intel®

APPLICATION NOTE

AP-121

June 1981

Software Design and Implementation of Floppy Disk Subsystems

© Intel Corporation, 1981.

207885-001

Software Design and Implementation of Floppy Disk Subsystems

Contents

1. **INTRODUCTION**

   The Physical Interface Level
   The Logical Interface Level
   The File System Interface Level
   Scope of this Note

2. **DISK I/O TECHNIQUES**

   FDC Data Transfer Interface
   Overlapped Operations
   Buffers

3. **THE 8272 FLOPPY DISK CONTROLLER**

   Floppy Disk Commands
   Interface Registers
   Command/Result Phases
   Execution Phase
   Multi-sector and Multi-track Transfers
   Drive Status Polling
   Command Details
   Invalid Commands

4. **8272 PHYSICAL INTERFACE SOFTWARE**

   INITIALIZE$DRIVERS
   EXECUTE$DOCB
   FDCINT
   OUTPUT$CONTROLS$TO$DMA
   OUTPUT$COMMAND$TO$FDC
   INPUT$RESULT$FROM$FDC
   OUTPUT$BYTE$TO$FDC
   INPUT$BYTE$FROM$FDC
   FDC$READY$FOR$COMMAND
   FDC$READY$FOR$RESULT
   OPERATION$CLEAN$UP
   Modifications for Polling Operation

5. **8272 LOGICAL INTERFACE SOFTWARE**

   SPECIFY
   RECALIBRATE
   SEEK
   FORMAT
   WRITE
   READ
   Coping With Errors

207885-001

Contents (Continued)

6. FILE SYSTEMS

File Allocation
The Intel File System
Disk File System Functions

7. KEY 8272 SOFTWARE INTERFACING CONSIDERATIONS

REFERENCES

APPENDIX A—8272 FDC DEVICE DRIVER SOFTWARE

APPENDIX B—8272 FDC EXERCISER PROGRAM

APPENDIX C—8272 DRIVER FLOWCHARTS

1. Introduction

Disk interface software is a major contributor to the efficient and reliable operation of a floppy disk subsystem. This software must be a well-designed compromise between the needs of the application software modules and the capabilities of the floppy disk controller (FDC). In an effort to meet these requirements, the implementation of disk interface software is often divided into several levels of abstraction. The purpose of this application note is to define these software interface levels and describe the design and implementation of a modular and flexible software driver for the 8272 FDC. This note is a companion to AP-116, "An Intelligent Data Base System Using the 8272."

The Physical Interface Level

The software interface level closest to the FDC hardware is referred to as the physical interface level. At this level, interface modules (often called disk drivers or disk handlers) communicate directly with the FDC device. Disk drivers accept floppy disk commands from other software modules, control and monitor the FDC execution of the commands, and finally return operational status information (at command termination) to the requesting modules.

In order to perform these functions, the drivers must support the bit/byte level FDC interface for status and data transfers. In addition, the drivers must field, classify, and service a variety of FDC interrupts.

The Logical Interface Level

System and application software modules often specify disk operation parameters that are not directly compatible with the FDC device. This software incompatibility is typically caused by one of the following:

1. The change from an existing FDC to a functionally equivalent design. Replacing a TTL based controller with an LSI device is an example of a change that may result in software incompatibilities.

2. The upgrade of an existing FDC subsystem to a higher capability design. An expansion from a single-sided, single-density system to a dual-sided, double-density system to increase data storage capacity is an example of such a system change.

3. The abstraction of the disk software interface to avoid redundancy. Many FDC parameters (in particular the density, gap size, number of sectors per track and number of bytes per sector) are fixed for a floppy disk (after formatting). In fact, in many systems these parameters are never changed during the life of the system.

4. The requirement to support a software interface that is independent of the type of disk attached to the system. In this case, a system generated ("logical") disk address (drive, head, cylinder, and sector numbers) must be mapped into a physical floppy disk address. For example, to switch between single- and dual-sided disks, it may be easier and more cost-effective for the software to treat the dual-sided disk as containing twice as many sectors per track (52) rather than as having two sides. With this technique, accesses to sectors 1 through 26 are mapped onto head 0 while accesses to sectors 27 through 52 are mapped onto head 1.

5. The necessity of supporting a bad track map. Since bad tracks depend on the disk media, the bad track mapping varies from disk to disk. In general, the system and application software should not be concerned with calculating bad track parameters. Instead, these software modules should refer to cylinders logically (0 through 76). The logical interface level procedures must map these cylinders into physical cylinder positions in order to avoid the bad tracks.

The key to logical interface software design is the mapping of the "logical disk interface" (as seen by the application software) into the "physical disk interface" (as implemented by the floppy disk drivers). This logical to physical mapping is tightly coupled to system software design and the mapping serves to isolate both applications and system software from the peculiarities of the FDC device. Typical logical interface procedures are described in Table 1.

The File System Interface Level

The file system typically comprises the highest level of disk interface software used by application programs. The file system is designed to treat the disk as a collection of named data areas (known as files). These files are cataloged in the disk directory. File system interface software permits the creation of new files and the deletion of existing files under software control. When a file is created, its name and disk address are entered into the directory; when a file is deleted, its name is removed from the directory. Application software requests the use of a file by executing an OPEN function. Once opened, a file is normally reserved for use by the requesting program or task and the file cannot be reopened by other tasks. When a task no longer needs to use an open file, the task closes the file, releasing it for use by other tasks.

Most file systems also support a set of file attributes that can be specified for each file. File attributes may be used to protect files (e.g., the WRITE PROTECT attribute ensures that an existing file cannot accidentally be overwritten) and to supply system configuration information (e.g., a FORMAT attribute may specify that a file should automatically be created on a new disk when the disk is formatted).

At the file system interface level, application programs need not be explicitly aware of disk storage allocation techniques, block sizes, or file coding strategies. Only a "file name" must be presented in order to open, read or write, and subsequently close a file. Typical file system functions are listed in Table 2.

Table 1: Examples of Logical Interface Procedures

Name	Description
FORMAT DISK	Controls physical disk formatting for all tracks on a disk. Formatting adds FDC recognized cylinder, head, and sector addresses as well as address marks and data synchronization fields (gaps) to the floppy disk media.
RECALIBRATE	Moves the disk read/write head to track 0 (at the outside edge of the disk).
SEEK	Moves the disk read/write head to a specified logical cylinder. The logical and physical cylinder numbers may be different if bad track mapping is used.
READ STATUS	Indicates the status of the floppy disk drive and media. One important use of this procedure is to determine whether a floppy disk is dual-sided.
READ SECTOR	Reads one or more complete sectors starting at a specified disk address (drive, head, cylinder, and sector).
WRITE SECTOR	Writes one or more complete sectors starting at a specified disk address (drive, head, cylinder, and sector).

Table 2: Disk File System Functions

Name	Description
OPEN	Prepare a file for processing. If the file is to be opened for input and the file name is not found in the directory, an error is generated. If the file is opened for output and the file name is not found in the directory, the file is automatically created.
CLOSE	Terminate processing of an open file.
READ	Transfer data from an open file to memory. The READ function is often designed to buffer one or more sectors of data from the disk drive and supply this data to the requesting program, as required.
WRITE	Transfer data from memory to an open file. The WRITE function is often designed to buffer data from the application program until enough data is available to fill a disk sector.
CREATE	Initialize a file and enter its name and attributes into the file directory.
DELETE	Remove a file from the directory and release its storage space.
RENAME	Change the name of a file in the directory.
ATTRIBUTE	Change the attributes of a file.
LOAD	Read a file of executable code into memory.
INITDISK	Initialize a disk by formatting the media and establishing the directory file, the bit map file, and other system files.

Scope of this Note

This application note directly addresses the logical and physical interface levels. A complete 8272 driver (including interrupt service software) is listed in Appendix A. In addition, examples of recalibrate, seek, format, read, and write logical interface level procedures are included as part of the exerciser program found in Appendix B. Wherever possible, specific hardware configuration dependencies are parametized to provide maximum flexibility without requiring major software changes.

2. Disk I/O Techniques

One of the most important software aspects of disk interfacing is the fixed sector size. (Sector sizes are fixed when the disk is formatted.) Individual bytes of disk storage cannot be read/written; instead, complete sectors must be transferred between the floppy disk and system memory.

Selection of the appropriate sector size involves a tradeoff between memory size, disk storage efficiency, and disk transfer efficiency. Basically, the following factors must be weighed:

1. Memory size. The larger the sector size, the larger the memory area that must be reserved for use during disk I/O transfers. For example, a 1K byte disk sector size requires that at least one 1K memory block be reserved for disk I/O.

2. Disk Storage efficiency. Both very large and very small sectors can waste disk storage space as follows. In disk file systems, space must be allocated somewhere on the disk to link the sectors of each file together. If most files are composed of many small sectors, a large amount of linkage overhead information is required. At the other extreme, when most files are smaller than a single disk sector, a large amount of space is wasted at the end of each sector.

3. Disk transfer efficiency. A file composed of a few large sectors can be transferred to/from memory more efficiently (faster and with less overhead) than a file composed of many small sectors.

Balancing these considerations requires knowledge of the intended system applications. Typically, for general purpose systems, sector sizes from 128 bytes to 1K bytes are used. For compatibility between single-density and double-density recording with the 8272 floppy disk controller, 256 byte sectors or 512 byte sectors are most useful.

FDC Data Transfer Interface

Three distinct software interface techniques may be used to interface system memory to the FDC device during sector data transfers:

1. DMA - In a DMA implementation, the software is only required to set up the DMA controller memory address and transfer count, and to initiate the data transfer. The DMA controller hardware handshakes with the processor/system bus in order to perform each data transfer.

2. Interrupt Driven - The FDC generates an interrupt when a data byte is ready to be transferred to memory, or when a data byte is needed from memory. It is the software's responsibility to perform appropriate memory reads/writes in order to transfer data from/to the FDC upon receipt of the interrupt.

3. Polling - Software responsibilities in the polling mode are identical to the responsibilities in the interrupt driven mode. The polling mode, however, is used when interrupt service overhead (context switching) is too large to support the disk data

rate. In this mode, the software determines when to transfer data by continually polling a data request status flag in the FDC status register.

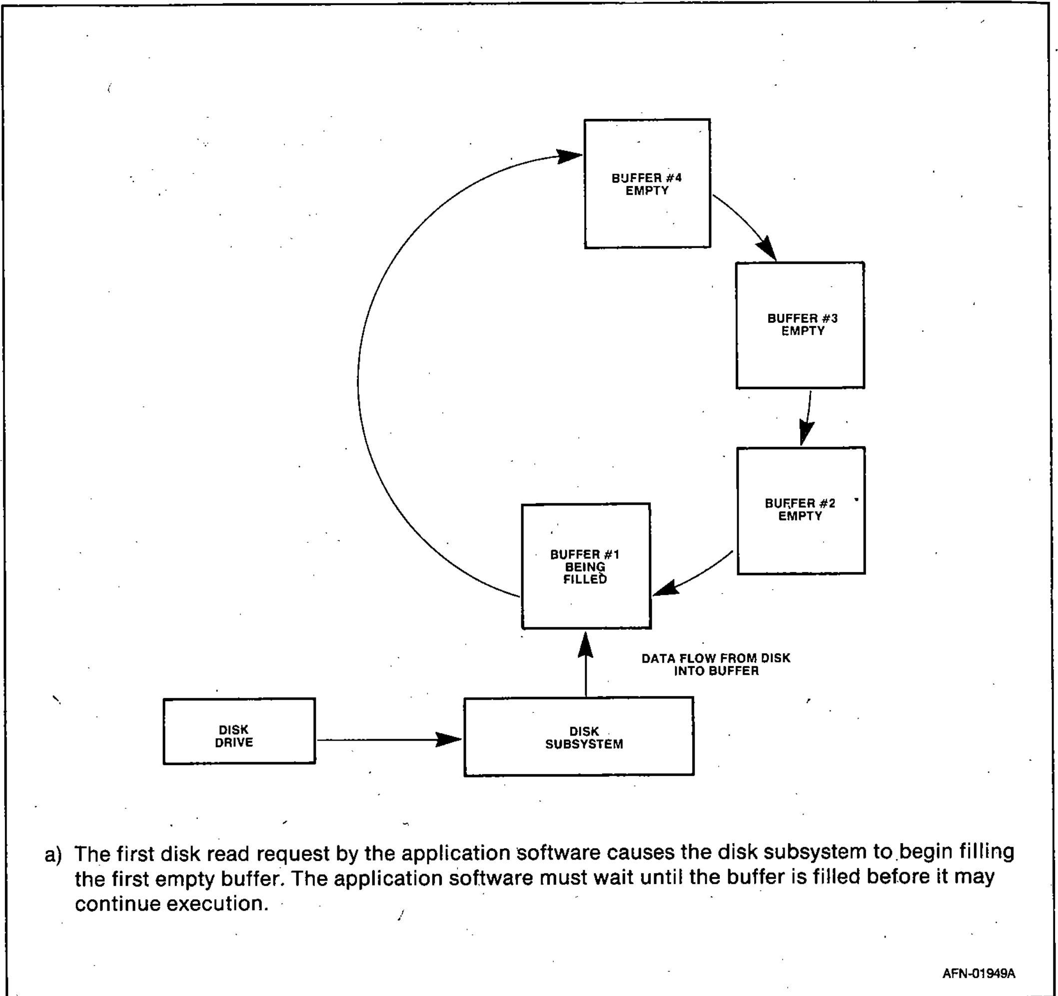
The DMA mode has the advantage of permitting the processor to continue executing instructions while a disk transfer is in progress. (This capability is especially useful in multiprogramming environments when the operating system is designed to permit other tasks to execute while a program is waiting for I/O.) Modes 2 and 3 are often combined and described as non-DMA operating modes. Non-DMA modes have the advantage of significantly lower system cost, but are often performance limited for double-density systems (where data bytes must be transferred to/from the FDC every 16 microseconds).

Overlapped Operations

Some FDC devices support simultaneous disk operations on more than one disk drive. Normally seek and recalibrate operations can be overlapped in this manner. Since seek operations on most floppy drives are extremely slow, this mode of operation can often be used by the system software to reduce overall disk access times.

Buffers

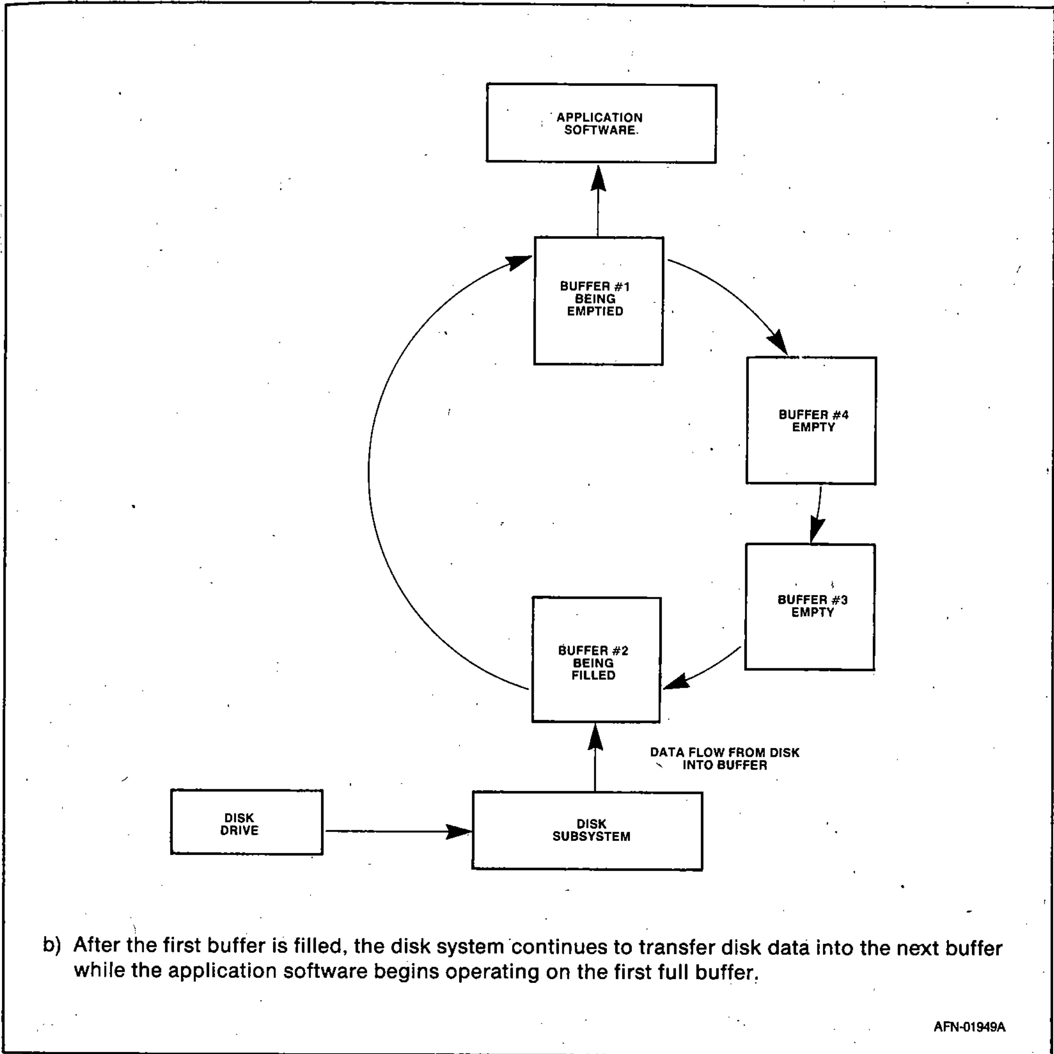
The buffer concept is an extremely important element in advanced disk I/O strategies. A buffer is nothing more than a memory area containing the same amount of data as a disk sector contains. Generally, when an application program requests data from a disk, the system software allocates a buffer (memory area) and transfers the data from the appropriate disk sector into the buffer. The address of the buffer is then returned to the application software. In the same manner, after the application program has filled a buffer for output, the buffer address is passed to the system software, which writes data from the buffer into a disk sector. In multitasking systems, multiple buffers may be allocated from a buffer pool. In these systems, the disk controller is often requested to read ahead and fill additional data buffers while the application software is processing a previous buffer. Using this technique, system software attempts to fill buffers before they are needed by the application programs, thereby eliminating program waits during I/O transfers. Figure 1 illustrates the use of multiple buffers in a ring configuration.



a) The first disk read request by the application software causes the disk subsystem to begin filling the first empty buffer. The application software must wait until the buffer is filled before it may continue execution.

