MULTI-USER DISK OPERATING SYSTEM

FOR 8080 BASED MICROCOMPUTERS

By

REGINALD BYRON MASON
II

Bachelor of Science in Electrical Engineering

University of Oklahoma

Norman, Oklahoma

1970

Submitted to the Faculty of the Graduate College
of the Oklahoma State University
in partial fulfillment of the requirements
for the Degree of
MASTER OF SCIENCE
July, 1981

MULTI-USER DISK OPERATING SYSTEM

FOR 8080 BASED MICROCOMPUTERS

Thesis Approved:

Thesis Adviser

Dean of the Graduate College

PREFACE

This project involved the design of hardware and writing of software necessary to implement a multi-user environment on a single processor microcomputer system. The primary objective was to develop a versatile time sharing system for use by the microcomputer lab in teaching microprocessor fundamentals.

The author wishes to express his appreciation to his thesis adviser, Dr. Edward Shreve, for his guidance in the initial stages of this project. The author also wishes to thank Dr. Louis Johnson for his assistance in the final installation and benchmarking of the system.

The author extends a note of acknowledgement to Dr. Richard Cummins for his assistance and advice throughout the course of study.

A note of gratitude goes to my boss, Mr. Truman Hefner, and to the Western Electric Company, Oklahoma City Works, for providing the physical resources necessary for the culmination of this project.

Special acknowledgement is especially due to my wife, Becky, for her understanding and constant encouragement. She sacrificed many long hours diligently typing the early drafts and the final preparation of this manuscript.

Finally, special gratitude goes to my dedicated wife, Becky, our son, Brad, and our daughter, Beth, who often sacrificed their "family time" in order that I could pursue my graduate study, in addition to my work at Western Electric.

TABLE OF CONTENTS

Chapter Page

LIST OF FIGURES

NOMENCLATURE

Allocation Technique - The method of providing a process access to a shared resource.

Blocked - The state of a process waiting for either CPU time or a resource.

CDOS - Cromemco Disk Operating System.

Concurrency - The simultaneous logical and /or physical execution of several parts of a program.

Context Block - The state of a processor defined by the set of processor registers.

Context Switching - The swapping of a previously saved state of the processor with the current state of the processor in order to activate the previous process.

CP/M - Software operating system sold by Digital Research, Garden Grove, California.

CPU - Central Processing Unit (such as Z80, Intel 8080, etc.).

CRT - Cathode Ray Tube (user consoles).

CSB - Context Switching Board (provides time-multiplexing for system).

DMA - Direct Memory Access.

EOF - End-of-file mark (control Z = 1AH).

FAM - File Allocation Map.

ISIS - Intel System Implementation Supervisor. (MDS-800 Operating System).

Kernel - An operating system module that implements software processes and furnishes the means of interprocess communication; usually written in assembly language.

LDM - Load Module format file.

MDOS - Single user version of ZDOS that runs on an MDS-800.

Mutual Exclusion - The property that guarantees two or more different processes do not access a common resource, such as a buffer, simultaneously.

MUX - Multiplexer Circuit that provides interface to user consoles.

PL/M - Intel high level language similar to PL/I.

Priority - A property that designates a process' relative urgency.

Process - The basic unit of computation within an operating system. Also termed a software process to distinguish it from an abstract process, which is the task the software process implements.

Process-control Block - The data structure that defines a software process and its status.

Protection - A mechanism by which inviolate walls between any two processes is achieved; usually implemented in hardware.

PSB - Process Status Byte.

RAM - Random Access Memory.

RDOS - Firmware monitor supplied with Cromemco System.

Re-entrancy - A property of code that allows multiple copies of a code module to execute simultaneously; the code must not be self modifying, and data references must occur relative to the stack region.

Resource - Any device or item used by a computer, including special areas of memory such as buffers.

ROM - Read Only Memory.

SAM - Storage Allocation Map.

Time-slicing - The sharing of CPU time among several software processes by giving each process a defined interval (slice) of the CPU's time.

USART - Universal Synchronous-Asynchronous Receiver Transmitter.

Virtual Processor - The state of a processor normally defined by the set of internal registers, the stack pointer and the program counter.

ZDOS - Multi-user Disk Operating System designed for Cromemco systems.

CHAPTER I

INTRODUCTION

In September of 1978, Dr. Edward L. Shreve requested that the author investigate the possibility of providing the necessary hardware and software to implement a multi-user disk operating system for use by the Oklahoma State University Microcomputer Department. Since the author was familiar with Cromemco's Z80 based computer system, it was suggested that OSU purchase a Cromemco System Three. Cromemco was contacted about the possibility of converting CDOS (Cromemco Disk Operating System) software to a multi-user type environment. Cromemco had attempted this at one time, but ran into the problem of conflicting system calls. After consulting various references on time sharing techniques, it was decided that the system software had to be written from scratch. This thesis describes the culmination of this two and one-half year project.

In the beginning, the following design goals were formulated: (1) Write system software in a re-entrant form, i.e., no local variable storage, in order to facilitate conversion to a multi-user environment. (2) As a result of constraint #1, the system software could be ROM-based, which would add some intrinsic system memory protection and system reliability. (3) Include system files, such as the Editor and Assembler, as part of the ROM-based software, in order to save disk space for user files. This constraint was added to allow use of mini-

floppy disks as a mass storage device, despite their limited storage capabilities at the time. (4) Provide most system functions as system calls to enable the user to interface with system resources easily and efficiently. (5) Eliminate all the gingerbread that experience has shown to be relatively useless, e.g., default list files on the disk. As the project progressed, the above stated goals were modified as more experience was gained with the actual system results.

The Cromemco system software was not efficient and powerful enough to handle the task of creating the multi-user software on the target hardware environment. An MDS-800 Development System was available and provided the necessary software to complete the task. Early in the design cycle, the code was prepared on the MDS-800 and burned into ROMs for debugging on the target hardware system. But this proved to be tedious and very time consuming in light of the multitude of changes necessary. A scheme was therefore developed, whereby the overall system software as single-user could be simulated on the MDS-800.

Compatibility was maintained in passing parameters to disk drivers, so that Cromemco 4FDC disk controller drivers could be substituted in the final linking process to create the system software for use on the Cromemco computer. All code was located at 4000H and higher, so as to maintain the ISIS operating system in memory in case it was necessary to make code changes in the debugging process. It was therefore possible to boot from ISIS to MDOS (MDOS refers to the single-user version of the software that runs on the MDS-800) with relative ease, which made the development of the software run much smoother.

Another helpful aid in system development involved using the boot facility of the RDOS ROM in the Cromemco system. A program called

Loader was written on the MDS-800 using the ISIS system calls. This program copies a short binary bootstrap loader similar to the one that boots CDOS, onto a blank formatted disk located in Drive 1. It then copies ZDOS onto track 0 through 2 (about 8K). A modified version of RDOS now boots ZDOS in the same manner as it boots CDOS. In this way, it is not necessary to burn ROMs each time a change is made. If these methods were implemented earlier in the design process, countless hours of development time could have been spared. The current implementation of ZDOS is still being booted from a disk into RAM. Figure 1 gives a pictorial representation of the multi-user system.

Once the basic operating system reached a certain level of maturity and the Editor and Assembler were written, it was possible to use the system itself to make or try modifications to itself. In fact, since MDOS is written in assembly language, as opposed to PL/M in the case of ISIS, MDOS runs about 3 to 4 times faster than ISIS. It became easier to write patches to MDOS, using MDOS than to re-edit MDOS sources files and re-link to whole system. There are still, however, areas of MDOS that would have to be improved to bring it up to the full power of ISIS.

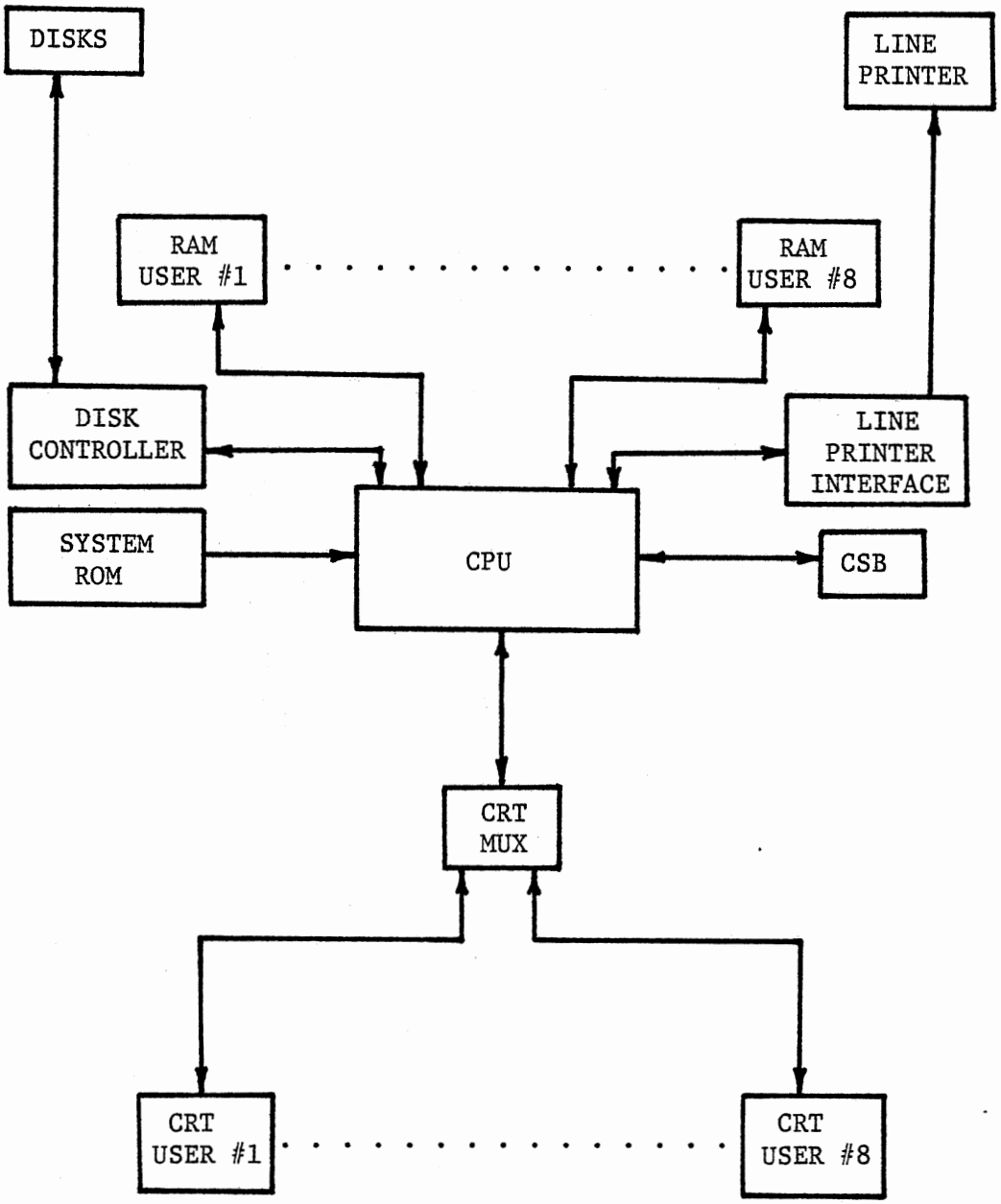


Figure 1. Multi-user System Configuration

CHAPTER II

SYSTEM HARDWARE

Computer Configuration

Multi-user ZDOS was developed for implementation on a Cromemco Z2-D or System 3, but should work with little or no modification on any S-100 bus microcomputer equipped with bank switchable memory boards. Any discussion of specific hardware will assume a Cromemco Z80 based system as the hardware environment. A Z80 processor is not necessary, as the system software was written in 8080 code. It would be necessary however, to change the interrupt scheme for context switching.

ZDOS was written in such a manner as to be ROM-based. The present configuration runs out of a 16K RAM board located at 8000H. This allows retaining the RDOS ROM at 0C000H, enabling the system to run the Cromemco supplied CDOS operating system. ZDOS software is written in a re-entrant fashion, and as such, does not store any variables local to itself. There is no system RAM to maintain. All data variables and symbols are stored in user RAM. All user RAM occupies the same address space in one of eight banks of memory. This hardware feature is available on Cromemco and other S-100 bus memory boards. Output port 40H is used as the bank switching port. Each bit corresponds to a bank of 64K of addressing space. The bank select logic circuit decodes commands sent to output port 40H by the CPU to determine if the memory bank requested should be active. If the CPU sends a logic one to any

bit position for which the bank select switch is on, the board will be enabled. Otherwise the board will be disabled. On power-up, the active memory bank is 0. Only one bank of memory is active at any given time. Each user may have as much as 32K of memory in the current configuration. The RAM board at 8000H on which the system software resides is set up with all eight bank switches on so that it will be enabled in all banks when context switching occurs. Cromemco supplies memory boards in 4K and 16K increments, but 16K of RAM per user is suggested as an ideal configuration. For more information, refer to the Cromemco 16KZ RAM Instruction Manual.

The Cromemco processor card is a Z80 based CPU designed to simulate all the S-100 bus signals that were originally defined for an 8080 processor chip. The CPU card is switch selectable to run at 2 or 4 MHz. This switch setting is irrelevant to ZDOS.

The Cromemco 4FDC Disk Controller card is a non-DMA, programmed I/O type interface. It uses a Western Digital 1771 Disk Controller chip. This type controller is not the best suited for a multi-user environment, but ZDOS makes up for this by the use of a novel interlocking scheme. This technique is discussed briefly in the next section.

A custom hardware board, CSB/MUX (Context Switching/CRT Multiplexer), is required to implement a multi-user environment. The CSB supplies interrupts at 33 msec. intervals (unless modified by system software). The CRT Multiplexer handles the interface of eight CRT consoles to the system. This special hardware will be described in more detail in the next section.

An ideal hardware configuration suitable for immediate implementation with current software and hardware available would consist of the following:

1 Cromemco Z2-D or System 3 computer

1 Cromemco Z80 CPU board

1 Cromemco 4FDC Disk Controller

1 Cromemco 16K RAM board for system software

1 Cromemco 16K RAM board per user

1 Custom built CSB/MUX board

Context Switching Board

The CSB is the heart of the multi-user system. The CSB generates the basic 33 msec. time slice used by the system to share system resources among 8 users. Refer to Figure 2 for a block diagram of the CSB used in the following discussion. The system clock is fed to a divider chain that produces a 33 msec. pulse train. This pulse train sets the interrupt flip-flop, which in turn applies a ground to the interrupt request line (PINT) of the processor. During software initialization, the I-register was loaded with the upper 8 bits of a vector address of the context switching subroutine and the processor was placed in interrupt mode 2 (IM 2). When the interrupt occurs, the CPU issues an interrupt acknowledge (SINTA) which gates the lower 8 bits of the vector address of the context switching subroutine onto the data bus. The CPU uses this vector to fetch the address of the context switching subroutine and branches to the same. Since the interrupt enable flag is disabled, no other interrupt can occur until context switching has been completed. Refer to the section on context switching

in Chapter VI for further explanation.

If a user is performing a compute bound function, i.e., not doing disk I/O or console I/O, a full 33 msec. time slice is given to the user. Otherwise, the length of the time slice is a function of the I/O being performed. The system software relinguishes use of the current time slice by the use of an output instruction to port 0EFH. This action causes the time slice clock to be cleared and forces an immediate context switch to the next user. In this manner, a 33 msec. time slice is not wasted looping on a CRT ready flag or doing a disk seek command. This scheme increases the throughput of the system several orders of magnitude. The average time slice length in this case is approximately 140 usec.

The CSB also maintains a user number port at 0EEH. Each bit of this port corresponds to a user (bit 0=user 1,...,bit 7=user 8). The context switching software maintains this port by reading the port, rotating the contents to the left, and writing it back out. The system reads this port during directory accesses to check to see if the current user owns the file being requested. This port is duplicated as a write only port for use by the CRT multiplexer to gate the proper USART status and data onto the data bus. The format of this port is identical to the format of bank switching port.

The CSB maintains a port at 0E8H called the Process Status Byte (PSB). The PSB can be considered as a collection of 8 flags that the system uses to keep track of the system resources. These flags form an interlocking mechanism that prevents fatal conflicting calls to shared resources. Refer to Figure 4 for an explanation of the PSB bits. For example, a user may request a sector of data from the disk, set bit 0 of

the PSB and force a context switch to the next user. In this manner, disk I/O does not hold up the system while doing a seek. The bit set in the PSB prevents the next user from initiating a conflicting disk request until the current disk I/O is complete. In using this method, the need for system memory to store I/O queues was eliminated. The only drawback is that disk I/O is not necessarily first come, first serve. For example, let us assume user #1 currently has control of the disk and user #7 requests disk services. User #7 sees that bit 0 of the PSB is set. User #7 forces a context switch to user #8. User #7 status is saved at the disk seek routine. Now assume user #1 has completed reading a sector of data. The system resets bit 0 in the PSB and forces a context switch. This prevents any one user from "hogging" the disk. User #2 is doing compute bound operations and utilizes a full 33 msec. time slice. But now assume user #3 decides to read the disk. Since bit 0 in the PSB is reset, user #3 is able to request a seek to the disk. Now user #3 has the disk tied up and when user #7 is switched in, he finds the disk busy again. If user #4 through #6 likewise in the interim queue up for the disk, user #7 must wait. On the average, this does not cause much of a wait. At no time can any one user access more than one sector before all other users have a chance to access a sector. For a clearer understanding of how this method of using the PSB works, refer to the listing of the DSEEK subroutine in module ZTYPE. The input port at 0EFH is used during booting and system initialization. The modified RDOS ROM checks to see if the CSB/MUX board is installed in the system. If not, the RDOS monitor is entered. If the board is present, bit 7 of port 0EFH contains the status of the multi-user switch. If the multi-user mode is enabled, the system bootstrap loader is read from

track 0, sector 1 into memory bank 0 and executed. This bootstrap in turn boots the system kernel into a 16K RAM board at 8000H. The system initialization routine inputs the BCD number of users switch setting from port 0EFH, bits 0-3. If the number of users selected is out of the range of 1 to 8, then the system reports:

ILLEGAL MAX USER

and halts. The input port at 0EBH is wired such that:

value=1 SHL (number of users)

i.e.

USER NO.	PORT VALUE
1	0000 0010
2	0000 0100
3	0000 1000
4	0001 0000
5	0010 0000
6	0100 0000
7	1000 0000
8	0000 0000

The reason for this implementation was to reduce the computation necessary by the context switching subroutine, thereby reducing the time overhead necessary to switch to the next user. For a better understanding of this, refer to the discussion of the context switching subroutine SWITCH in module ZTYPE.

CRT Multiplexer

The CRT Multiplexer circuit is responsible for interfacing all eight users to the system. The design utilizes eight Intel 8251A Universal Asynchronous-Synchronous Receiver-Transmitters (USART). Refer to Figure 3 for a block diagram of the MUX used in the following discussion. As mentioned in the last section, the user number port at 0EEH is duplicated as write only port. Whenever the I/O decoding

circuit detects an I/O DATA or STATUS, this signal is gated with the current user number, and is used to assert the chip select line on the current user's USART chip. Therefore the correct USART chip will drive the data bus in response to an I/O DATA or STATUS request. The USARTs are initialized to the proper mode of operation during system initialization at boot time.

Figure 5 gives a breakdown of the port addresses used by the CSB/MUX circuit board.

Startup Procedure

After following the instructions supplied by Cromemco and checking basic system sanity by booting up CDOS, perform the following steps to reconfigure the system for ZDOS:

1. Replace the RDOS ROM on the Disk Controller with the modified ROM.
2. Mount the eight user I/O cable to the back of the computer.
3. Install the CSB/MUX board in the system and hook up 8 user I/O cable.
4. Install a 16K RAM board at 8000H with all 8 bank switches enabled. (Note: this is very important as the system RAM must answer to all banks when a context switch occurs).
5. Install a 16K RAM board at 0000H for each of the users to be implemented. Set the bank switch for each user RAM board accordingly, i.e., user 1 in bank 0, user 2 in bank 1, etc.
6. Set multi-user mode switch on CSB/MUX board. LED should be on.
7. Turn on system and insert multi-user system disk in drive 0. System should boot and prompt on each user's CRT.

