

EINDHOVEN UNIVERSITY OF TECHNOLOGY.  
DEPARTMENT OF ELECTRICAL ENGINEERING.  
DIGITAL SYSTEMS DIVISION.

24 MEI 1983

DESIGN OF AN INTELLIGENT  
WINCHESTER DISK CONTROLLER

by: H.P.M. van Eijkelenburg.

Report on the graduation project conducted by  
H.P.M. van Eijkelenburg under the authority of  
Prof. ir. A. Heetman.

Period: April 4, 1982 - March 17, 1983.

Coached by: ir. M.P.J. Stevens.  
              ir. J.P. Kemper.

---

! The department of Electrical Engineering assumes !  
! no liability for the contents of this report. !

---

### SUMMARY.

This report covers the design of an intelligent Winchester Disk Controller. In contrast with traditional controller designs, the Disk Operating System, generally resident at the Host computer, is integrated in the controller.

After an introduction concerning the characteristics and applications of Winchester drives, the problems involved with interfacing these drives to a host computer are discussed.

Chapter three presents the general concept which lead to the final design. A separation is made between hard- and software.

The hardware consists of a Winchester Controller Module, rather than a random logic circuit, a processor system consisting of a 16 bit CPU and an IOP co-processor, buffer- and program memory and an interface to the Host computer.

Chapter five deals with the software involved in this project, in particular the Disk Operating Systems. The UNIX operating system was taken as a guide-line for this purpose.

Though no prototype was built and tested, some conclusions are drawn in the last chapter concerning the feasibility of the design and its advantages over traditional controller designs.

In addition, some recommendations for follow-up are made.

**KEYWORDS:** Design, Disk controller, Winchester, Universal, Communication, DMA, Hardware, Processor system, Software, Disk Operating System, UNIX.

## CONTENTS.

Summary.....	1
Contents.....	2
1. Introduction.....	5
1.1. Mass storage devices.....	5
1.2. Winchester disk drives.....	6
1.3. Disk controller.....	7
2. Problem definition.....	8
2.1.1. Disk controller operation.....	8
2.1.2. Controller functions.....	9
2.1.3. Conclusion.....	15
2.2. Disk Operating System.....	16
2.3. Project definition.....	17
3. Winchester controller concept.....	19
3.1. Controller hardware.....	19
3.1.1. Drive control unit.....	20
3.1.2. Processor system.....	20
3.1.3. Host interface unit.....	20
3.1.4. Memory system.....	21
3.2. Disk storage organization.....	22
3.2.1. Physical storage.....	22
3.2.2. File descriptor.....	25
3.2.3. Address information.....	26
3.2.4. Index list.....	28
3.3. Directory structure.....	28
3.4. Free space administration.....	33
3.4.1. Free block pointer list.....	33
3.4.2. Bit map.....	36
3.5. Command structure.....	38
3.5.1. File handling commands.....	38
3.5.2. Directory commands.....	47
3.6. Host Interface.....	47
3.6.1. Network solution.....	48
3.6.2. Register solution.....	49
3.6.3. Bus solution.....	50
3.6.4. Conclusion.....	50

4. Controller hardware design.....	51
4.1. Introduction.....	51
4.2. Processor system.....	51
4.2.1 Parallel- versus co-processing.....	51
4.2.2. I/O processor description.....	55
4.2.3. Central processing unit description.....	55
4.3. Memory system.....	56
4.3.1. Dynamic RAM system.....	57
4.3.2. Static RAM system.....	59
4.3.3. ROM system.....	60
4.3.4. Address decoding.....	60
4.4. Host interface.....	61
4.5. Drive control unit hardware.....	63
4.5.1. Standard logic.....	63
4.5.2. Programmable logic.....	63
4.5.3. Single chip controller.....	63
4.5.4. Processor module interface.....	65
4.5.5. DMA transfer timing.....	71
4.5.6. DMA termination.....	76
4.6. Drive interface.....	76
4.7. Interface standards.....	78
5. Software description.....	79
5.1. Introduction.....	79
5.2. Initialization.....	80
5.3. Host protocol handler.....	88
5.3.1. Layer model.....	90
5.3.2. Application layer.....	91
5.3.3. Presentation layer.....	92
5.3.4. Transport layer.....	93
5.3.5. Data link layer.....	98
5.4. Disk protocol handler.....	101
5.4.1. Commands.....	101
5.4.2. Data.....	101
5.4.3. Task block program.....	102
5.5. Disk operating system.....	104
5.5.1. Command handler.....	104
5.5.2. Free space administration.....	115
5.5.3. File management.....	121
5.6. Error handler.....	122
5.6.1. Recoverable errors.....	122
5.6.2. User errors.....	123
5.6.3. Fatal errors.....	123

6. Conclusions and recommendations.....	124
6.1. Conclusions.....	124
6.2. Recommendations.....	124
7. Acknowledgements.....	126
Literature reference list.....	127
Appendix A : Hardware circuit diagrams.....	128
Appendix B : MSC 9000 data sheet.....	134
Appendix C : WD controller modules overview.....	145
Appendix D : FPLS based interface.....	149

## Chapter 1. INTRODUCTION.

### 1.1. Mass storage devices.

Any user of a digital computer system will agree to the importance of a reliable and fast mass storage device. This background memory is generally used to store user programs and operating data in quantities too large to be kept in the computers core memory. Demands imposed on such storage devices are:

- Reliability, to ensure the correct retrieval of stored information.
- Speed, to minimize delays caused by data access on the device.
- Cost-efficiency, meaning the price per unit of stored information should be as low as possible, typically far below the cost of a unit in core memory.

As these requirements are to some extent contradictory, several compromises were made by storage device manufacturers.

Three important kinds of storage devices can be discerned, all of them based on magnetic storage techniques:

1. Disk drives.
2. Drum devices.
3. Tape units.

The first category is by far the most widely used and will be focussed upon in this report.

Within the range of disk types, a large variety exists, starting with 3 inch diameter micro floppy disks and ranging to hard disks.

The following table offers an overview of their capacity.

DRIVE TYPE.	DIAMETER.	STORAGE CAPACITY.
Floppy disk	3 inch	100 - 250 Kbyte.
	5.25 inch	125 -1000 Kbyte.
	8 inch	250 -2000 Kbyte.
Winchester disk	5.25 inch	2 - 10 Mbyte.
	8 inch	5 - 40 Mbyte.
	14 inch	20 - 200 Mbyte.
Hard disk	over 14 inch	over 200 Mbyte.

Table I.

The choice of a certain type of disk drive will largely depend on its application. Storage capacity will in most cases be the decisive factor although access-time and cost-effectiveness play an important role as well.

Very recently, a growing trend towards the application of Winchester drives has developed, especially for small and medium sized computer systems. The traditional floppy disk drives, commonly found in these applications, are gradually replaced by Winchester drives.

The explanation for this development is obvious. Due to finer and more accurate mechanical parts and new head materials, Winchester drives have become a very favourable alternative to floppy disk drives in terms of storage capacity, access-time, reliability and more important, price per bit. It is for these reasons that this report will concentrate on Winchester drives.

## 1.2. Winchester Disk Drives.

The classification "Winchester" stands for a drive technology that employs a sealed disk and head-positioning assembly. This eliminates the need for complicated and thus expensive air filtering provisions and air flow control required for hard disks.

The first Winchester type drive was developed around 1973 by IBM, contained a double 30 Mbyte disk and was given type number 3030 as a result of that. This model number must have been associated by several people with the notorious Winchester double barrel raffle since this nickname was soon used and has been used for this drive technology ever since.

The major advantage Winchester drives offer, as opposed to floppy disks, is the fact that the read/write heads fly above the disk surface on an air cushion, as is the case with hard disk drives. This air cushion is created by the speed of the disk in conjunction with the special shaped flexure, the flexure being the metal holder upon which the heads are mounted. The absence of physical head-medium contact enables faster disk speeds and associated data recording speeds. Presently, a recording speed of 5 Mbit/sec is quite common for the smallest Winchester drives.

Table II highlights some of the features of Winchester disk drives compared to floppy disk drives.

	DIAMETER (inch)	HEADS	CAPACITY (formatted)	RECORDING DENSITY	TRACK DENSITY
FLOPPY	5.25	1	100 Kb		48 tpi
		2	200 Kb		
	8	1	500 Kb		96 tpi
		2	1 Mb		
WINCHESTER	5.25	2	5 Mb	6 - 10	255-980 tpi
		4	10 Mb	6 - 10	255-980 tpi
	8	2	20 Mb	6 - 10	255-980 tpi
		4	50 Mb	6 - 10	255-980 tpi

Table II.

### 1.3. Disk controller.

Choosing a Winchester disk drive as a background mass-storage device, is not the complete solution to the problem. A link between the host computer and the drive will have to be established. This link is usually formed by a disk controller device. Either a disk controller is bought or it is developed and built by the purchaser of the drive. When buying a controller, the user has little or no knowledge of its operation and has to settle for a standard controller device. When designing and building however, all customer specific demands and requirements can be taken into account, leading up to a flexible customized controller.

The object of this graduation project was to design a controller, capable of controlling a wide range of Winchester drives and interfacing with an arbitrary host computer system. In the next chapter we will go into the details of this problem.

## Chapter 2. PROBLEM DEFINITION.

### Introduction.

Prior to formulating the project covered in this report, a survey of the functions performed by a traditional controller is given. This will give a better insight into the problem at hand. Subsequently, the position of the disk operating system within a computer environment is discussed. Equipped with this knowledge, the project definition is formulated in the last paragraph of this chapter.

### 2.1. Disk Controller operation.

Looking at a conventional computer system which uses disks as background memory, one can abstract a raw system model as is done in figure 2.1.

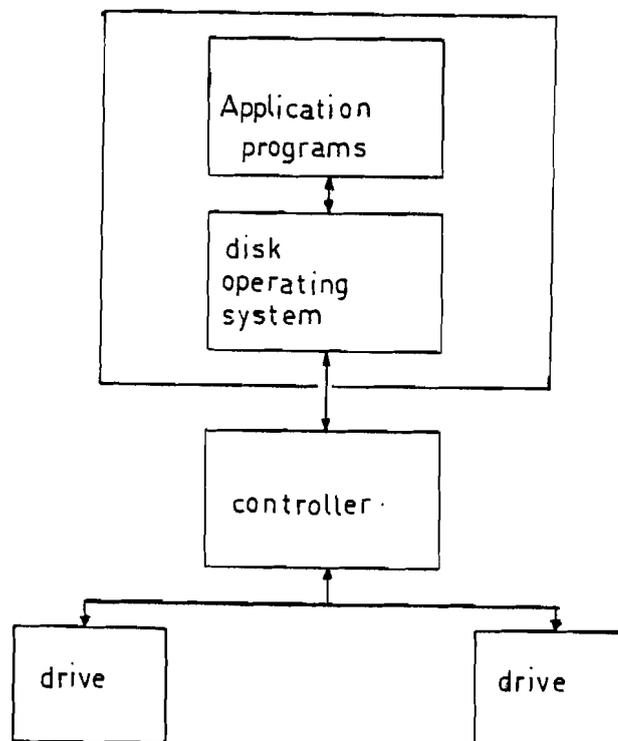


Figure 2.1: conventional computer system model.

The controller constitutes the link between what will henceforth be called the Host computer and one or more disk drives. For this purpose it entails a number of functional modules which will be described briefly in the next section.

### 2.1.2. Controller functions.

A functional block-diagram of a standard type disk controller is depicted in figure 2.2. In discussing the different modules we will distinguish between three kind of operations:

1. Disk write operations.
2. Disk read operations.
3. Drive control operations.

Disk write operations.

Serializer.

Data from the host is presented in either bytes or words, generally in a parallel format. Recording on the disk surface however is done serially, implicating the parallel data has to be serialized. The serializer performs this rather straightforward operation by means of a parallel in/serial out shift register.

CRC generator.

To increase data integrity, a cyclic redundancy checksum is calculated and added towards the end of a block of data - usually the size of one disk sector. This CRC control function enables verification of data upon reading and thus increases the reliability of the disk information.

Using CRC, the serial bits of information are treated as the coefficients of a binary polynomial  $P(x)$ . This polynomial is modified to an extent that makes it exactly divisible by a fixed polynomial  $G(x)$ . The divisor  $G(x)$  is referred to as the generator polynomial. The modified polynomial,  $M(x)$  of all the data bytes in a block is added. The result of this addition is recorded at the end of the datablock on the disk.

The reason CRC is used as an error detection scheme, is because of its advantages over other methods. Some of these advantages are:

- all errors within  $n$  successive bits are detected. ( $n$  is the number of bits in  $P(x)$ .)
- for even  $G(x)$ , all errors with an odd number of bits in error are detected (50 % of all possible random errors).
- All error patterns that are not divisible by  $G(x)$  are detected as erroneous.

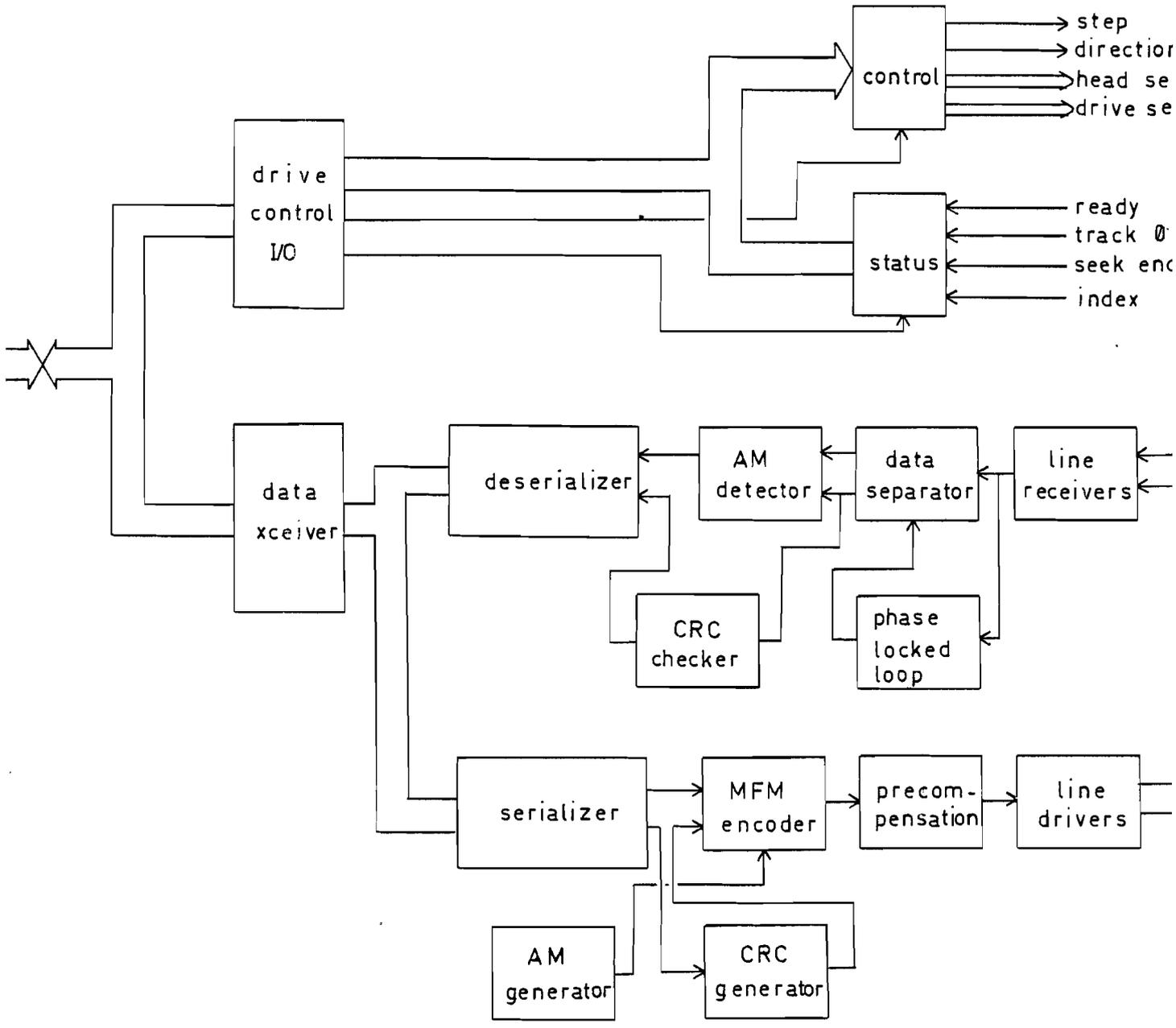


Figure 2.2: Controller function block diagram.

On top of this, CRC is efficient in that the number of control bits is relatively small compared to the number of data-bits.

**MFM-encoder.**

The format in which data from the serializer and the CRC generator is presented, is known as NRZ. (No Return to Zero). This format is unsuitable to cause flux reversals on the disk surface. Furthermore, timing information is stored on disk along with the databits to allow proper readback operation. To overcome this problem, a MFM-encoder is used, converting NRZ data and clock information into MFM data. (FM data representation is not discussed here since it is never used for Winchester drives.)

Figure 2.3 gives a timing diagram of different data representation waveforms used for disk storage.

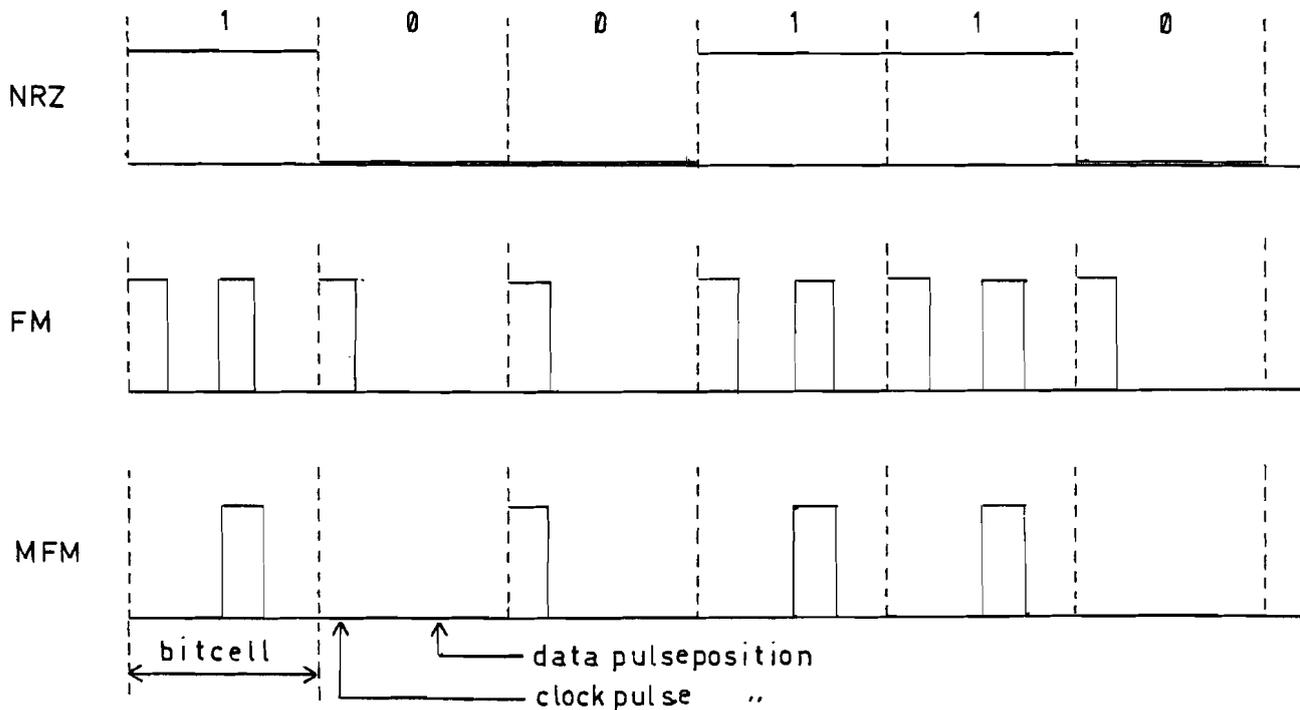


Figure 2.3: Data representation waveforms.

MFM is generated from NRZ according to the following rules.

1. Every bitcell contains either a data pulse, a clock pulse or no pulse at all.
2. A logic 1 in NRZ is represented by a data pulse in the corresponding bitcell.
3. If the logic bit in NRZ is 0, then no data pulse is present in its corresponding bitcell.

4. If the previous bitcell contained a data pulse, the clock pulse in the next bitcell is missing.
5. If the previous bitcell contained no data pulse, there will be a clock pulse in the next bitcell.

This scheme may seem confusing at first but it will no doubt become clear when reading the next paragraph on data reading.

#### Precompensation.

Due to the fact that the rotational velocity of the disk is uniform, there will be an increase in linear velocity of the disk surface passing under the read/write head. This increase is proportional to the track number, the most inner track having the highest track number.

As a result of this, the spacing between subsequent flux reversals of data and clock bits becomes smaller on the inner tracks of the disk. (bit crowding). If the spacing becomes too small, adjacent flux reversals tend to influence each other, resulting in bit shifts.

During a read operation, the read/write head develops a current as it encounters a flux reversal caused by either a clock or a data bit. It takes a finite time for this current to reach its peak value. During this time, the disk surface passes on and the next flux reversal crowds under the head, resulting in its peak current being summed with the previous one. In effect, this means the bit is shifted from its proper location. Figure 2.4 illustrates this effect.

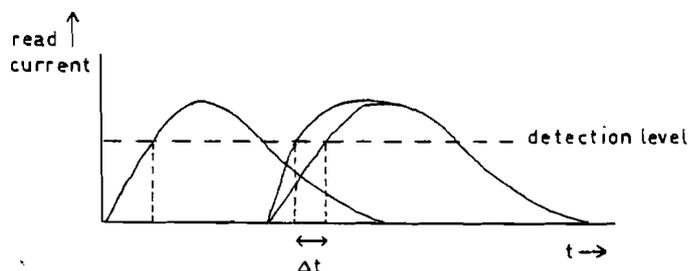


Figure 2.4: Bit shift

Fortunately, this effect is predictable and can be compensated by pre-shifting the bits in the opposite direction. This pre-shifting is done by the pre-compensation circuitry. As soon as writing is done on inner tracks - tracknumber greater than a predetermined value - the recorded bits are shifted back (early write), not shifted (on time) or delayed (late write) by a small amount of time. On reading back the recorded information, the bits will appear to be on time.

### Address Mark generator.

As a rule, Winchester disks are soft sectored. Thus address information concerning track and sector number has to be recorded on the disk. This information is present in the form of identity fields or in short ID-fields. The combination of an ID-field and a Data field forms a sector. The start of an ID field has to be detectable by the controller. For this purpose markers are present called Address Marks or AM. Address Marks distinguish themselves from other bytes by means of a missing clock pulse, i.e. a violation of the MFM encoding rules. Refer to figure 2.5.

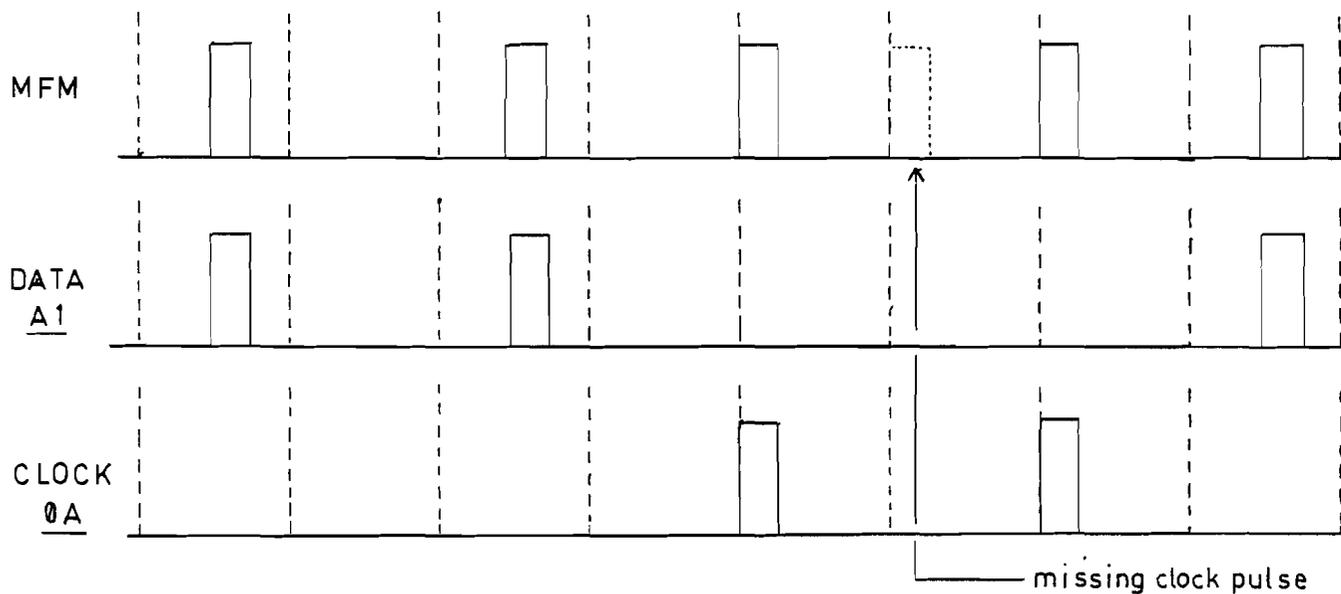


Figure 2.5: Address Mark generation.

The address mark generator takes care of this clock bit suppression.

### Line drivers.

To ensure distortion-free transmission of MFM data between controller and drives, differential line drivers are used. The exact electrical specification of these drivers is dependent on the drive manufacturer.

Disk read operations.

Phase locked loop.

The initial phase in the read process, consists of abstracting the timing information recorded on the disk. The most accurate method is to use a phase locked loop, consisting of a phase comparator, a low pass filter and a VCO. The VCO is constantly being adjusted by the information derived from the clock pulses on the disk. As a result of this, the output of the VCO is in synchronization with the data stream from the disk.

Data separator.

The data separator, effectively a MFM to NRZ data converter, generates separate data and clock pulses from its MFM input signal. The output of the forementioned VCO is used for synchronizing this data separator.

Address Mark Detection.

To find the beginning of an ID-field, the controller has to search for an Address mark. The address mark detector triggers on a missing clock pulse in the MFM data stream. Depending on whether the AM belonged to an ID-field or a Data field, subsequent bytes are read and interpreted by the controller.

CRC check.

The next phase in the read process is the CRC check. The CRC checksum, calculated by the CRC check circuitry conform paragraph 2.1, is compared with the checksum read from the disk at the end of the data field. A mismatch will result in an error signal, indicating the received data block contains errors.

Deserializer.

As is suggested by the name, the deserializer performs the inverse function of the serializer, converting the serial NRZ data from the data-separator in 8 or 16 bit words.

Drive control operations.

The third section of the controller exists of a number of control and status lines that enable it to control the mechanical parts of the drives connected to it. The most commonly used control and status lines are listed below.

Step	Moves the head assembly one track.
Direction	Indicates the direction of the head assembly movement. Combining the step- and direction signal allows positioning of the heads on any track of the disk.
Head select	Selects a single head of the head assembly for reading or writing.
Drive select	Selects one particular drive in the daisy chain connected to the controller.
Ready	Indicates the drive is ready for operation. After a start-up, it requires some time for the disk to reach its operating velocity. During this time the drive is not ready.
Track 0	Indicates the heads are at track zero. This signal is required for head calibration after a start-up or a reset.
Seek end	Indicates the heads have reached their destination and are stable after a head movement operation.
Write fault	As a result of an incorrect operation, more than one head has been selected for writing or write current is flowing through a deselected head.
Index	The index pulse signals the beginning of a track.

### 2.1.3. Conclusion.

The above presents a general overview of the most predominant characteristics of a disk controller. A more detailed discussion of this topic is considered superfluous since the controller concept discussed in this report uses advanced integrated modules for implementing these controller functions.

## 2.2. Disk operating system.

So far, we discussed the hardware provisions needed to connect one or more Winchester drives to a host computer. Using this hardware set-up, the host has the capability to control the drive and to store and retrieve data on a sector basis.

The next layer in the hierarchy of data storage lies between this physical disk control layer and the logical data structure of the host computer.

Any user of the host computer system, wanting to manipulate data, does so by using a logical data structure. The most common structure is a file structure. Every collection of data items that meets a certain format is called a file. Files can again be divided into one or more records.

To be able to read and write files to and from a disk, a translation has to be performed between a logical file or record and the physical sectors on the disk where the information contained by that file or record will be stored. Consequently, the host computer has to have knowledge of the physical organization of the disk.

Furthermore, the host computer has to supply the user with an access mechanism that allows easy reading, writing and manipulating of files. Thus a set of commands the user can apply has to be provided by the host computer.

Summarizing, one can list the functions of this intermediate layer between disk controller and host computer as such:

- File structure support.
- Free disk space allocation and administration.
- File directory support.
- File manipulation support.

Henceforth, the collection of these functions will be referred to as Disk Operating System (DOS), since they allow the user to operate the disk.

Figure 2.6. places the DOS layer in its context.

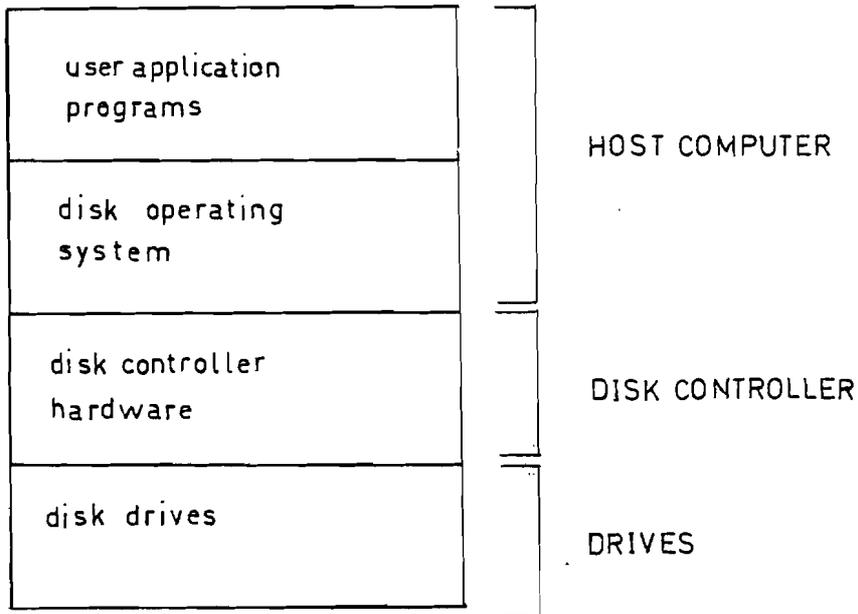


Figure 2.6: DOS layer position.

### 2.3. Project definition.

Traditionally, the Disk Operating System forms an integral part of the host computers' overall operating system, as indicated by figure 2.6.

The object of this graduation project was not only to design a universal Winchester disk controller, as mentioned in chapter 1, but also to add to the controller those functions concerning disk storage, normally performed by the host computer operating system. Taking this approach implies the design of an intelligent disk controller, to be used in conjunction with an arbitrary host computer. Figure 2.7. shows the situation that arises when applying an intelligent disk controller.

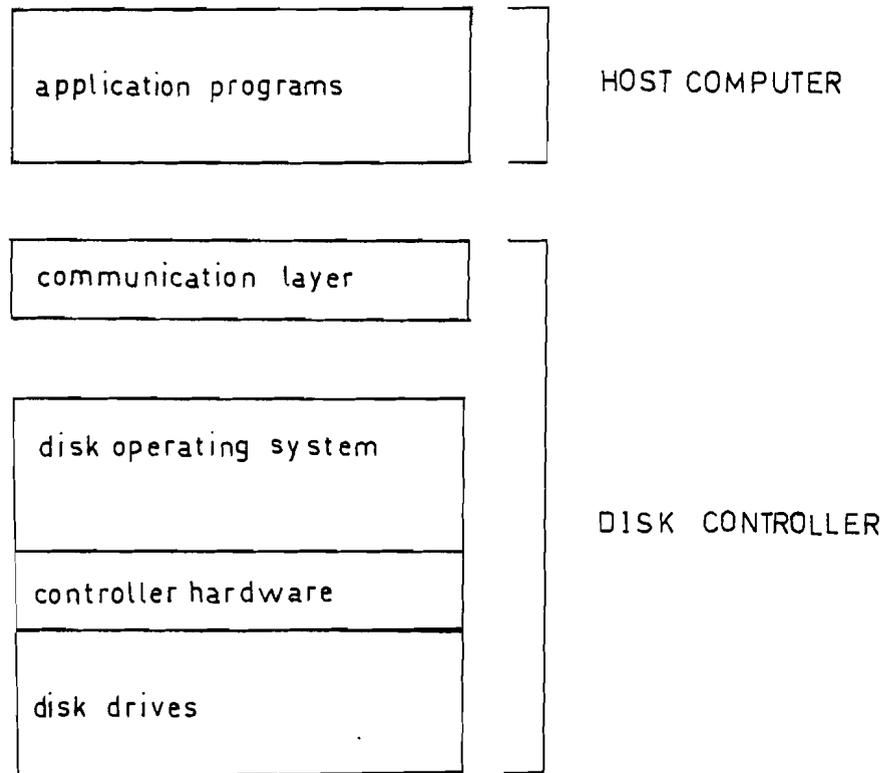


Figure 2.7: Intelligent Disk Controller position.

This constitutes a new approach towards controller design. Various responsibilities residing in the Host computer, are now being transferred to the controller. This should enable the controller to operate in conjunction with a large variety of host computers.

A further design requirement involves maximum universality of the controller towards different types of Winchester drives. The concept of the intelligent Winchester Disk Controller will be presented in the next chapter.

A summary of the system requirements is given below.

- Integrated Disk Operating System.
- High level communication between controller and Host.
- Support of multiple Winchester Drive types.

## Chapter 3. WINCHESTER CONTROLLER CONCEPT.

### 3. Winchester Disk Controller Concept.

From the requirements mentioned in the previous chapter, two subsystems can be deduced that constitute the Winchester Disk Controller. The first subsystem is formed by the hardware, which is drawn in figure 3.1. The second subsystem is the software which operates the controller.

#### 3.1. Controller hardware.

The hardware of the controller can be divided into four parts:

1. Drive control unit. (DCU)
2. Processor system. (PS)
3. Host Interface unit. (HIU)
4. Memory system. (MS)

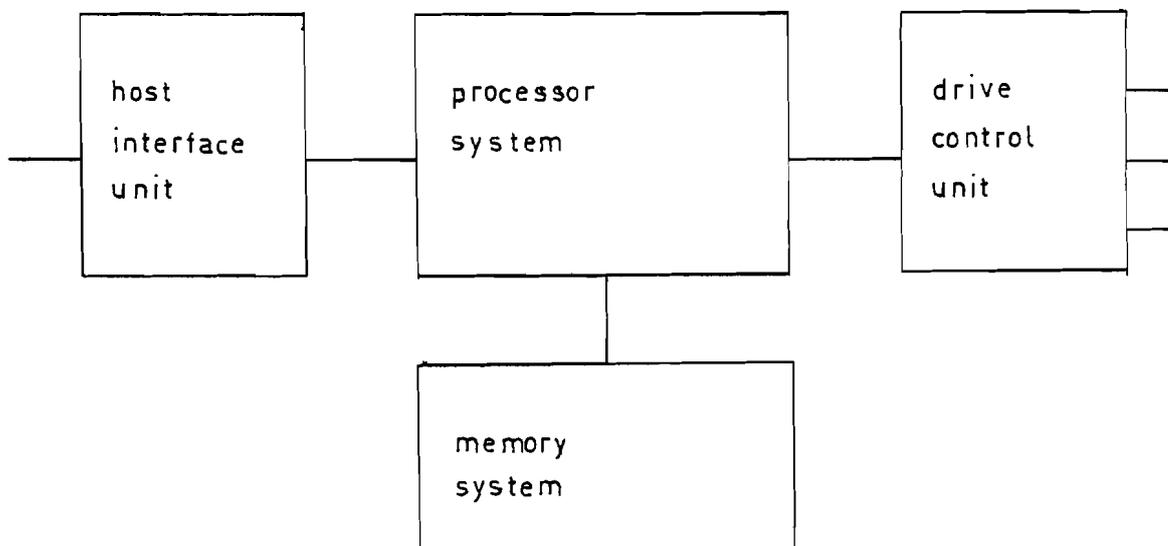


Figure 3.1: Controller block diagram.

### 3.1.1. Drive Control Unit.

The drive control unit interacts directly with one or more Winchester Disk Drives. Its functions include:

- Selecting the appropriate drive.
- Positioning the heads on a specified track.
- Formatting soft sectored disks.
- Write sector.
- Address Mark Detection.
- Read sector.

In short, this is the part which is commonly regarded as the actual disk controller. In our concept however, it only forms part of the integral controller.

### 3.1.2. Processor System.

The controllers processor system comprises three different major tasks:

- 1 DOS execution.
- 2 Communicating with the disk through the Disk Control Unit.
- 3 Communicating with the host computer system through the Host Interface Unit.

As the nature of these tasks allows for a division in processing (1) and I/O (2,3), the processor system will be separated in two sections. One central processing unit will be used for executing the Disk Operating System, a separate I/O processor will perform all necessary communication between the Winchester Disk Controller, Winchester Drives and Host Computer.

### 3.1.3. Host Interface Unit.

Obviously some sort of connection has to be established between the controller and the Host Computer. Whether this connection exists of a parallel or a serial link is of no major importance at this stage. There are however some requirements this connection has to meet:

- Maximum transfer speed of data to minimize delays caused by records or files in transport between Host and Controller. Thus, either a parallel link with DMA capability or a high speed serial data link will be needed.
- Reliability to prevent the necessity of retransmission or data inconsistency caused by transmission errors.

At this stage it is premature to dwell upon the possible imple-

mentations of such a communication link as these are to a large extent dependent on the Host Computer used.

#### 3.1.4. Memory System.

The Winchester Disk Controller's memory system serves a dual purpose. Firstly it incorporates the programs necessary for controller operation. These programs are either permanently stored in ROM or loaded from reserved tracks of a disk into RAM.