AFN-01949A

Figure 1. Using Multiple Memory Buffers for Disk I/O



b) After the first buffer is filled, the disk system continues to transfer disk data into the next buffer while the application software begins operating on the first full buffer.

Figure 1. Using Multiple Memory Buffers for Disk I/O (Continued)

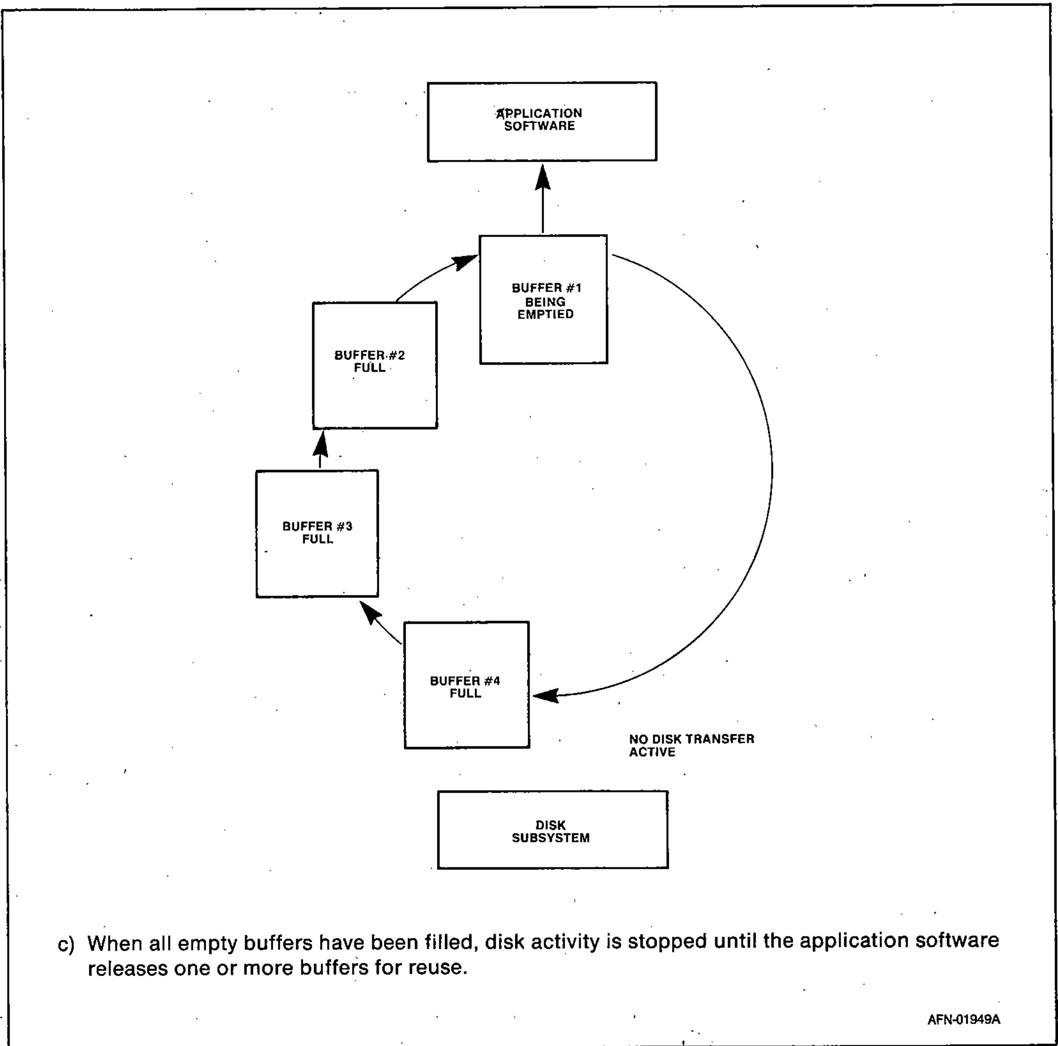
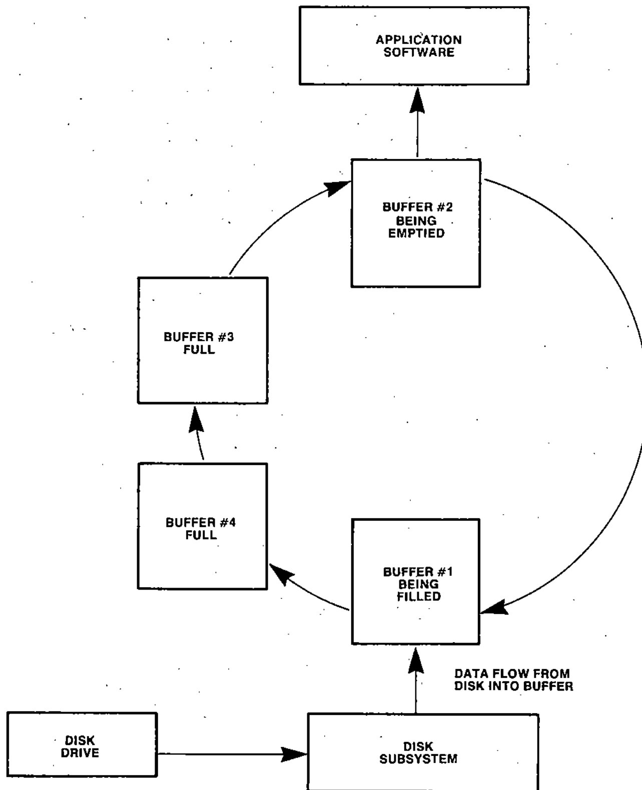


Figure 1. Using Multiple Memory Buffers for Disk I/O (Continued)



d) When the application software releases a buffer (for reuse), the disk subsystem begins a disk sector read to refill the buffer. This strategy attempts to anticipate application software needs by maintaining a sufficient number of full data buffers in order to minimize data transfer delays. If disk data is already in memory when the application software requests it, no disk transfer delays are incurred.

AFN-01949A

Figure 1. Using Multiple Memory Buffers for Disk I/O (Continued)

3. THE 8272 FLOPPY DISK CONTROLLER

The 8272 is a single-chip LSI Floppy Disk Controller (FDC) that implements both single- and double-density floppy disk storage subsystems (with up to four dual-sided disk drives per FDC). The 8272 supports the IBM 3740 single-density recording format (FM) and the IBM System 34 double-density recording format (MFM). The 8272 accepts and executes high-level disk commands such as format track, seek, read sector, and write sector. All data synchronization and error checking is automatically performed by the FDC to ensure reliable data storage and subsequent retrieval. The 8272 interfaces to microprocessor systems with or without Direct Memory Access (DMA) capabilities and also interfaces to a large number of commercially available floppy disk drives.

Floppy Disk Commands

The 8272 executes fifteen high-level disk interface commands:

Specify	Write Data
Sense Drive Status	Write Deleted Data
Sense Interrupt Status	Read Track
Seek	Read ID
Recalibrate	Scan Equal
Format Track	Scan High or Equal
Read Data	Scan Low or Equal
Read Deleted Data	

Each command is initiated by a multi-byte transfer from the driver software to the FDC (the transferred bytes contain command and parameter information). After complete command specification, the FDC automatically executes the command. The command result data (after execution of the command) may require a multi-byte transfer of status information back to the driver. It is convenient to consider each FDC command as consisting of the following three phases:

Command Phase: The driver transfers to the FDC all the information required to perform a particular disk operation. The 8272 automatically enters the command phase after RESET and following the completion of the result phase (if any) of a previous command.

Execution Phase: The FDC performs the operation as instructed. The execution phase is entered immediately after the last command parameter is written to the FDC in the preceding command phase. The execution phase normally ends when the last data byte is transferred to/from the disk or when an error occurs.

Result Phase: After completion of the disk operation, status and other housekeeping information are made available to the driver software. After this information is read, the FDC reenters the command phase and is ready to accept another command.

Interface Registers

To support information transfer between the FDC and the system software, the 8272 contains two 8-bit registers: the Main Status Register and the Data Register. The Main Status Register (read only) contains FDC status information and may be accessed at any time. The Main Status Register (Table 3) provides the system processor with the status of each disk drive, the status of the FDC, and the status of the processor interface. The Data Register (read/write) stores data, commands, parameters, and disk drive status information. The Data Register is used to program the FDC during the command phase and to obtain result information after completion of FDC operations.

In addition to the Main Status Register, the FDC contains four additional status registers (ST0, ST1, ST2, and ST3). These registers are only available during the result phase of a command.

Command/Result Phases

Table 4 lists the 8272 command set. For each of the fifteen commands, command and result phase data transfers are listed. A list of abbreviations used in the table is given in Table 5, and the contents of the result status registers (ST0-ST3) are illustrated in Table 6.

The bytes of data which are sent to the 8272 by the drivers during the command phase, and are read out of the 8272 in the result phase, must occur in the order shown in Table 4. That is, the command code must be sent first and the other bytes sent in the prescribed sequence. All bytes of the command and result phases must be read/written as described. After the last byte of data in the command phase is sent to the 8272 the execution phase automatically starts. In a similar fashion, when the last byte of data is read from the 8272 in the result phase, the result phase is automatically ended and the 8272 reenters the command phase.

It is important to note that during the result phase all bytes shown in Table 4 must be read. The Read Data command, for example, has seven bytes of data in the result phase. All seven bytes must be read in order to successfully complete the Read Data command. The 8272 will not accept a new command until all seven bytes have been read. The number of command and result bytes varies from command-to-command.

In order to read data from, or write data to, the Data Register during the command and result phases, the software driver must examine the Main Status Register to determine if the Data Register is available. The DIO (bit 6) and RQM (bit 7) flags in the Main Status Register must be low and high, respectively, before each byte of the command word may be written into the 8272. Many of the commands require multiple bytes, and as a result, the Main Status Register must be read prior to each byte transfer to the 8272. To read status bytes during the result phase, DIO and RQM in the Main Status Register must both be high. Note, checking the Main Status Register in this manner before each byte transfer to/from the 8272 is required only in the command and result phases, and is NOT required during the execution phase.

Table 3: Main Status Register Bit Definitions

BIT NUMBER	SYMBOL	DESCRIPTION
0	D_0B	Disk Drive 0 Busy. Disk Drive 0 is seeking.
1	D_1B	Disk Drive 1 Busy. Disk Drive 1 is seeking.
2	D_2B	Disk Drive 2 Busy. Disk Drive 2 is seeking.
3	D_3B	Disk Drive 3 Busy. Disk Drive 3 is seeking.
4	CB	FDC Busy. A read or write command is in progress.
5	NDM	Non-DMA Mode. The FDC is in the non-DMA mode when this flag is set (1). This flag is set only during the execution phase of commands in the non-DMA mode. Transition of this flag to a zero (0) indicates that the execution phase has ended.
6	DIO	Data Input/Output. Indicates the direction of a data transfer between the FDC and the Data Register. When DIO is set (1), data is read from the Data Register by the processor; when DIO is reset (0), data is written from the processor to the Data Register.
7	RQM	Request for Master. When set (1), this flag indicates that the Data Register is ready to send data to, or receive data from, the processor.

Table 4: 8272 Command Set

PHASE	R/W	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	REMARKS
READ DATA										
Command	W	MT	MFM	SK	0	0	1	1	0	Command Codes
	W	0	0	0	0	0	HDS	DS1	DS0	
	W				C					Sector ID information prior to Command execution
	W				H					
	W				R					
	W				N					
	W				EOT					
	W				GPL					
	W				DTL					
Execution										Data transfer between the FDD and the main-system
Result	R				ST 0					Status information after Command execution
	R				ST 1					
	R				ST 2					
	R				C					Sector ID information after command execution
	R				H					
	R				R					
	R				N					
READ DELETED DATA										
Command	W	MT	MFM	SK	0	1	1	0	0	Command Codes
	W	0	0	0	0	0	HDS	DS1	DS0	
	W				C					Sector ID information prior to Command execution
	W				H					
	W				R					
	W				N					
	W				EC 1					
	W				GPL					
	W				DTL					
Execution										Data transfer between the FDD and the main-system
Result	R				ST 0					Status information after Command execution
	R				ST 1					
	R				ST 2					
	R				C					Sector ID information after Command execution
	R				H					
	R				R					
	R				N					
WRITE DATA										
Command	W	MT	MFM	0	0	0	1	0	1	Command Codes
	W	0	0	0	0	0	HDS	DS1	DS0	
	W				C					Sector ID information prior to Command execution
	W				H					
	W				R					
	W				N					
	W				EOT					
	W				GPL					
	W				DTL					
Execution										Data transfer between the main-system and the FDD
Result	R				ST 0					Status information after Command execution
	R				ST 1					
	R				ST 2					
	R				C					Sector ID information after Command execution
	R				H					
	R				R					
	R				N					
WRITE DELETED DATA										
Command	W	MT	MFM	0	0	1	0	0	1	Command Codes
	W	0	0	0	0	0	HDS	DS1	DS0	
	W				C					Sector ID information prior to Command execution
	W				H					
	W				R					
	W				N					
	W				EOT					
	W				GPL					
	W				DTL					
Execution										Data transfer between the FDD and the main-system
Result	R				ST 0					Status information after Command execution
	R				ST 1					
	R				ST 2					
	R				C					Sector ID information after Command execution
	R				H					
	R				R					
	R				N					
READ A TRACK										
Command	W	0	MFM	SK	0	0	0	1	0	Command Codes
	W	0	0	0	0	0	HDS	DS1	DS0	
	W				C					Sector ID information prior to Command execution
	W				H					
	W				R					
	W				N					
	W				EOT					
	W				GPL					
	W				DTL					
Execution										Data transfer between the FDD and the main-system. FDC reads the complete track contents from the physical Index mark to EOT
Result	R				ST 0					Status information after Command execution
	R				ST 1					
	R				ST 2					
	R				C					Sector ID information after Command execution
	R				H					
	R				R					
	R				N					
READ ID										
Command	W	0	MFM	0	0	1	0	1	0	Command Codes
	W	0	0	0	0	0	HDS	DS1	DS0	
Execution										The first correct ID information on the track is stored in Data Register
Result	R				ST 0					Status information after Command execution
	R				ST 1					
	R				ST 2					
	R				C					Sector ID information during Execution Phase
	R				H					
	R				R					
	R				N					
FORMAT A TRACK										
Command	W	0	MFM	0	0	1	1	0	1	Command Codes
	W	0	0	0	0	0	HDS	DS1	DS0	
	W				N					Bytes/Sector
	W				SC					Sectors/Track
	W				GPL					Gap 3
	W				D					Filler Byte
Execution										FDC formats an entire track
Result	R				ST 0					Status information after Command execution
	R				ST 1					
	R				ST 2					
	R				C					In this case, the ID information has no meaning
	R				H					
	R				R					
	R				N					
SCAN EQUAL										
Command	W	MT	MFM	SK	1	0	0	0	1	Command Codes
	W	0	0	0	0	0	HDS	DS1	DS0	
	W				C					Sector ID information prior to Command execution
	W				H					
	W				R					
	W				N					
	W				EOT					
	W				GPL					
	W				STP					
Execution										Data compared between the FDD and the main-system
Result	R				ST 0					Status information after Command execution
	R				ST 1					
	R				ST 2					
	R				C					Sector ID information after Command execution
	R				H					
	R				R					
	R				N					

Note: 1. $A_0 = 1$ for all operations.

PHASE	R/W	D7	D6	D5	D4	D3	D2	D1	D0	REMARKS
		SCAN LOW OR EQUAL								
Command	W	MT	MFM	SK	1	1	0	0	1	Command Codes
	W	0	0	0	0	0	HDS	DS1	DS0	
	W	C								Sector ID information prior Command execution
	W	H								
	W	R								
	W	N								
	W	EOT								
	W	GPL								
	W	STP								
Execution										Data compared between the FDD and the main-system
Result	R	ST 0								Status information after Command execution
	R	ST 1								
	R	ST 2								
	R	C								Sector ID information after Command execution
	R	H								
	R	R								
	R	N								
		SCAN HIGH OR EQUAL								
Command	W	MT	MFM	SK	1	1	1	0	1	Command Codes
	W	0	0	0	0	0	HDS	DS1	DS0	
	W	C								Sector ID information prior Command execution
	W	H								
	W	R								
	W	N								
	W	EOT								
	W	GPL								
	W	STP								
Execution										Data compared between the FDD and the main-system
Result	R	ST 0								Status information after Command execution
	R	ST 1								
	R	ST 2								
	R	C								Sector ID information after Command execution
	R	H								
	R	R								
	R	N								

PHASE	R/W	D7	D6	D5	D4	D3	D2	D1	D0	REMARKS
		RECALIBRATE								
Command	W	0	0	0	0	0	1	1	1	Command Codes
	W	0	0	0	0	0	0	DS1	DS0	
Execution										Head retracted to Track 0
		SENSE INTERRUPT STATUS								
Command	W	0	0	0	0	1	0	0	0	Command Codes
Result	R	ST 0								Status information at the end of each seek operation about the FDC
	R	C								
		SPECIFY								
Command	W	0	0	0	0	0	0	1	1	Command Codes
	W	SPT				HUT				Timer Settings
	W	HLT							ND	
		SENSE DRIVE STATUS								
Command	W	0	0	0	0	0	1	0	0	Command Codes
	W	0	0	0	0	0	HDS	DS1	DS0	
Result	R	ST 3								Status information about the FDD
		SEEK								
Command	W	0	0	0	0	1	1	1	1	Command Codes
	W	0	0	0	0	0	HDS	DS1	DS0	
	W	C								
Execution										Head is positioned over proper Cylinder on Diskette
		INVALID								
Command	W	Invalid Codes								Invalid Command Codes (NoOp — FDC goes into Standby State)
Result	R	ST 0								ST 0 = 80 (16)

Table 5: Command/Result Parameter Abbreviations

SYMBOL	DESCRIPTION
C	Cylinder Address. The currently selected cylinder address (0 to 76) on the disk.
D	Data Pattern. The pattern to be written in each sector data field during formatting.
DS0,DS1	Disk Drive Select. DS1 DS0 0 0 Drive 0 0 1 Drive 1 1 0 Drive 2 1 1 Drive 3
DTL	Special Sector Size. During the execution of disk read/write commands, this parameter is used to temporarily alter the effective disk sector size. By setting N to zero, DTL may be used to specify a sector size from 1 to 256 bytes in length. If the actual sector (on the disk) is larger than DTL specifies, the remainder of the actual sector is not passed to the system during read commands; during write commands, the remainder of the actual sector is written with all-zeroes bytes. DTL should be set to FF hexadecimal when N is not zero.
EOT	End of Track. The final sector number of the current track.
GPL	Gap Length. The gap 3 size. (Gap 3 is the space between sectors.)
H	Head Address. Selected head: 0 or 1 (disk side 0 or 1, respectively) as encoded in the sector ID field.
HLT	Head Load Time. Defines the time interval that the FDC waits after loading the head before initiating a read or write operation. Programmable from 2 to 254 milliseconds (in increments of 2 ms).
HUT	Head Unload Time. Defines the time interval from the end of the execution phase (of a read or write command) until the head is unloaded. Programmable from 16 to 240 milliseconds (in increments of 16 ms).
MFM	MFM/FM Mode Selector. Selects MFM double-density recording mode when high, FM single-density mode when low.
MT	Multi-Track Selector. When set, this flag selects the multi-track operating mode. In this mode (used only with dual-sided disks), the FDC treats a complete cylinder (under both read/write head 0 and read/write head 1) as a single track. The FDC operates as if this expanded track started at the first sector under head 0 and ended at the last sector under head 1. With this flag set (high), a multi-sector read operation will automatically continue to the first sector under head 1 when the FDC finishes operating on the last sector under head 0.
N	Sector Size Code. The number of data bytes within a sector.