In order to boot CDOS, turn off multi-user mode switch, and switch user #1 CRT cable to I/O connector on 4FDC board. Insert CDOS disk in drive 0 and reset the computer. CDOS should boot into memory.

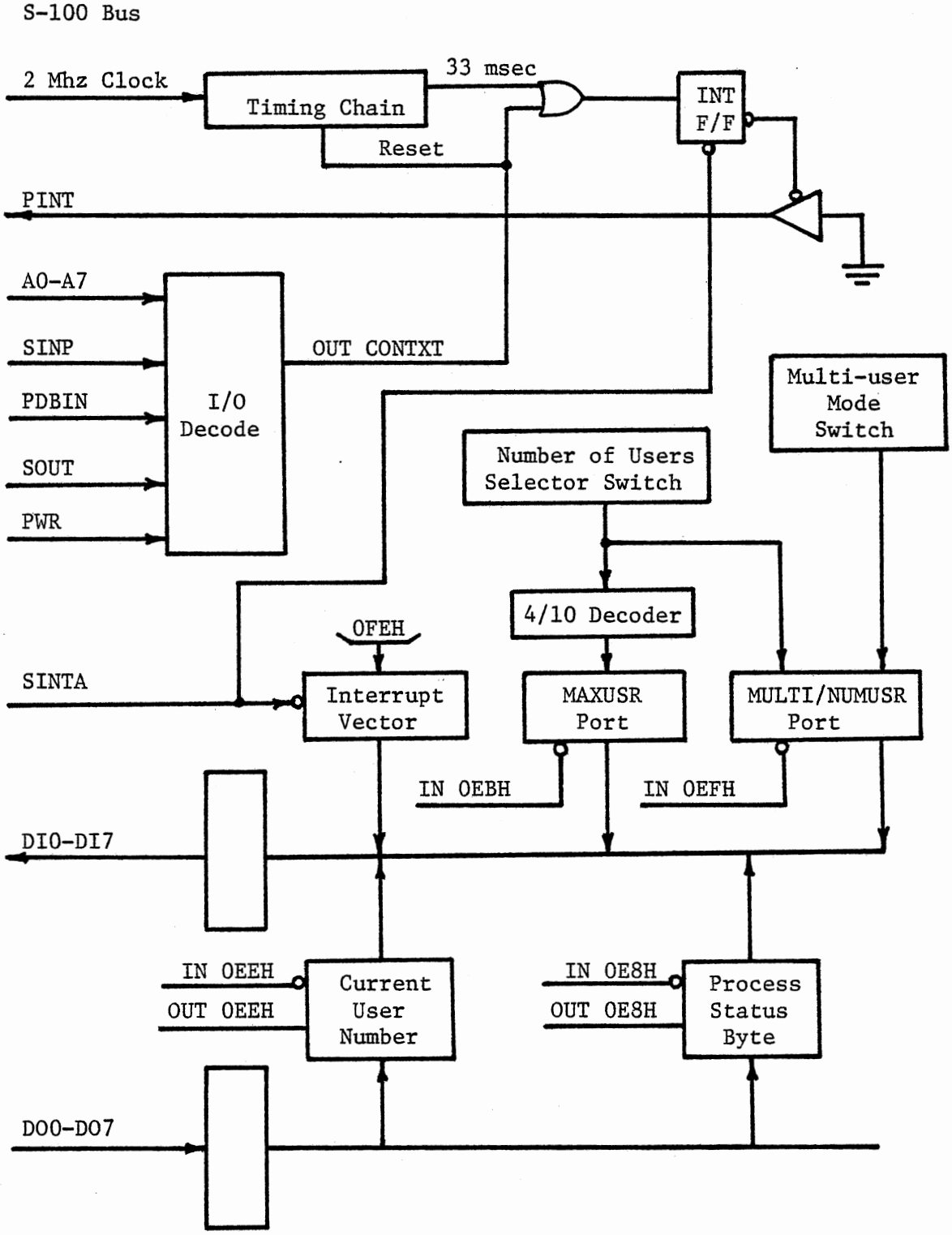


Figure 2. Context Switching Board Block Diagram

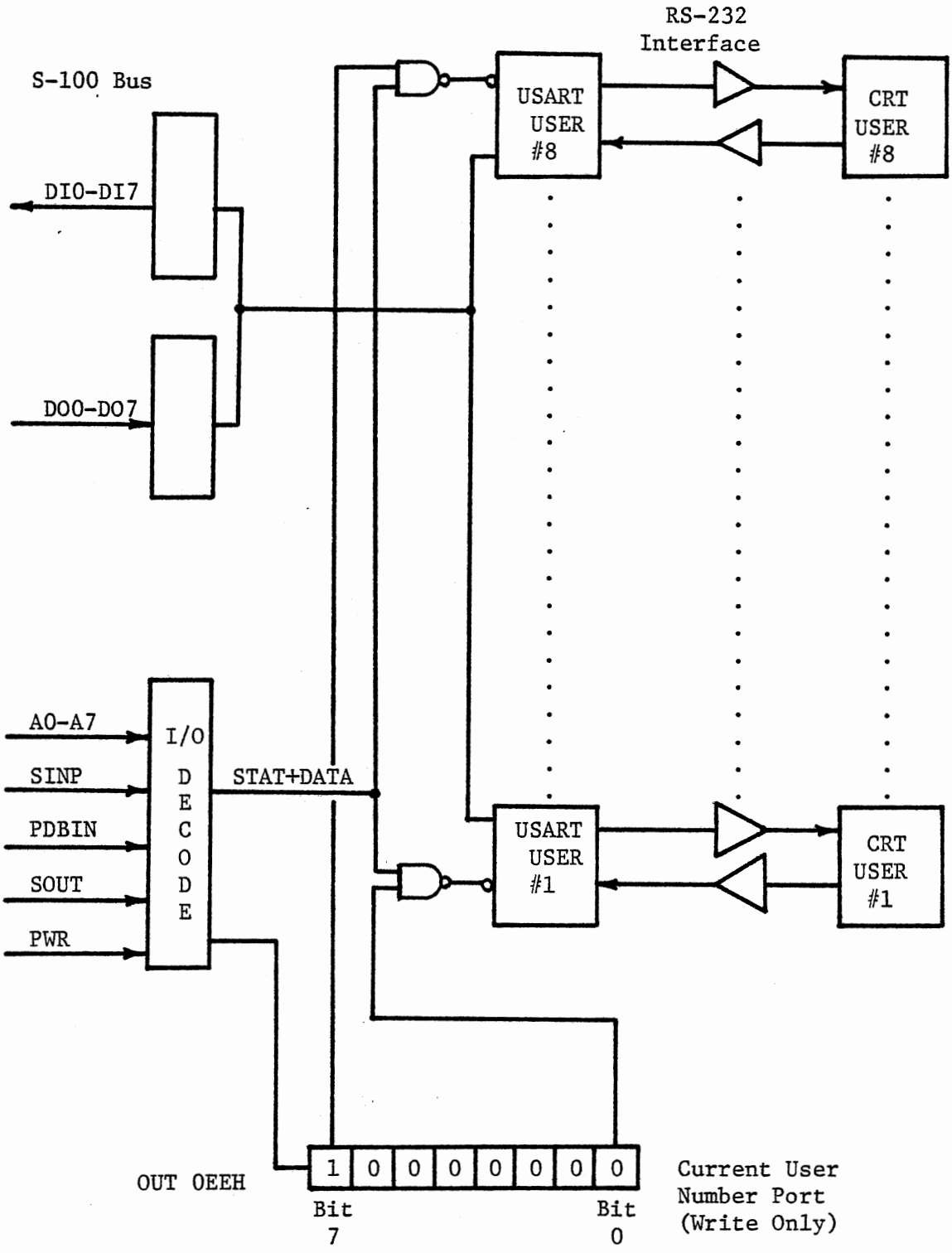
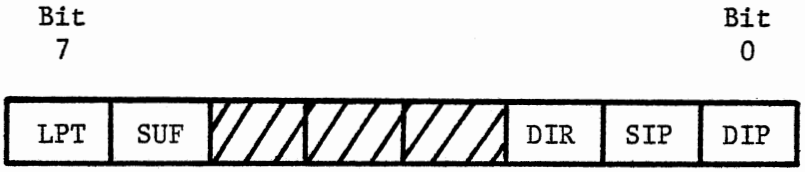


Figure 3. CRT Multiplexer Block Diagram



Port Address = 0E8H

Bit 0 - (DIP) Disk I/O in Progress
Bit 1 - (SIP) Sector Allocation in Progress
Bit 2 - (DIR) Directory Allocation in Progress
Bit 3-5 - Reserved for future use
Bit 6 - (SUF) Super User Flag
Bit 7 - (LPT) Line Printer in Use

Figure 4. Process Status Byte

PORT ADDR	INPUT	OUTPUT
E8	PSB	PSB
E9	-	-
EA	-	-
EB	MAXUSR	-
EC	CRT DATA	CRT DATA
ED	CRT STATUS	CRT COMMAND
EE	CURRENT USER	CURRENT USER
EF	MULTI/NUMUSR	CONTXT

Figure 5. CSB/MUX Input and Output Port Addresses

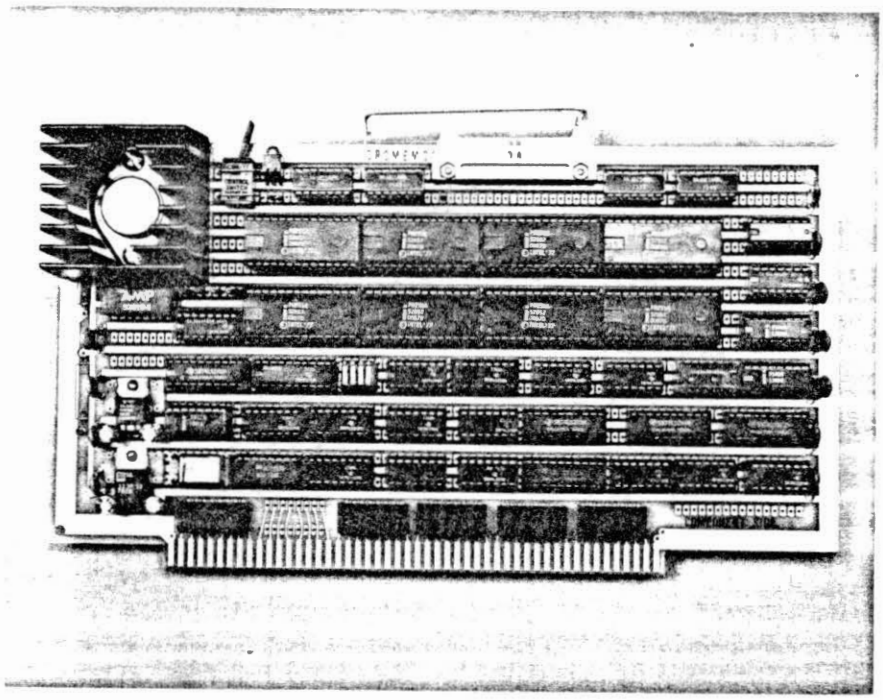


Figure 6. CSB/MUX Circuit Board

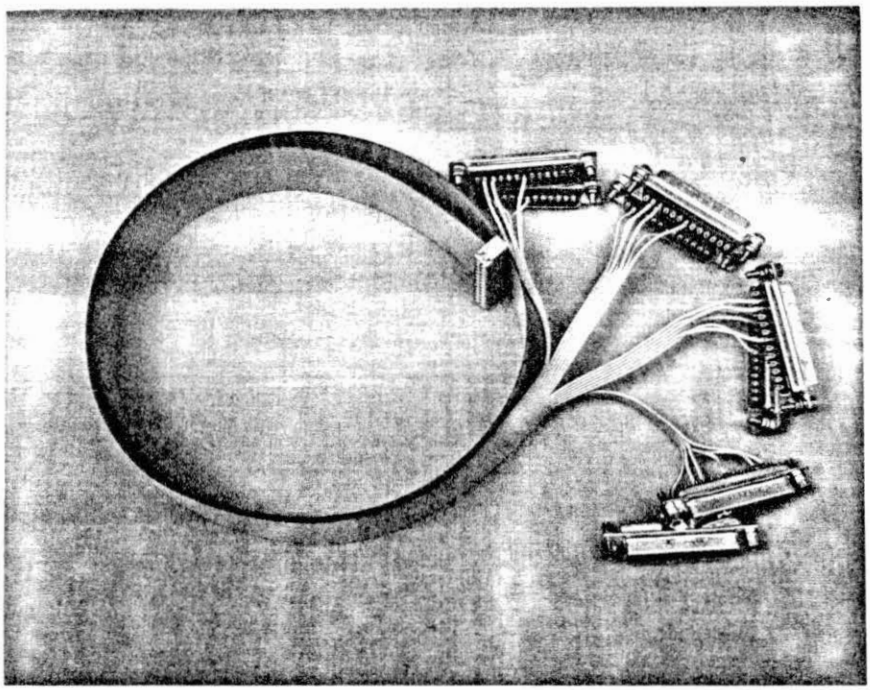


Figure 7. Eight-User RS-232 Serial I/O Cable

CHAPTER III

ZDOS ORGANIZATION

System Overview

At present, there are two implementations of the basic disk operating system software; MDOS and ZDOS. MDOS is the single-user version that runs on the Intel MDS-800 Development System. ZDOS is the multi-user version that runs on the Cromemco Z2-D or System Three. The only considerations that differentiate the two systems are the system I/O driver modules MTYPE and ZTYPE. These modules contain disk I/O software drivers, console and line printer I/O drivers and system initialization routines. ZTYPE contains additional routines to implement the multi-user system. The intended purpose in future discussions will center on descriptions of ZDOS, but general software descriptions not involving multi-user capability will also apply to MDOS.

ZDOS is a multi-user disk operating system designed to allow eight users to share a single processor system through time division techniques. Each of the users' CRT consoles are interfaced through a special CRT multiplexer circuit, hereafter referred to as the MUX. Existence of multiple terminals is invisible to system I/O software, i.e., all terminals are referenced through the same data and status port number. The context switching board (CSB) provides the necessary clock and timing control of interrupts to multiplex each user with the system.

Currently, ZDOS is loaded from the disk into a 16K Ram board addressed at 8000H. The first 600H bytes of user RAM is reserved for system variables, buffers, etc. This leaves 14,848 bytes of RAM available to each user. Track 3 of the disk is reserved for the storage allocation map (SAM) and disk directory. Track 0 through 2 contains the bootstrap loader and the ZDOS system software. This leaves 1924 blocks of 128 bytes each (246,272 bytes) available for user files. The directory may contain 192 files. Each file carries with it a user ID number which makes it accessible only to its owner. In this manner, a single disk directory appears to the users as eight separate directories, each of which are invisible to the other users. The basic interface between ZDOS and the user is accomplished through the use of system calls. Using a single system call, e.g., a user may fetch a single byte at a time from a disk file. ZDOS contains intrinsic commands that perform useful tasks such as, listing the directory, erasing a file from the directory or typing a file on the console. The user may create extrinsic command files using the ZDOS Text Editor and 8080/Z80 Assembler. ZDOS contains a rather useful intrinsic command called DISK which allows individual track and sectors to be read, modified and written back on the disk. Certain intrinsic commands, such as DISK, are reserved for the exclusive use of the system administrator and are controlled by the use of the "super user" function.

Diskette Organization

The mass storage medium for ZDOS is a dual-drive Per Sci Model 277 eight inch floppy disk supplied as standard equipment with Cromemco Computer Systems. This drive uses a standard IBM 3740 soft sectored format single density diskette. Each diskette can store 256 Kilobytes of data. An eight inch disk is organized as 77 concentric tracks, each containing 26 sectors of 128 bytes each. The ZDOS bootstap loader is located on track 0, sector 1. The modified RDOS ROM loads the bootstrap into bank 0 at address 80H in the same manner as it loads CDOS. The bootstrap reads the ZDOS kernel from track 0, sector 2 through track 2, sector 26 into the RAM board located at address 8000H. Track 3, sector 1 and 2 contains the storage allocation map for the entire diskette. Track 3, sector 3 through 26 contains the disk directory. Out of the possible 2002 sectors available on a disk, only 78 are reserved for system use. This leaves 1924 blocks of 128 bytes each (246,272 bytes) available for storage of user files. Refer to Figure 8 for the layout described above.

Memory Ogranization

Each user in the current implementation of ZDOS is allocated 16 Kilobytes of RAM located in the address space 0 to 3FFFH. The first 600H bytes of each user RAM is reserved for use by the system. This organization is defined by the module ZPAGE0. The remaining 14,848 bytes of user RAM is available for use by system or user defined extrinsic commands or programs. User RAM can be expanded by adding

another 16K RAM board in the address space 4000H to 7FFFH. This will give the user about 31K of useable RAM. ZDOS occupies the address space 8000H to 0BFFFH, residing in a 16K RAM or ROM board. Currently, ZDOS is executing out of RAM. The address space 0C000H to 0FFFFH is occupied by a 1K RDOS ROM (0C000H to 0C3FFH). Refer to Figure 9 for the layout of the system memory.

File Specification

All data stored on the disk, whether ASCII or binary, is referred to by the system through a file specification. A file specification consists of an optional disk drive specification, a file name, and an optional file type extension. All ZDOS device specifications contain two characters enclosed by colons. Drive 0 is specified by :D0: and drive one by :D1:. File names are limited to a maximum of six characters, the first of which must be alphabetic. File extensions are limited to a maximum of three characters, the first of which must be alphabetic. File type extensions are normally used to give an indication of the type of data in the file. ZDOS only reserves the use of .HEX for hexadecimal files, .JOB for submit command files, ASCII source files an extension of some kind, such as .ASC or .SRC. Most executable system programs or extrinsic commands have no extension.

Command String Interpreter

After the system hardware boots in ZDOS, control is transferred to the DOS base level routine which performs system initialization and some preliminary housekeeping chores. System initialization is quite involved and will be discussed in another section.

The stack pointer is set and system initialization subroutine is called. After this ZDOS looks for a file on the disk called POWRUP.HEX. If this file exists, it is loaded and executed. In this way, the user may define the way ZDOS is booted. POWRUP.HEX may contain instructions for a different boot message format to be printed on the console or may load still another user created program that would utilize ZDOS without allowing it to report its existence. If POWRUP.HEX does not exist, then ZDOS reports the following to the console:

ZDOS, Vx.x

where x.x is the current version number. The console bell is sounded with the boot message. ZDOS then prompts with the current logged-in drive which in this case is the default drive 0:

D0>

This is the prompt of the ZDOS base level entry point. Whenever the system completes an intrinsic command or performs a system exit, this prompt signifies to the user the successful completion of the task and the continued sanity of the system. Sometimes this prompt is provided by the batch file handler (SUB intrinsic command. At this point in the base level, the BATCH flag is checked. If set, the submit command re-entry point is fetched and entered. If BATCH=0, page 0 of memory is re-initialized. Page 0 initialization involves restoring the jump to base level instruction at address 0000H and restoring the jump to the system call handler at address 0005H. This insures basic system sanity, in case a users program destroyed ZDOS reserved RAM inadvertently. The default File Control Blocks, FCB 1 and FCB 2, are initialized in case they were closed out properly.

The base level prompt is actually provided by the command string interpreter subroutine, GETCMD. GETCMD inputs a command string from the user while checking for some basic syntax. GETCMD will only return with a valid command string. Extraneous spaces are eliminated from the command string as it is entered. If control R is typed, the current command string along with the prompt is reprinted for viewing by the user. If control X is typed, the command string buffer pointer is reset and the system reprompts for a new string. If an illegal character is inputted, GETCMD reports:

SYNTAX ERROR

on the console. Legal input characters are upper case alphanumerics and the following legal delimiters: period, space, comma, colon and carriage return- line feed. If the rubout key is typed, characters are erased and echoed on the console. If rubout past the beginning of the command string is attempted, the system will reprompt. The command string input buffer is 128 bytes long. Buffer overflow will result in a syntax error message.