Secondly, a data buffer has to be provided for, to compensate for speed differences between disk and host. Furthermore, the DOS needs memory for storage of administrative information. The more information that can be kept in the controllers workspace at a time, the faster file access can be achieved by minimizing the number of disk accesses.

Summarizing, we come to the following hardware concept block diagram. (Figure 3.2.)

In chapter 4 we will go into the details of every part of the block diagram.

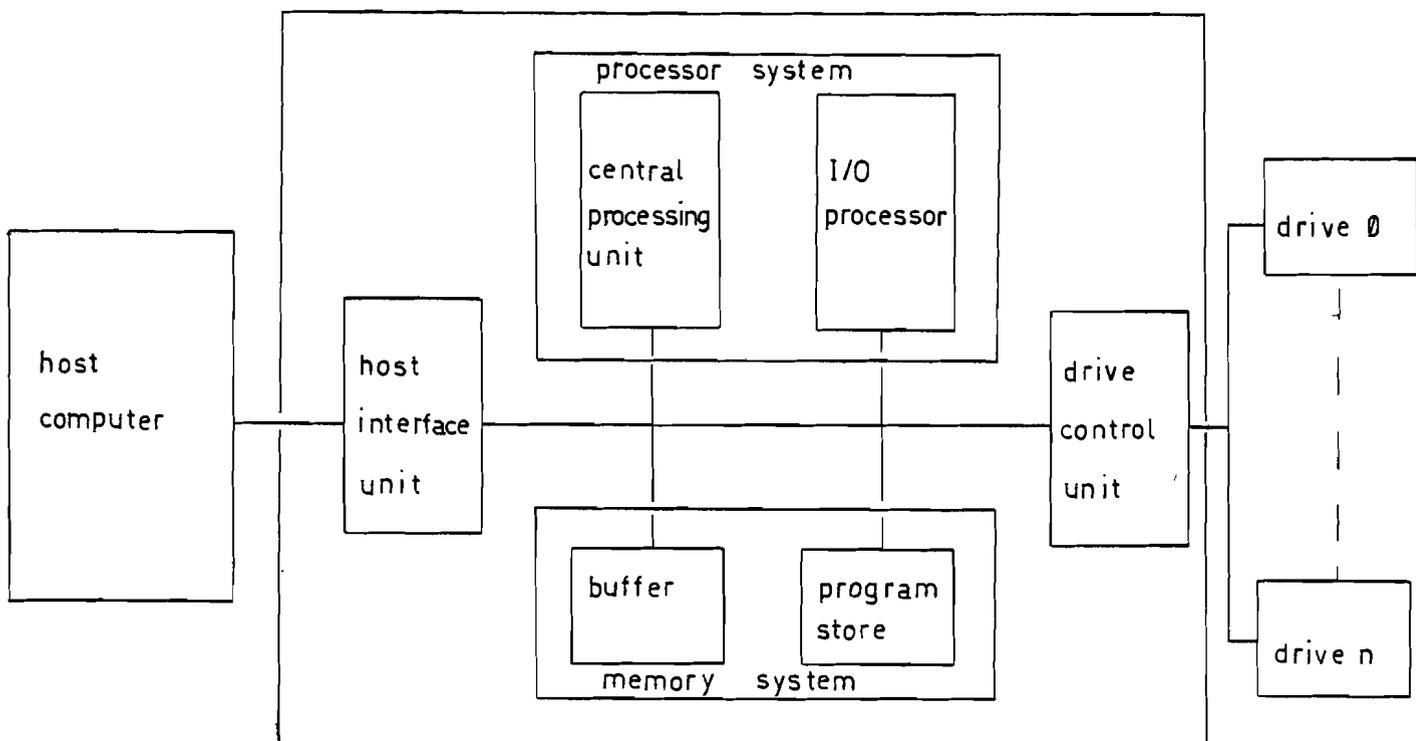


Figure 3.2: Controller functional block diagram.

### 3.2. Disk Storage Organization.

In order to map the users logical files to the physical disk storage space, an organization has to be set up that performs this task with a minimum amount of overhead.

The choice of the file structure employed is of extreme importance as it has a major impact on the systems performance. Rather than starting from scratch, the UNIX file structure was taken as a guideline. The reasons for this choice are twofold:

First of all, the UNIX Operating System can indulge in a fast growing popularity, especially for use in small to medium computer systems. The multi-user / multi-task facilities offered by UNIX are remarkably powerful. It is on these devices that Winchester Disks are used more and more.

Secondly, UNIX is quite a transparent operating system compared to others. This makes it relatively easy to adopt to specific needs. In spite of this, the software behind UNIX is considered to be sufficiently uncomplicated to be implemented partially - the file handling part to be exact - in the Winchester Disk Controller at hand.

As a result of this choice, the reader may discover various aspects in the following section that are very much like UNIX. However, it should be noted that alterations were made at certain points where they were considered useful for this particular application.

#### 3.2.1. Physical storage.

Information on the disk is stored in blocks of fixed length called sectors. The size of one sector on a Winchester disk is usually selectable between 256, 512, 1024 or more bytes/sector. This selection is made before formatting the disk. Unix assumes a sector length of 512 bytes, a size which is supported by almost all Winchester drive manufacturers. We will adopt this sector length as well.

In figure 3.3 a schematic view of a typical Winchester Disk Drive ( 2 disks, 4 heads ) is given. Each individual sector is addressable by specifying head, cylinder and sector number.

UNIX treats files as contiguous arrays of characters. Thus a file is mapped on a number of sectors on the disk, large enough to contain the file. Any structuring of data within a file is left to the program that operates on the file. Although this kind of unstructured files enables the use of them in many different ways, this loose scheme was considered unsuitable for the application at hand.

The file system we wish to offer the Host computer should have structured files. For this purpose, files are made of one or

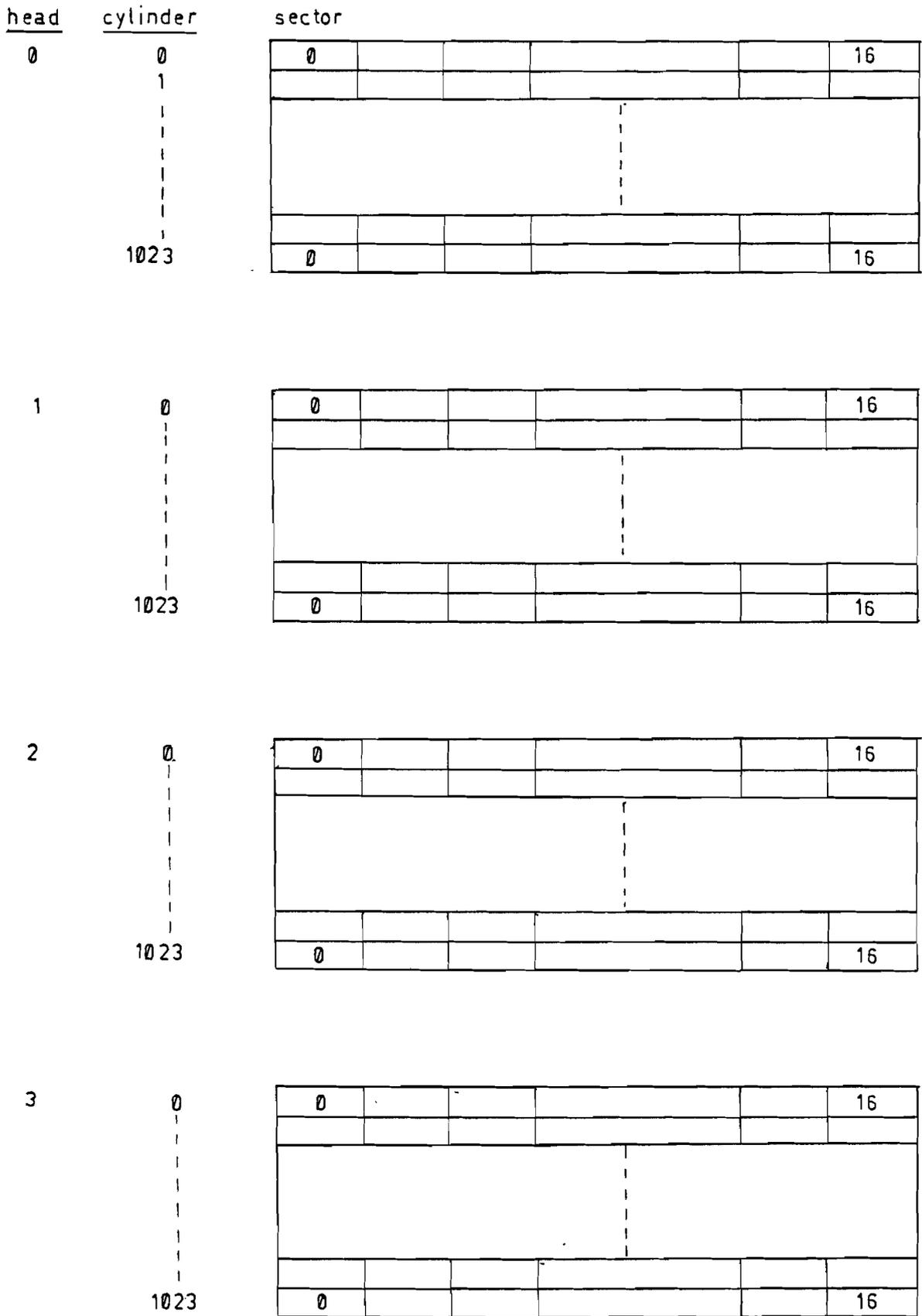


Figure 3.2 Disk schematic view.

more logical records. The length of these records is determined by the file type. The information concerning the structure of a file is kept in the file descriptor. (refer to 3.2.2.)

The most straightforward method to store structured files is to map each logical record to one or more sectors of the disk. As a rule, the last sector thus used will not be fully occupied with data. They will be filled with empty characters. Figure 3.4 shows a comparison between the UNIX strategy and the alternative solution.

I. UNIX.

BLOCK 1	BLOCK 2	BLOCK 3	BLOCK 4	BLOCK 5	BLOCK 6
file				null	

II.

	1		2		3		4		5		6
rec.1	null	rec 2	null	rec 3	null	rec 4	null	rec 5	null		

III.

	1		2		3		4		5		6
	record 1		record 2			record 3					

Figure 3.4: File storage.

From this comparison it is obvious that the UNIX file scheme uses the available disk space much more efficiently. Only when record lengths are a true multiple of one sectorlength they can be neatly mapped and can the other scheme be used effectively. This implicates a recordlength of 512 bytes for most Winchester drives. Since this is rarely the case, the UNIX scheme for storage will be adopted. However, the controller's DOS will be equipped with routines that offer structured files to the Host computer.

Implementation of these files is achieved by using the information on the record length of a file present in its file des-

criptor. After reading the required number of blocks in the controllers workspace, the structure of the file is assembled using the file descriptors information. Figure 3.5. illustrates this process. Obviously, this storage method is a trade-off between fast random access to file records and efficient disk space usage.

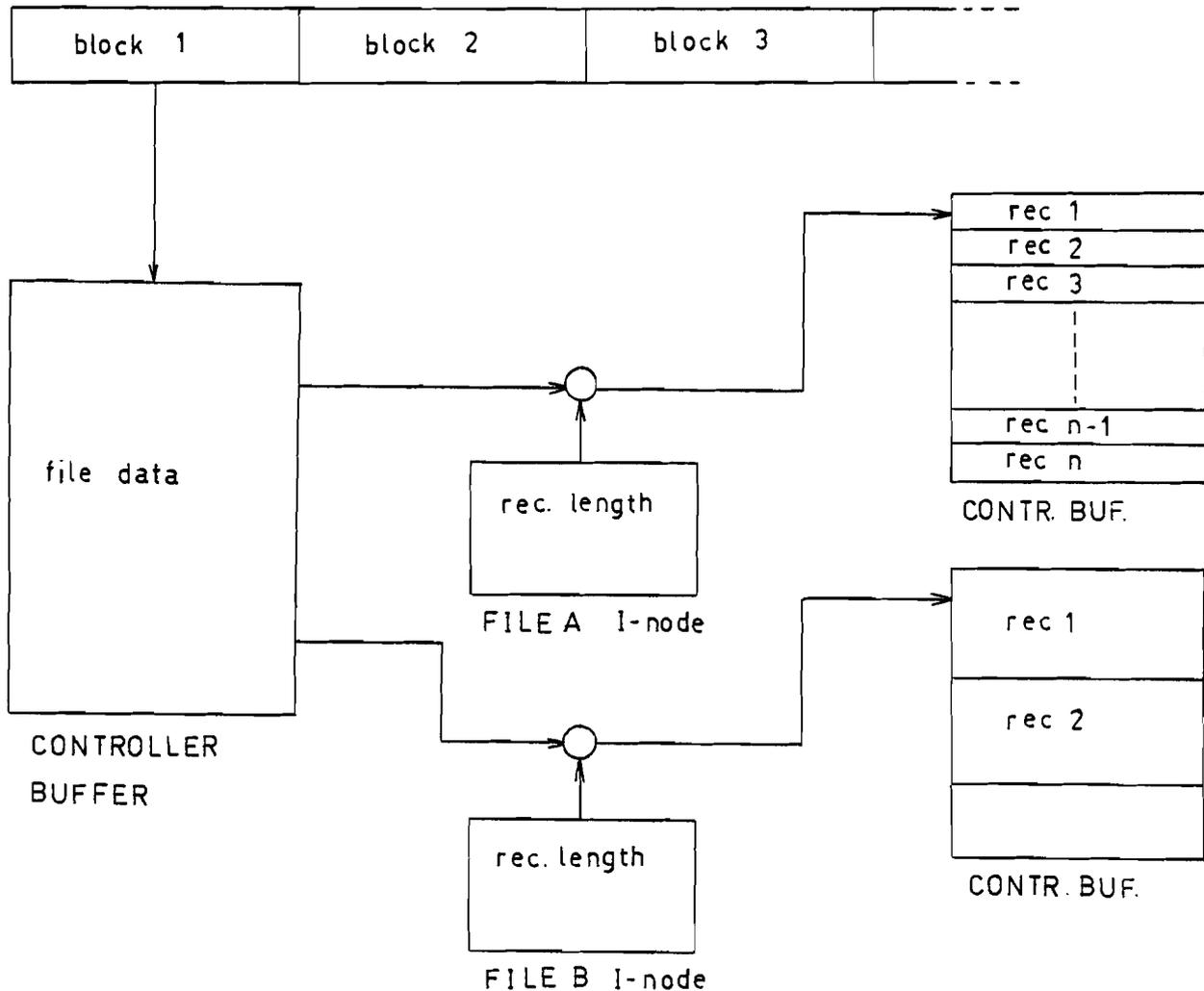


Figure 3.5: File structuring.

### 3.2.2. File descriptor.

Every file is described by a file descriptor containing all the relevant information on that particular file. In UNIX, file descriptors are referred to as Index-nodes or I-nodes. An Index-node contains the following information:

1. Identification of the user and owner of the file.
2. Access rights.
3. Address information concerning the physical location of the file on disk.
4. File size, i.e. the number of physical blocks it occupies.
5. Number of links to the file, i.e. number of times it appears in the directory.
6. File type: User file, record length ; Directory file.

The I-node describes the entire file and provides the information required to access its contents.

Notice that some modifications were made to the standard UNIX I-node.

### 3.2.3. Address information.

The address information on a file is organised as 13 4 byte pointers. The first 10 pointers correspond directly to the first ten blocks (disk-sectors) of the file. The 11th pointer points to a block containing 128 pointers to the next 128 blocks of the file. If the file is longer than 138 blocks, pointer 12 of the I-node is used as a double indirect pointer. It points to a block of 128 further pointers, each pointing in their turn to pointerblocks, containing pointers to the subsequent blocks of the file. Finally, pointer 13 is used as a tripple indirect pointer. As a result of this, each file can occupy a maximum of 2,113,674 512 byte blocks on the disk. This upper limit is considered sufficient to accomodate most file lengths. Note that large files become increasingly unpractical to operate upon.

This organization allows relatively short files (less than 5120 bytes) to be addressed directly through the I-node. Longer files require an extra indirection i.e. an extra disk access and thus a longer access time, as might have been expected. The end of a file can be signaled by a null pointer, following the pointer to the last block of the file.

Two different pointer types will be used:

- Indirect pointer (I): pointing to a pointer block.
- Sequential pointer (S): pointing to a file block.

The reason for this distinction will become clear in chapter 5 when discussing record insertions. Obviously pointer 1 through 10 in the I-node are S-type pointers whereas 11,12 and 13 are I-type.

Refer to figure 3.6. for a graphical representation of the pointer mechanism described.

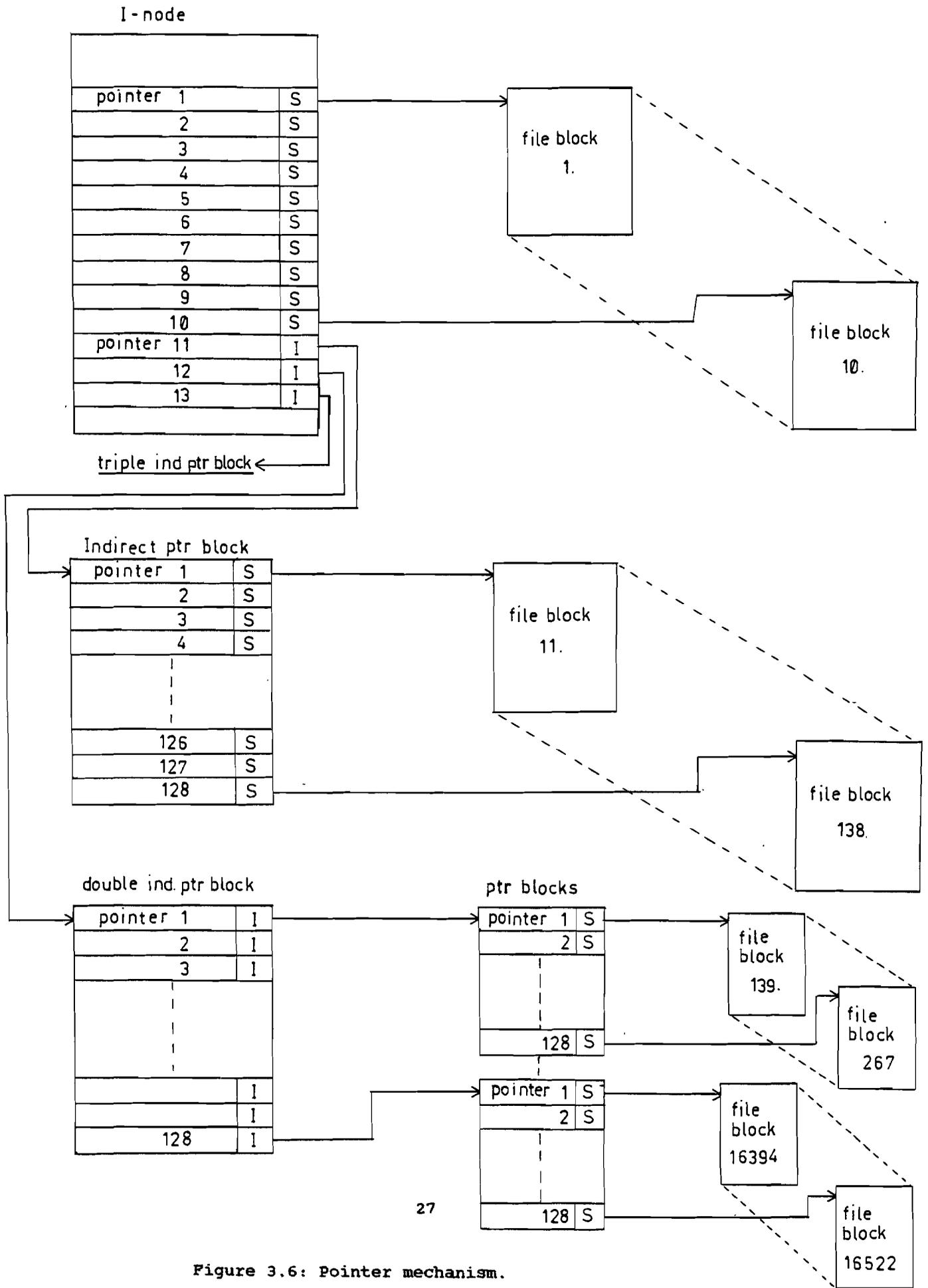


Figure 3.6: Pointer mechanism.

#### 3.2.4. Index list.

All index nodes, and thus all files, can be found by searching through the Index-list (I-list). The Index-list is a list of all I-nodes in the system and starts at a fixed location on the disk. The length of this list, determined by the number of I-nodes, may be such that it covers more than one block of the disk. Therefore the I-list will be dispersed over a number of blocks. We will discuss to possible solutions to organize this Index-list.

- a) **Linked list:** Using a linked list method, every block of the I-list is terminated by a sequential pointer to the next block of the list. The major advantage of this method is the possibility to expand the I-list when required. However, searching through the list for a particular I-node can only be done linearly by starting from the first block, which is at a known position. This can be a very time consuming process, especially when the I-list becomes dispersed over the disk due to multiple extensions. To overcome this problem, it would be favourable to keep the I-list in the controllers' workspace as much as possible. This would eliminate the need for multiple disk accesses but would on the other hand claim considerable amounts of controller workspace.
- b) **Fixed list :** Using a fixed list means the I-list is stored on a fixed number of contiguous blocks on the disk. This means a predetermined number of blocks will have to be reserved for the I-list. Expansion of the I-list becomes impossible. Once it is filled, no new I-nodes can be added, unless another I-node is deleted first. Thus there is a maximum number of files that can be resident in the system. Though this method may seem unfavourable, its strength lies in the ability of the controller to access an I-node directly by calculating its position on the disk. Thus the I-list need not be kept in workspace since one integer number is sufficient to locate the proper I-node.

In Figure 3.7. both methods are shown.

Although a linked I-list offers more flexibility, a fixed list will be used to allow easy access to an I-node and thus a file. The consequence of this choice is the limitation to a predetermined number of files.

#### 3.3. Directory structure.

Access to a file is obtained through the Index-node. Rather than searching through the index-list linearly, looking for a particular Index-node, a hierarchy of directories is added to the controller concept.

In UNIX, a directory is considered as an ordinary file with limited access rights. Only the system can create, write and

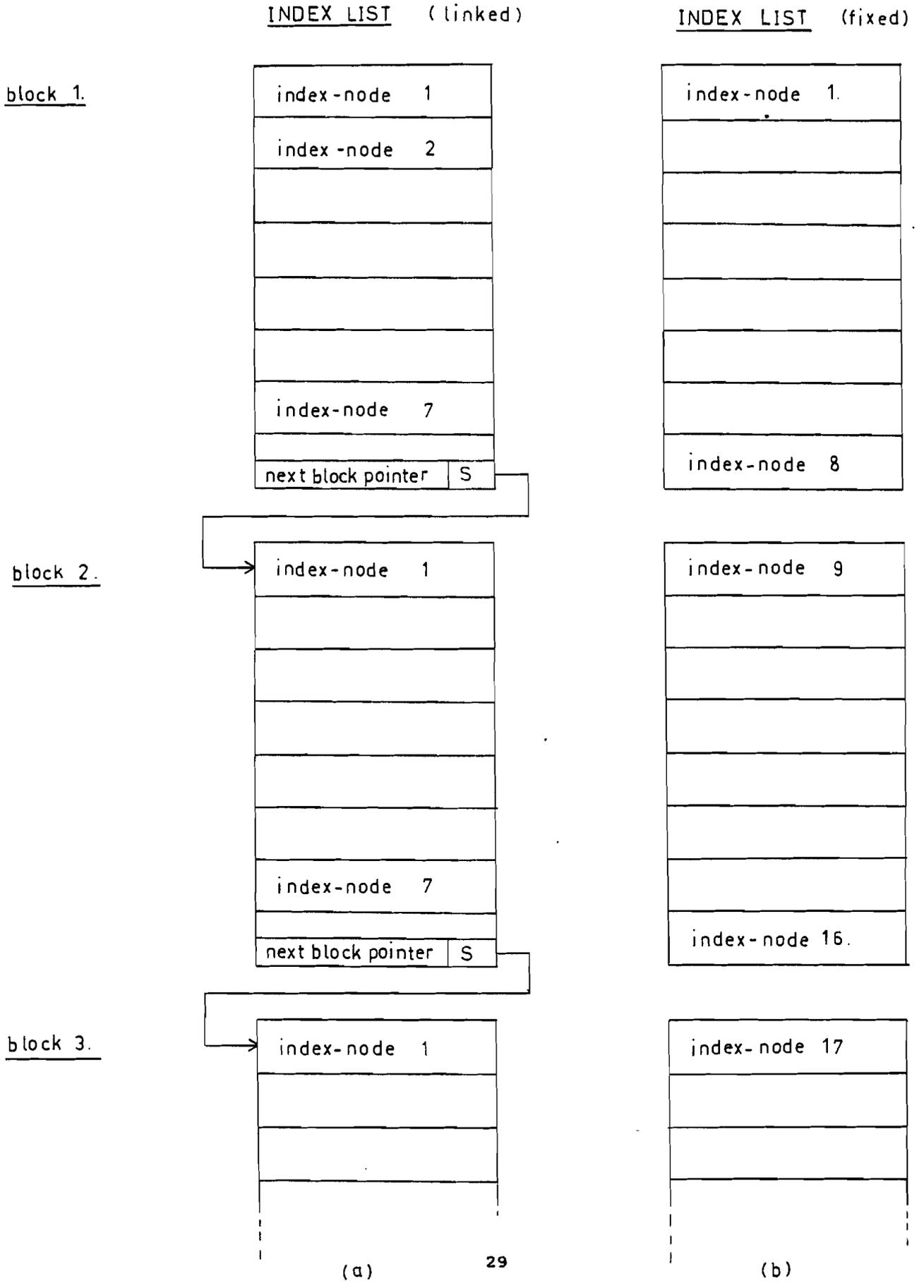


Figure 3.7: Index list.

modify a directory file. A directory file is marked as such in the corresponding Index-node.

An entry in a directory file comprises two fields. The first is occupied by a symbolic alpha-numeric name of limited length. The UNIX convention is 14 characters maximum and as this is considered a reasonable amount, it will be used in this concept as well.

The second field contains a two byte integer number, called the Index-number or I-number. The Index-number is an offset in the Index-list, leading directly to the Index-node of the file that corresponds to the symbolic file name.

Refer to figure 3.8. for a graphic representation of the directory mechanism.

Due to the fact that files referred to in a directory can be either user-files or user defined sub-directories, a tree-like directory structure is created. The user can thus specify a file by its path through this tree structure. The controller converts this pathname into the desired Index-node by starting to look in the current directory of the user at hand. The first component of the pathname is searched in this directory and when found, results in an Index-node. If it was the last component of the pathname, the Index-node found is the final result. If not, the next pathname component is searched in the file corresponding to the Index-node just found. Note that in this case the file type had to be directory. Figure 3.9. illustrates this tree-directory structure.

Apart from its straightforwardness and simplicity of implementation, this directory mechanism offers advantages in respect of access-right validation and file protection.

Two standard entries in every directory are worth mentioning distinctively. UNIX uses a single and a double dot as symbolic name to refer to these entries. The first refers to the directory itself enabling the user to read his current directory without knowing its name. The second refers to the parent of the directory in which it appears, this parent being the directory in which it was created. This entry allows for backtracking through the directory tree.

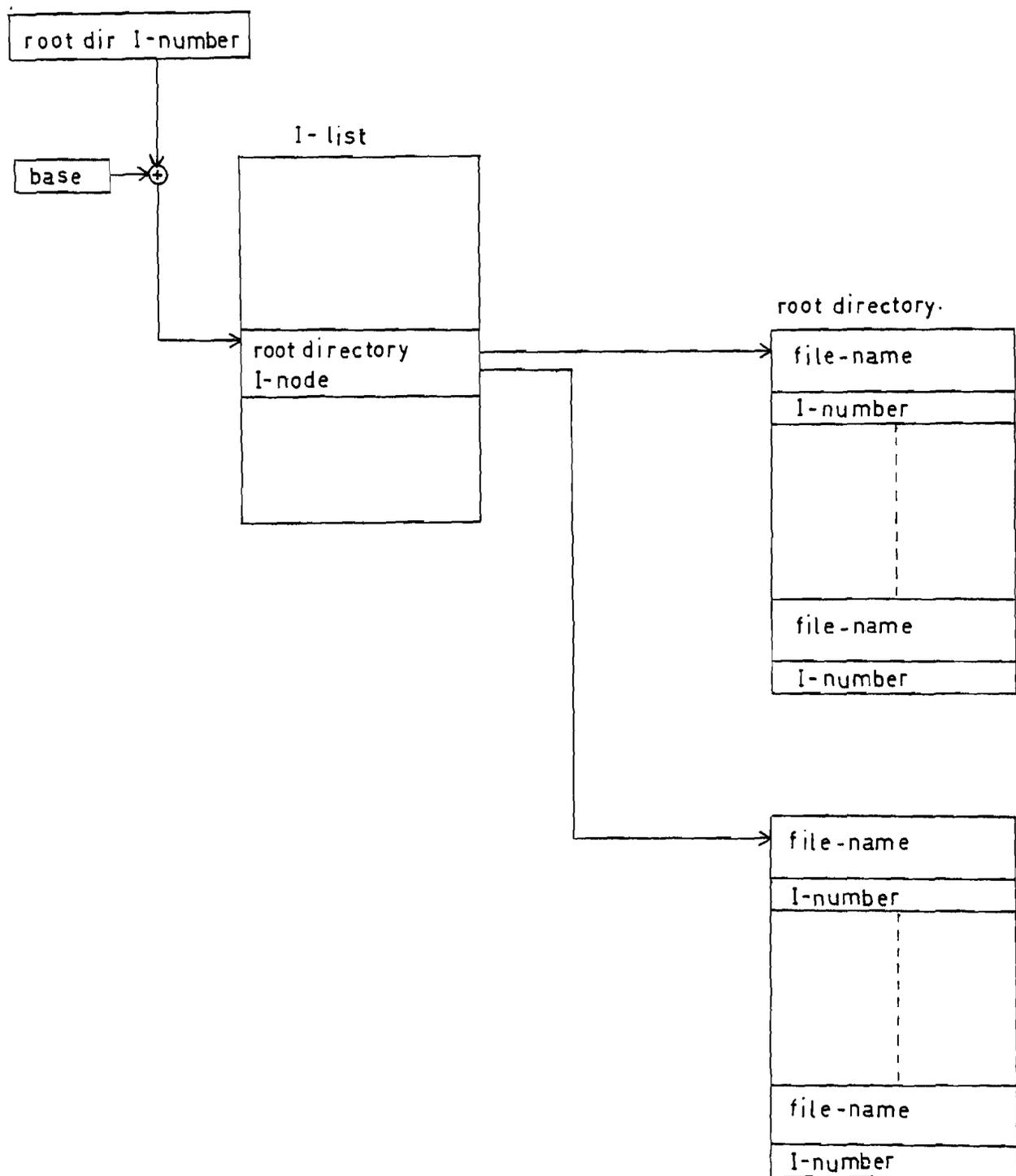


Figure 3.8: Directory mechanism.

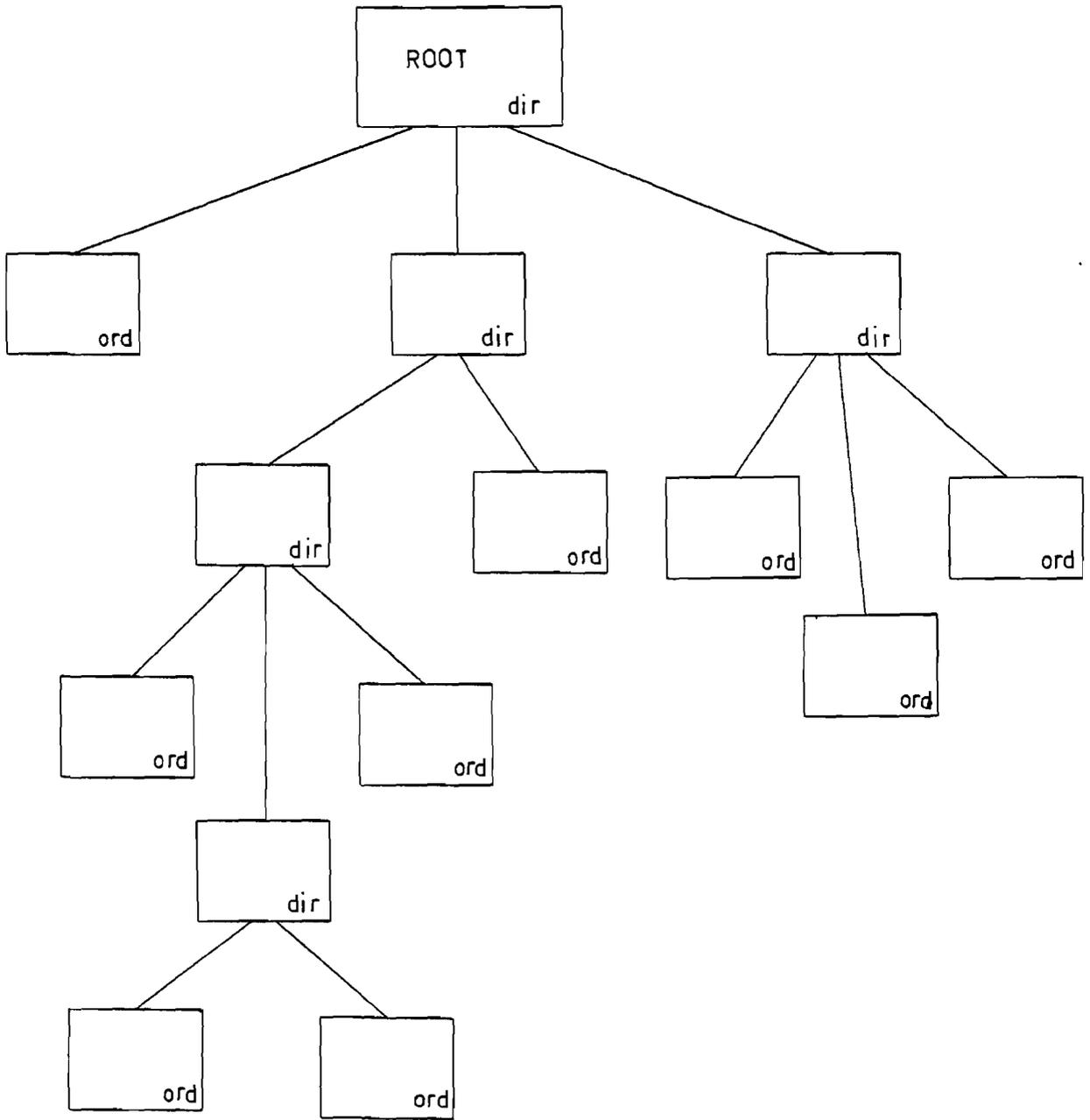


Figure 3.9: Tree structure.

### 3.4. Free space administration.

In order to prevent the controller from writing different files on the same physical location of the disk, it has to keep track of the available space on the disk. Furthermore, it is this part of the DOS that is responsible for allocating disk-space to a new file and re-allocation of space that becomes available on the deletion of a file.

Looking at the UNIX operating system again, one encounters a strategy of blocks containing 50 pointers to free blocks. These pointer blocks form a linked list.

The alternative to this scheme, found in many Disk Operating Systems is the use of a bit-map. A bit-map is an array of bits, each corresponding to one sector or block on the disk. The value of a bit indicates whether the corresponding block is free or not.

The following two paragraphs deal with the advantages and disadvantages of both methods. As a result of this comparison, a choice for either method will have to be made.

#### 3.4.1. Free block pointer list.

Two disadvantages of the UNIX free block pointer list stand out:

1. On starting with an empty disk, pointers to all free blocks on the disk have to be initialized. This leads to a tremendously long free block pointer list. Alternatively, one could start off with a limited list and add more blocks to the list as disk usage increases.
2. Whenever space has to be re-allocated, for instance after deletion of a file, the pointers to the freed blocks have to be added to the list.

The process of re-allocation can best be illustrated by an example.

Assume the controller has a pointer block to the free blocks on the disk in its workspace. This particular block can be considered as the current free block pointer. Part of the blocks pointed to has already been allocated to different files and therefore marked null. The other part is still available as free space. Refer to figure 3.10.

Two situations can occur at this point:

1. A new file is created and space has to be allocated for this file. The number of blocks to be allocated depends on the file type. Say ten blocks have to be assigned to this file initially. The controller starts looking in its current free block pointer block until it encounters a non-zero pointer which is subsequently assigned to the new file. After the

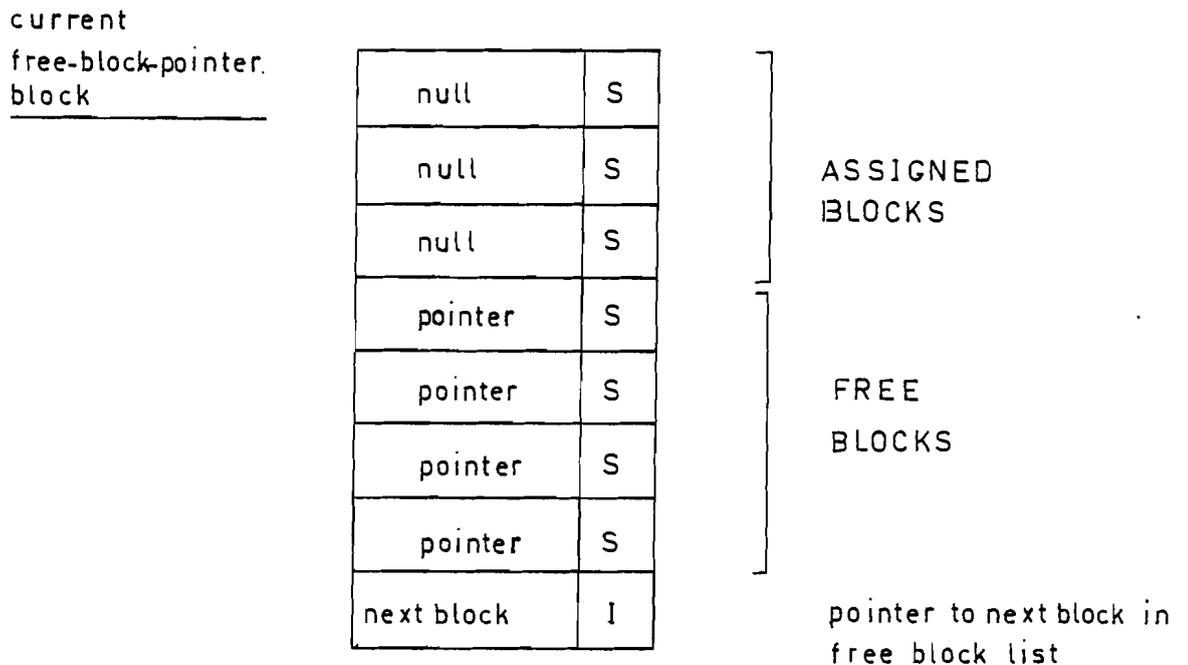


Figure 3.10: Current free block pointer.

assignment, the pointer in the free block pointer block is set to zero. After assigning the last pointer in the free pointer block, the controller will encounter the indirect pointer which points to the next block in the free block list. This block is then loaded in the controllers workspace and becomes the new free block pointer block.

