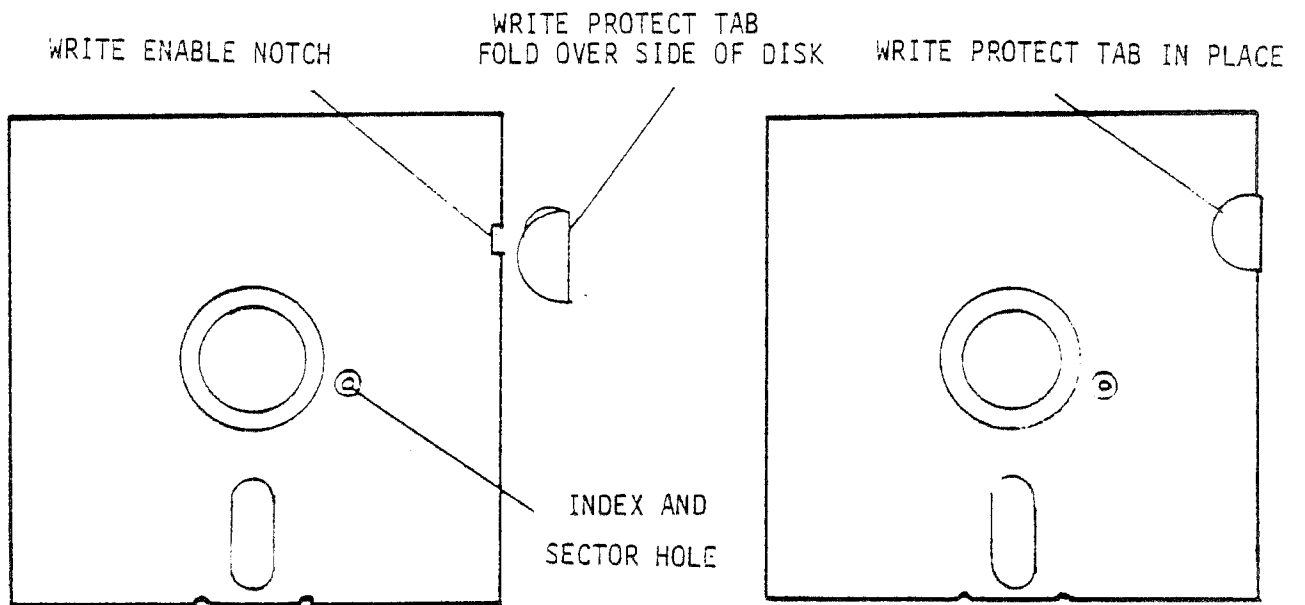


Figure 2.2 How To Mount Write Protect Tab

III DAY TO DAY OPERATIONS FOR MDOS AND M.BASIC

3.0 SUMMARY OF NORMAL START UP PROCEDURE

- 1) Power-on the mainframe, then the peripherals.
- 2) If yours is a MEMORITE system, depress RESET key.
- 3) Insert and mount MDOS System diskette in drive 0.
- 4) Enter B on keyboard. MDOS comes on.
- 5) Enter BASIC (return) on keyboard. M.BASIC comes on.

(return) means press the RETURN key.

Please read the rest of this chapter thoroughly. The above does not give all the information you need.

3.1 SUMMARY OF PROMPTS

When one of these prompts appears, it indicates the corresponding system is loaded and its executive routine is waiting for operator input.

- | | |
|----------|---------|
| 1) * | Monitor |
| 2) > | MDOS |
| 3) READY | M.BASIC |

3.2 POWER-ON

- 1) No diskette may be in mounted position, (i.e. rotating) but it may be inserted in drive.
- 2) Turn the power key on the mainframe. The RESET button will light up.
- 3) If yours is a MEMORITE system, depress the RESET button.
- 4) Switch on all desired peripherals.
- 5) Depress RESET on printer, if printer will be used and if printer has one.
- 6) An asterisk and cursor will appear on the console indicating the Extended Systems Monitor executive is available for commands. A few Monitor commands are covered in this chapter. The remaining will be found in the Extended Systems Monitor manual. Look it over. Some may be useful. Monitor commands can be entered at this time or at any other time that the Monitor executive is called back into control, indicated by the Monitor prompt (*).

3.3 LOAD MDOS

- 1) Insert, if not done already, and mount an MDOS System diskette in drive 0. In place of the MDOS System diskette, you may substitute an M.BASIC-only diskette.
- 2) Enter B. MDOS will be loaded into memory and control will be transferred to the MDOS executive. The screen will look like this:

```
*B
Vector MZ MDOS X.XX
>
```

You may now enter MDOS commands (Chapter 4).

If MDOS should come up but does not, refer to Appendix I for troubleshooting.

If a M.BASIC-only diskette was in drive 0, the screen will look like this:

```
*B
MICROPOLIS BASIC VS. X.X.-COPYRIGHT 19XX
READY
```

In this case, you may begin entering M.BASIC commands immediately (chapter 5) and skip Section 3.4. Section 2.3.6 discusses BASIC-only diskettes.

3.4 LOAD M.BASIC FROM MDOS

You may work in MDOS for some time and then transfer control to M.BASIC, or you may desire to go immediately to M.BASIC as your first MDOS command. In either case, enter BASIC (return). The screen will appear like this:

```
>BASIC
MICROPOLIS BASIC VS. X.X.-COPYRIGHT 19XX
READY
```

You may now enter M.BASIC commands. (Chapter 5).

3.5 OTHER OPERATING SYSTEMS AND LANGUAGES

This manual deals primarily with the MDOS operating system, as it is normally delivered. For commands in other operating systems, including how to load their associated BASIC's or other languages, refer to the manuals for those systems, included if and when they are ordered.

3.6 RETURNING TO MDOS FROM M.BASIC

- 1) Make sure there is a System diskette with MDOS mounted in drive 0.
- 2) Enter LINK "MDOS" (return). (See Section 5.21.2.7 for how LINK works and for other uses of LINK command).
- 3) Screen will look like this:

```
READY
LINK "MDOS"
Vector MZ MDOS X.XX
>
```

You may now enter MDOS commands.

To return to M.BASIC, enter BASIC (return) as usual (see Section 3.4.)

3.7 RETURNING TO MONITOR FROM ANYPLACE

- 1) Depress control-Q (hold CTRL key down while depressing Q); or press the RESET key on the mainframe front panel. Control-Q is preferred.
- 2) You may now enter Extended System Monitor commands.

NOTE: For systems without the version 3.1 Systems Monitor, control-Q will not work when you try it. If you find this to be the case, then either the ESC key or control-X WILL work instead. To find out which will work in your system, get MDOS running and try them. Control-X and the ESC key each have a special function in the MDOS and M.BASIC editors. If one of these causes a return to the Monitor, then obviously, you cannot use that function in the MDOS and M.BASIC editors. Make a mental note of this when reading the MDOS and M.BASIC editor instructions. If ESC or control-X causes a return to the Monitor instead of control-Q then substitute it wherever control-Q appears in this chapter.

Returning to the Monitor is useful when Monitor commands are needed for trouble-shooting MDOS or M.BASIC programs. It is also used if there is no other way to break out of an undesired loop or output sequence in any program. Always use control-Q rather than RESET if possible, because on extremely rare occasions, RESET may change some of the contents of memory.

Control-Q will not work when certain special purpose programs are operating. The most important of these are disk access routines, and the Word Management System and MEMORITE word processing software. RESET is necessary in these cases if you want to return to the Monitor.

Avoid using RESET to abort a disk write operation, if possible, because if at that moment the directory is being written, then all the data on the disk can be effectively lost. (The same holds true if you dismount the disk at that time.)

In addition, aborting a disk read or write operation may leave the file in an "open" state, which can cause an error message next time the drive is accessed. This can be cleared by executing the FILES command in MDOS. Enter FILES (return), then return to your program and access the disk.

The best advice is, in general, allow disk read and write operations to go to their natural conclusions. Only abort if the operation is looping indefinitely.

3.8 RETURNING TO MDOS (OR M.BASIC) FROM MONITOR IF MDOS (OR M.BASIC) IS ALREADY IN MEMORY

This is the MDOS (or M.BASIC) warm-start command.

Depress J after the Monitor prompts with *.

3.9 RETURNING TO MDOS OR M.BASIC EXECUTIVE FROM WITHIN A ROUTINE RUNNING UNDER THAT EXECUTIVE

Depress control-C. (Hold the CTRL key down while depressing C.)

Response is MDOS prompt (>) if MDOS is the executive, or BASIC prompt (READY) if BASIC is the executive.

Control-C is used to leave a routine at other than the normal end point. Use it when the routine is waiting for any type of keyboard input. It is sometimes also effective for interrupting an overly long or unending stream of output.

If it does not work, then control-Q is the alternative. Since this returns control to the Monitor, depress J then to return to MDOS or M.BASIC.

3.10 VIDEO COMMANDS

This section is ONLY relevant to systems using memory mapped video, such as the Vector Graphic Mindless Terminal. If a serial terminal such as Hazeltine is used, then refer to the manual for that terminal to find how you can control the screen image from the keyboard.

These commands may also not work if another operating system, such as CP/M is in control of the system. They will definitely not work when word processing, using the Word Management System or MEMORITE, is in control.

Most of the time, when the system is waiting for keyboard input,

operator may perform the following operations on the screen image. These commands are made possible by the Extended Systems Monitor. For more information of a technical nature, refer to the Extended Systems Monitor manual.

3.10.1 CLEAR SCREEN

Depress control-D.

3.10.2 SCROLL SCREEN UP ONE LINE

Depress control-J or LF key.

3.10.3 BACKSPACE CURSOR

Depress BACKSPACE key, underscore key, or control-H. Also, the DEL key will have this effect IF MDOS or M.BASIC is running.

These commands will always work when MDOS or M.BASIC executives are waiting for input, and when any M.BASIC program is waiting for input.

In other situations, for example, when an assembly language program is waiting for input, these commands may or may not work depending on how the program in control was written.

3.10.4 CONVERT THE SYSTEM TO REVERSE VIDEO

For variation, you can cause the screen to display characters black-on-white rather than white-on-black. Just depress control-T (hold down CTRL key while depressing T) If you depress this again, the video will return to white-on-black. Characters already entered will remain on the screen the way they were entered.

3.10.5 TAB CURSOR TO NEXT TAB LOCATION (EVERY 8 SPACES)

Depress TAB key or control-I

3.10.6 ELIMINATE CURSOR FROM THE SCREEN

Depress control-N

3.10.7 MOVE CURSOR TO TOP OF SCREEN

Depress control-B

3.10.8 MOVE CURSOR DOWN, UP, LEFT, OR RIGHT

Depress one of the keys with an arrow on it. If your keyboard has no arrow keys, then depress control-R, control-U, control-W, or control-Z to move cursor down, up, left, or right respectively. However, Control-U and the up-arrow key will not work under while in MDOS or M.BASIC, though it will work under certain machine language programs and when in the Extended Systems Monitor echo mode (Y command).

3.10.9 RETURN CURSOR TO LEFT EDGE OF SCREEN

Depress RETURN key or control-M.

3.11 POWER-DOWN

1. Make sure you have stored on diskette all the programs and data you wish to save.
2. Dismount all diskettes. They may be left inserted and clicked in, so long as they are not mounted (rotating).
3. Turn off all peripherals.
4. Turn the power key on the mainframe front panel.

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) When DEL or 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. Imbedded 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 0 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 0 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 0 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 overlaid 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 2D 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 IN HEX	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 overlay 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     MVI     B,250
3000 GET     CALL     GETDATE
3500     JC      EXIT           ;CLOSE FILE & EXIT
4000     CALL    @WTINXPOSI
5000     DCR     B
6000     JNZ     GET
7000     CALL    @INCRECPOS
8000     JMP     GET
```

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 users 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 0 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 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.

4.3.1.20 @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 @ONLINEOUT - NEW LINE OUTPUT

The @ONLINEOUT 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 @AHXTBIN 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 \emptyset 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 \emptyset 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 \emptyset 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 (Ø length) record written to it. It is left open and ready for access with the index position set to Ø 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 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 0.

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 0.

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 0 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 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 0 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 0 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 filename 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.

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.

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 0 in this case.

It expects the data in the C register, and the filename 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.

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.

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 from 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.

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.

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.

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 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.

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 @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.

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 prered. It requires that the prered 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 @DEM0DHL - BC=DE%HL

The @DEM0DHL 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 @DEM0DHL.

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.

@PARAMLEN - A one byte parameter used by the @SEAR routine. It contains the length of the functions in the table to be searched.

@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) When DEL or 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 128 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 on 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 TAG01A
20 TAGOFF
30 TAG22A
CHANGE
SEARCH MASK ? TAG??A
CHANGE TO ? LABEL??B
10 LABEL01B
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 01 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 depressing the ESC key. 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          @RFINXPOSI
2000                DCR              C
3000                JZ              ENDOFFILE
4000                MVI             D,0
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=0) 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 ZSM - Z-80 ASSEMBLER

ZSM is an MDOS program to convert Z-80 assembly language source code into object code, which consists of a sequence of binary codes that can be loaded into the computer's memory and executed. ZSM takes the place of ASSM, the earlier 8080/8085 assembler for MDOS. Any references in this manual to ASSM should be understood as references to ZSM.

As input ZSM expects a type 4, 5, 6, or 7 text file, such as that produced by LINEEDIT. The output file produced will be a type 8 file. This type of file may be scatter loaded into memory, meaning that it need not be contiguous code; rather, it can be several groups of individual code.

Note that this is a disk assembler, so memory size is not a constraint on the size of file that may be assembled.

ZSM is a copyrighted piece of software. Any reproduction or redistribution of it or this manual is expressly forbidden.

4.5.1 HOW TO RUN ZSM

ZSM is invoked from the MDOS executive by typing its name, followed by the assembly parameters. The format is as follows:

```
>ZSM "<source filename>" "<object filename>" "<options>" [<offset>]
```

The <source filename> must be the assembly language source program as explained above. The <object filename> is the name of the output file. It must be included, but may be blank if the S or M option, below, is used.

The <options> are instructions to ZSM pertaining to how to assemble the program. The number of options specified varies with what is desired and may be blank, but the field must nevertheless be included. The options are as follows.

- E Only lines containing assembly errors will be listed.
- P The assembly listing will be paginated.
- S The assembly listing will be produced, but no object code.
- M The object code will be written into memory, not to a disk file.
- L The line numbers from the source file will not appear on the listing.
- T The symbol table created by ZSM will be printed following the listing.

"SM" is the only combination not allowable, since they are mutually exclusive. If they are both present, though, the S option will prevail.

The <offset> parameter indicates an offset to be added before the object code is placed into memory (via the M option). For example, it would be impossible to assemble a program into memory at 2B00, since that is where ZSM resides. Therefore, to put a program into memory that was designed to run at 2B00, you would have to specify an offset, for example 3000. This would result in code destined for 2B00 to be actually put into memory at 5A00 (2B00 + 3000).

Here are some examples of valid commands:

1. ZSM "SFILE" "OFILE" ""
2. ZSM "SFILE" "" "PTS"
3. ZSM "SFILE" "" "ML" 3000
4. ZSM "SFILE" "OFILE" "E"

Line 1 would assemble SFILE into the file OFILE, and produce a normal listing. Line 2 would assemble SFILE, producing a paginated listing including a symbol table, but not produce an object file. Line 3 would assemble SFILE, putting the object code into memory with an offset of 3000; it would produce no object file; and it would produce a normal listing, but without line numbers. Line 4 would assemble SFILE into the file OFILE, and only list those lines (if any) containing errors.

Assembling a file with the M option in such a way that the operating system or assembler would be overwritten will cause a 'Load address error'. Including the wrong number of parameters in the command line, or forgetting a quote symbol, will cause a 'Syntax error'. Specifying an object file which already exists will cause a 'Duplicate name' error, meaning there already exists a file with that name. Either SCRATCH that file, or select a new name for the object file.

4.5.2 LANGUAGE ELEMENTS

The source file has a general format as follows:

```
#### LABEL: OPCODE OPERANDS ;comments
```

The #### represents the four digit line number assigned each line by the line editor. Although the line number itself is ignored, it *must* be present, and must be four characters long, followed by a space.

The LABEL is optional. If present, it will be entered into the symbol table. Whether or not it is present, its position must be followed by a space or colon. That is,

```
#### LABEL OPC or #### LABEL: OPC or #### OPC
```

are valid, while

```
#### OPC
```

is not.

Labels may include any of the following characters:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 @ . [] { } \ | ' ^ ~

To avoid ambiguity, however, the first character may not be . or 0-9. In addition, a label may be of any length up to 47 characters. All characters are significant. In normal use, though, up to 12 characters should suffice; and over 14 characters will look a little strange on the listing.

The OPCODE must either be a Z-80 opcode or a pseudo-op. Both are explained later.

The OPERANDS vary. There can be any number of them, depending on whether they are operands for an opcode or a pseudo-op. There are also instances where there are no operands, and therefore this field can, in some cases, be omitted. If more operands are supplied that are needed, the extras are ignored.

The COMMENT field is totally ignored by the assembler, except for printing it on the listing. Comments are used only for documentation or clarity, and can be omitted altogether. If present, comments should be preceded by a semicolon (;). The semicolon will cause a TAB to the third TAB setting, whereas its absence will result in the comment appearing immediately to the right of the operand field.

There is one exception to the above format, and that is the case of an all-comment line. If the first character of the line (after the line number and space) is either an asterisk (*) or semicolon, the entire line will be treated as a comment.

4.5.2.1 CONSTANTS

ZSM provides for constants of two varieties, numeric and ASCII.

ASCII constants are indicated by enclosing the appropriate character in single quotes ('). Any ASCII character can appear between the quotes, except for (1) control characters, having an ASCII code of under 20 hex; (2) the single quote character, ASCII code 27 hex; (3) the underscore character _, ASCII code 5F hex; and (4) the DEL character, 7F hex.

Numeric constants may be in any of four bases - 2, 8, 10, and 16. A specific base is indicated as follows:

###H indicates hexadecimal (base 16) - for example 1C7H
###Q indicates octal (base 8) - for example 62Q
###B indicates binary (base 2) - for example 10101B
###D or just ### indicates decimal (base 10) - for example 193D or 193

Regardless of base, all numeric constants `~must~` begin with a digit, 0-9. (This is to prevent ambiguity with labels.) Thus A07 hex would have to be written as 0A07H.

There is one special numeric constant, denoted by the symbol `$`. This constant is always equal to the address of the current line; that is, the memory location that the current line will be written into when it is loaded. Note that this reflects the address of the beginning of the current line, `~not~` the next line (as in some assemblers). As an example, consider that

```
0010      JMP  $
```

would cause an infinite loop, since it would jump to itself.

4.5.2.2 OPERATORS

ZSM recognizes 10 operators. They are as follows:

```
+  addition
-  subtraction, or negative (as in -1)
*  multiplication
/  division
%  modulo (remainder of division)
&  logical AND
!  logical OR
#  logical EXCLUSIVE-OR
>  rotate right (110101B>3 yields 101110B)
<  rotate left (1110110B<1 yields 1101101B)
```

All arithmetic operators treat their operands as unsigned 16-bit quantities, and answers are truncated to 16 bits. All logical operators perform their function on a bit-by-bit basis, and they also treat their operands as 16-bit values.

Operators combine with constants to form expressions. In an expression, all operators are evaluated in a strict left-to-right order, with no precedence of operators.

Thus consider the following situation:

```
TEST has been assigned the value 1000H.
INC has been assigned the value 6.
```

The expression encountered is TEST*6+INC!7<8.

The procedure would be TEST*6 (6000H) +INC (6006H) !7 (6007H) <8 (0760H). Thus the resulting value is 760H.

4.5.2.3 REGISTERS

The Z-80 has a number of registers, all of which have a specific symbolic reference. ZSM supports these references, as follows.

register designation

register B - B Also called BC for register-pair instructions
register C - C
register D - D Also called DE for register-pair instructions
register E - E
register H - H Also called HL for register-pair instructions
register L - L
accumulator- A
memory - M Also called (HL), but ZSM does not allow this.
A & flags - PSW Program Status Word, may also be called AF
Stack Ptr - SP
Index reg X- IX Also may be called X for brevity
Index reg Y- IY Also may be called Y for brevity

Of course, the Z-80 also has registers A', B', C', D', E', H', L', F', PC, I, and R, but these are never explicitly referred to in an instruction, so no special designation is needed.

4.5.2.4 PSEUDO-OPS

ZSM supports a large number of pseudo-ops. They will be explained now.

ORG Set origin

The ORG pseudo-op specifies where the object code is to be put. Assembled code and data is assembled starting at the address specified as the operand to the ORG pseudo-op, and proceeds upward, until the end of the program or another ORG. A program can contain as many ORGs as desired. Since ORG is handled in pass 1, any symbol appearing in the operand must already be defined.

LINK Link to a file

The LINK pseudo-op allows separate program files on the disk to be 'linked together' and assembled as one file. The LINK operand is a source file name, enclosed in single quotes. No drive specification is needed for the LINK file, as all units will be searched (starting with the unit the original source file is on) to locate the file. If the file is not found, a 'File not found' error will be issued, and the assembly aborted.

Linking to a file is like a subroutine; that is, when the linked-to file is exhausted, assembly of the original program will continue from where it was left off at. For example,

```
0010        LXI     H,4000H
0020        LINK    'TEST'
0030        MOV     A,M
```

will cause the entirety of the file TEST to be assembled between the LXI and the MOV.

Files that are linked to must not contain an END pseudo-op.

END End of assembly

The END pseudo-op indicates to ZSM that the end of the program has been reached. As such, it may be omitted, since the physical end of a program has the same effect.

In addition, though, an operand may be included. This operand, if present, indicates the starting address of the program. This address is not where the program is loaded, but instead where execution will begin. This allows the program to begin execution at any point in memory, rather than the beginning of the program. If this is omitted, then the beginning of the program is used as the starting address.

In order for the starting address to be effective, the object file would be changed to an implicit command file under MDOS (type 0C-0F).

EQU Equate

The EQU pseudo-op simply equates the label associated with it to the value of the operands.