ND	Non-DMA Mode Flag. When set (1), this flag indicates that the FDC is to operate in the non-DMA mode. In this mode, the processor participates in each data transfer (by means of an interrupt or by polling the RQM flag in the Main Status Register). When reset (0), the FDC interfaces to a DMA controller.
R	Sector Address. Specifies the sector number to be read or written. In multi-sector transfers, this parameter specifies the sector number of the first sector to be read or written.
SC	Number of Sectors per Track. Specifies the number of sectors per track to be initialized by the Format Track command.
SK	Skip Flag. When this flag is set, sectors containing deleted data address marks will automatically be skipped during the execution of multi-sector Read Data or Scan commands. In the same manner, a sector containing a data address mark will automatically be skipped during the execution of a multi-sector Read Deleted Data command.
SRT	Step Rate Interval. Defines the time interval between step pulses issued by the FDC (track-to-track access time). Programmable from 1 to 16 milliseconds (in increments of 1 ms).
ST0 ST1 ST2 ST3	Status Register 0-3. Registers within the FDC that store status information after a command has been executed. This status information is available to the processor during the Result Phase after command execution. These registers may only be read after a command has been executed (in the exact order shown in Table 4 for each command). These registers should not be confused with the Main Status Register.
STP	Scan Sector Increment. During Scan operations, this parameter is added to the current sector number in order to determine the next sector to be scanned.

Table 6: Status Register Definitions

Status Register 0		
BIT NUMBER	SYMBOL	DESCRIPTION
7,6	IC	Interrupt Code. 00 - Normal termination of command. The specified command was properly executed and completed without error. 01 - Abnormal termination of command. Command execution was started but could not be successfully completed. 10 - Invalid command. The requested command could not be executed. 11 - Abnormal termination. During command execution, the disk drive ready signal changed state.
5	SE	Seek End. This flag is set (1) when the FDC has completed the Seek command and the read/write head is positioned over the correct cylinder.
4	EC	Equipment Check Error. This flag is set (1) if a fault signal is received from the disk drive or if the track 0 signal is not received from the disk drive after 77 step pulses (Recalibrate command).
3	NR	Not Ready Error. This flag is set if a read or write command is issued and either the drive is not ready or the command specifies side 1 (head 1) of a single-sided disk.
2	H	Head Address. The head address at the time of the interrupt.
1,0	DS1,DS0	Drive Select. The number of the drive selected at the time of the interrupt.

Status Register 1		
BIT NUMBER	SYMBOL	DESCRIPTION
7	EN	End of Track Error. This flag is set if the FDC attempts to access a sector beyond the final sector of the track.
6		Undefined
5	DE	Data Error. Set when the FDC detects a CRC error in either the the ID field or the data field of a sector.
4	OR	Overrun Error. Set (during data transfers) if the FDC does not receive DMA or processor service within the specified time interval.

3		Undefined
2	ND	Sector Not Found Error. This flag is set by any of the following conditions. a) The FDC cannot locate the sector specified in the Read Data, Read Deleted Data, or Scan command. b) The FDC cannot locate the starting sector specified in the Read Track command. c) The FDC cannot read the ID field without error during a Read ID command.
1	NW	Write Protect Error. This flag is set if the FDC detects a write protect signal from the disk drive during the execution of a Write Data, Write Deleted Data, or Format Track command.
0	MA	Missing Address Mark Error. This flag is set by either of the following conditions: a) The FDC cannot detect the ID address mark on the specified track (after two rotations of the disk). b) The FDC cannot detect the data address mark or deleted data address mark on the specified track. (See also the MD bit of Status Register 2.)

Status Register 2

BIT NUMBER	SYMBOL	DESCRIPTION
7		Undefined
6	CM	Control Mark. This flag is set when the FDC encounters one of the following conditions: a) A deleted data address mark during the execution of a Read Data or Scan command. b) A data address mark during the execution of a Read Deleted Data command.
5	DD	Data Error. Set (1) when the FDC detects a CRC error in a sector data field. This flag is not set when a CRC error is detected in the ID field.
4	WC	Cylinder Address Error. Set when the cylinder address from the disk sector ID field is different from the current cylinder address maintained within the FDC.
3	SH	Scan Hit. Set during the execution of the Scan command if the scan condition is satisfied.
2	SN	Scan Not Satisfied. Set during execution of the Scan command if the FDC cannot locate a sector on the specified cylinder that satisfies the scan condition.

1	BC	Bad Track Error. Set when the cylinder address from the disk sector ID field is FF hexadecimal and this cylinder address is different from the current cylinder address maintained within the FDC. This all "ones" cylinder number indicates a bad track (one containing hard errors) according to the IBM soft-sectored format specifications.
0	MD	Missing Data Address Mark Error. Set if the FDC cannot detect a data address mark or deleted data address mark on the specified track.

Status Register 3

BIT NUMBER	SYMBOL	DESCRIPTION
7	FT	Fault. This flag indicates the status of the fault signal from the selected disk drive.
6	WP	Write Protected. This flag indicates the status of the write protect signal from the selected disk drive.
5	RDY	Ready. This flag indicates the status of the ready signal from the selected disk drive.
4	T0	Track 0. This flag indicates the status of the track 0 signal from the selected disk drive.
3	TS	Two-Sided. This flag indicates the status of the two-sided signal from the selected disk drive.
2	H	Head Address. This flag indicates the status of the side select signal for the currently selected disk drive.
1,0	DS1,DS0	Drive Select. Indicates the currently selected disk drive number.

Execution Phase

All data transfers to (or from) the floppy drive occur during the execution phase. The 8272 has two primary modes of operation for data transfers (selected by the specify command):

1) DMA mode
2) non-DMA mode

In the DMA mode, execution phase data transfers are handled by the DMA controller hardware (invisible to the driver software). The driver software, however, must set all appropriate DMA controller registers prior to the beginning of the disk operation. An interrupt is generated by the 8272 after the last data transfer, indicating the completion of the execution phase, and the beginning of the result phase.

In the non-DMA mode, transfer requests are indicated by generation of an interrupt and by activation of the RQM flag (bit 7 in the Main Status Register). The interrupt signal can be used for interrupt-driven systems and RQM can be used for polled systems. The driver software must respond to the transfer request by reading data from, or writing data to, the FDC. After completing the last transfer, the 8272 generates an interrupt to indicate the beginning of the result phase. In the non-DMA mode, the processor must activate the "terminal count" (TC) signal to the FDC (normally by means of an I/O port) after the transfer request for the last data byte has been received (by the driver) and before the appropriate data byte has been read from (or written to) the FDC.

In either mode of operation (DMA or non-DMA), the execution phase ends when a "terminal count" signal is sensed by the FDC, when the last sector on a track (the EOT parameter - Table 4) has been read or written, or when an error occurs.

Multi-sector and Multi-track Transfers

During disk read/write transfers (Read Data, Write Data, Read Deleted Data, and Write Deleted Data), the FDC will continue to transfer data from sequential sectors until the TC input is sensed. In the DMA mode, the TC input is normally set by the DMA controller. In the non-DMA mode, the processor directly controls the FDC TC input as previously described. Once the TC input is received, the FDC stops requesting data transfers (from the system software or DMA controller). The FDC, however, continues to read data from, or write data to, the floppy disk until the end of the current disk sector. During a disk read operation, the data read from the disk (after reception of the TC input) is discarded, but the data CRC is checked for errors; during a disk write operation, the remainder of the sector is filled with all-zero bytes.

If the TC signal is not received before the last byte of the current sector has been transferred to/from the system, the FDC increments the sector number by one and initiates a read or write command for this new disk sector.

The FDC is also designed to operate in a multi-track mode for dual-sided disks. In the multi-track mode (specified by means of the MT flag in the command byte - Table 4) the FDC will automatically increment the head address (from 0 to 1) when the last sector (on the track under head 0) has been read or written. Reading or writing is then continued on the first sector (sector 1) of head 1.

Drive Status Polling

After the power-on reset, the 8272 automatically enters a drive status polling mode. If a change in drive status is detected (all drives are assumed to be "not ready" at power-on), an interrupt is generated. The 8272 continues this status polling between command executions (and between step pulses in the Seek command). In this manner, the 8272 automatically notifies the system software whenever a floppy disk is inserted, removed, or changed by the operator.

Command Details

During the command phase, the Main Status Register must be polled by the driver software before each byte is written into the Data Register. The DIO (bit 6) and RQM (bit 7) flags in the Main Status Register must be low and high, respectively, before each byte of the command may be written into the 8272. The beginning of the execution phase for any of these commands will cause DIO to be set high and RQM to be set low.

Operation of the FDC commands is described in detail in Application Note AP-116, "An Intelligent Data Base System Using the 8272."

Invalid Commands

If an invalid (undefined) command is sent to the FDC, the FDC will terminate the command. No interrupt is generated by the 8272 during this condition. Bit 6 and bit 7 (DIO and RQM) in the Main Status Register are both set indicating to the processor that the 8272 is in the result phase and the contents of Status Register 0 must be read. When the processor reads Status Register 0 it will find an 80H code indicating that an invalid command was received. The driver software in Appendix B checks each requested command and will not issue an invalid command to the 8272.

A Sense Interrupt Status command must be sent after a Seek or Recalibrate interrupt; otherwise the FDC will consider the next command to be an invalid command. Also, when the last "hidden" interrupt has been serviced, further Sense Interrupt Status commands will result in invalid command codes.

4. 8272 Physical Interface Software

PL/M software driver listings for the 8272 FDC are contained in Appendix A. These drivers have been designed to operate in a DMA environment (as described in Application Note AP-116, "An Intelligent Data Base System Using the 8272"). In the following paragraphs, each driver procedure is described. (A description of the driver data base variables is given in Table 7.) In addition, the modifications necessary to reconfigure the drivers for operation in a polled environment are discussed.

INITIALIZE$DRIVERS

This initialization procedure must be called before any FDC operations are attempted. This module initializes the DRIVE$READY, DRIVE$STATUS$CHANGE, OPERATION$IN$PROGRESS, and OPERATION$COMPLETE arrays as well as the GLOBAL$DRIVE$NO variable.

EXECUTE$DOCB

This procedure contains the main 8272 driver control software and handles the execution of a complete FDC command. EXECUTE$DOCB is called with two parameters: a) a pointer to a disk operation control block and b) a pointer to a result status byte. The format of the disk operation control block is illustrated in Figure 2 and the result status codes are described in Table 8.

Before starting the command phase for the specified disk operation, the command is checked for validity and to determine whether the FDC is busy. (For an overlapped operation, if the FDC BUSY flag is set – in the Main Status Register – the command cannot be started; non-overlapped operations cannot be started if the FDC BUSY flag is set, if any drive is in the process of seeking/recalibrating, or if an operation is currently in progress on the specified drive.)

After these checks are made, interrupts are disabled in order to set the OPERATIONINPROGRESS flag, reset the OPERATION$COMPLETE flag, load a pointer to the current operation control block into the OPERATION$DOCB$PTR array and set GLOBAL$DRIVE$NO (if a non-overlapped operation is to be started).

At this point, parameters from the operation control block are output to the DMA controller and the FDC command phase is initiated. After completion of the command phase, a test is made to determine the type of result phase required for the current operation. If no result phase is needed, control is immediately returned to the calling program. If an immediate result phase is required, the result bytes are input from the FDC. Otherwise, the CPU waits until the OPERATION$COMPLETE flag is set (by the interrupt service procedure).

Finally, if an error is detected in the result status code (from the FDC), an FDC operation error is reported to the calling program.

Table 7: Driver Data Base

NAME	DESCRIPTION
DRIVE$READY	A public array containing the current "ready" status of each drive.
DRIVE$STATUS$CHANGE	A public array containing a flag for each drive. The appropriate flag is set whenever the ready status of a drive changes.
OPERATION$DOCB$PTR	An internal array of pointers to the operation control block currently in progress for each drive.
OPERATIONINPROGRESS	An internal array used by the driver procedures to determine if a disk operation is in progress on a given drive.
OPERATION$COMPLETE	An internal array used by the driver procedures to determine when the execution phase of a disk operation is complete.
GLOBAL$DRIVE$NO	A data byte that records the current drive number for non-overlapped disk operations.
VALID$COMMAND	A constant flag array that indicates whether a specified FDC command code is valid.
COMMAND$LENGTH	A constant byte array specifying the number of command/parameter bytes to be transferred to the FDC during the command phase.
DRIVENOPRESENT	A constant flag array that indicates whether a drive number is encoded into an FDC command.
OVERLAP$OPERATION	A constant flag array that indicates whether an FDC command can be overlapped with other commands.
NO$RESULT	A constant flag array that is used to determine when an FDC operation does not have a result phase.
IMMED$RESULT	A constant flag array that indicates that an FDC operation has a result phase beginning immediately after the command phase is complete.
POSSIBLE$ERROR	A constant flag array that indicates if an FDC operation should be checked for an error status indication during the result phase.

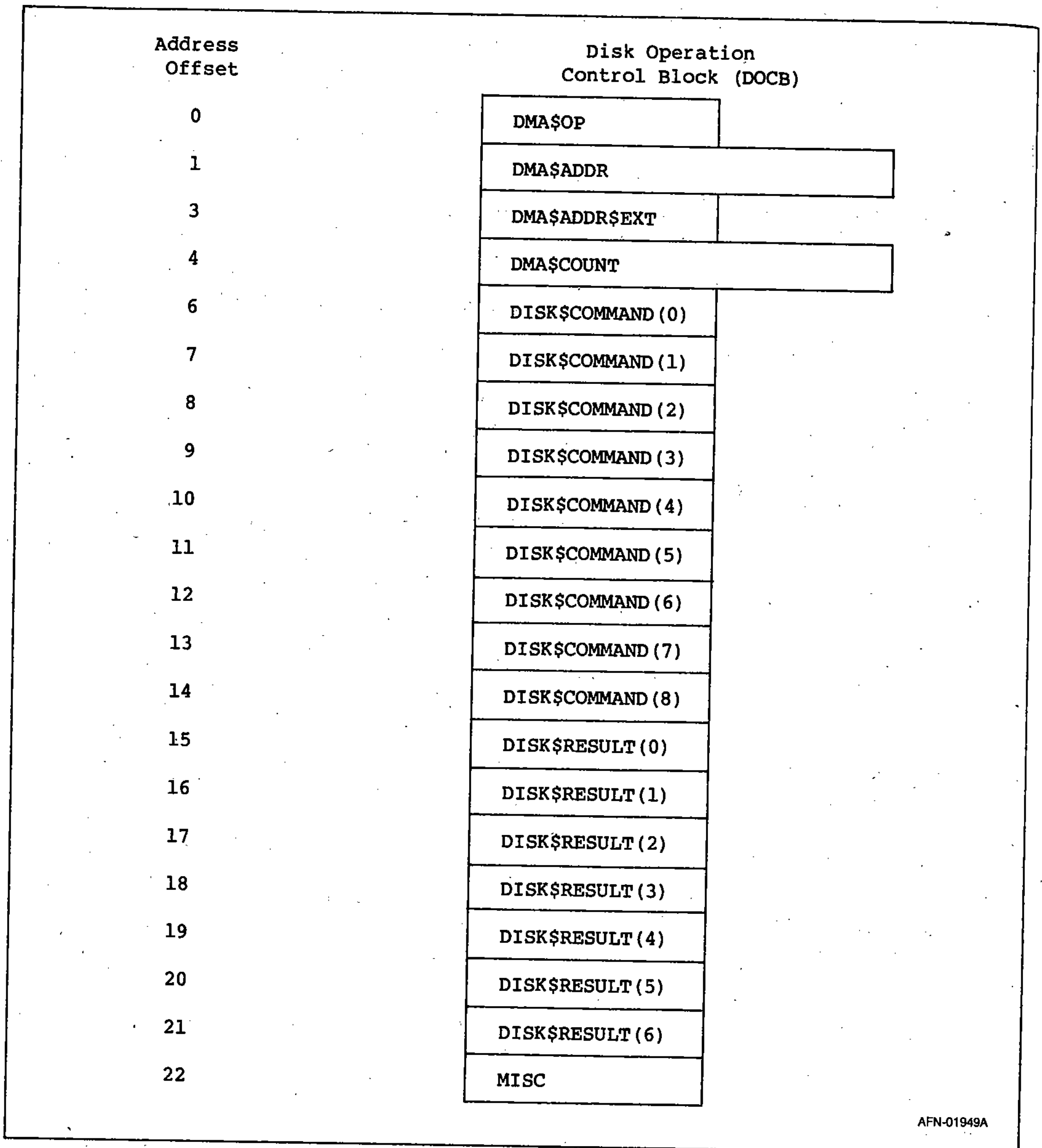


Figure 2. Disk Operation Control Block (DOCB) Format

Table 8: EXECUTE$DOCB Return Status Codes

Code	Description
0	No errors. The specified operation was completed without error.
1	FDC busy. The requested operation cannot be started. This error occurs if an attempt is made to start an operation before the previous operation is completed.
2	FDC error. An error was detected by the FDC during the execution phase of a disk operation. Additional error information is contained in the result data portion of the disk operation control block (DOCB.DISK$RESULT) as described in the 8272 data sheet. This error occurs whenever the 8272 reports an execution phase error (e.g., missing address mark).
3	8272 command interface error. An 8272 interfacing error was detected during the command phase. This error occurs when the command phase of a disk operation cannot be successfully completed (e.g., incorrect setting of the DIO flag in the Main Status Register).
4	8272 result interface error. An 8272 interfacing error was detected during the result phase. This error occurs when the result phase of a disk operation cannot be successfully completed (e.g., incorrect setting of the DIO flag in the Main Status Register).
5	Invalid FDC Command.