When GETCMD returns with a valid (syntactically only) command string, the system compares the first name in the command string with the list of current intrinsic commands. Intrinsic commands are utilities resident in the kernel of the system which perform certain mundane tasks necessary for maintenance of the system, e.g., ERA erases a file from the disk or DIR lists the directory. See the section on intrinsic commands for their descriptions. If an intrinsic command is found, the command string buffer pointer is updated to the first character following the command delimiter and control is passed to the command. If the name was not in the intrinsic command list, then the

system assumes this is the name of an extrinsic command (either system or user created). ZDOS now searches the current logged-in disk for the filename. If not found, the system reports to the console:

:Dx:FILE.EXT, NO SUCH FILE

where x is the current logged-in disk. If the file is found, the system attempts to load and execute the program. If the user inadvertently typed the name of a non-hex or non-binary format file, the system reports:

ACCESS ERROR

If the file was indeed an executable program, the program is loaded, the command string pointer is updated to the first byte following the filename delimiter and control is transferred to the program. Through the use of the GETUSR system call, the user may pass switches and parameters to the program from the input command string.

Once the program is loaded, the system is at the mercy of the user's program. ZDOS will maintain checks on the proper use of its resources through the use of system calls, but otherwise cannot prevent an invalid jump or infinite program loop from hanging up the system. If this occurs, the user may utilizes the local rebooting facility of the BREAK key. User programs should use the EXIT system call to re-enter ZDOS properly.

Intrinsic Commands

Intrinsic commands are resident utility programs that perform useful system functions. Most of the commands use the PARSER system call to process their command string. Therefore the command format is input specification to output specification. Some input specifications

only involve a drive number and some output specification only a list device. In all cases the default drive is 0 and the default list device is the CRT. Optional parts of the command will be enclosed in <>. We will define filespec as :Dx:filename.extension. The following is a list and brief description the current commands available. These commands can be found in the module INTRIN.

*** ERA ***

Erase a file intrinsic command is used to delete files from the disk directory. The format of the command is:

```
ERA filespec 1,filespec 2,...,filespec n
```

There are three situations reported by ERA:

```
filespec, ERASED
filespec, NO SUCH FILE
filespec, WRITE PROTECTED
```

*** FREE ***

Free blocks intrinsic command reports to the console the number of free sectors remaining on the disk. The format of the command is:

```
FREE <:Dx:> <TO> <:list dev:>
```

The number of free blocks is reported to the console:

```
xxxxx FREE BLOCKS ON DRIVE x
```

where xxxxx is the decimal number of free sectors and x is the drive number.

*** ATR ***

Attribute intrinsic command modifies the write protect flag of the file in the disk directory. The command format is:

```
ATR filespec W0          ;to reset the flag
ATR filespec W1          ;to set the flag
```

The command reports "SYNTAX ERROR" if the W switch is missing or if the 0 or 1 is incorrect.

*** REN ***

Rename intrinsic command renames a file on the disk from the old name to the new name. The command format is:

REN oldfilespec TO newfilespec

If the oldfilespec does not exist:

oldfilespec, NO SUCH FILE

If the oldfilespec is write protected:

oldfilespec, WRITE PROTECTED

If the newfilespec name already exists on the disk:

newfilespec, ALREADY EXISTS

If the old file name and the new file name are not on the same drive:

SYNTAX ERROR

This prevents a user from inadvertently renaming a file of the same name on another drive.

*** TYPE ***

Type a file intrinsic command lists a file on the system list device. The format of the command is:

TYPE filespec <TO> <:list dev:>

If an attempt is made to type a non-ASCII file, the system reports:

ACCESS ERROR

*** LOG ***

Log intrinsic commands allows a user to change the default disk drive from drive 0 to drive 1 or back again. The format of the command is:

LOG :Dx:

For example, if a user logs on drive 0, ZDOS will prompt with 'D1>', instead of 'D0>'. If the user types DIR without a device specification,

the directory for drive 1 will be listed. To get the directory for drive 0, the command would have to be DIR :D0:.

*** BYE ***

Bye intrinsic command exits the ZDOS base level to the system monitor level, if the current ZDOS implementation allows it. In the multi-user configuration, ZDOS can't allow exitting to Cromemco's RDOS monitor, due to the differences in the CRT port addresses. In MDOS, the MDS-800 monitor is entered.

*** DEBUG ***

Debug intrinsic command sets the system debug mode flag, loads the file specified and reports the starting address to the console. The format of the command is:

DEBUG filespec

The system reports:

START ADDRESS=xxxx

where xxxx is the hex starting address of the loaded program. This address is passed to the debug monitor by the user. The current implementation of ZDOS has the resident debugger disabled. In the MDOS single-user mode, the user passes the starting address to the monitor with the G command.

*** GET ***

Get file intrinsic command loads a file into memory without transfering control. This command is useful in loading several modules of a program into memory. The user can create a submit command file consisting of several GET command lines and then load the mainline control program to begin execution. GET is also very handy in overlaying patches created by the system to modify itself. The format

of the command is:

GET filespec

*** DIR ***

Directory intrinsic command lists the disk directory on the list device. The format of the command is:

DIR <:Dx:> <TO> <:list dev:>

The DIR command lists the filename and extension, user ID, number of bytes in the file, the number of disk blocks associated with the file, the write protect flag and the file type attribute. The number of blocks used out of 2002 blocks available is also listed.

*** SUB ***

Submit intrinsic command allows command strings to be entered to the system from a disk file. The user creates the submit file as an ASCII file using the system editor. The file must have a .JOB extension in order to be recognized by the system as a submit file. It is not necessary to include the .JOB extension in the SUB command string. The format of the command is:

SUB filename<.JOB>

The system opens the submit file for input using FCB 3 and continues reading an executing commands until an end-of-file mark is encountered. The submit file is fully interactive, i.e., a command string in the submit file may call the Editor. The user can edit the file opened by this command. When an exit from the editor is performed, the submit command immediately takes up where it left off. The SUB command has the interesting characteristic in that it can call itself recursively. If for example, the last command line in a submit file called EXAM.JOB was SUB EXAM, the system will simply re-open EXAM.JOB and start all over

again. The only way to escape from the infinite loop is to use the BREAK key to reboot oneself. Caution must be used in doing this, as a reboot does not close out any files currently open for output and will therefore not deallocate unused blocks. These blocks will be lost for use by the system. Only type the BREAK key during commands that are simply listing files or the directory. The main function of this recursive ability of SUB is to set up non-interactive jobs to test the system operation of all 8 users by one person.

*** DISK ***

Disk intrinsic command is actually a collection of commands into a resident monitor that allows real-time examination and repair of the disk resource. Since this monitor allows the modification of individual sectors on the disk, its use by the average user should be discouraged. The DISK command prompts for local commands with an asterisk (*). The local commands are as follows:

*R - read a disk sector

The read command prompts for drive number, track number and sector number. It then displays the sector read on the list device in an 8 by 16 matrix. The ASCII equivalent of the bytes in the sector are also displayed. If a line feed or a space is typed, the next sequential sector is read and displayed. This can continue until track 76, sector 26 is displayed. The track and sector numbers are entered as hexadecimal values (tracks 0 to 4CH, sectors 1 to 1AH).

*W - write a disk sector

The write command prompts for drive number, track number and sector number. It then writes out the current disk I/O buffer

to the sector specified.

*E - exit to DOS

The exit command returns to the DOS base level.

*Dxxxx,yyyy - display command

The display command displays all bytes on the list device from xxxx to yyyy. Both are hexadecimal values.

*Sxxxx - substitute command

The substitute memory command will continue to display the contents of memory locations starting at xxxx as long as the space bar is typed. The contents of these locations may be modified by entering the new value. A cr-lf terminates the command.

*Fxxxx,yyyy,zz - fill command

The fill memory command fills all locations from xxxx to yyyy with the value zz.

The substitute command can be used to modify the IOBYT to change the list device from the console to the line printer. The display, fill and substitute command follow the same command input format as the Intel monitor.

*** COPY ***

Copy intrinsic command copies files. The format of the command is:

COPY filespec 1, filespec 2,..., filespec n TO output filespec

Copy allows copying of ASCII or hex files only. If any of the input file specifications do not exist on the disk, the output file is closed and deleted, and the non-existent file is reported to the console. The following intrinsic commands exist only in the multi-user ZDOS system:

*** WHO ***

The who intrinsic command reports user ID number to the user's console.

USER x

where x is in the range 1 to 8.

*** CID ***

The change ID intrinsic command changes the user ID number to allow access to a file by more than one user. The format of the command is:

CID filespec oldID newID

The old ID is replaced with the new ID on filespec. Since access to user files is privileged, only the super user is allowed to change ID numbers. The CID command prompts for a password before allowing its execution. The password is fixed at system generation and cannot be changed in real-time.

*** SLIST ***

The super list directory intrinsic command allows the super user to get a directory listing of all the files on the disk, regardless of ID number. As with the CID command, SLIST prompts for the password.

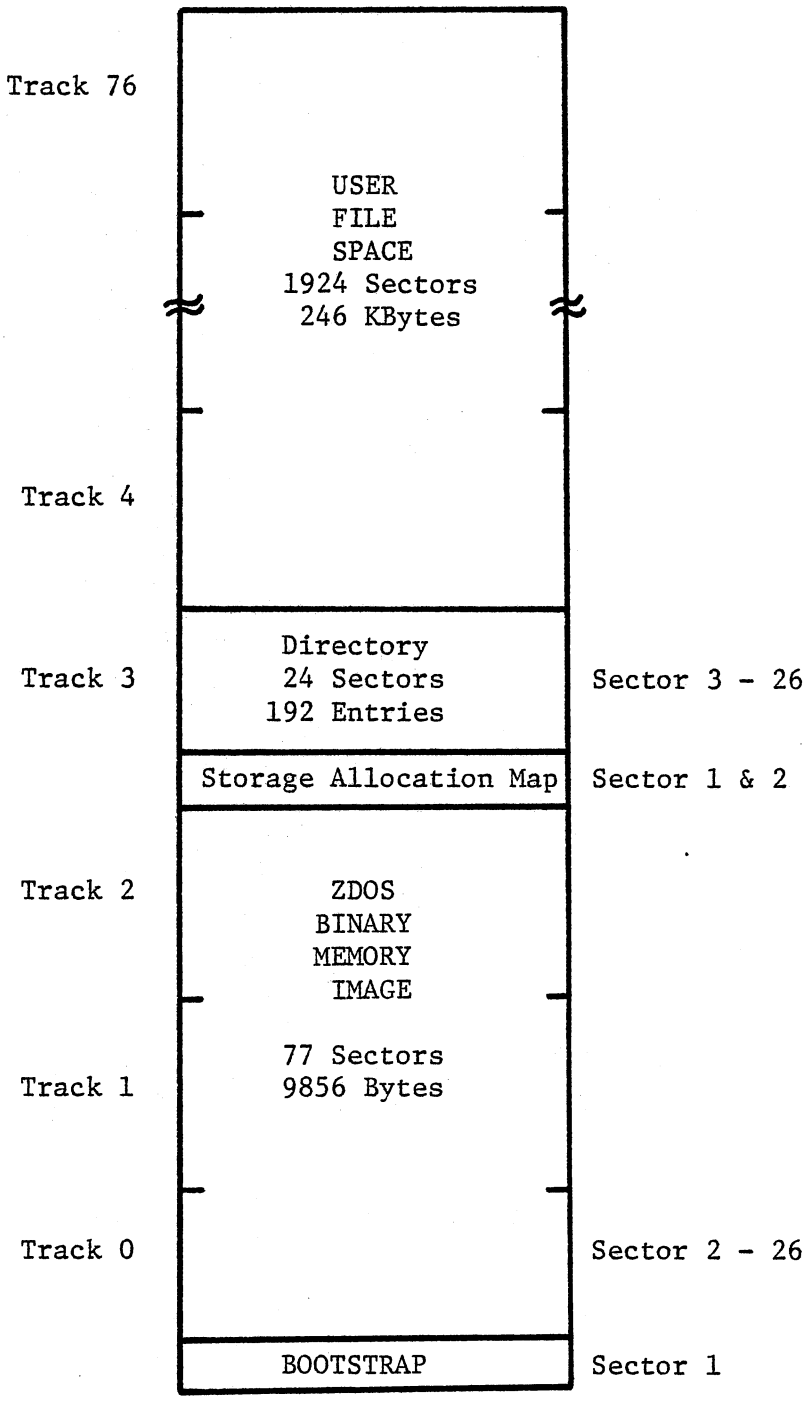


Figure 8. Eight Inch System Diskette Organization

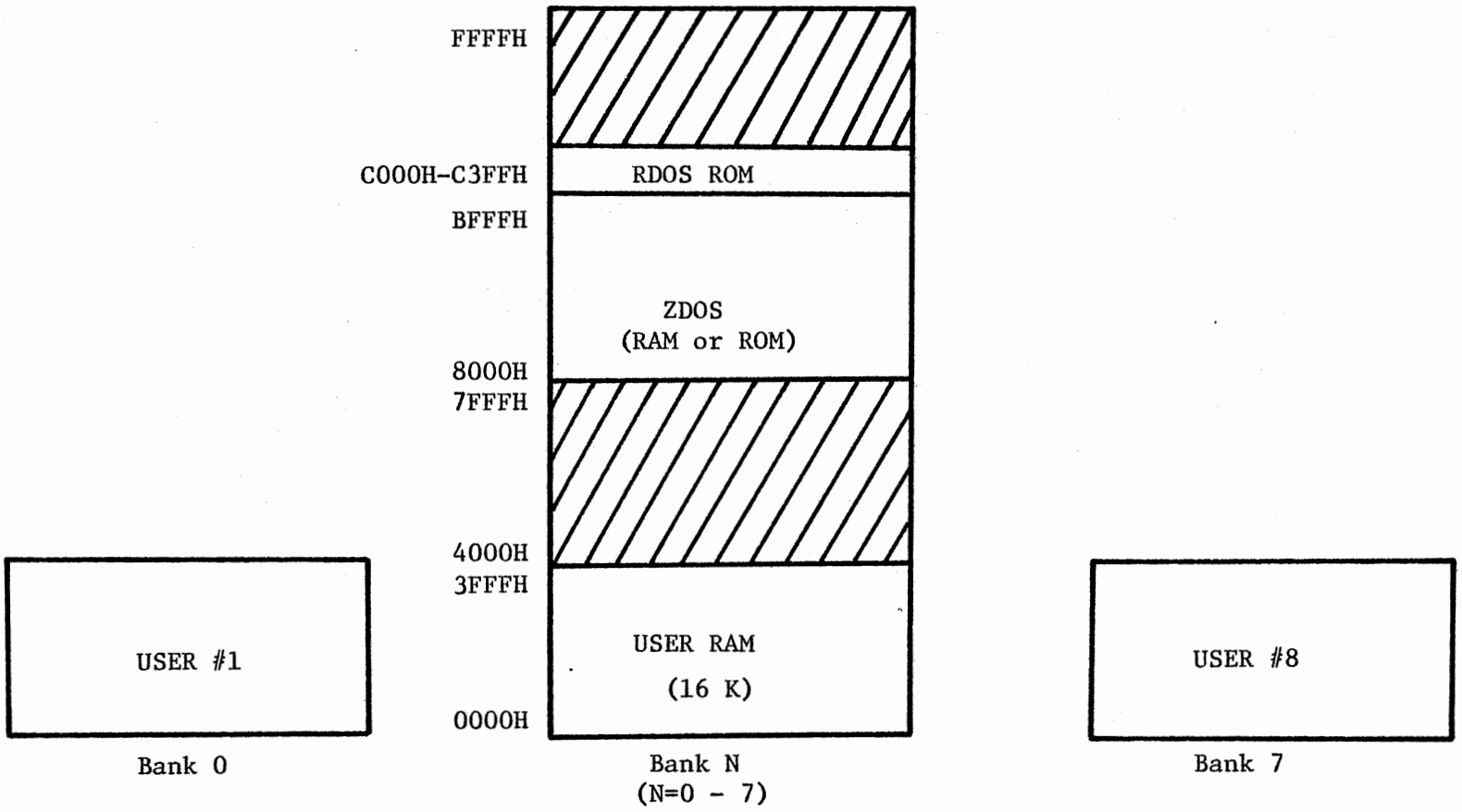


Figure 9. System Memory Map

CHAPTER IV

ZDOS FILE SYSTEM

Directory Organization

The ZDOS directory resides on track 3, sector 3 through 26 of the disk. These 24 directory sectors each contain space for 8 entries for a total of 192 possible files. Directory entries are 16 bytes in length. The entry format is as follows:

Byte 1-6	Six character file name, left justified, padded with spaces.
Byte 7-9	Three character extension, left justified, padded with spaces.
Byte 10	User identification number
Byte 11 bit 0,1	File type (00=ASCII, 01=BINARY, 10=HEX, 11=SYSTEM)
Byte 11 bit 7	Write protect flag
Byte 12	Byte count less than a complete block (1 to 80H)
Byte 13	Low block count of number of blocks allocated to file.
Byte 14	High block count of number of blocks allocated to file.
Byte 15	Sector containing first File Allocation Map block.
Byte 16	Track containing first File Allocation Map block.

The total byte count of a file is defined by byte 12, 13, and 14 (BYTNUM,LBLKNO and HBLKNO respectively) according to the following algorithm:

BYTE COUNT=128*(BLOCKS - (BLOCKS/62 + 2)) + BYTNUM

where BLOCKS=HBLKNO SHL 8 + LBLKNO. The maximum byte count of a file is only limited by the physical storage space on the disk.

Byte 15 and 16 (FAMSEC and FAMTRK respectively) contain a sector and track which hold the first File Allocation Map (FAM) of the file.

The FAM is a 128 byte block that contains a list of track and sectors associated with a particular file. The FAM is actually considered as part of the file. The number of blocks required for a file of N bytes can be found by:

TOTAL BLOCKS=63*(N/128)/62.

Directory entries are created and maintained by the system invisible to the user.

Storage Allocation Map

Program module SAMIO (Storage Allocation Map Input/Output) includes seven subroutines necessary to control allocation of disk storage resources; namely SAMSCH, READSM, WRTSAM, ALLCAT, DEALLO, COMPUT, and FILSAM. A description of each will be covered in a moment.

The Storage Allocation Map (SAM) resides on sector 1 and 2 of the directory track (normally track 3). The information contained in these two sectors should be viewed as a sequence of 2002 bits, each bit representing one of 2002 sectors on the disk (77 tracks times 26 sectors/track equals 2002 sectors). A one bit indicates the particular sector represented is not currently associated with any file (free disk space). A zero bit indicates this sector is not free. The allocation of sectors for files is totally random in nature (except in the case of LDM files as will be explained later). When a sector is requested, the next available sector starting at the left end of the bit stream is returned. See Figure 10. A better understanding of the mechanics of disk sector allocation can be provided by briefly covering the individual routines in module SAMIO.

SAMSCH (Storage Allocation Map Search)

SAMSCH is made up of two routines, Samsch and Getnxt. Getnxt is the re-entrant part of SAMSCH. Samsch is always called first in order to initialize SAMPTR to point at the SAMBUF (SAMBUF is 256 bytes reserved by the system strictly for use by SAMIO) and SAMCTR to 0 to count the bit position from the beginning of the buffer (0 - 2001). SAMPTR and SAMCTR are maintained by the system so that each subsequent call to Getnxt returns the next available free sector on the disk without starting the count from the beginning. Getnxt does this by fetching each byte from the SAMBUF and rotating all 8 bits through the carry flag until a non-zero bit is found. The carry flag is set to zero and the bits are rotated back the correct number of times until the current byte is restored and placed back in the SAMBUF. Since SAMCTR was bumped for each bit rotated, it now contains the bit position from the beginning of SAMBUF which corresponds to the track and sector number to be returned. By maintaining SAMPTR and SAMCTR, Getnxt is made re-entrant, thereby saving valuable time that would be wasted by making Getnxt start looking for the next free sector at the beginning of the bit stream each time. SAMSCH reports "DISK FULL" to the console and flags an error if no more sectors are available.