```
0010 TEN        EQU    10
0020 TWENTY    EQU    2*10
```

The above code would cause the label TEN to have the value 10, and TWENTY to have the value 20.

REQ Request value

The REQ pseudo-op is similar to the EQU pseudo-op, only instead of an explicit value being specified, the system console is prompted for the value. The prompt is specified as the operand. For example,

```
0010 TEST        REQ    'Input:'
```

Would cause the message

Input:

to be displayed on the console during pass 1 of the assembly. The operator must then type the value to be associated with the label. For example, if the operator had typed '56H' in response to the prompt, then TEST would have a value of 56 hex.

PRT Print

The PRT pseudo-op allows information to be displayed on the console during pass 2. If operands are present, they are displayed, otherwise, just a carriage return/linefeed is printed. For example,

```
0010 TEST    EQU    7000H
0020        PRT    'This is a test ',TEST
```

would cause

This is a test 7000

to be printed on the console during pass 2.

TAB Tab settings

The TAB pseudo-op changes the tab settings for the assembly listing. Normally, they are at positions 15, 22, and 36. If it is desired to change them, then the TAB pseudo-op is used. It expects three operands, one for each tab setting. If a particular operand is zero, then that position is set to the default. The three settings represent the location of the opcode, operand, and comment fields respectively.

NLIST No list

The NLIST pseudo-op will cause code following it not to be listed. Note that this overrides any options which may have been specified in the command string; If the E option was used, nothing will be listed (errors or not) after a NLIST.

LIST List

The LIST pseudo-op cancels the effect of the NLIST pseudo-op. If there has been no NLIST, then this has no effect.

FORM Form feed

The FORM pseudo-op produces a formfeed in the listing when encountered.

IFF If false - conditional assembly

The block of code following the IFF pseudo-op will be assembled only if the operand evaluates to 0.

IFT If true - conditional assembly

The block of code following the IFT pseudo-op will be assembled only if the operand evaluates to anything other than 0.

ENDIF End of IF block

The ENDIF pseudo-op is used to mark the end of an IFT or IFF block.

DB Define byte

The DB pseudo-op assigns its operands to successive memory locations. Either numeric or ASCII operands may be present, but either one must evaluate to only 8 bits. This means that only one ASCII character may be included per operand. For example,

```
0010 LOCATION DB     1,20H,11B,'D',TEST,14
```

would put each operand into a successive memory location.

'Z' is a special case of the DB pseudo-op, and it is equivalent to DB 0. For example,

```
0010 XXX        Z                    and
0010 XXX        DB     0
```

are equivalent.

DW Define word

The DW pseudo-op is basically similar to DB, only it defines two bytes at a time, rather than 1. Also, the two bytes are in Intel standard low/high format.

DD Define data

The DD pseudo-op is exactly like DW, only the two bytes are put in high/low format.

DT Define text

The DT pseudo-op allows ASCII text to be put into memory. The desired text must be enclosed by single quotes. For example,

```
0010 TEST       DT     'ABCDEF'
```

would produce the following object code: 41 42 43 44 45 46 (hex).

DTH Define text terminated high

The DTH pseudo-op is like DT, only the last character is ORed with 80H before it is written out. In the above example, the last byte would be C6 hex.

DTZ Define text terminated with zero

The DTZ pseudo-op is like DT also, only it causes a byte of 00 to be

appended to the text string. Thus the example would be 41 42 43 44 45 46 00.

DS Define storage

The DS pseudo-op causes the assembler to skip over the number of bytes specified by the operand. Since the object file is scatter loaded, the area skipped over will remain undisturbed.

FILL fill storage

The FILL pseudo-op is similar to DS, only it fills the area with a constant, rather than skipping over it. The constant to fill with is specified with the second operand. For example,

```
0010          FILL  5,3
```

would produce the output

```
03 03 03 03 03.
```

4.5.3 ASSEMBLY ERRORS

There are ten assembly errors. Note that an error doesn't necessarily cause the program to assemble wrong, particularly if the error is a syntax error in something like a TAB statement. Nevertheless, all errors should be avoided.

The errors are as follows.

A Argument error - This is caused by an invalid character in an operand field, or an ASCII constant which is out of range.

D Duplicate label error - This indicates that a symbolic name was used more than once as a label. The first value will be used.

J Jump error - This indicates a relative jump (JR, JRZ, JRNZ, JRC, JRNC, DJNZ) to a label which is out of range. The relative jump should be replaced with an absolute one.

L Label error - This is caused by a label which contains invalid characters.

M Missing label error - This indicates that an EQU or REQ pseudo-op was encountered, but there was no label on the line. Obviously, a label is necessary for either of these.

O Opcode error - This is caused by an illegal or missing opcode.

R Register error - This indicates that an illegal value was found where a register was expected.

S Syntax error - This is caused by missing operands or improper use of operators.

U Undefined symbol error - This indicates that a symbol was used, but that the symbol has not been defined.

V Value error - This indicates that the value computed is out of range for the operation being used, specifically a two-byte instruction, or a DB.

4.5.4 INSTRUCTION SET

ZSM supports the complete Z-80 instruction set, using the TDL-style mnemonics. These mnemonics represent the Z-80 instruction set as a logical superset of the 8080 mnemonics. The reason that these 'superset' mnemonics were chosen over the Zilog mnemonics is for ease of use. All 8080 programs will run unmodified on ZSM, but they wouldn't on a Zilog-mnemonic assembler. In addition, someone familiar with 8080 mnemonics will find the superset easy to learn, since they are a logical extension of 8080 mnemonics.

One thing that is important to grasp is how indexing is handled. Under Zilog mnemonics, an operand might appear as (IX+d) where d is the offset and IX is the index register. Under ZSM, it would be d(X). Thus instead of

```
0010      LD   HL,(IX+12)
```

the following notation is used:

```
0010      LXI  H,12(X)
```

The same is true of IY, only it would appear as (Y) instead of (X). In addition, an offset of zero may be omitted entirely. That is, (IX+0) needn't be written as 0(X), it can simply be (X).

The next sections outline the instruction set. It is not meant as a tutorial on the Z-80, but rather a guide to the specific mnemonics used. Following that is a test program. If you have a Mostek or Zilog Z-80 Programming Manual, notice that in the back is an alphabetic list of all possible instructions. That list is in Zilog mnemonics. The test program herein is an exact duplicate of that list, only in the superset mnemonics. You are not expected to enter and assemble this program, but to use it as reference for the mnemonics.

In the following section, certain general conventions are used. They are as follows:

n	an 8 bit value
nn	a 16 bit value
d	an 8 bit value, specifically a displacement
r	register, such as A, B, C, D, E, H, L, M, d(X), d(Y)
I	one of the index registers, IX or IY (abbreviated X or Y)
rp	register pair, such as B, D, H, SP, PSW, IX, IY

b a bit, value 0 - 7

8 bit load group

Instruction	Zilog equivalent
MOV r,r	LD r,r
MOV r,M	LD r,(HL)
MOV r,d(I)	LD r,(I+d)
MOV M,r	LD (HL),r
MOV d(I),r	LD (I+d),r
MVI r,n	LD r,n
MVI M,n	LD (HL),n
MVI d(I),n	LD (I+d),n
LDA nn	LD A,(nn)
STA nn	LD (nn),A
LDAX rp	LD A,(rp)
STAX rp	LD (rp),A
LDAI	LD A,I
LDAR	LD A,R
LDIA	LD I,A
LDRA	LD R,A

16 bit load group

Instruction	Zilog equivalent
LXI rp,nn	LD rp,nn
LBCD nn	LD BC,(nn)
SBCD nn	LD (nn),BC
LDED nn	LD DE,(nn)
SDED nn	LD (nn),DE
LHLD nn	LD HL,(nn)
SHLD nn	LD (nn),HL
LSPD nn	LD SP,(nn)
SSPD nn	LD (nn),SP
LIXD nn	LD IX,(nn)
SIXD nn	LD (nn),IX
LIYD nn	LD IY,(nn)
SIYD nn	LD (nn),IY
SPHL	LD SP,HL
SPIX	LD SP,IX
SPIY	LD SP,IY
PUSH rp	PUSH rp
POP rp	POP rp

Exchange, block transfer, and search group

Instruction	Zilog equivalent
XCHG	EX DE,HL
EXAF	EX AF,AF'
EXX	EXX
XTHL	EX (SP),HL
XTIX	EX (SP),IX
XTIY	EX (SP),IY
LDI	LDI
LDIR	LDIR
LDD	LDD
LDDR	LDDR
CCI	CPI
CCIR	CPIR
CCD	CPD
CCDR	CPDR

Input / Output group

Instruction	Zilog equivalent
IN n	IN A,n
OUT n	OUT n,A
INP r	IN r,(C)
OUTP r	OUT (C),r
INI	INI
INIR	INIR
IND	IND
INDR	INDR
OUTI	OUTI
OUTIR	OTIR
OUTD	OUTD
OUTDR	OTDR

8 bit airthmetic/logical group

Instruction		Zilog equivalent
ADD	r	ADD A,r
ADD	M	ADD A,(HL)
ADD	d(I)	ADD A,(I+d)
ADI	n	ADD A,n
ADC	r	ADC A,r
ACI	n	ADC A,n
		(references to M and d(I) are like ADD)
SUB	r	SUB A,r
SUI	n	SUB A,n
SBB	r	SBC A,r
SBI	n	SBC A,n
ANA	r	AND A,r
ANI	n	AND A,n
ORA	r	OR A,r
ORI	n	OR A,n
XRA	r	XOR A,r
XRI	n	XOR A,n
CMP	r	CP A,r
CPI	n	CP A,n
INR	r	INC r
DCR	r	DEC r

16 bit arithmetic group

Instruction		Zilog equivalent
DAD	rp	ADD HL,rp
DADC	rp	ADC HL,rp
DSBC	rp	SBC HL,rp
DADX	rp	ADD IX,rp
DADY	rp	ADD IY,rp
INX	rp	INC rp
DCX	rp	DEC rp

General purpose arithmetic and control group

Instruction	Zilog equivalent
DAA	DAA
CMA	CPL
NEG	NEG
CMC	CCF
STC	SCF
NOP	NOP
HLT	HALT
DI	DI
EI	EI
IM0	IM 0
IM1	IM 1
IM2	IM 2

Rotate and shift group

Instruction		Zilog equivalent	
RLC		RLCA	
RAL		RLA	
RRC		RRCA	
RAR		RRA	
RLCR	r	RLC	r
RLCR	M	RLC	(HL)
RLCR	d(I)	RLC	(I+d)
			(references to M and d(I) are like RLCR)
RALR	r	RL	r
RRCR	r	RRC	r
RARR	r	RR	r
SLAR	r	SLA	r
SRAR	r	SRA	r
SRLR	r	SRL	r
RLD		RLD	
RRD		RRD	

Bit manipulation group

Instruction		Zilog equivalent	
BIT	b,r	BIT	b,r
BIT	b,M	BIT	b,(HL)
BIT	b,d(I)	BIT	b,(I+d)
RES	b,r	RES	b,r
			(references to M and d(I) are like BIT)
SET	b,r	SET	b,r

Jump, call, and return group

Instruction	Zilog equivalent
JMP nn	JP nn
JZ nn	JP Z,nn
JNZ nn	JP NZ,nn
JC nn	JP C,nn
JNC nn	JP NC,nn
JPO nn (or JNO)	JP PO,nn
JPE nn (or JO)	JP PE,nn
JM nn	JP M,nn
JP nn	JP P,nn
JR nn (or JMPR)	JR d
JRZ nn	JR Z,d
JRNZ nn	JR NZ,d
JRC nn	JR C,d
JRNC nn	JR NC,d
DJNZ nn	DJNZ d
PCHL	JP (HL)
PCIX	JP (IX)
PCIY	JP (IY)
CALL nn	CALL nn
CZ nn	CALL Z,nn
CNZ nn	CALL NZ,nn
CC nn	CALL C,nn
CNC nn	CALL NC,nn
CPO nn (or CNO)	CALL PO,nn
CPE nn (or CO)	CALL PE,nn
CM nn	CALL M,nn
CP nn	CALL P,nn
RET	RET
RZ	RET Z
RNZ	RET NZ
RC	RET C
RNC	RET NC
RPO (or RNO)	RET PO
RPE (or RO)	RET PE
RM	RET M
RP	RET P
RETI	RETI
RETN	RETN
RST n	RST m (m = 8 * n)

Addr	B1	B2	B3	B4	E	Line	Label	Opcd	Operand
0000						0001	;		
0000						0002	; Test file for ZSM		
0000						0003	; by Neale Brassell		
0000						0004	;		
0000						0005	; This uses all instructions		
0000						0006	;		
0000	8E					0007	A.0000	ADC	M
0001	DD	88	05			0008		ADC	IND(X)
0004	FD	86	05			0009		ADC	IND(Y)
0007	8F					0010		ADC	A
0008	8B					0011		ADC	B
0009	89					0012		ADC	C
000A	8A					0013		ADC	D
000B	8B					0014		ADC	E
000C	8C					0015		ADC	H
000D	8D					0016		ADC	L
000E	CE	20				0017		ACI	N
0010	ED	4A				0018		DADC	B
0012	ED	5A				0019		DADC	D
0014	ED	6A				0020		DADC	H
0016	ED	7A				0021		DADC	SP
0018						0022	;		
0018	86					0023	A.0018	ADD	M
0019	DD	86	05			0024		ADD	IND(X)
001C	FD	86	05			0025		ADD	IND(Y)
001F	87					0026		ADD	A
0020	80					0027		ADD	B
0021	81					0028		ADD	C
0022	82					0029		ADD	D
0023	83					0030		ADD	E
0024	84					0031		ADD	H
0025	85					0032		ADD	L
0026	C6	20				0033		ADI	N
0028	09					0034		DAD	B
0029	19					0035		DAD	D
002A	29					0036		DAD	H
002B	39					0037		DAD	SP
002C	DD	09				0038		DADX	B
002E	DD	19				0039		DADX	D
0030	DD	29				0040		DADX	X
0032	DD	39				0041		DADX	SP
0034	FD	09				0042		DADY	B
0036	FD	19				0043		DADY	D
0038	FD	29				0044		DADY	X
003A	FD	39				0045		DADY	SP
003C						0046	;		
003C	A6					0047	A.003C	ANA	M
003D	DD	A6	05			0048		ANA	IND(X)
0040	FD	A6	05			0049		ANA	IND(Y)
0043	A7					0050		ANA	A
0044	A0					0051		ANA	B
0045	A1					0052		ANA	C
0046	A2					0053		ANA	D
0047	A3					0054		ANA	E
0048	A4					0055		ANA	H
0049	A5					0056		ANA	L

Addr	B1	B2	B3	B4	E	Line	Label	Opcd	Operand
004A	E6	20				0057		ANI	N
004C						0058	;		
004C	CB	46				0059	A.004C	BIT	0,M
004E	DD	CB	05	46		0060		BIT	0,IND(X)
0052	FD	CB	05	46		0061		BIT	0,IND(Y)
0056	CB	47				0062		BIT	0,A
0058	CB	40				0063		BIT	0,B
005A	CB	41				0064		BIT	0,C
005C	CB	42				0065		BIT	0,D
005E	CB	43				0066		BIT	0,E
0060	CB	44				0067		BIT	0,H
0062	CB	45				0068		BIT	0,L
0064						0069	;		
0064	CB	4E				0070		BIT	1,M
0066	DD	CB	05	4E		0071		BIT	1,IND(X)
006A	FD	CB	05	4E		0072		BIT	1,IND(Y)
006E	CB	4F				0073		BIT	1,A
0070	CB	48				0074		BIT	1,B
0072	CB	49				0075		BIT	1,C
0074	CB	4A				0076		BIT	1,D
0076	CB	4B				0077		BIT	1,E
0078	CB	4C				0078		BIT	1,H
007A	CB	4D				0079		BIT	1,L
007C						0080	;		
007C	CB	56				0081		BIT	2,M
007E	DD	CB	05	56		0082		BIT	2,IND(X)
0082	FD	CB	05	56		0083		BIT	2,IND(Y)
0086	CB	57				0084		BIT	2,A
0088	CB	50				0085		BIT	2,B
008A	CB	51				0086		BIT	2,C
008C	CB	52				0087		BIT	2,D
008E	CB	53				0088		BIT	2,E
0090	CB	54				0089		BIT	2,H
0092	CB	55				0090		BIT	2,L
0094						0091	;		
0094	CB	5E				0092		BIT	3,M
0096	DD	CB	05	5E		0093		BIT	3,IND(X)
009A	FD	CB	05	5E		0094		BIT	3,IND(Y)
009E	CB	5F				0095		BIT	3,A
00A0	CB	58				0096		BIT	3,B
00A2	CB	59				0097		BIT	3,C
00A4	CB	5A				0098		BIT	3,D
00A6	CB	5B				0099		BIT	3,E
00A8	CB	5C				0100		BIT	3,H
00AA	CB	5D				0101		BIT	3,L
00AC						0102	;		
00AC	CB	66				0103		BIT	4,M
00AE	DD	CB	05	66		0104		BIT	4,IND(X)
00B2	FD	CB	05	66		0105		BIT	4,IND(Y)
00B6	CB	67				0106		BIT	4,A
00B8	CB	60				0107		BIT	4,B
00BA	CB	61				0108		BIT	4,C
00BC	CB	62				0109		BIT	4,D
00BE	CB	63				0110		BIT	4,E
00C0	CB	64				0111		BIT	4,H
00C2	CB	65				0112		BIT	4,L

Addr	B1	B2	B3	B4	Line Label	Opco	Operand	Addr	B1	B2	B3	B4	Line Label	Opco	Operand	
00C4					0113 ;			0136	FE	20				CPI	H	
00C4	CB	6E			0114	BIT	5,M	0138								
00C6	DD	CB	05	6E	0115	BIT	5,IND(X)	0138	ED	A9			0171	A.0138	CCD	
00CA	FD	CB	05	6E	0116	BIT	5,IND(Y)	013A	ED	B9			0172		CCDR	
00CE	CH	8F			0117	BIT	5,A	013C	ED	A1			0173		CCI	
00D0	CB	68			0118	BIT	5,B	013E	ED	B1			0174		CCIR	
00D2	CB	69			0119	BIT	5,C	0140					0175 ;			
00D4	CB	6A			0120	BIT	5,D	0140	2F				0176	A.0140	CMA	
00D6	CB	6B			0121	BIT	5,E	0141					0177 ;			
00E8	CB	6C			0122	BIT	5,H	0141	27				0178	A.0141	DAA	
00DA	CB	6D			0123	BIT	5,L	0142					0179 ;			
00DC					0124 ;			0142	35				0180	A.0142	DCR	M
00DC	CB	76			0125	BIT	6,M	0143	DD	35	05		0181		DCR	IND(X)
00DE	DD	CB	05	76	0126	BIT	6,IND(X)	0146	FD	35	05		0182		DCR	IND(Y)
00E2	FD	CB	05	76	0127	BIT	6,IND(Y)	0149	3D				0183		DCR	A
00E6	CB	77			0128	BIT	6,A	014A	05				0184		DCR	B
00E8	CB	70			0129	BIT	6,B	014B	0B				0185		DCX	B
00EA	CS	71			0130	BIT	6,C	014C	0D				0186		DCR	C
00EC	CS	72			0131	BIT	6,D	014D	15				0187		DCR	D
00EE	CB	73			0132	BIT	6,E	014E	1B				0188		DCX	D
00F0	CB	74			0133	BIT	6,H	014F	1D				0189		DCR	E
00F2	CB	75			0134	BIT	6,L	0150	25				0190		DCR	H
00F4					0135 ;			0151	2B				0191		DCX	H
00F4	CB	7E			0136	BIT	7,M	0152	DD	2B			0192		DCX	X
00F6	DD	CB	05	7E	0137	BIT	7,IND(X)	0154	FD	2B			0193		DCX	Y
00FA	FD	CB	05	7E	0138	BIT	7,IND(Y)	0156	2D				0194		DCR	L
00FE	CB	7F			0139	BIT	7,A	0157	3B				0195		DCX	SP
0100	CB	78			0140	BIT	7,B	0158					0196 ;			
0102	CB	79			0141	BIT	7,C	0158	F3				0197	A.0158	DI	
0104	CB	7A			0142	BIT	7,D	0159					0198 ;			
0106	CB	7B			0143	BIT	7,E	0159	10	2E			0199	A.0159	DJNZ	\$+DIS
0108	CB	7C			0144	BIT	7,H	015B					0200 ;			
010A	CB	7D			0145	BIT	7,L	015B	FB				0201	A.015B	EI	
010C					0146 ;			015C					0202 ;			
010C	DC	88	05		0147	A.010C	CC	NN	015C	E3			0203	A.015C	XTHL	
010F	FC	88	05		0148		CM	NN	015D	DD	E3		0204		XTIX	
0112	D4	88	05		0149		CNC	NN	015F	FD	E3		0205		XTIY	
0115	CD	88	05		0150		CALL	NN	0161	08			0206		EXAF	
0118	C4	88	05		0151		CNZ	NN	0162	EB			0207		XCHG	
011B	F4	88	05		0152		CP	NN	0163	D9			0208		EXX	
011E	EC	88	05		0153		CPE	NN	0164				0209 ;			
0121	E4	88	05		0154		CPO	NN	0164	76			0210	A.0164	HLT	
0124	CC	88	05		0155		CZ	NN	0165				0211 ;			
0127					0156 ;			0165	ED	46			0212	A.0165	TMO	
0127	3F				0157	A.0127	CNC		0167	ED	56		0213		IM1	
0128					0158 ;			0169	ED	5E			0214		IM2	
0128	BE				0159	A.0128	CMP	M	016B				0215 ;			
0129	DD	BE	05		0160		CMP	IND(X)	016B	ED	78		0216	A.016B	INP	A
012C	FD	BE	05		0161		CMP	IND(Y)	016D	DD	20		0217		IN	N
012F	BF				0162		CMP	A	016F	ED	40		0218		INP	B
0130	B6				0163		CMP	B	0171	ED	43		0219		INP	C
0131	B9				0164		CMP	C	0173	ED	50		0220		INP	D
0132	BA				0165		CMP	D	0175	ED	58		0221		INP	E
0133	BB				0166		CMP	E	0177	ED	60		0222		INP	H
0134	BC				0167		CMP	H	0179	ED	68		0223		INP	L
0135	BD				0168		CMP	I	017B				0224 ;			

Addr	Op	Opnd	Opnd
0178	SH	0225 A.0175	
017C	BD	0226	
017E	BD	0227	
0182	3C	0228	
0183	04	0229	
0184	03	0230	
0185	0C	0231	
0186	14	0232	
0187	13	0233	
0188	1C	0234	
0189	24	0235	
018A	23	0236	
018B	DD	0237	
018D	DD	0238	
018E	DD	0239	
018F	2C	0240	
0190	33	0241	
0191	BD	0242 A.0191	
0191	BD	0243	
0193	BD	0244	
0195	BD	0245	
0197	ED	0246	
0199		0247 A.0199	
019A	DD	0248	
019C	PD	0249	
019E	DA	0250	
01A1	FA	0251	
01A4	E2	0252	
01A7	C3	0253	
01AA	C2	0254	
01AD	F2	0255	
01B0	EA	0256	
01B3	E2	0257	
01B6	CA	0258	
01B9	3E	0259 A.01B9	
01BB	18	0261	
01BD	30	0262	
01BF	20	0263	
01C1	28	0264	
01C3	02	0265 A.01C3	
01C4	12	0267	
01C5	77	0268	
01C6	70	0269	
01C7	71	0270	
01C8	72	0271	
01C9	73	0272	
01CA	74	0273	
01CB	75	0274	
01CC	35	0275	
01CE	BD	0276	
01D1	BD	0277	
01D4	BD	0278	
01D7	BD	0279	
01DA	BD	0280	
0178	SH	0281	
017C	BD	0282	
017E	BD	0283	
0182	3C	0284	
0183	04	0285 A.0187	
0184	03	0286	
0185	0C	0287	
0186	14	0288	
0187	13	0289	
0188	1C	0290	
0189	24	0291	
018A	23	0292	
018B	DD	0293 A.0200	
018D	DD	0294	
018E	DD	0295	
018F	2C	0296	
0190	33	0297	
0191	BD	0298	
0191	BD	0299	
0193	BD	0300	
0195	BD	0301	
0197	ED	0302 A.021A	
0199		0303	
019A	DD	0304	
019C	PD	0305	
019E	DA	0306	
01A1	FA	0307	
01A4	E2	0308	
01A7	C3	0309	
01AA	C2	0310	
01AD	F2	0311	
01B0	EA	0312	
01B3	E2	0313	
01B6	CA	0314	
01B9	3E	0315	
01BB	18	0316	
01BD	30	0317	
01BF	20	0318	
01C1	28	0319 A.0233	
01C3	02	0320	
01C4	12	0321	
01C5	77	0322	
01C6	70	0323	
01C7	71	0324	
01C8	72	0325	
01C9	73	0326	
01CA	74	0327	
01CB	75	0328	
01CC	35	0329	
01CE	BD	0330	
01D1	BD	0331 A.0243	
01D4	BD	0332	
01D7	BD	0333	
01DA	BD	0334 A.024A	
0178	SH	0335	
017C	BD	0336	
017E	BD	0337	
0182	3C	0338	
0183	04	0339	
0184	03	0340	
0185	0C	0341	
0186	14	0342	
0187	13	0343	
0188	1C	0344	
0189	24	0345	
018A	23	0346	
018B	DD	0347	
018D	DD	0348	
018E	DD	0349	
018F	2C	0350	
0190	33	0351	
0191	BD	0352	
0191	BD	0353	
0193	BD	0354	
0195	BD	0355	
0197	ED	0356	
0199		0357	
019A	DD	0358	
019C	PD	0359	
019E	DA	0360	
01A1	FA	0361	
01A4	E2	0362	
01A7	C3	0363	
01AA	C2	0364	
01AD	F2	0365	
01B0	EA	0366	
01B3	E2	0367	
01B6	CA	0368	
01B9	3E	0369	
01BB	18	0370	
01BD	30	0371	
01BF	20	0372	
01C1	28	0373	
01C3	02	0374	
01C4	12	0375	
01C5	77	0376	
01C6	70	0377	
01C7	71	0378	
01C8	72	0379	
01C9	73	0380	
01CA	74	0381	
01CB	75	0382	
01CC	35	0383	
01CE	BD	0384	
01D1	BD	0385	
01D4	BD	0386	
01D7	BD	0387	
01DA	BD	0388	
0178	SH	0389	
017C	BD	0390	
017E	BD	0391	
0182	3C	0392	
0183	04	0393	
0184	03	0394	
0185	0C	0395	
0186	14	0396	
0187	13	0397	
0188	1C	0398	
0189	24	0399	
018A	23	0400	
018B	DD	0401	
018D	DD	0402	
018E	DD	0403	
018F	2C	0404	
0190	33	0405	
0191	BD	0406	
0191	BD	0407	
0193	BD	0408	
0195	BD	0409	
0197	ED	0410	
0199		0411	
019A	DD	0412	
019C	PD	0413	
019E	DA	0414	
01A1	FA	0415	
01A4	E2	0416	
01A7	C3	0417	
01AA	C2	0418	
01AD	F2	0419	
01B0	EA	0420	
01B3	E2	0421	
01B6	CA	0422	
01B9	3E	0423	
01BB	18	0424	
01BD	30	0425	
01BF	20	0426	
01C1	28	0427	
01C3	02	0428	
01C4	12	0429	
01C5	77	0430	
01C6	70	0431	
01C7	71	0432	
01C8	72	0433	
01C9	73	0434	
01CA	74	0435	
01CB	75	0436	
01CC	35	0437	
01CE	BD	0438	
01D1	BD	0439	
01D4	BD	0440	
01D7	BD	0441	
01DA	BD	0442	
0178	SH	0443	
017C	BD	0444	
017E	BD	0445	
0182	3C	0446	
0183	04	0447	
0184	03	0448	
0185	0C	0449	
0186	14	0450	
0187	13	0451	
0188	1C	0452	
0189	24	0453	
018A	23	0454	
018B	DD	0455	
018D	DD	0456	
018E	DD	0457	
018F	2C	0458	
0190	33	0459	
0191	BD	0460	
0191	BD	0461	
0193	BD	0462	
0195	BD	0463	
0197	ED	0464	
0199		0465	
019A	DD	0466	
019C	PD	0467	
019E	DA	0468	
01A1	FA	0469	
01A4	E2	0470	
01A7	C3	0471	
01AA	C2	0472	
01AD	F2	0473	
01B0	EA	0474	
01B3	E2	0475	
01B6	CA	0476	
01B9	3E	0477	
01BB	18	0478	
01BD	30	0479	
01BF	20	0480	
01C1	28	0481	
01C3	02	0482	
01C4	12	0483	
01C5	77	0484	
01C6	70	0485	
01C7	71	0486	
01C8	72	0487	
01C9	73	0488	
01CA	74	0489	
01CB	75	0490	
01CC	35	0491	
01CE	BD	0492	
01D1	BD	0493	
01D4	BD	0494	
01D7	BD	0495	
01DA	BD	0496	
0178	SH	0497	
017C	BD	0498	
017E	BD	0499	
0182	3C	0500	
0183	04	0501	
0184	03	0502	
0185	0C	0503	
0186	14	0504	
0187	13	0505	
0188	1C	0506	
0189	24	0507	
018A	23	0508	
018B	DD	0509	
018D	DD	0510	
018E	DD	0511	
018F	2C	0512	
0190	33	0513	
0191	BD	0514	
0191	BD	0515	
0193	BD	0516	
0195	BD	0517	
0197	ED	0518	
0199		0519	
019A	DD	0520	
019C	PD	0521	
019E	DA	0522	
01A1	FA	0523	
01A4	E2	0524	
01A7	C3	0525	
01AA	C2	0526	
01AD	F2	0527	
01B0	EA	0528	
01B3	E2	0529	
01B6	CA	0530	
01B9	3E	0531	
01BB	18	0532	
01BD	30	0533	
01BF	20	0534	
01C1	28	0535	
01C3	02	0536	
01C4	12	0537	
01C5	77	0538	
01C6	70	0539	
01C7	71	0540	
01C8	72	0541	
01C9	73	0542	
01CA	74	0543	
01CB	75	0544	
01CC	35	0545	
01CE	BD	0546	
01D1	BD	0547	
01D4	BD	0548	
01D7	BD	0549	
01DA	BD	0550	
0178	SH	0551	
017C	BD	0552	
017E	BD	0553	
0182	3C	0554	
0183	04	0555	
0184	03	0556	
0185	0C	0557	
0186	14	0558	
0187	13	0559	
0188	1C	0560	
0189	24	0561	
018A	23	0562	
018B	DD	0563	
018D	DD	0564	
018E	DD	0565	
018F	2C	0566	
0190	33	0567	
0191	BD	0568	
0191	BD	0569	
0193	BD	0570	
0195	BD	0571	
0197	ED	0572	
0199		0573	
019A	DD	0574	
019C	PD	0575	
019E	DA	0576	
01A1	FA	0577	
01A4	E2	0578	
01A7	C3	0579	
01AA	C2	0580	
01AD	F2	0581	
01B0	EA	0582	
01B3	E2	0583	
01B6	CA	0584	
01B9	3E	0585	
01BB	18	0586	
01BD	30	0587	
01BF	20	0588	
01C1	28	0589	
01C3	02	0590	
01C4	12	0591	
0			

Addr	B1	B2	B3	B4	E	Line	Label	Opcod	Operando	Addr	B1	B2	B3	B4	E	Line	Label	Opcod	Operando
0251	4F					0337		MOV	C,A	02A1	FD	2A	8B	05		0393	A.02A1	LJYD	NN
0252	48					0338		MOV	C,B	02A5	FD	21	8B	05		0394		LXI	Y,NN
0253	49					0339		MOV	C,C	02A9						0395			
0254	4A					0340		MOV	C,D	02A9	6E					0396	A.02A9	MOV	L,H
0255	4B					0341		MOV	C,E	02AA	D0	6E	05			0397		MOV	L,IND(X)
0256	4C					0342		MOV	C,H	02AD	FD	6E	05			0398		MOV	L,IND(Y)
0257	4D					0343		MOV	C,I	02B0	6F					0399		MOV	L,A
0258	0E	20				0344		MVI	C,N	02B1	68					0400		MOV	L,B
025A						0345				02B2	69					0401		MOV	L,C
025A	56					0346	A.025A	MOV	D,M	02B3	6A					0402		MOV	L,D
025B	DD	56	05			0347		MOV	D,IND(X)	02B4	6B					0403		MOV	L,E
025E	FD	56	05			0348		MOV	D,IND(Y)	02B5	6C					0404		MOV	L,H
0261	57					0349		MOV	D,A	02B6	6D					0405		MOV	L,L
0262	50					0350		MOV	D,B	02B7	2E	20				0406		MVI	L,N
0263	51					0351		MOV	D,C	02B9						0407			
0264	52					0352		MOV	D,D	02B9	ED	4F				0408	A.02B9	LDRA	
0265	53					0353		MOV	D,E	02BB						0409			
0266	54					0354		MOV	D,H	02BB	ED	7B	8B	05		0410	A.02BB	LSPD	NN
0267	55					0355		MOV	D,L	02BF	F9					0411		SPHL	
0268	16	20				0356		MVI	D,H	02C0	D0	F9				0412		SPIX	
026A						0357				02C2	FD	F9				0413		SPIY	
026A	ED	5B	8B	05		0358	A.026A	LDIED	NN	02C4	31	8B	05			0414		LXI	SP,NN
026E	11	8B	05			0359		LXI	D,NN	02C7						0415			
0271						0360				02C7	ED	AB				0416	A.02C7	LDD	
0271	5E					0361	A.0271	MOV	E,M	02C9	ED	80				0417		LDDH	
0272	DD	5E	05			0362		MOV	E,IND(X)	02CB	ED	A0				0418		LDI	
0275	FD	5E	05			0363		MOV	E,IND(Y)	02CD	ED	B0				0419		LDIR	
0278	5F					0364		MOV	E,A	02CF						0420			
0279	58					0365		MOV	E,B	02CF	ED	44				0421	A.02CF	NEG	
027A	59					0366		MOV	E,C	02D1						0422			
027B	5A					0367		MOV	E,D	02D1	00					0423	A.02D1	NOP	
027C	5B					0368		MOV	E,E	02D2						0424			
027D	5C					0369		MOV	E,H	02D2	B6					0425	A.02D2	ORA	M
027E	5D					0370		MOV	E,L	02D3	DD	B6	05			0426		ORA	IND(X)
027F	1E	20				0371		MVI	E,N	02D6	FD	B6	05			0427		ORA	IND(Y)
0281						0372				02D9	B7					0428		ORA	A
0281	66					0373	A.0281	MOV	H,M	02DA	B0					0429		ORA	B
0282	DD	66	05			0374		MOV	H,IND(X)	02DB	B1					0430		ORA	C
0285	FD	66	05			0375		MOV	H,IND(Y)	02DC	B2					0431		ORA	D
0288	67					0376		MOV	H,A	02DD	B3					0432		ORA	E
0289	60					0377		MOV	H,B	02DE	B4					0433		ORA	H
028A	61					0378		MOV	H,C	02DF	B5					0434		ORA	L
028B	62					0379		MOV	H,D	02E0	F6	20				0435		ORI	N
028C	63					0380		MOV	H,E	02E2						0436			
028D	64					0381		MOV	H,H	02E2	ED	BB				0437	A.02E2	OUTDR	
028E	65					0382		MOV	H,L	02E4	ED	B3				0438		OUTIR	
028F	26	20				0383		MVI	H,N	02E6						0439			
0291						0384				02E6	ED	79				0440	A.02E6	OUTP	A
0291	2A	8B	05			0385	A.0291	LJLD	NN	02E8	ED	41				0441		OUTP	B
0294	21	8B	05			0386		LXI	H,NN	02EA	ED	49				0442		OUTP	C
0297						0387				02EC	ED	51				0443		OUTP	D
0297	ED	47				0388	A.0297	LJLA		02EE	ED	59				0444		OUTP	E
0299						0389				02F0	ED	61				0445		OUTP	H
0299	DD	2A	8B	05		0389	A.0299	LJLD	NN	02F2	ED	69				0446		OUTP	L
029D	DD	21	8B	05		0391		LXI	X,NN	02F4	B3	20				0447		OUT	N
02A1						0392				02F6						0448			

Addr	B1	B2	B3	B4	Line	Label	Opnd	Operands	Addr	B1	B2	B3	B4	Line	Label	Opnd	Operands
02F6	ED	AD			0449	A.02F6	OUTD		0364	CB	9B			0505		RES	3,E
02F8	ED	A3			0450		OUTI		0366	CB	9C			0506		RES	3,H
02FA					0451				0368	CB	9D			0507		RES	3,L
02FA	F1				0452	A.02FA	POP	PSW	036A					0508			
02FB	C1				0453		POP	B	036A	CB	A6			0509		RES	4,H
02FC	D1				0454		POP	D	036C	DD	CB	05	A6	0510		RES	4,IND(X)
02FD	E1				0455		POP	H	0370	FD	CB	05	A6	0511		RES	4,IND(Y)
02FE	DD	E1			0456		POP	X	0374	CB	A7			0512		RES	4,A
0300	FD	E1			0457		POP	Y	0376	CB	A8			0513		RES	4,B
0302	F5				0458		PUSH	PSW	0378	CB	A1			0514		RES	4,C
0303	C5				0459		PUSH	B	037A	CB	A2			0515		RES	4,D
0304	D5				0460		PUSH	D	037C	CB	A3			0516		RES	4,E
0305	E5				0461		PUSH	H	037E	CB	A4			0517		RES	4,H
0306	DD	E5			0462		PUSH	X	0380	CB	A5			0518		RES	4,L
0308	FD	E5			0463		PUSH	Y	0382					0519			
030A					0464				0382	CB	AE			0520		RES	5,M
030A	CB	86			0465	A.030A	RES	0,H	0384	DD	CB	05	AE	0521		RES	5,IND(X)
030C	DD	CB	05	86	0466		RES	0,IND(X)	0388	FD	CB	05	AE	0522		RES	5,IND(Y)
0310	FD	CB	05	86	0467		RES	0,IND(Y)	038C	CB	AF			0523		RES	5,A
0314	CB	87			0468		RES	0,A	038E	CB	A8			0524		RES	5,B
0316	CB	80			0469		RES	0,B	0390	CB	A9			0525		RES	5,C
0318	CB	81			0470		RES	0,C	0392	CB	AA			0526		RES	5,D
031A	CB	82			0471		RES	0,D	0394	CB	AB			0527		RES	5,E
031C	CB	83			0472		RES	0,E	0396	CB	AC			0528		RES	5,H
031E	CB	84			0473		RES	0,H	0398	CB	AD			0529		RES	5,L
0320	CB	85			0474		RES	0,L	039A					0530			
0322					0475				039A	CB	B6			0531		RES	6,M
0322	CB	8E			0476		RES	1,M	039C	DD	CB	05	B6	0532		RES	6,IND(X)
0324	DD	CB	05	8E	0477		RES	1,IND(X)	03A0	FD	CB	05	B6	0533		RES	6,IND(Y)
0328	FD	CB	05	8E	0478		RES	1,IND(Y)	03A4	CB	B7			0534		RES	6,A
032C	CB	8F			0479		RES	1,A	03A6	CB	B8			0535		RES	6,B
032E	CB	88			0480		RES	1,B	03A8	CB	B1			0536		RES	6,C
0330	CB	89			0481		RES	1,C	03AA	CB	B2			0537		RES	6,D
0332	CB	8A			0482		RES	1,D	03AC	CB	B3			0538		RES	6,E
0334	CB	8B			0483		RES	1,E	03AE	CB	B4			0539		RES	6,H
0336	CB	8C			0484		RES	1,H	03B0	CB	B5			0540		RES	6,L
0338	CB	8D			0485		RES	1,L	03B2					0541			
033A					0486				03B2	CB	BE			0542		RES	7,M
033A	CB	96			0487		RES	2,M	03B4	DD	CB	05	BE	0543		RES	7,IND(X)
033C	DD	CB	05	96	0488		RES	2,IND(X)	03B8	FD	CB	05	BE	0544		RES	7,IND(Y)
0340	FD	CB	05	96	0489		RES	2,IND(Y)	03BC	CB	BF			0545		RES	7,A
0344	CB	97			0490		RES	2,A	03BE	CB	B8			0546		RES	7,B
0346	CB	90			0491		RES	2,B	03C0	CB	B9			0547		RES	7,C
0348	CB	91			0492		RES	2,C	03C2	CB	BA			0548		RES	7,D
034A	CB	92			0493		RES	2,D	03C4	CB	BB			0549		RES	7,E
034C	CB	93			0494		RES	2,E	03C6	CB	BC			0550		RES	7,H
034E	CB	94			0495		RES	2,H	03C8	CB	BD			0551		RES	7,L
0350	CB	95			0496		RES	2,L	03CA					0552			
0352					0497				03CA	CB	C9			0553	A.03CA	RET	
0352	CB	9E			0498		RES	3,M	03CB	DD				0554		RC	
0354	DD	CB	05	9E	0499		RES	3,IND(X)	03CC	FD				0555		RM	
0358	FD	CB	05	9E	0500		RES	3,IND(Y)	03CD	DD				0556		RMC	
035C	CB	9F			0501		RES	3,A	03CE	DD				0557		RWZ	
035E	CB	98			0502		RES	3,B	03CF	FD				0558		RP	
0360	CB	93			0503		RES	3,C	03D0	ED				0559		RPE	
0362	CB	9A			0504		RES	3,D	03D1	ED				0560		RPO	

REV. 8.1 2/5/79

4-67J

Addr	B1	B2	B3	B4	E	Line	Label	Opnd	Operand
0302	CB					0561		RZ	
0303						0562			
0304	ED	4D				0563	A.03D3	RETI	
0305	ED	45				0564		RETR	
0307						0565			
0307	CB	16				0566	A.03D7	HALR	M
0309	DD	CB	05	16		0567		HALR	IND(X)
030D	FD	CB	05	16		0568		HALR	IND(Y)
03E1	CB	17				0569		HALR	A
03E3	CB	10				0570		HALR	B
03E5	CB	11				0571		HALR	C
03E7	CB	12				0572		HALR	D
03E9	CB	13				0573		HALR	E
03EB	CB	14				0574		HALR	H
03ED	CB	15				0575		HALR	L
03EF						0576			
03EF	17					0577	A.03EF	RAL	
03F0						0578			
03F0	CB	06				0579	A.03F0	RLCR	M
03F2	DD	CB	05	06		0580		RLCR	IND(X)
03F6	FD	CB	05	06		0581		RLCR	IND(Y)
03FA	CB	07				0582		RLCR	A
03FC	CB	00				0583		RLCR	B
03FE	CB	01				0584		RLCR	C
0400	CB	02				0585		RLCR	D
0402	CB	03				0586		RLCR	E
0404	CB	04				0587		RLCR	H
0406	CB	05				0588		RLCR	L
0408						0589			
0408	07					0590	A.0408	RLC	
0409						0591			
0409	ED	6F				0592	A.0409	RLD	
040B						0593			
040B	CB	1E				0594	A.040B	RARR	M
040D	DD	CB	05	1E		0595		RARR	IND(X)
0411	FD	CB	05	1E		0596		RARR	IND(Y)
0415	CB	1F				0597		RARR	A
0417	CB	18				0598		RARR	B
0419	CB	19				0599		RARR	C
041B	CB	1A				0600		RARR	D
041D	CB	1B				0601		RARR	E
041F	CB	1C				0602		RARR	H
0421	CB	1D				0603		RARR	L
0423						0604			
0423	1F					0605	A.0423	RAR	
0424						0606			
0424	CB	0E				0607	A.0424	RRCR	M
0426	DD	CB	05	0E		0608		RRCR	IND(X)
042A	FD	CB	05	0E		0609		RRCR	IND(Y)
042E	CB	0F				0610		RRCR	A
0430	CB	08				0611		RRCR	B
0432	CB	09				0612		RRCR	C
0434	CB	0A				0613		RRCR	D
0436	CB	0B				0614		RRCR	E
0438	CB	0C				0615		RRCR	H
043A	CB	0D				0616		RRCR	L

Addr	B1	B2	B3	B4	E	Line	Label	Opnd	Operand
043C						0617			
043C	CF					0618	A.043C	RHC	
043D						0619			
043D	ED	67				0620	A.043D	RHD	
043F						0621			
043F	C7					0622	A.043F	RST	0
0440	CF					0623		RST	1
0441	D7					0624		RST	2
0442	DF					0625		RST	3
0443	E7					0626		RST	4
0444	EF					0627		RST	5
0445	F7					0628		RST	6
0446	FF					0629		RST	7
0447						0630			
0447	9E					0631	A.0447	SBB	M
0448	DD	9E	05			0632		SBB	IND(X)
044B	FD	9E	05			0633		SBB	IND(Y)
044E	9F					0634		SBB	A
044F	9B					0635		SBB	B
0450	99					0636		SBB	C
0451	9A					0637		SBB	D
0452	9B					0638		SBB	E
0453	9C					0639		SBB	H
0454	9D					0640		SBB	L
0455	DE	20				0641		SBI	N
0457						0642			
0457	ED	42				0643	A.0457	DSBC	B
0459	ED	52				0644		DSBC	D
045B	ED	62				0645		DSBC	H
045D	ED	72				0646		DSBC	SP
045F						0647			
045F	37					0648	A.045F	STC	
0460						0649			
0460	CB	C6				0650	A.0460	SET	0,M
0462	DD	CB	05	C6		0651		SET	0,IND(X)
0466	FD	CB	05	C6		0652		SET	0,IND(Y)
046A	CB	C7				0653		SET	0,A
046C	CB	C0				0654		SET	0,B
046E	CB	C1				0655		SET	0,C
0470	CB	C2				0656		SET	0,D
0472	CB	C3				0657		SET	0,E
0474	CB	C4				0658		SET	0,H
0476	CB	C5				0659		SET	0,L
047B						0660			
047B	CB	CE				0661		SET	1,M
047A	DD	CB	05	CE		0662		SET	1,IND(X)
047E	FD	CB	05	CE		0663		SET	1,IND(Y)
0482	CB	CF				0664		SET	1,A
0484	CB	CB				0665		SET	1,B
0486	CB	C9				0666		SET	1,C
0488	CB	CA				0667		SET	1,D
048A	CB	CB				0668		SET	1,E
048C	CB	CC				0669		SET	1,H
048E	CB	CD				0670		SET	1,L
0490						0671			
0490	CB	DB				0672		SET	2,M

Addr	B1	B2	B3	B4	E	Line Label	Opnd	Operand
0492	DD	CB	05	D6		0673	SET	2,IND(X)
0496	FD	CB	05	D6		0674	SET	2,IND(Y)
049A	CB	D7				0675	SET	2,A
049C	CB	D0				0676	SET	2,B
049E	CB	D1				0677	SET	2,C
04A0	CB	D2				0678	SET	2,D
04A2	CB	D3				0679	SET	2,E
04A4	CB	D4				0680	SET	2,H
04A6	CB	D5				0681	SET	2,I
04A8						0682 ;		
04A8	CB	DE				0683	SET	3,M
04AA	DD	CB	05	DE		0684	SET	3,IND(X)
04AC	FD	CB	05	DE		0685	SET	3,IND(Y)
04B2	CB	DF				0686	SET	3,A
04B4	CB	D8				0687	SET	3,B
04B6	CB	D9				0688	SET	3,C
04B8	CB	DA				0689	SET	3,D
04BA	CB	DB				0690	SET	3,E
04BC	CB	DC				0691	SET	3,H
04BE	CB	DD				0692	SET	3,L
04C0						0693 ;		
04C0	CB	E6				0694	SET	4,M
04C2	DD	CB	05	E6		0695	SET	4,IND(X)
04C6	FD	CB	05	E6		0696	SET	4,IND(Y)
04CA	CB	E7				0697	SET	4,A
04CC	CB	E0				0698	SET	4,B
04CE	CB	E1				0699	SET	4,C
04D0	CB	E2				0700	SET	4,D
04D2	CB	E3				0701	SET	4,E
04D4	CB	E4				0702	SET	4,H
04D6	CB	E5				0703	SET	4,I
04D8						0704 ;		
04D8	CB	EE				0705	SET	5,M
04DA	DD	CB	05	EE		0706	SET	5,IND(X)
04DC	FD	CB	05	EE		0707	SET	5,IND(Y)
04E2	CB	EF				0708	SET	5,A
04E4	CB	E8				0709	SET	5,B
04E6	CB	E9				0710	SET	5,C
04E8	CB	EA				0711	SET	5,D
04EA	CB	EB				0712	SET	5,E
04EC	CB	EC				0713	SET	5,H
04EE	CB	ED				0714	SET	5,L
04F0						0715 ;		
04F0	CB	F6				0716	SET	6,M
04F2	DD	CB	05	F6		0717	SET	6,IND(X)
04F6	FD	CB	05	F6		0718	SET	6,IND(Y)
04FA	CB	F7				0719	SET	6,A
04FC	CB	F0				0720	SET	6,B
04FE	CB	F1				0721	SET	6,C
0500	CB	F2				0722	SET	6,D
0502	CB	F3				0723	SET	6,E
0504	CB	F4				0724	SET	6,H
0506	CB	F5				0725	SET	6,L
0508						0726 ;		
0508	CB	FE				0727	SET	7,H
050A	DD	CB	05	FE		0728	SET	7,IND(X)

Addr	B1	B2	B3	B4	E	Line Label	Opnd	Operand
050E	FD	CB	05	FE		0729	SET	7,IND(Y)
0512	CB	FF				0730	SET	7,A
0514	CB	F8				0731	SET	7,B
0516	CB	F9				0732	SET	7,C
0518	CB	FA				0733	SET	7,D
051A	CB	FB				0734	SET	7,E
051C	CB	FC				0735	SET	7,H
051E	CB	FD				0736	SET	7,L
0520						0737 ;		
0520	CB	26				0738 A.0520	SLAR	M
0522	DD	CB	05	26		0739	SLAR	IND(X)
0526	FD	CB	05	26		0740	SLAR	IND(Y)
052A	CB	27				0741	SLAR	A
052C	CB	20				0742	SLAR	B
052E	CB	21				0743	SLAR	C
0530	CB	22				0744	SLAR	D
0532	CB	23				0745	SLAR	E
0534	CB	24				0746	SLAR	H
0536	CB	25				0747	SLAR	L
0538						0748 ;		
0538	CB	2E				0749 A.0538	SRAR	M
053A	DD	CB	05	2E		0750	SRAR	IND(X)
053E	FD	CB	05	2E		0751	SRAR	IND(Y)
0542	CB	2F				0752	SRAR	A
0544	CB	28				0753	SRAR	B
0546	CB	29				0754	SRAR	C
0548	CB	2A				0755	SRAR	D
054A	CB	2B				0756	SRAR	E
054C	CB	2C				0757	SRAR	H
054E	CB	2D				0758	SRAR	L
0550						0759 ;		
0550	CB	3E				0760 A.0550	SRLR	M
0552	DD	CB	05	3E		0761	SRLR	IND(X)
0556	FD	CB	05	3E		0762	SRLR	IND(Y)
055A	CB	3F				0763	SRLR	A
055C	CB	38				0764	SRLR	B
055E	CB	39				0765	SRLR	C
0560	CB	3A				0766	SRLR	D
0562	CB	3B				0767	SRLR	E
0564	CB	3C				0768	SRLR	H
0566	CB	3D				0769	SRLR	L
0568						0770 ;		
0568	96					0771 A.0568	SUB	M
0569	DD	96	05			0772	SUB	IND(X)
056C	FD	96	05			0773	SUB	IND(Y)
056F	97					0774	SUB	A
0570	90					0775	SUB	B
0571	91					0776	SUB	C
0572	92					0777	SUB	D
0573	93					0778	SUB	E
0574	94					0779	SUB	H
0575	95					0780	SUB	I
0576	DD	2D				0781	SUI	N
0578						0782 ;		
0578	AE					0783 A.0578	XRA	M
0579	DD	AE	05			0784	XRA	IND(X)

4-67 K

Rev. 8.1 2/5/79

Addr	B1	B2	B3	B4	E	Line	Label	Oped	Operand
057C	FD	AE	05			0785		XRA	IND(Y)
057F	AF					0786		XRA	A
0580	A8					0787		XRA	B
0581	A9					0788		XRA	C
0582	AA					0789		XRA	D
0583	AB					0790		XRA	E
0584	AC					0791		XRA	H
0585	AD					0792		XRA	L
0586	EE	20				0793		XRI	N
0588						0794			;
0588						0795			; Now for the definitions
0588						0796			;
0588		0020		=		0797	N	EQU	20H
0588	00	00				0798	NN	DW	0
058A		0005		=		0799	IND	EQU	5
058A		0030		=		0800	DIS	EQU	30H
058A						0801			;
058A						0802	A.058A	END	

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.

THIS PAGE LEFT BLANK INTENTIONALLY.

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 overlaid 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 AS(248)
20 Z$=CHAR$(13):REM CARRIAGE RET
30 OPEN 1 "N:TEXTFILE":REM NEW FILE
40 INPUT AS:REM GET A LINE OF TEXT FROM CONSOLE
50 IF AS="EXIT" THEN 80:REM END INPUT BY TYPING EXIT
60 PUT 1 AS+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,0 ;@ASCBUFF0
0080 CALL @TRANSFILENAME ;MOVE INTO @ASCIIBUFFER
0090 MVI B,0 ;FILE NUMBER
0100 LDA @DRIVEN0 ;UNIT NUMBER
0110 MOV C,A ;INTO C FOR OPEN
0120 LXI H,@FILEBUFFER0 ;USE SYSTEM BUFFER 0
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 0FCH ;TYPE NOT ATTRIBUTES
0190 ORA A ;BASIC DATA FILES=0
0200 MVI A,17 ;WRONG FILE TYPE MESSAGE
0210 JNZ @DISKERROR ;ERROR
0220 NEXTCHR MVI B,0 ;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 0DH ;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 60, 140, 160, 210, 240, and 310-340.

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 files 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 cannot be used with non-808 280 code.

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) When DEL or BACKSPACE 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 (CNTRL/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 program's 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
00 ZCMEH 0000 0000 0000 1234 00 00 00 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  @B  @D  @H  @SP
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
01 2354
03 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 3014
```

```
A  FLAGS  BC  DE  HL  SP  @B  @D  @H  @SP
00  Z  C  0012 0341 3674 0195 00  00  00  3054
3507 JMP 3643
*
```