FDCINT

This procedure performs all interrupt processing for the 8272 interface drivers. Basically, two types of interrupts are generated by the 8272: (a) an interrupt that signals the end of a command execution phase and the beginning of the result phase and (b) an interrupt that signals the completion of an overlapped operation or the occurrence of an unexpected event (e.g., change in the drive "ready" status).

An interrupt of type (a) is indicated when the FDC BUSY flag is set (in the Main Status Register). When a type (a) interrupt is sensed, the result bytes are read from the 8272 and placed in the result portion of the disk operation control block, the appropriate OPERATION$COMPLETE flag is set, and the OPERATIONINPROGRESS flag is reset.

When an interrupt of type (b) is indicated (FDC not busy), a sense interrupt status command is issued (to the FDC). The upper two bits of the result status register (Status Register Zero - ST0) are used to determine the cause of the interrupt. The following four cases are possible:

1) Operation Complete. An overlapped operation is complete. The drive number is found in the lower two bits of ST0. The ST0 data is transferred to the active operation control block, the OPERATION$COMPLETE flag is set, and the OPERATIONINPROGRESS flag is reset.

2) Abnormal Termination. A disk operation has abnormally terminated. The drive number is found in the lower two bits of ST0. The ST0 data is transferred to the active control block, the OPERATION$COMPLETE flag is set, and the OPERATIONINPROGRESS flag is reset.

3) Invalid Command. The execution of an invalid command (i.e., a sense interrupt command with no interrupt pending) has been attempted. This interrupt signals the successful completion of all interrupt processing.

4) Drive Status Change. A change has occurred in the "ready" status of a disk drive. The drive number is found in the lower two bits of ST0. The DRIVE$READY flag for this disk drive is set to the new drive "ready" status and the DRIVE$STATUS$CHANGE flag for the drive is also set. In addition, if a command is currently in progress, the ST0 data is transferred to the active control block, the OPERATION$COMPLETE flag is set, and the OPERATIONINPROGRESS flag is reset.

After processing a type (b) interrupt, additional sense interrupt status commands must be issued and processed until an "invalid command" result is returned from the FDC. This action guarantees that all "hidden" interrupts are serviced.

In addition to the major driver procedures described above, a number of support procedures are required. These support routines are briefly described in the following paragraphs.

OUTPUT$CONTROLS$TO$DMA

This procedure outputs the DMA mode, the DMA address, and the DMA word count to the 8237 DMA controller. In addition, the upper four bits of the 20-bit DMA address are output to the address extension latch. Finally, the disk DMA channel is started.

OUTPUT$COMMAND$TO$FDC

This software module outputs a complete disk command to the 8272 FDC. The number of required command/parameter bytes is found in the COMMAND$LENGTH table. The appropriate bytes are output one at a time (by calls to OUTPUT$BYTE$TO$FDC) from the command portion of the disk operation control block.

INPUT$RESULT$FROM$FDC

This procedure is used to read result phase status information from the disk controller. At most, seven bytes are read. In order to read each byte, a call is made to INPUT$BYTE$FROM$FDC. When the last byte has been read, a check is made to insure that the FDC is no longer busy.

OUTPUT$BYTE$TO$FDC

This software is used to output a single command/parameter byte to the FDC. This procedure waits until the FDC is ready for a command byte and then outputs the byte to the FDC data port.

INPUT$BYTE$FROM$FDC

This procedure inputs a single result byte from the FDC. The software waits until the FDC is ready to transfer a result byte and then reads the byte from the FDC data port.

FDC$READY$FOR$COMMAND

This procedure assures that the FDC is ready to accept a command/parameter byte by performing the following three steps. First, a small time interval (more than 20 microseconds) is inserted to assure that the RQM flag has time to become valid (after the last byte transfer). Second, the master request flag (RQM) is polled until it is activated by the FDC. Finally, the DIO flag is checked to ensure that it is properly set for FDC input (from the processor).

FDC$READY$FOR$RESULT

The operation of this procedure is similar to the FDC$READY$FOR$COMMAND with the following exception. If the FDC BUSY flag (in the Main Status Register) is not set, the result phase is complete and no more data is available from the FDC. Otherwise, the procedure waits for the RQM flag and checks the DIO flag for FDC output (to the processor).

OPERATION$CLEAN$UP

This procedure is called after the execution of a disk operation that has no result phase. OPERATION$CLEAN$UP resets the OPERATIONINPROGRESS flag and the GLOBAL$DRIVE$NO variable if appropriate. This procedure is also called to clean up after some disk operation errors.

Modifications for Polling Operation

To operate in the polling mode, the following modifications should be made to the previous routines:

1. The OUTPUT$CONTROLS$TO$DMA routine should be deleted.
2. In EXECUTE$DOCB, immediately prior to WAITFOROP$COMPLETE, a polling loop should be inserted into the code. The loop should test the RQM flag (in the Main Status Register). When RQM is set, a data byte should be written to, or read from, the 8272. The buffer address may be computed from the base address contained in DOCB.DMA$ADDR and DOCB.DMA$ADDR$EXT. After the correct number of bytes have been transferred, an operation complete interrupt will be issued by the FDC. During data transfer in the non-DMA mode, the NON-DMA MODE flag (bit 5 of the Main Status Register) will be set. This flag will remain set for the complete execution phase. When the transfer is finished, the NON-DMA MODE flag is reset and the result phase interrupt is issued by the FDC.

5. 8272 Logical Interface Software

Appendix B of this Application Note contains a PL/M listing of an exerciser program for the 8272 drivers. This program illustrates the design of logical interface level procedures to specify disk parameters, recalibrate a drive, seek to a cylinder, format a disk, read data, and write data.

The exerciser program is written to operate a standard single-sided 8" floppy disk drive in either the single- or double-density recording mode. Only the eight parameters listed in Table 9 must be specified. All other parameters are derived from these 8 basic variables.

Each of these logical interface procedures is described in the following paragraphs (refer to the listing in Appendix B).

SPECIFY

This procedure sets the FDC signal timing so that the FDC will interface correctly to the attached disk drive. The SPECIFY procedure requires four parameters, the step rate (SRT), head load time (HLT), head unload time (HUT), and the non-DMA mode flag (ND). This procedure builds a disk operation control block (SPECIFY$DOCB) and passes the control block to the FDC driver module (EXECUTE$DOCB) for execution. (Note carefully the computation required to transform the step rate (SRT) into the correct 8272 parameter byte.)

RECALIBRATE

This procedure causes the floppy disk read/write head to retract to track 0. The RECALIBRATE procedure requires only one parameter — the drive number on which the recalibrate operation is to be performed. This procedure builds a disk operation control block (RECALIBRATE$DOCB) and passes the control block to the FDC driver for execution.

SEEK

This procedure causes the disk read/write head (on the selected drive) to move to the desired cylinder position. The SEEK procedure is called with three parameters: drive number (DRV), head/side number (HD), and cylinder number (CYL). This software module builds a disk operation control block (SEEK$DOCB) that is executed by the FDC driver.

FORMAT

The FORMAT procedure is designed to initialize a complete floppy disk so that sectors can subsequently be read and written by system and application programs. Three parameters must be supplied to this procedure: the drive number (DRV), the recording density (DENS), and the interleave factor (INTLVE). The FORMAT procedure generates a data block (FMTBLK) and a disk operation control block (FORMAT$DOCB) for each track on the floppy disk (normally 77).

Table 9: Basic Disk Parameters

Name	Description
DENSITY	The recording mode (FM or MFM).
FILLER$BYTE	The data byte to be written in all sectors during formatting.
TRACKSPERDISK	The number of cylinders on the floppy disk.
BYTESPERSECTOR	The number of bytes in each disk sector. The exerciser accepts 128, 256, and 512 in FM mode, and 256, 512, and 1024 in MFM mode.
INTERLEAVE	The sector interleave factor for each disk track.
STEP$RATE	The disk drive step rate (1-16 milliseconds).
HEAD$LOAD$TIME	The disk drive head load time (2-254 milliseconds).
HEAD$UNLOAD$TIME	The head unload time (16-240 milliseconds).

The format data block specifies the four sector ID field parameters (cylinder, head, sector, and bytes per sector) for each sector on the track. The sector numbers need not be sequential; the interleave factor (INTLVE parameter) is used to compute the logical to physical sector mapping.

After both the format data block and the operation control block are generated for a given cylinder, control is passed to the 8272 drivers for execution. After the format operation is complete, a SEEK to the next cylinder is performed, a new format table is generated, and another track formatting operation is executed by the drivers. This track formatting continues until all tracks on the diskette are formatted.

In some systems, bad tracks must also be specified when a disk is formatted. For these systems, the existing FORMAT procedure should be modified to format bad tracks with a cylinder number of OFFH.

WRITE

The WRITE procedure transfers a complete sector of data to the disk drive. Five parameters must be supplied to this software module: the drive number (DRV), the cylinder number (CYL), the head/side number (HD), the sector number (SEC) and the recording density (DENS). This procedure generates a disk operation control block (WRITE$DOCB) from these parameters and passes the control block to the 8272 driver for execution. When control returns to the calling program, the data has been transferred to disk.

READ

This procedure is identical to the WRITE procedure except the direction of data transfer is reversed. The READ procedure transfers a sector of data from the floppy disk to system memory.

Coping With Errors

In actual practice all logical disk interface routines would contain error processing mechanisms. (Errors have been ignored for the sake of simplicity in the exerciser programs listed in Appendix B.) A typical error recovery technique consists of a two-stage procedure. First, when an error is detected, a recalibrate operation is performed followed by a retry of the failed operation. This procedure forces the drive to seek directly to the requested cylinder (lowering the probability of a seek error) and attempts to perform the requested operation an additional time. Soft (temporary) errors caused by mechanical or electrical interference do not normally recur during the retry operation; hard errors (caused by media or drive failures), on the other hand, will continue to occur during retry operations. If, after a number of retries (approximately 10), the operation continues to fail, an error message is displayed to the system operator. This error message lists the drive number, type of operation, and failure status (from the FDC). It is the operator's responsibility to take additional action as required.

6. File Systems

The file system provides the disk I/O interface level most familiar to users of interactive microcomputer and minicomputer systems. In a file system, all data is stored in named disk areas called files. The user and applications programs need not be concerned with the exact location of a file on the disk — the disk file system automatically determines the file location from the file name. Files may be created, read, written, modified, and finally deleted (destroyed) when they are no longer needed. Each floppy disk typically contains a directory that lists all the files existing on the disk. A directory entry for a file contains information such as file name, file size, and the disk address (track and sector) of the beginning of the file.

File Allocation

File storage is actually allocated on the disk (by the file system) in fixed size areas called blocks. Normally a block is the same size as a disk sector. Files are created by finding and reserving enough unused blocks to contain the data in the file. Two file allocation methods are currently in widespread use. The first method allocates blocks (for a file) from a sequential pool of unused blocks. Thus, a file is always contained in a set of sequential blocks on the disk. Unfortunately, as files are created, updated, and deleted, these free-block pools become fragmented (separated from one another). When this fragmentation occurs, it often becomes impossible for the file system to create a file even though there is a sufficient number of free blocks on the disk. At this point, special programs must be run to "squeeze" or compact the disk, in order to re-create a single contiguous free-block pool.

The second file allocation method uses a more flexible technique in which individual data blocks may be located anywhere on the disk (with no restrictions). With this technique, a file directory entry contains the disk address of a file pointer block rather than the disk address of the first data block of the file. This file pointer block contains pointers (disk addresses) for each data block in the file. For example, the first pointer in the file pointer block contains the track and sector address of the first data block in the file, the second pointer contains the disk address of the second data block, etc.

In practice, pointer blocks are usually the same size as data blocks. Therefore, some files will require multiple pointer blocks. To accommodate this requirement without loss of flexibility, pointer blocks are linked together, that is, each pointer block contains the disk address of the following pointer block. The last pointer block of the file is signalled by an illegal disk address (e.g., track 0, sector 0 or track OFFH, sector OFFH).

The Intel File System

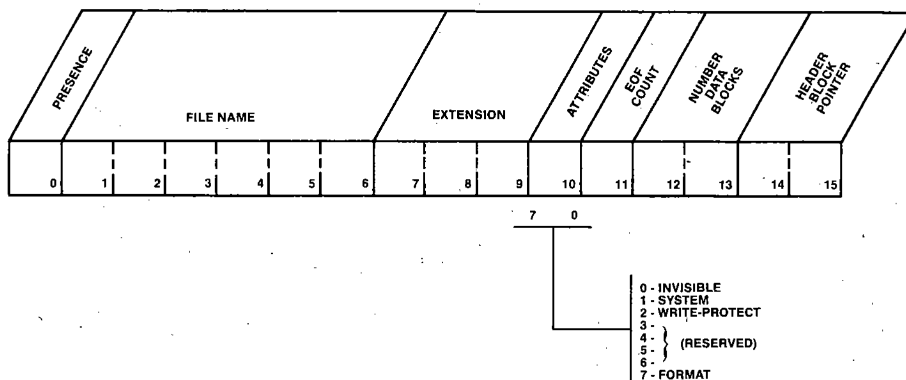
The Intel file system (described in detail in the RMX-80 Users Guide) uses the second disk file allocation method (previously discussed). In order to lower the system overhead involved in finding free data blocks, the Intel file system incorporates a free space management data structure known as a bit map. Each disk sector is represented by a single bit in the bit map. If a bit in the bit map is set to 1, the corresponding disk sector has been allocated. A zero in the bit map indicates that the corresponding sector is free. With this technique, the process of allocating or freeing a sector is accomplished by simply altering the bit map.

File names consist of a basic file name (up to six characters) and a file extension (up to three characters). The basic file name and the file extension are separated by a period (.). Examples of valid file names are: DRIV72.OBJ, XX.TMP, and FILE.CS. In addition, four file attributes are supported (see Figure 3 for attribute definitions).

The bit map and the file directory are placed on prespecified disk tracks (reserved for system use) beginning at track zero.

Disk File System Functions

Table 2 illustrates the typical functions implemented by a disk file system. As an example, the disk directory function (DIR) lists disk file information on the console display terminal. Figure 3 details the contents of a display entry in the Intel file system. The PL/M procedure outlined in Figure 4 illustrates a disk directory algorithm that displays the file name, the file attributes, and the file size (in blocks) for each file in the directory.



Directory Entry

Presence is a flag that can contain one of three values:

000H - The file associated with this entry is present on the disk.

07FH - No file is associated with this entry; the content of the rest of the entry is undefined. The first entry with its flag set to 07FH marks the current logical end of the directory and directory searches stop at this entry.

0FFH - The file named in this entry once existed on the disk but is currently deleted. The next file added to the directory will be placed in the first entry marked 0FFH. This flag cannot, therefore, be used to (reliably) find a file that has been deleted. A value of 0FFH should be thought of as simply marking an open directory entry.

File Name is a string of up to 6 non-blank ASCII characters specifying the name of the file associated with the directory entry. If the file name is shorter than six characters, the remaining bytes contain binary zeros. For example, the name ALPHA would be stored as: 414C50484100H.

Extension is a string of up to 3 non-blank ASCII characters that specifies an extension to the file name. Extensions often identify the type of data in the file such as OBJ (object module), or PLM (PL/M source module). As with the file name, unused positions in the extension field are filled with binary zeros.

Figure 3. Intel Directory Entry Format

Attributes are bits that identify certain characteristics of the file. A 1 bit indicates that the file has the attribute, while a 0 bit means that the file does not have the attribute. The bit positions and their corresponding attributes are listed below (bit 0 is the low-order or rightmost bit, bit 7 is the leftmost bit):

0: Invisible. Files with this attribute are not listed by the ISIS-II DIR command unless the I switch is used. All system files are invisible.

1: System. Files with this attribute are copied to the disk in drive 1 when the S switch is specified with the ISIS-II FORMAT command.

2: Write-Protect. Files with this attribute cannot be opened for output or update, nor can they be deleted or renamed.

3-6: These positions are reserved for future use.

7: Format. Files with this attribute are treated as though they are write-protected. In addition, these files are created on a new diskette when the ISIS-II FORMAT command is issued. The system files all have the FORMAT attribute and it should not be given to any other files.

EOF Count contains the number of the last byte in the last data block of the file. If the value of this field is 080H, for example, the last byte in the file is byte number 128 in the last data block (the last block is full).

Number of Data Blocks is an address variable that indicates the number of data blocks currently used by the file. ISIS-II and the RMX/80 Disk File system both maintain a counter called LENGTH that is the current number of bytes in the file. This is calculated as:

((NUMBER OF DATA BLOCKS - 1) x 128 + EOF COUNT.

Header Block Pointer is the address of the file's header block. The high byte of the field is the sector number and the low byte is the track number. The system "finds" a disk file by searching the directory for the name and then using the header block pointer to seek to the beginning of the file.