This algorithm is straightforward and causes no serious problems.

2. A file is deleted and the blocks previously occupied by this file have to be registered as free blocks. If the number of blocks involved is less than the number of empty entries in the current free block pointer block, no problem occurs. The pointers to the blocks returned by the deleted file are recorded in the current free block pointer block.

If however the number of blocks to be freed exceeds the number of unoccupied entries - and this will usually be the case - then a new element in the free block pointer list will have to be created. The problem with this new element is, where to locate it on the disk.

From this example we learn that the major problem using a free block pointer list lies in the expansion of this list. We offer three alternatives to overcome this problem:

First of all, one could use one of the blocks turned free by the deleted file as the next block in the free block pointer list. Conveniently this would be the block following the last block that could be placed in the current pointer block.

This would however lead to a dispersed free block pointer list since the block thus added could be located anywhere on the disk. It is considered good policy to try and confine this list to a restricted area of the disk. This way long access times for allocating free space to a new file can be avoided.

Secondly, one could use a double pointer list, meaning every block in the list contains a pointer to its predecessor and a pointer to its successor. Thus a rather fixed structure of pointer blocks is created which can be restricted to a predetermined area of the disk.

Thirdly, a fixed number of contiguous blocks could be reserved on a restricted area of the disk, similar to the Index-list described in a previous paragraph. This method however comes very close to that of a bit-map, every bit being replaced by a 4 byte pointer.

Summarizing one can say the first method leads to a dispersed free block pointer list which is quite undesirable. The second method requires a separate administration of blocks that can be attached to the free block pointer list. This separate administration can be conceptually simple, e.g. a bit-map, but the extra overhead involved for expanding the list, is considered a major disadvantage. The third solution finally, involves massive waste of disk space since a 4 byte pointer is reserved for every physical block on the disk.

In figure 3.11. the three different ways to organize the free block pointer list are drawn.

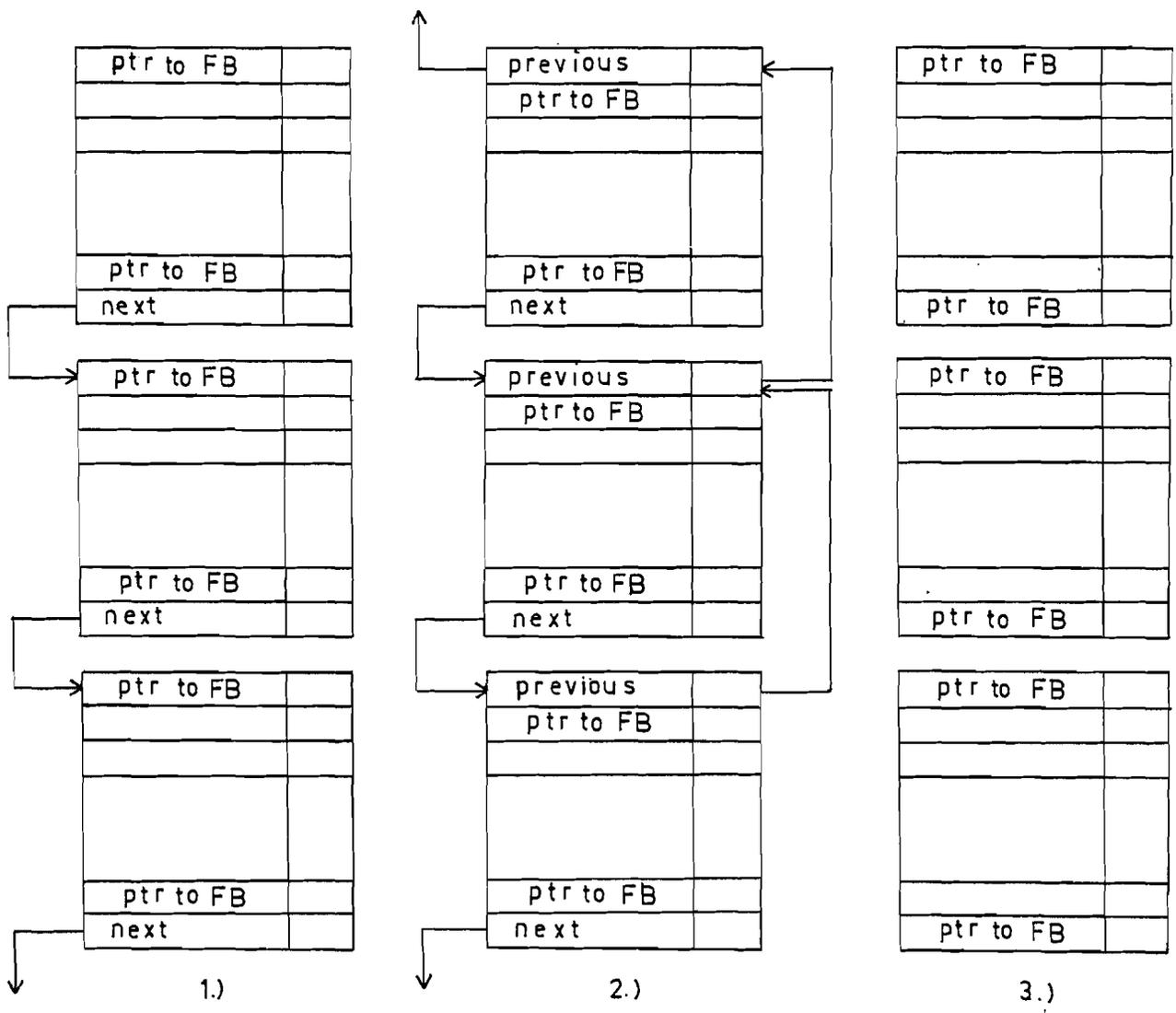


Figure 3.11: Free Block List.

3.4.2. Bit-map.

The use of a bit map for free space administration is quite common, due to its conceptual simplicity. Since every block on the disk requires only one bit of administration, the overhead is minimal. The most predominant disadvantage of a bit map solution, lies in the required calculation from bit position within the bit-map to actual block address. Furthermore it is difficult to define a fast algorithm for efficient allocation and re-allocation of free space.

From the previous discussion, it is obvious that the bitmap method will be used in this controller, as opposed to the free block pointer list. The processing power of the controller is sufficient enough to overcome the problem of translating bit position within the map to physical disk address.

Concluding this section of chapter 3, I would like to state that some features of the UNIX file system are not present in the concept described here. Partially this is due to the fact that modifications were considered necessary to meet the requirements of a controller capable for operating in a non-UNIX environment, partially it resulted from pragmatic choices made to enable an easier implementation of the file system on the controllers CPU.

### 3.5. Command Structure.

The most essential part of a Disk Operating System, from the user point of view, is the command set it offers. The scope of these commands determines the degree of intelligence of the controller as it manifests itself to the user.

In order to reach a functional set of commands to be implemented by the DOS, a few requirements the controller has to meet, have to be kept in mind:

- Function as background storage for user and system files.
- Easy and fast reading and writing of files.
- Both sequential and random access.
- Support of a logical file structure for the purpose of random access.
- The set of commands offered must be as small as possible for ease of use yet sophisticated enough to perform any manipulation the user wishes to do with his files.

Some of these requirements are contradictory to a certain extent so compromising need be done.

In order to obtain a general purpose set of commands, a survey was made on existing operating systems and the set of commands they offer. This survey resulted in a command set, described in the following section of this chapter, which is considered to be a reasonable compromise between complexity of implementation and file manipulation capability. Two different classes of functions are distinguished, file handling commands - covered in paragraph 3.5.1. - and directory commands - covered in paragraph 3.5.2.

#### 3.5.1. File handling commands.

Below, a list of available file handling commands is given. Every command will be elaborated upon by giving a functional description, the result of the operation following the command and a list of possible error conditions.

- Create
- Open
- Close
- Delete
- Read
- Write
- Recover
- Seek
- Insert
- Erase

## Create.

**Function** : Creation of a new file by assembling its Index-node. Information regarding the users identity must be supplied for protection information and access-right validation.

**Result** : The create command results in the creation of an Index-node containing all known information on the file created. The Index-node is added to the Index-list for future reference. Furthermore, an entry in the users current directory is made. If necessary, space for either the new Index-node or the directory entry must be allocated.

**Error conditions** : 1. Illegal filename. The filename and file-type combination supplied by the user, already exist in his directory.  
2. No space. The physical space required to create the new Index-node and directory entry is not available.

## Open.

**Function** : The open command activates the file specified by the user.

**Result** : Activation of a file means the Index-node of the file is fetched from the Index-list and added to the active Index-node table in the controllers workspace. Access to this active Index-node table is obtained through the users open file table. The open file table consists of the users file names which are currently open and an offset into the active Index-node table.  
This substructure allows for fast access to the file for future reference.  
Figure 3.12. illustrates this scheme.

**Error conditions** : 1. File already open. Filename and type is encountered in the users open file table, indicating it was opened previously. This is not an error in the true sense since the file will be open after issuing the command.  
2. Illegal access. User has no legal access rights to the file he specified.  
3. No file. Filename specified by the user is not present in his directory or any of the directories he has access to.

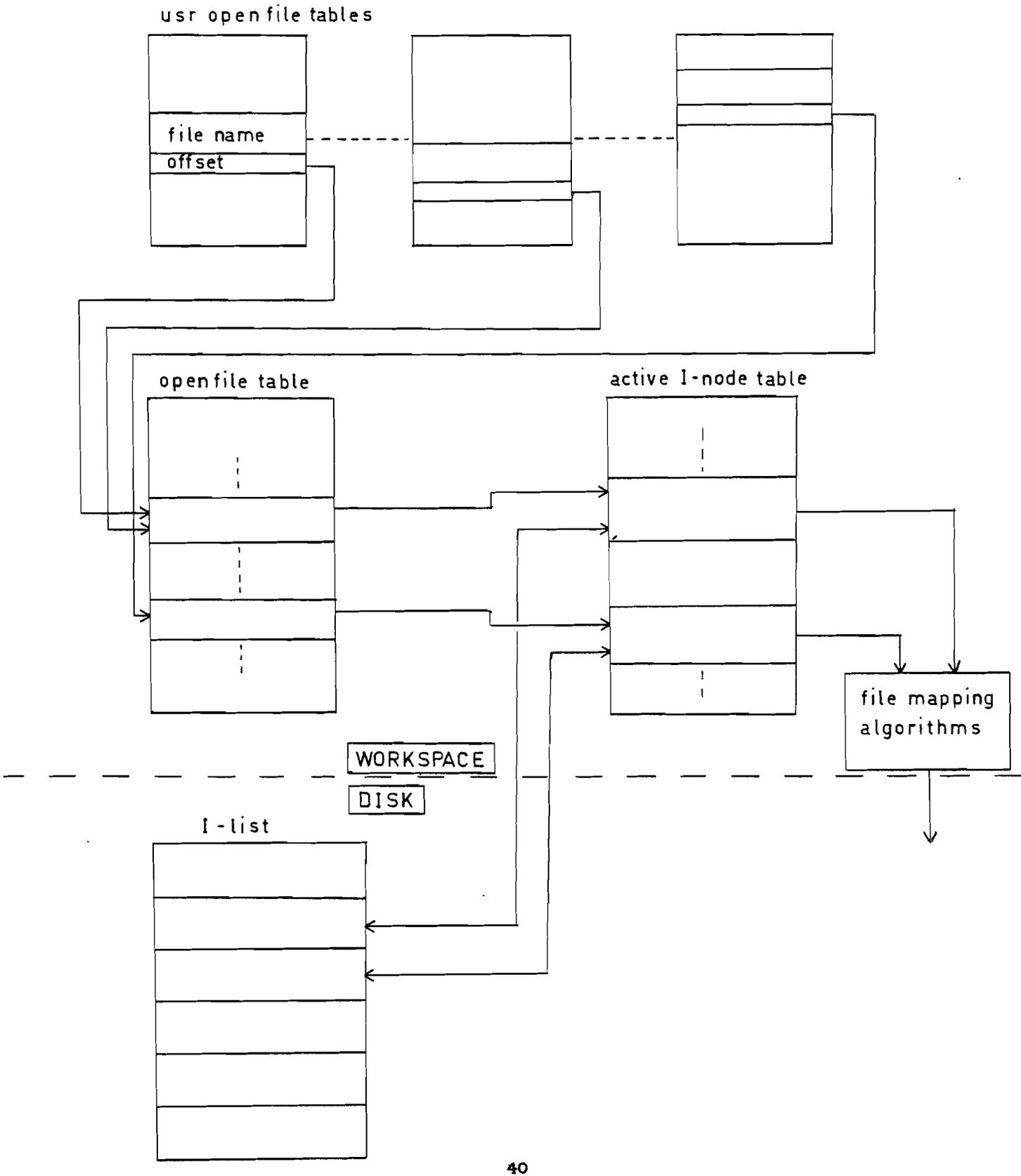


Figure 3.12: File access substructure.

Close.

Function : Closing a file, i.e. remove the access mechanism to the file.

Result : As a result of the close command, the entry in the users open file table is marked "deleted". Thus the information stored in the Index-node is maintained. However, opening of another file by the same user can result in the entry in his open file table being overwritten.

Error conditions : 1. File not open. Files which were not opened can't be closed. Again, this is not an error in the true sense since the file will be closed after the command is given.  
2. Illegal access. User tries to close a file he has no access rights to.  
3. No file. File specified is not present in the users directory. Note the difference between this error condition and number 1. 'File not open' means the file exists but was not open. 'No file' means the file does not exist within the users' scope.

Delete.

Function : Deletion of a file, specified by the user, from his directory.

Result : Deletion of a file is actuated by marking its Index-node as "deleted". Thus the Index-node is maintained in the hierarchy but is not available. The space occupied by a deleted Index-node can be overwritten as the result of a subsequent creation of a new file.

Error conditions : 1. Illegal access. User has no access rights that allow deletion of a file. Files can only be deleted by their owner.  
2. No file. File specified cannot be found in the users directory. In essence this is not a real error condition since the result of issuing the command will be the absence of the specified file.

Read.

Function : Read a logical record from the specified file. The record is obtained from the controllers workspace and transferred to the

Host. If the requested record is not present there, part of the file is read from the disk and stored in the controllers workspace. The requested record can subsequently be extracted using the record size information contained in the Index-node.

**Result** : The result of a read operation is the transfer of a file record in sequential order from the file specified. Since files are stored on disk as contiguous blocks of data, the division of a file into logical records has to be done by the controller. Thus, files are always transferred through the controller buffer.

**Error conditions** : 1. Illegal access. User does not have the proper access rights to read the specified file.  
2. No file. Filename specified cannot be found in the users directory.  
3. File not open. File wasn't opened prior to read command.  
4. End of file. Either the file being read is empty or the last record of the file has been read on a previous occasion.

**Write.**

**Function** : Write a logical record on disk. The record is added towards the end of the specified file.

**Result** : The record to be written is transferred to the controller buffer. Physical writing to the disk is not necessarily done immediately.

Writing from buffer to disk is done whenever a fixed number of blocks - minimum one - can be written at once and bufferspace has to be made available to service another request. As a matter of course, all active records of a file, present in the controllers buffer but not written to disk yet, are transferred to disk prior to a close or read command execution on the same file.

**Error conditions** : 1. Illegal access. User has no write access to the specified file.  
2. No file. Filename specified is not present in the users' directory.  
3. File not open. File not open i.e. no entry in users' open file table.

4. Not End of File. Present access location within the specified file is not positioned at the end of the file. Since records are added towards the end of the file, a seek to file end has to precede the write command.
5. No space. Space required to store the record is not available on disk or in the controller buffer.

Recover.

Function : Recovering a previously deleted file. The information contained in the recovered file can either be intact or distorted. Information concerning the integrity of the recovered file is supplied to the user after execution of this command.

Result : The result of this somewhat unusual command, is based on retrieving the Index-node of the file to be recovered, from the Index-list. As described previously, deleting a file results in its Index-node being marked "deleted". This means it is still present in the Index-list where it can be found by the controller, unless the creation of a new file caused the entry to be overwritten by the new Index-node. In this case, recovery of the deleted file is impossible.

The Index-number of the deleted file has to be obtained from the recover file which was specially created for this purpose, since the normal entry in the directory has been removed.

There is yet another complication concerning the disk space previously occupied by the deleted file. Upon deletion of the file, the blocks it occupied were marked as being free. When the file is recovered, a comparison between the address information in the Index-node and the free space administration table has to be made. If the blocks of the recovered file are still registered as free blocks, they may still contain the proper information, though this is not necessarily the case. This however is an uncertainty one has to accept. Therefore the user, after issuing a recover command, should also verify the information contained in the recovered file.

From the above one can conclude that this is a very complicated command with uncertain results. When implementing a command like

this, one has to consider whether the advantages compensate for the problems it incurs

Error conditions : 1. No file. File name specified is not present in the recover file, meaning it was not deleted.  
2. No I-node. Index-node of the specified file has been overwritten.  
3. No data. One or more blocks previously occupied by the file have been re-allocated to other files.

Seek.

Function : The seek command moves the window, a pointer to a record within a file, to a specified position. Thus random access within a file is possible.

Result : The window of a file points to a record within the file. The value of this pointer is stored in the users open file table. Normally the window is incremented after every read or write operation. Thus sequential access is achieved. By means of the seek command, the window can be moved to an arbitrary position within the file.

Error conditions : 1. Illegal access. User has no read access rights to the file he specified.  
2. No file. Specified file cannot be found in users open file table. Either it was not opened previously or it doesn't exist at all.  
3. No record. Specified record number is out of range. The window is set to EOF as a result of this.

Insert.

Function : Insertion of a record in a file at the location pointed to by the window.

Result : As the result of an Insert operation, a record is added to a file at the current position of the window. Records following the inserted record are shifted one position as is illustrated in figure 3.13.

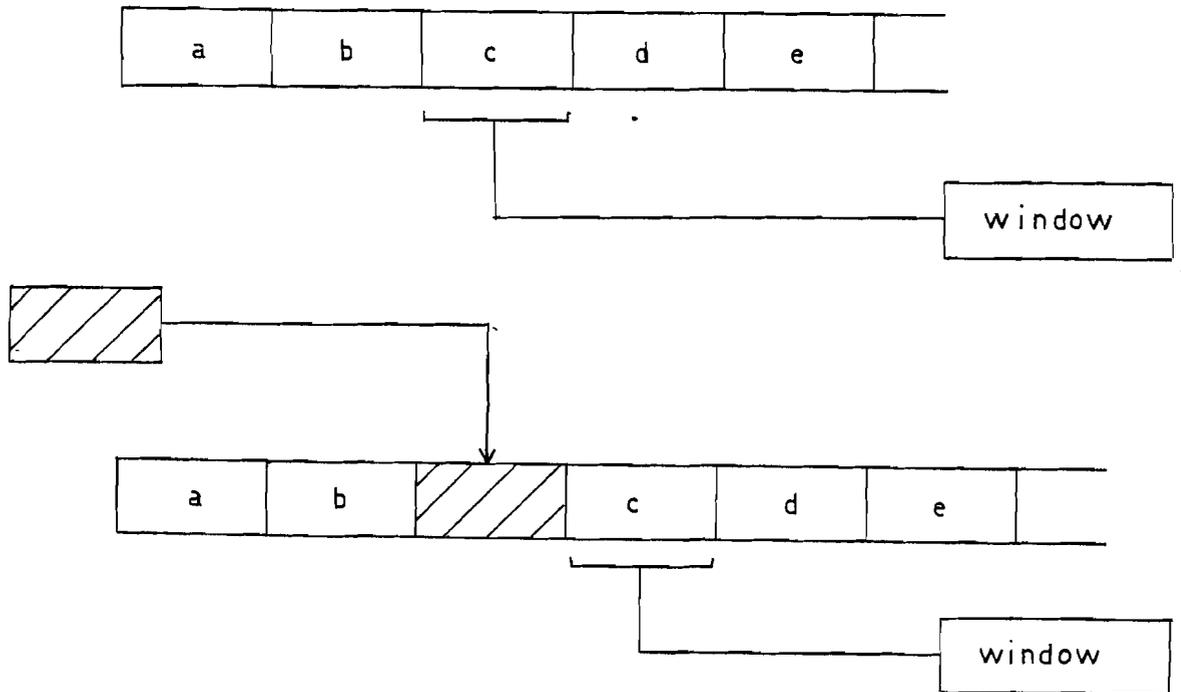


Figure 3.13: Record insertion.

After insertion of a record, the controller will perform a readback operation to verify correct operation. Afterwards, the window is moved to the next record of the file. This enables multiple insertions to be performed sequentially.

- Error conditions :**
1. Illegal access. User has no write access rights to specified file.
  2. No file. Specified filename does not occur in users directory.
  3. Not open. File exists but is not open.
  4. No space. Physical disk space required to store the inserted record is not available.

**Erase.**

**Function :** Erasure of a particular record from a specified file, the erased record being the record currently pointed to by the window.

**Result :** The erase command is the functional counterpart of the insert command. Therefore, the result is opposite to that of an insertion.

Only, an erasure does not affect the position of the window within the file. Thus, after an erasure, the window will be at the next record position. This allows for multiple record erasures.

Error conditions :

1. Illegal access. User has no write access rights to the specified file.
2. No file. File does not occur in users directory.
3. Not open. File specified was not opened.
4. End of file. Window has reached end of file. No more erasures can be done unless a seek is executed to a previous record position. The same error condition occurs when the file is empty.

### 3.5.2. Directory commands.

The second set of commands involves manipulation of the directory mechanism in the controller. There are two possible ways to allow the controller access to the directory. Both possibilities will be discussed briefly.

The first results from the fact that directories are just a special kind of ordinary files. Thus, it is possible to perform the same operations on directories as on files. The user can define his own directory command set by manipulating directory files the same way he would manipulate ordinary files. This however requires a detailed knowledge of the controllers directory mechanism. This is contradictory to the purpose of the controller concept. All knowledge concerning file organization should be concentrated in the controller, not in the Host or at the user.

The alternative is to provide a number of directory commands at the same level as the file commands. The scope of these directory commands will have to be limited to requiring information on a file. Alterations in the controllers directory by the user can not be permitted since they would require in depth knowledge of its operation, which is assumed absent at the user level.

Three different directory commands will be supported:

- DIRLIST : Lists the contents of a directory specified by the user. The specification is done by supplying a pathname as described in paragraph 3.3.
- ENQUIRY : Supplies the user with an overview of the information contained in a file's I-node.
- RENAME : Allows the user to alter a file's name.

### 3.6. Host Interface.

To ensure proper communication between the Disk Controller and the Host computer, one has to provide for an interface between the two. In general, two approaches exist:

1. Network solution.
2. Register solution.
3. Bus solution.

### 3.6.1. Network solution.

Taking a network view towards the communication, the controller is regarded as one of the devices connected to the Host computer by a link. See figure 3.14.

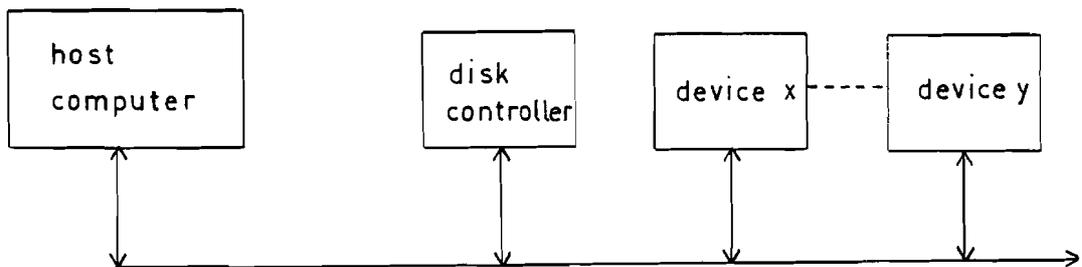


Figure 3.14: Network model.

Every device in the network has its own address and monitors the communication channel to see whether there is a message for him distributed on the network. If so, the message is read and interpreted.



SOM : start of message.  
ADR : device address.  
TYP : message type.  
LEN : message length.  
OPC : operation code.  
SOD : start of data field.  
DATA : data field.  
CTRL : control code.  
EOD : end of data field.  
EOM : end of message.

Figure 3.15: Message format.

In a network environment, messages between members are packed in blocks of variable length according to a certain format. A possible message format might be what is shown in figure 3.15.

The advantage of this solution is twofold. First of all, it's very flexible in that the message blocks can contain any information desired. The protocol only provides for sending and receiving messages, regardless of their contents. Secondly, the physical transport can be done serially or parallel without any implications for the protocol. In both cases, the transfer would be asynchronous, inflicting no time constraints on either host or controller.

As with any solution, there are some disadvantages as well, one of them being the relatively large amount of overhead involved. This overhead has a negative impact on the link's performance.

### 3.6.2. Register solution.

A far simpler solution compared to the previous one, is the use of a set of parallel I/O ports, one for data transfer and one for command/status transfer. Thus, the Host sees the controller as a peripheral located somewhere in his memory map.

Though this approach is conceptually simple, it offers little flexibility.

To obtain a maximum performance level, the data-channel at least should allow for DMA transfers.

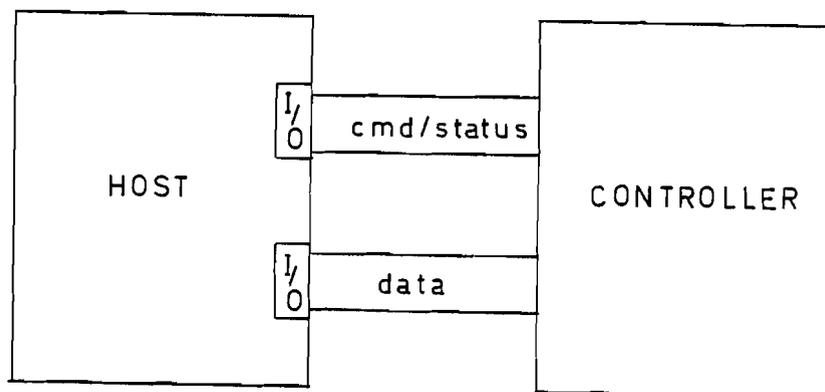


Figure 3.16: Peripheral connection.

### 3.6.3. Bus solution.

The third possibility is to connect the controller directly to the Host computer's bus system. This enables the controller to directly access the Host computer's memory. Memory to memory transfers from the controller's buffer to the Host's workspace and vice versa.

Clearly this means that the controller has to be designed for a specific Host computer system or at least a specific bus system, e.g. Motorola's VME or Intel's Multibus bus.

The design covered by this report does not however imply in depth knowledge of a certain bus system. Therefore this possibility is dropped.

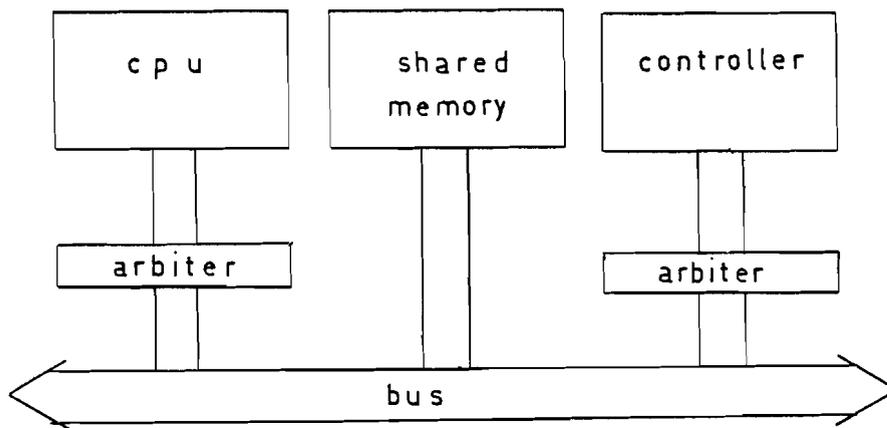


Figure 3.17: Bus solution.

### 3.6.4. Conclusion.

Due to its conceptual simplicity and reasonable compatibility to various computer systems, the register solution was chosen in this particular design.

## Chapter 4. CONTROLLER HARDWARE DESIGN.

### 4.1. Introduction.

The hardware required for a controller described in this report is more complex than that of a conventional disk controller which functions as a mere peripheral. The blockdiagram shown in Figure 4.1. gives a general idea of the scope of the hardware for an intelligent Winchester Disk controller.

Description of this hardware system will be done in five sections:

- Processor system.
- Memory system.
- Host interface.
- Controller module.
- Disk drive interface.

### 4.2. Processor system.

The processor system constitutes the central part of the hardware and serves a dual purpose.

The most predominant task to be executed by the processor system is the transport of data between Host and Disk Drives. This requires massive I/O, preferably using Direct Memory Access (DMA). Furthermore, the processor system should be able to meet the speed requirements of both Host and Disk Drives in order to avoid unnecessary delays.

The second task imposed upon the processor system is the execution of the controller's Disk Operating System (DOS). To this extent, it requires flexible addressing techniques, memory management capabilities, a strong set of instructions and fast execution speeds.

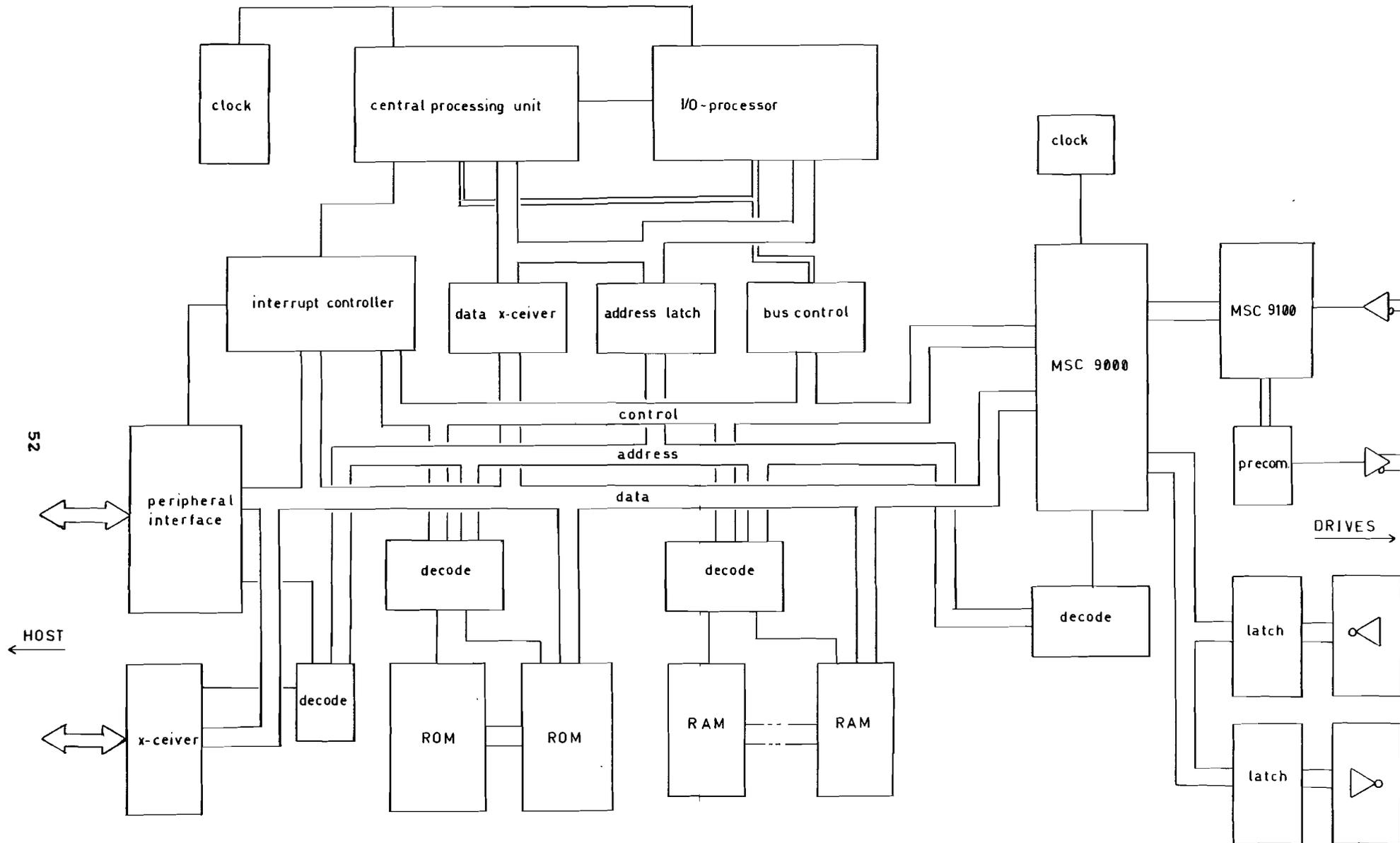
These two obviously distinct functions, each imposing special demands on the processor's capabilities, lead to the choice of a multi-processor system in which every task has a dedicated processor.

#### 4.2.1. Parallel- versus Co-processing.

The use of more than one processor calls for some sort of communication and collaboration between the different processors involved. Two different approaches can be used, either parallel processing or co-processing.

- Parallel processing.

Parallel processing means two or more processors, either of the same or of a different type, are simultaneously executing part



of a task. Thus the work-load is shared amongst several processors, increasing the overall throughput. Peripherals, memory, bus etc are shared resources, meaning they can be used by any of the processors in the system, only not at the same time. Communication between different processors is performed through shared memory.

The use of shared resources requires strict synchronization of concurrently running processes, as well as arbitration of simultaneous resource requests. Use of the shared bus is usually handled by a so called bus arbiter which allocates the bus to a requesting processor as soon as it becomes available. All bus requests are coordinated by this bus arbiter. The processor which gains control of the bus is usually also given access to the other shared resources such as memory and peripherals.

As a rule, parallel processing is advantageous in the execution of tasks that require considerable processing. The extra investments in arbitration hardware will have to be weighed against the increase in performance.

Refer to figure 4.2.

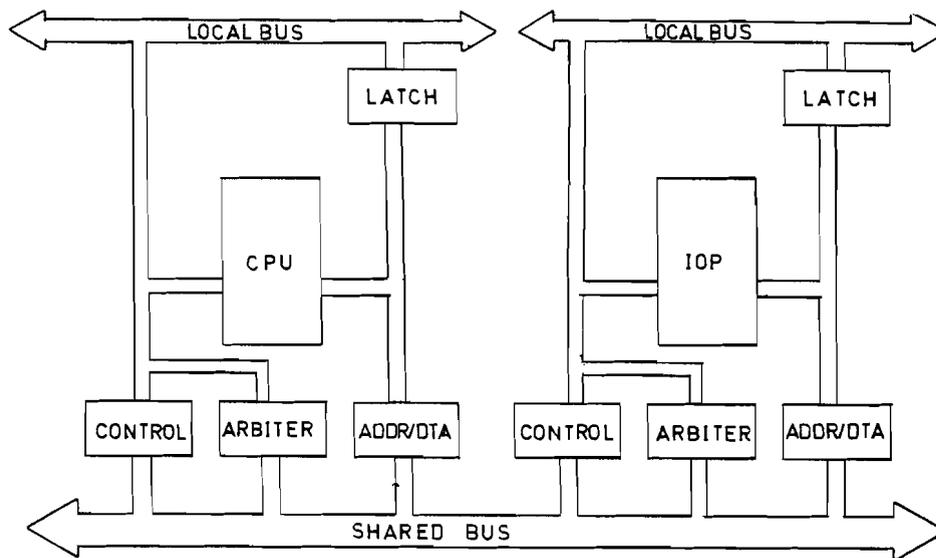


Figure 4.2: Parallel processing.

- Co-processing.

In a co-processing situation, two or more processors are each assigned a specific part of a task. However, execution of these sub-tasks is done sequentially rather than simultaneously, thus only one processor is active at a time and has complete control of all the resources. Obviously, the processors involved are of a different nature and especially fit for their specific sub-

tasks.

Transfer of control from one processor to another is usually done after completion of a sub-task, unless another processor requires immediate service. To achieve this, some sort of communication between the different processors in the system must exist.

A situation found in most co-processor systems is that of a master-slave hierarchy in which one master and one or more slave processors exist. Whenever the master wants a slave processor to execute a task, it activates the desired slave processor. Once activated, the slave processor takes control of the bus, executes its task and signals the master upon completion. Exchange of information between processors is done through parameter blocks in memory.

The advantage of co-processing lies in its conceptual simplicity compared to parallel processing. The price for this simpler approach is a lower performance level.

Refer to figure 4.3.