COMPUTE (Compute track and sector)

The bit position in SAMCTR is passed to the COMPUT subroutine which computes the track and sector number of the next available sector for use by the system disk I/O routines. This accomplished by the following algorithm:

TRACK=SAMCTR/26 SECTOR=SAMCTR MOD 26 + 1

It is necessary to add one to the sector number since sectors count from 1 to 26, instead of 0 to 25.

READSM (Read SAM into SAMBUF)

READSM reads the SAM from the disk directory track into memory.

WRTSAM (Write SAM from SAMBUF to the disk)

WRTSAM writes the SAM from the SAMBUF in memory to the directory track on the disk.

Both READSM and WRTSAM check the MAPSW to determine whether the SAM is being read from a mini-floppy (1 sector) or an 8 inch maxi-floppy (2 sectors).

ALLCAT (Allocate a full FAM block)

Since ZDOS has been written in a manner to facilitate conversion to a multi- user environment, ALLCAT and DEALLO were written specifically for multi-user implementation of the system software. In a multi-user mode, writing a single sector of 128 bytes would require the following sequence: 1) reading the SAM into memory 2) obtaining the next available sector 3) writing out the SAM to the disk (as all users are using the same copy of the SAM) 4) writing out the block of 128 bytes to the disk file. An ASCII source file of 50 blocks is not uncommon. Writing out this file would require 150 accesses to the disk. Disk accesses slow system response time, especially if several of the users were doing similar tasks of writing to the disk. Disk activity would be frantic to say the least. One possible alternative to remedy this problem would be to allocate certain portions of the disk to each user. But this proposal ties up valuable disk space. The method chosen uses a preallocation scheme to reduce disk accesses. Each FAM (File Allocation Map) block points to a possible 62 sectors associated with each file.

Therefore instead of filling the FAM block one sector at a time, the entire FAM is preallocated with the full complement of 62 sectors. As the file is written out to the disk, the FAMBUF is used as the source of the next available sector. After the file is completely written out, the unused sectors are returned to the SAM. Now the sequence of events for writing a file are: 1) read SAM into memory 2)preallocate FAM with 62 sectors 3) write out SAM to disk 4) write each block to file on disk 5) read SAM again 6) return unused sectors to SAM 7) write SAM to disk. In a file of 50 blocks, only 54 disk accesses are required, as compared with 150. In general, most ASCII files are less than 62 blocks. If greater than 62 blocks, it still only requires 4 accesses per each additional 62 blocks, as opposed to 124 additional accesses. There are slight drawbacks to this method. Take for example the case where the disk is nearly full. ALLCAT grabs all sectors remaining for current use (less than or equal to 62). Even though this user may only require 5 sectors for his use, all other users will be advised that the disk is full. After this user closes the output file and DEALLO returns unused sectors to the SAM, free blocks will be shown as available. Any other user at this time could ignore the warning and grab some more blocks as ALLCAT will preallocate sectors even though there are less than 62 remaining. Since this problem presents itself only when the disk is nearly full, preallocation is still the best method of reducing disk activity to a minimum.

FILSAM (Fill in the SAM)

FILSAM uses the current track and sector number to set the appropriate bit in the SAM to indicate that this sector is now available for use. It performs the reverse function of SAMSCH. FILSAM is used by

the DELETE system call to erase files from the disk by means of the following algorithm:

SAM BIT OFFSET=TRACK*(SECTORS/TRACK) + (SECTOR - 1)

SAMIO is an integral and important part of the disk file handling system. Incorrect handling of the SAM results in a crashed disk as we shall see in the section on multi-user implementation of the storage allocation map.

Directory Map

The directory map was implemented to decrease disk accesses necessary to list the system directory. This implementation aided both single-user and multi-user modes of operation. This was especially important to the multi-user mode in that directory listing is a function performed quite often and when multiplied by 8 users, amounts to a lot of disk activity, most of which contained no information useful to the user.

The directory consists of 24 sectors each containing 8 file entries. On a disk with as few as 32 files, only 4 sectors in the directory would contain useful information. This would mean 20 seeks were unnecessary. Assuming all 8 users requested directory listings at the same time, this would amount to 160 disk seeks and reads. Therefore it was decided to implement a directory map using three bytes of the SAM. Refer to Figure 10.

As with the SAM, the DIRMAP can be thought of as a bit stream of 24 bits. Each bit represents a single sector of the directory (or 8 possible entries). If any one of the 8 entries in a directory sector is occupied, the DIRMAP bit is set to a zero. When all 8 entries in a

sector are empty, the bit is only then set to one. The directory listing intrinsic command DIR then uses the DIRMAP to determine if a sector will be searched for an entry or not. Response time to DIR is greatly enhanced and without much software overhead to implement DIRMAP.

File Allocation Map

Sectors allocated to a file are kept track of by the use of FAM blocks. FAM blocks can be thought of as a sub-directory of a file. A FAM block is organized as a 128 byte sector containing a reverse link to the previous block, a forward link to the next block and 62 pairs of sector and track numbers assigned to this particular file. Refer to Figure 11. The reverse link in the first FAM block is always zero, and the forward link in the last FAM block is always zero.

Refer to Figure 12 as we discuss the operation of the FETCH subroutine found in module FTCNXT, in order to gain a better understanding of the FAM block. FETCH is used by the RDBYTE (read a byte) system call to read the next byte from a file open for input. The FAMENT byte (FAM entry number) in the FCB (File Control Block) points to the current pair of track and sector numbers (1 to 62) located in the current FAM block resident in the system FAMBUF (reserved by each FCB). FAMENT also doubles as a flag to maintain proper control of reading in new FAM blocks. FAMENT is set to 0 by the OPENI system call. This signifies that there is no FAM block resident in the FAMBUF associated with this FCB. When FAMENT=0, FETCH gets the FAMTRK and FAMSEC from the FCB. FAMTRK (FAM track) and FAMSEC (FAM sector) point to the location on the disk of the first FAM block. This block is read into the FAMBUF associated with this file and FAMENT is set to a one to be used as an

offset to the first pair of track and sector numbers. Notice also since it is non-zero, FAMENT also now signifies that a FAM block is resident in memory. If the pair equals zero, there are no more sectors to be read from this file. If the pair is non-zero, they are returned in TRACK and SECTOR to the calling routine and FAMENT is bumped to point to the next pair in the FAMBUF. When FAMENT=63, this is one more than the number of possible pairs in this FAM. When this occurs, the forward link to the next FAM block is fetched and the next FAM block is read into the FAMBUF. FAMENT is then set to one again and the process of fetching sectors and FAMs is continued until a zero pair is encountered. At this time, FETCH signals the calling program that there are no more blocks to be read. Although the system does not currently use the reverse link, it is maintained by the system to allow backing up in a file at some future time.

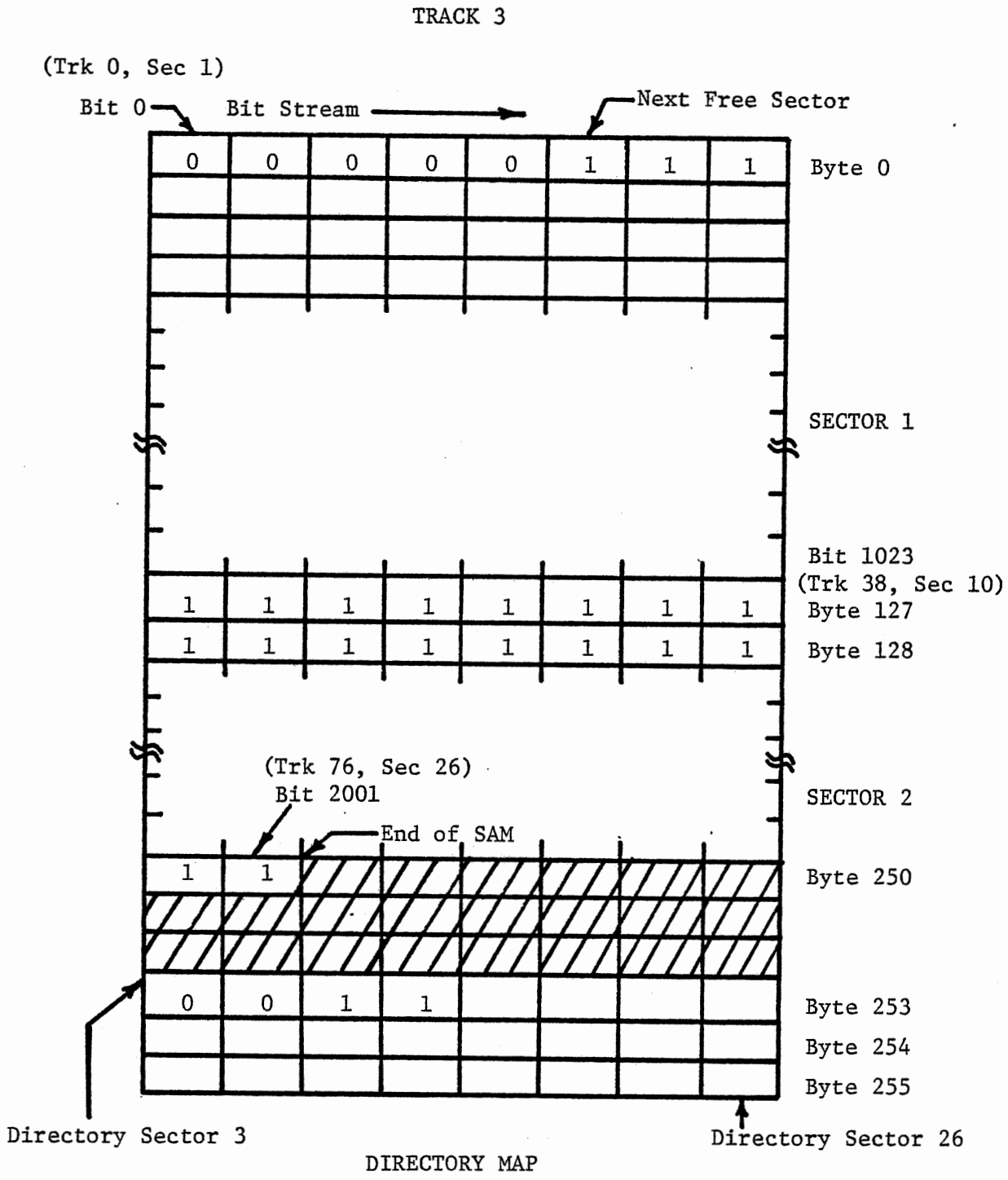
WRNBLK performs the reverse role of FETCH in that it creates and maintains the FAM blocks as records are being written to a file. It uses FAMENT as a flag in the same manner as FETCH. OPENO (Open for output) system call sets FAMENT=0. When FAMENT=0, WRNBLK fetches a sector from the SAM and assigns it to the FCB as the current FAMTRK and FAMSEC. At this time, it also preallocates the FAM block with 62 pairs of track and sector numbers from the SAM. As each call to WRNBLK is made, the next track and sector is fetched from the FAMBUF until all 62 pairs are exhausted. At this time, FAMENT=63 signaling that the current FAMBUF must be written out to the disk and a new FAM allocated. Before the current FAM is written out, the forward and reverse links are updated, thereby linking the FAMs. When all bytes have been written to the file, the CLOSE system call deallocates all unused track and sector

numbers from the current FAMBUF, and writes out this FAM to the disk.

The DELETE system call uses the linked FAM blocks to erase a file from the disk. FAMTRK and FAMSEC from the directory image is used to read in the first FAM block. All track and sector pairs are deallocated and returned to the SAM. Deletion is terminated when the first zero pair is encountered. If the FAM was full, then the forward link track and sector is used to fetch the next FAM block. This process continues until the file has been effectively erased from the disk. Please note that this system does not fill erased sectors with deleted file marks (0E5H) as some systems do. It is therefore possible to regenerate an erased file by recreating the directory entry as long as the first FAMTRK and FAMSEC were known and no disk output was performed in the meantime.

All file handling and disk storage allocation described up to this point is totally invisible to the user and is maintained accurately by the system. The RDBYTE and WRBYTE system calls are the main means of data transfer between the system and the user. In many ways, RDBYTE and WRBYTE are very similar to FETCH and WRNBLK, respectively. As with FAMENT, BYTNUM is an offset maintained by the system in the FCB. BYTNUM points at the current byte being read or written (1 to 128) in the FCBUF (File Control Block Buffer) associated with this file. BYTNUM also doubles as a flag. In RDBYTE, when BYTNUM=0, this indicatesno bytes have been read from the file. After FETCH has been called to make FAM block resident and return next block to read, the block is read into the FCBUF and BYTNUM is set to a one as an offset to the first byte to be returned. Each call to RDBYTE returns a byte to the user in the A-register and bumps BYTNUM. When BYTNUM=81H, this signals the system to

read in the next data record into FCBUF. This continues until an end-of-file mark (EOF=1AH) is encountered. An error is reported if FETCH a no more blocks to read indication before an EOF is found. A similar discussion applies to BYTNUM as used by WRNBLK and can been readily seen by examining the module WRNBLK.



TRACK = BIT NUMBER / 26

SECTOR = BIT NUMBER MOD 26 + 1

Figure 10. Storage Allocation Map (SAM)

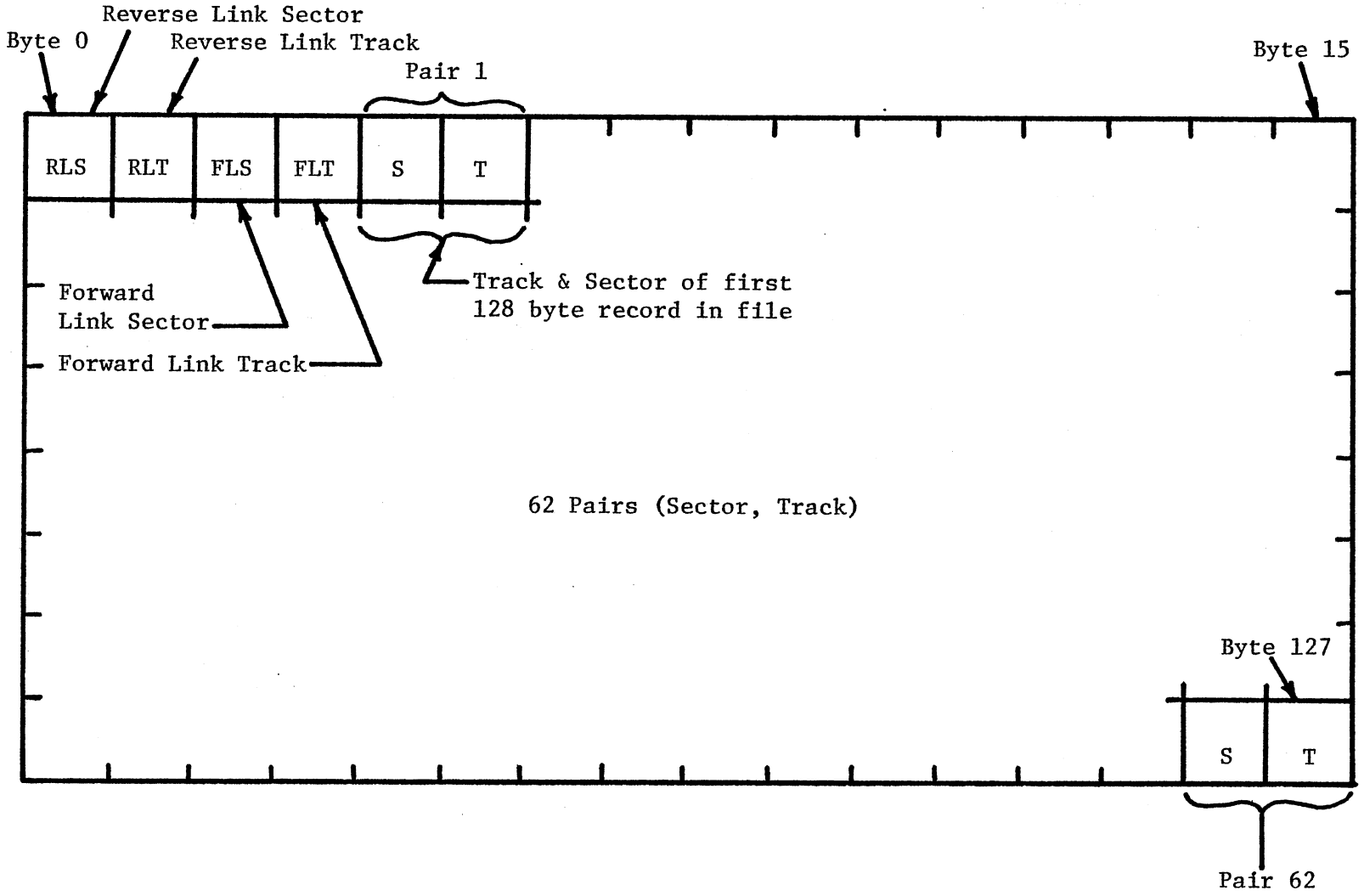


Figure 11. File Allocation Map (FAM) Block

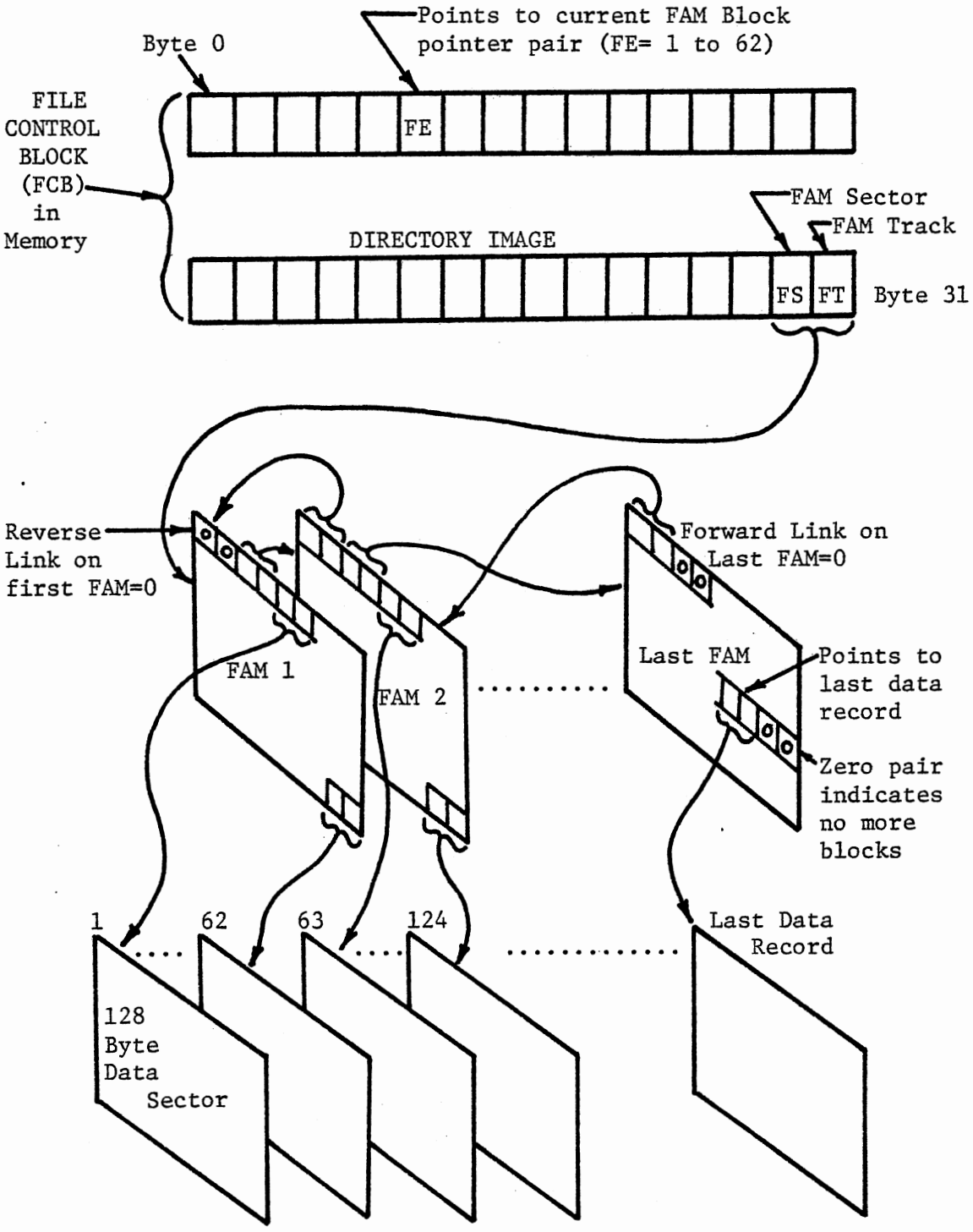


Figure 12. Disk File Organization

CHAPTER V

INTERFACING WITH ZDOS

File Control Block

All files on the disk are referenced by the user through file control blocks. FCBs are 32 bytes of memory space that contain information the system uses to access a file on the disk. The first 16 bytes of information is temporary in nature, in that it is maintained only so long as the file is open for I/O. The second 16 bytes is the exact image of the directory entry and is saved in the directory when the file is closed. The format of the FCB is as follows:

FCBSTT - Byte 0 is the file control block status word.

bits 0-3 are unused

bit 4 defines the file access type. if bit 4=0, the file is open for input. if bit 4=1, the file is open for output.

bit 5 is unused.

bit 6 is set if directory update is required when the file is closed.

bit 7 is set if the file control block is active.