Figure 3. Intel Directory Entry Format (Continued)

```
dir: procedure(drv,dens)    public;
  declare  drv              byte,
           dens             byte,
           sector           byte,
           i                byte,
           dir$ptr          byte,
           dir$entry        based rdbptr structure (presence byte,
                            file$name(6) byte,extension(3) byte,
                            attribute byte,eof$count byte,
                            data$blocks address,header$ptr address),
           size (5)         byte,

           invisible$flag   literally '1',
           system$flag      literally '2',
           protected$flag   literally '4',
           format$flag      literally '80H';

  /* The disk directory starts at cylinder 1, sector 2 */
  call seek(drv,1,0);
  do sector=2 to 26;
    call read(drv,1,0,sector,dens);
    do dir$ptr=0 to 112 by 4;
      if dir$entry.presence=7FH then return;
      if dir$entry.presence=0
        then do;
          do i=0 to 5; call co(dir$entry.file$name(i)); end;
          call co(period);
          do i=0 to 2; call co(dir$entry.extension(i)); end;
          do i=0 to 4; call co(space); end;
          call convert$to$decimal(@size,dir$entry.data$blocks);
          do i=0 to 4; call co(size(i)); end;
          If (dir$entry.attribute and invisible$flag) <> 0 then call co('I');
          If (dir$entry.attribute and system$flag) <> 0 then call co('S');
          If (dir$entry.attribute and protected$flag) <> 0 then call co('W');
          If (dir$entry.attribure and format$flag) <> 0 then call co('F');
        end;
    end;
  end;

end dir;
```

AFN-01949A

Figure 4. Sample PL/M Directory Procedure

7. Key 8272 Software Interfacing Considerations

This section contains a quick review of Key 8272 Software design features and issues. (Most items have been mentioned in other sections of this application note.) Before designing 8272 software drivers, it is advisable that the information in this section be thoroughly understood.

1. Non-DMA Data Transfers

In systems that operate without a DMA controller (in the polled or interrupt driven mode), the system software is responsible for counting data transfers to/from the 8272 and generating a TC signal to the FDC when the transfer is complete.

2. Processor Command/Result Phase Interface

In the command phase, the driver software must write the exact number of parameters in the exact order shown in Table 5. During the result phase, the driver must read the complete result status. For example, the Format Track command requires six command bytes and presents seven result bytes. The 8272 will not accept a new command until all result bytes are read. Note that the number of command and result bytes varies from command-to-command. **Command and result phases cannot be shortened.**

During both the command and result phases, the Main Status Register must be read by the driver before each byte of information is read from, or written to, the FDC Data Register. Before each command byte is written, DIO (bit 6) must be low (indicating a data transfer from the processor) and RQM (bit 7) must be high (indicating that the FDC is ready for data). During the result phase, DIO must be high (indicating a data transfer to the processor) and RQM must also be high (indicating that data is ready for the processor).

Note: After the 8272 receives a command byte, the RQM flag may remain set for approximately 16 microseconds (with an 8 MHz clock). The driver should not attempt to read the Main Status Register before this time interval has elapsed; otherwise, the driver may erroneously assume that the FDC is ready to accept the next byte.

3. Sector Sizes

The 8272 does not support 128 byte sectors in the MFM (double-density) mode.

4. Drive Status Changes

The 8272 constantly polls all drives for changes in the drive ready status. This polling begins immediately following RESET. An interrupt is generated every time the FDC senses a change in the drive ready status. After reset, the FDC assumes that all drives are "not ready". If a drive is ready immediately after reset, the 8272 generates a drive status change interrupt.

5. Seek Commands

The 8272 FDC does not perform implied seeks. Before issuing a data read or write command, the read/write head must be positioned over the correct cylinder by means of an explicit seek command. If the head is not positioned correctly, a cylinder address error is generated.

6. Interrupt Processing

When the processor receives an interrupt from the FDC, the FDC may be reporting one of two distinct events:

a) The beginning of the result phase of a previously requested read, write, or scan command.

b) An asynchronous event such as a seek/recalibrate completion, an attention, an abnormal command termination, or an invalid command.

These two cases are distinguished by the FDC BUSY flag (bit 4) in the Main Status Register. If the FDC BUSY flag is high, the interrupt is of type (a). If the FDC BUSY flag is low, the interrupt was caused by an asynchronous event (b).

A single interrupt from the FDC may signal more than one of the above events. After receiving an interrupt, the processor must continue to issue Sense Interrupt Status commands (and service the resulting conditions) until an invalid command code is received. In this manner, all "hidden" interrupts are ferreted out and serviced.

7. Skip Flag (SK)

The skip flag is used during the execution of Read Data, Read Deleted Data, Read Track, and various Scan commands. This flag permits the FDC to skip unwanted sectors on a disk track.

When performing a Read Data, Read Track, or Scan command, a high SK flag indicates that the FDC is to skip over (not transfer) any sector containing a deleted data address mark. A low SK flag indicates that the FDC is to terminate the command (after reading all the data in the sector) when a deleted data address mark is encountered.

When performing a Read Deleted Data command, a high SK flag indicates that sectors containing normal data address marks are to be skipped. Note that this is just the opposite situation from that described in the last paragraph. When a data address mark is encountered during a Read Deleted Data command (and the SK flag is low), the FDC terminates the command after reading all the data in the sector.

8. Bad Track Maintenance

The 8272 does not internally maintain bad track information. The maintenance of this information must be performed by system software. As an example of typical bad track operation, assume that a media test determines that track 31 and track 66 of a given floppy disk are bad. When the disk is formatted for use, the system software formats physical track 0 as logical cylinder 0 (C=0 in the command phase parameters), physical track 1 as logical track 1 (C=1), and so on, until physical track 30 is formatted as logical cylinder 30 (C=30). Physcial track 31 is bad and should be formatted as logical cylinder FF (indicating a bad track). Next, physical track 32 is formatted as logical cylinder 31, and so on, until physioal track 65 is formatted as logical cylinder 64. Next, bad physical track 66 is formatted as logical cylinder FF (another bad track marker), and physical track 67 is formatted as logical cylinder 65. This formatting continues until the last physical track (77) is formatted as logical cylinder 75. Normally, after this formatting is complete, the bad track information is stored in a prespecified area on the floppy disk (typically in a sector on track 0) so that the system will be able to recreate the bad track information when the disk is removed from the drive and reinserted at some later time.

To illustrate how the system software performs a transfer operation on a disk with bad tracks, assume that the disk drive head is positioned at track 0 and the disk described above is loaded into the drive. If a command to read track 36 is issued by an application program, the system software translates this read command into a **seek** to physical track 37 (since there is one bad track between 0 and 36, namely 31) followed by a **read** of logical cylinder 36. Thus, the cylinder parameter C is set to 37 for the Seek command and 36 for the Read Sector command.

REFERENCES

1. Intel, "8272 Single/Double Density Floppy Disk Controller Data Sheet," Intel Corporation, 1980.
2. Intel, "An Intelligent Data Base System Using the 8272," Intel Application Note, AP-116, 1981.
3. Intel, iSBC 208 Hardware Reference Manual, Manual Order No. 143078, Intel Corporation, 1980.
4. Intel, RMX/80 User's Guide, Manual Order No. 9800522, Intel Corporation, 1978
5. Brinch Hansen, P., Operating System Principles, Prentice-Hall, Inc., New Jersey, 1973.
6. Flores, I., Computer Software: Programming Systems for Digital Computers, Prentice-Hall, Inc., New Jersey, 1965.
7. Knuth, D. E., Fundamental Algorithms, Addison-Wesley Publishing Company, Massachusetts, 1975.
8. Shaw, A. C., The Logical Design of Operating Systems, Prentice-Hall, Inc., New Jersey, 1974.
9. Watson, R. W., Time Sharing System Design Concepts, McGraw-Hill, Inc., New York, 1970.
10. Zarrella, J., Operating Systems: Concepts and Principles, Microcomputer Applications, California, 1979.