Program execution was started at location 3014 (hex). A breakpoint was encountered at location 3507 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 0000 00 00 00 0000
3000 DCR B
*00 1F00 0000 0000 0000 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
00 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.

```
      Call MESSAGE
TEXT  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 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	0010	LXI	H,280H
3005	CD	30	0020	CALL	SUB
3008	25		0030	DCR	H
3009	C2	05 30	0040	JNZ	LOOP
300C	7D		0050	MOV	A,L
300D	0F		0060	RRC	
300E	6F		0070	MOV	L,A
300F	D2	05 30	0080	JNC	LOOP
3012	C9		0090	RET	
3013	F5		0100	PUSH	PSW
3014	7C		0110	MOV	A,H
3015	B5		0120	ORA	L
3016	F1		0130	POP	PSW
3017	C9		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 H
*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 (2C-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 (2C-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 output the message INPUT OVERFLOW and cancel the entire line.

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) when DEL or 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 (CNTRL/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 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 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 depressing the ESC key. 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 renumbers 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 too 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 "*" ERROR 5200
5020 AS="ZZ9":BS="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 L/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 " " " "). 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)

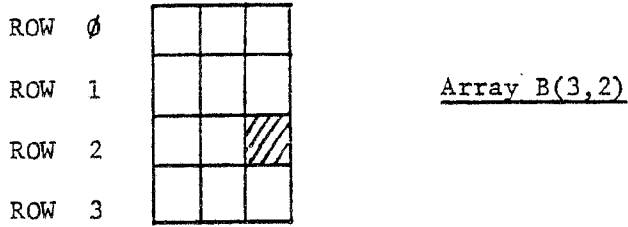
An element of a one dimensional array is referenced by the array name and by the index of the element within the array, enclosed in parentheses. The 4th element of array A in the above example is A (3). The index may be specified by a constant, as in this example, a numeric variable, or a numeric expression.

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: (-)xxxxxxx \textasciitilde
Decimal : (-)xxx ... x.xxx \textasciitilde
Scientific: (-)n.xxxxx E(-)TT \textasciitilde

(-) = minus sign if negative, blank if positive
x = digit position
n = one non-zero digit
E = signifies exponent
TT = exponent
 \textasciitilde = 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, i.e., by trying to output 2 strings of 250 characters with the same print statement. The characters in excess of 250 are truncated and the message "WARNING--TRUNCATED OUTPUT" is output.

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>
↑	Exponentiation
/	Division
*	Multiplication
\	Integer Division ($X \setminus Y = \text{Int}(X/Y)$)
-	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>															
AND	VAL 1 AND VAL 2	<table border="1"><thead><tr><th>VAL 1</th><th>VAL 2</th><th>RESULT</th></tr></thead><tbody><tr><td>True</td><td>True</td><td>True</td></tr><tr><td>True</td><td>False</td><td>False</td></tr><tr><td>False</td><td>True</td><td>False</td></tr><tr><td>False</td><td>False</td><td>False</td></tr></tbody></table>	VAL 1	VAL 2	RESULT	True	True	True	True	False	False	False	True	False	False	False	False
VAL 1	VAL 2	RESULT															
True	True	True															
True	False	False															
False	True	False															
False	False	False															

<u>Operator</u>	<u>Expression</u>	<u>Truth Table</u>															
OR	VAL 1 OR VAL 2	<table border="1"><thead><tr><th>VAL 1</th><th>VAL 2</th><th>RESULT</th></tr></thead><tbody><tr><td>True</td><td>True</td><td>True</td></tr><tr><td>True</td><td>False</td><td>True</td></tr><tr><td>False</td><td>True</td><td>True</td></tr><tr><td>False</td><td>False</td><td>False</td></tr></tbody></table>	VAL 1	VAL 2	RESULT	True	True	True	True	False	True	False	True	True	False	False	False
VAL 1	VAL 2	RESULT															
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>						
NOT	NOT VAL	<table border="1"><thead><tr><th>VAL</th><th>RESULT</th></tr></thead><tbody><tr><td>True</td><td>False</td></tr><tr><td>False</td><td>True</td></tr></tbody></table>	VAL	RESULT	True	False	False	True
VAL	RESULT							
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) AND 1: 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 $x - (y * INT(x/y))$. Both x and y must be numeric expressions.

Table 5.2 (cont)

Function Reference	Value
RND(x)	<p>Generates a pseudo random number between 0 and 1. The argument x is a numeric expression which controls the number generated as follows:</p> <p>If x is non zero, RND generates a number using x as the seed. If x=0, the last random number generated is used as the seed. Repeatedly calling RND with x=0 generates a sequence of pseudo random numbers.</p>
SGN(x)	<p>+1 if the sign of x is positive, -1 if the sign of x is negative, 0 if x is 0.</p>
SIN(x)	<p>The sine of x where x is a numeric expression in radians.</p>
SQR(x)	<p>The positive square root of x, where x is a positive numeric expression.</p>
TAN(x)	<p>The tangent of x, where x is a numeric expression in radians.</p>

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> <ul style="list-style-type: none"> 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>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 \emptyset 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 \emptyset .

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
  7
? 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 04C6 RSIZE EQU ARG1+9
0000 04C8 ISIZE EQU ARG1+10
0000 04CA SSIZE EQU ARG1+11
0000 *
0000 *
0000 *      CRG      6240H
0000 *
0000 * THIS DEMO ACCEPTS TWO ARGUMENTS
0000 * WHICH ARE STRINGS AND RETURNS
0000 * ARG1 CONCATENATED WITH ARG2.
0000 *
0000 *
0000 6240 3A 04 04 NBRCK LDA NARGS ;CHECK FOR TWO
0000 6243 FE 02 CPI 2 ;ARGUMENTS.
0000 6245 02 8D 03 JNZ NBRER ;IF NOT TWO - ERROR.
0000 6248 2A 30 04 TYPCK LHLD ARG1 ;ELSE, CHECK TYPE OF
0000 624B 7E MOV A,M ;ARG1. IT MUST
0000 624C FE 03 CPI 3 ;BE A STRING.
0000 624E 02 87 00 JNZ TYPERR ;IF NOT - ERROR.
0000 6251 2A BF 04 LHLD ARG2 ;ELSE, CHECK ARG2
0000 6254 7E MOV A,M ;IT ALSO MUST
0000 6256 FE 03 CPI 3 ;BE A STRING.
0000 6257 02 87 00 JNZ TYPERR ;IF NOT - ERROR.
0000 *
0000 * BOTH ARGUMENTS ARE VALID STRINGS
0000 *
0000 625A 11 A0 01 LXI D,RESULT ;SETUP RETURN
0000 625D 3E 03 MVI A,3 ;PARAMETER AS A
0000 625F 12 STAX D ;STRING TYPE.
0000 6260 13 INX D ;SKIP OVER
0000 6261 13 INX D ;LENGTH FOR
0000 6262 13 INX D ;NOW
0000 6263 AF XRA A ;ZERO LENGTH
0000 6264 47 MOV B,A ;COUNTER.
0000 6265 2A BC 04 LHLD ARG1 ;MOVE FIRST
0000 6268 0D 79 00 MSTP CALL MOVE ;ARGUMENT TO RESULT
0000 626B 2A BF 04 LHLD ARG2 ;MOVE SECOND
0000 626E 0D 79 00 CALL MOVE ;ARGUMENT TO RESULT
0000 6271 78 MOV A,B ;GET LENGTH COUNT
0000 6272 32 A1 01 STA RESULT+1 ;PUT COUNT INTO
0000 6275 32 A2 01 STA RESULT+2 ;RESULT.
0000 6278 09 RET ;DONE. RETURN TO BASIC

```

```

6279
6279
6279
6279
6279
6279
6279 23
627A 23
627B 4E
627C 23
627D 7E
627E 12
627F 13
6280 23
6281 04
6282 0D
6283 C2 7D 62
6286 C9
6287
6287 21 9E 60
628A C3 90 60
628D
628D 21 AB 60
6290 11 A0 01
6293 3E 03
6295 12
6296 13
6297 13
6298 13
6299 AF
629A 47
629B C3 68 60
629E
629E
629E
629E 00 00 2A
62A1 54 59 50
62A4 45 20 45
62A7 52 52 4F
62AA 52
62AE
62AB 00 00 0E
62AE 41 52 47
62B1 55 4D 45
62B4 4E 54 20
62B7 45 52 52
62BA 4F 52
62BC
62BC
*
* MOVE ARGUMENTS TO RESULT.
* HL REGISTERS HAS ARGUMENT ADDRESS.
* DE REGISTERS HAS POSITION IN RESULT.
* B REGISTER IS COUNT
*
MOVE INX H ;SKIP TYPE
INX H ;SKIP MAX LENGTH
MOV C,M ;GET LENGTH OF STRING
INX H
MOVE1 MOV A,M ;GET CHARACTER
STAX D ;PUT IT INTO RESULT
INX D ;NEXT
INX H
INR B ;COUNT +1
DCR C ;LENGTH -1
JNZ MOVE1 ;LOOP TILL DONE
RET ;DONE
*
*
TYPERR LXI H,TYPMSG
JMP EMSG
*
NBRER LXI H,NBRMSG
EMSG LXI D,RESULT ;PUT MESSAGE IN RESULT
MVI A,3 ;STRING TYPE
STAX D
INX D
INX D
INX D
XRA A ;ZERO COUNT
MOV B,A
JMP MSTR ;MOVE TO RESULT
*
* ERROR MESSAGES
*
TYPMSG DB 0,0,10
DT 'TYPE ERROR'
*
NBRMSG DB 0,0,14
DT 'ARGUMENT ERROR'
*
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$)
80 PRINT C$
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.

5.19 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<=3 AND B+3<5 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>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) $\left. \begin{array}{l} A \\ B \\ C \end{array} \right\}$ 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 $\left\{ \begin{array}{l} \text{numeric constant} \\ \text{string constant} \end{array} \right\}$, $\left\{ \begin{array}{l} \text{numeric constant} \\ \text{string constant} \end{array} \right\}$, . . .

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)

1Ø DIM A (2,4)
2Ø DIM B%(2,3,4,5)
3Ø DIM A\$(4Ø)
4Ø DIM A\$(2,3,4Ø)

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
1ØØØØ 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 TO numeric [STEP numeric
variable expression expression 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 { linenumbr
 { 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.

5.20.12 INPUT ["promptstring"^{{;}}] variable list

```
10 INPUT A,AS  
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 AS = "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 30. 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 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
```

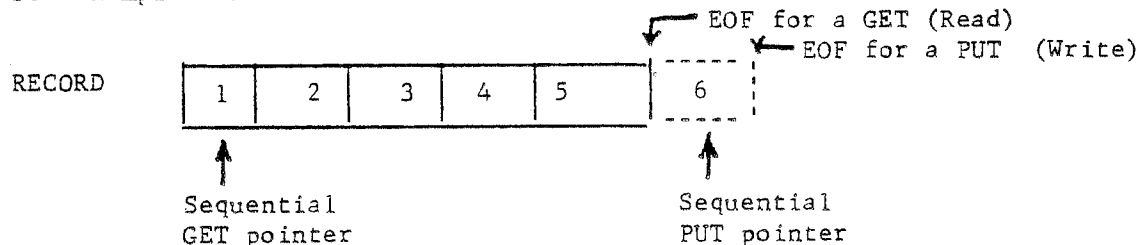
The OPEN statement opens the specified file for access by disk I/O statements. The file is selected by the string expression which must evaluate to the form:

```
"[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 filename 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(;).

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 first 250 characters will be written and the remaining characters will be truncated. A warning message WARNING - TRUNCATED OUTPUT will be output to the terminal.

5.21.3.3 GET filename 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 filename expression and assigns the values read to the variable list. The filename 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 filename

```
100 CLOSE 1
```

The CLOSE statement causes the file specified by the filename expression to be closed for disk I/O. The filename 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 filename. 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 filename

100 FREESPACE 1

The FREESPACE statement de-allocates any tracks allocated to the file referenced by filename which are beyond the current end of file. Filename 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 (filename) = numeric expression

50 GETSEEK (1) = 20

The GETSEEK statement sets the sequential GET pointer associated with the filename to the value of the numeric expression. The filename 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 (filename) = numeric expression

100 PUTSEEK (2) = 30

The PUTSEEK statement sets the sequential PUT pointer associated with the filename to the value of the numeric expression. The filename 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.

5.21.3.10 RENAME (filenumber) = string expression

100 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 0 - 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 0 - 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 11 - Invalid File Name 12 - Printer Attention <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)

```
1Ø OPEN 1 "*P" PAGESIZE 66 ENDPAGE 9ØØ
2Ø OPEN 2 "*T"
3Ø 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 linenumber 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 filename 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 filename. The filename 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 filename

```
90 CLOSE 6
99 CLOSE 2
```

The CLOSE statement causes the file specified by the filename expression to be closed for output. The filename must be in the range 0 - 9. When a print file is closed the associated filename 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 filename

```
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 filename and thereby position the output device to the beginning of the next logical page. The filename 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 filename 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 filename is open to *T. Logical output stream 2 consists of all output generated by LISTP commands and by PUT statements when the corresponding filename 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
300 !
310 !   *** PRINT OUT CHART HEADINGS
320 !
330 OPEN 9 "♦P"
340 PUT 9:PUT 9
350 PUT 9 "DEPRECIATION SCHEDULE FOR $ " ;A;" OVER " ;T;" YEAR(S)"
360 PUT 9:PUT 9
370 PUT 9 " YEAR", "ST. LN. DEP.", "BALANCE", "200% DEP.", "BALANCE"
380 PUT 9
400 !
410 !   *** COMPUTE AND PRINT EACH LINE
420 !
430 B1=A:B2=A:S=A/T:F$="%ZZZZZZV.99"
440 FOR K=1TOT
450 B1=B1-S
460 D=2♦B2/T
470 B2=B2-D
480 PUT 9 Y,FMT(S,F$),FMT(B1,F$),FMT(D,F$),FMT(B2,F$)
490 Y=Y+1
500 NEXT K
510 CLOSE 9
999 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 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	\$ 51321.89
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
1992	\$ 4000.00	\$ 48000.00	\$ 2941.33	\$ 33825.30
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
2004	\$ 4000.00	\$.00	\$ 1081.42	\$ 12436.42

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$="$ZZZZZZZV.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 1

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	\$ 51321.89
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 2

DEPRECIATION SCHEDULE FOR \$ 100000 OVER 25 YEAR(S)

YEAR	ST. LN. DEP.	BALANCE	200% DEP.	BALANCE
1992	\$ 4000.00	\$ 48000.00	\$ 2941.33	\$ 33825.30
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 3

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 AS(A)
20 STRING CHAR$(16RFF)
30 INPUT "ENTER NAME OF FILE TO BE PRINTED";AS
40 OPEN 1 AS END 90
50 OPEN 2 "*P"
60 GET 1 AS
70 PUT 2 AS
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 AS 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.)