BUFPTR - Byte 1,2 is a buffer pointer to a 128 byte disk I/O buffer associated with this FCB.

FAMPTR - Byte 3,4 is a buffer pointer to a 128 byte file allocation map (FAM) block buffer associated with this FCB.

FAMENT - Byte 5 is an offset to the next track and sector pair in the current FAM block.

ENTCNT - Byte 6 is an offset to the directory entry (0-7) located on DIRSEC and DIRTRK.

DIRSEC - Byte 7 is the directory sector where the entry associated with this file resides.

DIRTRK - Byte 8 is the directory track where the entry associated with this file resides.

BYTNUM - Byte 9 is an offset to the current byte position in the file I/O buffer (FCBUF). The system uses BYTNUM=0 to indicate that no bytes have been read from or written to a file. BYTNUM is in the range from 1 to 80H. BYTNUM=81H signals the system to read another sector into the file I/O buffer.

UNUSED - Bytes 10 through 14 are reserved for future use.

DRSPEC - Byte 15 contains the disk drive specification associated with this file.

DIRECTORY IMAGE

FILNAM - Byte 16 - 21 contains the six character file name, left-justified, padded with spaces.

EXTEN - Byte 22 - 24 contains the three character extension, left-justified, padded with spaces.

USERID - Byte 25 contains the user identification number associated with this file.

ATTRIB - Byte 26 contains the file attributes.

bit 7 is set if the file is write protected.

bits 6 - 2 are unused at present.

bits 1,0 specify the file type; 00=ASCII, 01=HEX, 10=BINARY,

11=SYSTEM.

BYTES - Byte 27 contains the number of bytes less than a block in a file (128).

LBLKNO - Byte 28 is the low byte of the number of blocks count.

HBLKNO - Byte 29 is the high byte of the number of blocks count.

FAMSEC - Byte 30 is the sector of the first FAM block associated with this file.

FAMTRK - Byte 31 is the track of the first FAM block associated with this file.

At present, the system maintains two default file control blocks that it uses for intrinsic commands and extrinsic file accesses. For example, EDIT uses FCB1 for the input file and FCB2 for the output file. The default FCBs are available for use by the user. More FCBs may be created by the user at any time. Each additional FCB requires 32 bytes for itself and two contiguous 128 byte buffers for disk I/O and FAM block maintenance. The INTFCB system call will initialize the extended FCBs for the user. Once created, the extended FCB is maintained by the system software. One additional FCB is maintained by the system for use by the batch file (SUB command) handler. This FCB is not available for use by the user.

System Calls

Most of the physical resources of the system can be accessed efficiently and easily through the use of system calls. During system initialization, address 5 (4005H in MDOS) is set to contain a jump instruction to the ZDOS system call handler. The normal call procedure involves passing the call number in the C-register, passing the address

of the File Control Block (FCB) in the DE-register and performing a call subroutine instruction to address 5. For example, assuming FCB 1 contains the name of a file, doing a directory search for this file name would be performed as follows:

```
MVI C,10        ;DNSRCH system call
LXI D,FCB1
CALL ZDOS       ;where ZDOS was equated to 5 earlier
ORA A
JZ FOUND
```

Since the address of the default FCBs are not known by the user, the system call handler accepts FCB1=1 and FCB2=2 as parameters passed in the DE-register. Extended FCBs are addressed by their absolute addresses supplied by the user. The system call handler saves all registers, but each system call may or may not return certain registers intact to the user. The current legal range of system call numbers is 0 to 63, although not all numbers are used at present. If a non-implemented or number greater than 63 is passed to the handler, the following message is reported to the console:

```
ILLEGAL SYSTEM CALL n
```

where n is a decimal number in the range 0 to 255. An illegal system call is treated as a fatal error condition. Any open files are closed and control is not returned to the user's program.

The following is a list of the currently available system calls and a brief description of their function. Included in each description are input parameters to be passed to the call, output parameters returned by the call and a list of registers destroyed or modified by the call. The number in parentheses is the system call number.

The following system calls are found in the module SCALL0:

*** EXIT *** (0)

Exit system call closes both default file control blocks before returning to the DOS base level routine. No registers are saved since there is no return to the calling program. No parameters are passed.

*** CNSIN *** (1)

Console in system call returns a character from the console in the A-register with the parity bit stripped off. No parameters passed. A-register is only one modified.

*** CNSOUT *** (2)

Console out system call outputs a character passed in the A-register to the console device. A-register is only one modified. Certain special control characters are filtered by the call. Control S halts the output listing to the console. Any other key will restart the listing. Control C does a system exit to DOS base level. Any other key entered is returned in the A-register.

*** CNSTAT *** (3)

Console status system call returns the console ready indication in the A-register. If A=0FFH, then there is a character awaiting input from the console. If A=0, then there is no input. No parameters are passed.

*** LSTOUT *** (4)

List out system call outputs a character passed in the A-register to the list device specified by the IOBYT (either the console or the line printer). System call IOSET is used to set the I/O byte. Control S and control C perform the same function as in CNSOUT. If any key is pressed during listing, the character inputted is returned in the A-

register.

*** IOSET *** (5)

I/O set system call sets the system I/O byte to the list device passed in the B-register. Line printer is specified by 80H and the console is specified by 40H. A and B registers only ones used.

*** IOGET *** (6)

I/O get system call fetches the system I/O byte into the A-register. The IOBYT format is compatible with Intel's definition, although only bits 0,6 and 7 are used. Bit 0 is normally set to a one to indicate the CRT is the console device. Bits 6 and 7 define the system list device. Bit 6 set indicates CRT (40H) and bit 7 set indicates line printer (80H). The user must mask the IOBYT to get the bits of interest.

*** LISTHL *** (7)

List HL system call lists the string addressed by the HL-register on the list device specified by the IOBYT. The string address is the only parameter passed. The string must be terminated by a 0. The HL-register is returned pointing at the first byte passed the 0 terminator. Tabs (ctrl I) are maintained at 8 character position intervals. Registers used are A,B and C.

*** BINASC *** (8)

Binary to ASCII system call lists a binary number on the list device passed in the B-register as a string of 5 ASCII characters, right justified, with leading zeroes suppressed and printed as spaces. The binary number is passed in the HL-register. The IOBYT remains unchanged after the return from the system call. B=40H specifies the CRT as list device and B=80H specifies the line printer.

The following system calls are found in the module SCALL1:

*** DNSRCH *** (10)

Directory search system call searches for the existence of a file name on the disk. The file name is assumed to have been already appended to the FCB by the MOVNAM system call. The FCB address is passed in the DE-register. DNSRCH returns the status of the name in the A-register. A=0 if the file, extension and user ID matched the FCB. A=0FFH if the name was not found in the directory. All registers are maintained.

*** PRTNAM *** (11)

Print name system call prints the filename specification contained in the FCB passed in the DE-register. No cr-lf preceeds or follows the output to the console. E.g., if filename was HELLO, the extension was ASC and the drive specified was 1, the following would be printed on the console:

```
:D1:HELLO.ASC
```

*** RENAM *** (12)

Rename system call renames the name in the FCB specified in the DE-register to the new name pointed to by the address in the HL-register. The new name string must be terminated by a legal terminator (cr, lf, space, tab). The status of the call is returned in the A-register. A=0 if the rename was successful. A=1 if the old name in the FCB is write protected. A=2 if the new name already exists on the disk. A=0FFH if the old name does not exist on the disk. Note: The RENAME intrinsic command checks to see if both names have the same drive specification before renaming, RENAM system call does not. Therefore there it is possible to change the name of a file on drive 0 that you

intended to change on drive 1.

*** DELETE *** (13)

Delete system call erases a file name specified in the FCB passed in the DE-register from the disk. The status of the deletion is returned in the A-register. A=0 if name was found and deletion was successful. A=1 if file is write protected. A=0FFH is file was not found. All registers are maintained.

*** MOVNAM *** (14)

Move a name system call appends a name pointed to by the HL-register to an FCB specified by the DE-register. The HL-register is returned pointing to the first character following the string delimiter which must be legal (cr, comma, space). The filename must be 6 characters or less and the extension must be 3 characters or less. The first character of the filename and extension must be alphabetic. If a legal drive specification is included (:D0:,:D1: or :D2:) the DRSPEC in the FCB is updated. MOVNAM reports "ILLEGAL DEVICE" or "SYNTAX ERROR" to the console and exits to DOS base level. The current user ID is appended to the FCB at this time. If no drive is specified with the name, the default drive 0 is appended.

*** RENEXT *** (15)

Rename the extension system call renames only the extension in the FCB specified in the DE-register to the new extension pointed to by the HL-register. HL-register is returned still pointing at new extension string. The extension string must consist of 3 characters. If less than three alpha-numerics, left justify and pad with spaces.

The following system calls are found in the module SCALL2:

*** OPENI *** (16)

Open for input system call opens the file specified in the FCB passed in the DE-register for input of bytes from the file to the user. OPENI copies the directory entry information for this file into the FCB. It performs all the necessary initialization. Since no links are created between the disk file and the FCB, the same file may be opened for input by several users. No parameters are passed or returned by the call. A and C registers are used.

*** OPENO *** (17)

Open for output system call prepares a file specified in the FCB passed in the DE-register for write access. OPENO provides all the necessary initialization of the FCB for outputting of bytes to the file. A directory entry for this file is reserved on the disk. OPENO reports "ALREADY EXISTS" if file name is not unique. Any file opened for output must be closed, otherwise bytes still in the file disk I/O buffer are lost.

*** CLOSE *** (18)

Close file system call performs the necessary housekeeping required to terminate I/O with a file. Closing a file open for output writes out remaining bytes in the disk I/O buffer and outputs the current FAM block to the disk. The directory entry for this file is updated on the disk to contain the final byte count of the file. The FCB address is passed in the DE-register. No parameters are returned.

*** REWIND *** (20)

Rewind system call resets file pointers in the FCB to the beginning of a file open for input. The FCB address is passed in the DE-register.

This call allows making another pass through the file without having to close it and then re-open it. It is used by the Assembler at the end of pass 1 to re-initialize for pass 2.

*** RDSECT *** (21)

Read a sector system call reads a physical sector from the disk into a buffer pointed to by the HL-register. This buffer must be 128 bytes in length. It is necessary to specify an FCB since the disk I/O drivers fetch the drive specification from the current FCB. Both default FCBs default to drive 0. If drive 1 is desired, use the STORDR system call to set the correct drive specification. The HL-register is returned pointing to the first byte following the 128 byte buffer. No other parameters are returned.

*** WRSECT *** (22)

Write a sector system call writes a physical sector from a buffer pointed to by the HL-register to the disk. See RDSECT above for further details.

*** INTFCB *** (23)

Initialize an FCB system call clears the buffer and initializes the FCB buffer pointer (FCBUF) and FAM block buffer pointer (FAMPTR). The FCB address is passed in the DE-register. The address of two contiguous 128 byte buffers is passed in the HL-register. This call can be used by the user to create extended FCBs.

The following system calls are found in the module SCALL3:

*** RDBYTE *** (24)

Read a byte system call returns the next byte in a file to the user in the A-register. The system handles reading of the FAM blocks and data records to the disk I/O buffer in a manner invisible to the user.

The end-of-file mark (EOF=1AH) is returned to indicate no more bytes to be read from this file. The FCB address is passed in the DE-register. All registers are maintained.

*** WRBYTE *** (25)

Write a byte system call writes a byte passed in the A-register to to the file specified in the FCB passed in the DE-register. The call handles all creation of FAM blocks and writing of the disk I/O buffers at the proper time in a manner invisible to the user. An end-of-file mark (EOF=1AH) should be written to all ASCII files before closing. All registers are maintained.

*** RDLINE *** (26)

Read a line system call reads a line terminated by CR or EOF to a buffer from a disk file specified by an FCB passed in the DE-register. The HL-register points to a user supplied line buffer of the following format: The length of the buffer is passed in the first byte of the buffer. The actual number of bytes returned is placed in the second byte of the buffer by the system call. If buffer overflow occurred, the call returns 0FFH in the second byte. The HL-register is returned pointing at the actual byte count (second byte) of the buffer. An error is reported if an EOF was encountered, but was not the first character in the line (this would constitute an incomplete line in the file). The average length of a line buffer reserved should be 84. This would allow 2 bytes for the buffer length and bytes transfered, 80 bytes for the maximum CRT line length and 2 bytes for a cr-lf. This system call is useful in reading sources that are to be processed a line at a time, e.g., an assembler.

*** WRTFIL *** (27)

Write a file system call writes out a buffer terminated by an EOF mark to a file specified by an FCB passed in the DE-register. Mainly used to output entire ASCII buffers such as those used by Text Editors.

*** LOADHX *** (28)

Load hex file system call loads an Intel standard hexadecimal format file from the disk into memory. Control is transferred to the loaded program at the starting address specified in the end-of-file record. If debug flag is set, a breakpoint is set at the starting address and the debug monitor is entered. If the file is of the ZDOS binary load module (LDM) format, the binary loader is called (see the section on the LDM format). Both loaders perform checksum comparisons in order to assure successful file transfers. Also checks are made to insure that ZDOS reserved ram is not overloaded by the file. There is no return to ZDOS from this call.

*** SAVHEX *** (29)

Save a hex record system call saves a memory segment as a hexadecimal record in a file specified by the FCB address passed in the DE-register. The HL-register points to a segment data block. The first word of the data block is the load address of the file. The second word is the starting address of the memory segment to be saved and the third word points to the end of the memory segment. The system call handles all conversion of bytes to ASCII and checksums for the hex records. In this manner, an assembler can create a binary image of a program segment in memory and save it as a hex file without resident conversion routines. The file has to be currently open for output. For more information, see the section on hexadecimal file formats.

*** EOFHEX *** (30)

End of hex record system call writes an end-of-file data record to a hexadecimal file specified by the FCB passed in the DE-register. The HL-register contains the starting address of the file. If there is no starting address, HL=0. An end-of-file mark (1AH) is appended to the end of the file since the hex file is basically an ASCII file. The file has to be currently open for output.

The following system calls are found in the module SCALL4:

*** FTCHAT *** (32)

Fetch attributes system call returns the file type attributes in the A-register from the FCB passed in the DE-register. The file types are: 00=ASCII, 01=BINARY, 10=HEX, 11=SYSTEM. All registers are maintained.

*** STORAT *** (33)

Store attributes system call sets the file type attributes in the FCB passed in the DE-register. The new attributes are passed in the A-register. Only bits 0 and 1 are stored according to the file types specified in the previous system call. All registers are maintained.

*** FTCHDR *** (34)

Fetch the drive specification system call returns the disk drive specification associated with the FCB passed in the DE-register. The drive is returned in the A-register. The legality of the drive specification is not checked by this routine. All registers are maintained.

*** STORDR *** (35)

Store the drive specification system call sets the drive specification passed in the A-register in the FCB passed in the DE-

register. The legality of the drive specification is not checked by this routine. All registers are maintained.

*** SETPRT *** (36)

Set protection system call sets the write protect bit in the directory entry associated with the file specified by the FCB passed in the DE-register. A and C registers are used.

*** UNPROT *** (37)

Unprotect system call clears the write protect bit in the directory entry associated with the file specified by the FCB passed in the DE-register. A and C registers are used.

*** CPARID *** (38)

Compare ID system call compares the user identification port value with the user ID stored in the FCB specified by the DE-register. User ID=0 and user ID=0FFH always match with any ID (for further explanation, see section on user IDs). If a match occurs, the Z-flag is a one.

*** STORID *** (39)

Store ID system call stores the user identification port value in the FCB specified by the DE-register. A and C register are used.

The following system calls are found in the module SCALL5:

*** GETDSK *** (40)

Get disk system call returns the currently logged-in disk drive specification in the A-register. No parameters are passed. No other registers are used.

*** SETDSK *** (41)

Set disk system call stores the disk drive specification passed in the A-register into the current logged-in disk location. No other registers or parameters are used. Changing the current logged-in disk

is normally a system function provided by the LOG intrinsic command.

*** GETCUR *** (42)

Get current pointer system call returns the current command string buffer pointer in the HL-register. This call can be used by programs to fetch the remainder of a command string for processing. See the section on the command string interpreter for more information.

*** GETTOP *** (43)

Get top of ram system call returns the address of the last available byte of ram computed by the system at initialization in the HL-register.

*** REPORT *** (44)

Report errors system call allows the user to utilize the system error handler to report errors to the console. The error number is passed in the B-register. There is no return to the user program. The system closes out all open default FCBs. All error numbers 0 to 127 are reserved for system use. User defined error numbers are in the range from 128 to 254. Error number 255 is reserved by the system to report a fatal hardware error. The error handler prints the error number and the value of the program counter (PC) where the error occurred. Fatal system errors (greater than 50) are reported as such. Certain error numbers cause the system to print the filename associated with an FCB in addition to a descriptive error message. For more information, see the section on system error handling.

*** GETUSR *** (45)

Get user number system call converts the user identification ID port value to a BCD number (0 to 9) and returns it in the A and C register.

*** DVCHEK *** (48)

Device check system call checks the command string pointed to by the HL-register with the list of current legal system devices. The string must contain a legal two character device name enclosed in colons. The call is entered with HL pointing at the first colon and exits with HL pointing at the first location past the second colon. If the device is not legal, the system reports:

ILLEGAL DEVICE

and does a system exit to the DOS base level. The current list of legal devices is:

```
:D0:    disk drive 0
:D1:    disk drive 1
:D2:    disk drive 2
:C0:    console device
:LP:    line printer
```

No registers are maintained by this call.

The following system calls are found in the module SCALL6:

*** PARSER *** (50)

Command string parser system call parsers a command string in the system input buffer into input and output file specifications. The command string specification is as follows:

```
<:input device:>filename<.extension><,file list...>< TO >
<:output device:><filename><.extension><parameters>cr-lf
```

All specifications enclosed in <> are optional. No parameters are passed to the call. The call returns the HL-register pointing at the input file name in the command string and DE-register pointing at the output file name in the string. DE=0 if there was no output file name. The current command buffer pointer is returned pointing at the parameter list and can be accessed using the GETCUR system call. All registers

are used. See the section on the command string interpreter for further information.