APPENDIX A
8272 FDC DEVICE DRIVER SOFTWARE

```
PL/M-86 COMPILER    8272 FLOPPY DISK CONTROLLER DEVICE DRIVERS

ISIS-II PL/M-86 V1.2 COMPILATION OF MODULE DRIVERS
OBJECT MODULE PLACED IN :F1:driv72.OBJ
COMPILER INVOKED BY:  plm86 :F1:driv72.p86 DEBUG

              $title('8272 floppy disk controller device drivers')
              $nointvector
              $optimize(2)
              $large

  1           drivers: do;

  2    1      declare
                /* floppy disk port definitions */
                fdc$status$port        literally '30H',        /* 8272 status port */
                fdc$data$port          literally '31H';        /* 8272 data port */

  3    1      declare
               /* floppy disk commands */
                sense$int$status       literally '08H';

  4    1      declare
                /* interrupt definitions */
                fdc$int$level          literally '33';     /* fdc interrupt level */

  5    1      declare
                /* return status and error codes */
                error                  literally '0',
                ok                     literally '1',
                complete               literally '3',
                false                  literally '0',
                true                   literally '1',
                error$in               literally 'not',
                propagate$error        literally 'return error',

                stat$ok                literally '0',          /* fdc operation completed without errors */
                stat$busy              literally '1',          /* fdc is busy, operation cannot be started */
                stat$error             literally '2',          /* fdc operation error */
                stat$command$error     literally '3',          /* fdc not ready for command phase */
                stat$result$error      literally '4',          /* fdc not ready for result phase */
                stat$invalid           literally '5';          /* invalid fdc command */

  6    1      declare
                /* masks */
                busy$mask              literally '10H',
                DIO$mask               literally '40H',
                RQM$mask               literally '80H',
                seek$mask              literally '0FH',
                result$error$mask      literally '0C0H',
                result$drive$mask      literally '03H',
                result$ready$mask      literally '08H';

  7    1      declare
                /* drive numbers */
                max$no$drives          literally '3',
                fdc$general            literally '4';

  8    1      declare
                /* miscellaneous control */
                any$drive$seeking      literally '((input(fdc$status$port) and seek$mask) <> 0)',
                command$code           literally '(docb.disk$command(0) and 1FH)',
                DIO$set$for$input      literally '((input(fdc$status$port) and DIO$mask)=0)',
                DIO$set$for$output     literally '((input(fdc$status$port) and DIO$mask)<>0)',
                extract$drive$no       literally '(docb.disk$command(1) and 03H)',
                fdc$busy               literally '((input(fdc$status$port) and busy$mask) <> 0)',
                no$fdc$error           literally 'possible$error(command$code) and ((docb.disk$result(0)
                                                  and result$error$mask) = 0)',
                wait$for$op$complete   literally 'do while not operation$complete(drive$no); end',
                wait$for$RQM           literally 'do while (input(fdc$status$port) and RQM$mask) = 0; end;';

  9    1      declare
                /* structures */
                docb$type              literally              /* disk operation control block */
                     '(dma$op byte,dma$addr word, dma$addr$ext byte,dma$count word,
                      disk$command(9) byte,disk$result(7) byte,misc byte)';

              $eject
 10    1      declare
                drive$status$change(4) byte public,         /* when set - indicates that drve status changed
                drive$ready(4) byte public;                 /* current status of drives */
```

```
11  1    declare
           operation$in$progress(5) byte,          /* internal flags for operation with multiple drives */
           operation$complete(5) byte,             /* fdc execution phase completed */
           operation$docb$ptr(5) pointer,          /* pointers for operations in progress */
           interrupt$docb structure docb$type,     /* temporary docb for interrupt processing */
           global$drive$no byte;                   /* drive number of non-overlapped operation
                                                      in progress - if any */

12  1    declare
           /* internal vectors that contain command operational information */
           no$result(32) byte                      /* no result phase to command */
             data(0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0),
           immed$result(32) byte                   /* immediate result phase for command */
             data(0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0),
           overlap$operation(32) byte              /* command permits overlapped operation of drvies */
             data(0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0),
           drive$no$present(32) byte               /* drive number present in command information */
             data(0,0,1,0,1,1,1,1,0,1,1,0,1,1,0,1,0,1,0,0,0,0,0,0,0,1,0,0,0,1,0,0),
           possible$error(32) byte                 /* determines if command can return with an error */
             data(0,0,1,0,0,1,1,1,1,1,1,0,1,1,0,1,0,1,0,0,0,0,0,0,0,1,0,0,0,1,0,0),
           command$length(32) byte                 /* contains number of command bytes for each command */
             data(0,0,9,3,2,9,9,2,1,9,2,0,9,6,0,3,0,9,0,0,0,0,0,0,0,9,0,0,0,9,0,0),
           valid$command(32) byte                  /* flags invalid command codes */
             data(0,0,1,1,1,1,1,1,1,1,1,0,1,1,0,1,0,1,0,0,0,0,0,0,0,1,0,0,0,1,0,0);

         $eject

         /**** initialization for the 8272 fdc driver software.  This procedure must
               be called prior to execution of any driver software.     ****/

13  1    initialize$drivers: procedure public;
         /* initialize 8272 drivers */
14  2      declare drv$no byte;

15  2      do drv$no=0 to max$no$drives;
16  3        drive$ready(drv$no)=false;
17  3        drive$status$change(drv$no)=false;
18  3        operation$in$progress(drv$no)=false;
19  3        operation$complete(drv$no)=false;
20  3      end;

21  2      operation$in$progress(fdc$general)=false;
22  2      operation$complete(fdc$general)=false;
23  2      global$drive$no=0;

24  2    end initialize$drivers;

         /**** wait until the 8272 fdc is ready to receive command/parameter bytes
               in the command phase.  The 8272 is ready to receive command bytes
               when the RQM flag is high and the DIO flag is low.     ****/

25  1    fdc$ready$for$command: procedure byte;

           /* wait for valid flag settings in status register */
26  2      call time(1);

           /* wait for "master request" flag */
27  2      wait$for$RQM;

           /* check data direction flag */
30  2      if DIO$set$for$input
             then return ok;
32  2        else return error;

33  2    end fdc$ready$for$command;

         /**** wait until the 8272 fdc is ready to return data bytes in the result
               phase.  The 8272 is ready to return a result byte when the RQM and DIO
               flags are both high.  The busy flag in the main status register will
               remain set until the last data byte of the result phase has been read
               by the processor.     ****/

34  1    fdc$ready$for$result: procedure byte;

           /* wait for valid settings in status register */
35  2      call time(1);

           /* result phase has ended when the 8272 busy flag is reset */
36  2      if not fdc$busy
             then return complete;
```

```
                  /* wait for "master request" flag */
38    2           wait$for$RQM;

                  /* check data direction flag */
41    2           if DIO$set$for$output
                    then return ok;
43    2             else return error;

44    2         end fdc$ready$for$result;


                /**** output a single command/parameter byte to the 8272 fdc.  The "data$byte"
                      parameter is the byte to be output to the fdc.    ****/

45    1         output$byte$to$fdc: procedure(data$byte) byte;
46    2           declare data$byte byte;

                  /* check to see if fdc is ready for command */
47    2           if not fdc$ready$for$command
                    then propagate$error;

49    2           output(fdc$data$port)=data$byte;

50    2           return ok;
51    2         end output$byte$to$fdc;


                /**** input a single result byte from the 8272 fdc.  The "data$byte$ptr"
                      parameter is a pointer to the memory location that is to contain
                      the input byte.    ****/

52    1         input$byte$from$fdc: procedure(data$byte$ptr) byte;
53    2           declare data$byte$ptr pointer;
54    2           declare
                    data$byte based data$byte$ptr byte,
                    status byte;

                  /* check to see if fdc is ready */
55    2           status=fdc$ready$for$result;
56    2           if error$in status
                    then propagate$error;

                  /* check for result phase complete */
58    2           if status=complete
                    then return complete;

60    2           data$byte=input(fdc$data$port);
61    2           return ok;
62    2         end input$byte$from$fdc;

                $eject

                /**** output the dma mode, the dma address, and the dma word count to the
                      8237 dma controller.  Also output the high order four bits of the
                      address to the address extension latch.  Finally, start the disk
                      dma channel.  The "docb$ptr" parameter is a pointer to the appropriate
                      disk operation control block.    ****/

63    1         output$controls$to$dma: procedure(docb$ptr);
64    2           declare docb$ptr pointer;
65    2           declare docb based docb$ptr structure docbtype;

66    2           declare
                    /* dma port definitions */
                    dma$upper$addr$port    literally '10H',        /* upper 4 bits of current address */
                    dma$disk$addr$port     literally '00H',        /* current address port */
                    dma$disk$word$count    literally '01H',        /* word count port */
                    dma$command$port       literally '08H',        /* command port */
                    dma$mode$port          literally '0BH',        /* mode port */
                    dma$mask$sr$port       literally '0AH',        /* mask set/reset port */
                    dma$clear$ff$port      literally '0CH',        /* clear first/last flip-flop port */
                    dma$master$clear$port  literally '0DH',        /* dma master clear port */
                    dma$mask$port          literally '0FH',        /* parallel mask set port*/

                    dma$disk$chan$start    literally '00H',        /* dma mask to start disk channel */
                    dma$extended$write     literally 'shl(1,5)',   /* extended write flag */
                    dma$single$transfer    literally 'shl(1,6)';   /* single transfer flag */

67    2           if docb.dma$op < 3
                    then do;
                      /* set dma mode and clear first/last flip-flop */
69    3               output(dma$mode$port)=shl(docb.dma$op,2) or 40H;
70    3               output(dma$clear$ff$port)=0;
```

```
                    /* set dma address */
 71    3            output(dma$disk$addr$port)=low(docb.dma$addr);
 72    3            output(dma$disk$addr$port)=high(docb.dma$addr);
 73    3            output(dma$upper$addr$port)=docb.dma$addr$ext;

                    /* output disk transfer word count to dma controller */
 74    3            output(dma$disk$word$count)=low(docb.dma$count);
 75    3            output(dma$disk$word$count)=high(docb.dma$count);

                    /* start dma channel 0 for fdc */
 76    3            output(dma$mask$sr$port)=dma$disk$chan$start;
 77    3          end;

 78    2        end output$controls$to$dma;

              /**** output a high-level disk command to the 8272 fdc.  The number of bytes
                    required for each command is contained in the "command$length" table.
                    The "docb$ptr" parameter is a pointer to the appropriate disk operation
                    control block.    ****/

 79    1        output$command$to$fdc: procedure(docb$ptr) byte;
 80    2          declare docb$ptr pointer;

 81    2          declare
                    docb based docb$ptr structure docb$type,
                    cmd$byte$no byte;

 82    2          disable;

                  /* output all command bytes to the fdc */
 83    2          do cmd$byte$no=0 to command$length(command$code)-1;
 84    3            if error$in output$byte$to$fdc(docb.disk$command(cmd$byte$no))
                      then do; enable; propagate$error; end;
 89    3          end;

 90    2          enable;
 91    2          return ok;
 92    2        end output$command$to$fdc;

              /**** input the result data from the 8272 fdc during the result phase (after
                    command execution).  The "docb$ptr" parameter is a pointer to the
                    appropriate disk operation control block.    ****/

 93    1        input$result$from$fdc: procedure(docb$ptr) byte;
 94    2          declare docb$ptr pointer;
 95    2          declare
                    docb based docb$ptr structure docb$type,
                    result$byte$no byte,
                    temp byte,
                    status byte;

 96    2          disable;

 97    2          do result$byte$no=0 to 7;
 98    3            status=input$byte$from$fdc(@temp);
 99    3            if error$in status
                      then do; enable; propagate$error; end;
104    3            if status=complete
                      then do; enable; return ok; end;
109    3            docb.disk$result(result$byte$no)=temp;
110    3          end;

111    2          enable;
112    2          if fdc$busy
                    then return error;
114    2            else return ok;
115    2        end input$result$from$fdc;

              /**** cleans up after the execution of a disk operation that has no result
                    phase.  The procedure is also used after some disk operation errors.
                    "drv" is the drive number, and "cc" is the command code for the
                    disk operation.    ****/

116    1        operation$clean$up: procedure(drv,cc);
117    2          declare (drv,cc) byte;

118    2          disable;
119    2          operation$in$progress(drv)=false;
```

```
 120   2         if not overlap$operation(cc)
                   then global$drive$no=0;
 122   2         enable;

 123   2       end operation$clean$up;

               $eject

               /**** execute the disk operation control block specified by the pointer
                     parameter "docb$ptr".  The "status$ptr" parameter is a pointer to
                     a byte variable that is to contain the status of the requested
                     operation when it has been completed.  Six status conditions are
                     possible on return:

                       0    The specified operation was completed without error.
                       1    The fdc is busy and the requested operation cannot be started.
                       2    Fdc error (further information is contained in the result
                            storage portion of the disk operation control block - as
                            described in the 8272 data sheet).
                       3    Transfer error during output of the command bytes to the fdc.
                       4    Transfer error during input of the result bytes from the fdc.
                       5    Invalid fdc command.    ****/

 124   1       execute$docb: procedure(docb$ptr,status$ptr) public;
               /* execute a disk operation control block */

 125   2         declare docb$ptr pointer, status$ptr pointer;
 126   2         declare
                   docb based docb$ptr structure docb$type,
                   status based status$ptr byte,
                   drive$no byte;

                 /* check command validity */
 127   2         if not valid$command(command$code)
                   then do; status=stat$invalid; return; end;

                 /* determine if command has a drive number field - if not, set the drive
                    number for a general fdc command */
 132   2         if drive$no$present(command$code)
                   then drive$no=extract$drive$no;
 134   2           else drive$no=fdc$general;

                 /* an overlapped operation can not be performed if the fdc is busy */
 135   2         if overlap$operation(command$code) and fdc$busy
                   then do; status=stat$busy; return; end;

                 /* for a non-overlapped operation, check fdc busy or any drive seeking */
 140   2         if not overlap$operation(command$code) and (fdc$busy or any$drive$seeking)
                   then do; status=stat$busy; return; end;

                 /* check for drive operation in progress - if none, set flag and start operation */
 145   2         disable;
 146   2         if operation$in$progress(drive$no)
                   then do; enable; status=stat$busy; return; end;
 152   2           else operation$in$progress(drive$no)=true;

                 /* at this point, an fdc operation is about to begin, so:
                    1. reset the operation complete flag
                    2. set the docb pointer for the current operation
                    3. if this is not an overlapped operation, set the global drive
                       number for the subsequent result phase interrupt.  */
 153   2         operation$complete(drive$no)=0;
 154   2         operation$docb$ptr(drive$no)=docb$ptr;

 155   2         if not overlap$operation(command$code)
                   then global$drive$no=drive$no+1;
 157   2         enable;

 158   2         call output$controls$to$dma(docb$ptr);
 159   2         if error$in output$command$to$fdc(docb$ptr)
                   then do;
 161   3             call operation$clean$up(drive$no,command$code);
 162   3             status=stat$command$error;
 163   3             return;
 164   3           end;

                 /* return immediately if the command has no result phase or completion interrupt - specify */
 165   2         if no$result(command$code)
                   then do;
 167   3             call operation$clean$up(drive$no,command$code);
 168   3             status=stat$ok;
 169   3             return;
 170   3           end;
```

```
171   2          if immed$result(command$code)
                   then do;
173   3              if error$in input$result$from$fdc(docb$ptr)
                       then do;
175   4                  call operation$clean$up(drive$no,command$code);
176   4                  status=stat$result$error;
177   4                  return;
178   4                end;
179   3            end;
180   2            else do;
181   3              wait$for$op$complete;
183   3              if docb.misc = error
                       then do; status=stat$result$error; return; end;
188   3            end;

189   2          if no$fdc$error
                   then status=stat$ok;
191   2            else status=stat$error;

192   2        end execute$docb;

               $eject

               /**** copy disk command results from the interrupt control block to the
                     currently active disk operation control block if a disk operation is
                     in progress.    ****/

193   1        copy$int$result: procedure(drv);
194   2          declare drv byte;
195   2          declare
                   i byte,
                   docb$ptr pointer,
                   docb based docb$ptr structure docb$type;

196   2          if operation$in$progress(drv)
                   then do;
198   3              docb$ptr=operation$docb$ptr(drv);
199   3              do i=1 to 6; docb.disk$result(i)=interrupt$docb.disk$result(i); end;
202   3              docb.misc=ok;
203   3              operation$in$progress(drv)=false;
204   3              operation$complete(drv)=true;
205   3            end;

206   2        end copy$int$result;

               /**** interrupt processing for 8272 fdc drivers.  Basically, two types of
                     interrupts are generated by the 8272: (a)when the execution phase of
                     an operation has been completed, an interrupt is generated to signal
                     the beginning of the result phase (the fdc busy flag is set
                     when this interrupt is received), and (b) when an overlapped operation
                     is completed or an unexpected interrupt is received (the fdc busy flag
                     is not set when this interrupt is received).

                     When interrupt type (a) is received, the result bytes from the operation
                     are read from the 8272 and the operation complete flag is set.

                     When an interrupt of type (b) is received, the interrupt result code is
                     examined to determine which of the following four actions are indicated:

                       1. An overlapped option (recalibrate or seek) has been completed.  The
                          result data is read from the 8272 and placed in the currently active
                          disk operation control block.
                       2. An abnormal termination of an operation has occurred.  The result
                          data is read and placed in the currently active disk operation
                          control block.
                       3. The execution of an invalid command has been attempted.  This
                          signals the successful completion of all interrupt processing.
                       4. The ready status of a drive has changed.  The "drive$ready" and
                          "drive$ready$status" change tables are updated.  If an operation
                          is currently in progress on the affected drive, the result data
                          is placed in the currently active disk operation control block.

                     After an interrupt is processed, additional sense interrupt status commands
                     must be issued and processed until an invalid command result is returned
                     from the fdc.  This action guarantees that all "hidden" interrupts
                     are serviced.    ****/
```

```
207   1       fdcint: procedure public interrupt fdc$int$level;
208   2         declare
                  invalid byte,
                  drive$no byte,
                  docb$ptr pointer,
                  docb based docb$ptr structure docb$type;

209   2         declare
                  /* interrupt port definitions */
                  ocw2                        literally '70H',
                  nseoi                       literally 'shl(1,5)';

210   2         declare
                  /* miscellaneous flags */
                  result$code                 literally 'shr(interrupt$docb.disk$result(0) and result$error$mask,6)',
                  result$drive$ready          literally '((interrupt$docb.disk$result(0) and result$ready$mask) = 0)',
                  extract$result$drive$no     literally '(interrupt$docb.disk$result(0) and result$drive$mask)',
                  end$of$interrupt            literally 'output(ocw2)=nseoi';

                /* if the fdc is busy when an interrupt is received, then the result
                   phase of the previous non-overlapped operation has begun */
211   2         if fdc$busy
                  then do;
                    /* process interrupt if operation in progress */
213   3             if global$drive$no <> 0
                      then do;
215   4                 docb$ptr=operation$docb$ptr(global$drive$no-1);
216   4                 if error$in input$result$from$fdc(docb$ptr)
                          then docb.misc=error;
218   4                   else docb.misc=ok;
219   4                 operation$in$progress(global$drive$no-1)=false;
220   4                 operation$complete(global$drive$no-1)=true;
221   4                 global$drive$no=0;
222   4               end;
223   3           end;

                /* if the fdc is not busy, then either an overlapped operation has been
                   completed or an unexpected interrupt has occurred (e.g., drive status
                   change) */
224   2           else do;
225   3             invalid=false;
226   3             do while not invalid;

                      /* perform a sense interrupt status operation - if errors are detected,
                         in the actual fdc interface, interrupt processing is discontinued */
227   4               if error$in output$byte$to$fdc(sense$int$status) then go to ignore;
229   4               if error$in input$result$from$fdc(@interrupt$docb) then go to ignore;

231   4               do case result$code;

                        /* case 0 - operation complete */
232   5                 do;
233   6                   drive$no=extract$result$drive$no;
234   6                   call copy$int$result(drive$no);
235   6                 end;

                        /* case 1 - abnormal termination */

236   5                 do;
237   6                   drive$no=extract$result$drive$no;
238   6                   call copy$int$result(drive$no);
239   6                 end;

                        /* case 2 - invalid command */
240   5                 invalid=true;

                        /* case 3 - drive ready change */
241   5                 do;
242   6                   drive$no=extract$result$drive$no;
243   6                   call copy$int$result(drive$no);
244   6                   drive$status$change(drive$no)=true;
245   6                   if result$drive$ready
                            then drive$ready(drive$no)=true;
247   6                     else drive$ready(drive$no)=false;
248   6                 end;
249   5               end;
250   4             end;
251   3           end;

252   2         ignore: end$of$interrupt;
253   2       end fdcint;

254   1       end drivers;
```

```
MODULE INFORMATION:
     CODE AREA SIZE     = 0615H   1557D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 0050H     80D
     MAXIMUM STACK SIZE = 0032H     50D
     564 LINES READ
     0 PROGRAM ERROR(S)
END OF PL/M-86 COMPILATION
```

APPENDIX B
8272 FDC EXERCISER PROGRAM

```
PL/M-86 COMPILER    8272 FLOPPY DISK DRIVER EXERCISE PROGRAM

ISIS-II PL/M-86 V1.2 COMPILATION OF MODULE RUN72
OBJECT MODULE PLACED IN :F1:run72.OBJ
COMPILER INVOKED BY:  plm86 :F1:run72.p86 DEBUG

             $title ('8272 floppy disk driver exercise program')
             $nointvector
             $optimize(2)
             $large
  1          run72: do;

  2   1        declare
                 docb$type                literally         /* disk operation control block */
                   '(dma$op byte,dma$addr word,dma$addr$ext byte,dma$count word,
                    disk$command(9) byte,disk$result(7) byte,misc byte)';

  3   1        declare
                 /* 8272 fdc commands */
                 fm                       literally '0',
                 mfm                      literally '1',
                 dma$mode                 literally '0',
                 non$dma$mode             literally '1',
                 recalibrate$command      literally '7',
                 specify$command          literally '3',
                 read$command             literally '6',
                 write$command            literally '5',
                 format$command           literally '0DH',
                 seek$command             literally '0FH';

  4   1        declare
                 dma$verify               literally '0',
                 dma$read                 literally '1',
                 dma$write                literally '2',
                 dma$noop                 literally '3';

  5   1        declare
                 /* disk operation control blocks */
                 format$docb              structure docb$type,
                 seek$docb                structure docb$type,
                 recalibrate$docb         structure docb$type,
                 specify$docb             structure docb$type,
                 read$docb                structure docb$type,
                 write$docb               structure docb$type;

  6   1        declare
                 step$rate                byte,
                 head$load$time           byte,
                 head$unload$time         byte,
                 filler$byte              byte,
                 operation$status         byte,
                 interleave               byte,
                 format$gap               byte,
                 read$write$gap           byte,
                 index                    byte,
                 drive                    byte,
                 density                  byte,
                 multitrack               byte,
                 sector                   byte,
                 cylinder                 byte,
                 head                     byte,             /* disk drive head */
                 tracks$per$disk          byte,
                 sectors$per$track        byte,
                 bytes$per$sector$code    byte,
                 bytes$per$sector         word;             /* number of bytes in a sector on the disk */

  7   1        declare
                 /* read and write buffers */
                 fmtblk(104)              byte public,
                 wrbuf(1024)              byte public,
                 rdbuf(1024)              byte public;

  8   1        declare
                 /* disk format initialization tables */
                 sec$trk$table(3)         byte data(26,15,8),
                 fmt$gap$table(8)         byte data(1BH,2AH,3AH,0,0,36H,54H,74H),
                 rd$wr$gap$table(8)       byte data(07H,0EH,1BH,0,0,0EH,1BH,35H);
```

```
  9   1       declare
                /* external pointer tables and interrupt vector */
                rdbptr(2)              word external,
                wrbptr(2)              word external,
                fbptr(2)               word external,
                intptr(2)              word external,
                intvec(80H)            word external;


 10   1       execute$docb: procedure(docb$ptr,status$ptr) external;
 11   2         declare docb$ptr pointer, status$ptr pointer;
 12   2       end execute$docb;

 13   1       initialize$drivers: procedure external;
 14   2       end initialize$drivers;

            $eject

            /**** specify step rate ("srt"), head load time ("hlt"), head unload time ("hut"),
                  and dma or non-dma operation ("nd").   ****/

 15   1       specify: procedure(srt,hlt,hut,nd);
 16   2         declare (srt,hlt,hut,nd) byte;

 17   2         specify$docb.dma$op=dma$noop;
 18   2         specify$docb.disk$command(0)=specify$command;
 19   2         specify$docb.disk$command(1)=shl((not srt)+1,4) or shr(hut,4);
 20   2         specify$docb.disk$command(2)=(hlt and 0FEH) or (nd and 1);
 21   2         call execute$docb(@specify$docb,@operation$status);

 22   2       end specify;


            /****  recalibrate disk drive
                   8272 automatically steps out until the track 0 signal is activated
                   by the disk drive.    ****/

 23   1       recalibrate: procedure(drv);
 24   2         declare drv byte;

 25   2         recalibrate$docb.dma$op=dma$noop;
 26   2         recalibrate$docb.disk$command(0)=recalibrate$command;
 27   2         recalibrate$docb.disk$command(1)=drv;
 28   2         call execute$docb(@recalibrate$docb,@operation$status);

 29   2       end recalibrate;


            /**** seek drive "drv", head (side) "hd" to cylinder "cyl".    ****/

 30   1       seek: procedure(drv,cyl,hd);
 31   2         declare (drv,cyl,hd) byte;

 32   2         seek$docb.dma$op=dma$noop;
 33   2         seek$docb.disk$command(0)=seek$command;
 34   2         seek$docb.disk$command(1)=drv or shl(hd,2);
 35   2         seek$docb.disk$command(2)=cyl;
 36   2         call execute$docb(@seek$docb,@operation$status);

 37   2       end seek;


            /**** format a complete side ("head") of a single floppy disk in drive "drv".  The density,
                  (single or double) is specified by flag "dens".    ****/

 38   1       format: procedure(drv,dens,intlve);
              /*  format disk */
 39   2         declare (drv,dens,intlve) byte;
 40   2         declare physical$sector byte;

 41   2         call recalibrate(drv);
 42   2         do cylinder=0 to tracks$per$disk-1;
                  /* set sector numbers in format block to zero before computing interleave */

 43   3           do physical$sector=1 to sectors$per$track; fmtblk((physical$sector-1)*4+2)=0; end;
                  /* physical sector 1 equals logical sector 1 */
 46   3           physical$sector=1;

                  /* assign interleaved sectors */
 47   3           do sector=1 to sectors$per$track;
 48   4             index=(physical$sector-1)*4;
```