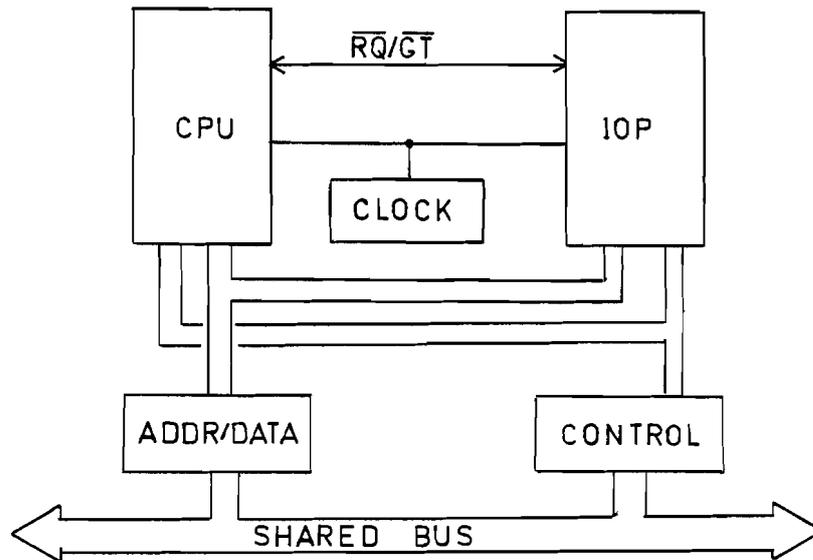


Figure 4.3: Co-processing.

For the design at hand, a co-processing configuration of a 8086 16 bit Central Processing Unit (CPU) and a 8089 I/O processor (IOP) was chosen based on the following arguments.

Transferring large quantities of data to or from the Disk under

DMA requires constant use of the system bus. During these transports, which are assumed to consume a large percentage of the overall time, no other processor can use the shared bus, thus no other processor can operate outside of its local environment. Local operation of the IOP would be no solution since the Disk Operating System, running on the CPU has to operate on the data obtained from Disk. Therefore this data has to be stored in shared memory which is only accessible through the shared bus.

This heavy I/O load on the shared bus makes parallel processing impractical, especially when the extra hardware needed for a parallel processing system is taken into account.

#### 4.2.2. I/O processor description.

The Intel 8089 I/O processor is a single chip, high performance, general purpose I/O system, equipped with two independent I/O channels, each combining CPU capabilities with a flexible DMA controller. Each channel can execute a Task block program, using 53 instructions specially designed for efficient I/O program execution. On top of that, every channel can perform DMA transfers with simultaneous data manipulation.

The architecture of this IOP and its capacity to operate in conjunction with a 8086 CPU in a co-processing environment, make it very suitable for this application.

#### 4.2.3. Central Processing Unit description.

The 16 bit 8086 CPU, family member of the IOP, was chosen for this application, rather than the 8088, a similar CPU with a 8 bit data bus. Although most disk systems in the Winchester field currently still operate on 8 bit data units, the possibility of future extension to 16 bits was taken into account. Besides, the 8086 is perfectly capable of handling bytes and - can obtain a higher performance level than all existing 8 bit micros, including the 8088.

Finally, the ease of applying the 8086 and 8089 in a co-processing environment, lead to this choice.

In figure 4.4. the processor system is shown schematically. Circuit diagrams are all concentrated in appendix A of this report and references to this appendix will be made frequently.

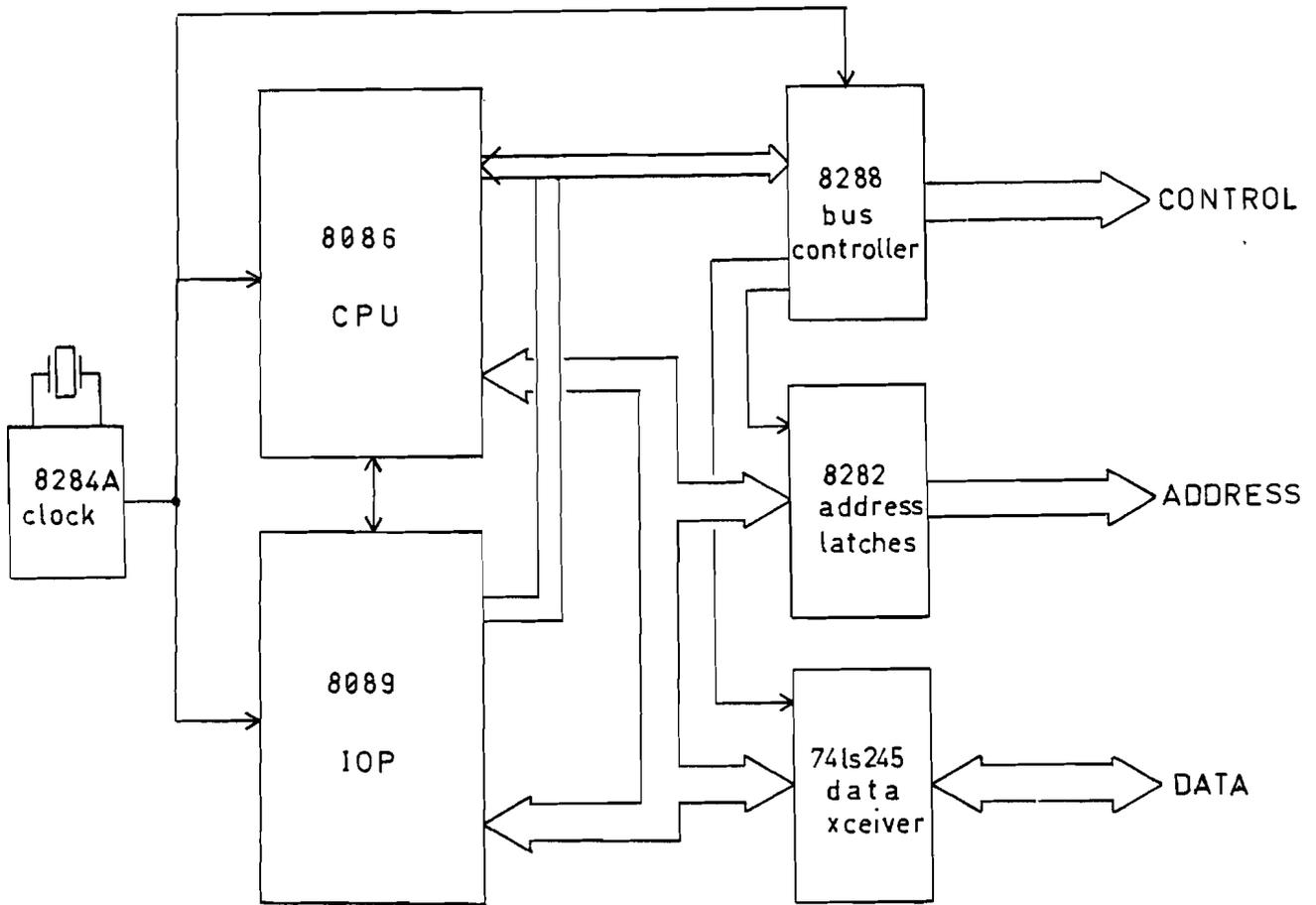


Figure 4.4: Processor System.

#### 4.3. Memory System.

The memory system required for the controller can be divided in two parts, Random Access Memory (RAM) which constitutes the controllers workspace and Read Only Memory (ROM) which stores the required software permanently, as well as the necessary Reset and interrupt vectors.

In order to construct a read/write memory system, either static or dynamic RAM chips can be used. The use of static memory elements results in a conceptually simple, fast and reliable memory system. However, due to the fairly large power consumption and lower packing density of static RAM chips, dynamic memory elements are usually applied for larger memory systems. Power consumption is approximately 10 times, packing density 4 times better compared to static memory.

The major disadvantage of dynamic memory is the fact that it has to be refreshed regularly to avoid loss of information. During these refresh periods, the memory is not available for the CPU. Furthermore, the timing requirements for the refresh actions are rather strict and somewhat complicated. Thus the use of a specially designed Dynamic RAM Controller is necessary. The purpose of this controller is to take the burden of refreshing the dynamic chips at regular intervals away from the CPU. As a result, the memory system becomes transparent for the CPU and acts as static RAM but only to a certain degree. If the CPU wants to access the memory during a refresh cycle, it is delayed by means of the insertion of WAIT states until termination of the refresh cycle. Thus, dynamic RAM may affect the processor system's performance.

Several possible configurations for the RAM system were examined. Two of the most interesting possibilities are discussed below to give an impression of what options exist. Memory sizes of 64 Kwords and 32 Kwords were assumed respectively in case of a Dynamic and a Static system. Memory systems larger than 32 Kwords are generally not made using static RAM.

#### 4.3.1. Dynamic RAM system.

Designing a dynamic RAM system requires the choice of a DRAM chip on the one hand and a compatible DRAM controller on the other. Since the processor system is made of Intel devices, an Intel DRAM controller seems an obvious choice. The 8207 Advanced Dynamic RAM controller was taken in conjunction with 2164 DRAM chips.

#### 2164 DRAM.

The 2164 DRAM is a 64K x 1 bit dynamic RAM chip, built around four matrices of 128 rows by 128 columns. Row and Column addresses, each 8 bit wide, are multiplexed through an 8 bit wide address bus. For use in a 16 bit wide datapath system, 16 chips is a minimum requirement, yielding a 64 Kword memory system.

#### 8207 Advanced Dynamic RAM Controller.

As is suggested by the name, the 8207 is a very complex controller module. Refer to figure 4.5. for a functional block diagram of this controller.

The ADRC is a peripheral chip which takes care of the addressing, refreshing and required drive capability for 16K, 64K and 256K dynamic RAM chips. It provides a dual port function allowing two independent processor systems to access the same memory. A built-in arbiter determines the order in which requests are serviced. Furthermore, provisions were made to connect a 8206 Error Detection and Correction Unit (EDCU), yield-

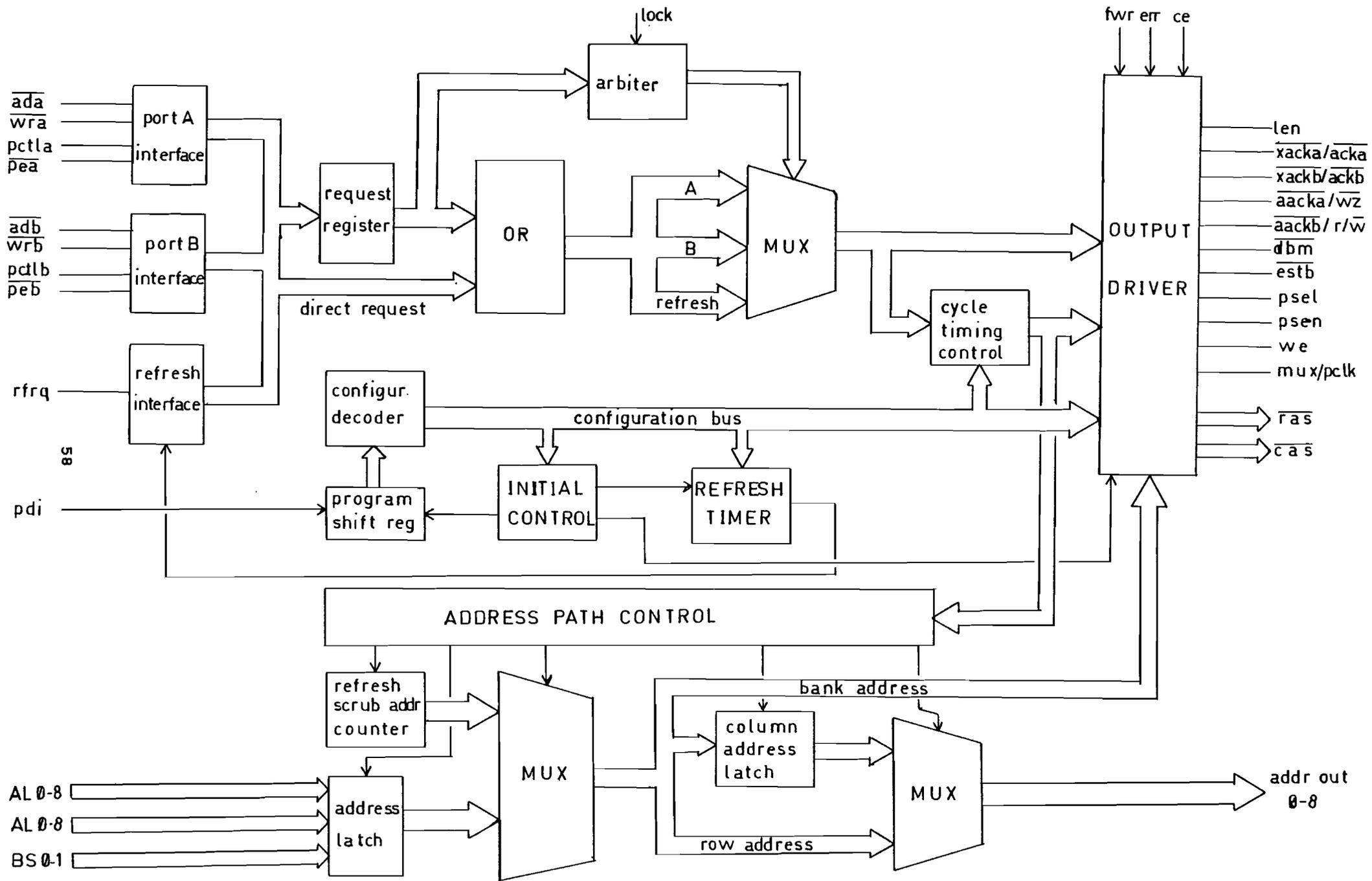


Figure 4.5 Intel 8207 Dynamic Ram Controller

ding a very reliable memory system. It is beyond the scope of this report to list all the possibilities of this DRAM controller.

The use of this DRAM controller in a RAM systems, leads to a very compact design. See figure 4.6. However, the insertion of WAIT states as mentioned before, cannot be avoided using this controller. The negative impact on the IOP's DMA performance is considered a major disadvantage.

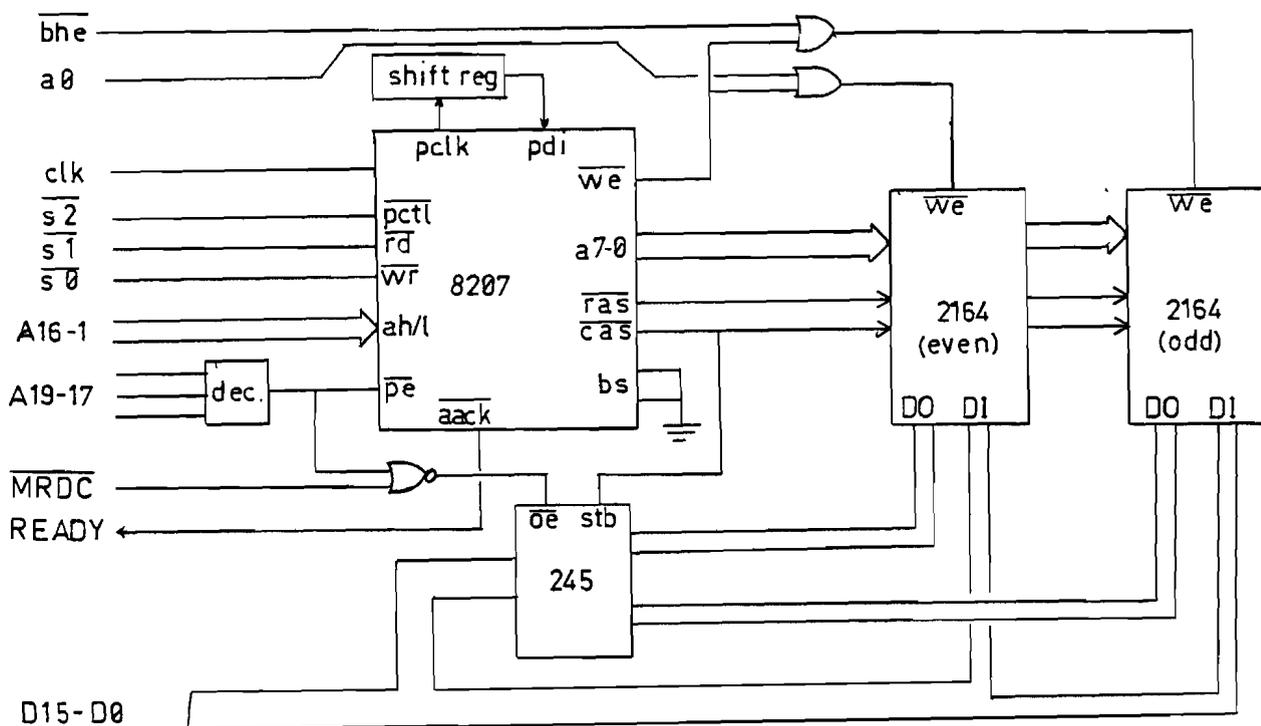


Figure 4.6: Dynamic RAM system.

#### 4.3.2. Static RAM system.

All the problems regarding timing requirements introduced by dynamic RAM chips can obviously be avoided using static memory. The Intel 2167 High Speed 16K x 1 bit static RAM is the highest package density static RAM chip currently available. Due to its built in power-down mode, power consumption is kept within reasonable limits in spite of the fast access times. (100 nsec slow version) A 32 Kword RAM system, made from 32 chips, would require about 8 Watts under normal operating conditions as opposed to 2 Watts for a similar dynamic system.

The advantages of a static RAM memory, simple timing, fast access times, high reliability and easy expandability, lead to the choice of a static RAM memory for this particular design. If however the assumed memory size of 32 Kwords should prove to be insufficient, the choice of DRAM is obvious.

See figure 4.7 and appendix A.

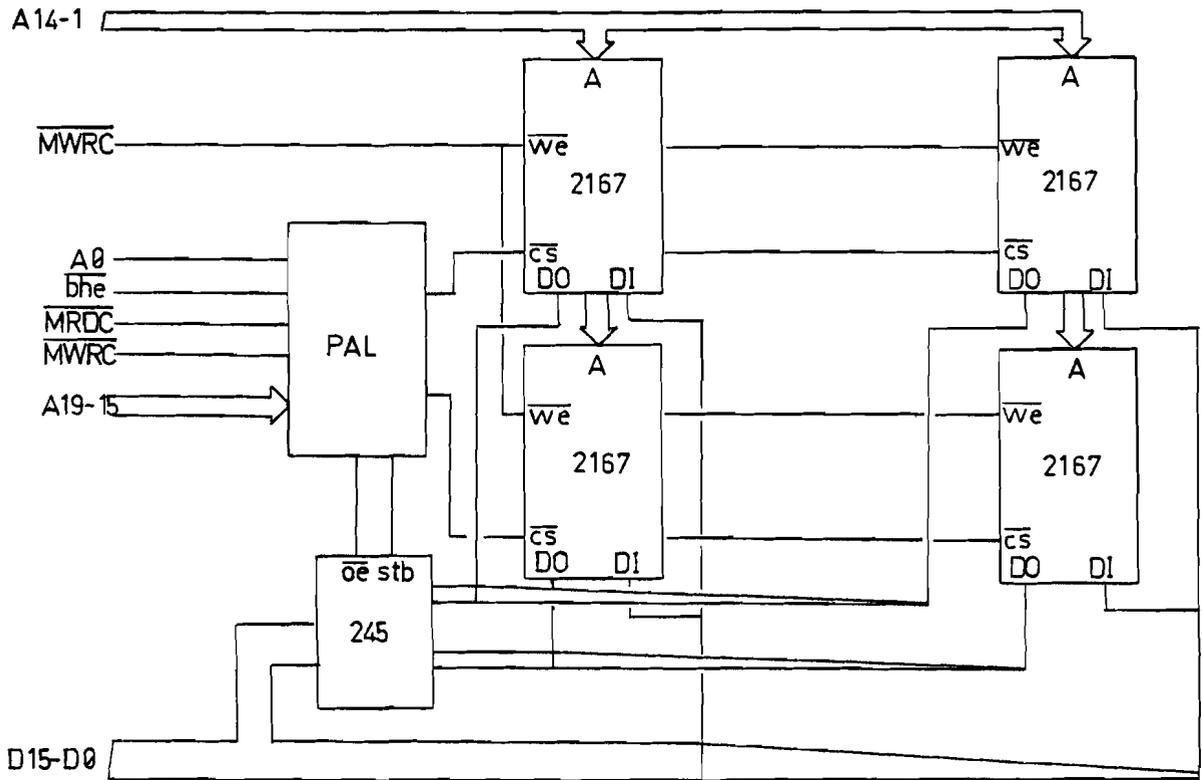


Figure 4.7: Static RAM system.

#### 4.3.3. ROM system.

The ROM system shown in appendix A can be hard-wired to accommodate either 2732 4 Kbyte or 2764 8 Kbyte ROM chips. When equipped with 2764 chips, the circuitry shown offers 32 Kwords of Read Only Memory. Whether this size is sufficient to accommodate the software of the controller is as yet uncertain. Expansion of this ROM capacity is a very simple operation. Alternatively, operational software could be loaded from disk after start-up and stored in RAM. This way, only a bootstrap ROM would be needed. The consequence of this approach is a larger requirement for RAM.

#### 4.3.4. Address Decoding.

Address decoding is covered in this paragraph as it is considered to be an integral part of the memory system. Decoding is done in a way that places the RAM in the lowest region of the processor's memory map. This enables modification of the interrupt vectors located from 80H to 400H by the user. These vectors have to be loaded from ROM immediately after a

system reset.

Since the 8086 CPU, as well as the 8089 IOP, expects its reset vector to be in the high region of the memory map, the ROM memory is placed there.

Rather than using conventional TTL decoders, programmable logic is used for address decoding. Using Programmable Array Logic introduces flexibility in that changing of the memory map can be obtained by simply replacing the PAL with a differently programmed version. Three PALs were used, one for RAM, one for ROM and one for I/O decoding. In figure 4.8. both methods are shown, the conventional way as well as the programmable way.

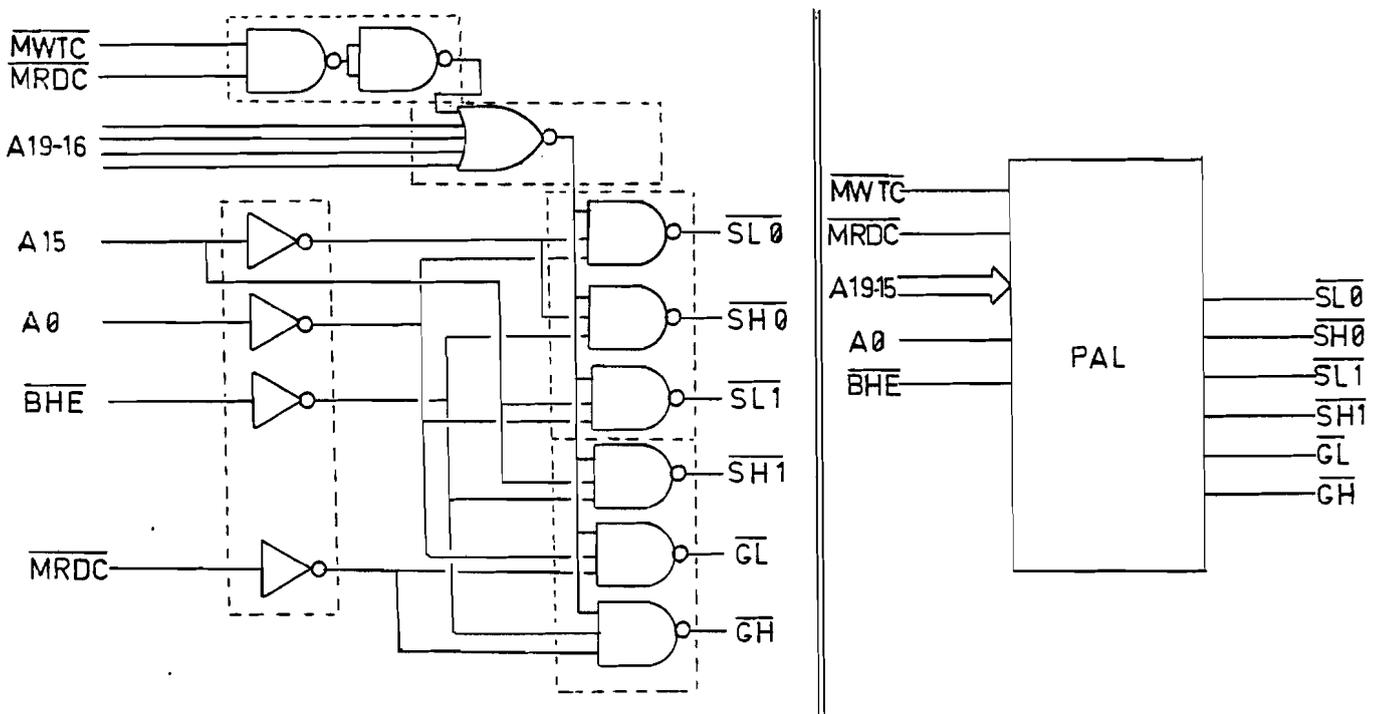


Figure 4.8: Address decoding.

#### 4.4. Host Interface.

As mentioned in chapter 3, a register-like approach was chosen for interfacing with the Host computer. Since the properties of the Host are not known, it will depend largely on the application of the controller whether this kind of parallel interface is suitable.

A block diagram of the interface hardware is given in figure 4.9. below whereas a circuit diagram can be found in appendix A.

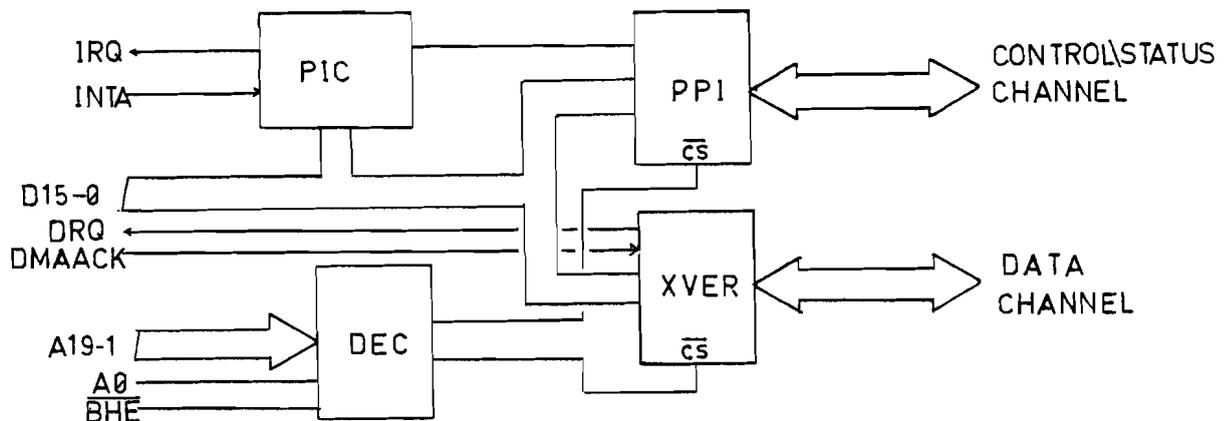


Figure 4.9: Interface block diagram.

The interface hardware is made of two I/O channels, one for strobed I/O using a 8255 Programmable Peripheral Interface device and the second for DMA transfers.

**Bi-directional I/O channel:** Programmed to operate on a strobed I/O bi-directional basis, this I/O channel is used to transfer status and command information between controller and host.

**DMA transfer channel:** Using two data transceivers, a 16 bit wide, bi-directional data channel is created between host and controller for the purpose of transferring file data at maximum speed under IOP control.

The Programmable Interrupt Controller (PIC) shown in figure 4.9 should be considered as part of the processor system rather than the interface hardware. Its purpose is to allow external interrupts to the processor. The PIC is capable of placing a vector on the processors data bus after the first interrupt request from the CPU.

The relatively large number of lines between host and controller - 31 in all, excluding ground lines - stems from the facts mentioned below:

- Parallel I/O
- 16 bit wide data channel.
- Separate data and control/status channel.

Each of these facts add to the throughput capability of the interface and were introduced for that purpose.

#### 4.5. Drive Control Unit Hardware.

The hardware of the controller unit itself is a very essential part in the whole design. Generally, three possible implementations of the functions mentioned in chapter two exist:

1. Standard logic.
2. Programmable logic.
3. Single chip controller.

##### 4.5.1. Standard logic.

Designing a controller using small and medium scale integration logic chips results in a complex and fixed design. The PC board area is generally quite large and modifications, when necessary are difficult to make. Since this method is becoming extinct, no further attention was given to it.

##### 4.5.2. Programmable logic.

As the next step towards modern controller design, the use of programmable logic chips to replace fixed logic should be considered. When carefully designed and programmed, this method could yield a major improvement over traditional designs. However, due to the rise of specially designed Winchester controller chips - see appendix C - this option was not studied. It might prove useful to do so. Refer also to appendix D for a slight impression of designing with programmable logic.

##### 4.5.3. Single chip controller.

To avoid a large number of logic chips, high power consumption and little flexibility, which is typical of a random logic controller design, several semi-conductor manufacturers have come up with VLSI controller chips, capable of performing nearly all the functions of a disk controller. Appendix C gives an overview of controller modules which are or will soon be available. In this report, the Microcomputer Systems Corporation (MSC) 9000 Winchester Disk Controller Module was chosen. This choice was rather pragmatic since most information was available on this particular device. For this reason, the design described here is not claimed to be the optimal solution.

##### MSC 9000 Controller Module.

The Microcomputer Systems Corporation MSC 9000 Winchester Disk controller module contains most of the elements required for disk control as can be seen in the block diagram of figure 4.10

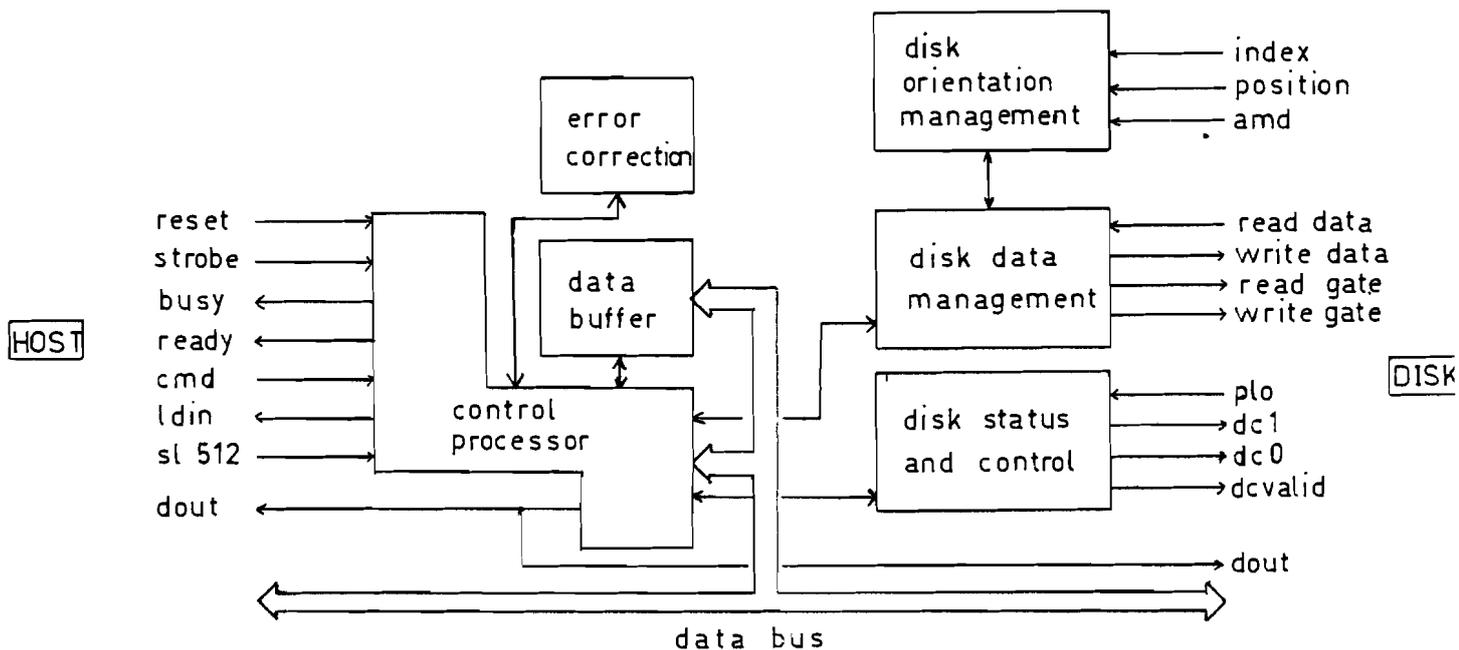


Figure 4.10: MSC 9000 controller module.

Unfortunately, two different classes of Winchester disk drives have to be distinguished, those who have a built in data separator and Address Mark (AM) generator/detector and those who lack these facilities. The latter ones are generally found in the smaller and lower cost Winchester drives whereas high performance drives always provide a built in data separator. Since the quality of the data separator determines to a large extent the data integrity of the drive, this is the only way a manufacturer can guarantee a certain reliability.

Since the object of this graduation project was to design a universal Winchester Disk controller, a data separator and AM generator/detector will have to be incorporated in order to be able to accommodate low-cost drive types as well. For this purpose, Microcomputer Systems Corporation offers a module, labelled MSC 9100 to be used in conjunction with the MSC 9000 to create a complete controller.

At this point a problem evolves. Apparently the MSC philosophy did not foresee the use of its controller device in a universal controller environment. Two different types of MSC 9000 modules are offered, each slightly different in specifications and labelled 9016 and 9056 respectively.

Although the differences are marginal (refer to appendix B), it would be tedious to try to use either of these modules to handle both drive types. It would mean by-passing the MSC 9100 to control Winchester drives with a built in data separator. Though possible, this option will not be covered in this report since it offers little additional insight.

A block diagram of the controller hardware is shown in figure 4.11. The circuit diagram can be found in appendix A.

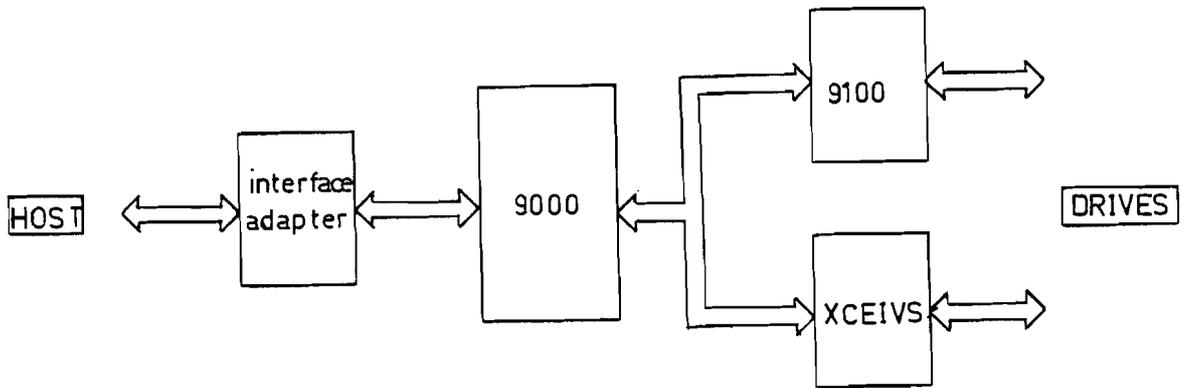


Figure 4.11: Controller hardware.

#### 4.5.4. Processor module interface.

Since the MSC 9000 can be regarded as a processor system in its own within the hierarchy of the controller, it will have to be interfaced with the processor system in some convenient way. Two possible configurations can be considered:

1. SASI bus concept.
2. Special purpose interface.

#### SASI bus.

SASI stands for Shugart Associates System Interface, a byte-wide intelligent bus that interfaces host systems, like the processor system in this controller, to control units, like the MSC 9000. Currently this bus is being standardized by the ANSI X3T9.2 subcommittee under a new name, the Small Computer System Interface. Figure 4.12 shows the physical characteristics of this bus.

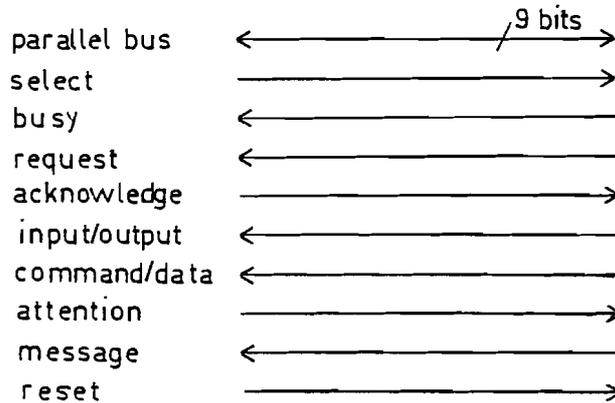


Figure 4.12. SASI bus.

The major advantage of using a bus structure like the SASI bus is the inherent possibility of incorporating several types of device controllers within the same controller unit. For instance, the same controller could be configured to drive any combination of Winchester drives, tape units and floppy disks. Below this architecture is shown.

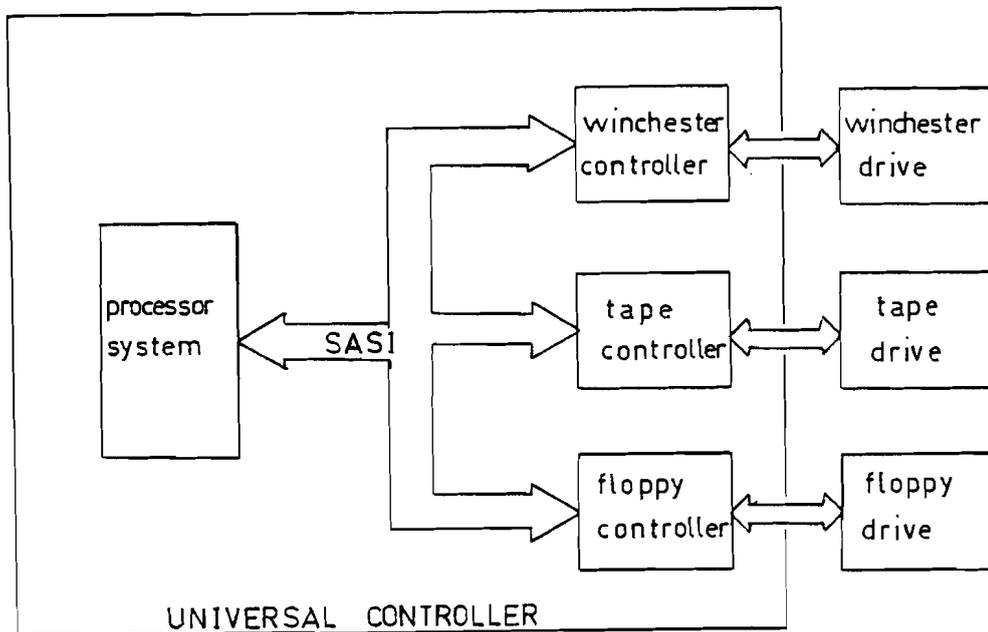


Figure 4.13: Bus architecture.