*** NEXTIN *** (51)

Next input file system call loads FCB 1 with the next input file specification from the command string buffer. If there are no more input files, the carry flag is returned set. NEXTIN must not be called until after PARSER has scanned the command string first. All registers are used. NOTE: PARSER and NEXTIN were not initially intended for general use by all users, but more advanced programmers with a better understanding of the system could make good use of these two calls and the GETCMD routine to process their own command strings.

This is the end of the list of currently available system calls. Gaps in the number sequence were left intentionally for future expansion. Existing call numbers are fixed and will not be changed to insure proper operation with user programs.

System Error Handling

There are two types of errors reported by the system to the user; fatal and non-fatal. Non-fatal errors are mostly informational warnings, whereas fatal errors result in immediate cessation of the current process. Error numbers 0 through 49 decimal are reserved for non-fatal errors. Error numbers 50 through 127 are reserved for fatal system errors. Error numbers 128 through 254 are reserved for user defined errors. Error number 255 is used by the system to report a probable system hardware failure. Error 255 is bad news. A list of the current reserved list of system errors is given at the end of this section. For information on the mechanism for reporting errors to the

system, refer to the REPORT system call (44) in the chapter on system calls.

A fatal error is reported to the console as:

```
FATAL ERROR xxx PC=yyyy
```

where xxx is the decimal error number in the range 50 to 127 and yyyy is the hexadecimal user program counter pointing to the CALL ERROR instruction in the user program. If the error occurred in a command that was being executed from a submit command file, the batch mode flag is cleared and the submit file is closed in FCB 3. The system exits to the DOS base level through the EXIT system call in order to close out any default FCBs.

Non-fatal errors 0,1 and 2 report a file name associated with the current active FCB. Errors 3 through 8 report simple warnings.

```
Error 0      filespec, NO SUCH FILE
Error 1      filespec, ALREADY EXISTS
Error 2      filespec, WRITE PROTECTED
Error 3      SYNTAX ERROR
Error 4      ILLEGAL DEVICE
Error 5      SYSTEM FILE
Error 6      ACCESS ERROR
Error 7      DIRECTORY FULL
Error 8      LOAD ERROR
```

Error 0 (no such file) occurs when the file specified in the FCB does not exist on the disk drive specified. Error 1 (already exists) occurs when the file specified in the FCB matches a name that already resides on the disk drive specified. Error 2 (write protected) occurs when an attempt is made to delete or rename a write protected file. Error 3 (syntax error) occurs when an illegal character or file name is inputted to the command string interpreter. Error 4 (illegal device) occurs when a device is specified in a command string that does not match the current list of legal system devices. Error 5 (system file) occurs when

any attempt is made access or use the reserved system file names. Error 6 (access error) occurs when an illegal access is made to a file, e.g., attempting to print a non-ASCII file. Error 7 (directory full) occurs when a request is made for a directory entry after all 192 entries a occupied. Error 8 (load error) occurs when a user's program attempts to load over ZDOS reserved RAM locations 0 through 5FFH. The following is a list of currently defined fatal errors reported by the system:

Error #	description
50	Attempt to read a null file.
51	Attempt to close an inactive FCB.
52	Attempt to open an active FCB.
53	not used
54	Attempt to read an inactive FCB.
55	Attempt to write to an inactive FCB.
56	Missing EOF mark in file.
57	EOF not first character in line buffer.
58	Attempt to rewind an inactive FCB.
59	Attempt to rewind a file open for output.
60	Attempt to write a hex segment to an inactive FCB.
61	Attempt to write a hex segment to a file open for input.
62	LDM track > 76.
63	Hex segment start address less than stop address.
101	Bad checksum in binary file.
102	Bad LDM checksum.
255	Fatal hardware error.

CHAPTER VI

MULTI-USER IMPLEMENTATION

System Initialization

System initialization is accomplished at two levels, namely system and user. System initialization begins by clearing the Process Status Byte (PSB) which unlocks all system resources. Next the contents of the vector address used by interrupt mode 2 and the CSB is initialized to point at the context switching subroutine. The number of users switch is read to check for its proper setting. If the setting is illegal, the following is reported to user 1's console:

ILLEGAL MAX USER

The processor halts until the setting is corrected and the system is rebooted. At this point, the user level of initialization begins. First the user's console interface is initialized in case the subsequent test fails. The user ram is then checked to see if it is present in the system. If not present, the system reports to user 1's console:

USER x HAS NO MEMORY

where x is the user number (1 to 8). The processor halts until the system configuration error has been corrected.

The stack pointer is set and the cold start subroutine is called. This routine computes the top of the user's memory bank and saves it in TOPRAM. The default list device (CRT) is set in IOBYT. The current logged-in disk is set to drive 0. The batch mode flag is reset. The

user's copy of the PSB byte is cleared. Finally, the ZDOS boot message is sent to the console. At this time, the DOS base level re-entry address is pushed on the user's stack five times, in order to simulate previous saved status. The current value of the stack after the pushes is stored in SAVSTK. The reason for this is that after the last user has been initialized, the interrupts are going to be enabled, at which time context switches are going to start to occur. It is therefore necessary to have some previous state to which the virtual processor can return. Since the DOS base level address was pushed on the stack as the PC, the processor will return to the base level. If a previous state was not simulated, a stack underflow would have occurred and the state returned to would have been undefined.

Now that a single user has been initialized, the next user must be switched to an active state. The user number port is read, the bit position rotated to the left and returned to the user port. This bit pattern is also outputted to the bank switching port. This enables the next user's ram bank. This user number is compared to the previously saved number of users value to determine if all users are initialized. If not, the user initialization process repeats as above until done.

At this point, all users have been initialized. User 1 is switched to an active state. The I-register in the processor is set to the high byte of the vector address. The interrupt mode of the processor is set to IM 2. Now the processor interrupts are enabled with an EI instruction. In case there is not an interrupt pending, the program jumps to the DOS base level. The ZDOS multi- user mode is now active and under complete control of the context switching hardware.

Context Switching Subroutine

When the CSB generates an interrupt, interrupt mode 2 (IM 2) of the Z80 produces a vector address to the context switching routine. The function of this routine is to save the status of the current user, switch to the next active user, and restore the status of the user that was saved previously. Since this subroutine is an essential part of the system software, additional comments are appropriate at this time. The routine begins with the following code:

```
SWITCH: PUSH PSW        ;Save registers
        PUSH B
        PUSH D
        PUSH H
        LXI H,0
        DAD SP
        SHLD SAVSTK     ;Save current stack pointer
```

At this point, the "virtual processor" has been saved in the current memory bank. In the large mainframes or minicomputer machines, all the virtual processors normally reside in a single memory space. In the ZDOS system, each user or "virtual processor" reside in their own physical memory board (or bank). This method adds some inherent memory protection in that if a user's program loses control, the only virtual processor he will likely destroy is his own.

The system must now determine the next user to restore. This routine is considered as part of the software overhead necessary to implement a multi- user system. In other words, this routine has a direct bearing on the system response time. In order to reduce this overhead, a constraint in the method of adding additional users was formulated; i.e., all user memory banks are contiguous, starting with bank 0, and any additional user must occupy the next sequential memory bank. Through the use of the number of users switch and the specially

implemented maximum user port on the CSB, the number of users can be changed at boot time by adding the appropriate number of RAM boards and setting the current number of users switch accordingly.

The next user is accessed as follows:

```
IN USERN
RLC
OUT USERN       ;Rotate bit in port 0EEH
OUT BANK        ;Switch to next bank.
```

At this point, we may or may not have switched to the next active bank of memory. If the next bank or user has not been implemented, user #1 (bank 0) must be again selected. The maximum number of users port at 0EFH is read and compared with the current user port at 0EEH. If MAXUSR=0, then all 8 users are on line.

```
IN MAXUSR
MOV C,A
ORA A
JZ OK
```

If MAXUSR > USERN, the limit of current users has not been reached.

```
IN USERN
CMP C
JC OK
```

Otherwise, wrap around to user #1.

```
MVI A,1
OUT USERN
OUT BANK
```

The switch routine checks the USART for a framing error on every pass. A framing error occurs whenever the BREAK key on the console is pressed. Control is then transferred to a routine that re-initializes the current user. In this manner, a user can recover from a catastrophic program crash or infinite loop without having to reboot the whole system.

```
IN STAT
ANI FRMERR
JNZ RBUSER
```

The previously stored status of the current user is now restored to the CPU, interrupts are enabled and a return to the current program counter location is performed.

```
LHLD SAVSTK
SPHL
POP H
POP D
POP B
POP PSW
EI
RET
```

The average time necessary to perform a context switch is approximately 120 usecs.

Interlock System

A very important aspect of the ZDOS Multi-user System is the interlock system provided by the Process Status Byte. ZDOS does not maintain job queues or lists for scheduling each of the tasks requested by the users. In keeping with the constraint of a re-entrant, ROM-based operating system, it was necessary to devise another scheme for process scheduling. The PSB provides a rotational, non-prioritized sheduling method to meet this constraint. In order to explain more fully this scheme, the console input routine will be examined.

In a typical single-user, programmed I/O approach, a device driver would be coded as follows:

```
LABEL:  IN STATUS       ;input device status byte
        ANI RDY         ;check device ready status bit
        JZ LABEL        ;loop back if not ready
        IN DATA         ;device now ready, input data byte
        RET             ;return to calling program
```

In a single-user environment, the system can afford to wait forever until the device is ready with a new byte of information. In the multi-user mode, this waiting slows system response. If enough users are on line, this may even cause input characters to be lost. If, for example, ZDOS waited the full 33 millisecond time slice for console output flags and there were eight users on line, each user would see approximately four character per second output rate, which about two and one-half times slower than a teletype.

All the device drivers in ZDOS utilize the PSB and the context switching hardware on the time slice clock to shorten the overall time intervals by many orders of magnitude. The console input routine is coded as follows:

```
CI:     IN STAT         ;input console status byte
        ANI DAV         ;is a data byte available for input
        JNZ CIRDY       ;if there is, jump to input code.
        OUT CONTXT      ;reset time slice clock, and force a context
                        ;switch to the next user.
        JMP CI          ;when the system rotates back around to this
                        ;user, this will be the point of entry.  The
                        ;flag will be checked again.

CIRDY:  IN DATA         ;data is available. input the data byte.
        ANI 7FH         ;strip parity bit.
        RET             ;return to calling program.
```

The above example shows how the context switching hardware is used with a device driver that can be interrupted at any point in the routine without any loss of data or any conflict with any other device. Each user console has its own dedicated hardware interface. This method will not hold true for shared resources such as a disk or line printer.

With shared resources, it is necessary to use the PSB to lock out the other users during critical access and to use the enable and disable interrupt instructions during the setting and resetting of the interlocks. Failure to do so at the proper time and in the proper

sequence results in system hangups and crashed disks. As an example of a driver for such a shared resource, the DSEEK (disk seek) routine is coded as follows:

```
DSEEK:                  ;check bit 0 of PSB to see if disk is busy.
SWAIT:  DI              ;disable further interrupts until this routine
                        ;is finished checking the DIP bit (disk I/O in
                        ;progress).
        IN PSB          ;fetch Process Status Byte
        ANI DIP         ;test disk busy bit.
        JZ NBUSY        ;if not busy, go seize control for this user.
        OUT CONTXT      ;disk is busy with another user. force a
                        ;context switch to the next user.
        EI              ;re-enable the interrupts so that the context
                        ;switch can occur.
        JMP SWAIT       ;when the system rotates around to this user
                        ;again, this will be the entry point.

NBUSY:  IN PSB          ;disk is not busy. set it so now.
        ORI DIP         ;set disk I/O in progress bit.
        OUT PSB         ;restore Process Status Byte.
```

The code from this point on is the disk seek hardware driver. At this point, the steps taken depend on the type of disk hardware available on the system. If the disk is DMA or interrupt driven, one could force a context switch and enable the interrupts to further improve system throughput. Since the current ZDOS implementation runs on a Cromemco system that has a programmed I/O type disk controller, the interrupts are not re-enabled until the disk read or write has been completed. At this time, the DIP bit in the PSB is cleared, a context switch is forced on the system and the interrupts re-enabled. This prevents a single user from seizing control of the disk for multiple sector transfers, which could cause another user to lose console input data. As mentioned in the hardware section, a DMA disk controller would greatly enhance the throughput and data transfers of the system.

Load Module File Format

In some of the earlier configurations of ZDOS, the Assembler and Editor were linked with the system as resident intrinsic commands. Although this had the advantage of very fast response to assembly and editing requests, it had the disadvantages of overflowing the 16K resident RAM constraint and required the system to be regenerated each time an update was made to the assembler or editor. It was decided to remove the assembler and editor to disk as extrinsic command files. Since the format of these files were hexadecimal, they proved to be very slow in loading, even in the single-user mode. On the MDS-800, the assembler took 19 seconds to load and the editor took 11 seconds. This was a nuisance in the single-user mode, but a disaster in the multi-user mode.

The solution involved the use of core image files. Core image files, located in contiguous sectors on the disk load extremely fast. For example, the same assembler loaded from an MDS-800 disk takes 1.5 seconds and the editor takes less than a second to load. It was therefore decided to create a new file type called an LDM (Load Module).

The format of the LDM file is as follows; the file resides on the disk as a block of contiguous sectors, the first of which contains information necessary to load, check the success of the load and transfer control to the program. This first sector is called the header block. The header blocks contains the load address where the subsequent sectors are to be read and stored, the number of sequential sectors to be read, the starting address of the loaded program, and a 16-bit checksum to check the results of the transfer. The sectors read from the LDM file are written directly into memory, instead of being read to

a disk I/O buffer first. The FAMSEC and FAMTRK in the directory entry point to the track and sector where the header block resides on the disk.

Since the LDM format file requires a contiguous block of sectors for storage, the normal system SAMSCH subroutine could not be used to allocate sectors for the file. It was therefore necessary to write a special LDM converter program. This LDM converter program converts a standard INTEL hexadecimal format file to a ZDOS LDM format file. It also searches the ZDOS disk for a contiquous block of sectors for storage. The current implementation of this converter program reads the hexadecimal file from an ISIS disk and outputs the LDM file to a ZDOS disk in drive 1. This has the advantage that any executable object format file that runs on the MDS-800, can be made to run on the ZDOS hardware by simply converting the .OBJ file to a .HEX file using OBJHEX and then pass the .HEX file to the LDM converter.

The main use of the LDM converter is to create system programs on the disk at system generation time. If desired, the current LDM converter program could be modified to convert ZDOS hex files to LDM files, but in general, user hex files are short and load fast enough.

Because the LDM file format does not conform the usual system file format, it was necessary to make additions to the DELETE and LOADHX system calls to handle LDMs. Also, LDM files cannot be listed on the console or copied from one disk to another (except possibly in a wholesale disk fast copy routine).

Since the LDM file resides on the disk as contiguous, sequential sectors, it was considered allowing the LDM loader to seize control of the disk for the duration of the load in the multi-user environment.

This idea was discarded for the following reasons: 1) The nature of the current disk I/O drivers only allow a user to read one sector at a time. 2) The disk hardware is not DMA, which would prevent the system from responding to console I/O during this seizure.

Due to the fact that only one sector is read at a time, when more than one user is loading an LDM file, there is some increase in load time due to rotational latency of the disk. If the LDM files reside on different tracks, the disk is activity is frantic due to the alternate seeks from one track to the next and back again. In order to get around the problem of not having DMA disk hardware, a future version the CRT Multiplexer board should include an on-board processor whose sole function would be to collect user command strings for transfer to the system only when a cr-lf is entered.

CHAPTER VII

SUMMARY AND CONCLUSIONS

The ZDOS Multi-User Operating System was designed and written to fill the need for a relatively inexpensive, dedicated training aid for the teaching of microprocessor programming fundamentals to electrical engineering students. This project was not an attempt to build on traditional multi-user approaches, but rather to devise a concept tailored to a specific microcomputer hardware environment. What has evolved is a very viable operating system that can be used as a host operating system or can be linked with user programs to give even the simplest task full disk file handling capabilities.

ZDOS utilizes a time-multiplexing technique to achieve apparent concurrency of programs on the processor. By the use of a specially implemented blocking technique, the level of actual concurrency has been increased in the I/O hardware. This is accomplished by the use of the context switch request function on the CSB. The same blocking scheme prevents conflicting requests for use of shared resources, such as the disk.

Another diversion from traditional time share systems, is in the implementation of the console interface. Most systems utilize separate I/O hardware drivers for each user console, thus requiring separate software driver modules. ZDOS uses the user identification number in conjunction with a hardware gating method, in order to address all

consoles at the same I/O address. This allows the use of a single software driver for all consoles. This eliminates the need of a scheduling method for access to console drivers.

Traditional systems have need of a memory manager to dynamically allocate the system memory resource. This method was departed from by the use of bank switching capabilities of the memory board hardware. Since each user addressable memory space is in a separate bank, the amount of potentially accessable memory by each user is increased. That is, eight users in a single 64K memory space can potentially have less than 8K each, but eight users in a bank switched system can potentially have 64K (assumming of course the system overhead is zero). Since the bank switching is a function performed by the hardware, the need for a software memory manager has been eliminated. Additionally, by the use of this method of memory allocation, some inherent memory protection is afforded. Since each user occupies a separate memory space, an out of control process cannot destroy the sanity of other users.

Additional memory protection is provided if the system software is placed in ROM. This is possible because of the re-entrant nature of the system software. Since the hardware provides many of the scheduling functions that the processor normally handles in traditional approaches, ZDOS does not require system RAM. There are no system tables, queues or stacks to maintain. Each user's virtual processor is saved on their own stack, resident in their own memory space.

In the course of this project, one concept has become readily apparent; the software design process must become more disciplined. In the past two and one-half years, thousands of hours have been spent at a hit and miss approach by an engineer to a computer science problem. As

the project progressed, new approaches to software design were tested and discarded for still better methods. As this particular design project draws to completion, it is hoped that the thousands of hours spent in a hit and miss approach to software design by an engineer has been replaced by a more disciplined structured approach of a computer scientist.

ZDOS is currently running on three different target hardware systems in both single and multi-user modes. It is the basic operating system for two different testing systems at Western Electric. Although the basic kernel of the system is fairly finalized, extrinsic programs are being constantly created by a small group of users. Utility programs are available for interchange of data files between ZDOS and ISIS-II. There is in the works, a version of ZDOS that will run CP/M based programs such as CBASIC, SID, TEX and RMAC. The main advantage to a system such as ZDOS is that if the user does not care for a feature or wants to add features, the source listing is available for such changes and a custom version suited to his needs can be generated.

SELECTED BIBLIOGRAPHY

(1) Cromemco, Incorporated. CDOS User's Manual. Mountain View, California, 1977.

(2) Cromemco, Incorporated. RDOS Instruction Manual. Mountain View, California, 1978.

(3) Intel Corporation. ISIS-II User's Guide. Santa Clara, California, 1976.

(4) Intel Corporation. 8080/8085 Assembly Language Programming Manual. Santa Clara, California, 1977.

(5) Intel Corporation. 8080/8085 Macro Assembler Operator's Manual. Santa Clara, California, 1977.

(6) Texas Instruments Incorporated. The TTL Data Book for Design Engineers. Dallas, Texas, 1973.

(7) Zilog Incorporated. Z80 CPU Technical Manual. Cupertino, California, 1977.