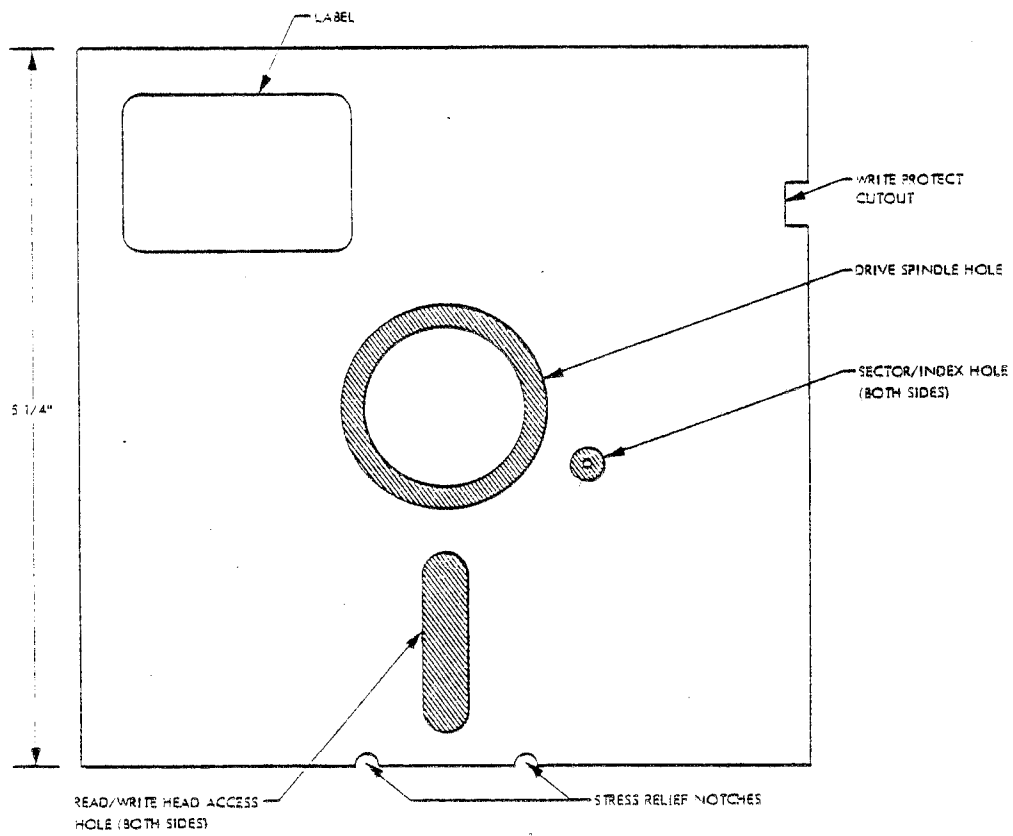


Figure 6.1

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.

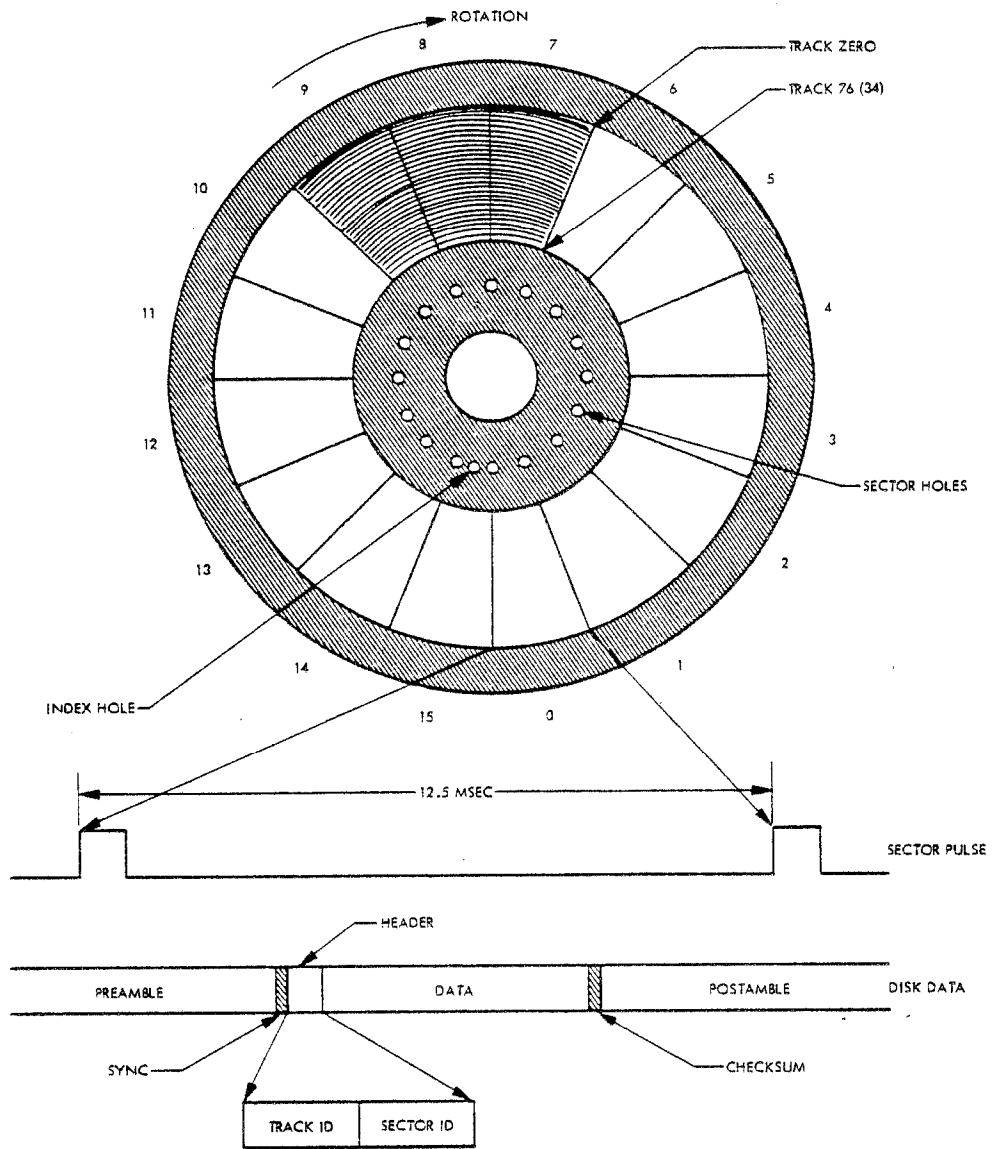


Figure 6.2

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.

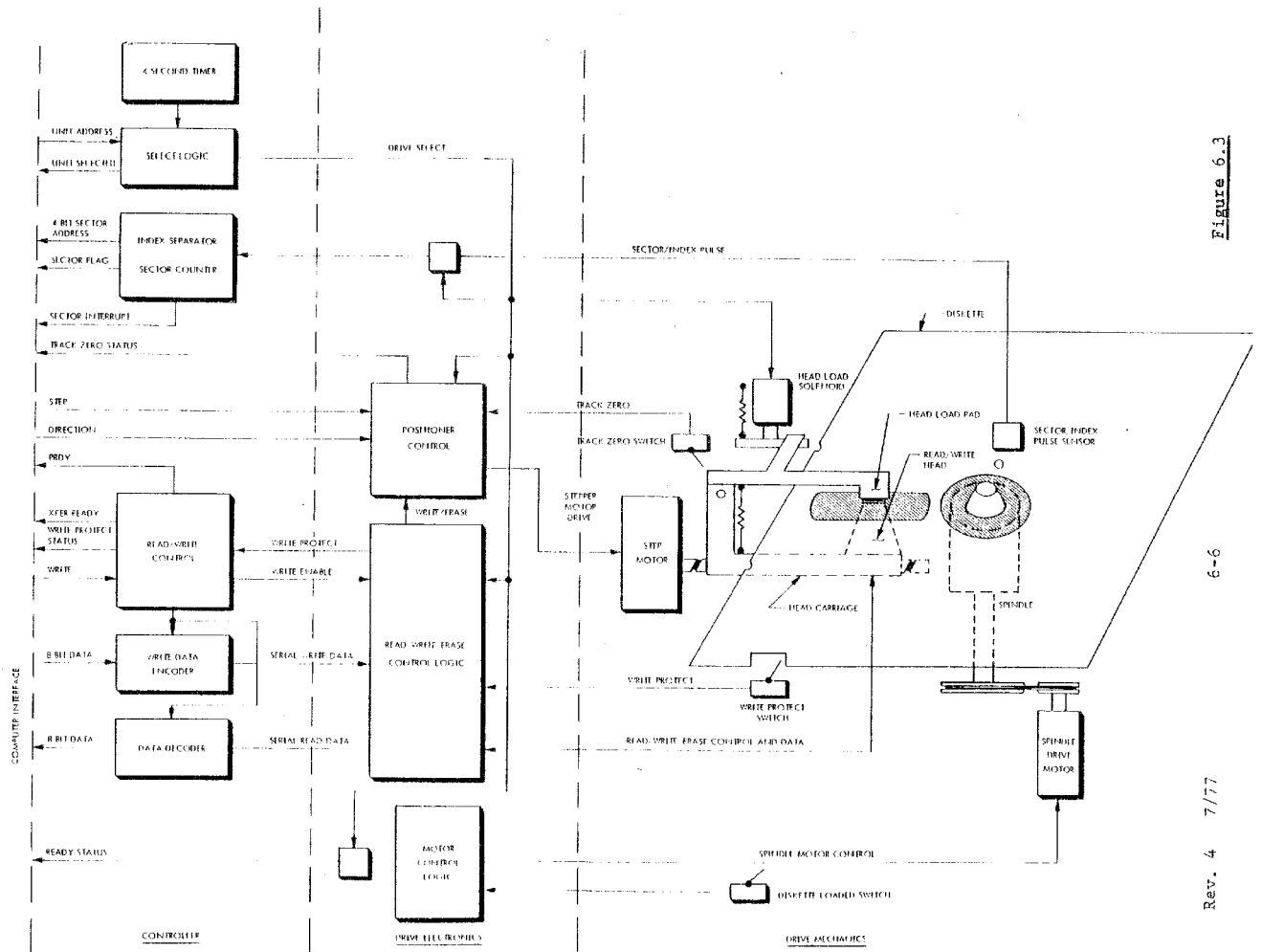


Figure 6.3

6-6

Rev. 4 7/77

- 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 deselected 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

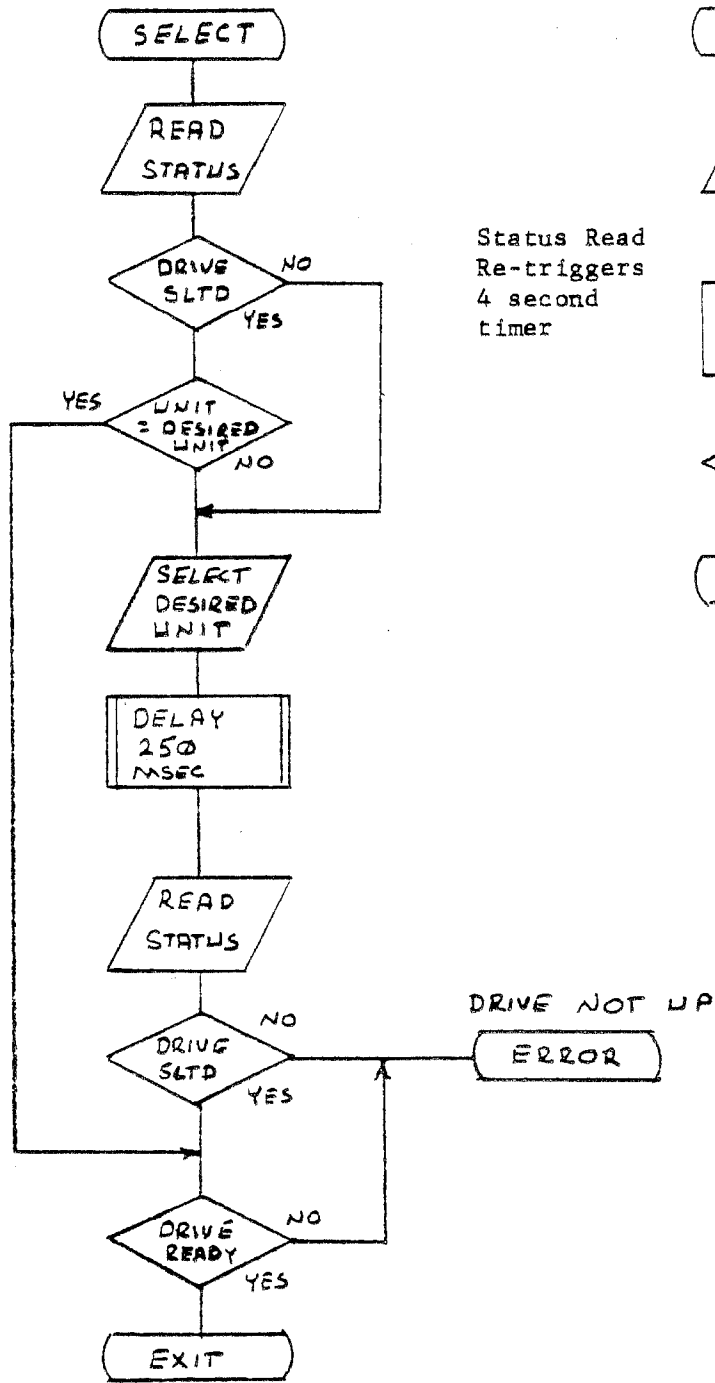
Bits	Definition
0-1	<u>Unit Address</u> : Address of the currently selected drive. Address is valid only if SLTD is true.
2	<u>SLTD</u> : Unit selected. This flag is low true, i.e., 0 = Selected 1 = Not selected SLTD is true if a drive has been selected and the 4-second timer has not expired. SLTD is low true so that the software may detect when the controller is not installed (non-existent memory references yield 0FFH).
3	<u>TK0</u> : Track 0 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

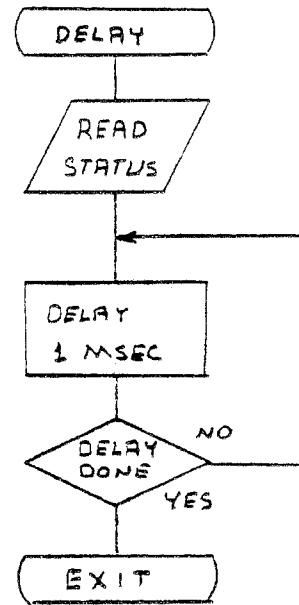
F602H If the read data register is accessed when the transfer flag is set during a read operation, the controller will hold the PRDY line false until a byte of data is available. The controller will then place the data on the S100 BUSS data lines and set PRDY true for 1 bit time (4 usec). The data will only be available for this 1 bit time period.

Figure 6.4

DRIVE SELECT LOGIC



N MILLISECOND TIMER



Status Read
Re-triggers
4 second
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 0

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 0 switch is made. If the drive already indicates track 0 status when first selected, the head is stepped in 8 tracks, then out to ensure a good track 0 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

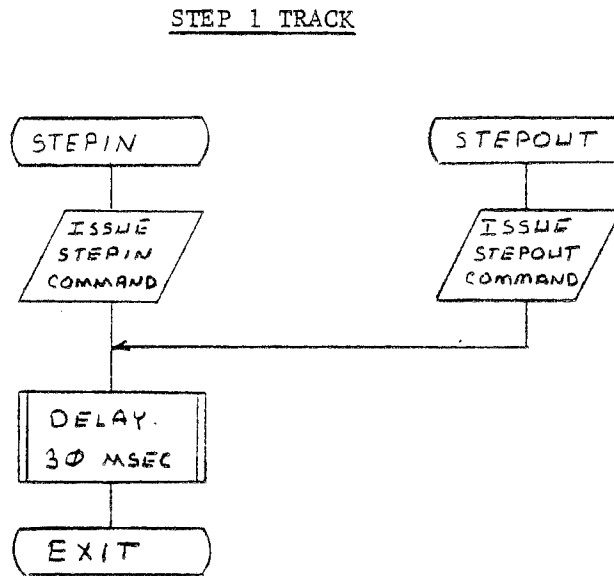


Figure 6.6

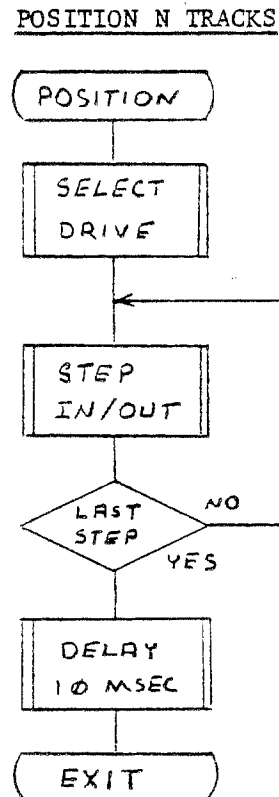
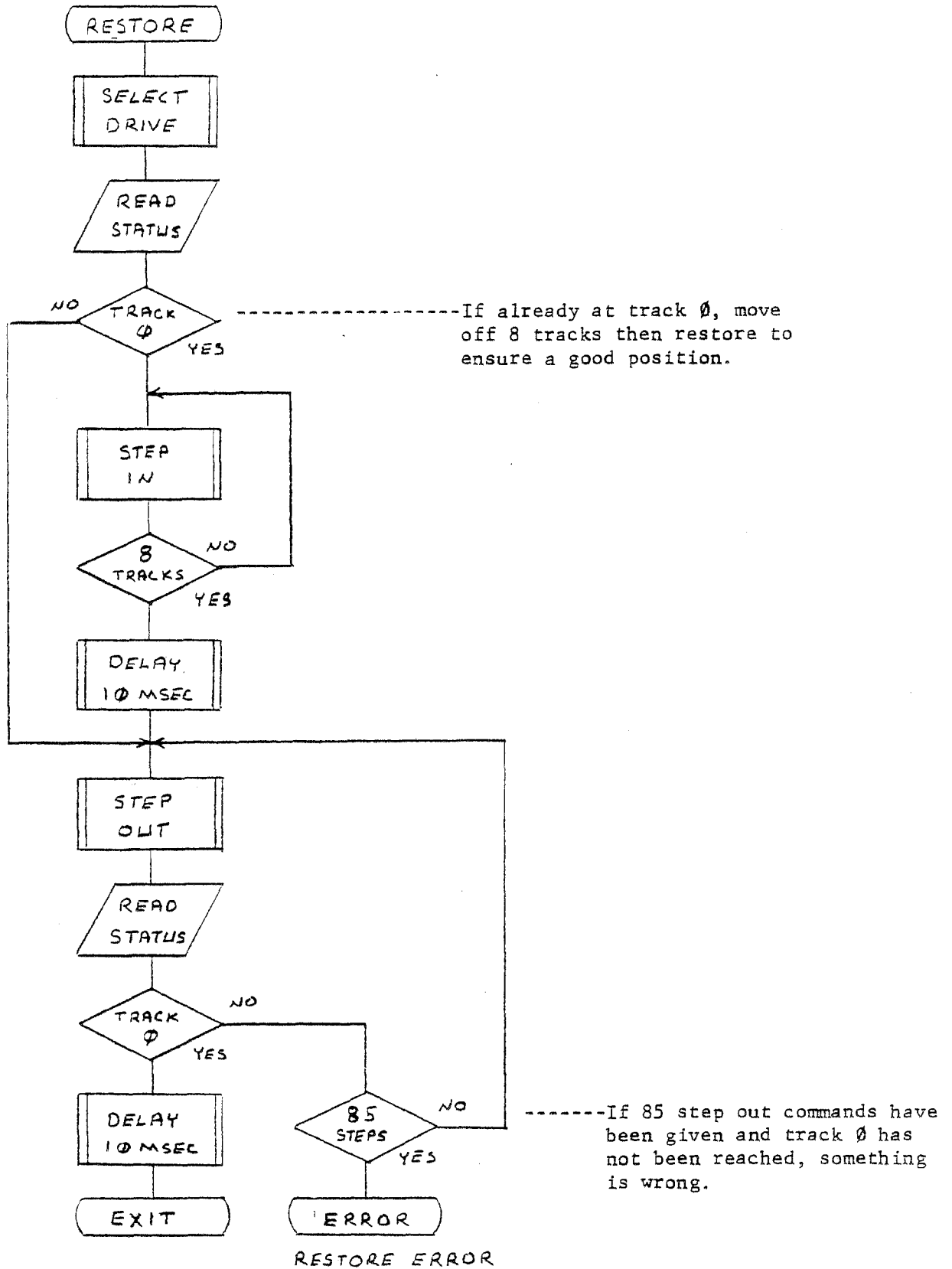


Figure 6.7

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:

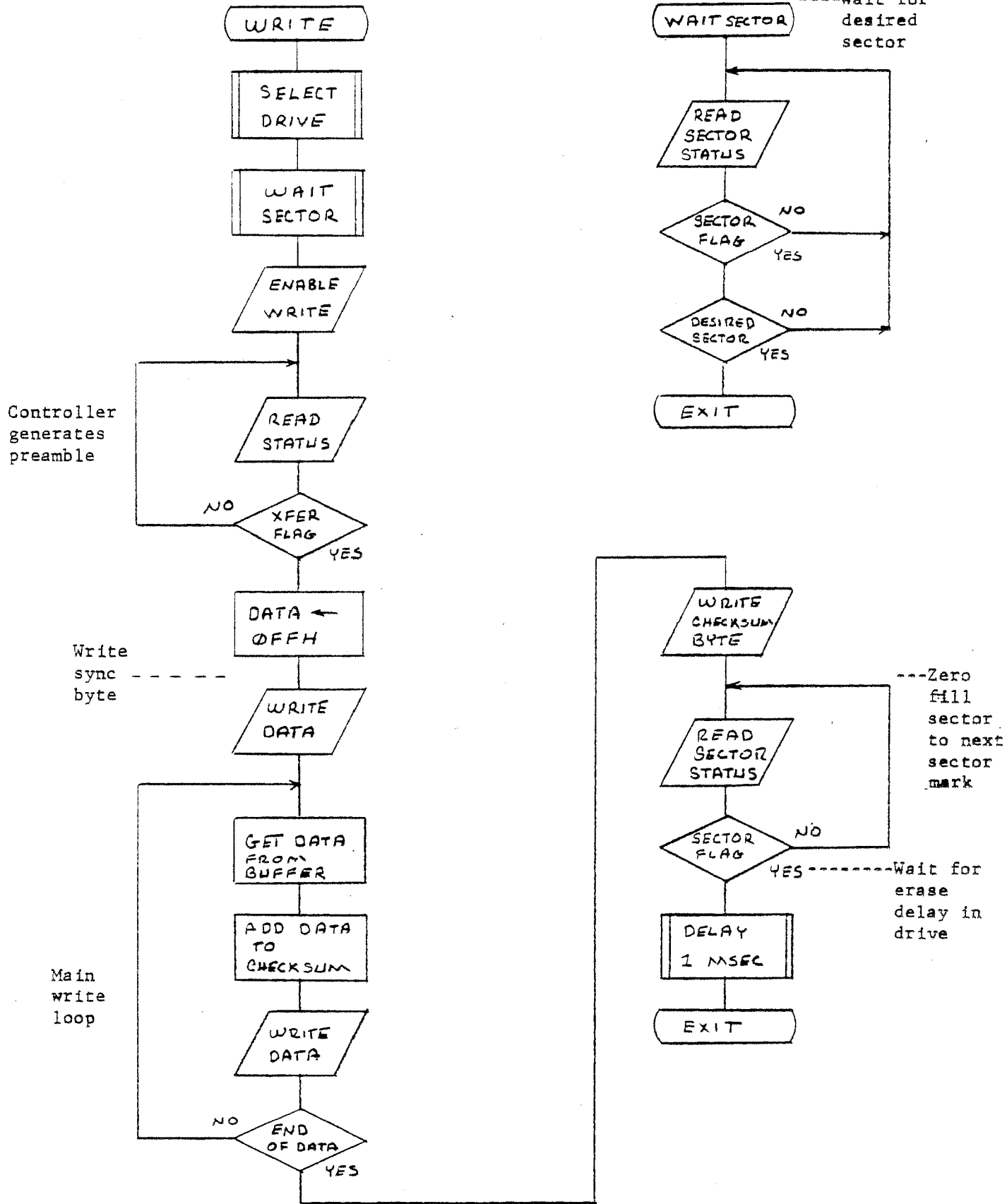
(HL = F601H and A = 0 when this loop is entered)