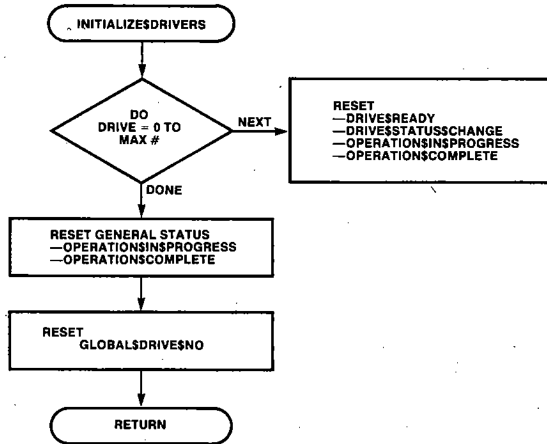
```
                      /* change sector and index if sector has already been assigned */
 49   4               do while fmtblk(index+2) <> 0; index=index+4; physical$sector=physical$sector+1; end;

                      /* set cylinder, head, sector, and size code for current sector into table */
 53   4               fmtblk(index)=cylinder;
 54   4               fmtblk(index+1)=head;
 55   4               fmtblk(index+2)=sector;
 56   4               fmtblk(index+3)=bytes$per$sector$code;

                      /* update physical sector number by interleave */
 57   4               physical$sector=physical$sector+intlve;
 58   4               if physical$sector > sectors$per$track
                        then physical$sector=physical$sector-sectors$per$track;
 60   4             end;

                    /* seek to next cylinder */
 61   3             call seek(drv,cylinder,head);

                    /* set up format control block */
 62   3             format$docb.dma$op=dma$write;
 63   3             format$docb.dma$addr=fbptr(0)+shl(fbptr(1),4);
 64   3             format$docb.dma$addr$ext=0;
 65   3             format$docb.dma$count=sectors$per$track*4-1;
 66   3             format$docb.disk$command(0)=format$command or shl(dens,6);
 67   3             format$docb.disk$command(1)=drv or shl(head,2);
 68   3             format$docb.disk$command(2)=bytes$per$sector$code;
 69   3             format$docb.disk$command(3)=sectors$per$track;
 70   3             format$docb.disk$command(4)=format$gap;
 71   3             format$docb.disk$command(5)=filler$byte;
 72   3             call execute$docb(@format$docb,@operation$status);
 73   3           end;

 74   2         end format;

            /**** write sector "sec" on drive "drv" at head "hd" and cylinder "cyl".  The
                  disk recording density is specified by the "dens" flag.  Data is expected to be
                  in the global write buffer ("wrbuf").    ****/

 75   1         write: procedure(drv,cyl,hd,sec,dens);
 76   2           declare (drv,cyl,hd,sec,dens) byte;

 77   2           write$docb.dma$op=dma$write;
 78   2           write$docb.dma$addr=wrbptr(0)+shl(wrbptr(1),4);
 79   2           write$docb.dma$addr$ext=0;
 80   2           write$docb.dma$count=bytes$per$sector-1;
 81   2           write$docb.disk$command(0)=write$command or shl(dens,6) or shl(multitrack,7);
 82   2           write$docb.disk$command(1)=drv or shl(hd,2);
 83   2           write$docb.disk$command(2)=cyl;
 84   2           write$docb.disk$command(3)=hd;
 85   2           write$docb.disk$command(4)=sec;
 86   2           write$docb.disk$command(5)=bytes$per$sector$code;
 87   2           write$docb.disk$command(6)=sectors$per$track;
 88   2           write$docb.disk$command(7)=read$write$gap;
 89   2           if bytes$per$sector$code = 0
                     then write$docb.disk$command(8)=bytes$per$sector;
 91   2             else write$docb.disk$command(8)=0FFH;
 92   2           call execute$docb(@write$docb,@operation$status);

 93   2         end write;

            /**** read sector "sec" on drive "drv" at head "hd" and cylinder "cyl".  The
                  disk recording density is defined by the "dens" flag.  Data is read into
                  the global read buffer ("rdbuf").    ****/

 94   1         read: procedure(drv,cyl,hd,sec,dens);
 95   2           declare (drv,cyl,hd,sec,dens) byte;

 96   2           read$docb.dma$op=dma$read;
 97   2           read$docb.dma$addr=rdbptr(0)+shl(rdbptr(1),4);
 98   2           read$docb.dma$addr$ext=0;
 99   2           read$docb.dma$count=bytes$per$sector-1;
100   2           read$docb.disk$command(0)=read$command or shl(dens,6) or shl(multitrack,7);
101   2           read$docb.disk$command(1)=drv or shl(hd,2);
102   2           read$docb.disk$command(2)=cyl;
103   2           read$docb.disk$command(3)=hd;
104   2           read$docb.disk$command(4)=sec;
105   2           read$docb.disk$command(5)=bytes$per$sector$code;
106   2           read$docb.disk$command(6)=sectors$per$track;
107   2           read$docb.disk$command(7)=read$write$gap;
```

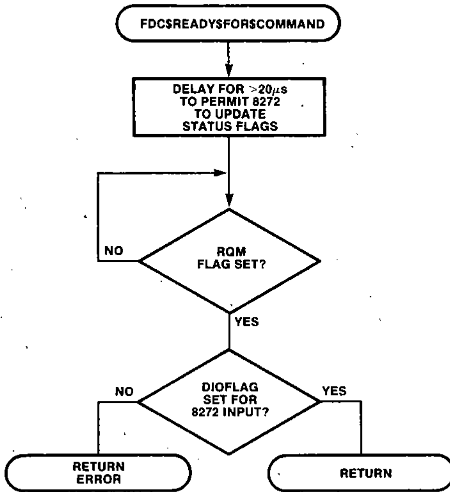
```
108    2          if bytes$per$sector$code = 0
                    then read$docb.disk$command(8)=bytes$per$sector;
110    2            else read$docb.disk$command(8)=0FFH;
111    2          call execute$docb(@read$docb,@operation$status);

112    2        end read;

              $eject

              /**** initialize system by setting up 8237 dma controller and 8259A interrupt
                    controller.    ****/

113    1        initialize$system: procedure;
114    2          declare
                    /* I/O ports */
                    dma$disk$addr$port          literally '00H',       /* current address port */
                    dma$disk$word$count$port literally '01H',          /* word count port */
                    dma$command$port            literally '08H',       /* command port */
                    dma$mode$port               literally '0BH',       /* mode port */
                    dma$mask$sr$port            literally '0AH',       /* mask set/reset port */
                    dma$clear$ff$port           literally '0CH',       /* clear first/last flip-flop port */
                    dma$master$clear$port       literally '0DH',       /* dma master clear port */
                    dma$mask$port               literally '0FH',       /* parallel mask set port*/
                    dma$c1$addr$port            literally '02H',
                    dma$c1$word$count$port      literally '03H',
                    dma$c2$addr$port            literally '04H',
                    dma$c2$word$count$port      literally '05H',
                    dma$c3$addr$port            literally '06H',
                    dma$c3$word$count$port      literally '07H',
                    icw1                        literally '70H',
                    icw2                        literally '71H',
                    icw4                        literally '71H',
                    ocw1                        literally '71H',
                    ocw2                        literally '70H',
                    ocw3                        literally '70H';

115    2          declare
                    /* misc masks and literals */
                    dma$extended$write          literally 'shl(1,5)',  /* extended write flag */
                    dma$single$transfer         literally 'shl(1,6)',  /* single transfer flag */
                    dma$disk$mode               literally '40H',
                    dma$c1$mode                 literally '41H',
                    dma$c2$mode                 literally '42H',
                    dma$c3$mode                 literally '43H',
                    mode$8088                   literally '1',
                    interrupt$base              literally '20H',
                    single$controller           literally 'shl(1,1)',
                    level$sensitive             literally 'shl(1,3)',
                    control$word$4$required     literally '1',
                    base$icw1                   literally '10H',
                    mask$all                    literally '0FFH',
                    disk$interrupt$mask         literally '1';

116    2          output(dma$master$clear$port)=0;             /* master reset */
117    2          output(dma$mode$port)=dma$extended$write;    /* set dma command mode */

                  /* set all dma registers to valid values */
118    2          output(dma$mask$port)=mask$all;              /* mask all channels */

                  /* set all addresses to zero */
119    2          output(dma$clear$ff$port)=0;                 /* reset first/last flip-flop */
120    2          output(dma$disk$addr$port)=0;
121    2          output(dma$disk$addr$port)=0;
122    2          output(dma$c1$addr$port)=0;
123    2          output(dma$c1$addr$port)=0;
124    2          output(dma$c2$addr$port)=0;
125    2          output(dma$c2$addr$port)=0;
126    2          output(dma$c3$addr$port)=0;
127    2          output(dma$c3$addr$port)=0;

                  /* set all word counts to valid values */
128    2          output(dma$clear$ff$port)=0;                 /* reset first/last flip-flop */
129    2          output(dma$disk$word$count$port)=1;
130    2          output(dma$disk$word$count$port)=1;
131    2          output(dma$c1$word$count$port)=1;
132    2          output(dma$c1$word$count$port)=1;
133    2          output(dma$c2$word$count$port)=1;
134    2          output(dma$c2$word$count$port)=1;
135    2          output(dma$c3$word$count$port)=1;
136    2          output(dma$c3$word$count$port)=1;
```

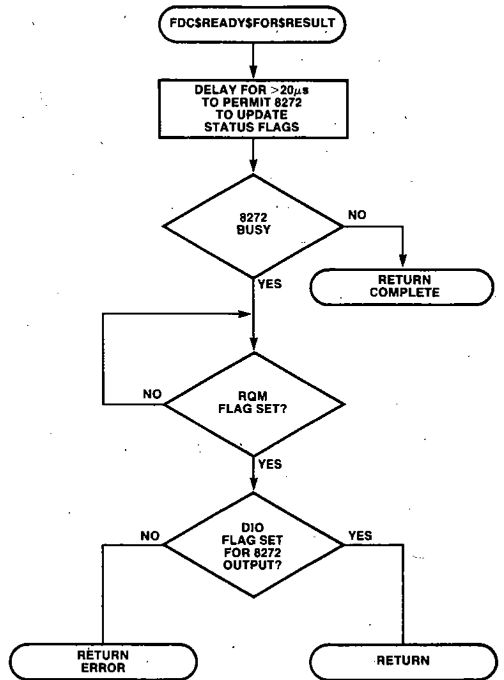
```
                  /* initialize all dma channel modes */
137    2          output(dma$mode$port)=dma$disk$mode;
138    2          output(dma$mode$port)=dma$c1$mode;
139    2          output(dma$mode$port)=dma$c2$mode;
140    2          output(dma$mode$port)=dma$c3$mode;

                  /* initialize 8259A interrupt controller */
141    2          output(icw1)=single$controller or level$sensitive or control$word4$required or base$icw1;
142    2          output(icw2)=interrupt$base;
143    2          output(icw4)=mode$8088;                 /* set 8088 interrupt mode */
144    2          output(ocw1)=not disk$interrupt$mask;   /* mask all interrupts except disk */

                  /* initialize interrupt vector for fdc */
145    2          intvec(40H)=intptr(0);
146    2          intvec(41H)=intptr(1);

147    2        end initialize$system;
              $eject

              /**** main program:  first format disk (all tracks on side (head) 0.  Then
                    read each sector on every track of the disk forever.    ****/

148    1        declare drive$ready(4) byte external;

                /* disable until interrupt vector setup and initialization complete */
149    1        disable;

                /* set initial floppy disk parameters */
150    1        density=mfm;                             /* double-density */
151    1        head=0;                                  /* single sided */
152    1        multitrack=0;                            /* no multitrack operation */
153    1        filler$byte=55H;                         /* for format */
154    1        tracks$per$disk=77;                      /* normal floppy disk drive */
155    1        bytes$per$sector=1024;                   /* 1024 bytes in each sector */
156    1        interleave=6;                            /* set track interleave factor */
157    1        step$rate=11;                            /* 10ms for SA800 plus 1 for uncertainty */
158    1        head$load$time=40;                       /* 40ms head load for SA800 */
159    1        head$unload$time=240;                    /* keep head loaded as long as possible */

                /* derive dependent parameters from those above */
160    1        bytes$per$sector$code=shr(bytes$per$sector,7);
161    1        do index=0 to 3;
162    2          if (bytes$per$sector$code and 1) <> 0
                    then do; bytes$per$sector$code=index; go to donebc; end;
167    2            else bytes$per$sector$code=shr(bytes$per$sector$code,1);
168    2        end;

169    1        donebc:
                sectors$per$track=sec$trk$table(bytes$per$sector$code-density);
170    1        format$gap=fmt$gap$table(shl(density,2)+bytes$per$sector$code);
171    1        read$write$gap=rd$wr$gap$table(shl(density,2)+bytes$per$sector$code);

                /* initialize system and drivers */
172    1        call initialize$system;
173    1        call initialize$drivers;

                /* reenable interrupts and give 8272 a chance to report on drive status
                   before proceeding */
174    1        enable;
175    1        call time(10);

                /* specify disk drive parameters */
176    1        call specify(step$rate,head$load$time,head$unload$time,dma$mode);

177    1        drive=0;                                 /* run single disk drive #0 */

                /* wait until drive ready */
178    1        do while 1;
179    2          if drive$ready(drive)
                    then go to start;
181    2        end;

182    1        start:
                call format(drive,density,interleave);

183    1        do while 1;
184    2          do cylinder=0 to tracks$per$disk-1;
185    3            call seek(drive,cylinder,head);
186    3            do sector=1 to sectors$per$track;

                      /* set up write buffer */
187    4              do index=0 to bytes$per$sector-1; wrbuf(index)=index+sector+cylinder; end;
```