APPENDIX A

SYSTEM CALLS

The following is an abbreviated list of system calls:

No.	Name	Description
0	EXIT	Exit to ZDOS base level
1	CNSIN	Return character from console in A-register
2	CNSOUT	Output character passed in A to console.
3	CNSTAT	Return console status. A=0 if not ready.
4	LSTOUT	List character passed in A on list device in IOBYT.
5	IOSET	Set IOBYT to list device passed in B. CRT=40H, LPT=80H.
6	IOGET	Return IOBYT in A-register.
7	LISTHL	List string pointed to by HL on list device in IOBYT.
8	BINASC	List binary number in HL on list device in B.
9		
10	DNSRCH	Search directory for file in FCB.
11	PRTNAM	Print file specification in FCB on console.
12	RENAM	Rename old name in FCB to new name in HL.
13	DELETE	Delete file name specified in FCB.
14	MOVNAM	Append name pointed to by HL to FCB.
15	RENEXT	Rename extension in FCB to new extension in HL.
16	OPENI	Open file in FCB for input.
17	OPENO	Open file in FCB for output.
18	CLOSE	Close file in FCB.
19		
20	REWIND	Rewind file in FCB to its head.
21	RDSECT	Read a physical sector from the disk.
22	WRSECT	Write a physical sector to the disk.
23	INTFCB	Initialize an FCB. Buffer address passed in HL.
24	RDBYTE	Read a byte from an open file to the A-register.
25	WRBYTE	Write a byte from the A-register to an output file.
26	RDLINE	Read a line from an open file to buffer in HL.
27	WRTFIL	Write a buffer terminated by an EOF to a disk file.
28	LOADHX	Load an executable file into memory.
29	SAVHEX	Save a binary memory segment as a hexadecimal file.
30	EOFHEX	Write an end of file record to a hex file.
31		
32	FTCHAT	Return file attributes in A-register.
33	STORAT	Store file attributes passed in A in FCB.
34	FTCHDR	Return drive specification from FCB in A.
35	STORDR	Store drive specification in A into FCB.
36	SETPRT	Set write protect attribute in FCB.
37	UNPROT	Reset write protect attribute in FCB.

```
38  CPARID      Compare user ID in FCB with user port.
39  STORID      Store user ID in FCB.
40  GETDSK      Return current logged-in disk in A-register.
41  SETDSK      Set current logged-in disk passed in A-register.
42  GETCUR      Return current command string buffer pointer in HL.
43  GETTOP      Return address of last RAM location at top of memory.
44  REPORT      Report error number passed in B-register to console.
45  GETUSR      Return current user ID number in A and C.
46
47
48  DEVCHK      Check legality of device pointed to by HL.
49
50  PARSER      Parse incoming command string.
51  NEXTIN      Fetch next input file in list into FCB.
```

APPENDIX B

INTRINSIC COMMANDS

The following is an abbreviated list of intrinsic commands:

Command	Description
ERA file1,...,file2	Erase file list from disk.
FREE <:Dx:> TO <:list:>	Display number of free blocks on list device.
ATR file WO/W1	Set or reset write protect flag on file.
REN oldfile TO newfile	Rename old file name to new file name.
TYPE file TO <:list:>	Type file on list device.
LOG :Dx:	Log in on device specified.
BYE	Exit to the system monitor.
DEBUG file	Load file, set debug mode, and enter monitor.
GET file	Load file into memory and return to ZDOS.
DIR <:Dx:> TO <:list:>	Print directory on list device.
SUB file<.JOB>	Execute indirect command file.
DISK	Disk examination utility monitor.
COPY file1,... TO file	Copy input file list to output file.
WHO	Report current user ID number to console.
CID file oldID newID	Change file user identification number.
SLIST	Super user directory listing command.

APPENDIX C

SYSTEM ERROR MESSAGES

The following is an abbreviated list of system error messages:

No.	Description
0	filespec, NO SUCH FILE
1	filespec, ALREADY EXISTS
2	filespec, WRITE PROTECTED
3	SYNTAX ERROR
4	ILLEGAL DEVICE
5	SYSTEM FILE
6	ACCESS ERROR
7	DIRECTORY FULL
8	LOAD ERROR
50	Attempt to read a null file.
51	Attempt to close an inactive FCB.
52	Attempt to open an active FCB.
53	
54	Attempt to read an inactive FCB.
55	Attempt to write to an inactive FCB.
56	Missing EOF mark in a file.
57	EOF not first character in line buffer.
58	Attempt to rewind an inactive FCB.
59	Attempt to rewind a file open for output.
60	Attempt to write a hex segment to an inactive FCB.
61	Attempt to write a hex segment to a file open for input.
62	LDM track greater than 76.
63	Hex segment start address less than stop address.
101	Bad checksum in a binary file.
102	Bad LDM checksum.
255	Fatal hardware error.

APPENDIX D

HEXADECIMAL FILE FORMAT

The system object code is stored on the disk in the INTEL standard hexadecimal paper tape format. The code is blocked into records, each of which contains the record type, length, type, memory load address, and checksum in addition to the data. Each record is stored as ASCII and is terminated by a carriage-return line feed. The record mark is a colon(3AH) and is used to signal the start of a record. The record length is the count of the data bytes in the record. A record length of zero indicates end-of-file. The load address specifies the address at which the first data byte will be loaded. The successive data bytes will be stored in successive memory locations. The record type specifies the type of this record. All data records are type 0. End-of-file records can be type 0 or 1. The data consists of two ASCII characters per memory byte. The data is represented by hexadecimal values 00H through FFH. The checksum is the negative of the sum of all 8-bit bytes in the record, beginning with the record length and ending with the last data byte, evaluated modulo 256. The sum of all bytes in the record (including the checksum) should be zero.

APPENDIX E

ZDOS TEXT EDITOR

Introduction

The ZDOS Text Editor enables a user to create and edit ASCII text files. The Text Editor can be used to manipulate and edit text on a line or character basis. One or more characters can be inserted in, deleted from, or changed in a line of text. Insertions and deletions can be made that cover more than one line of text. The point of insertion or deletion can be freely selected to be at the beginning of text, the end of text, the beginning of a line, the end of a line, or at any point within the text. Line numbers and other extraneous information need not be added to the text in order for the text editor to operate correctly.

Overall Flow of Text Editting

The usual procedure for creating a new text file is to call the text editor, enter the text from the system console, perform whatever editting functions are desired, and then output the text file to a disk for storage. To edit an existing file, call the text editor, input the text from the file, edit the text, and output the editted version to a disk file.

Memory Requirements, Work Space, and Text Buffer.

The ZDOS Text Editor is loaded from the disk into memory starting at 800H. In the current version of EDIT, the text buffer starts at 1B27H. The remaining RAM space to 3FFFH is available as buffer and work space.

The text buffer is the portion of the RAM currently being used by the text editor to store text. The size of the text buffer varies, increasing as text is entered, and decreasing as text is deleted. When the buffer is empty, beginning and end of the buffer coincide.

Buffer Pointer.

The buffer pointer is used to locate the position in the text buffer where editting is to occur. The buffer pointer can be positioned as follows:

1. Before the first character in the text buffer (beginning of the buffer).

2. Immediately following the last character in the text buffer (end of the buffer).

3. Between two adjacent characters in the text buffer.

The buffer pointer is never positioned directly on a particular character but points before it or after it. Text is placed into the buffer at a point immediately preceding the buffer pointer.

You can move the buffer pointer to any position inside the buffer. Any command attempting to move the buffer pointer past the boundaries of the text buffer is terminated when the buffer pointer reaches the boundary, even though the command specified has not been completed.

The buffer pointer can be moved by characters or by lines. A line of text in the text buffer is a string of characters having a line feed as its last character. The next character in the text buffer immediately following the line feed is in the next line. If no line feed characters are used in the text, the entire text is consisdered to be one line. Line feeds are automatically inserted following a carriage return whenever the carriage return key is depressed.

Calling the Editor

The ZDOS Text Editor is an extrinsic command invoked by the name EDIT. As opposed to the Intel ISIS software, the ZDOS Text Editor can be executed from a submit command file. The syntax of the EDIT command is:

```
EDIT filespec1 <TO filespec2>
```

where

filespec1 is either the name of a new file to be created and editted or the name of an existing file to be editted.

If filespec1 is a diskette file and filespec2 is not specified, the action taken by the editor depends on whether filespec1 is a new or existing file. If filespec1 is new, it is created for

output. If it exists and is not write-protected, it is opened for input. A special file, EDITOR.TMP, is created for output. When the editting session is ended with the EXIT command, the files are closed and the following occurs:

1. filespec1 is renamed to filespec1.BAK.

2. EDITOR.TMP is renamed to filespec1.

For example, after an editting session that started with the command

```
EDIT FILE.TXT
```

the files FILE.BAK and FILE.TXT are on drive 0. FILE.BAK is a backup version that represents the state of the text file before the editting session.

filespec2, if specified, is the name of the output file to which the results of the editting session are written. The action of the editor depends on whether filespec2 is a new file or an existing file. If it is a new file, the editor creates it for output. If filespec2 already exists, an error is reported to the console.

ZDOS will not allow filespec1 and filespec2 to be the same.

The filename EDITOR.TMP should be reserved for the editors' use as any file on the same disk as the input file called EDITOR.TMP will give a warning error. Note that when an editting session is aborted through means other than a valid command, the temporary output file, EDITOR.TMP, remains on the disk.

Example 1: Suppose you want to create and edit a new file called FILE1.ASC on drive 1. You would enter the following in response to the ZDOS command prompt D0>:

```
D0>EDIT :D1:FILE.ASC
```

The system responds with:

```
ZDOS TEXT EDITOR, Vx.x
NEW FILE
*
```

where x.x is the current version number of the editor. NEW FILE is stated confirming the creation of a new file. Finally, the text editor prompts with an asterisk (*).

Example 2: Suppose on some other occassion you want to edit an existing file on drive 1 called CRUD.SRC. You would enter the following:

```
D0>EDIT :D1:CRUD.SRC
```

The system responds with:

```
ZDOS TEXT EDITOR, Vx.x
*
```

The NEW FILE message is not given in this case because an existing file is being editted.

Editor Commands and Command Syntax

The text editor signals its readiness to accept commands by prompting with an asterisk (*).

Commands can be entered singley or in a command string. The commands in a command string are executed in the order they are entered. A command string is terminated with two successive ESC characters (or ALT MODE characters on some terminals). The ESC characters are displayed on the console as dollar signs ($). However, text strings within commands must be terminated by one ESC. A substitution command includes a new text string to be entered into a file in place of an old text string.

This command is replacing old text with new text, thus

```
Sold text$new text$$
```

Command strings are stored from the top of available RAM memory down. The first command character is stored in the highest available location, with succeeding characters in descending locations. If a command attempts to overwrite the text buffer, a bell sounds. Press the rubout key to delete the command you are entering. Then write out the buffer with the W command to free up space for further editting.

Carriage Return and Line Feed.

When you enter text, the text editor automatically generates a line feed following a carriage return. Thus, if you want to delete a carriage return, you must delete two characters to eliminate the line feed also.

Deleting Typographic Errors (RUBOUT and Control-X).

Any typographic errors in entering a command can be removed by pressing the RUBOUT key once for each character to be removed. As each character starting from the last one entered is deleted, the editor echoes the deleted character on the console. Entering a Control-X causes characters to be deleted from the command back to the last carriage return and line feed.

Inserting Tabs (Control-I).

The editor generates a horizontal tab when you enter a Control-I (or TAB key on some terminals). A Control-I is stored in the text buffer as single character (09H). The system accepts this character to generate a sufficient number of spaces to position the next character at the next tab position. Tab stops are located every eight character positions accross a line of text (0, 8, 16, 24, etc.).

Command Descriptions

Two Basic Commands - Insert and Type.

The insert and type commands allow you to insert text in the buffer and to type or echo the text back onto the console for your inspection.

I - Insert Text in Buffer.

The insert (I) command is used to insert text in the text buffer from the system console. The new text is entered immediately before the buffer pointer. The syntax of the I command is:

```
Itext$$
```

where text can include carriage returns. After recognizing the letter it encounters an ESC or ALT MODE.

Example:

```
*Ito be or not to be,<CR>
that is the question.<CR>
$$
```

In this example, two lines are entered before the buffer pointer.

T - Type Text in Buffer on Console.

To type back the entered text in order to verify it was entered correctly, use the type (T) command. The syntax of the T command is:

```
nT
```

where n is a decimal integer ranging from

```
-9999 to +9999
```

if n is positive, the text is typed from the current position of the buffer pointer forward to the nth line feed. If n is negative, typing begins from n lines before the beginning of the current line (the line containing the buffer pointer) and continues until the buffer pointer is reached. If n is zero, typing is from the beginning of the current line up to the buffer pointer. If no value is specified for n, the editor assumes a default value of 1, which causes typing from the buffer pointer to the end of the current line.

Buffer Pointer Positioning.

The following four text editor commands, B, Z, L, and C, all move the buffer pointer. B moves the pointer to the beginning of the text buffer. Z moves the pointer to the end of the text buffer. L moves the pointer a specified number of lines forward or backward. C moves the buffer pointer a specified number of characters forward or backward.

B - Move Pointer to Beginning of Buffer.

The B command, which moves the buffer pointer to the beginning of the text buffer, has several uses. For example:

Setting a reference point for counting lines of text or searching for a word or phrase.

Defining a starting point when thwhole text buffer contents are to be typed out.

Inserting text at the beginning of the text buffer, in front of text already in the buffer.

Z - Move Pointer to End of Buffer.

The Z command, which moves the buffer pointer to the end of the text buffer immediately following the last character in the buffer, is used primarily tprepare to append text to the end of the buffer.

L - Move Pointer to Next Line.

The L command moves the pointer a specified number of lines forward or backward. The syntax of the L command is:

nL

where n is a decimal number in the range

-9999 to +9999

A posive value of n advances the buffer pointer to the beginning of the nth line following the current line. A negative value of n moves the buffer pointer back to the beginning of the nth line preceding the current line. When the argument value is -1 or just

-, the buffer pointer is moved back to the beginning of the line preceding the current line. If n is zero, the buffer pointer is moved back to the beginning of the current line.

C - Move Pointer over One Character.

The C command moves the pointer a specified number of character positions. The syntax of the C command is:

nC

where n is a decimal number in the range

-9999 to +9999

If n is positive, the pointer is moved foward n characters. If n is negative, the pointer is moved back n characters. If n is omitted, a value of 1 is assumed. A value of zero produces an error.

The C command is not an efficient way to move the pointer over large blocks of text, but is best utilized when pointer movement is restricted to one line of text. When you need to move the pointer over lines or paragraphs, the L and F command should be used.

Editting Examples with I, T, B, Z, and L.

The following examples make use of five of the editting commands so far presented: I, T, B, Z, and L.

1. Suppose you have entered data into the text buffer using the I command, and you want to type out the entire buffer. The following command string can be used:

*B500T

The B command move the buffer pointer to the beginning of the text buffer. The 500T command types out the entire buffer if it contains 500 lines or less. Otherwise, the first 500 lines are typed.

2. You can use the following command to type the current line of text if the buffer pointer is not at the beginning of the line, without moving the buffer pointer from its current position:

*0TT$$

The 0T part of the command types from the beginning of the line up to the buffer pointer. The next T types from the buffer pointer to the end of the line.

3. The current line of text can also be typed by:

*0LT$$

The 0L part of the command moves the buffer pointer to the beginning of the current line. The T then types from the pointer to the end of the line.

4. You can move the pointer back 5 lines of text and have the line where the pointer is positioned typed out by the following command string:

```
*-5LT$$
```

The buffer pointer is moved back to the start of the fifth line before the current line. This new line becomes the current line. The T causes the line to be typed out.

Deletion of Text.

The K and D command are used for deleting text from the buffer.

K - Kill Lines of Text.

The K command deletes lines of text. The syntax of the K command is:

```
nK
```

where n is a decimal number in the range

```
-9999 to +9999
```

If n is positive, the text is deleted from the current position of the buffer pointer forward to and including the nth carriage return and line feed. If n is negative, deletion starts from n lines before the beginning of the current line (the line containing the buffer pointer) and continues until the buffer pointer is reached. If n is zero, deletion is from the beginning of the current line up to the buffer pointer. If no value is specified for n, the editor assumes a default value of 1, which causes deletion from the buffer pointer to the end of the current line.

D - Delete Characters from Buffer.

The D command is used to delete a specified number of characters from the text. The syntax of the D command is:

```
nD
```

where n is a decimal number in the range

```
-9999 to +9999
```

A positive value of n causes deletion of n characters following the buffer pointer. A negative value of n causes deletion of the n characters preceding the buffer pointer. If n is omitted, a value

of 1 is assumed. A command of 0D produces an error.

Editting Examples Using B, T, L, C, K, D

The following examples make use of the editting commands thus far described:

1. Suppose you want to delete ("kill") the first 20 lines of text.

```
*B20K5T$$
```

This command string causes the buffer pointer to be moved to the start of the buffer, 20 lines of text to be deleted, and the following five lines to be output to the system console device. The command terminator is placed at the end of the command string; the individual commands do not need separate terminators.

2. Consider the following line of text, where the word MULTUIPLY can be corrected by simply deleting the extraneous letter U.

```
;THIS ROUTINE WILL MULTUIPLY TWO 16-BIT NUMBERS.
```

Because the buffer pointer is at the beginning of the text line, it must be moved 23 character positions to the point immediately before the letter U. Then the letter U can be deleted with the D command. The command to perform these operations follows:

```
*23CD$$
```

This is a clumsy way to delete a character. It is included here as an example to show the use of the C command. A better command to perform the same operation is the S (substitute) command to be described later.

3. Suppose the following text is present in the text buffer:

```
THIS IS LINE 1
THIS IS LINE 2
THIS IS LINE 3
THIS IS LINE 4
THIS IS LINE 5
THIS IS LINE 6
THIS IS LINE 7
THIS IS LINE 8
THIS IS LINE 9
THIS IS LINE 10
```

A carriage return and line feed terminates each line, but is not shown here. Assume that the buffer pointer is in line 6, positioned between the 'I' and the 'S' in the word 'IS'. This example shows the effect of various T commands and then shows how the L and K commands can be used to delete several lines.

```
*0T$$
```

Types out from the start of the current line (line 6) up to the buffer pointer.

```
THIS I*
```

```
*T$$
```

Types from the buffer pointer to the end of the line.

```
S LINE 6
*
```

```
0TT$$
```

Types the whole line without moving the buffer pointer.

```
THIS IS LINE 6
*
```

```
-5T$$
```

Types 5 lines preceding the current line plus the current line from its start to the pointer position.

```
THIS IS LINE 1
THIS IS LINE 2
THIS IS LINE 3
THIS IS LINE 4
THIS IS LINE 5
THIS I*
```

```
5T$$
```

Types five lines including part of the current line from position of the buffer pointer. In this case, five lines were typed. However, if the command were '6T', the sixth line would not be typed because the sixth line after line 6 does not exist inside the text buffer boundaries. The five lines would be printed as in the example below, then the command would be terminated.

```
S LINE 6
THIS IS LINE 7
THIS IS LINE 8
THIS IS LINE 9
THIS IS LINE 10
*
```

```
-5T5T$$
```

Types all ten lines of the buffer. Includes the five lines preceding the buffer pointer, the line containing

the buffer pointer (from the beginning of the line up to the pointer), the remainder of the current line (from the pointer to the end), and the four remaining lines.

```
THIS IS LINE 1
THIS IS LINE 2
THIS IS LINE 3
THIS IS LINE 4
THIS IS LINE 5
THIS IS LINE 6
THIS IS LINE 7
THIS IS LINE 8
THIS IS LINE 9
THIS IS LINE 10
*
```

Now, suppose you want to delete lines 3, 4, 5 and 6. First, the buffer pointer should be positioned either before or after the lines that are to be deleted. We can move the pointer in front of the lines and use a positive argument K command. Because the pointer is in line 6, it must be moved in front of line 3. Then the four lines 3, 4, 5 and 6 can be deleted. The command is as follows:

```
*-3L4K$$
```

The L moves the pointer to the start of line 3. The 4K deletes line 3, 4, 5, and 6.

Alternatively, the pointer can be moved to the line following the lines to be deleted and a negative argument K command used, as follows:

```
*L-4K$$
```

The L moves the pointer to the start of line 7. The -4K deletes lines 3, 4, 5, and 6.