*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 milli-second 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:

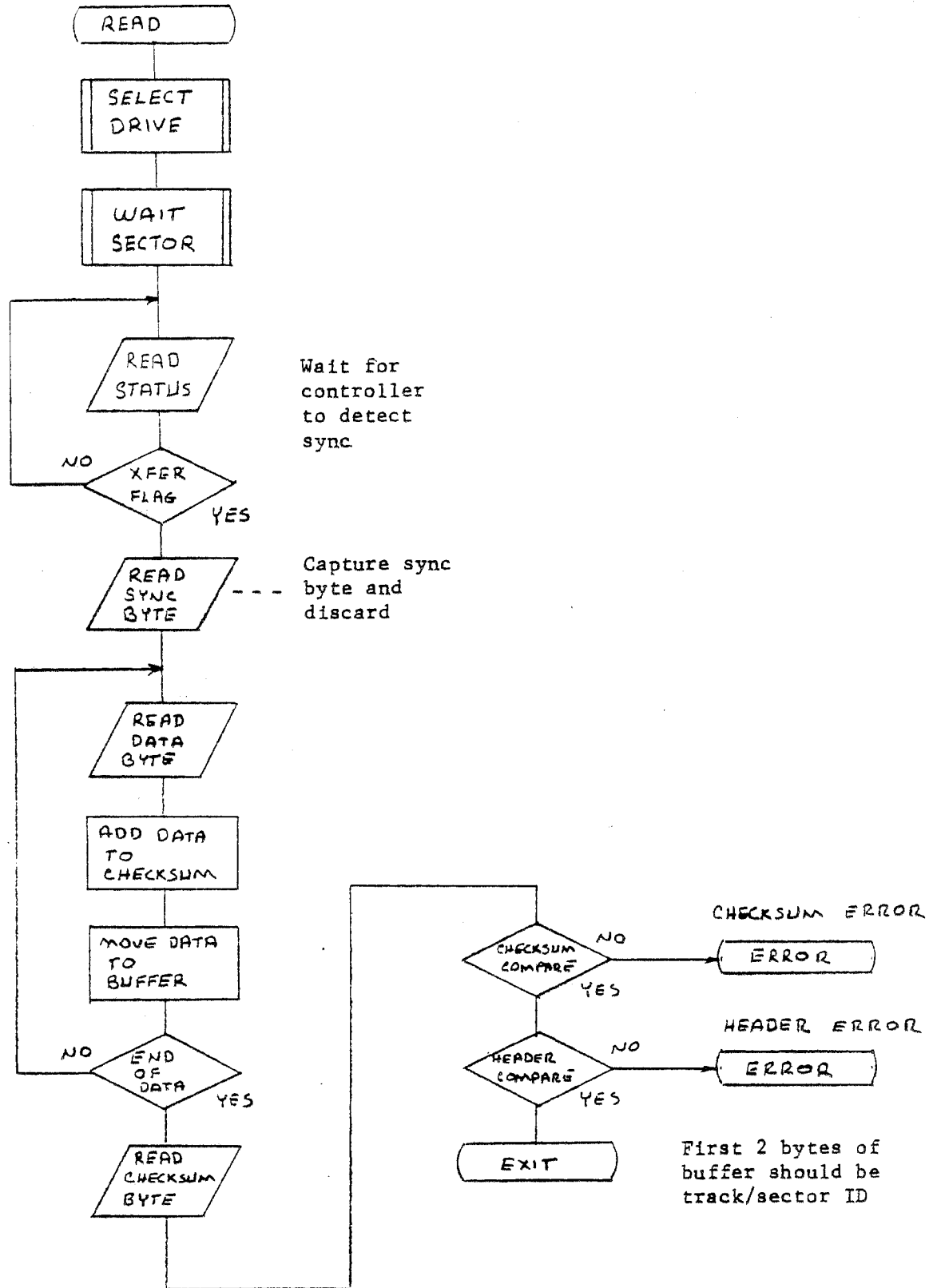
(HL = F601H and A = 0 when this loop is entered)

* 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



- 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 0FFH. 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 F L A G	R A W F L A G	/ / / / / / / /					UNIT ADDR.
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							LSB
DCB + 5	B U F F E R A D D R E S S							MSB

The DCB entries are described as follows:

- FN CODE Function code
 0 = Seek only
 1 = Seek and read sector
 2 = Seek and write sector
 3 = Seek and verify sector
- ID FLAG Pre-Write Header (ID) Check Flag
 0 = Perform check
 1 = Inhibit check
- RAW FLAG Read-After-Write Check Flag
 0 = Perform check
 1 = Inhibit check
- UNIT ADDR. Drive Unit Address
 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.


```

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,DCBLN   INTERNAL DCE
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
*
*
*              RETRY CONTROL FOR READ/WRITE

```



```

047D CDB106          CALL READCK      DO PRE-WRITE HDR
0480 C2A204          JNZ DS090        CHECK - ABORT ERR
0483 CD2F06 DS070    CALL WSECT        GO WRITE
0486 3AF706          LDA DCBSC        DO RAW CHECKSUM
0489 47              MOV B,A          READ CHECK
048A 3AF606          LDA DCEUN        UNLESS INHIBITED
048D E640            ANI RAFI
048F EE40            XRI PAFI
0491 C4B106          CNZ READCK
0494 C3A204          JMP DS090        GO CHECK FOR ERR
0497 3D              DCR A
0498 C2D205 DS080    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
*                   AND BACK AND REPEAT
*
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 DPIPE SLTD
04D8 E5          PUSH H          AND READY
04D9 CDE05      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 SEEKI          YES-CONTINUE
04E2 CD7905     CALL RESTOR        CALIBRATE POSITION
04E5 3AF806 SEEKI 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 SEEKOUT
04F0 CD0705 SEEKIN CALL STEPIN
04F3 3D         DCR A
04F4 C2F004     JNZ SEEKIN
04F7 C30105     JMP SEEKR1
04FA CD1D05 SEEKOUT CALL STPOUT
04FD 3C         INR A
04FE C2FA04     JNZ SEEKOUT
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 STPI 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      LHL DADR
0528 3660        MVI M,STEP   STEP OUT ONE TRK
052A C31305      JMP STPI     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      LHL 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
*
079 E5      RESTOR PUSH H
07A C5      PUSH B
07B CDED05  CALL LDTRK      POINT HL TO TRACK
07E 36FF    MVI M,X'FF'    PRESET TO BAD TRK
080 CD8805  CALL RESTRI    RESTORE TO TK 0
083 3600    MVI M,0          SET TRACK=0
085 C1      POP B
086 E1      POP H
087 C9      RET

*
*          RESTORE TO TK 0
*
588 E5      RESTRI PUSH H
589 CDE405  CALL SLCT      ENSURE UNIT SLCTD
58C D5      PUSH D          AND READY
58D C5      PUSH B
58E 2A0207  LHLD DADR      POINT TO STATUS
591 23      INX H          BYTE
592 7E      MOV A,M        ALREADY AT
593 E608    ANI TK0        TRACK 0 ?
595 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
05EA D1 POP D
05BB E1 POP H
05BC C9 RET

```

```

*
* LOAD ADDRESS OF CURRENT TRACK ON
* CURRENT UNIT INTO HL
*

```

```

05ED 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 PARMER 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 BPROM+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
*
*

```

```

*
0086      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  LHL D 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  LHL D 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  1        DELAY LOOP=1.008
0622 B7      ORA  A        MSEC 0500 NSEC

```

```

0623 C22006      JNZ  TI010+1
*
*      1MSEC 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,SCLN     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 BUFADR     GET BUFFER ADDR
0647 E5         PUSH H
0648 D1         POP  D          MOVE TO DE
0649 3AF806     LDA  DCBTK     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,SCLN    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,SLEN-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 LHL D BUFADR          GET BUFFER ADDRESS
06D9 EB          XCHG
06DA 2A0207      LHL D 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'          OK -IS THIS THE
06F0 A8          XRA  B          ONE WE WANT?
06F1 C2E906      JNZ  GETSEC          NO-WAIT
06F4 C9          RET          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   DCBTK  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 0 or to raise 0 to a negative or 0 power, which are undefined operations.

ZERO DIVIDE - Attempt to divide by zero which is an undefined operation.

APPENDIX B - BASIC UTILITY

B.0 DESCRIPTION

The MDOS System diskette included with each system contains a BASIC utility program.

The functions provided are:

- 1) Initializing a diskette: This must be done before a newly purchased diskette can be used by MDOS or M.BASIC to store data or programs. The INIT command in MDOS may be used instead. It has the same effect.
- 2) Examining and modifying memory: Used to examine, and change if desired, the contents of any location in memory. The P command in the Extended Systems Monitor is similar.
- 3) Saving BASIC: Writes a copy of M.BASIC plus the RES module onto a diskette. Used for creating a BASIC-only system diskette.

B.1 CALLING THE UTILITY

- 1) Make sure you have mounted in drive 0 an MDOS system diskette, or BASIC-only diskette with both M.BASIC and the BASIC Utility on it.
- 2) Use normal operating procedures to get M.BASIC in control, indicated by the READY prompt.
- 3) Enter the command LOAD "UTILITY" (return).
- 4) When the system responds with READY, enter RUN (return). The Utility will sign on with:

```
BASIC UTILITY PROGRAM-VERSION X.X
ENTER KEY TO SELECT DESIRED FUNCTION
```

```
F      FORMAT DISK
M      MEM EXAM/MODIFY
S      SAVE BASIC
E      EXIT
```

```
FUNCTION?
```

- 5) To select a function, enter the associated letter, followed by a return.
- 6) After completing a function, the program will request another command. Enter E to return to M.BASIC.
- 7) If the program is for some reason unable to complete a function, it may return to the M.BASIC executive. To get back to the Utility, begin again at step 3.

B.2 INITIALIZING A DISKETTE - FUNCTION F

The Utility refers to this as the FORMAT DISK function. 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.

Since initialization essentially erases a diskette, DO NOT initialize the System diskettes included with your system.

- 1) In response to F (return), the Utility will output:

SPECIFY UNIT NUMBER?

- 2) Enter the number of the drive to be used (0 to 3) and press return. The program responds with:

INSERT BLANK DISKETTE IN UNIT X.
ARE YOU READY?

If you wish to get out of this function, press (return), otherwise, continue.

- 3) Load the diskette you wish to initialize into the specified drive.
- 4) Enter Y (return).

The Utility will initialize the diskette. This takes about 70 seconds.

When completed, the Utility will request the next function command.

B.3 MEMORY EXAM/MODIFY - FUNCTION M

In response to M (return), the Utility will output:

ENTER ADDRESS?

Type the hexadecimal representation of the desired memory address followed by a carriage return. The Utility will print 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 contents of the next sequential memory location are then displayed and the Utility prompts for the next response.
- 2) If a carriage return only is entered, the 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.4 SAVE BASIC - FUNCTION S

- 1) In response to S (return), the Utility will output:

ARE YOU READY?

If you want to get out of this function, press (return), otherwise, continue.

- 2) Mount the diskette on which you wish to store M. BASIC in drive 0. (The diskette MUST be newly initialized. If it is not, do section B.2 above, first.)
- 3) Enter Y (return).
- 4) The Utility will save M. BASIC on the diskette, and set its attributes to permanent and write-protected. When completed, the Utility will request the next function command.
- 5) If you wish to copy the BASIC Utility onto the BASIC-only diskette, exit from the Utility by entering E as the next function. After the BASIC prompt READY, enter SAVE "N:UTILITY" (return).
- 6) To copy any assembly language utility, such as DISKCOPY, onto the BASIC-only diskette, use the FILECOPY command in MDOS.

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 overlaid 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.

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

See Section 4.5.3.

APPENDIX E - RES.I/O SOURCE LISTING

Addr	B1	B2	B3	B4	E	Line	Label	Opcd	Operand
0000						0010	*		
0000						0020	* I/O source file for Micropolis RES module		
0000						0030	* on Vector MZ, version 4.1		
0000						0040	*		
0000						0050	* by Neale Brassell,		
0000						0060	* 2/79		
0000						0070	* Vector Graphic Inc.		
0000						0080	*		
0000						0090	* first, general system equates from SYSQ		
0000						0100	*		
0000	04EC	=				0110	@CONSOLEADDR	EQU	04ECH
0000	04F0	=				0120	@CIOTABLE	EQU	04FOH
0000	0502	=				0130	@LIOTABLE	EQU	0502H
0000	078D	=				0140	@CDIN	EQU	078DH
0000	0792	=				0150	@CDOUT	EQU	0792H
0000	0797	=				0160	@CDBRK	EQU	0797H
0000	07E5	=				0170	@LDOUT	EQU	07E5H
0000	07EA	=				0180	@LDATN	EQU	07EAH
0000	04EA	=				0190	@D1PORT	EQU	04EAH
0000	04E7	=				0200	RES	EQU	04E7H
0000						0210	*		
0000						0220	* now, general equates		
0000						0230	*		
0000	000D	=				0240	CR	EQU	0DH
0000	000A	=				0250	LF	EQU	0AH
0000	0008	=				0260	BS	EQU	08H
0000	0003	=				0270	CNTC	EQU	03H
0000	0013	=				0280	CNTS	EQU	13H
0000	0015	=				0290	CNTU	EQU	15H
0000	0018	=				0300	CNTX	EQU	18H
0000	007F	=				0310	DEL	EQU	7FH
0000	005F	=				0320	USCORE	EQU	5FH
0000						0330	*		
0000	0010	=				0340	CANCELLED	EQU	16
0000						0350	*		
0000	C0DC	=				0360	MIN	EQU	0C0DCH
0000	C098	=				0370	MOUT	EQU	0C098H
0000						0380	*		
0000						0390	* Get printer etc. from user, and compute ports		
0000						0400	*		
0000	0000	=				0410	DIAB	REQ	'Diablo (1=Yes, 0=No):'
0000	0000	=				0420	CENT	REQ	'Centronics (1=Yes, 0=No):'
0000	0000	=				0430	DECW	REQ	'Decwriter, TTY, etc (1=Yes, 0=No):'
0000	0000	=				0440	OTHR	REQ	'Any other printer (1=Yes, 0=No):'
0000	0000	=				0450	BASE	REQ	'Bitstreamer base address:'
0000	0000	=				0460	ANY	EQU	DIAB!CENT!DECW!OTHR
0000						0470	*		
0000	0003	=				0480	SSTAT	EQU	BASE+3 ;serial status
0000	0002	=				0490	SDATA	EQU	BASE+2 ;serial data
0000	0000	=				0500	PSTAT	EQU	BASE ;parallel 0
0000	0001	=				0510	PDATA	EQU	BASE+1 ;parallel 1
0000						0520	*		
0000						0530	* Ok! first, the vectors to the i/o tables		
0000						0540	*		
0000						0550		ORG	@CONSOLEADDR
04EC						0560	*		

Addr	B1	B2	B3	B4	E	Line	Label	Opcd	Operand	
04EC	FO	04				0570		DW	@CIOTABLE	
04EE	02	05				0580		DW	@LIOTABLE	
04F0						0590	*			
04F0						0600	* the console i/o table			
04F0						0610	*			
04F0						0620		ORG	@CIOTABLE	
04F0						0630	*			
04F0	14	05				0640		DW	CIN	;logical input
04F2	2E	05				0650		DW	COUT	;logical output
04F4	77	05				0660		DW	CBRK	;logical break check
04F6	F8	05				0670		DW	CDIN	;physical input
04F8	00	06				0680		DW	CDOUT	;physical output
04FA	04	06				0690		DW	CDBRK	;physical break check
04FC	0F	06				0700		DW	CDINIT	;physical initialization
04FE						0710	*			
04FE	00					0720	WRAPFLAG	DB	0	;wraparound flag
04FF	01					0730	NULLCT	DB	1	;null count (+1)
0500	4F					0740	WIDTH	DB	79	;width (-1)
0501	03					0750	CURSOR	DB	3	;cursor position
0502						0760	*			
0502						0770	* next, the list i/o table			
0502						0780	*			
0502						0790		ORG	@LIOTABLE	
0502						0800	*			
0502	00	00				0810		DW	0	;placeholder for input
0504	8E	05				0820		DW	LOUT	;logical output
0506	EA	05				0830		DW	LATN	;logical attention check
0508	00	00				0840		DW	0	;placeholder for input
050A	27	06				0850		DW	LDOUT	;physical output
050C	11	06				0860		DW	LDATN	;physical attention check
050E	13	06				0870		DW	LDINIT	;physical initialization
0510						0880	*			
0510	00					0890	PWRAPFLAG	DB	0	;wraparound flag
0511	01					0900	PNULLCT	DB	1	;null count (+1)
0512	83					0910	PWIDTH	DB	131	;width (-1)
0513	01					0920	PCURSOR	DB	1	;line position
0514						0930	*			
0514						0940	* now for the logical i/o routines			
0514						0950	*			
0514						0960	* {CIN} logical console input			
0514						0970	*			
0514	CD	8D	07			0980	CIN	CALL	@CDIN	;get character
0517	78					0990		MOV	A,B	
0518	FE	03				1000		CPI	CNTC	;^C?
051A	C8					1010		RZ		;return if so
051B	FE	15				1020		CPI	CNTU	;^U?
051D	C2	22	05			1030		JNZ	\$/+5	
0520	06	18				1040		MVI	B,CNTX	;xlate ^U into ^X
0522	FE	5F				1050		CPI	USCORE	;underscore?
0524	CA	2A	05			1060		JZ	BSPCE	
0527	EE	7F				1070		XRI	DEL	;DEL?
0529	C0					1080		RNZ		
052A	06	08				1090	BSPCE	MVI	B,BS	;make backspace
052C	3C					1100		INR	A	;force NZ
052D	C9					1110		RET		
052E						1120	*			

Addr	B1	B2	B3	B4	E	Line	Label	Opcd	Operand	
052E						1130	* {COUT} logical console output			
052E						1140	*			
052E	ED	5B	FE	04		1150	COUT	LDED	WRAPFLAG	
0532	2A	00	05			1160		LHLD	WIDTH	;get wrap, null, width, cursor
0535	78					1170		MOV	A,B	;get character
0536	FE	0A				1180		CPI	LF	;linefeed?
0538	CA	92	07			1190		JZ	@CDOUT	;output, ignor if so
053B	FE	0D				1200		CPI	CR	;return?
053D	CA	5F	05			1210		JZ	CROUT	;handle special
0540	FE	18				1220		CPI	CNTX	;^X character?
0542	CA	6F	05			1230		JZ	CNTXOUT	;handle special also
0545	FE	08				1240		CPI	BS	
0547	C2	4D	05			1250		JNZ	COU1	;print if not BS
054A	25					1260		DCR	H	
054B	25					1270		DCR	H	;adjust cursor counter
054C	00					1280		NOP		;(space for patch)
054D	CD	92	07			1290	COU1	CALL	@CDOUT	;print character
0550	7B					1300	WRAP	MOV	A,E	;get wrap flag
0551	B7					1310		ORA	A	
0552	C0					1320		RNZ		;return if no wrap
0553	7C					1330		MOV	A,H	;get cursor
0554	BD					1340		CMP	L	;end of line?
0555	C2	69	05			1350		JNZ	DONE	;done if not
0558	06	0D				1360	CCRLF	MVI	B,CR	
055A	CD	92	07			1370		CALL	@CDOUT	;print LF
055D	06	0A				1380		MVI	B,LF	
055F	CD	92	07			1390	CROUT	CALL	@CDOUT	;and CR
0562	06	00				1400		MVI	B,0	;make a null
0564	15					1410		DCR	D	;decrement counter
0565	C2	5F	05			1420		JNZ	CROUT	;loop
0568	AF					1430		XRA	A	
0569	3C					1440	DONE	INR	A	;increment cursor ptr
056A	32	01	05			1450		STA	CURSOR	;save
056D	B7					1460		ORA	A	
056E	C9					1470		RET		;return
056F	06	5C				1480	CNTXOUT	MVI	B,'\'	;print \ instead of ^X
0571	CD	92	07			1490		CALL	@CDOUT	
0574	C3	58	05			1500		JMP	CCRLF	;go print CRLF
0577						1510	*			
0577						1520	* {CBRK} logical console break check			
0577						1530	*			
0577	CD	97	07			1540	CBRK	CALL	@CDBRK	
057A	C0					1550		RNZ		;return if no char
057B	78					1560		MOV	A,B	;get char
057C	FE	13				1570		CPI	CNTS	;^S?
057E	C2	89	05			1580		JNZ	CANC	
0581	CD	8D	07			1590	PAUSE	CALL	@CDIN	;get char
0584	FE	13				1600		CPI	CNTS	;another ^S?
0586	CA	81	05			1610		JZ	PAUSE	
0589	FE	03				1620	CANC	CPI	CNTC	;^C?
058B	3E	10				1630		MVI	A,CANCELLED	;error code, just in case
058D	C9					1640		RET		;return
058E						1650	*			
058E						1660	* {LOUT} logical list output			
058E						1670	*			
058E	ED	5B	10	05		1680	LOUT	LDED	PWRAPFLAG	;get wrap, nulls

Addr	B1	B2	B3	B4	E	Line	Label	Opcd	Operand	
0592	2A	12	05			1690		LHLD	PWIDTH	;and width, cursor
0595	78					1700		MOV	A,B	
0596	FE	0A				1710		CPI	LF	;linefeed?
0598	C2	A2	05			1720		JNZ	LOUTO	
059B	CD	E5	07			1730		CALL	@LDOUT	;print directly if LF
059E	DA	EE	05			1740		JC	ATT	;handle if ATTN
05A1	C9					1750		RET		
05A2	FE	0D				1760	LOUTO	CPI	CR	;return?
05A4	CA	CC	05			1770		JZ	LCROUT	
05A7	FE	18				1780		CPI	CNTX	;^X character?
05A9	CA	DF	05			1790		JZ	LCNTXOUT	
05AC	FE	08				1800		CPI	BS	;backspace?
05AE	C2	B4	05			1810		JNZ	LOUT1	
05B1	25					1820		DCR	H	;adjust cursor
05B2	25					1830		DCR	H	
05B3	00					1840		NOP		;(spot for patch)
05B4	CD	E5	07			1850	LOUT1	CALL	@LDOUT	;print character
05B7	DA	EE	05			1860		JC	ATT	;handle if ATTN
05BA	7B					1870	LWRAP	MOV	A,E	;wraparound?
05BB	B7					1880		ORA	A	
05BC	C0					1890		RNZ		;return if not
05BD	7C					1900		MOV	A,H	;get cursor
05BE	BD					1910		CMP	L	;too far right?
05BF	C2	D9	05			1920		JNZ	LDONE	
05C2	06	0D				1930	LCRLF	MVI	B,CR	
05C4	CD	E5	07			1940		CALL	@LDOUT	;print LF
05C7	DA	EE	05			1950		JC	ATT	;check ATTN
05CA	06	0A				1960		MVI	B,LF	
05CC	CD	E5	07			1970	LCROUT	CALL	@LDOUT	;print CR
05CF	DA	EE	05			1980		JC	ATT	
05D2	06	00				1990		MVI	B,0	;create a null
05D4	15					2000		DCR	D	;count
05D5	C2	CC	05			2010		JNZ	LCROUT	;print nulls
05D8	AF					2020		XRA	A	
05D9	3C					2030	LDONE	INR	A	;inc cursor
05DA	32	13	05			2040		STA	PCURSOR	;save it
05DD	B7					2050		ORA	A	
05DE	C9					2060		RET		;return
05DF	06	5C				2070	LCNTXOUT	MVI	B,'\'	;xlate here, too
05E1	CD	E5	07			2080		CALL	@LDOUT	;print
05E4	D2	C2	05			2090		JNC	LCRLF	;handle CRLF if no ATTN
05E7	C3	EE	05			2100		JMP	ATT	;go to ATTN routine
05EA						2110	*			
05EA						2120	* {LATN}	list logical	attention check	
05EA						2130	*			
05EA	CD	EA	07			2140	LATN	CALL	@LDATN	;do it
05ED	D0					2150		RNC		;done if NC
05EE	21	EA	04			2160	ATT	LXI	H,@D1PORT	;on ATTN,
05F1	3E	01				2170		MVI	A,1	; reset
05F3	77					2180		MOV	M,A	; assignments
05F4	3C					2190		INR	A	; to their
05F5	23					2200		INX	H	; defaults,
05F6	77					2210		MOV	M,A	; and indicate
05F7	C9					2220		RET		; an error.
05F8						2230	*			

Addr	B1	B2	B3	B4	E	Line	Label	Opcd	Operand
05F8						2250	*		
05F8						2260	* now for the physical i/o drivers		
05F8						2270	*		
05F8						2280	* {CDIN} physical console input		
05F8						2290	*		
05F8	CD	DC	CO			2300	CDIN	CALL	MIN ;get stat/char
05FB	CA	F8	05			2310		JZ	CDIN ;none yet
05FE	47					2320		MOV	B,A ;satisfy requirements
05FF	C9					2330		RET	;that's that
0600						2340	*		
0600						2350	* {CDOUT} physical console output		
0600						2360	*		
0600	78					2370	CDOUT	MOV	A,B ;get character
0601	C3	98	CO			2380		JMP	MOUT ;go print it
0604						2390	*		
0604						2400	* {CDBRK} physical console break check		
0604						2410	*		
0604	CD	DC	CO			2420	CDBRK	CALL	MIN ;get stat/char
0607	CA	0D	06			2430		JZ	CB1 ;no char
060A	47					2440		MOV	B,A ;save char
060B	AF					2450		XRA	A ;set Z
060C	C9					2460		RET	;return
060D	3C					2470	CB1	INR	A ;clear Z
060E	C9					2480		RET	;and return
060F						2490	*		
060F						2500	* {CDINIT} physical console initialization		
060F						2510	*		
060F	AF					2520	CDINIT	XRA	A ;clear CY
0610	C9					2530		RET	;console is always init'ed
0611						2540	*		

Addr	B1	B2	B3	B4	E	Line	Label	Opcd	Operand
0611						2560	*		
0611						2570	* Now for the physical list routines.		
0611						2580	*		
0611						2590	* {LDATN} physical list ATTN check		
0611						2600	*		
0611	AF					2610	LDATN	XRA	A ;none of our devices
0612	C9					2620		RET	; have this feature
0613						2630	*		
0613						2640	* {LDINIT} physical list initialization		
0613						2650	*		
0613	AF					2660	LDINIT	XRA	A ;send nulls
0614	D3	03				2670		OUT	SSTAT
0616	D3	03				2680		OUT	SSTAT
0618	D3	03				2690		OUT	SSTAT
061A	3E	40				2700		MVI	A,40H ;send reset
061C	D3	03				2710		OUT	SSTAT
061E	3E	CE				2720		MVI	A,OCEH ;send mode
0620	D3	03				2730		OUT	SSTAT
0622	3E	27				2740		MVI	A,27H ;send command
0624	D3	03				2750		OUT	SSTAT
0626	C9					2760		RET	;return
0627						2770	*		
0627						2780	* {LDOUT} physical list output		
0627						2790	*		
0627	CD	EA	07			2800	LDOUT	CALL	@LDATN ;formality
062A	D8					2810		RC	
062B						2820	* -----		
062B						2830		IFT	DECW ;if TTY, Decwriter, etc.
062B						2840		PRT	'General selected'
062B						2850	*		
062B						2860	LO1	IN	SSTAT ;get status
062B						2870		RAR	
062B						2880		JNC	LO1 ;wait till ready
062B						2890		MOV	A,B
062B						2900		OUT	SDATA ;output data
062B						2910		XRA	A ;clear C
062B						2920		RET	;return
062B						2930	*		
062B						2940		ENDIF	
062B						2950	* -----		
062B						2960		IFT	DIAB ;if Diablo
062B						2970		PRT	'Diablo selected'
062B						2980	*		
062B						2990	LO1	IN	SSTAT ;get status
062B						3000		RAR	
062B						3010		JNC	LO1 ;wait till ready
062B						3020		MOV	A,B
062B						3030		OUT	SDATA ;output character
062B						3040		CPI	LF ;linefeed?
062B						3050		JNZ	XARET ;return if not
062B						3060		MVI	B,CNTC ;send ETX char
062B						3070		CALL	LDOUT
062B						3080	LO2	IN	SSTAT ;get return status
062B						3090		ANI	2
062B						3100		JZ	LO2 ;wait till reply ready
062B						3110		IN	SDATA ;get reply

Addr	B1	B2	B3	B4	E	Line	Label	Opcd	Operand
062B						3120		MVI	B,LF ;restore LF
062B						3130	XARET	XRA	A ;zap carry flag
062B						3140		RET	;return
062B						3150	*		
062B						3160		ENDIF	
062B						3170	* -----		
062B						3180		IFT	CENT ;if Centronics
062B						3190		PRT	'Centronics selected'
062B						3200	*		
062B						3210	LO1	IN	PDATA ;get status
062B						3220		RAR	
062B						3230		JC	LO1 ;wait till not busy
062B						3240		MOV	A,B
062B						3250		ORI	128 ;strobe on
062B						3260		OUT	PDATA
062B						3270		ANI	127 ;strobe off
062B						3280		OUT	PDATA
062B						3290		ORI	128 ;strobe on
062B						3300		OUT	PDATA
062B						3310		XRA	A ;clear C flag
062B						3320		RET	;return
062B						3330	*		
062B						3340		ENDIF	
062B						3350	* -----		
062B						3360		IFF	ANY ;if no printer at all
062B						3370		PRT	'No printer'
062B						3380	*		
062B	C3	92	07			3390	LO1	JMP	@CDOUT ;dummy routine
062E						3400	*		
062E						3410		ENDIF	
062E						3420	* -----		
062E						3430	*		
062E						3440		IFT	OTHR ;special driver
062E						3450		PRT	'Special printer'
062E						3460	*		
062E						3470	LO1	RET	;user must write special driver
062E						3480	*		
062E						3490		ENDIF	
062E						3500	* -----		
062E						3510	*		
062E						3520		PRT	'End = ', \$
062E						3530	*		
062E						3540		END	RES



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 relocater 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
*
F620      DISK EQU BPROM+X'0200'
*
*      DATA REGISTERS
*
F602      WDATA EQU DISK+X'02'
F602      RDATA EQU WDATA
*
*      STATUS REGISTERS
*
F620      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     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   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
*          NC MODIFIER USED
*
*          DISK PARAMETERS
*
000F    SDLY  EQU   15          STEP+SETTLE TIME
*          DIVIDED BY 2.6775
0086    BYTCT EQU   134        BYTCT/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 CTORG EQU X'A0' CONTROLLER BASE
* ADDRESS SAVED HERE