Both flexibility and expandability are high using this approach. However, according to Murphy's law, there has to be some disadvantage as well.

Firstly, the use of a bus implies adaption of signal lines on two occasions, from processor system to bus and from bus to device controller. To put it another way, the incompatibility between processor system signal lines, device controller signal lines and bus signal lines requires extra hardware provisions as is illustrated by figure 4.14.

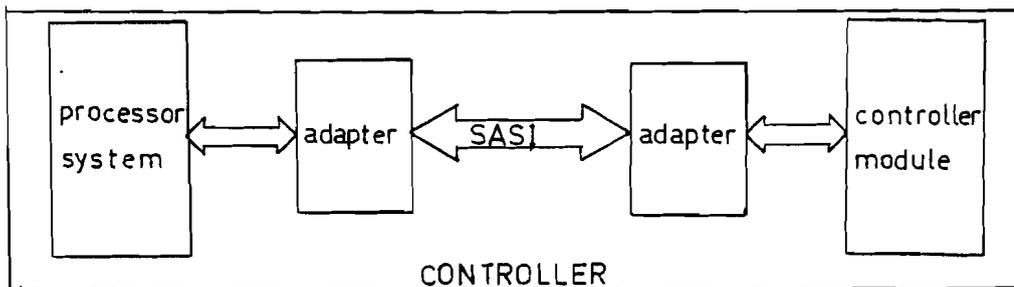


Figure 4.14: Bus adaptations.

Secondly, the SASI bus poses speed restraints on data transmissions thus obstructing the IOP from working at maximum speed.

Special Purpose Interface.

Using a specially designed interface adapter circuitry to connect the MSC 9000 to the controller's processor system yields an optimal solution as far as data transfer speed is concerned. A circuit diagram of this hardware interface can be found in appendix A.

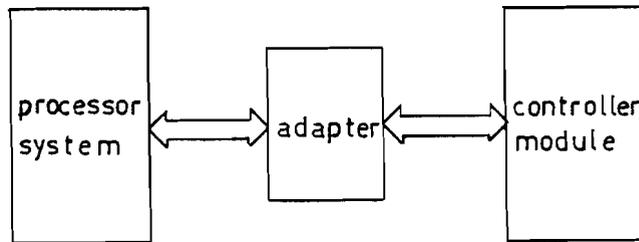


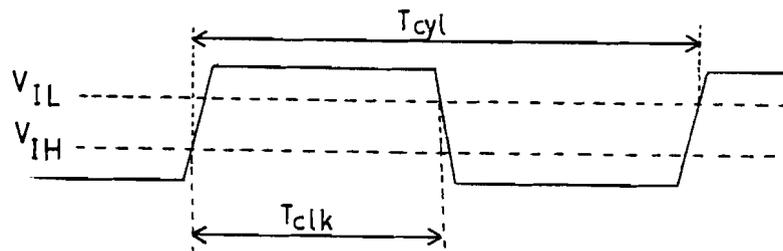
Figure 4.15. Special Purpose Interface.

In order to explain its operation, it will be described in four sections:

- Clock.
- Data latches.
- Communication protocol.
- Reset circuitry.

Clock.

The clock signal required by the MSC 9000 is specified in accordance with the time diagram of figure 4.16.



$$T_{cyl} = 250 \text{ ns} \pm 1\%$$

$$T_{clk} = 110 \text{ ns (min)}$$

$$V_{IH} = V_{CC} - 0.8$$

$$V_{IL} = 0.8$$

Figure 4.16: MSC 9000 clock waveform.

**Levels:** To obtain the required voltage levels for the clock signal from a standard TTL output, the buffering circuitry shown in appendix A is needed.

**Timing:** The accuracy of the generated clock signal is determined to a large extent by the crystal used. When using a 74ls124 VCO chip for clock generation, a series resonant fundamental mode crystal with a series resistance less than 200 Ohms is recommended. A 20 pF capacitor is placed in series with the crystal. Refer to figure 4.17.

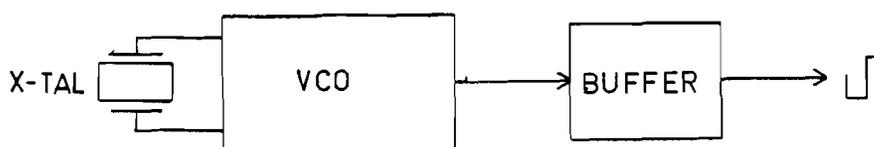


Figure 4.17: MSC 9000 clock circuitry.

#### Data latches.

The MSC 9000 has a tri-state, active high data bus which is used for communicating with both, the host processor and the disk drives. Externally driving this bus may only be done when the RDY and LDI signals from the module are both active. To obtain this, a data-input latch was added which is enabled only when the forementioned condition is satisfied.

Bus buffering in the opposite direction is necessary as well, hence the other data latch clocked when RDY and DOUT are both active. Both latches introduce propagation delay times.

#### Communication Protocol.

The lowest level of communication between processor system and controller module involves the transport of command and data bytes. For this purpose, several interface signals are present which are listed below.

**MODBSY** : MODULE BUSY - indicates whether the controller is ready to accept a command or is still executing a previous command.

**CMDSGN** : COMMAND SIGNAL - initiates a command transfer sequence.

**DREQ** : COMMAND REQUEST - request signal for next command byte.

**Q1** : DATA REQUEST ON CHANNEL ONE - request signal for next data byte.

ACK : ACKNOWLEDGE - acknowledge command or data byte transfer.

Both processor input lines MODBSY and CMDREQ are connected to the data bus through a tri-state buffer, selectable by the proper I/O address. This obliterates the need for an extra I/O port in the processor system.

A timing diagram, illustrating the protocol of the communication between processor system and controller module is given in figure 4.18. The flowcharts shown in appendix D, figure D8a and D8b can also be used as a reference.

The hardware in which this protocol is largely implemented is formed by a pulsed mode driven, asynchronous circuitry consisting of several D-flip-flops and random logic gates. Though this approach is quite common in interface design and not difficult, the result is a messy circuit, prone to errors caused by glitches. Alternatively, a synchronous circuit using programmable logic could be used. Appendix D discusses this possibility.

Reset circuitry.

In order to reset the controller module, a somewhat awkward reset signal is required on the controllers CLR input, as is shown below.

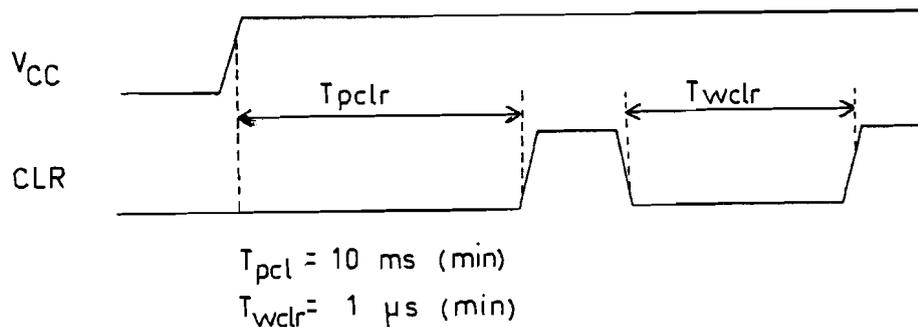


Figure 4.19: Reset timing.

In order to obtain a CLR input timing like the one shown, a so called reset delay circuitry is needed, consisting of two one shots.

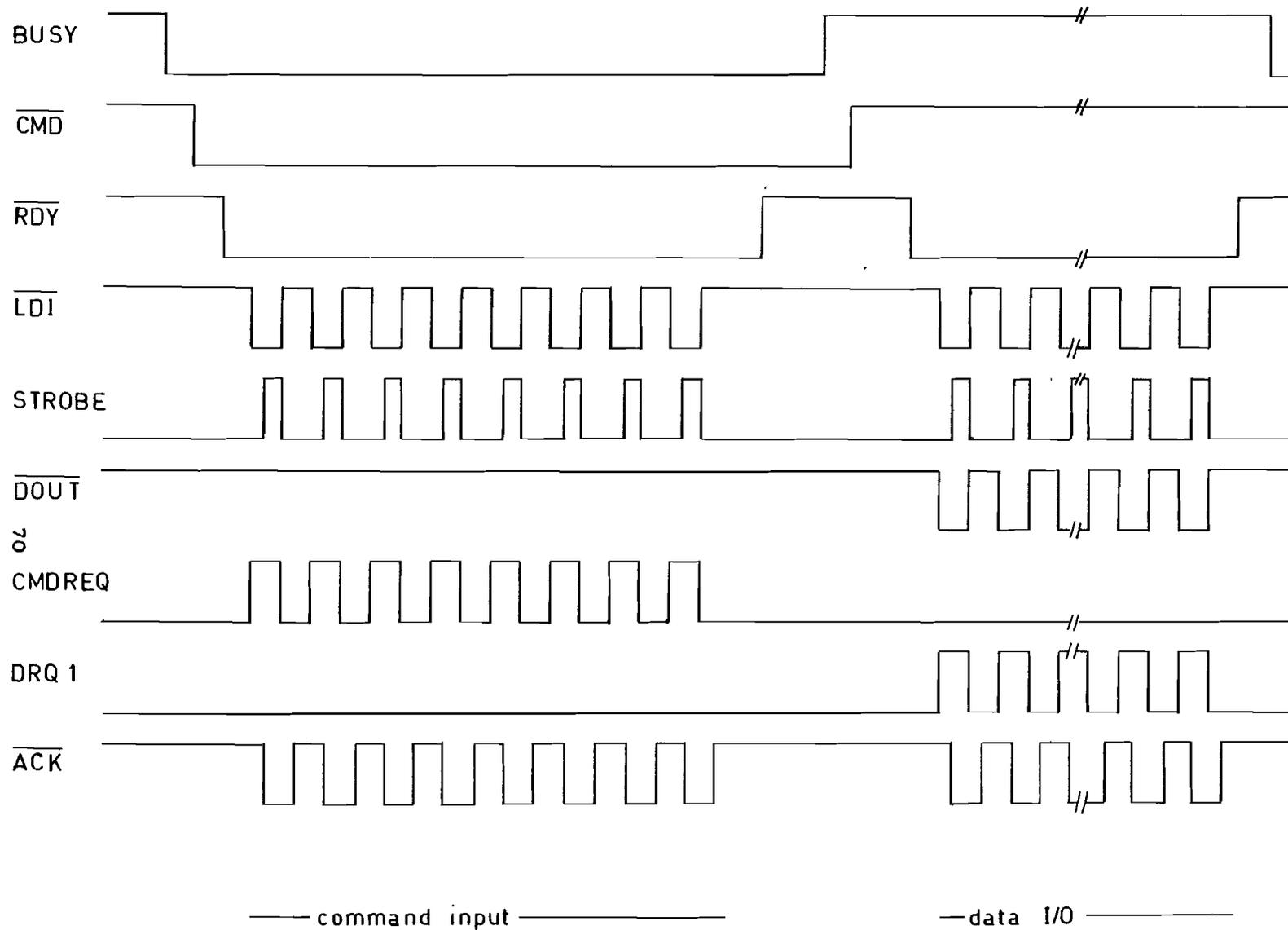


Figure 4.18: MSC 9000 INPUT/OUTPUT HANDSHAKE

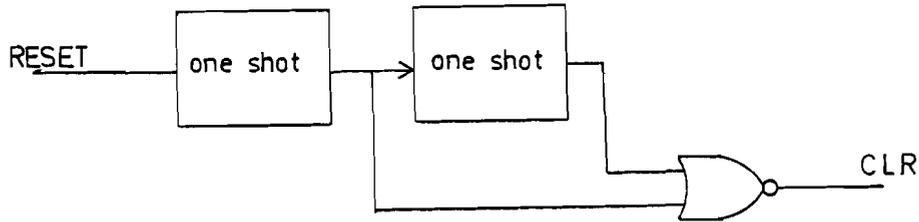


Figure 4.20: Reset delay circuit.

A positive going RESET signal triggers the first one-shot, causing a 10 msec long, low CLR to the module. Following this period, the second one shot is triggered, causing the CLR line to go low for another 1 usec.

#### 4.5.5. DMA transfer timing.

Considering the importance of the DMA data transfers between processor system and controller module, a more detailed discussion of the timing involved was made. The results of this survey, given below, are somewhat disappointing.

#### Data input timing.

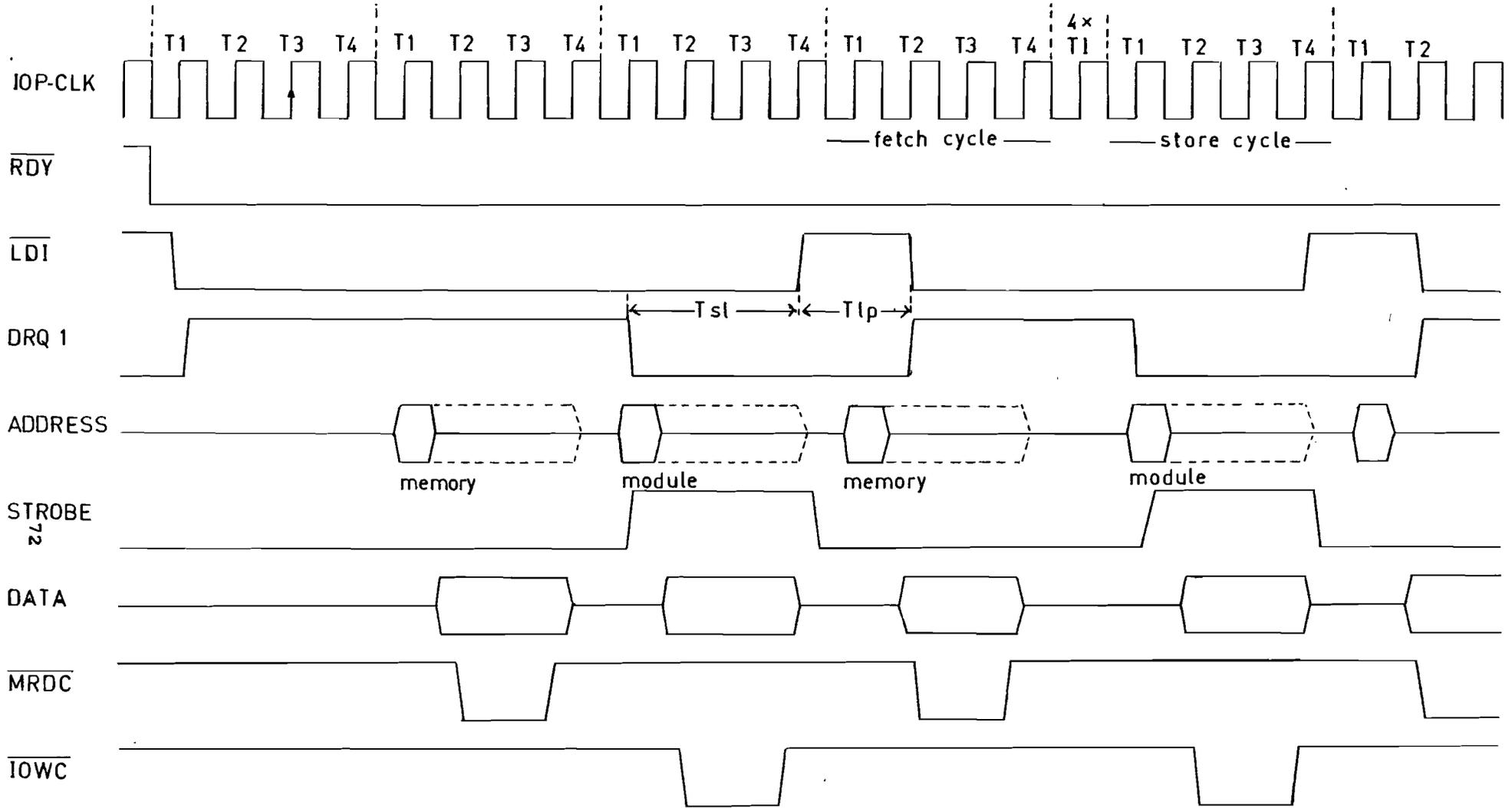
The MSC 9000 data bus, unfortunately is only 8 bit wide which leaves the choice 8/8 bit transfers or 8/16 bit transfers, the latter option by using the IOP's packing and unpacking facilities. Both possibilities are shown in the timing diagrams of figure 4.21 and 4.22 respectively.

#### WID 8/8.

WID 8/8 operation means data is fetched from memory on a byte by byte basis and sent to the controller module one byte at a time. The timing diagram of figure 4.21. shows this in detail. The situation shown here is a best case in which time delays, caused by the controller module are minimal, i.e.  $T_{s1}$  and  $T_{1p}$  are 580 nsec and 248 nsec respectively.

However, even with these minimal time delays, the data request (DRQ1) for the next transfer cycle appears after the T4 state of the current transfer cycle, yielding the insertion of 4 idle states, a delay of  $4 \times 200$  nsec, 0.8 usec per transfer cycle. Thus, best case data transfers require 2.4 usec/byte.

In a worst case situation, maximum values for  $T_{s1}$  and  $T_{1p}$ , 20 IOP clock cycles are needed for every transfer cycle, yielding 4 usec/byte.



	<u>min</u>	<u>max</u>	
Tsl	580	1235	ns
Tlp	248	748	ns

Figure 4.21: MSC 9000 DATA INPUT TIMING (DMA)

WID 16/8.

Using the IOP's capability to unpack a 16 bit word into two 8 bit bytes during a DMA transfer cycle, a situation as is shown in figure 4.22. arises. Again this is a best case situation. The unpacking operation requires four extra clock cycles between two synchronous store cycles to allow the peripheral to remove its first data request. Using this feature, the transfer of one word requires a minimum of 20 states, equivalent to 2 usec per byte. Worst case values are 2.8 usec/byte.

Remark.

In spite of the MSC 9000 controller module's inability to handle 16 bit words, word/byte transfers can be used successfully to obtain a higher data transfer rate as compared to byte/byte transfers. However, the theoretically achievable maximum transfer speed of 957 Kbyte/sec for data input to the MSC 9000 cannot be obtained. This disappointing result originates from the fact that communication between processor system and controller module is asynchronous.

	WID 8/8		WID 16/8		
	min	max	min	max	
SPEED	250	416	357	500	Kbyte/sec.

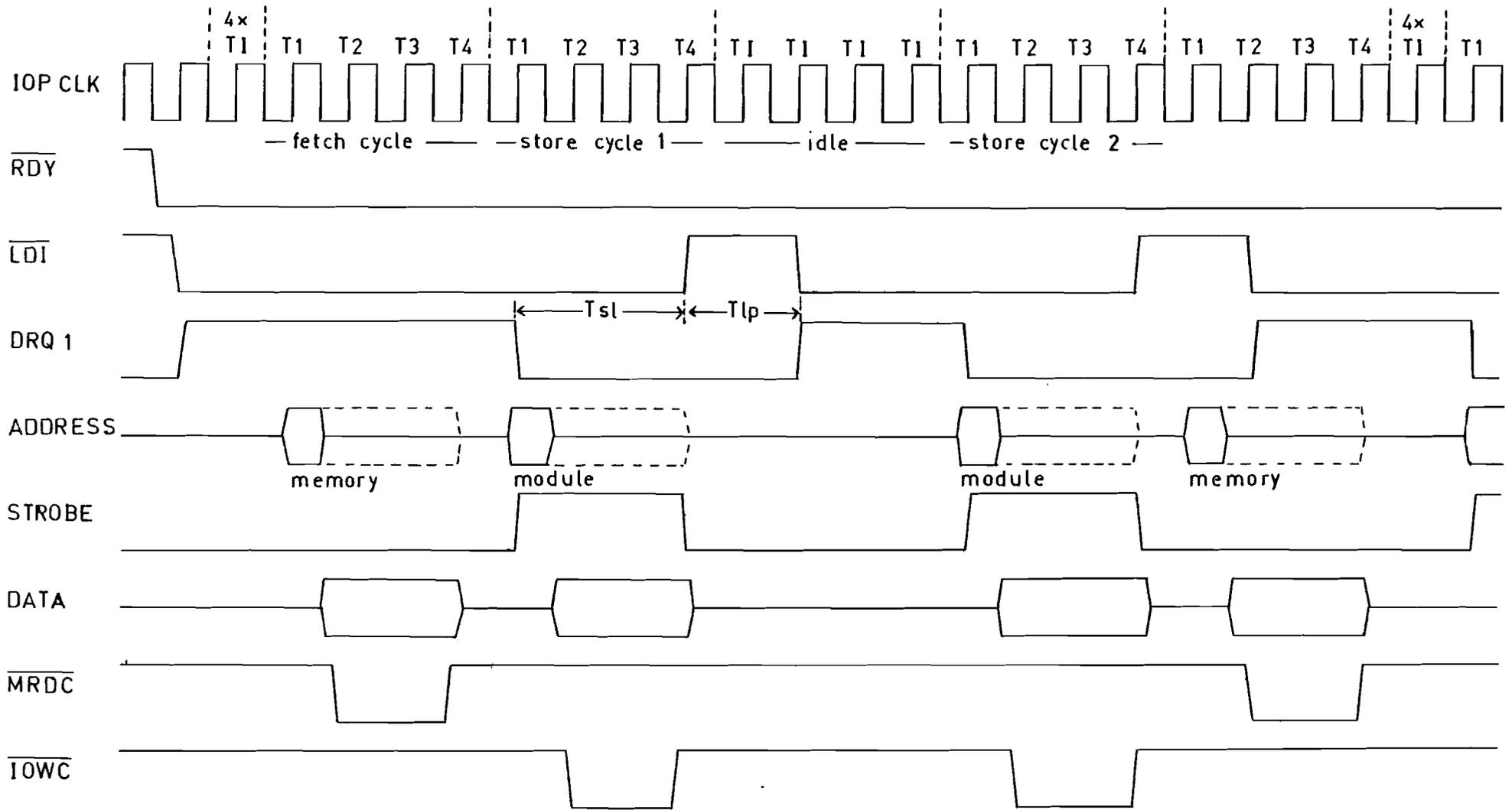
Data output timing.

In view of the results mentioned in the previous section concerning data input timing, only the 8/16 option was taken into consideration.

Two sources of time delays can be discerned in figure 4.23:

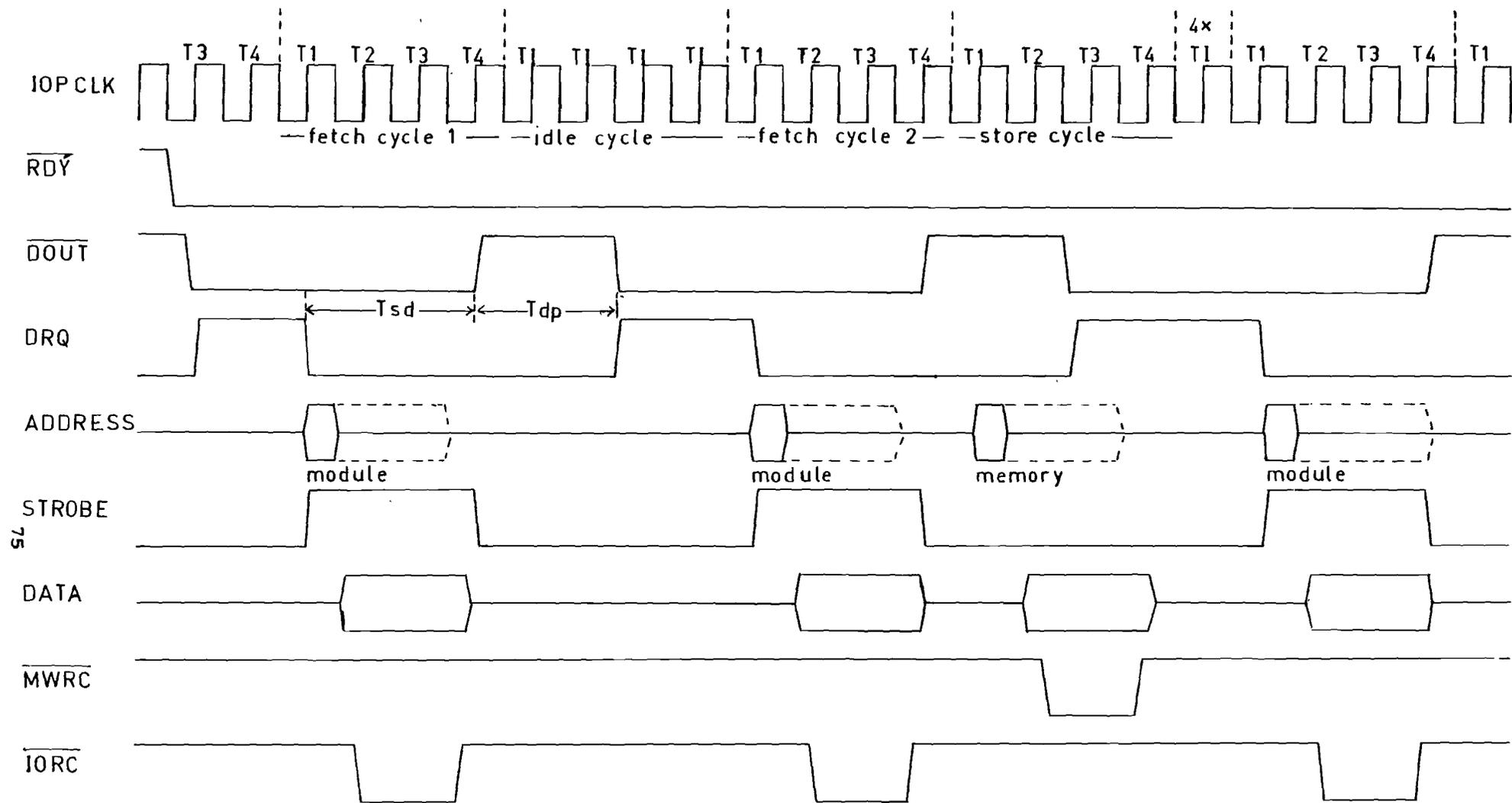
1. Four Idle states are inserted by the IOP between two synchronous fetch cycles. This delay is intended to allow the peripheral to remove its previous data request signal. In this case however, the DRQ is already gone in time, so Idle states are not necessary. Unfortunately this feature is not optional so these four Idle states will have to be taken for granted.
2. The data request following the store cycle of the previous word, arrives at best during state T3 of the store cycle. As a result of this, the IOP inserts four Idle states immediately following the current store cycle. To avoid these Idle states, DRQ from the module has to be active before state T4 of the second fetch cycle. The minimum set-up times for Tsd and Tdp render this impossible.

As a result of the facts mentioned above, a transfer cycle for one word requires at least 20 states, yielding a maximum trans-



	<u>min</u>	<u>max</u>	
Tsl	580	1235	ns
Tlp	248	748	ns

Figure 4.22: MSC 9000 DATA INPUT TIMING (DMA)



	<u>min</u>	<u>max</u>	
Tsd	560	1175	ns
Tdp	485	915	ns

Figure 4.23: MSC 9000 DATA OUTPUT TIMING (DMA)

fer speed of 500 Kbytes/sec. Assuming maximum values for Tsd and Tdp, a transfer cycle will require 28 states. (357 KB/s)

#### 4.5.6. DMA termination.

Any DMA transfer running on the IOP's I/O channels, can be terminated on one of the following conditions:

- Terminate on single transfer.
- Terminate on byte count.
- Terminate on masked compare.
- Terminate on external signal.

Both channels will wait for a Data Request signal (DRQ) after a non-terminating transfer cycle. If this request never occurs, the DMA channel involved will wait indefinitely. Though such a hang-up on one channel does not affect the other channels operation - unless chained channel operation was selected - it will prevent the IOP from terminating the Task Block Program on the held up channel.

To prevent this, the external terminate option has to be selected on any DMA transfer operation since it is the only way to exit a hang-up situation.

Using a Programmable Interval Timer (PIT) as a one shot, re-triggered after every DMA transfer cycle, will cause an external interrupt to the I/O channel after a programmable delay. Should a DRQ signal occur beyond a reasonable period of time, the external interrupt will cause the TBP to be resumed. A jump condition in the TBP can determine the occurrence of an EXT interrupt and transfer control to the appropriate error handling section. Refer to appendix A for the PIT connections.

#### 4.6. Drive interface.

The interface circuit between drives and controller module consists of the following sections:

- Data separator : The purpose of a data separator was explained in chapter 2. As mentioned earlier, a MSC 9000 module was used for this purpose.
- Precompensation: The precompensation circuitry is built around three D-FF's clocked at a 40 Mhz clock frequency. Select information for the multiplexer is derived from the MSC 9100 module. Refer to figure 4.24.

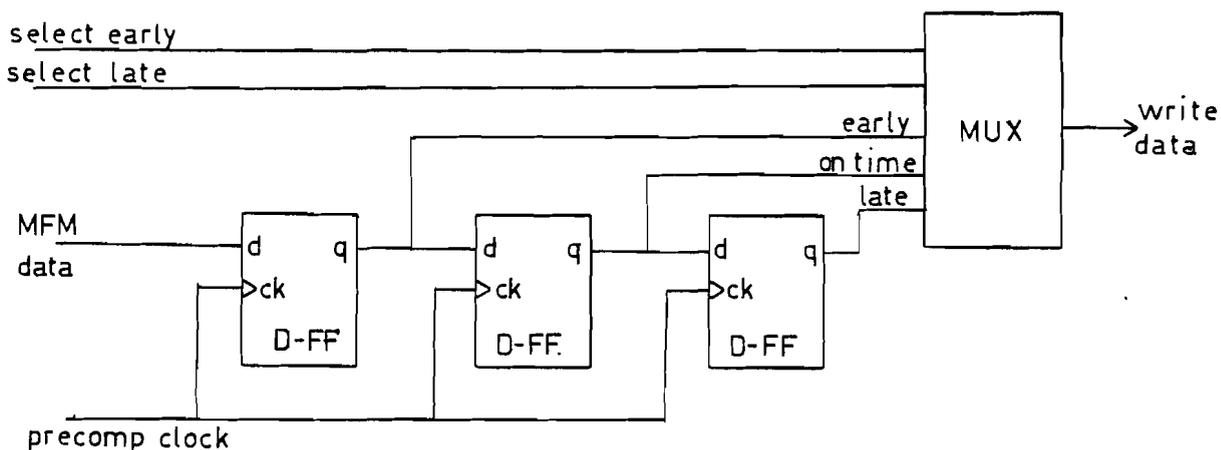


Figure 4.24: Precompensation.

- Decoder : A decoder/multiplexer is used to decode the control information from the MSC 9000 for proper driver and latch control. Refer to appendix B for the required decoding scheme.
- Latches : Drive control signals are latched from the module's data bus and transferred to the bus driver circuits. Select information for these latches is derived from the MSC 9000 through the decoder/multiplexer.

A block diagram of the controller disk drives interface circuitry is shown in figure 4.25.

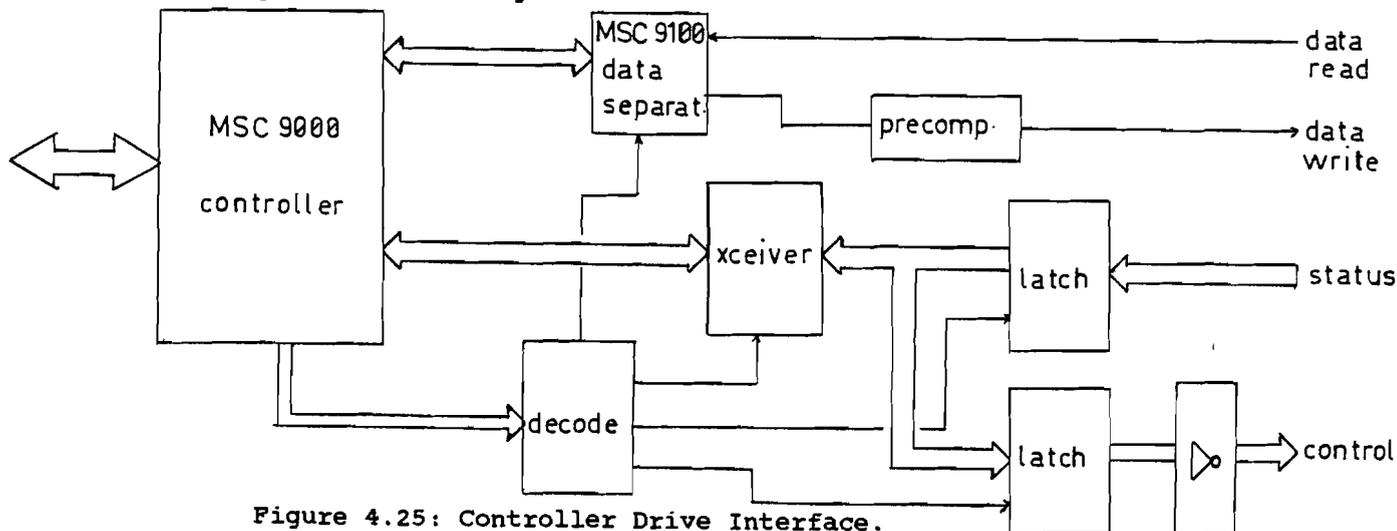


Figure 4.25: Controller Drive Interface.

#### 4.7. Interface standards.

The physical interface to the drives, created by the hardware concept described in the previous paragraph is known as the Floppy Extension Interface. Though not a universal standard, it is widely spread and found on most low cost 5.25 and 8 inch Winchester drives. In fact, until now no universal interface standard exists and due to several de facto standards penetrating the market, the question whether or not such a standard will ever exist is justified.

Appendix B contains a section concerning some of the interfaces currently in use or being suggested by the ANSI.

## Chapter 5. SOFTWARE DESCRIPTION.

### 5.1. Introduction.

The software required to operate an intelligent controller like the one covered in this report is quite extensive. When designing software of this scope, it is generally considered good policy to divide it into functional sections and develop a modular structured software package, rather than a large bulk of code. An attempt to make such a functional division was made in the block diagram of figure 5.1.

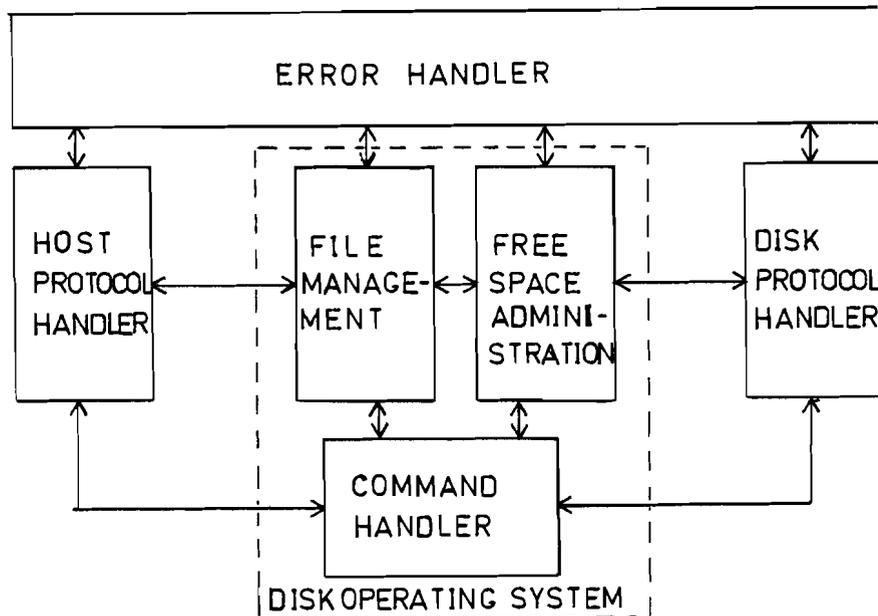


Figure 5.1: Software block diagram.

The following sections are discerned.

**Host Protocol Handler:** Handles all communication between Host computer system and controller.

**Disk Protocol Handler:** Handles all communication between Disk controller module and the controller's processor system.

**Disk Operating system:** Handles file management and manipulation.

**Error Handler:** The error handler is responsible for dealing with unforeseen occurrences within the control-

ler software. After entering this software section, a known and defined state has to be reached from which point the controller can resume proper operation.

Prior to describing each of these software modules, the controller's initialization phase will be discussed.

## 5.2. Initialization.

After a power-up of the controller, it will receive an automatic reset signal, bringing both the controller's processors in a defined state and resetting the controller module as well. As the 8086 CPU is strapped as a master processor, it will start executing from memory location FFFFO H where it should encounter an intersegment direct jump which target is the starting location of the actual initialization routine. At this stage all interrupts are disabled.

The tasks performed by the initialization routine are listed below in sequential order.

1. CPU initialization.
2. IOP initialization.
3. Peripheral initialization.
4. RAM test.
5. Interrupt vector field initialization.
6. Controller module initialization.
7. File management administration set up.
8. Interrupt enabling.

After execution of this sequence, the controller is ready to receive and execute its first command.

### 5.2.1. CPU initialization.

Initializing the CPU means its segment registers have to be given a predetermined value. After a reset, the code segment register has the value FFFF H, the instruction pointer 0000 H. Thus the first instruction is fetched from location FFFF0 H as stated before. As a result of the intersegment direct jump instruction located there, the code segment register as well as the instruction pointer are given a new value, transferring control to the initialization program located in ROM.

The stack segment register is set to the top of stack, the stacksize being restricted to 756 words, due to the expected low degree of interrupt nesting.

The Data segment is used by the DOS to store its tables and pointers. The size chosen (7 Kword) may prove to be insufficient and can be expanded easily.

Finally, the Extra segment register is initialized, pointing to the area of the controllers workspace where file data is stored. The memory map created thus can be found in figure 5.2.