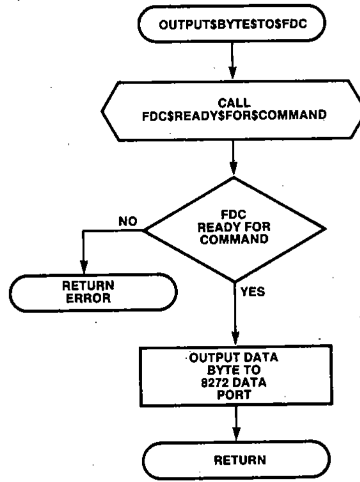
```
190   4               call write(drive,cylinder,head,sector,density);
191   4               call read(drive,cylinder,head,sector,density);

                      /* check read buffer against write buffer */
192   4               if cmpw(@wrbuf,@rdbuf,shr(bytes$per$sector,1)) <> 0FFFFH
                        then halt;
194   4             end;
195   3           end;
196   2         end;

197   1       end run72;

MODULE INFORMATION:

     CODE AREA SIZE     = 0570H   1392D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 0907H   2311D
     MAXIMUM STACK SIZE = 0022H     34D
     412 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

APPENDIX C
8272 DRIVER FLOWCHARTS

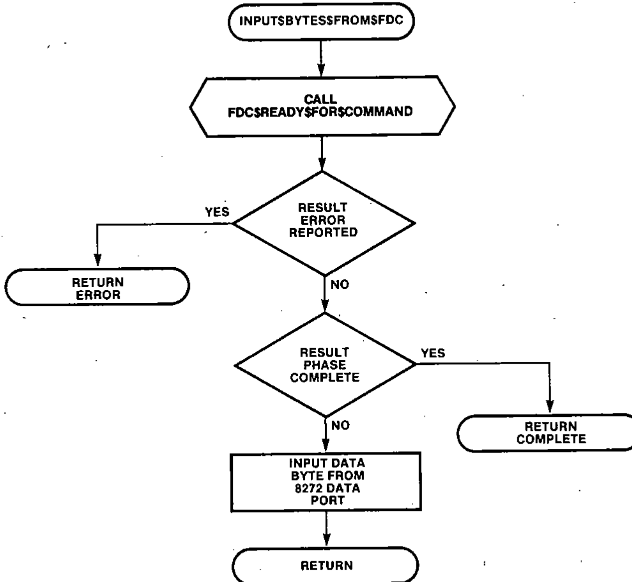

INITIALIZE$DRIVERS
DO
DRIVE = 0 TO
MAX #
NEXT
RESET
—DRIVE$READY
—DRIVE$STATUS$CHANGE
—OPERATIONINPROGRESS
—OPERATION$COMPLETE
DONE
RESET GENERAL STATUS
—OPERATIONINPROGRESS
—OPERATION$COMPLETE
RESET
GLOBAL$DRIVE$NO
RETURN


FDC$READY$FOR$COMMAND
DELAY FOR >20μs
TO PERMIT 8272
TO UPDATE
STATUS FLAGS
NO
RQM
FLAG SET?
YES
NO
DIOFLAG
SET FOR
8272 INPUT?
YES
RETURN
ERROR
RETURN

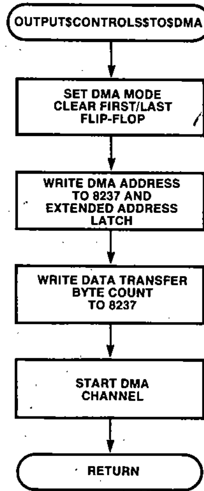

FDC$READY$FOR$RESULT
DELAY FOR >20μs
TO PERMIT 8272
TO UPDATE
STATUS FLAGS
8272
BUSY
NO
RETURN
COMPLETE
YES
NO
RQM
FLAG SET?
YES
NO
DIO
FLAG SET
FOR 8272
OUTPUT?
YES
RETURN
ERROR
RETURN



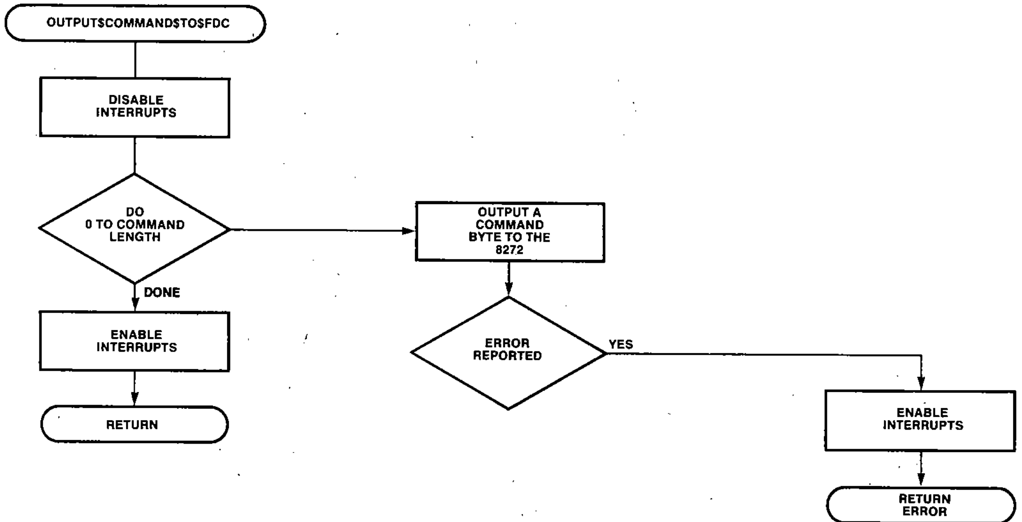
OUTPUT$BYTE$TO$FDC
CALL FDC$READY$FOR$COMMAND
FDC READY FOR COMMAND
NO
RETURN ERROR
YES
OUTPUT DATA BYTE TO 8272 DATA PORT
RETURN



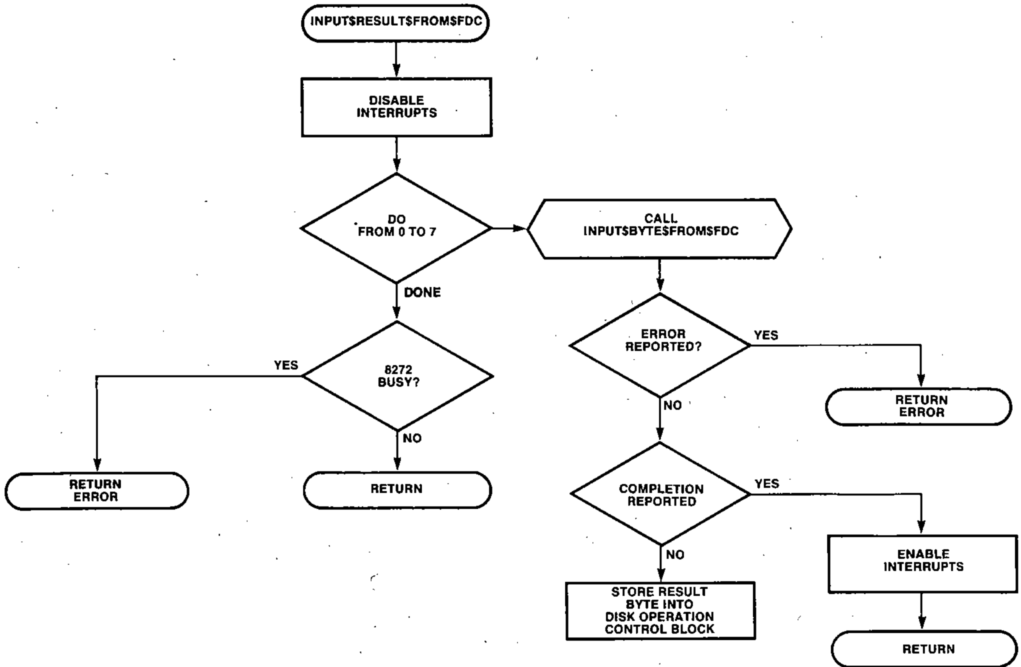
INPUT$BYTES$FROM$FDC
CALL FDC$READY$FOR$COMMAND
RESULT ERROR REPORTED
YES
RETURN ERROR
NO
RESULT PHASE COMPLETE
YES
RETURN COMPLETE
NO
INPUT DATA BYTE FROM 8272 DATA PORT
RETURN

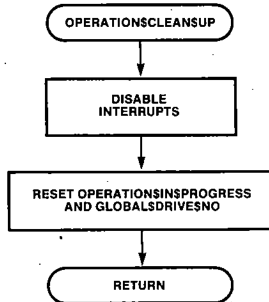


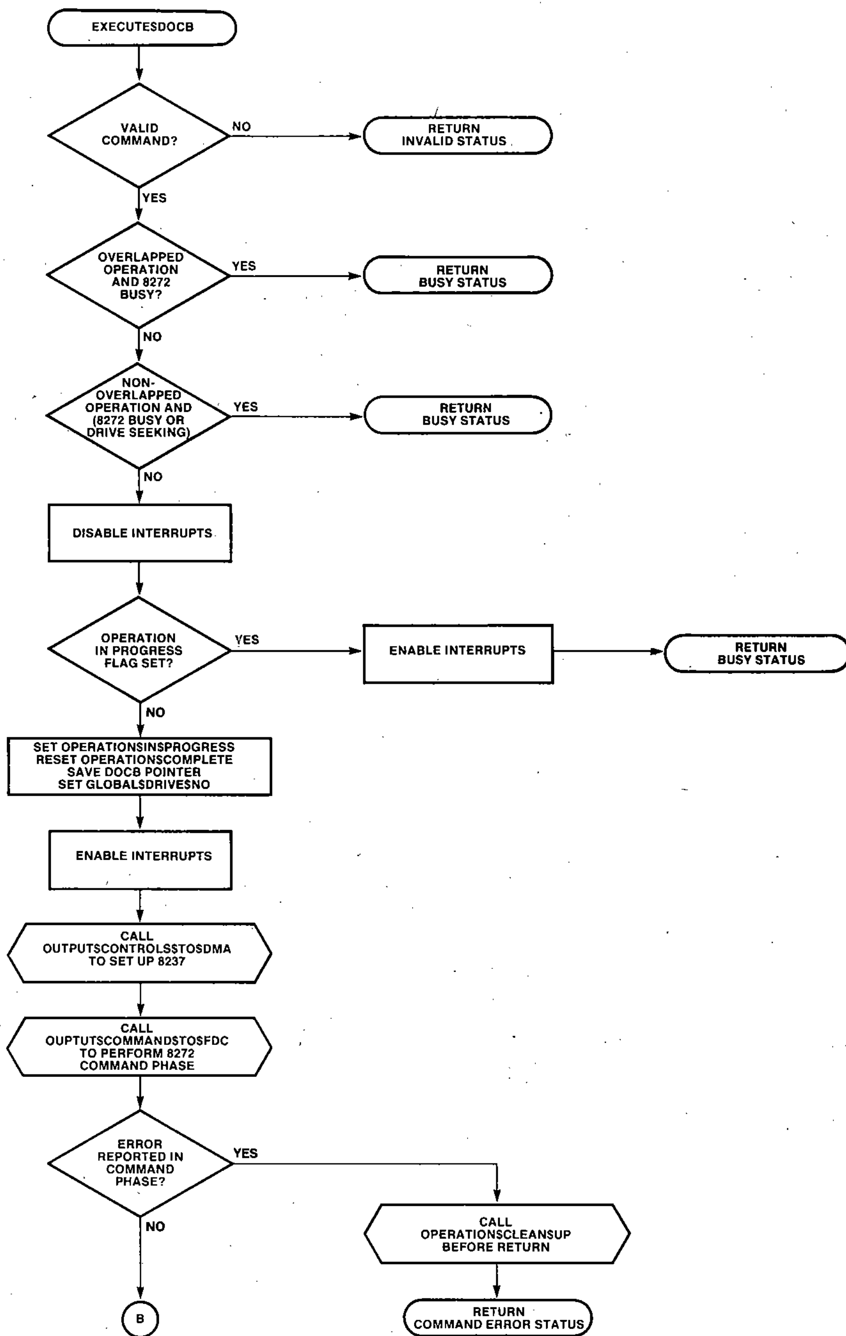
OUTPUT$CONTROLS$TO$DMA
SET DMA MODE
CLEAR FIRST/LAST
FLIP-FLOP
WRITE DMA ADDRESS
TO 8237 AND
EXTENDED ADDRESS
LATCH
WRITE DATA TRANSFER
BYTE COUNT
TO 8237
START DMA
CHANNEL
RETURN

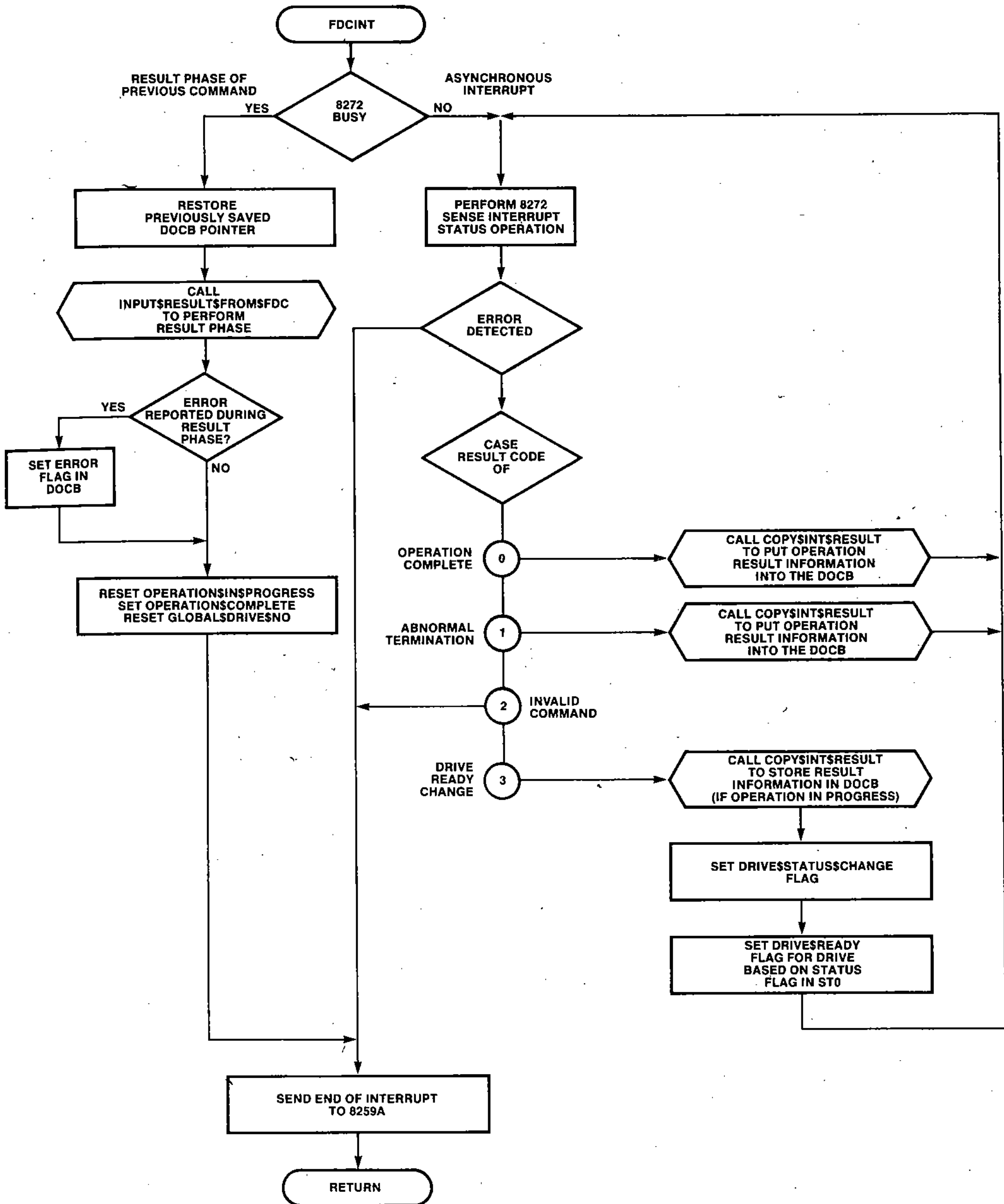


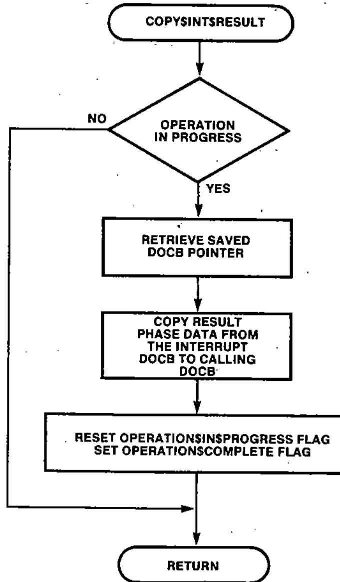
OUTPUT$COMMAND$TO$FDC
DISABLE
INTERRUPTS
DO
0 TO COMMAND
LENGTH
OUTPUT A
COMMAND
BYTE TO THE
8272
DONE
ENABLE
INTERRUPTS
ERROR
REPORTED
YES
RETURN
ENABLE
INTERRUPTS
RETURN
ERROR

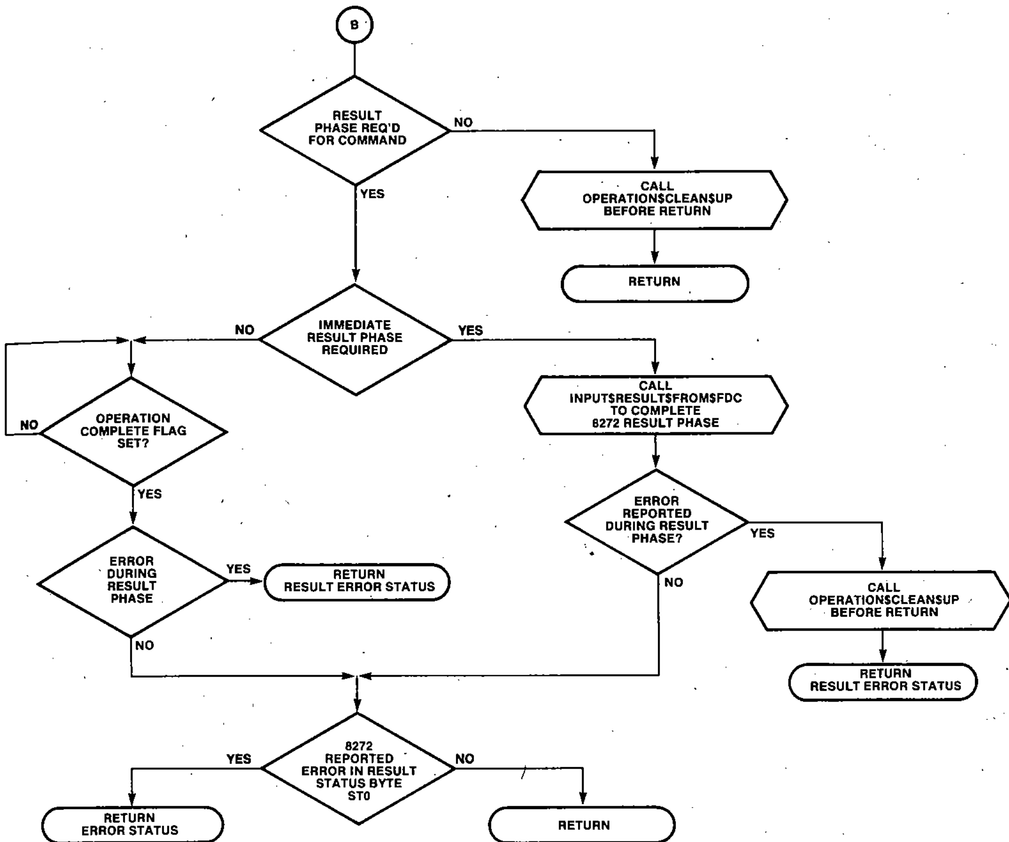

INPUT$RESULT$FROM$FDC
DISABLE INTERRUPTS
DO FROM 0 TO 7
CALL INPUT$BYTE$FROM$FDC
DONE
ERROR REPORTED?
YES
RETURN ERROR
NO
8272 BUSY?
YES
NO
RETURN ERROR
RETURN
COMPLETION REPORTED
YES
NO
ENABLE INTERRUPTS
STORE RESULT BYTE INTO DISK OPERATION CONTROL BLOCK
RETURN


OPERATION$CLEAN$UP
DISABLE INTERRUPTS
RESET OPERATIONINPROGRESS AND GLOBAL$DRIVE$NO
RETURN


EXECUTE$DOCB
VALID COMMAND?
NO
RETURN INVALID STATUS
YES
OVERLAPPED OPERATION AND 8272 BUSY?
YES
RETURN BUSY STATUS
NO
NON-OVERLAPPED OPERATION AND (8272 BUSY OR DRIVE SEEKING)
YES
RETURN BUSY STATUS
NO
DISABLE INTERRUPTS
OPERATION IN PROGRESS FLAG SET?
YES
ENABLE INTERRUPTS
RETURN BUSY STATUS
NO
SET OPERATIONINPROGRESS
RESET OPERATION$COMPLETE
SAVE DOCB POINTER
SET GLOBAL$DRIVE$NO
ENABLE INTERRUPTS
CALL OUTPUT$CONTROLS$TO$DMA TO SET UP 8237
CALL OUPTUT$COMMAND$TO$FDC TO PERFORM 8272 COMMAND PHASE
ERROR REPORTED IN COMMAND PHASE?
YES
CALL OPERATION$CLEAN$UP BEFORE RETURN
RETURN COMMAND ERROR STATUS
NO
B


FDCINT
RESULT PHASE OF PREVIOUS COMMAND
ASYNCHRONOUS INTERRUPT
YES
8272 BUSY
NO
RESTORE PREVIOUSLY SAVED DOCB POINTER
PERFORM 8272 SENSE INTERRUPT STATUS OPERATION
CALL INPUT$RESULT$FROM$FDC TO PERFORM RESULT PHASE
ERROR DETECTED
YES
ERROR REPORTED DURING RESULT PHASE?
NO
SET ERROR FLAG IN DOCB
CASE RESULT CODE OF
OPERATION COMPLETE
0
CALL COPYINTRESULT TO PUT OPERATION RESULT INFORMATION INTO THE DOCB
RESET OPERATIONINPROGRESS SET OPERATION$COMPLETE RESET GLOBAL$DRIVE$NO
ABNORMAL TERMINATION
1
CALL COPYINTRESULT TO PUT OPERATION RESULT INFORMATION INTO THE DOCB
2
INVALID COMMAND
DRIVE READY CHANGE
3
CALL COPYINTRESULT TO STORE RESULT INFORMATION IN DOCB (IF OPERATION IN PROGRESS)
SET DRIVE$STATUS$CHANGE FLAG
SET DRIVE$READY FLAG FOR DRIVE BASED ON STATUS FLAG IN ST0
SEND END OF INTERRUPT TO 8259A
RETURN


COPYINTRESULT
OPERATION IN PROGRESS
NO
YES
RETRIEVE SAVED DOCB POINTER
COPY RESULT PHASE DATA FROM THE INTERRUPT DOCB TO CALLING DOCB
RESET OPERATIONINPROGRESS FLAG
SET OPERATION$COMPLETE FLAG
RETURN


B
RESULT PHASE REQ'D FOR COMMAND
NO
YES
CALL OPERATION$CLEAN$UP BEFORE RETURN
RETURN
IMMEDIATE RESULT PHASE REQUIRED
NO
YES
OPERATION COMPLETE FLAG SET?
NO
YES
CALL INPUT$RESULT$FROM$FDC TO COMPLETE 8272 RESULT PHASE
ERROR REPORTED DURING RESULT PHASE?
YES
NO
CALL OPERATION$CLEAN$UP BEFORE RETURN
RETURN RESULT ERROR STATUS
ERROR DURING RESULT PHASE
YES
RETURN RESULT ERROR STATUS
NO
8272 REPORTED ERROR IN RESULT STATUS BYTE ST0
YES
NO
RETURN ERROR STATUS
RETURN