```

```

*
* 0000 ORG CTORG-X'35' CTORG+2-RLCLEN)
*
* RELOCATOR -- MOVES BOOTSTRAP INTO
* RAM AND STARTS BOOTSTRAP

```

```

*
*
* 006E 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 CTORG+2 OF CONTROLLER
* 0075 EB XCHG SAVE RAM ADDR
* 0076 2AA000 LHD CTORG GET ADDRESS WHICH
* 0079 2E00 MVI L,0 WAS PUSHED ON STAC
* 007B E5 PUSH H MSB IS CTRLR 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 CTORG    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 GC START BOOT
*
001D          BTDSP1 EQU RE010-RELOC
001A          BTDSP2 EQU *-RE010
0037          RLLEN 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 CA900      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
00DE EE20    XRI  RDY      WAIT UNTIL OK
00E0 C2D400  JNZ  SL010    TO PROCEED

*
*          WAIT 250 MSEC
*          FOR SECTOR CNTR
*          TO SYNC
00E3 0E5E    MVI  C,94
00E5 CD4901  CALL TIMER
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
010E 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 LERST GO LOAD PGM
0128 CD3701 CALL RZERO
012B C2D400 JNZ SL010 RESEEK IF HDR BAD
012E 2AA400 LHLD LERST 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 PUSE H SAVE RAM ADDRESS
0138 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

```


APPENDIX G - "FEATURES" PROGRAM TO OPTIONALLY SHORTEN BASIC

M.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 (user's program space) is enlarged. Without removing them, the program buffer begins at 5D86 (Hex) whereas when all the features which can be removed are removed, the program buffer begins at 5700. This is the same place it did in version 3.0 of M.BASIC.

A special assembly language program called FEATURES is supplied to selectively remove features from BASIC. The three features which can be removed are MERGE, RENUM, and EDIT. The procedure is as follows:

- 1) Load BASIC from an MDOS system diskette or from a BASIC-only diskette. This must be BASIC version 4.0.
- 2) Type LINK "FEATURES" then depress (return).
- 3) The program will then begin by displaying:

```
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. You have only the 3 choices given. The program will begin executing as soon as you touch one of the number keys. If you want to return to BASIC rather than executing the program, depress control-C (hold CTRL key down while depressing the letter C) instead of one of the numbers.
- 5) When the selected features are removed, the system is returned to BASIC automatically.

NOTE: If you run the FEATURES program using a disk whose BASIC is already shortened and if you select any of the features which had been removed, then the program will set the beginning of the program buffer back to where it was originally, as if the feature had not been removed, but the feature itself will not be added back on. Thus, the program buffer will be shrunk, but you will not have the feature. In short, be careful that you do not try to remove a feature that has already been removed.

The shortened BASIC created by the FEATURES program may be saved on a newly initialized diskette for use as a BASIC-only diskette. Follow

the procedure in Appendix B before you exit from BASIC, in order to do this.

The shortened BASIC can also be saved on your Personalized MDOS System Diskette, or a copy of it. To do this, type the following lines after BASIC's "READY" prompt, with the desired system disk in drive 0 (each line is followed by depressing return):