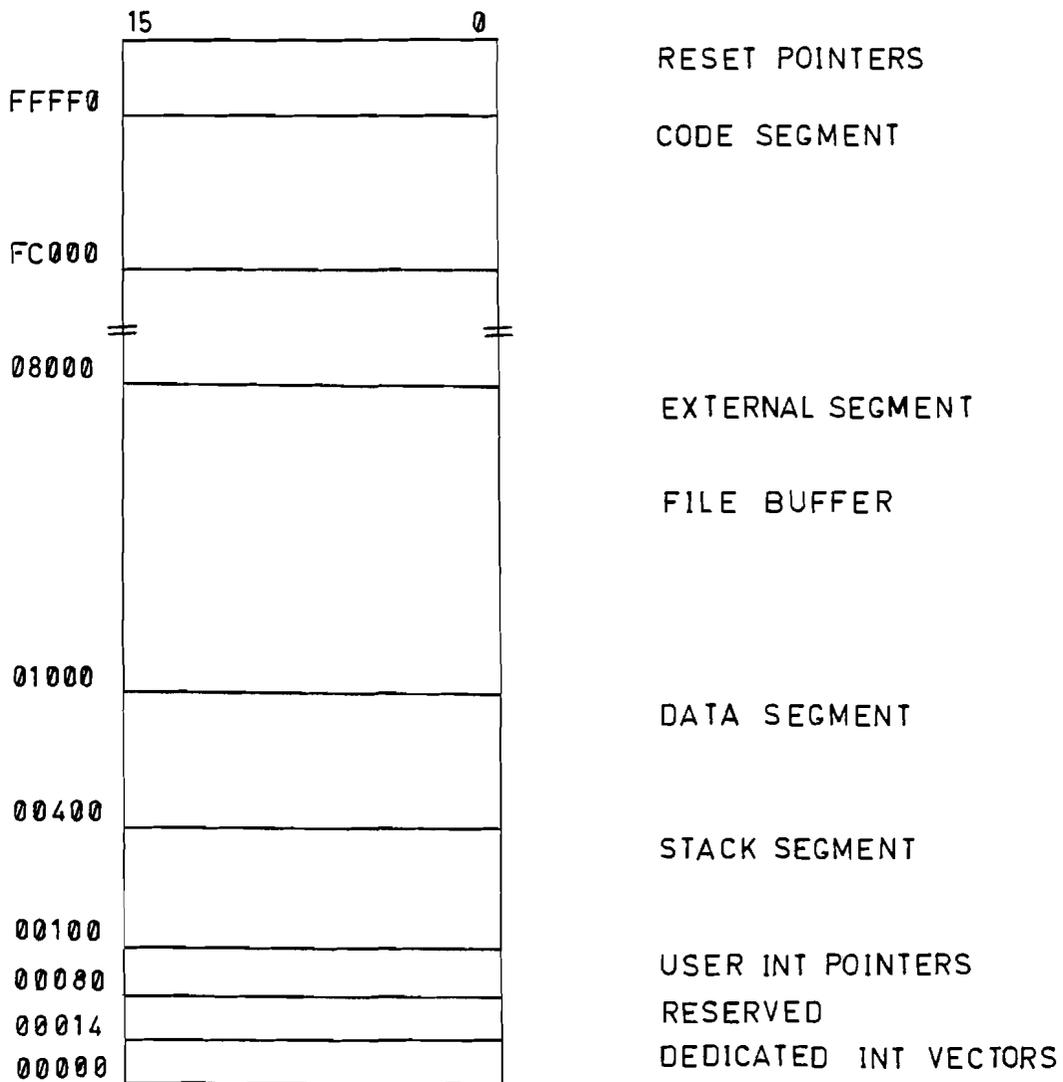


Figure 5.2: Memory map.

### 5.2.2. IOP initialization.

The next phase in the initialization process, is to configure the IOP for its task. The IOP enters a HALT state after receiving a reset signal. To start its initialization sequence, it has to receive a channel attention (CA) from the CPU. During this CA, the select line is used to indicate whether the IOP is to function as master or as slave processor. In this case, the IOP will be a slave processor (SEL= high). Subsequently the IOP will request and obtain control over the system bus which it will assume to be eight bits wide. The SYSBUS field located at memory location FFFF6 will inform the IOP about the actual bus

width, 16 bits in this case. The SYSBUS field is part of the system configuration pointer block, located from FFFF6 through FFFFB H in ROM. This block also contains a pointer to the system configuration block itself where the IOP can find the system operation command, containing information about the request/grant mode and the physical I/O bus width. Furthermore the IOP stores the value of the channel Control Block (CCB) in an internal register. As a result of this, the CCB cannot be relocated unless another system reset is given.

This completes initialization of the IOP. To signal the host, the Channel 1 Busy Flag, set to FF H by the CPU prior to initializing the IOP, is cleared and control of the system bus is redirected to the CPU.

The CCB contains two identical sections, one for each channel. In each section, a busy flag, indicating whether the corresponding channel is active or not, a Channel Command Word (CCW), indicating the channels mode of operation and a Parameter Block pointer (PB) are present. The PB contains the task block pointer plus any other parameters the Task Block Program (TBP) requires for proper operation.

At this stage the CCB is not initialized yet since it can be done by the CPU at any time before starting an I/O program.

Figure 5.3 shows the IOP initialization sequence, figure 5.4. the various control blocks involved and their locations.

### 5.2.3. Peripheral initialization.

In order to configure the controller for its task, the peripheral chips present in the system need be initialized properly. This entails programming each peripheral to function in the desired mode.

#### Programmable Peripheral Interface.:

The PPI is programmed to operate in mode 2 on channel A and C. Thus strobed I/O is obtained through these channels. Port B can be configured for mode 0 operation, either input or output. Programming is done by writing the proper control word in the peripheral's control register.

#### Programmable Interrupt Controller.:

As the PIC was designed to accommodate both 8080/8085 and 8086 /8088 processor systems, careful attention should be given to proper initialization of this peripheral since both systems require different interrupt handling procedures. Initialization is performed in two phases;

- Initialization Command Words. A sequence of 2,3 or 4 control words is stored in the PIC. In this particular application, three ICW's are needed to configure the PIC as follows:

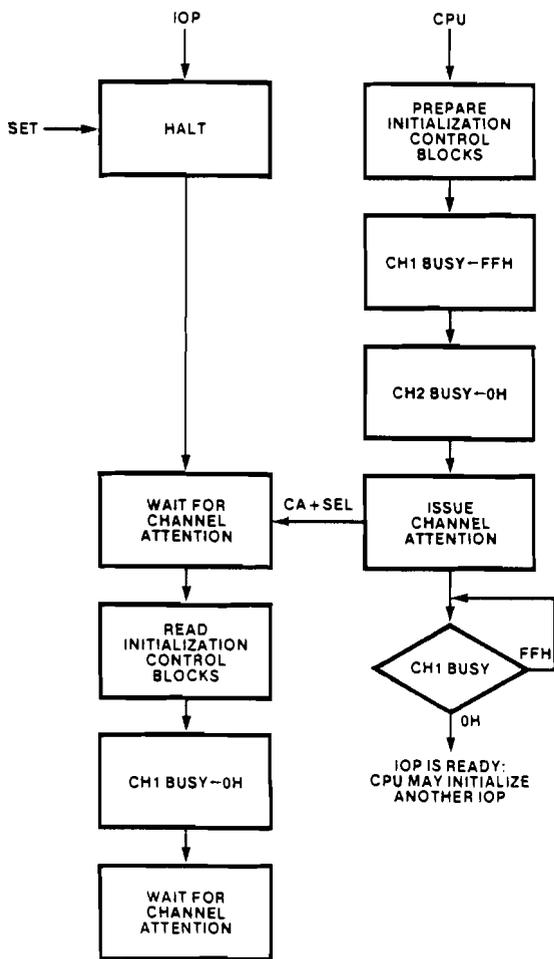


Figure 5.3: Initialization Sequence

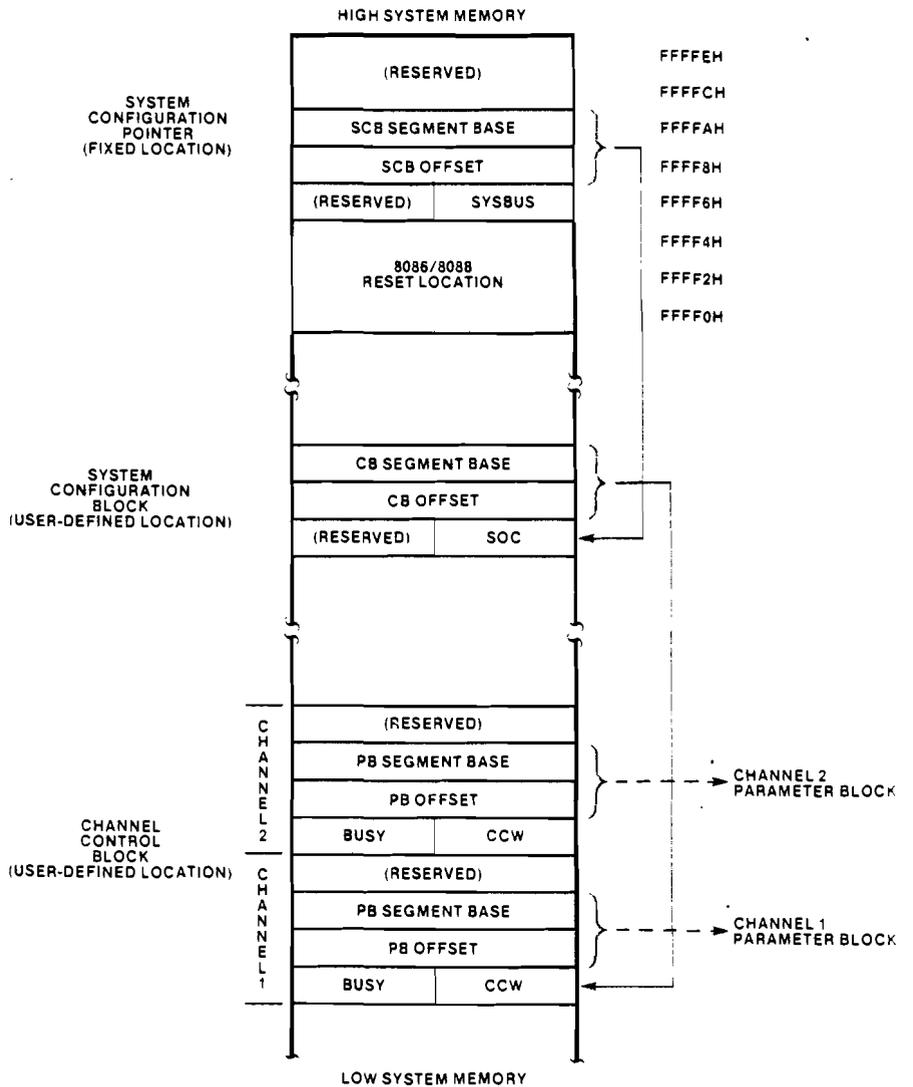


Figure 5.4: Initialization Control Blocks

- Single controller.
- Edge triggered interrupt.
- Interrupt vector field starts at location 080 Hex.
- No slave interrupt controller.
- 8086/8088 mode operation.
- No special end of interrupt.
- Non buffered mode.
- Not special fully nested mode.

After loading the ICW's, the PIC is ready for operation. Further additional options can be programmed subsequently by writing Operation Command Words (OCW)

- Operation Command Words. Three OCW's are required to select the desired mode of operation:

OCW1: The first OCW is used to set an interrupt mask on the eight interrupt lines. Only those lines that are not masked off will cause an interrupt request to the CPU.

OCW2: The second OCW specifies whether a rotating priority scheme is used which is not the case here.

OCW3: The third OCW enables the user to apply the PIC in a polled interrupt environment, rather than a vectored interrupt one. This feature is not used.

This terminates the initialization of the PIC. It is now capable of detecting interrupts on eight levels, each having a different priority. IRO, having the highest priority, is used to service host requests, providing the IOP is not executing a DMA transfer. In the latter case, the interrupt will remain pending until the CPU regains control of the system bus.

#### Programmable Interval Timer.:

After a system reset, all counter modes are undefined. Initialization is done by writing the appropriate sequence of control words to the PIT. In order to configure the PIT for retriggerable one-shot operation as indicated in paragraph 4.5.6., counter 0 and counter 1 are to be programmed in mode 1. during DMA transfers. However, as long as no DMA transfer is in progress, both counters have to be inhibited. For this purpose, both counters are programmed to mode 0. This will force the outputs to remain low as long as the gate inputs, connected to the data request outputs of the IOP remain low. Thus no EXT interrupt is generated. It is the responsibility of the Task Block Program that initiates the DMA transfer to reprogram its corresponding counter to mode 1 and loading it with a suitable count value. After proper termination of the DMA transfer, the corresponding timer has to be disabled again. Timer 3 remains unused.

#### 5.2.4. RAM test.

A RAM test is performed to verify correct operation of the controller's workspace and to determine its size. Testing is done by an algorithm like the one shown below.

```
BEGIN PROCEDURE RAMTEST

VAR  ADDR,PATTERN,TEST : INTEGER.
      END                : BOOLEAN.

      ADDR:=000 H; PATTERN:=5555 H;TEST:=AAAA H;END:=FALSE

      WHILE NOT END DO
          MEM(ADDR):=PATTERN;
          IF MEM(ADDR) NEQ PATTERN THEN END:=TRUE
              ELSE BEGIN
                  SHIFLEFT MEM(ADDR)
                  IF MEM(ADDR) NEQ TEST
                      THEN END:=TRUE
                      ELSE ADDR:=ADDR+1
                  FI
              END
          FI
      OD
END (RAMTEST)
```

#### 5.2.5. Interrupt Vector field initialization.

The interrupt vector field is located in RAM from location 0 H to 0FE H according to Intel convention. The contents of this field can be loaded from the controller's ROM by a REP MOVSW instruction which moves an entire data block from memory to memory. The value of these interrupt service routine pointers can be altered during program execution.

#### 5.2.6. Controller module initialization.

After reception of the hardware reset signal, the controller will signal the processor system by asserting its ready line. Issuing a DIAGNOSTIC command to the controller module will cause it to perform an internal check. The result of this check routine can be obtained by means of a STATUS command. Refer to paragraph 5.4.

Subsequently, the CPU can configure a Task Block Program for the IOP to read track zero of drive 0. The information thus obtained should at least contain parameters such as:

- No of heads.
- No of drives.
- No of tracks per drive.
- Physical block size.
- No of blocks per track.
- Index list pointer.
- Root directory pointer.
- Free space table pointer.

This information is stored in the controller's workspace for future reference.

#### 5.2.7. File management administration set-up.

Initialization of the Disk Operating System means resetting all relevant pointers such as those that point to the current open file table which is empty at this stage, the active I-node table which is loaded with the Index node of the Root directory. Also pointers into the controller's data buffer are cleared.

The Root directory is fetched from the disk since the user will have to refer to it initially. Figure 5.5. shows the situation as it occurs after these steps.

### 5.2.8. Interrupt enabling.

At this stage in the initialization process, the controller is ready to receive and execute commands from the Host. As these commands are given on an interrupt basis, the interrupt enable flag of the processor has to be set as the final step.

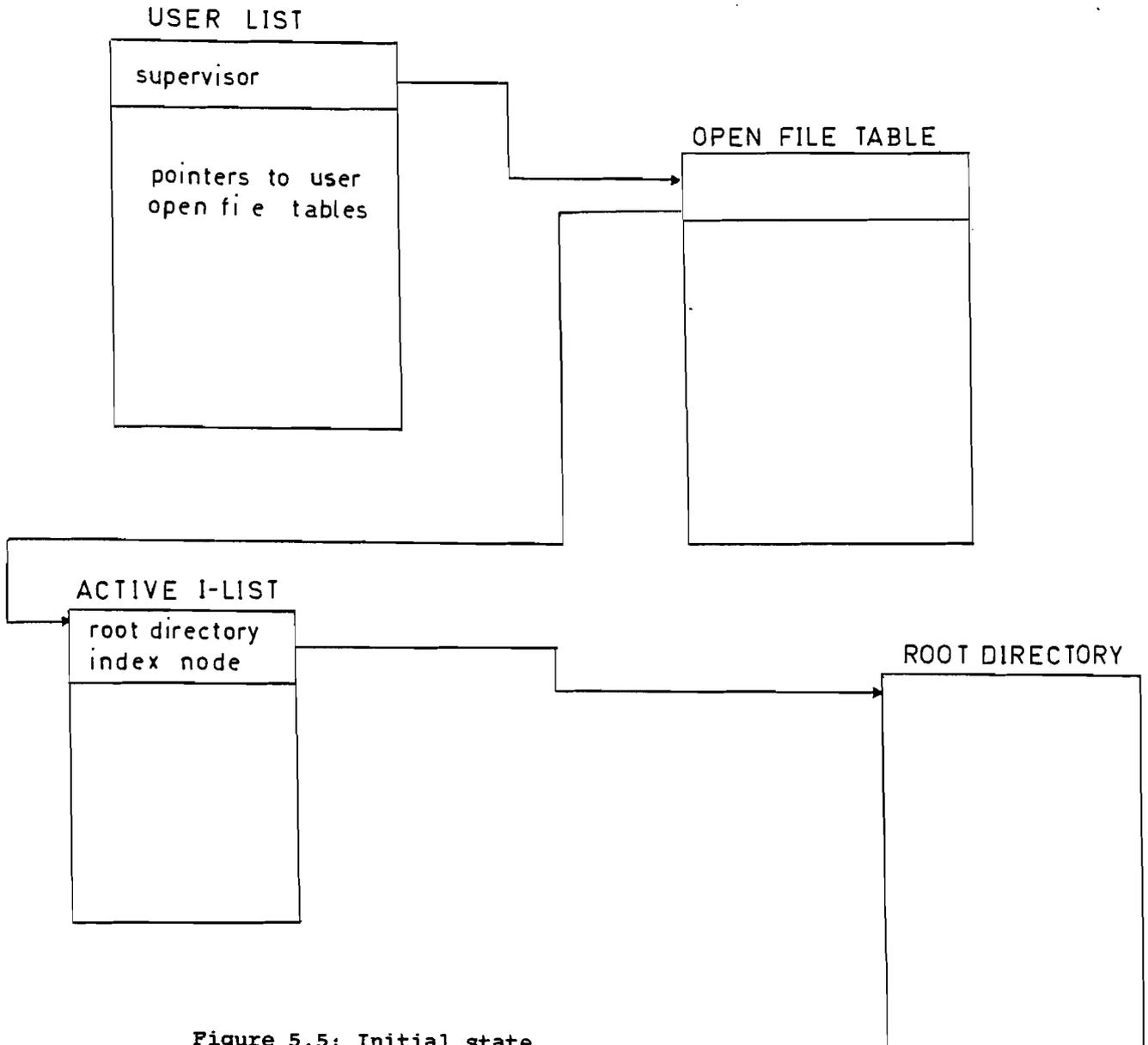


Figure 5.5: Initial state.

### 5.3. Host Protocol Handler.

As stated before, the Host Protocol Handler performs all communication between Host Computer and Controller. This communication is bi-directional and involves the exchange of the following information:

- Commands from Host to controller.
- Status from controller to Host.
- File data between controller and Host.

The protocol by which this communication takes place will be described shortly but prior to that, a brief discussion concerning synchronous and a-synchronous I/O will have to decide for either of these forms of communication, a choice that has considerable impact on the protocol itself.

#### Synchronous I/O.

Synchronous I/O means every request from the Host is serviced immediately by the controller. During this service request, the Host waits for the result of his request before continuing processing. Figure 5.7. illustrates this.

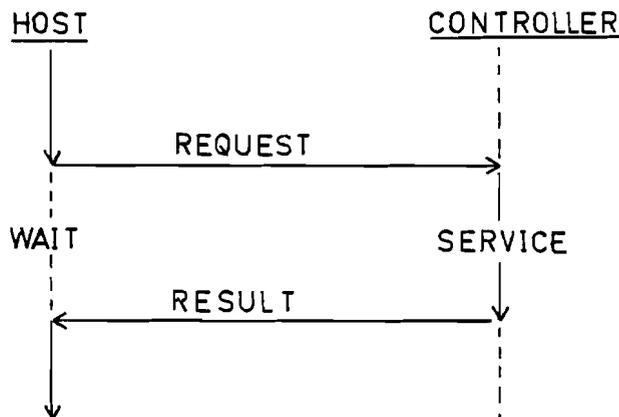


Figure 5.7: Synchronous I/O.

#### A-synchronous I/O.

In the case of asynchronous I/O, service requests from the Host are queued by the controller. After issuing a service request to the controller, the Host continues processing. Upon termination of a requested service, the controller interrupts the Host to transmit the result. Furthermore the controller has to inform the Host of the request to which this result belongs since Host requests are not necessarily serviced in incoming order. Using A-synchronous I/O, multiple service requests from the

Host can be buffered and serviced in an order best suited for the controller. Refer to figure 5.8.

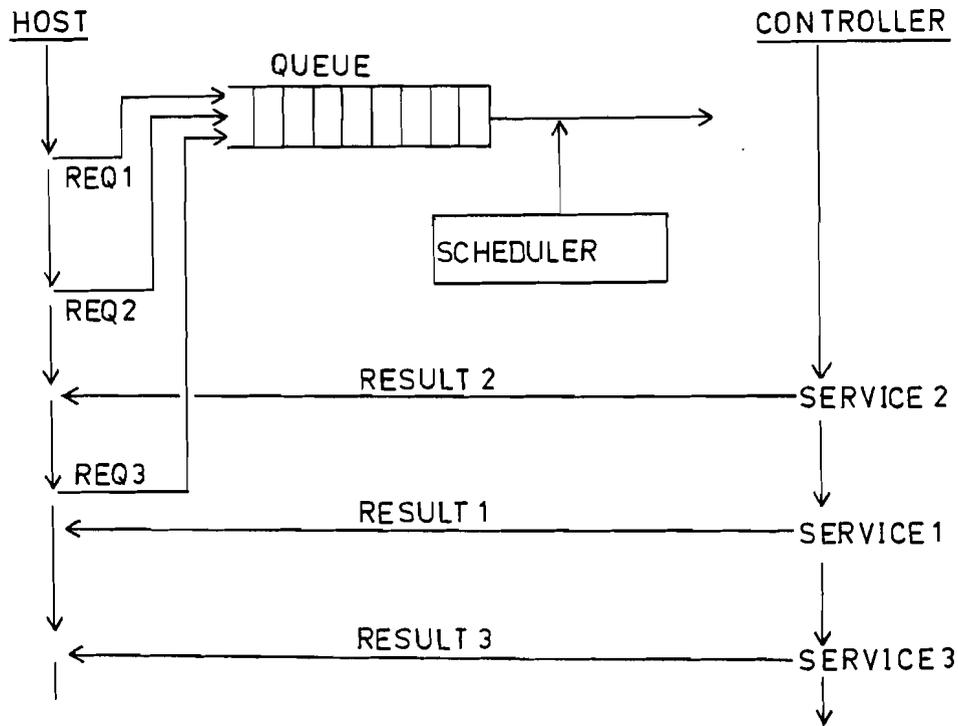


Figure 5.8: A-synchronous I/O.

The major advantage of synchronous I/O is its relative simplicity, no request queueing, no scheduling, no interrupts to the Host. On the other hand, the host is inactive during the time his request is being serviced by the controller, leading to a decreased performance of the Host.

A-synchronous I/O entails scheduling of requests by the controller's Host Protocol Handler. Furthermore, the Host might need the results of his request before being able to continue its current process. Thus the apparent advantage of parallel processing could be deceiving. On top of that, the Host has to keep track of outstanding requests and the order in which he receives the results from the controller.

Based on these considerations, synchronous I/O was chosen whereby the Host interrupts the controllers current activity by issuing a service request and subsequently waits for the result. Figure 5.9. illustrates the controller's control flow.

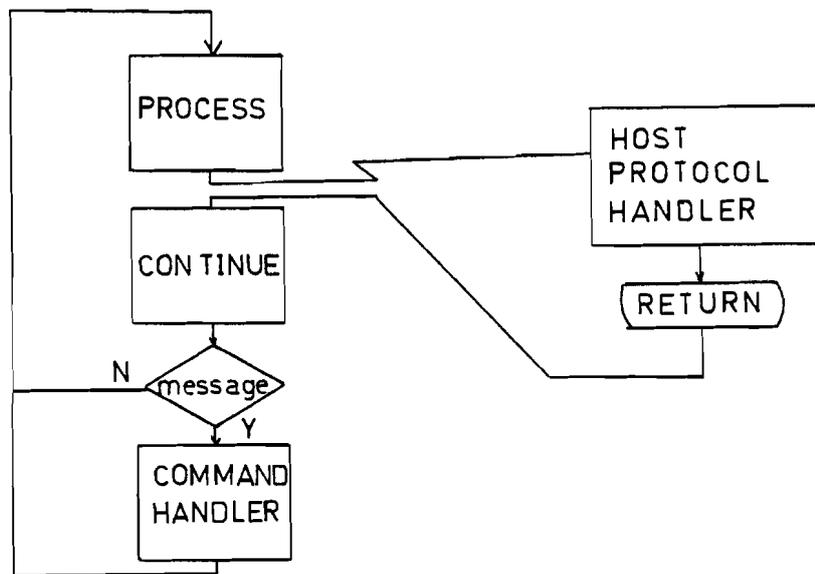


Figure 5.9: Control flow.

#### 5.3.1. Layer model.

The problems involved with communication protocols between computer systems, which is what we are dealing with here, are extensive. No attempt is made in this report to present a universal solution to communication protocol problems. The approach chosen here is somewhat related to the ISO Open Systems Interconnection reference model although some modifications were made. One of these modifications is the reduction to four rather than seven layers as can be seen in figure 5.10.

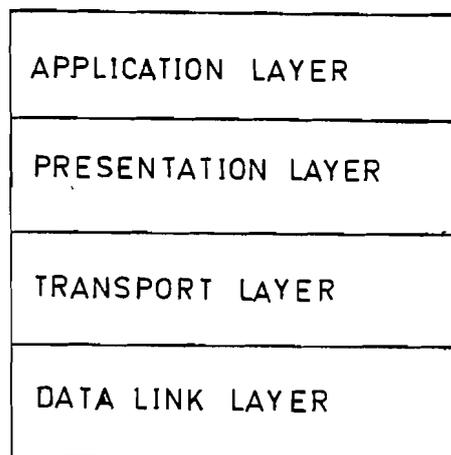


Figure 5.10: Protocol layer model.

### 5.3.2. Application Layer.

The application layer acts as an intermediary between the controller's Disk Operating System and the Host Protocol Handler. This interaction is bi-directional. The DOS calls the application layer as a subroutine whenever it wants to transfer status information or file data to the Host. The application layer in return, signals the reception of a command request or a data file from the Host to the DOS. This signalling is done by means of a flag rather than on interrupt basis.

The information obtained by the application layer, originates from the layer underneath. Communication between the diverse layers in the model is performed strictly through system memory.

Figure 5.11. illustrates the interaction between DOS and Application layer.

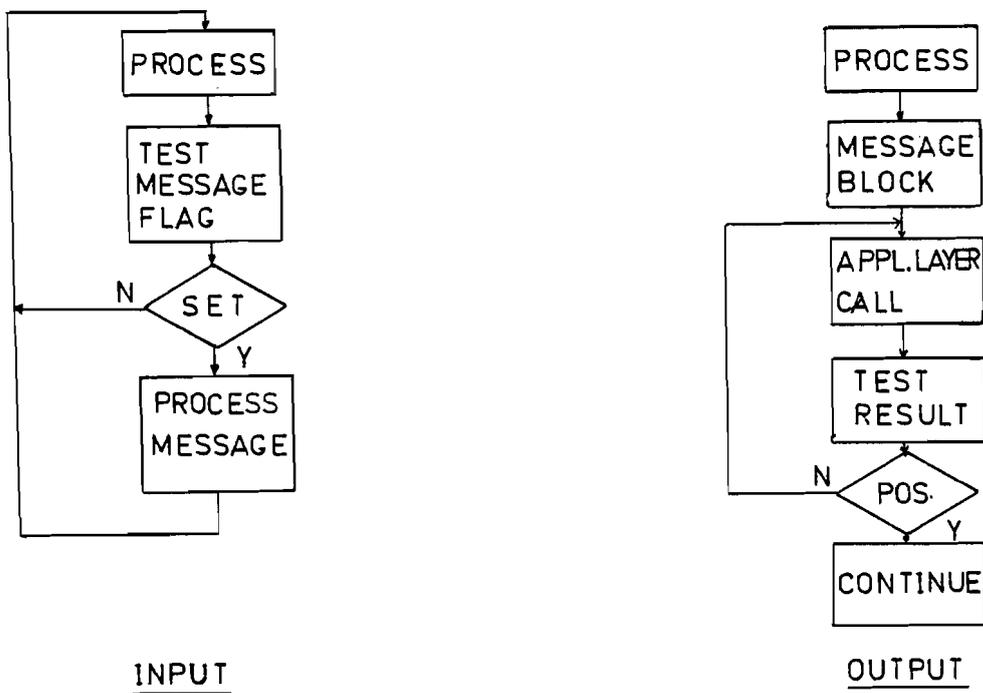


Figure 5.11: DOS-Application Layer interaction.

### 5.3.3. Presentation Layer.

The task of the presentation layer is to arrange the incoming and outgoing data into a fixed format to avoid mis-interpretation. To this extent, three different types of so called I/O blocks are distinguished:

- Command Block.
- Status Block.
- Data Block.

#### Command Block.

COMMAND CODE
USER ID
FILE NAME
RECORD NUMBER
DATA FIELD LENGTH
CONTROL FIELD

COMMAND CODE: Block identification.  
USER ID: User identification.  
FILE NAME: String of characters terminated by an ETX symbol specifying the file.  
RECORD NUMBER: Record to be accessed within file.  
DATA FIELD LENGTH: Length of data involved.  
CONTROL FIELD: Error control information.

The length of a command block is variable and depends largely on the file reference information, i.e. filename or path name. The other fields within the command block are of a fixed length.

#### Status Block.

STATUS CODE
USER ID
STATUS FIELD
CONTROL FIELD

STATUS CODE: I/O block identification.  
USER ID: User identification.  
STATUS FIELD: String of characters terminated by an ETX symbol containing the actual status information.  
CONTROL FIELD: Error control information.

The status block contains information concerning the controller, in particular any errors that occurred as a result of the execution of a user command. The length of this I/O block is determined by the information contained in the status field.

### Data Block.

The format of a data block is similar to that of a status block in that it has the same fields. In a data block however, file data is contained in the last field.

DATA CODE
USER ID
DATA FIELD
CONTROL FIELD

DATA CODE: I/O block identification.  
USER ID: User identification.  
DATA FIELD: String of characters.  
CONTROL FIELD: Error control information.

A Data block always follows after either a command or a status block. For this purpose, the command and status code fields contain a flag indicating whether the block is followed by a data block or not.

#### 5.3.4. Transport Layer.

The transport layer serves a dual purpose:

1. Receiving command and data blocks.
2. Sending status and data blocks.

The condition imposed on this transport of I/O blocks, is that it is done error-free. To this extent, error checking and retransmission is performed. In case of failure, an error status signal is generated, signalling the presentation layer.

##### 1. Receiving.

The Host can interrupt the controllers current activity by writing to the peripheral interface port. Providing the IOP is not active, i.e. the CPU is running, it will receive an interrupt from the Command/Status I/O port. As a result of this, an interrupt service routine is called which initializes the IOP to perform the required task block program. This TBP proceeds with the following steps:

- + Acknowledge the interrupt and disable further interrupts.
- + Read and interpret the contents of the PPI.
- + Transfer control to the required software routine.

What routine this should be depends on the value of the first byte sent by the Host. Three possibilities exist:

1. Command code.
2. Status code.
3. Data code.

#### Command routine.

After receiving a command code, the IOP polls the PPI to obtain two further bytes, indicating the length of the I/O block the Host will sent over the Data channel. Based on this information, an area in the controllers workspace is allocated as I/O buffer. Subsequently a DMA channel is prepared and a DMA transfer from host to controller is initiated.

The actual DMA transfer is performed by the physical layer described in the next paragraph.

After termination of the DMA transfer, a checksum is calculated and compared with the information contained in the I/O block's control field. In case of a mismatch, a retransmission is requested.

The Host, after termination of the DMA transfer, waits for a message from the controller through the command/status channel to verify whether the I/O block was received correctly. If not, the entire block is sent again. This process repeats itself until a retry counter at the host decrements to zero after which the attempt is aborted.

#### Data routine.

As the result of a succesful I/O block transfer, the IOP task block program checks the flag in the I/O block's command code field to see whether a Data block follows. If so, the data block is transfered to the controller according to the same principle as a command block.

#### Illegal routine.

Upon detection of a unknown or illegal code sent by the Host, the communication is broken off. A message byte is sent to the Host informing it of the fact that the code received was not legal and the communication process at the controller was aborted. Hence the Host will have to try again.

In figure 5.12 an attempt was made to draw a flow-chart of the receive routine in the transport layer.

After returning from the interrupt service routine, the CPU will resume its previous activity. Whatever process is running

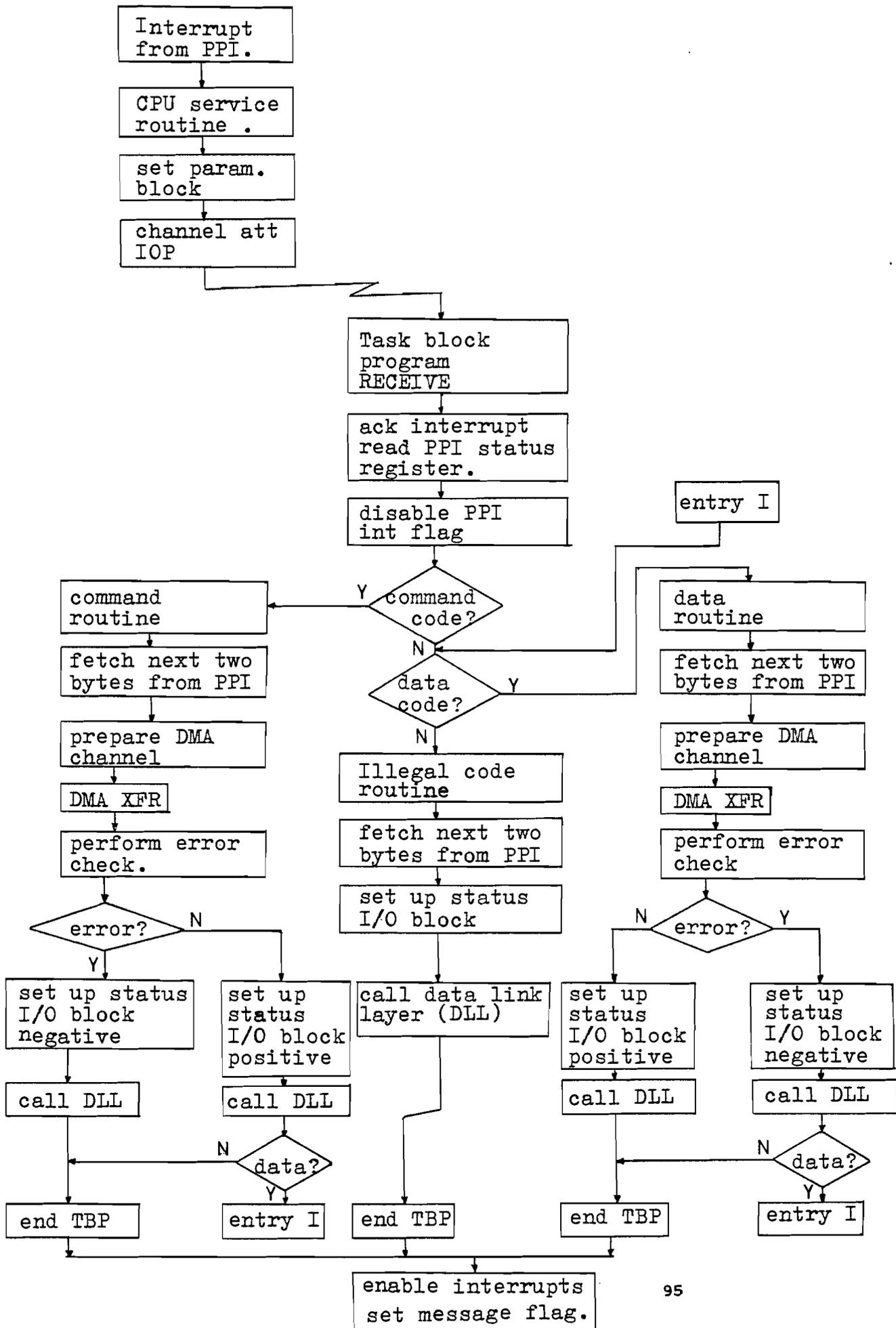


Figure 5.12: Transport Layer receive routine.

on the CPU, it will frequently have to check the I/O buffer flag for reception of a command or data block. Refer to figure 5.13 for this polling sequence.

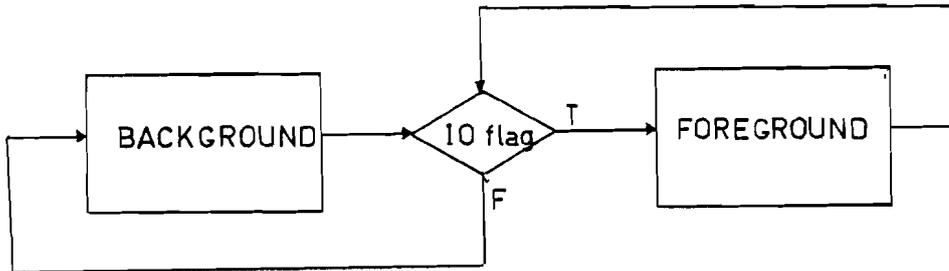


Figure 5.13: Controller DOS polling.

## 2. Sending.

The routine which transfers I/O blocks from controller to host is always called from the presentation layer. It executes the following sequence of actions:

- A status code is written to the PPI which places it on the command/status bus.
- The status register of the PPI is polled to determine whether the host has accepted the status code.
- Two further bytes are sent according to the same principle which contain the length of the I/O block to be transferred.
- DMA channel is prepared.
- DMA transfer is executed.
- The status register of the PPI is polled to determine whether the host has received the I/O block properly. If not, the sequence is repeated.

The flowchart of figure 5.14 shows this sequence.