To verify the deletion, you can type out the entire text buffer.

```
*B10T$$
```

The buffer pointer is moved to the beginning of the buffer. The command 10T attempts to type out 10 lines, but only six lines remain. The command terminates when all six lines have been typed.

```
THIS IS LINE 1
THIS IS LINE 2
THIS IS LINE 7
THIS IS LINE 8
THIS IS LINE 9
THIS IS LINE 10
```

Search Commands

Two editor commands are available to search for specified text strings. The F command finds a particular string of text. The S command finds the text string and substitutes another one in its place. Both commands start searching at the pointer and search forward until the search is satisfied or until the end of the buffer is reached.

F - Find Text String

The F command finds a text string of up to 255 characters. The characters ESC and ALT MODE are not considered to be text because of their control functions and cannot be included in the set of text characters.

The ZDOS Text Editor F command differs from the INTEL editor in the number of characters allowable in the string (255 versus 16). Also ZDOS text editor allows finding the nth occurrence of the string. The value of n can be in the range 1 to 9999. A negative n or 0n produces an error message.

The syntax of the F command is as follows:

```
nFtext$$
```

where text is the specified text string of 1 to 255 characters, including non-printing characters.

The F command causes the editor to search for the nth occurrence of a character string matching the character string specified in the command. All characters must match, including printing and non-printing characters (such as cr-lf). The search is started at the current location of the buffer pointer and continues until either the end of buffer is reached or a successful match is made.

If a successful match is made, the editor terminates the command, leaving the buffer pointer immediately following the last character. A prompt character is output, requesting the next command.

If no match is found before the end of the buffer is reached, the editor prints the message:

```
CANNOT FIND "text"
```

where text represents the specified string. The buffer pointer, in this case, remains unchanged. If the specified string is larger than 255 characters, the editor reports:

```
STRING TOO LONG
```

If there is no text between the F and the first ESC, the editor

reports:

```
NULL STRING
```

If the F command is part of a command string, a single ESC character terminates the text string, allowing additional commands to be appended. If no other commands are to be included, the text string and command string can both be terminated with ESC ESC (which is echoed on the console as $$).

It is important to remember to terminate the text string before additional commands are appended. Otherwise, the additional commands are treated as part of the text string. For example, the command:

```
FDIVIDEOLT$$
```

initiates a search for the string 'DIVIDEOLT', instead of the intended string 'DIVIDE'. The correct format for this command is:

```
FDIVIDE$OLT$$
```

It is wise to verify that the search has been successful by typing out the line. In some cases, the search string can appear in several unexpected places prior to the line being sought. For example, if the label 'DIV:' is being searched for and several occurrences of the string search string can produce spurious results. A unique combination of characters is required. In this case, it would be better to search for "DIV:" and then verify search results by typing out the line.

If a carriage return occurs in the search string of an F or S command, the editor automatically generates a line feed following it. Thus:

```
*FEND.
NEXT$$
```

search for the string 'END.cr-lfNEXT'.

S - Substitute Text String

The S command finds a text string and substitutes another in its place. The substitution is made only if the search is successful. The syntax of the S command is:

```
Soldtext$newtext$$
```

where newtext is substituted for oldtext if it is found. Each of the strings must be terminated with an ESC. The first string, oldtext, is the search string. Only the first 255 characters are used. The second string, newtext, is the substitution string and can contain any number of characters (excluding the characters ESC, ALT MODE, and CONTROL-C. The substitute string must be terminated

with an ESC or ALT MODE.

If newtext is omitted, the search string is found and deleted. The S command can be used in this manner to selectively delete strings up to 255 characters long.

In searches using the S command, the string searched for must exactly match the string specified in the command, including upper and lower case, punctuation, carriage returns and line feeds. At the completion of the S command, the buffer pointer is located after the last character substituted.

Examples Using Search Commands (F and S)

1. Suppose you want to delete the string 'OFF' from a line in your program that reads:

```
PARAM:  CALL BACKOFF    ;BACK IS THE RETURN.
```

An F command is used to position the buffer pointer before the string to be deleted. A command to accomplish this is:

```
*BFBACK$3DOTT$$
```

This command string moves the pointer to the start of the buffer, then commences a search for the string 'BACK'. At the first occurrence of this string, the pointer is positioned following the K in BACK. The next three characters, OFF, are deleted, and finally the line is typed out. The $$ terminates the command string.

When completed, the line of text will appear as follows:

```
PARAM:  CALL BACK       ;BACK IS THE RETURN.
```

Notice that the S command could be used to produce exactly the same result, as follows:

```
SBACKOFF$BACK$OTT$$
```

In this case, the string 'BACKOFF' is replaced with 'BACK', having the same effect as deleting 'OFF'.

If you are uncertain whether the search string occurs in a prior line of the program, you can do a simple search for the string with the F command and type out the line found with OTT before substituting or deleting, to make sure you have the desired line of text.

2. Suppose you want to search for a string and delete it. This can be done by combining the F and D commands. For example, to delete the label 'PARAM:' in the following line:

```
PARAM:  CALL SUB1
```

a command such as the one below could be used:

```
BFPARAM:$-6D$$
```

The string 'PARAM:' is found with the F command, the buffer pointer is left after the colon (:), and the preceding 6 characters are deleted. The S command would be easier to use in this case, as it would not be necessary to count the number of characters to delete in the search string.

```
BSPARAM:$$
```

3. An error to corrected consists of a misspelled word. The following command is used to search for the incorrect word and replace it with the correct one. Once corrected, a typeout is specified to verify the operation:

```
SINITAIL$INITIAL$0TT$$
```

The editor responds by performing the substitution and typing out the corrected line. At the termination of the operation, the buffer pointer is positioned at the location between the L in INITIAL and the following space. It is more common practice to use '0LT' instead of '0TT' to verify the correction. This leaves the pointer at the beginning of the line.

Input and Output Commands

The next five commands are concerned with input from and output to files on which the text editting is being performed.

R - Read Text into Buffer

The R command is used to fetch text from the input file on the disk into the text buffer. The R command appends the text read to the end of the buffer, regardless of where the buffer pointer is pointing. The R command is normally the first command to be used when you want to edit an already existing file, since you must bring the text from the input file into the text buffer to edit it. Once initiated, the R command continues reading text until one of the following conditions is satisfied:

* An end of file character (CONTROL Z) is read. The end of file character is not placed in the text buffer.

* The work space is full. The work space will be filled only to within 80 bytes of the upper end of RAM memory.

* n lines of text are input to the text buffer.

* The end of file is reached.

The ZDOS Text Editor R command reads n lines of text into the

buffer (n is a positive number in the range 1 to 9999). If the input file is small enough to fit in the buffer, one can use a command such as 1000R$$ to read the entire file into memory. If the file is too large, the buffer will be filled to within 80 bytes of the top of RAM and the editory will report:

```
BUFFER FULL
```

Example:

To input 150 lines of text from the existing diskette file FLOW.SRC on drive 1, perform the following command:

```
D0>EDIT :D1:FLOW.SRC

ZDOS TEXT EDITOR, V3.3
*150R$$
*
```

W - Write Text to Output File

The W command is used to write out n lines of the text buffer to the output file specified in the EDIT command. The text is always taken from the beginning of the text buffer regardless of the current position of the buffer pointer. After n lines of the text is written to the output file, n lines are deleted from the text buffer, and the remaining text is moved up to the beginning of the text buffer.

The read and write commands can be used in any order with any argument. E.g., one can read 1 line, read 10 more lines, write 2 lines, read 5 more lines, insert some text, write 5 lines at a time until the buffer is empty. ZDOS and the Text Editor will maintain the file and buffer pointers.

A - Append 50 Lines of Text from the Disk

The Append command was implemented in the ZDOS Text Editor in order to maintain compatability with the INTEL Editor for those who are used to using the A command. The A command appends 50 lines of text from the input file to the end of the text buffer. If an argument n is supplied, it appends n times 50 lines.

E - Exit After Writing Buffer to Output File

The E command is used to write out the contents of the text buffer, as well as the rest of the input file text still remaining on the disk and exit to ZDOS. The system closes all files and returns to base level.

Q - Quit the Editor

The Q command can be performed to terminate an editting session in progress without doing any output. The input file is unchanged.

The output file is closed and deleted from the disk. Caution must be used in using the Q command, as any text in the buffer or already written to the output file is lost forever.

M - Memory

The M command computes and displays on the console the amount of work space remaining for use by the editor. The editor reports:

```
MEMORY LOCATIONS=xxxxx
```

where xxxxx is a decimal integer with a maximum value 65,535. Note: if you are editting a large file and are running out of work space, use the W command to write out part of your file to the disk. This will free additional work space for the editor's use.

Command Iterations

A command or command string can be repeated any number of times by enclosing the string in angle brackets '<' and '>', preceded by a number that specifies how many times the iteration is to be performed. The syntax of the command is as follows:

```
n<command string>$$
```

where n specifies the number of times the command enclosed in angle brackets is executed.

For example, if your program contains a label 'DIVID' ten times in the source file and you want to shorten the label to 'DIV', you can use an iterative S command, as follows:

```
*B10<SDIVID$DIV$0LT>$$
```

The B command moves the pointer to the beginning of the text buffer. The substitute and print the line command string is repeated ten times.

The command iteration can be a boon to a user who knows how to use it, but can be disaster to the uninitiated. For example, suppose the label 'DIVID' appears ten times in the source file and you want to change it to 'XDIVID'. Please observe what the at first seem obvious command string does for you:

```
*B10<SDIVID$XDIVID$0LT>$$
```

At the first occurrence of the label your would have:

```
XXXXXXXXXXDIVID
```

The correct command string is:

```
*B1O<SDIVID$XDIVID$OLT$L>$$
```

Normally, one doesn't know the number of times a certain string exists in the source file. You can therefore pick a large number for n, such as 100 or 1000. When the iteration runs out of text, the editor will report:

```
CANNOT FIND "text"
```

and terminate the command iteration. It quite obvious that if a large value of n is chosen, you better know how to construct a workable command iteration string. It is a good idea to perform these command iterations at the beginning of an editting session, before large amounts of console input text is appended to the buffer, so that if the iteration goes berserk, then you can quit the editting session without losing text.

The ZDOS Text Editor does not support nested command iterations, as a single level is complex enough for most purposes. Just think what an eight level command iteration could do to your buffer if you got your angle brackets out of line.

Additional Commands

The ZDOS Text Editor has three additional commands not supported by the INTEL Text Editor.

P - Display Next Page on Screen

The P command effectively performs the same function as the command string 23L23T$$. On a CRT with 24 lines, this command allows the viewing of the text buffer as successive pages of 23 lines each. The buffer pointer is moved with each page request.

V - Verify Text Surrounding the Current Pointer Location

The V command effectively performs the same function as the command string -11T12T$$. On a CRT with 24 lines, this command allows viewing the previous 11 lines and the following 12 lines without moving the buffer pointer.

X - Display Address Pointed to by Buffer Pointer

The X command displays the address of the current position pointed to by the buffer pointer as a four digit hexadecimal number. This command is of no particular use to the average user and is mentioned only as reference in case an X is typed inadvertently as a command.

Text Editor Error Messages

The ZDOS Text Editor prints messages on the system console to notify the user of various status conditions.

"n" ILLEGAL HERE

The "n" represents the illegal command character that was incorrectly typed in. If the character is a non-printing ASCII character, the editor will make it visible. For example, a carriage return would be printed as <013>.

CANNOT FIND "text"

Where "text" represents the string which the editor could not find during an F or S command.

ARG > 9999

This message occurs if the value of n specified was greater than 9999.

STRING TOO LONG

A string in an F or S command was greater than 255 characters.

NULL STRING

The string between the F or S and the first ESC was missing.

APPENDIX F

ZDOS ASSEMBLER

Assembler Overview

An assembler performs the clerical function of converting one's assembly language program into machine-executable form. It accepts an ASCII source file and, depending on the output options selected, can produce an executable object file, a listing of the source and assembled code.

ZDOS supports two versions of assemblers; a 8080/Z80 Assembler called ASM and a 6502 Assembler called KIM. Both assemblers reside on the system disk as LDM files. The basic ZDOS assembler can be modified for other processors if desired. All that is required is modification of the opcode lookup tables and minimal modification of subroutines to handle various special instruction types. Operation of the basic assembler is the subject of this section. Specific references to ASM or KIM will be so noted.

Assembler Hardware Environment

The ZDOS Assembler requires the following hardware:

* Cromemco system with Multi-user hardware
* Console device (CRT)
* Diskette unit
* Line printer (if available)

Symbol Table

Each symbol requires 8 bytes of memory. With 16K of memory, there is space for about 900 symbols. Each additional 16K of RAM would add space for 2048 symbols. The assembler checks for symbol table overflow and reports to the console:

SYMBOL OVERFLOW

Input/Output Files

The input to the ZDOS Assembler is an ASCII source file consisting of lines containing instructions, comments and assembler directives. Only the assembly language instructions are converted into executable object code.

The output from the assembler is a hexadecimal format object file stored on the system disk. The HEX object file contains machine language instructions and data that can be loaded into memory for execution. (This applies to ASM output only, as the Cromemco system cannot run 6502 code. Output from KIM has to be down loaded to the KIM-1 boards for debugging using KLOAD.) In addition, the HEX file contains control information governing the loading process and the starting address of the program.

The list file is a formatted file designed to be output to a line printer or console. Included in the listing is the assembled object code, source program statements and a summary of assembly errors.

Assembler Controls

Assembly-Time Commands

The ZDOS Assembler is invoked by entering the extrinsic file name ASM or KIM. The command string includes the name of the file being assembled and any assembler controls desired. Items in the control list are separated by spaces. The command string is terminated by a carriage- return.

ASM filespec <control-list>

The control-list is optional. The filespec can consist of an optional device specification, filename and extension.

Control	Effect
E	List only those lines containing errors.
L:LP:	List assembler output on the line printer.
L:CO:	List assembler output on the console.

The ZDOS Assembler defaults to L:CO: and no errors printed.

Assembler Operation

The following command activates and completes a ZDOS assembly:

ASM FILE.ASC

By default, there will be no assembly listing and a HEX object file is created and output to the disk in a file named FILE.HEX. If this file had been assembled previously, the old copy of FILE.HEX would be erased first, before assembly. The assembler sends out a sign-on message to the console:

ZDOS 8080/Z80 ASSEMBLER, V4.1

After execution, the assembler issues a sign-off message and error summary:

```
        ASSEMBLY COMPLETE

            0 ERRORS(S)
```

In the program source input stage of development, valuable time can be saved by using the E (errors only) control. In this way, simple syntax errors can be corrected at the console without wasting line printer time.

Assembly Listing Format

The assembler outputs the list file to the line printer, providing all the necessary pagination and formatting. An output page consists of 66 lines, 80 characters wide, with a 3 line margin at the top and bottom of the page, a page header and assembly output lines.

Page Header

Columns	Description
1	The string "ZDOS 8080/Z80 ASSEMBLER, V4.1"
or	The string "KDOS 6502 ASSEMBLER, V4.0"
48	The module name supplied by the NAME directive.
64	The string "PAGE".
67-72	Five character position containing the page number.

Assembly Output Line

Columns	Description
1	Assembler error code. Blank if no error.
2	Blank
3-6	The address assigned to the first byte of the object

	code shown in columns 8-9 of this line is printed in hexadecimal. In addition, the result of the value-generating assembler directives ORG, EQU, and END will appear in this field. For END, the program start address value will appear in this field if specified; otherwise blank.
7	Blank
8-9	The first byte of object code produced by the assembler for this source line is printed here in hexadecimal. If the source statement produces no object code (comments and assembler directives), this field is blank.
10-11	Second byte of object code in hexadecimal. This field will be blank is the source statement generates only one byte of object code or no object code.
12-13	Third byte of object code in hexadecimal, if generated; otherwise, blank.
14-15	Fourth byte of object code in hexadecimal, if generated by a DB string statement; otherwise, blank.
16-17	Blank
18-21	Five character positions containing the source line number in decimal,right justified and left blank-filled.
22	Blank
23-...	Listing of assembler source text. This field terminates at column at column 80 for most output devices other than the line printer, which terminates at column 132.

Error Summary

After listing the last line of the assembly output and spacing one line, the assembler lists an error summary line in the following format:

Columns	Description
1-19	The string "ASSEMBLY COMPLETE".
Second line:	
1-5	Number of errors. Five character positions containing the number of errors in the source encountered during assembly. This decimal number is right-justified and left blank-filled.
6	Blank
7-14	The string "ERRORS(S)".

Error Detection and Reporting

The assembler detects and reports three classes of errors; source file errors, run time errors, and assembler control syntax errors.

Source file errors are indicated in the assembly listing by single letter codes listed in column 1 of the erroneous source statement. If multiple erros occur in the same statement, only the first error is reported. A summary of source file errors is sent to the console and list devices.

Run time errors cause the assembly to terminate abnormally. A run time error message is of the form:

```
error type ERROR
```

When such an error occurs, assembly is aborted, open files are closed, and control is returned to the ZDOS base level.

Assembler control errors in the command string are reported as syntax errors or device errors by the system. Under certain circumstances, fatal system errors may be reported.

Error Codes

Source File Errors

Code	Source
E	Expression error. An expression has been constructed erroneously; usually a missing operator or delimiter.
I	Illegal character. A statement contains an invalid ASCII character, or a specified number is illegal in the context of the number base in which it occurs. Also issued if a carriage-return is not followed by a line-feed.
O	Opcode or operand is illegal. An opcode or operand is illegal in this particular device's instruction set.
P	Phase error. Value of symbol being defined has changed between passes 1 and 2 of the assembly. Caused by a forward reference of an operand in an ORG or DS directive. This assembler does not flag multiple definitions with an M error code, as does the INTEL assembler. Instead the first occurrence of the multiply defined label is flagged as a phase error on pass 2.
Q	Questionable syntax. Invalid syntax, usually due to

	a missing opcode.
R	Register error.
U	Undefined symbol. Symbol used has not been defined.
V	Value illegal. Value exceeds permissible range for this operation or is null.

Run Time Errors

Message	Explanation
EOF ERROR	End-of-file (control-Z) has been encountered before an END directive or END was not terminated by a carriage-return, line-feed.
SYMBOL OVERFLOW	Assembler symbol table has overflowed the available memory space. Add more memory or reduce the the number of symbols.

Assembler Control Errors

Message	Explanation
SYNTAX ERROR	Assembler console command line syntax is illegal, usually due to a missing or illegal delimiter or parameter. The entire command line is ignored.
ILLEGAL DEVICE	Illegal disk drive or output list device specified.

For system errors, refer to the ZDOS system error messages.

Downloading 6502 Programs

The KIM assembler creates an absolute hex object file in INTEL hexadecimal format. The extrinsic command KLOAD converts this hex file to MOS Technology hex file format and then down loads to the KIM-1 boards at each user station. KLOAD is invoked as follows:

KLOAD FILE.HEX

When KLOAD has finished converting the hex file, it prompts with:

PRESS RS BUTTON ON KIM, THEN TYPE RETURN

After typing return, KLOAD sends rubouts to the KIM-1 to set the baud rate, and then outputs the hex file to the KIM-1. The hex file appears on the console in its ASCII format. The system then enters a dummy wait loop. The user can reboot ZDOS by using the BREAK key on his console. While ZDOS is asleep, the user can use the KIM-1 to run and debug his program without affecting the system.

VITA

Reginald Byron Mason

Candidate for the Degree of

Master of Science

Thesis: MULTI-USER DISK OPERATING SYSTEM FOR 8080 BASED MICROCOMPUTERS

Major Field: Electrical Engineering

Biographical:

Personal Data: Born in San Francisco, California, February 5, 1947, the son of Mr. and Mrs. Robert D. Mason.

Education: Graduated from Midwest City High School, Midwest City, Oklahoma, in May 1965; received Bachelor of Science in Electrical Engineering degree from the University of Oklahoma in 1970; completed requirements for the Master of Science degree at Oklahoma State University in July, 1981.

Professional Experience: Development Engineer for Western Electric in computer controlled test set design, 1970 to present.