```
OPEN 1 "BASIC":ATTRS(1)=0
SCRATCH "BASIC"
SAVE "BASIC" 16R1572, 16R5DFF
ATTRS(1)=16RF:CLOSE 1
```

Following the last line, your system diskette has a copy of the shortened version of BASIC, which will be used everytime you enter the command BASIC. You can use the DISKCOPY command in MDOS to copy this sytem diskette.

If you do not save your shortened BASIC in one of these ways, then since it only exists in the system's memory, it will be lost when you turn the power off or return to MDOS. Until then, you can use it for programming in BASIC.

APPENDIX H - INTERFACING TO A CENTRONICS PRINTER

Centronics Printer

VECTOR GRAPHIC PRINTER INTERFACE

General

The Vector Graphic Printer Interface provides the means to connect a Centronics line printer such as the 700 series of printers or equivalent to the Vector MZ or other Vector Graphic microcomputers. The interface is designed to utilize the Vector Graphic Bit Streamer I/O board parallel ports via connection to one input port and one output port.

The software driver program monitors the BUSY signal from the printer and when the printer is not BUSY (BUSY=0) the program may transfer a character of data at which time the printer BUSY signal goes true thus holding off data transmission until the printer is once again ready to accept data.

I/O PORT BIT ASSIGNMENTS

PORT 01 OUTPUT

BIT	7	6	5	4	3	2	1	0
	STROBE	DATA	DATA	DATA	DATA	DATA	DATA	DATA
		6	5	4	3	2	1	0

PORT 01 INPUT

BIT	7	6	5	4	3	2	1	0
	--	--	--	--	--	--	--	BUSY

INTERFACE PARTS LIST

- 1 ea Serial I/O cable (Bit Streamer to Vector MZ backpanel).
- 1 ea Printer cable (Vector MZ backpanel to printer)
- 1 ea 6 pin Molex connector.

INSTALLATION INSTRUCTIONS

CAUTION - Power must be off before proceeding with installation.

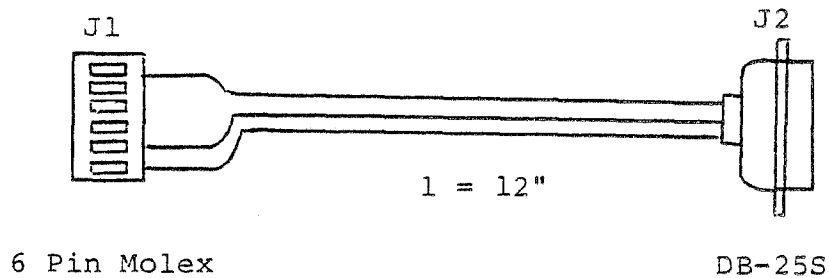
1. Bit Streamer I/O Board
 - a) Cut the circuit trace at J3-19.
 - b) Add a short jumper wire from J3-19 to J2-17.
 - c) Add the 6 pin Molex connector on the circuit or back side of the board as shown in Figure 1.
 - d) Install the Bit Streamer in a chassis slot near the back panel of the computer chassis.
 - e) Plug in the 24 pin dip plug (part of the I/O cable) into J3 of the Bit Streamer. If your computer does not have this cable (Vector P/N IO-1327) one must be obtained.

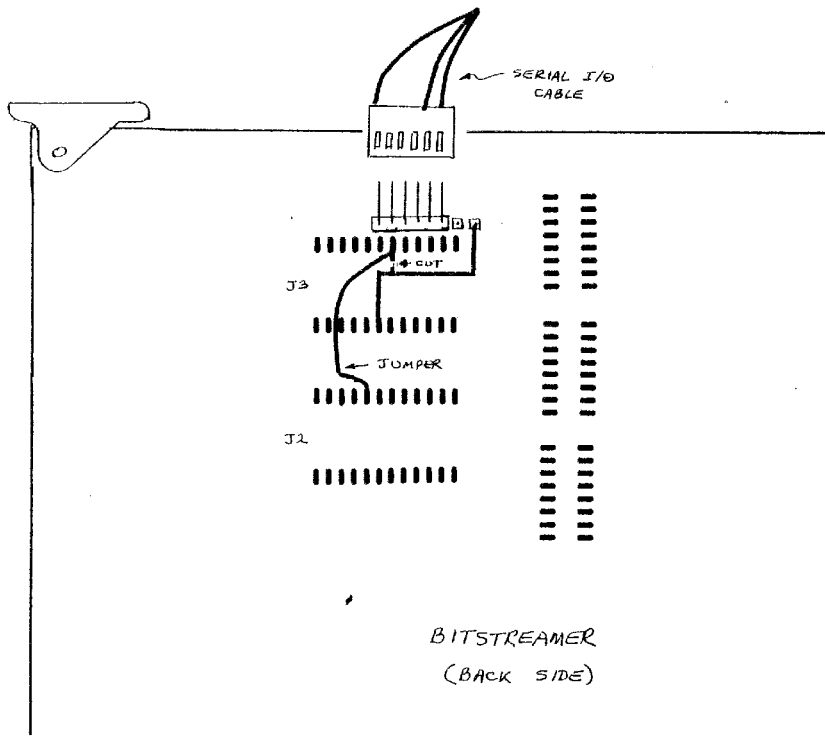
2. Serial I/O Cable
 - a) Install the serial I/O cable in the Vector MZ with the 25 pin connector attached in an available cutout on the rear panel and connect the 6 pin Molex plug to the Bit Streamer as shown in Figure 1. This now becomes the connector to use with your terminal (Hazeltine, etc).

CABLE WIRE LIST AND DIAGRAM

SERIAL I/O CABLE

<u>J1</u> <u>BIT STREAMER</u>	<u>J2</u> <u>VECTOR MZ BACK PANEL</u>	<u>SIGNAL NAME</u>
1	7	GROUND
5	3	TRANSMIT DATA
6	2	RECEIVE DATA





BITSTREAMER
(BACK SIDE)

FIGURE 1

APPENDIX I - TROUBLE SHOOTING IF MDOS DOES NOT LOAD

This section is applicable the first time you attempt to load MDOS using the Extended System Monitor B command.

1) If the disk drive select light does not go on in response to command B, check the connection between the mainframe and the console. If this is not the problem, then the system requires attention by the dealer or by Vector Graphic.

2) If the MDOS has not signed on within 20 seconds, but the disk drive select indicator light is still glowing, the bootstrap ROM has been unable to read the loader into memory from the diskette. Depress RESET. Check whether the correct diskette was inserted in the correct drive, that it is inserted facing correctly (label leftward or upward), and that it is fully inserted and fully mounted (snapped into place). If not the problem, inspect the diskette for obvious contamination or damage. Reload the diskette and begin again with the Monitor B command.

3) If the system has not signed on but the unit select indicator has extinguished, the loader may not have been able to read the rest of the system into memory. The probable cause is a malfunctioning memory chip. Use the Extended Systems Monitor command N to test memory. (See the Extended Systems Monitor manual.) If this test terminates at a value below C000H (i.e. 48K), it indicates malfunctioning memory: The ending address is the location of the malfunction. If N reveals no problem, then try command T, a more thorough memory test. Use the ending address given by command N, less 1, as the second address in the command T argument, and use 0000 as the first address. To use an Extended System Monitor command, depress RESET on the front panel of the mainframe. Then enter the command you want, after the Monitor prompt * appears.

4) If there is no problem with memory, the system requires attention by a service representative.



APPENDIX J - GAMES AND DISPLAYS ON THE MDOS SYSTEM DISKETTE

STARTREKG, CIVILWAR, and LUNAR are games written in BASIC. Get into BASIC, then enter:

PLOADG "<game name>" (return).

The games are self-explanatory, with the exception of STARTREKG. STARTREKG uses the classic set of rules familiar to all computer Startrek aficionados. For others, a little trial and error gets the player going.

FINANCE is a BASIC program for computing various interest and annuity problems. It is useful on a day-to-day basis for users working with investment problems. Its operation is self-explanatory. To start it, use the PLOADG command as with games, above.

FLASH7 is a demonstration of the graphics capability of the Flashwriter II board. Do not attempt to use it unless your system uses this board, indicated by 80 X 24 display on a video monitor or Mindless Terminal. Also, it will not work if the system is set up to run word-processing (i.e. it is a MEMORITE II system, or the Word Management System character generator PROM's have been installed on the Flashwriter II board.) It will only work if the system has the graphics character generator PROM's which are installed when MZ systems are manufactured.

To use FLASH7, mount an MDOS System diskette in drive 0. Get into the MDOS command mode (usually done by depressing B (return) after turning on or resetting the machine.) Then type FLASH7 (return). The program will begin executing, showing off the many features of the Flashwriter II board, including graphics, lack of glitches on screen, multiple cursors, reverse video, and so on. The program will execute indefinitely by repeating itself until halted by the operator with the RESET button. This program is an excellent demo for dealers. (Dealers who want to demo Word Management System on the same system must forego it, however.)

The operator may interact with FLASH7 (unlike the earlier FLASH6) in various ways. First, touching the space bar at most times will freeze the screen, for closer examination. Another space bar will resume the demo. Second, the operator can cause the demo to jump directly to any of several points within its cycle, if that particular part of the demo is of special interest. This is accomplished by pressing one of the following letters at almost any time while the demo is operating:

<u>letter</u>	<u>part of demo</u>	<u>letter</u>	<u>part of demo</u>
C	Character Set	B	Introducing System B
R	Sphere	L	Higher Level Languages
G	Gettysburg Address	S	Bubble Sort
D	Darth Vader		

APPENDIX K - CHANGING MICROPOLIS BOOTSTRAP ROM AND DISK I/O ADDRESSES

The disk Bootstrap ROM and Disk Controller I/O addresses are located in the 1K block from the base address D800 to DBFF.

The user may change this location by changing jumpers on the disk controller board. If this is done, however, the B command in the Extended Systems Monitor will no longer function, unless the new base address is F400. If not, in place of B, the operator must use the G command followed by the new base address.

No software changes are necessary. Disk I/O routines in the RES module automatically find the disk controller and Bootstrap addresses.

This is also true if the CP/M operating system is used. However, the MEMORITE and Word Management System word processing software, and the MZOS operating system, can only function with the disk controller and Bootstrap block beginning at the normal D800 location.

Use the following procedure to change the location of the block:

1. Refer to figure K.1, locate the base address desired and determine the jumpers required.
2. Referring to figure K.2, locate the address jumper locations on the controller board. Vector Graphic ships the board with jumpers W1 and W4 installed.
3. Remove one or both of the installed jumpers and replace with jumpers required for the desired address. Use short lengths of wire, a 25-30 watt soldering iron, and resin-core solder. To avoid blowing LSI chips with static electricity, do not work in a carpeted room. Touch the contacts on the board edge with one hand before beginning to solder.

BASE ADDRESS		ADDRESS BIT								JUMPER INSTALLED						
		JUMPER														
		A15	A14	A13	A12	A11	A10	A9	A8	W1	W2	W3	W4			
		N/A		W1	W2	W3	W4	N/A								
STANDARD ADDRESS	C0	00 - C3FF	1	1	0	0	0	0	0	0	0	0	Y	Y	Y	Y
	C4	00 - C7FF	1	1	0	0	0	1	0	0	0	0	Y	Y	Y	N
	C8	00 - CBFF	1	1	0	0	1	0	0	0	0	0	Y	Y	N	Y
	CC	00 - CFFF	1	1	0	0	1	1	0	0	0	0	Y	Y	N	N
	D0	00 - D3FF	1	1	0	1	0	0	0	0	0	0	Y	N	Y	Y
	D4	00 - D7FF	1	1	0	1	0	1	0	0	0	0	Y	N	Y	N
	D8	00 - DBFF	1	1	0	1	1	0	0	0	0	0	Y	N	N	Y
	DC	00 - DFFF	1	1	0	1	1	1	0	0	0	0	Y	N	N	N
	E0	00 - E3FF	1	1	1	0	0	0	0	0	0	0	N	Y	Y	Y
	E4	00 - E7FF	1	1	1	0	0	1	0	0	0	0	N	Y	Y	N
	E8	00 - EBFF	1	1	1	0	1	0	0	0	0	0	N	Y	N	Y
	EC	00 - EFFF	1	1	1	0	1	1	0	0	0	0	N	Y	N	N
	F0	00 - F3FF	1	1	1	1	0	0	0	0	0	0	N	N	Y	Y
	F4	00 - F7FF	1	1	1	1	0	1	0	0	0	0	N	N	Y	N
	F8	00 - FBFF	1	1	1	1	1	0	0	0	0	0	N	N	N	Y
	FC	00 - FFFF	1	1	1	1	1	1	0	0	0	0	N	N	N	N

As an example, if you wish to use base address F400 install jumper at W3.

Figure K.1 Controller Base Address Jumper Configurations

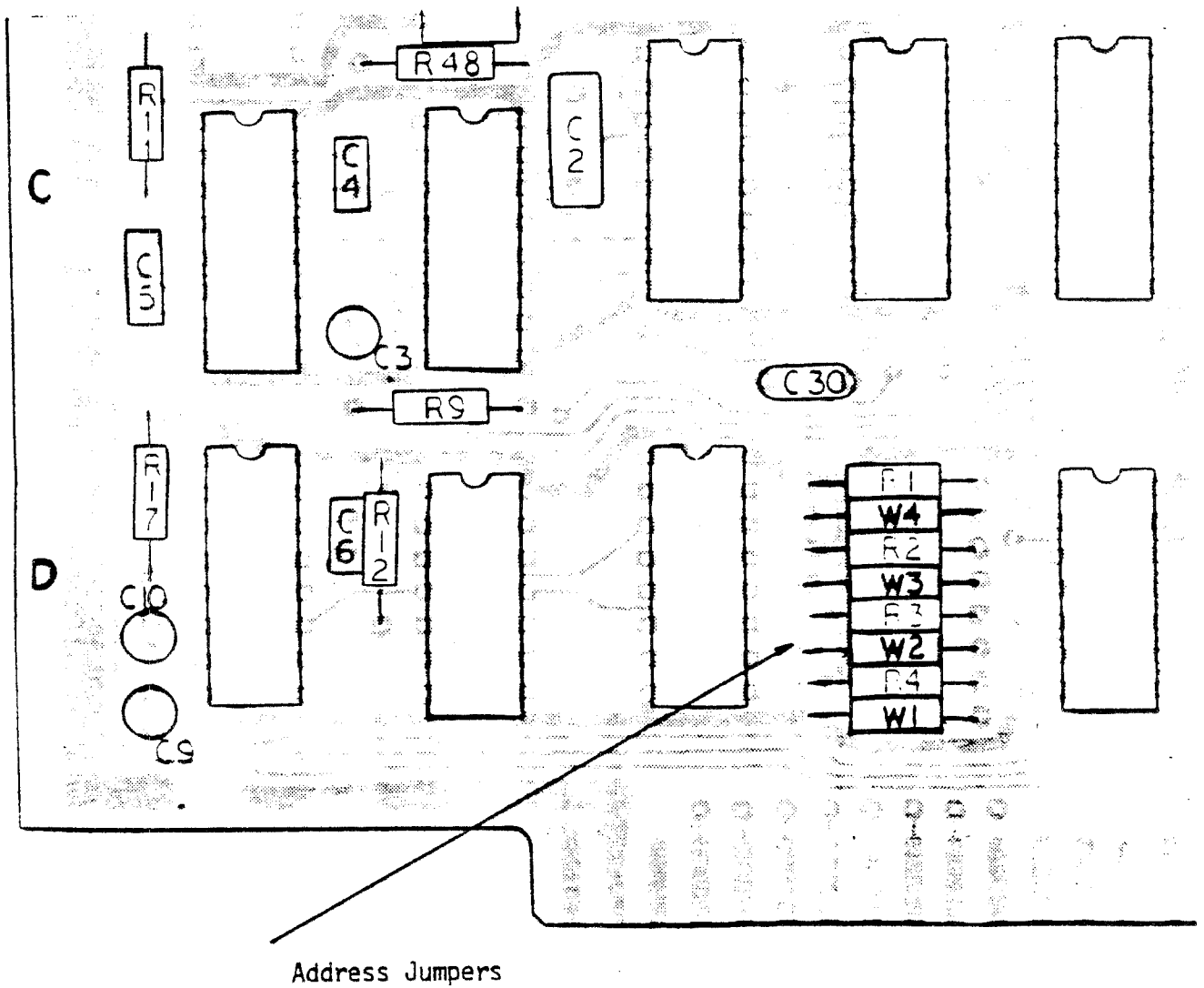


Figure K.2 Locating The Controller Address Jumpers

APPENDIX L - CHANGING CLOCK RATE TO 2 MHz

To operate the system at 2 MHz speeds, a jumper must be removed from the disk controller board, as follows:

- 1) Refer to figure L.1. Locate the ribbon cable edge connector and the resistors R25, R6 and R7.
- 2) Between R25 and R6 is a jumper location, W9. Remove the jumper there with a 25-30 watt soldering iron. To avoid blowing LSI chips with static electricity, do not work in a carpeted room. Touch the contacts on the board edge with one hand before beginning to solder.

A jumper must then be added to the Z-80 board at location "A". Location "A" will be found under the top row of chips, directly under the third chip from the left, U3. If necessary, refer to the figure found in the Z-80 Board User's Manual.

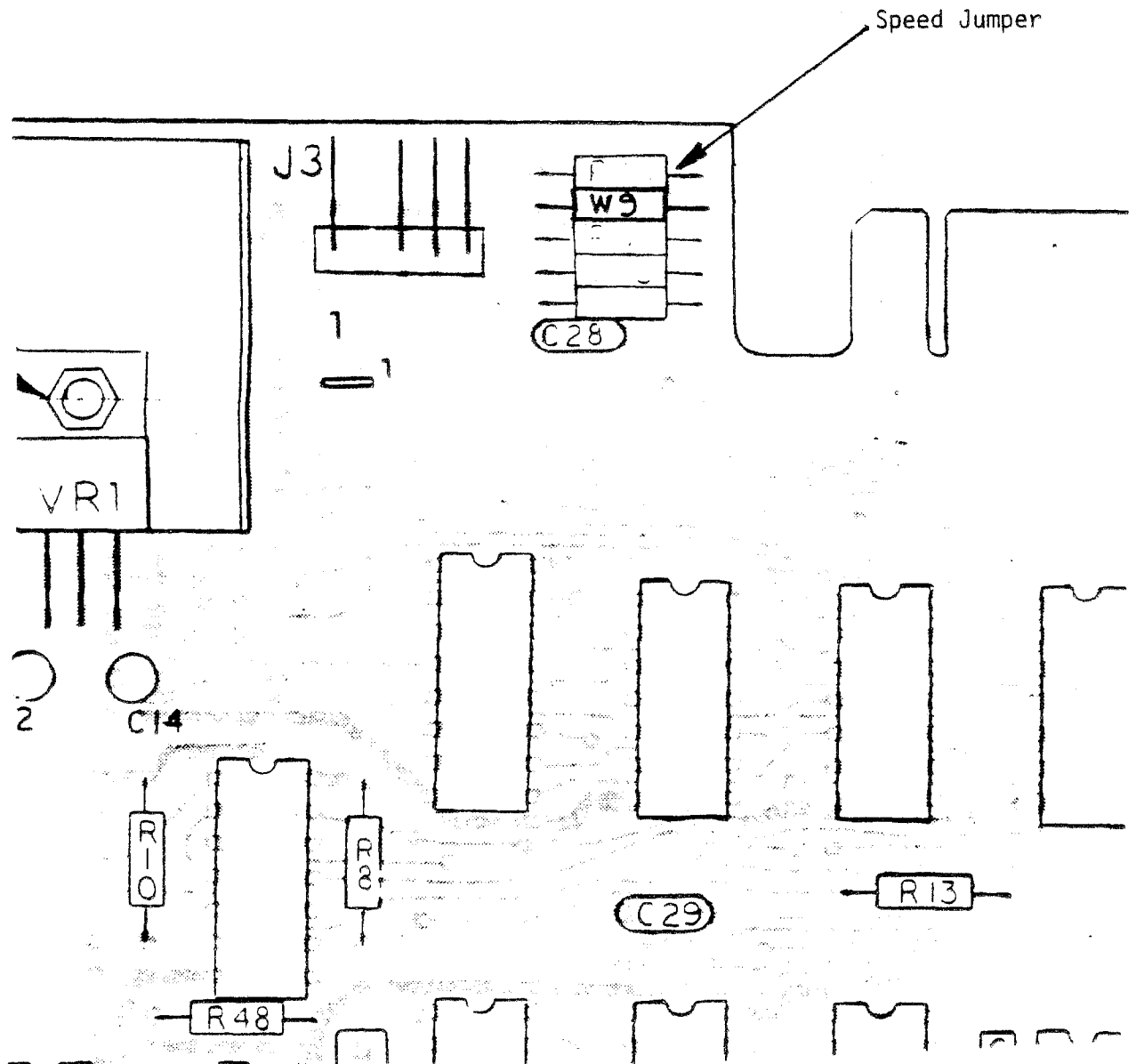


Figure L.1 Locating the controller processor speed jumper

APPENDIX M - WRITING A CONSOLE PHYSICAL I/O ROUTINE

For users wishing to replace the console physical driver in the Vector Graphic Extended Systems Monitor, this section describes the console I/O requirements of the RES module.

Your version should be written in place of the routines in RES.I/O found from lines 2250 to 2540. RES.I/O is the source code for the I/O routines in the RES module. It will be found on your MDOS System Diskettes. The listing is in Appendix E.

If there are any other routines to rewrite, such as printer routines, do this before assembling RES.I/O. To assemble RES.I/O refer to Appendix O where the procedures are explained.

- 1) Lines 720 - 750 in the @CIOABLE can be changed if required.
- 2) The logical input, output and break check routines (CIN, COUT, and CBRK) should not have to be changed. They are tailored to support all MDOS and BASIC requirements.
- 3) The console physical input routine (CDIN) must have the following characteristics:
 - a) It must return all registers except A & B unchanged.
 - b) It can use the A register (destroy it).
 - c) It must return an ASCII character including the parity bit if any, in the B register.
 - d) It must return the carry flag clear (NC). The other status flags can be in any state.
- 4) The console physical output routine (CDOUT) must have the following characteristics:
 - a) It must take an ASCII character in the B register.
 - b) It must return all registers except A unchanged.
 - c) It can use the A register (destroy it).
 - d) It must return the carry flag clear (NC).
 - e) The other status flags can be in any state.
- 5) The console physical break check routine (CDBRK) must have the following characteristics:
 - a) It must check the console input status port to determine if a key has been pressed.
 - b) If no key has been pressed it must return all registers except A unchanged and the zero flag clear (NZ).
 - c) If a key has been pressed it must return the byte, in the B register. The A register can be used (destroyed). All other registers must be unchanged. The zero flag must be set (Z).
 - d) The status flags other than zero can be in any state.
- 6) It will probably not be necessary to change the Physical

Console Device Initialize routine (CDINIT).

APPENDIX N - WRITING A PRINTER PHYSICAL I/O ROUTINE

This Appendix is used when you want to write a custom version of the printer physical I/O routines in the RES module.

- 1) RES.I/O is an assembly language source code file found on the MDOS System Diskettes. In order to rewrite any part of the I/O routines in the RES module, rewrite the relevant portions of this source file, using the Line Editor in MDOS. Note that RES.I/O is not the source file for the entire RES module, but only the I/O portion of it.
- 2) Write your printer driver routine beginning at line 3470 in the RES.I/O. Make sure the contents of lines 3520-3540 are at the end of your routine. The first line must use the label L01, not LDOUT. Do not over-write the other printer physical I/O routines in the Source code (only yours will be assembled, as you will see.) The parts of the new routine must have the following characteristics:
 - a) The character to be output is passed to the physical output routine in the B register in ASCII.
 - b) The physical output routine can use (destroy) the A register.
 - c) All registers except A must be returned unchanged.
 - d) 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. LDATN, the routine which handles printer attention, must not destroy any registers except A.
- 3) Lines 890-920 in the @LIOTABLE can be changed if desired.
- 4) The present contents of lines 3520 - 3540 must be at the end of your routine, or it will not assemble.
- 5) Turn to Appendix 0 when you are ready to assemble and save the new RES module.

APPENDIX O - REASSEMBLING AND SAVING THE RES MODULE

Follow the procedure in this appendix after you have modified the RES.I/O source code using the Line Editor in MDOS. When you are done with this appendix, the RES module program will be modified on your MDOS System Diskette, and ready to use.

- 1) Mount Personalized MDOS System Diskette in Drive 0.
- 2) In MDOS, enter ZSM "RES.I/O" "CRES" "E" (return).
- 3) Four questions will appear on the screen one after the other. Your answers to these questions tell the assembler which printer driver to include in the assembled code. Your choices will be a standard Diablo protocol driver, a standard Centronics protocol driver, a standard Decwriter and Teletype protocol driver, or a driver you have written yourself according to the instructions in Appendix N. As each question appears, depress a 0 if you do not want that driver, or a 1 if you do. Answer 1 to only one of the questions, and 0 to the other three. If you did not write your own and are not sure which of the three standard drivers you need, review section 2.2 in Chapter 2 which includes examples of the different types of printers.
- 4) After the fourth question, another question will appear on the screen, asking "Bitstreamer base address?" The answer to this question is 0 if the Bitstreamer board is jumpered to respond to port #'s 0, 1, 2, and 3, and it is 4 if the Bitstreamer board is jumpered to respond to port #'s 4, 5, 6, and 7. Generally, the former is the case if you are using a serial terminal such as a Hazeltine, and the latter is the case if you are using a Flashwriter board to interface a memory-mapped terminal such as Vector Graphic's Mindless Terminal or a memory-mapped video monitor such as a Hitachi. (If for some reason you have chosen to jumper the Bitstreamer board for a different set of port #'s, then respond with the lowest # of the set you are using.)
- 5) After the last question, the object code will be assembled. At the end of the assembly, the message "END = XXXX" will appear. This value must be under 0700. If it is not, then the routines you have written are too long, and must be shortened.
- 6) At this point, with the code successfully assembled, the new I/O portion of the RES module is on disk as a temporary file called "CRES" but not yet overlayed over the entire RES module on the system diskette. To do this, enter TYPE "CRES" C (return), and then CRES (return). Finally, type SCRATCH "CRES" (return) to clear this workfile from the disk.
- 7) The complete RES module with all changes is now in memory, but not yet stored on disk. Debug it now. Proceed when it

is finalized.

- 8) To save the new RES module, follow the instructions in Section 2.2.0 of this manual.
- 10) NOTE: Do not overlay one of the pre-written printer drivers as instructed in Chapter 2, sections 2.2.1 and 2.2.2. Steps 3 and 4 above already installed the correct driver.

APPENDIX P - MAP OF I/O PORTS

On the following page you will find a chart which lists all the I/O ports that an 8080 or Z80 can address, in hexadecimal notation. An I/O port is accessed when the processor executes an IN xx or OUT xx instruction, where xx is the one of the port numbers in the chart. The port number will appear on the lower half of the address bus instead of a memory address, and either SINP or SOUT will be active high which tells memory NOT to react and tells I/O devices that it is their turn. Each I/O device and board has gating circuitry which detects when its own port number is on the bus. Usually, I/O devices have dip-switches or jumper pads with which you can assign any port number.

Next to some of the ports, you will find the names of commonly used boards which respond to those port numbers. Some of these boards are Vector Graphic's and some are not. In the case of the Vector Graphic boards, most of them can be assigned ANY port number, either by dip-switch or jumper. The numbers shown for these boards are those that Vector Graphic software expects. Use this information to avoid present and future conflict when you are assigning port numbers to hardware. You can also use this sheet as a worksheet if you are assigning a number of ports.

MAP OF PORTS

00	V.G. Flash-	V.G.	40	V.G. 16K bank select	80	C0
01	Writers	Bit-	41		81	C1
02		Streamer	42		82	C2
03	V.G.	I	43		83	C3
04	Bit-	Alternate	44		84	C4
05	Streamer	Bit-	45		85	C5
06	II	Streamer	46		86	C6
07		I	47		87	C7
08			48		88	C8 VDM (not V.G.)
09			49		89	C9
0A			4A		8A	CA
0B			4B		8B	CB
0C			4C		8C	CC
0D			4D		8D	CD
0E			4E		8E	CE
0F			4F		8F	CF
10	A0		50		90	D0
11	A1		51		91	D1
12	A2	D+7A Board	52		92	D2
13	A3	(not V.G.)	53		93	D3
14	A4		54		94	D4
15	A5		55		95	D5
16	A6		56		96	D6
17	A7		57		97	D7
18			58		98	D8
19			59		99	D9
1A			5A		9A	DA
1B			5B		9B	DB
1C			5C		9C	DC
1D			5D		9D	DD
1E	On/Off	Dazzler	5E		9E	DE
1F	Mode	(Not V.G.)	5F		9F	DF
20			60		A0	E0
21			61		A1	E1
22			62		A2	E2
23			63		A3	E3
24			64		A4	E4
25			65		A5	E5
26			66		A6	E6
27			67		A7	E7
28			68		A8	E8
29			69		A9	E9
2A			6A		AA	EA
2B			6B		AB	EB
2C			6C	Status Tarbell	AC	EC
2D			6D	Data Tape	AD	ED
2E			6E	Status (Not V.G.)	AE	EE
2F			6F	Data	AF	EF
30			70		B0	FO
31			71		B1	F1
32			72		B2	F2
33			73		B3	F3
34			74		B4	F4
35			75		B5	F5
36			76		B6	F6 V.G.
37			77		B7	F7 Jovstick
38			78		B8	F8 Tarbell
39			79		B9	F9 Disk
3A			7A		BA	FA (not V.G.)
3B			7B		BB	FB
3C			7C		BC	FC
3D			7D		BD	FD
3E			7E	V.G. Video Digitizer	BE	FE Insai Memory (not V.G.)
3F			7F		BF	FF Front panel (not V.G.)

APPENDIX Q - MEMORY DIAGNOSTICS

MDIAG

If you have some reason to suspect that the computer's main memory is malfunctioning, use the Memory Diagnostic program on the MDOS System Diskette. Simply turn the system on, mount the system diskette in drive 0 (the right-hand drive), type B following the Monitor prompt (*), type MDIAG following the MDOS prompt (>), then depress the RETURN key. The program will load into the scratch-pad area of memory (not part of main memory) and execute.

MDIAG tests the contiguous memory beginning at 0000. There are actually two tests going on at the same time. Each repetition ("pass") of the program fills the next 8K block of memory with random numbers, and then tests it for changes. At the same time, it also fills all of memory with a certain fill code, and then tests all of it for changes. A display appears showing the result of each repetition. The program waits a certain "delay time" after filling before it tests memory. After all 8K blocks of main memory have been tested, the delay time is increased, and the program repeats beginning with 0000.

The display shows for each repetition the TOP OF MEMORY (the lowest address which is not in main memory), ACTIVE BLOCK (the first address of the 8K block currently subject to the random number test), the PASS NUMBER (incremented after each repetition), the FILL CODE (the code used to fill all of main memory as the second part of the test), the DELAY TIME currently being used, the number of ERRORS READ, and an ERROR DUMP showing the last 10 errors encountered, giving the address which malfunctioned, the code written, and the code read.

MDIAG will run indefinitely, with ever increasing delay times, if allowed to. It is used at the factory to burn the systems in for long periods of time. To stop it, depress the RESET key.

Monitor T Command

The T memory test is part of the Extended Systems Monitor PROM which comes in the system. To activate it, depress the RESET key, then type T, followed by the beginning address (in hex) of the block you want to test, then the ending address of the block. For example, to test the 48K of main memory, type T 0000 BFFF. The program will begin executing immediately.

The program stores random numbers into memory, then tests to see if any have changed. If memory is perfect, you will see nothing on the screen. However, if anything changes, the program displays the address, the code written, and the code read back. Then it continues testing. It will go on until stopped by depressing the RESET key.

Comparing with MDIAG, the strengths of the "T" test are 1) it allows

you to test portions of memory which are not contiguous beginning at 0000; for example an 8K block from E000 to FFFF, or the screen memory, normally from D000 to D7FF; 2) it displays ALL the errors, rather than the last 10, allowing you to pinpoint all malfunctioning locations, and 3) you can use it without disk drives, if needed.

The weaknesses are 1) it may not show up errors produced by dynamic memory over a delay time longer than T uses, whereas MDIAG increases the delay time to long enough intervals; and 2) you must know the ending address of memory. MDIAG is considered the better test for dynamic memories, which are used in the standard Vector Graphic systems.

Monitor N command

N is a non-destructive memory test. To activate it, depress the RESET key, then type N. It will make only one pass through memory, temporarily storing each byte, testing whether 00 and FF can be stored and recalled correctly, and then replacing the original contents. It does this until an error is found, whereupon it prints the address, the code written, and the code recalled, and then returns to the Monitor executive.

This program is most useful for determining how much main memory a given system has, because if no errors are found, it will print out the first address of ROM memory which is above main memory. The N test is not nearly as thorough as either the T test or MDIAG, and it only reports the first error found. However, it allows you to test memory without destroying any of the contents, unlike the others.

MAP

MAP is a utility which tells you whether RAM, ROM or no memory at all, is at each address in the system. This includes all special purpose memory such as video boards, scratch-pad, and so on. Use MAP if you are not sure what is in the system. If the system is standard, then the result should be the same as the map in Figure 1.2 (Chapter 1), with the PROMs appropriate for your configuration.

To run MAP, simply turn the system on, mount the system diskette in drive 0 (the right-hand drive), type B following the Monitor prompt (*), type MAP following the MDOS prompt (>), then depress the RETURN key. The program will load and execute. The resultant display is a matrix of memory that is fairly self-explanatory. You only have to know that the addresses increase from left to right in blocks of 100 Hex (256 bytes). You can run MAP with RAM holding data or program without losing anything; it is a non-destructive test, except for the area it uses itself, which is the scratch-pad area beginning from DC00 to DDFF.