Although the process described above is simple enough, there is one specific point of interest which deserves special attention. At several points in the routine, busy waits are introduced to verify correct acceptance and reception of certain bytes by the PPI. Circumstances may occur which will prevent these busy wait loops from terminating, causing a hang up of the controller. In order to prevent this, time out facilities have to be provided for.

Unfortunately, no external interrupt -generated by an Interval Counter for instance- can be given to the IOP unless it is performing a DMA transfer which is typically not the case during a busy wait loop.

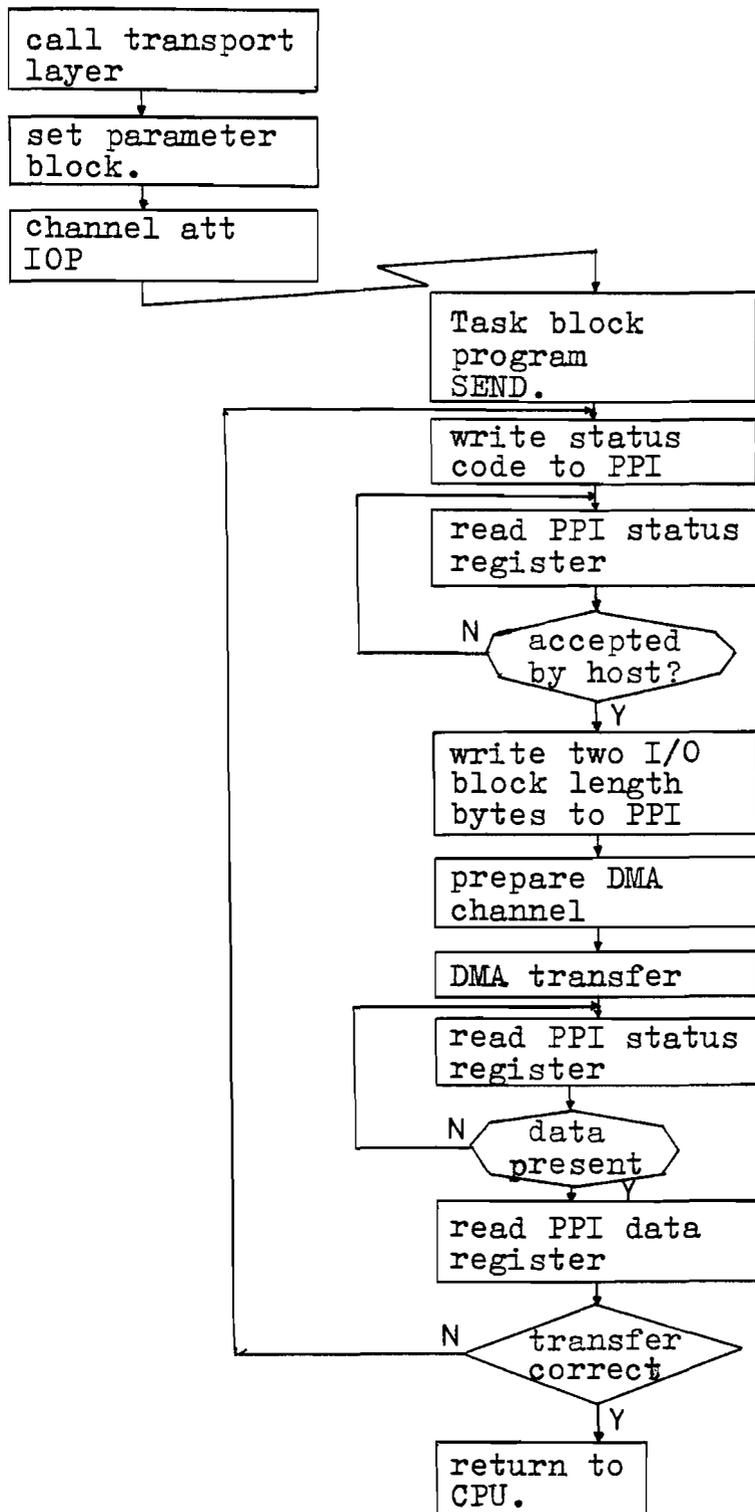


Figure 5.14: Transport Layer sent routine.

Rather than transferring the sent routine from executing on the IOP to the CPU which can be interrupted, a software time out like the one shown in figure 5.15 will have to be added. The software time out is created by a counter which is decremented after every unsuccessful attempt. As soon as the counter decrements to zero, an exit from the routine is made, specifying the cause of the exit to the calling routine. Between successive attempts, a delay is introduced, proportional to the time estimated for the host to respond.

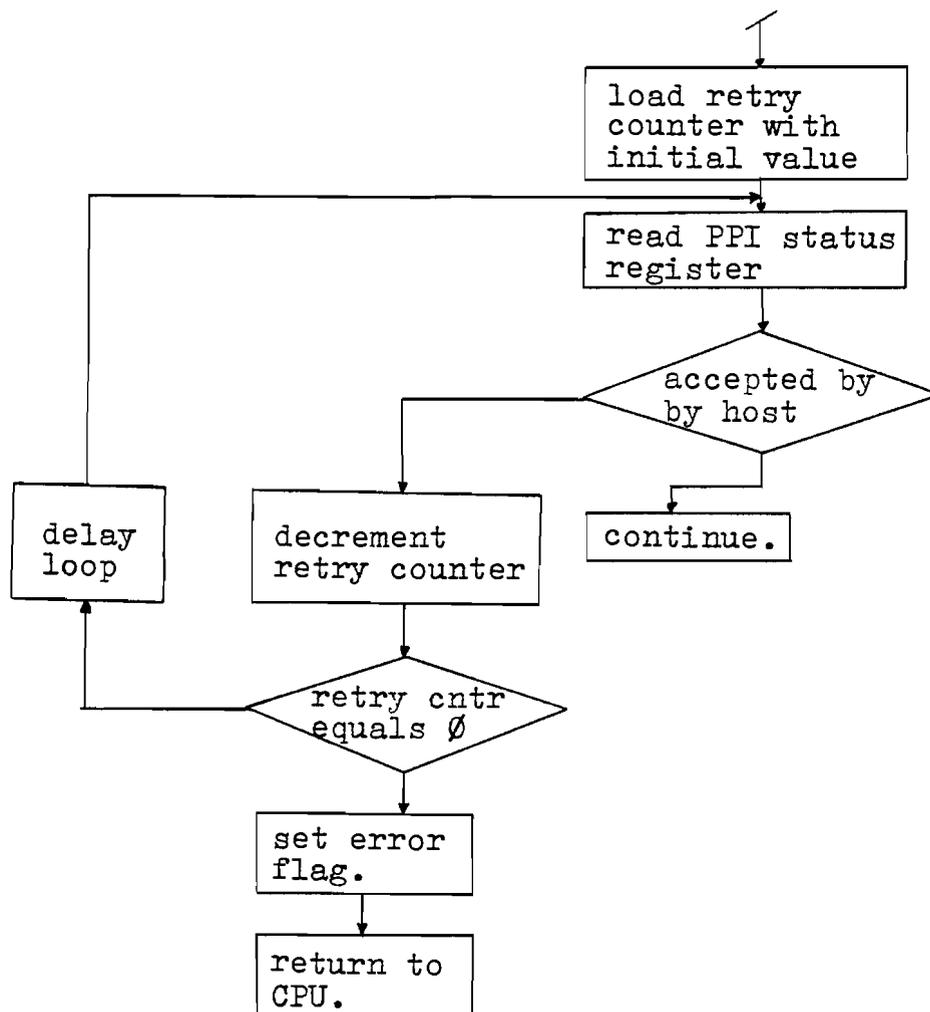


Figure 5.15: Time out.

### 5.3.5. Data Link Layer.

The data link layer is responsible for the actual transfer of bytes or words. For this purpose it has two independent I/O channels available as described in chapter 4. One channel

(command/status channel) uses strobed I/O, supported by the Programmable Peripheral Interface (PPI) and transfers command and status information. The second channel transfers the I/O blocks mentioned previously under DMA.

The communication protocols of both channels are shown in the state diagrams of figure 5.16 and 5.17 respectively.

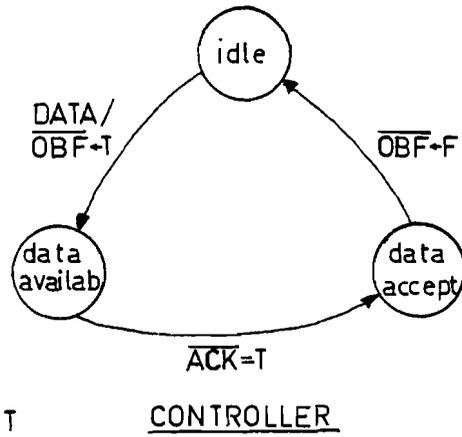
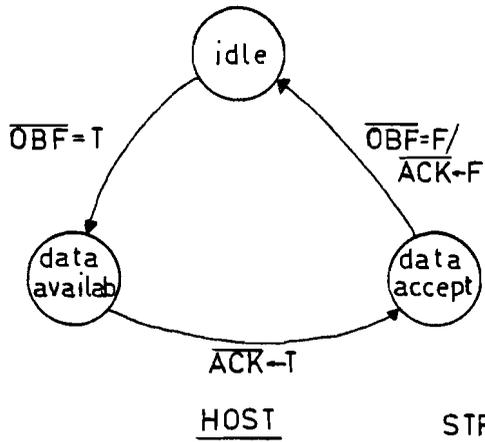
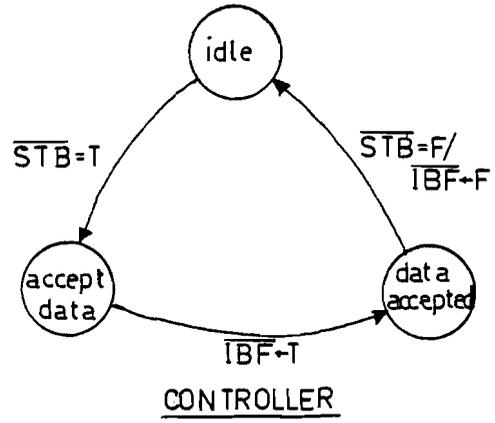
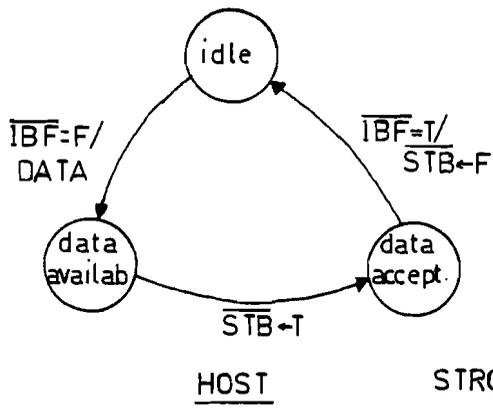
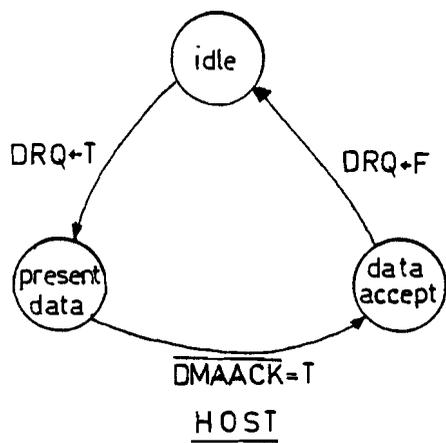
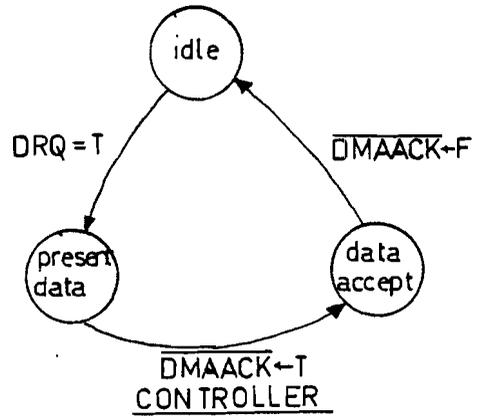
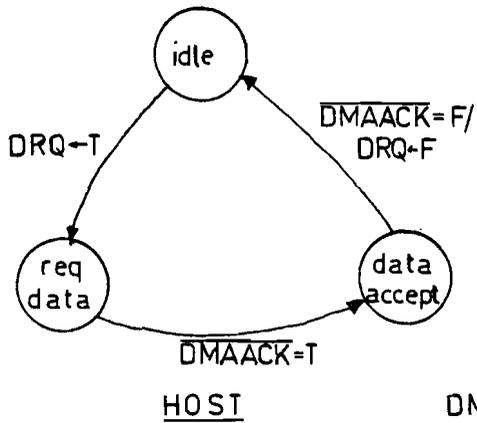


Figure 5.16:



100

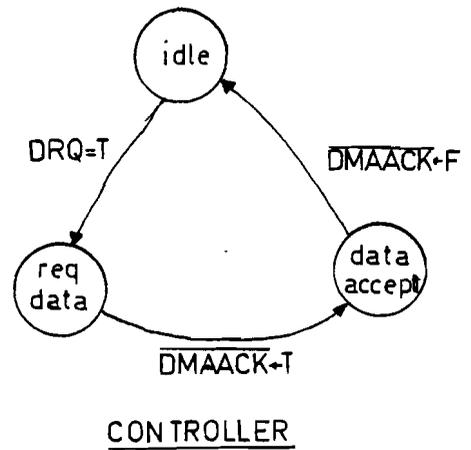


Figure 5.17:

#### 5.4. Disk Protocol Handler.

As an opponent towards the Host protocol handler, this section of the software is responsible for the communication between the controller's processor system and the Winchester disk control hardware. As a logical consequence of this, the protocol used for communicating between the two is determined primarily by the MSC 9000 controller module.

Two types of messages are involved in the communication process, commands and data.

##### 5.4.1. Commands.

The MSC 9000 supports eleven commands which can be divided in commands that don't cause additional data transfers and commands that do. All commands, including those that don't require disk access, conform to the eight byte format shown below.

BYTE 1 : COMMAND CODE.  
BYTE 2 : DRIVE SELECT.  
BYTE 3 : CYLINDER ADDRESS, MSB  
BYTE 4 : CYLINDER ADDRESS, LSB  
BYTE 5 : HEAD ADDRESS.  
BYTE 6 : SECTOR ADDRESS HI  
BYTE 7 : SECTOR ADDRESS LO

Commands that do not require the full eight bytes have to be adjusted by the insertion of pad bytes to maintain this eight byte command length convention.

##### 5.4.2. Data.

Certain commands, like read and write commands, are followed by a data transfer. The amount of data involved in the transfer is related to the nature of the command. Read and Write commands yield data transfers of one sector length. A Status command only requires the transfer of a single byte whereas a Sector Interleave command is followed by a 21 byte transfer. Thus the routine that issues the command should also initiate the data transfer.

COMMAND	CMD CODE	DRV NBR	CYL MSB	CYL LSB	HD ADR	SCT ADR	SCT HI	SCT LOW	Data Bytes	
									Number of Bytes	In/Out of Module
SEEK	0	X	X	X	—	—	—	—	—	—
READ	1	X	X	X	X	X	X	X	N	OUT
WRITE	2	X	X	X	X	X	X	X	N	IN
FORMAT	3	X	X	X	X	—	X	X	—	—
RECALIBRATE	4	X	—	—	—	—	—	—	—	—
STATUS	5	—	—	—	—	—	—	—	1	OUT
READ LONG	6	X	X	X	X	X	X	X	N+4	OUT
WRITE LONG	7	X	X	X	X	X	X	X	N+4	IN
WRITE ALT.	8	X	X	X	X	X	—	—	—	—
SET INTERLEAVE	9	—	—	—	—	—	—	—	S	IN
WRITE CHECK	A	X	X	X	X	X	X	X	—	—
DIAGNOSTIC	B	—	—	—	—	—	—	—	—	—

Figure 5.18: MSC 9000 command table.

#### 5.4.3. Task Block Program.

The task block program (TBP) that implements the relatively simple Disk Protocol Handler is shown in the flowchart of figure 5.19. Calling this routine after configuring the correct parameter block (PB) results in issuing a command to the MSC 9000 and subsequently transferring data if applicable. Figure 5.20 shows the format of the parameter block required for this routine.

TASK POINTER/CHANNEL STATE SAVE AREA		- Task block pointer field
		- Reserved
DRIVE SELECT	COMMAND CODE	- MSC 9016 command format.
CYLINDER ADR LSB	CYLINDER ADR MSB	
SECTOR ADR	HEAD ADDRESS	
SECTOR COUNT LO	SECTOR COUNT HI	
DATA LENGTH	DATA FLAG	- number of bytes for DMA XFR - data transfer flag.
BUFFER POINTER		- pointer to start of system memory where data is to be stored or fetched from.
	STATUS FIELD	

15

Ø

Figure 5.20: Disk Protocol Handler PB.

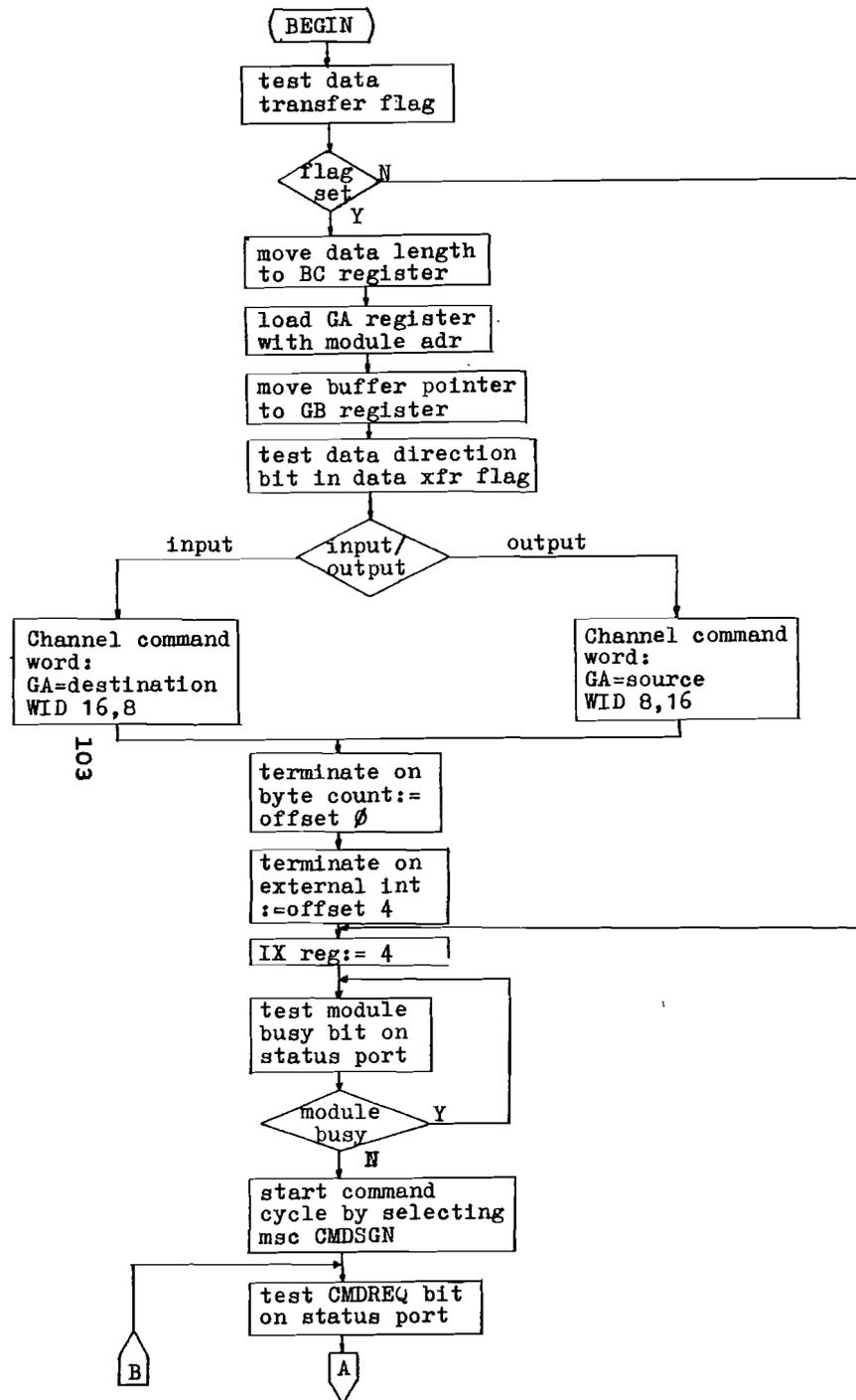
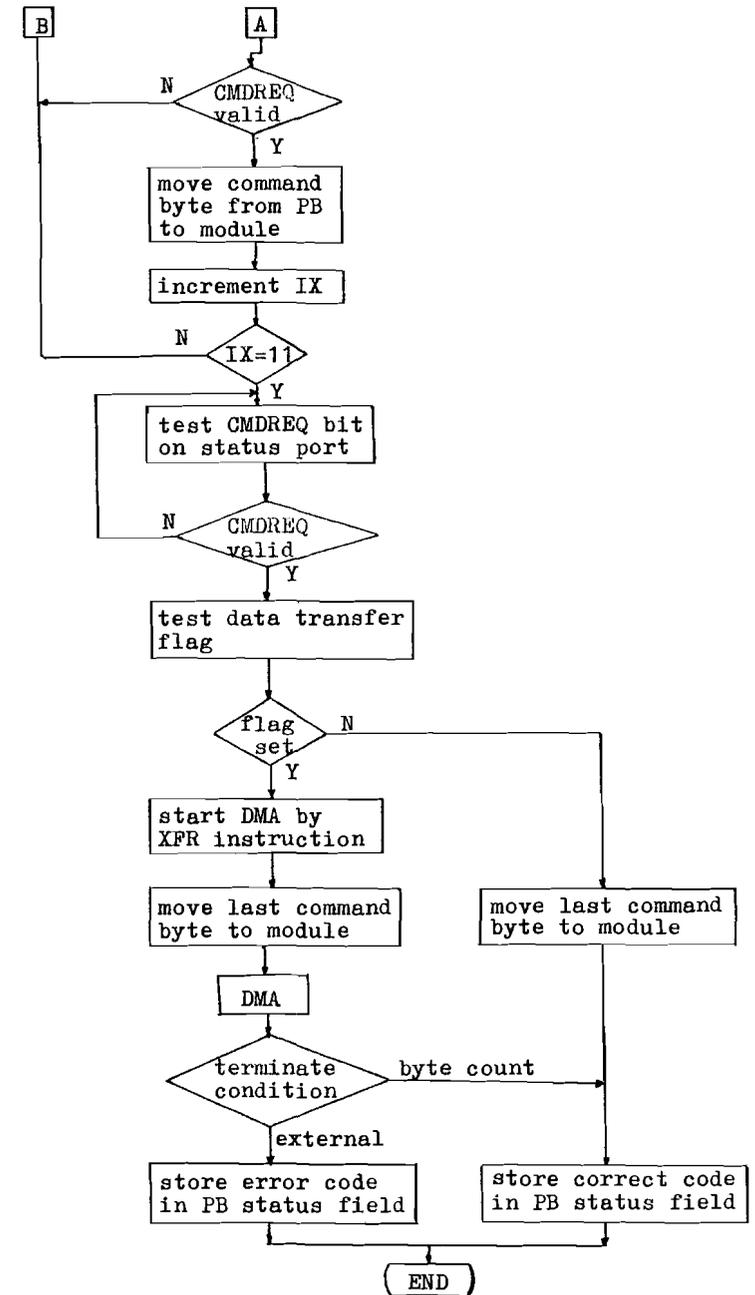


Figure 5.19: Disk protocol handler TBP.



The flowchart is thought to be self explaining. Note that no error conditions are detected by this routine. Therefore it is considered wise to perform a status read of the MSC 9000 after every command. Such a status read can be done by issuing a Status command to the module which will then transfer one status byte. As this is a normal command, the same routine could be used. It would be easier however to devote a special TBP for this task. Refer to figure 5.21 and 5.22 for a flowchart of this TBP and the corresponding PB.

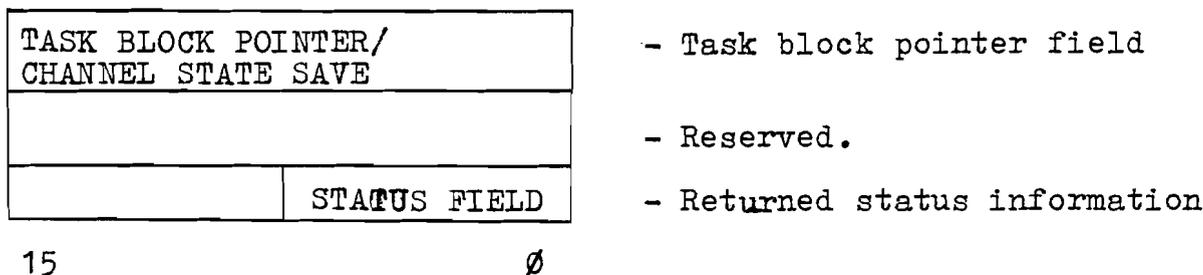


Figure 5.22. Status read PB.

### 5.5. Disk Operating System.

The Disk Operating System is defined as the collection of the following sub-systems:

- Command Handler.
- Free Space Administration.
- File management.

Each of these sub-systems will have to be executed by the 8086 CPU assisted where necessary by the IOP. A possible implementation of this DOS will be given in the form of flow diagrams in the following paragraphs.

#### 5.5.1. Command Handler.

Section 3.4. gave an overview and functional description of the commands supported by the DOS. Each of these commands will be discussed seperately.

#### Control flow.

After succesful completion of the initialization sequence, the controller will enter an idle state. During this idle period, the message flag of the Host Protocol Handler will be checked frequently to see whether the Host has sent a command. If so,

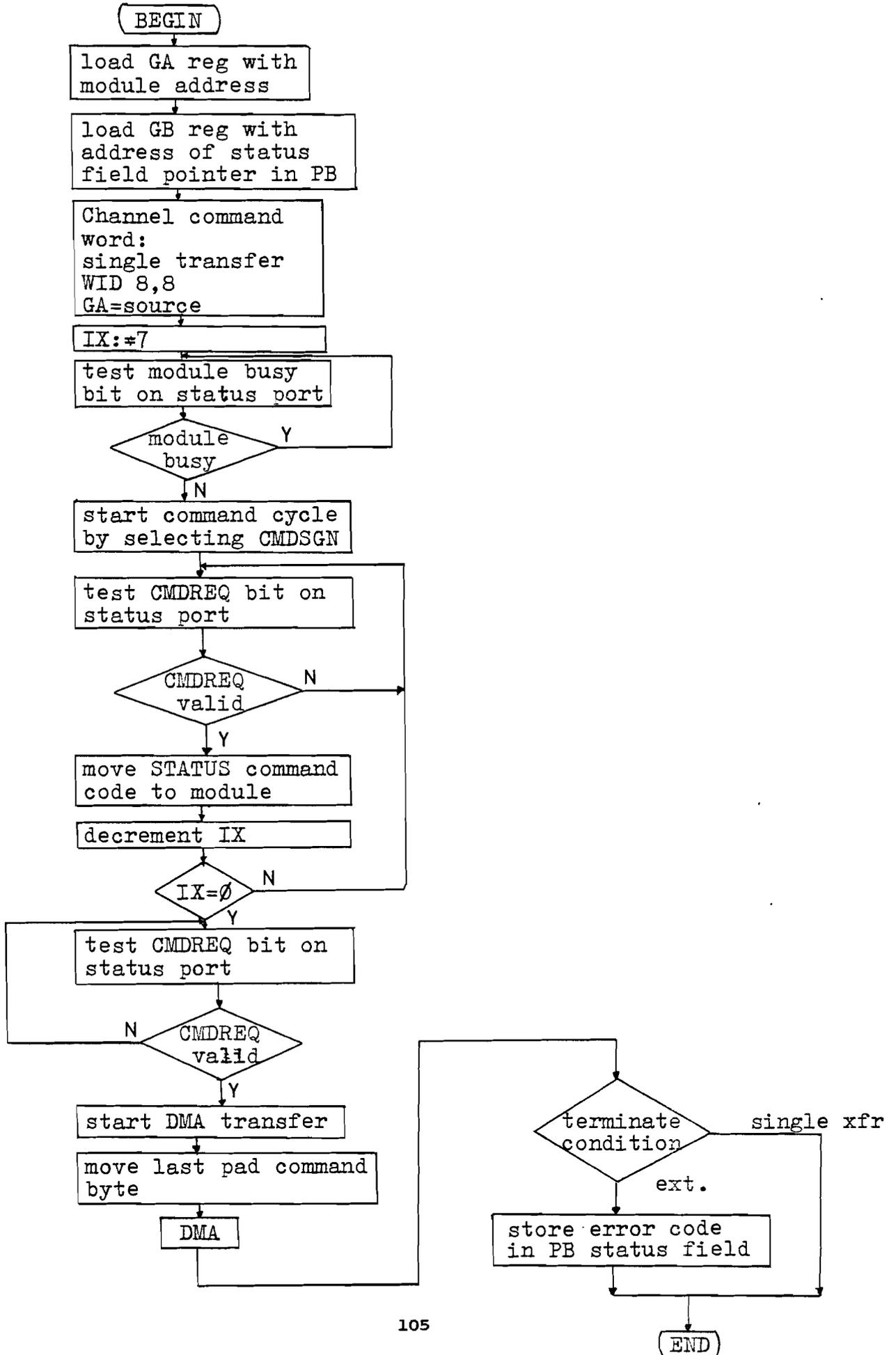


Figure 5.21: Status request TBP.

the command will be fetched from its I/O command block and be interpreted by the Command Handler.

#### CREATE routine.

Upon entering this routine, the name of the file to be created is fetched from the parameter block and a linear search in the directory is performed to determine whether the file exists or not. If not, space is allocated by invoking the Free Space Administration routine. An index node for the file is created and added to the I-list. Updating of the directory finishes the create routine.

#### OPEN routine.

The open routine performs two functions. Firstly it checks whether the file to be opened exists and whether the user has legal access rights, secondly it creates access to the file by adding its index node to the user's open file table. Furthermore the window - a pointer to the next record of the file to be read - is initiated to point to the first record of the file.

#### WRITE routine.

A write operation adds a record to the end of the file. In order for a write to be legal, the window of the file has to be at the end of the file. The file name is fetched from the PB after which access rights are checked. It may be necessary to allocate free space in which case the FSA routine is called. The actual write operation is done by calling the Disk Protocol Handler routine. After a successful write operation, the corresponding Index node is updated and the window incremented.

#### READ routine.

The read operation is to a large extent analogous to the write routine. Reading is done at the position pointed to by the window, one record at a time. The data read from the disk is placed in a buffer in system memory. After creating the proper I/O block, the Host Protocol Handler is called to transfer the contents of the buffer to the Host.

#### SEEK routine.

Whenever the user wants to do a random record read, he has to perform a seek operation, positioning the window at the record he wishes to read. This is done by the SEEK routine. Specifying the record is done by a number. Note that the SEEK rou-

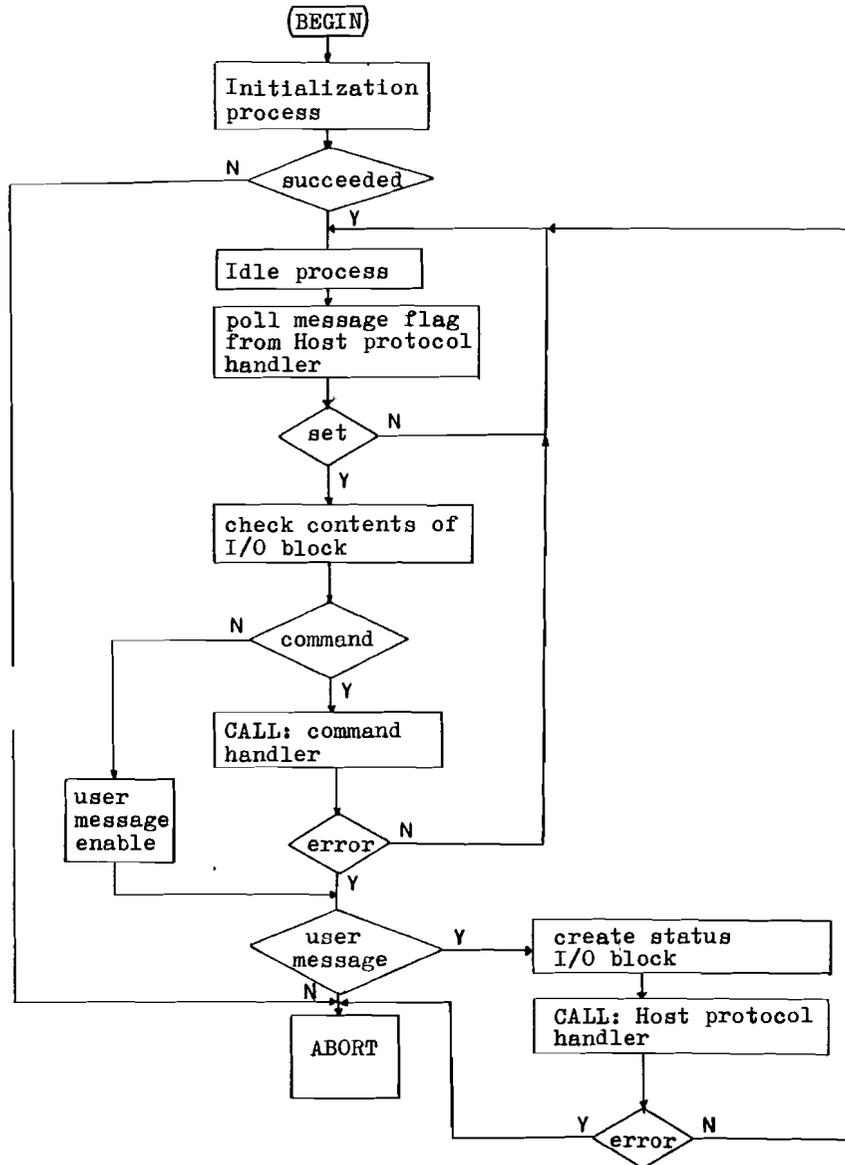


Figure 5.23: Control Flow.

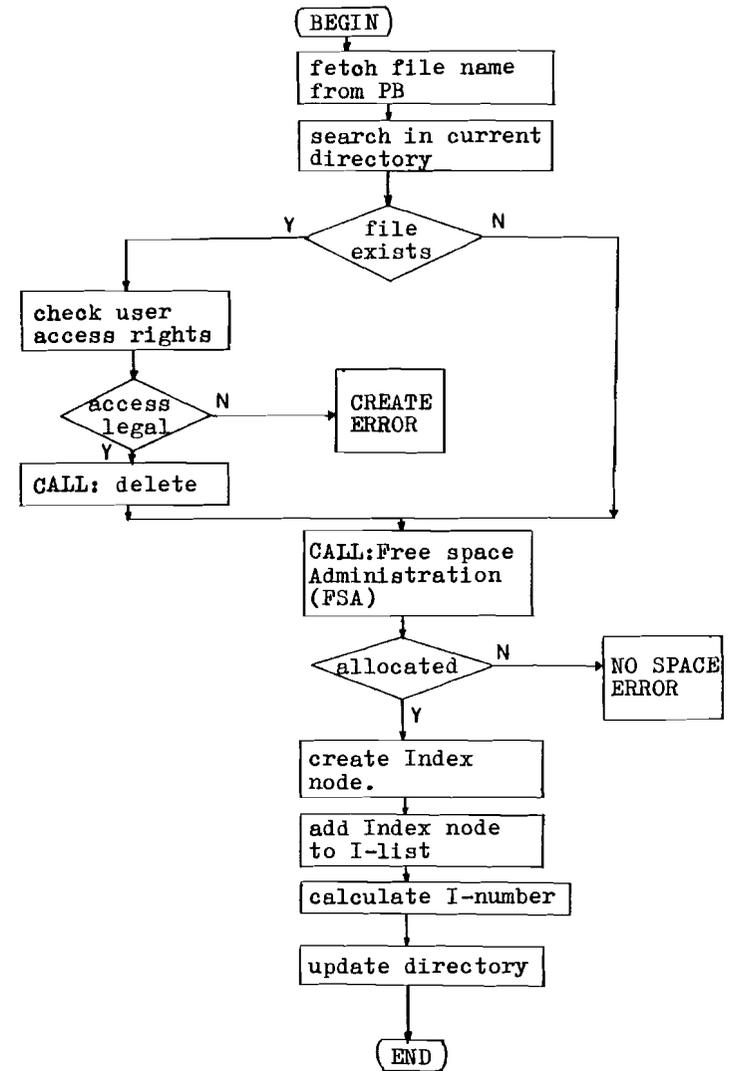


Figure 5.24: Create routine.

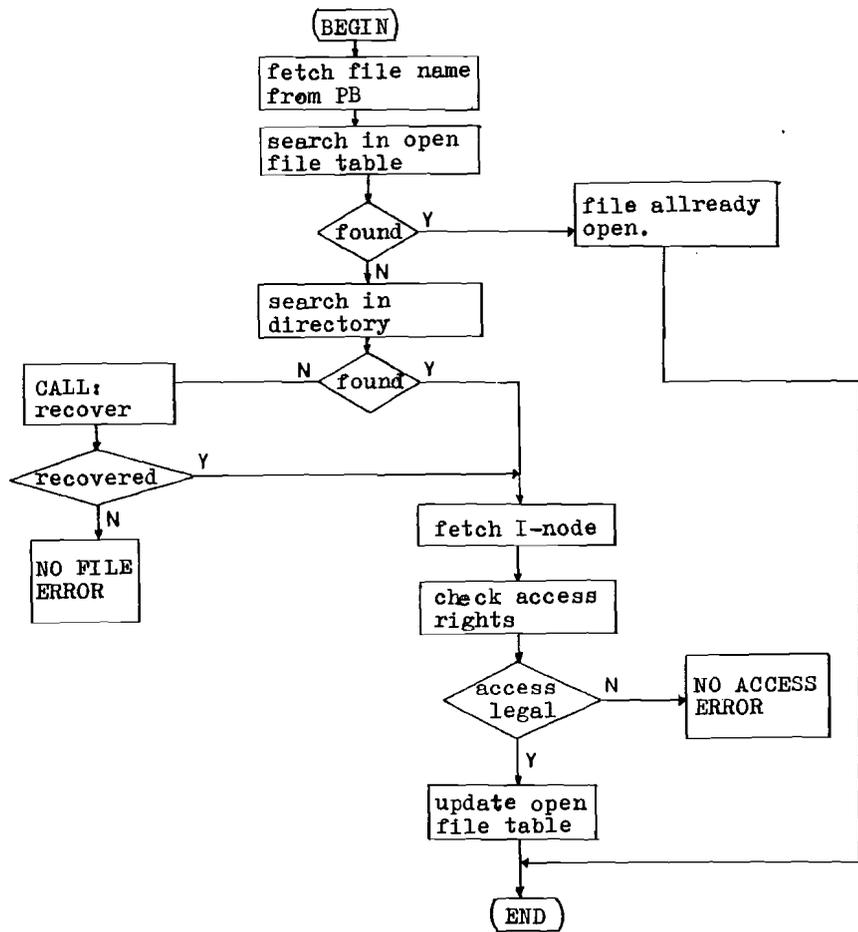


Figure 5.25: Open routine.

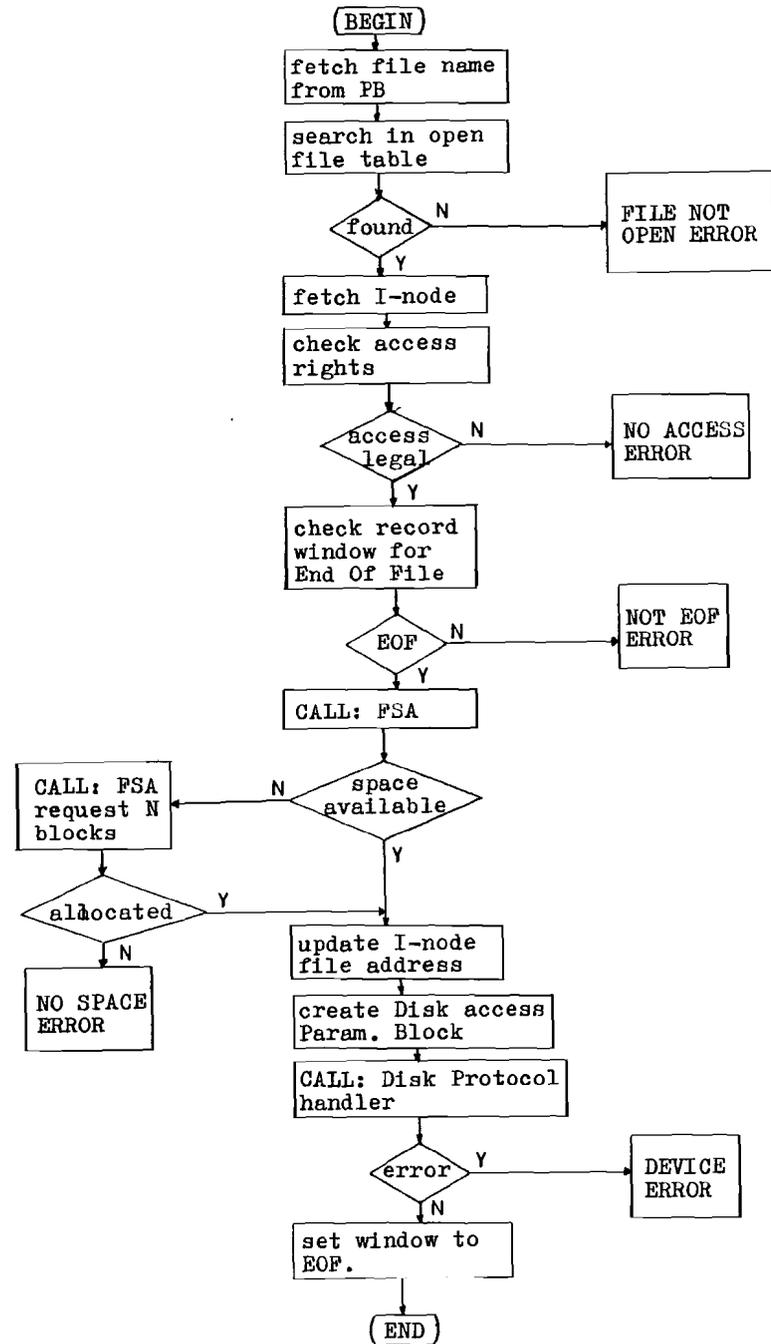


Figure 5.26: Write routine.

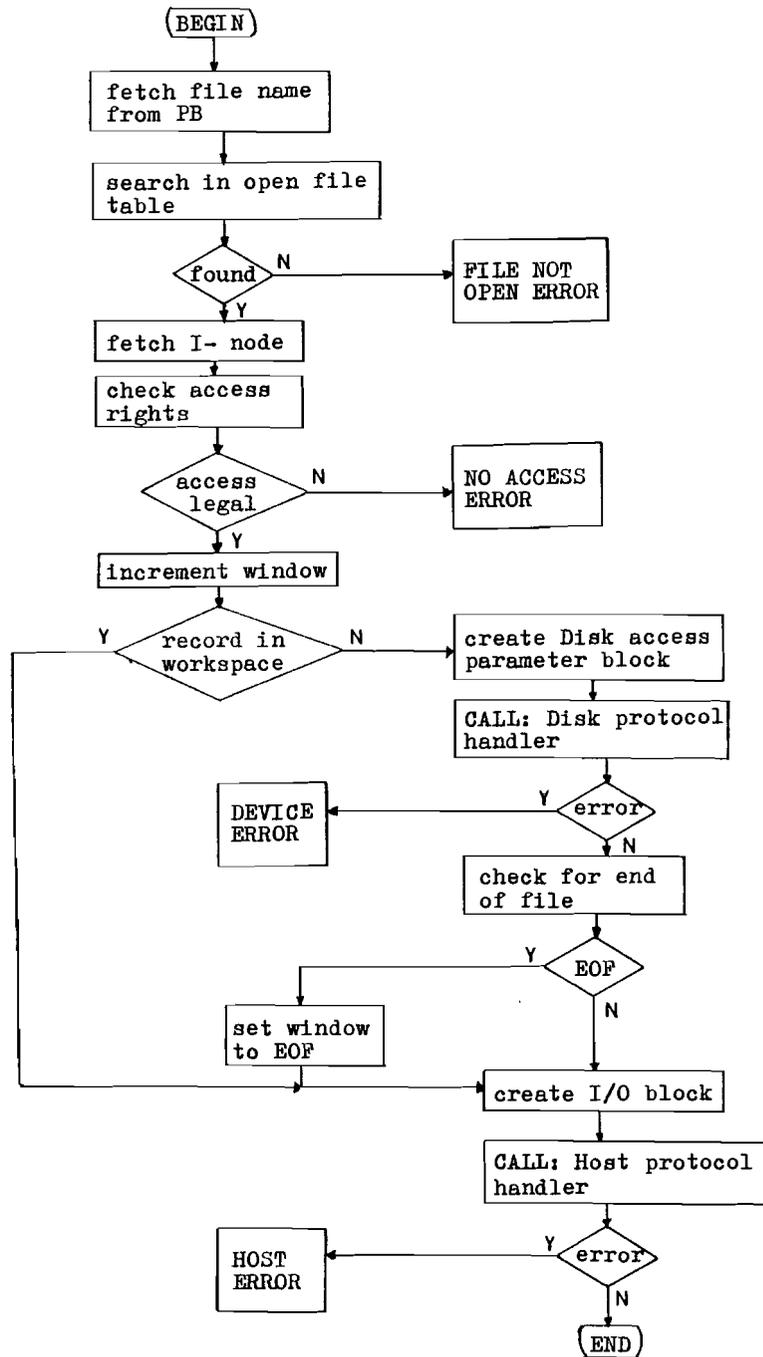


Figure 5.27: Read routine.

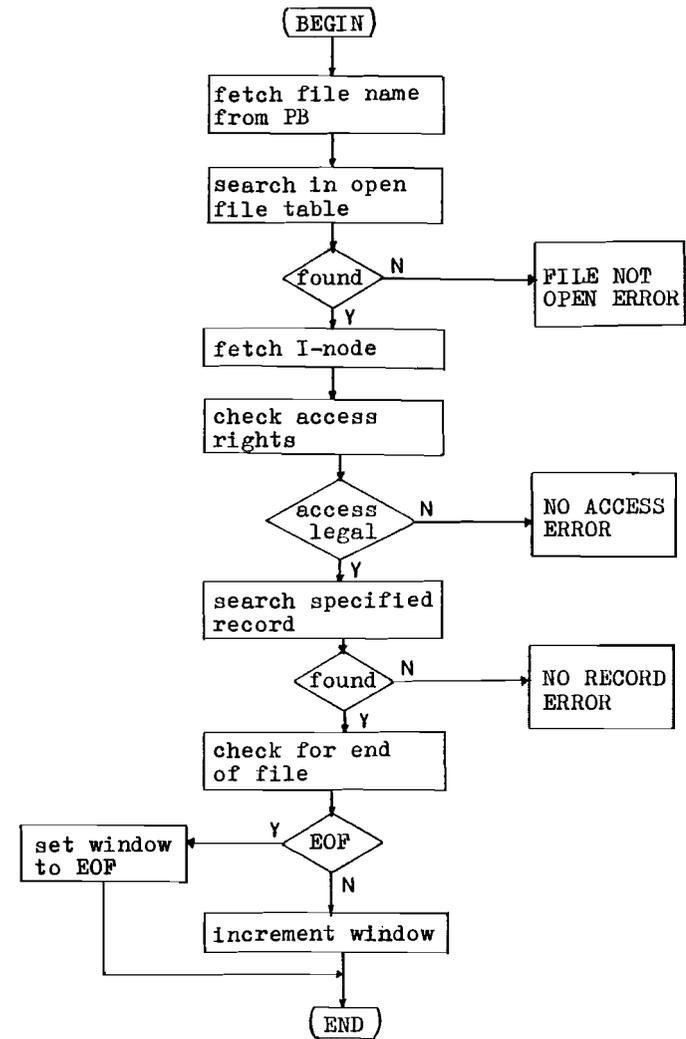


Figure 5.28: Seek routine.

tine does not imply a read operation.

**INSERT routine.**

The insert routine is complex in nature in that it implies distortion of the sequential order in which the records of a file are stored on the disk. Nevertheless it gives the user the very powerful option of adding a record to a file at an arbitrary position, thus enabling updating and extending of files.

An INSERT command has to be preceded by a SEEK command to position the window at the desired position in the file. To allow for insertions in the sequential storage structure, one has to introduce a new type of pointer, the indirect pointer. The alternative to this method is alter the pointers following the position of the record to be inserted. This would involve extensive data movements and would thus be time consuming.

By using indirect pointers, the only penalty paid is a slightly slower access to records of a file beyond an insertion point.

Figure 5.29 illustrates this method.

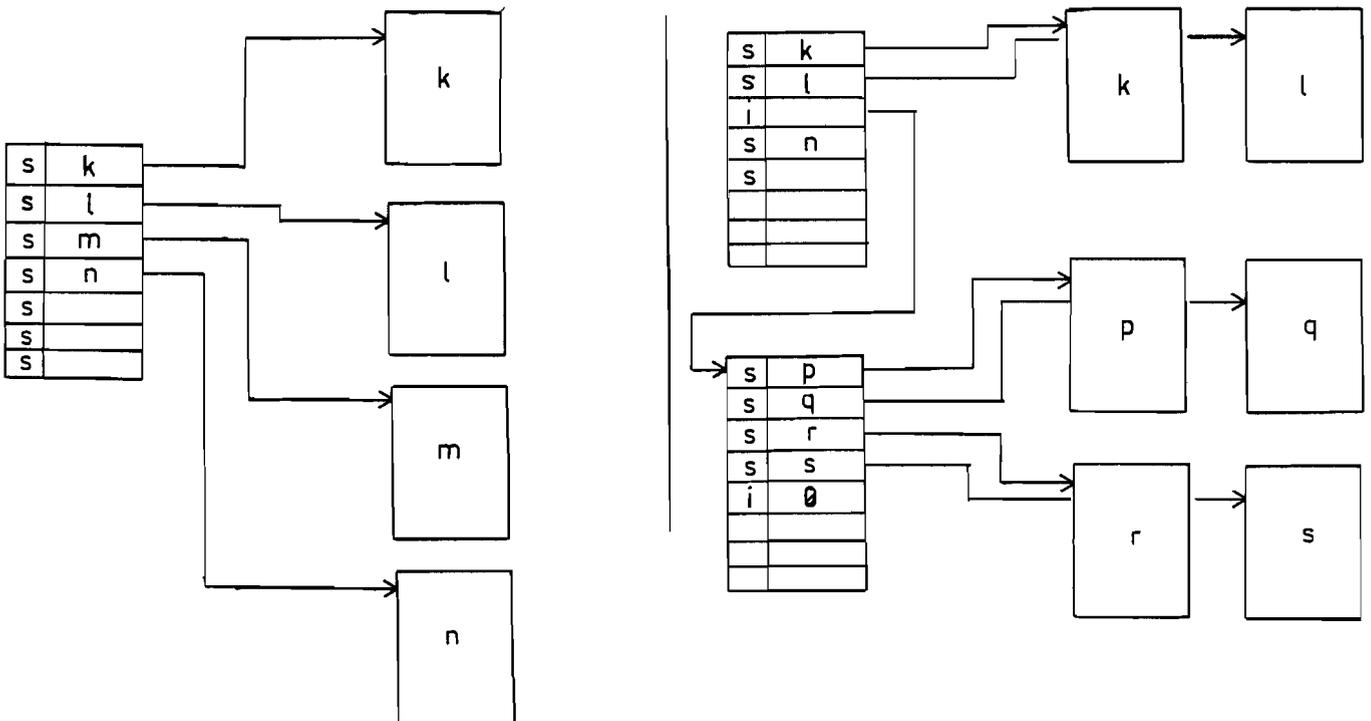


Figure 5.29: Record insertion.

Insertion of records implies a fair amount of administration

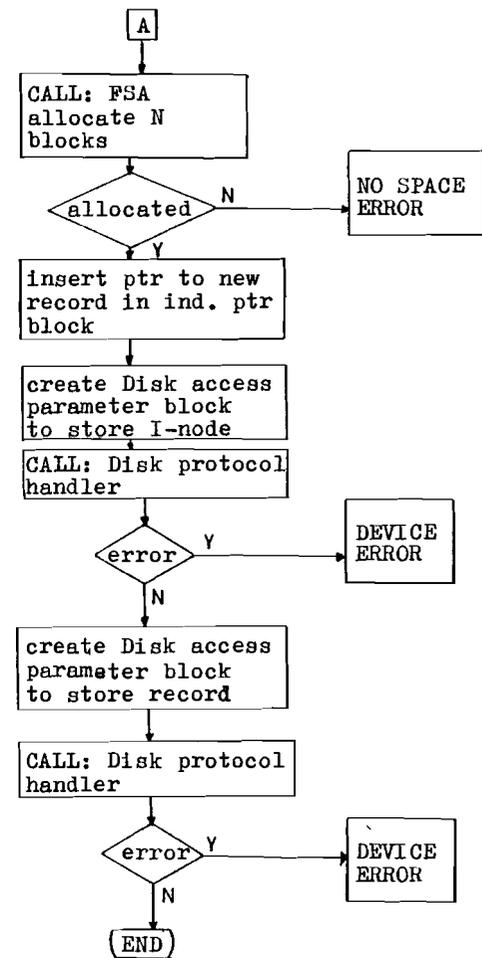
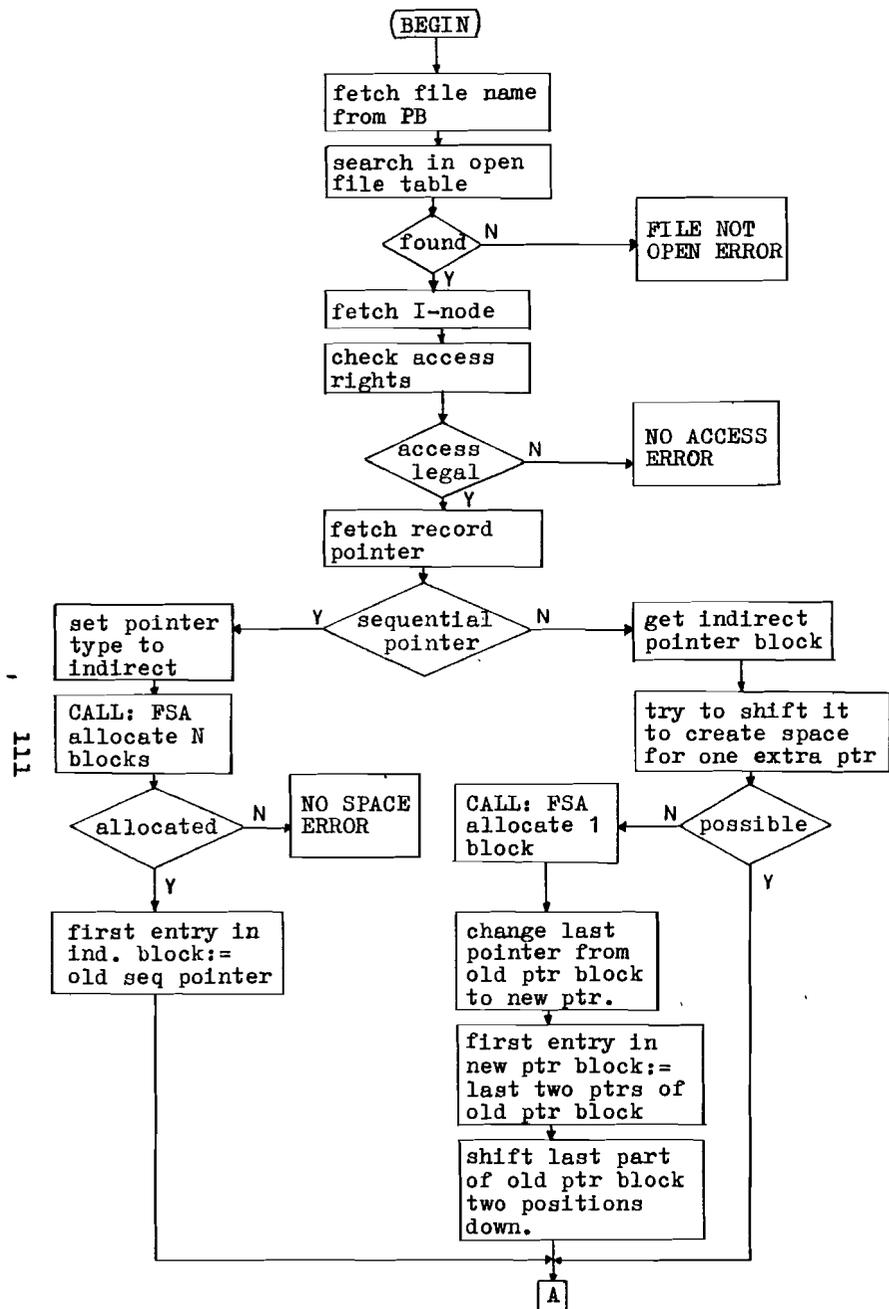


Figure 5.30: Insert routine.

and results in a slower access response. It would be good policy to try and restrict the use of this option to a selected group of users.

#### ERASE routine.

The user can remove a record from a file at an arbitrary position within the file. To do this, he has to perform a seek operation to the record he wishes to remove. Upon calling the erase routine, the pointers to the disk blocks containing the data of that record are set to deleted. This means these pointers are still present but not accessible. The blocks turned free by the removal of the record are registered as free blocks again by the Free Space Administration.

#### CLOSE routine.

Calling the close routine results in the removal of the specified file from the open file table. As such, this routine is quite similar to the open routine.

#### DELETE routine.

The delete routine removes the directory entry of the specified file from the directory list. All blocks on the disk containing the data of this file are returned to the Free Space Administration. Before returning control to the command handler, the deleted file name with its corresponding I node are added to the recover file.

#### RECOVER routine.

Whenever a user decides to try and recover a previously deleted file, the command handler will call the recover routine. After confirmation of the fact that the file to be recovered was actually deleted, the recover file is read into workspace. Whether it can be recovered or not depends on its presence in this file. If it is, a comparison is made between the I node contents and the Free Space Administration to see if the blocks previously occupied by the file are still free. If not, recovery is impossible since the data of the file was overwritten. If the blocks in question are still free, they may still contain the original data although there is no certainty. By creating a new I node the data of the file are made available to the user. It is the user's responsibility to verify the contents of the recovered file.

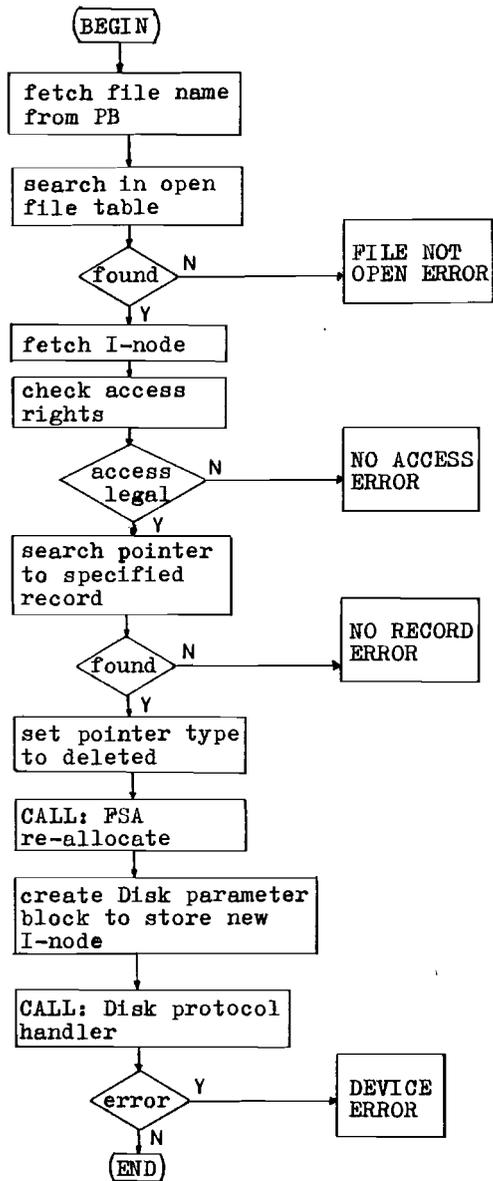


Figure 5.31: Erase routine.

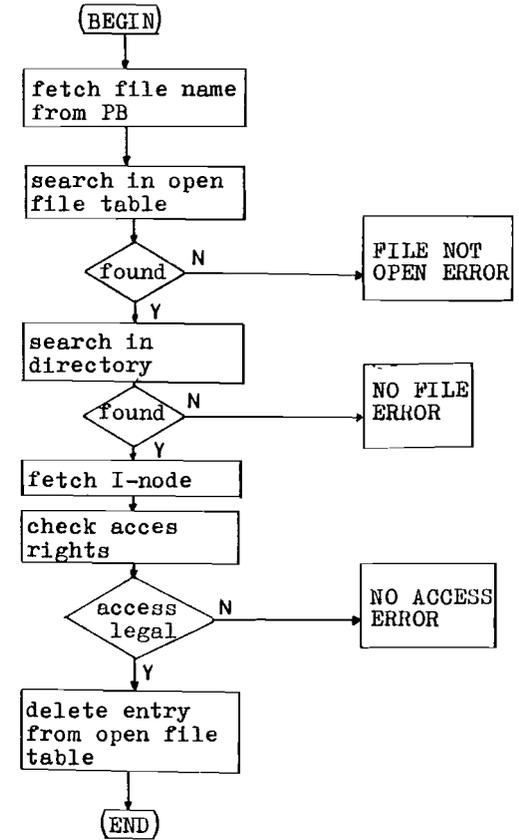


Figure 5.32: Close routine.

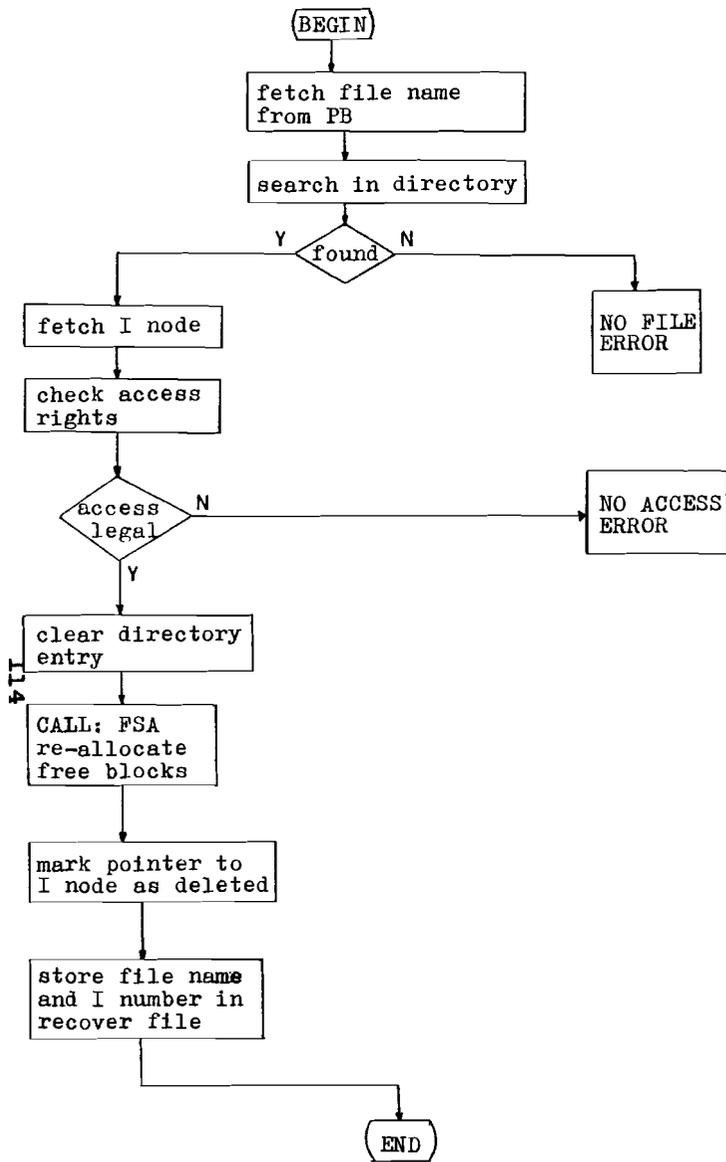


Figure 5.33: Delete routine.

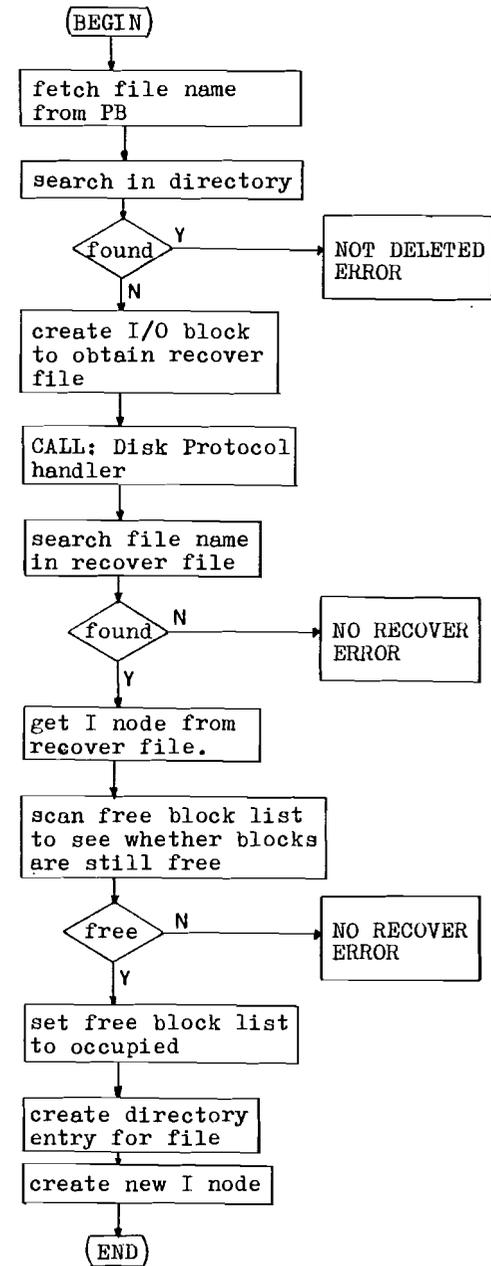


Figure 5.34: Recover routine.

DIRLIST routine.

The DIRLIST routine starts with inspecting the parameter block for a file name. If no file name is present, it assumes the user wants a listing of the root directory and makes a copy of the root directory's I node in an area of workspace called the Current Index Node (CIN). If a file name was specified, a check is performed on the file type of that file to see whether it actually is a directory type file. If not, an error condition occurs.

The I node which is present in the CIN is subsequently scanned to obtain all file names and file types it points to. This information is stored in a buffer. When the end of the directory has been reached, the contents of the buffer is transferred to the host by calling the Host Protocol Handler.

RENAME routine.

The renaming of a file is quite a simple operation. Some checks are built in. Firstly to verify whether the file to be renamed exists at all, secondly whether the new name supplied by the user was not previously used for naming another file.

ENQUIRY routine.

The explicit function of the enquiry routine is to fetch the information stored in the Index node of a file and make this information available to the Host. For this purpose, the directory is searched until the specified file name is found. The information present in the Index node is transferred to a buffer which is subsequently sent to the Host.

#### 5.5.2. Free Space Administration.

The Free Space Administration entails a dual function:

1. Registration of used and free disk blocks.
2. Allocation and re-allocation of blocks.

Registration.

For the registration of free and occupied blocks, a bit map approach was chosen as explained in chapter 3.

Since every position within the bit map corresponds to a physical block (or sector) on the disk, a translation from bit position to disk address must be made.

The bit map is made of an array of consecutive words, starting with word number zero. A bit map address consists of a word

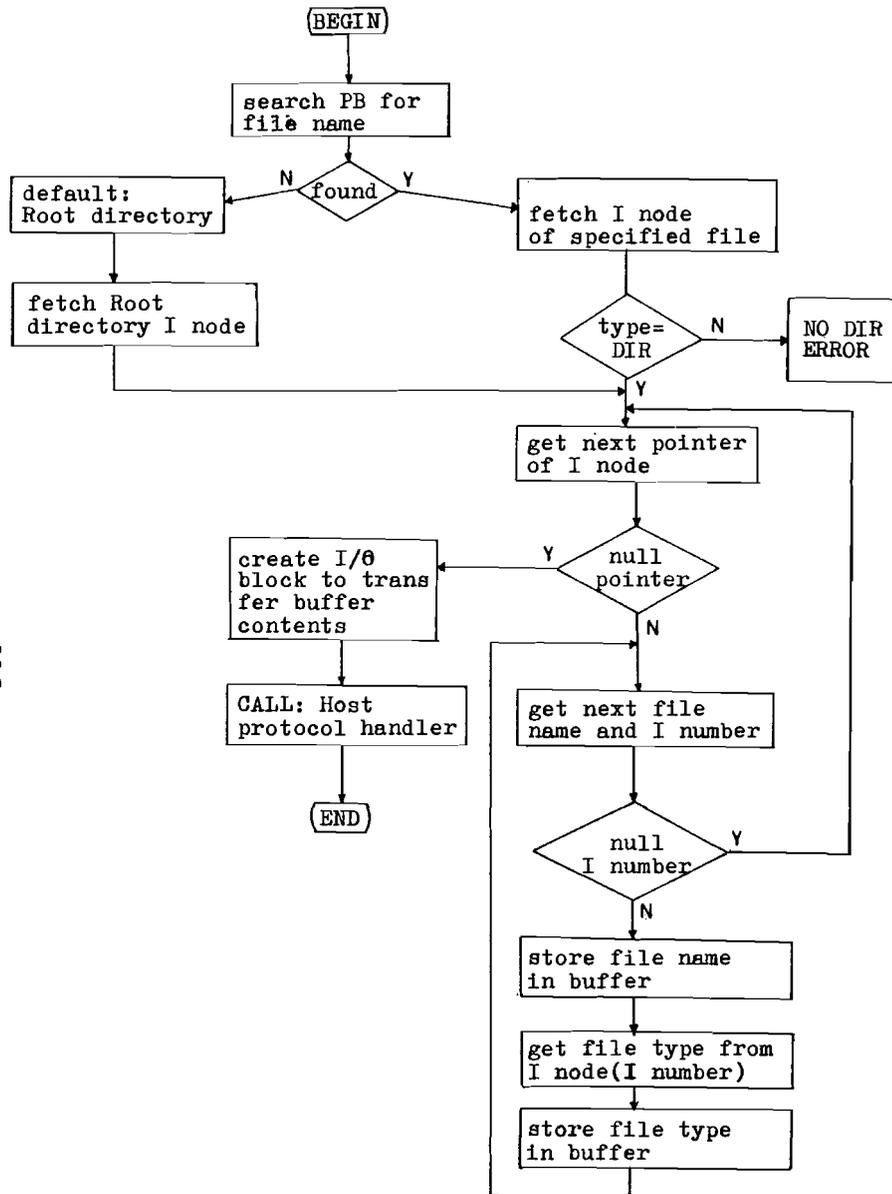


Figure 5.35: Dirlist routine.

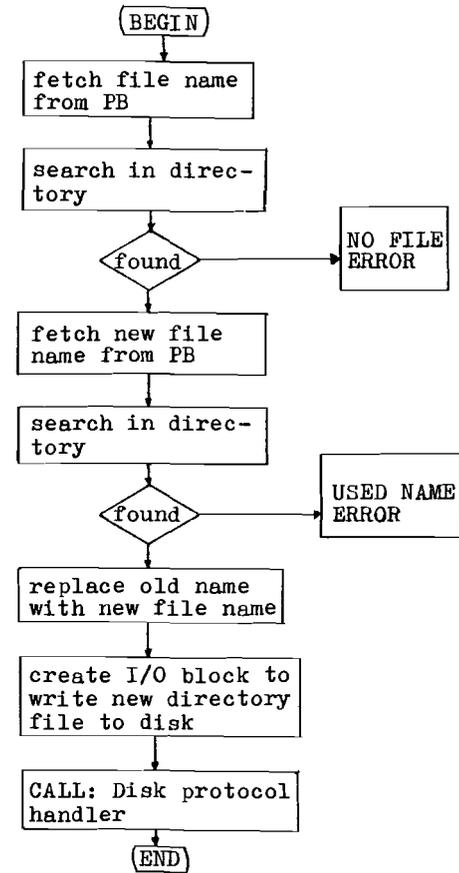
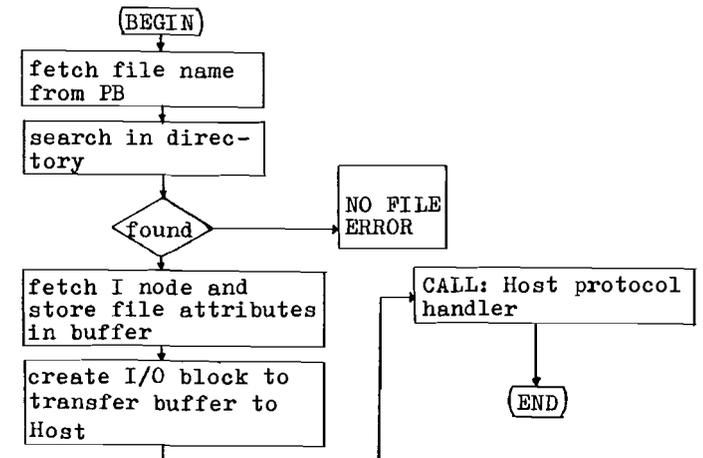


Figure 5.36: Rename/Enquiry routine.

ENQUIRY



number and an offset into that word. A disk address is made of a head, cylinder and sector number.

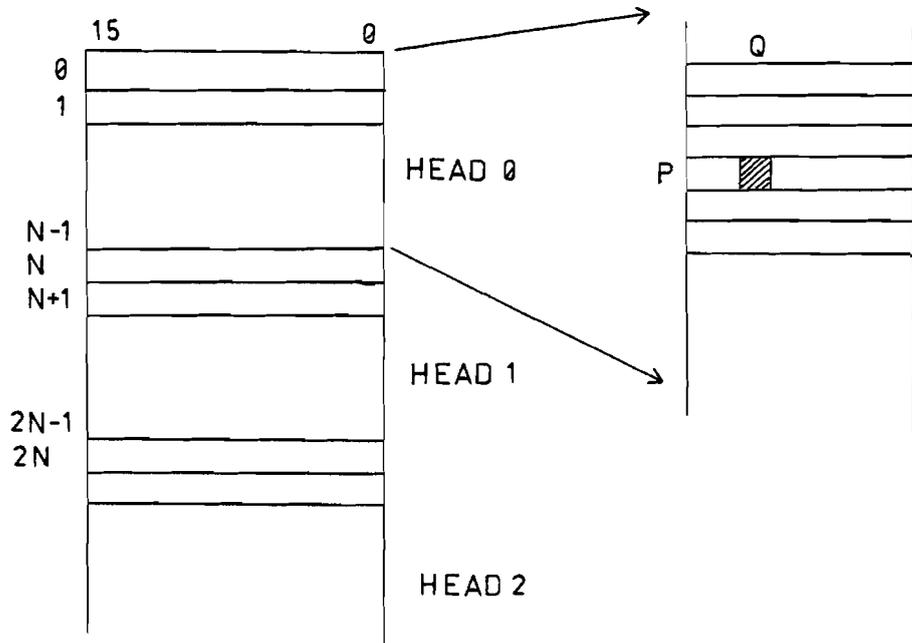


Figure 5.37: Bit map.

The mapping procedure is illustrated by figure 5.37. Given the position of a bit within the bit map as P,Q where P equals the word number and Q the offset value, the disk address can be calculated as such;

$$\begin{aligned} \text{Head number} &= P \text{ div } N \\ \text{Cylinder number} &= ((P - P \text{ div } N) * 16 + Q) \text{ div } M \\ \text{Sector number} &= ((P - P \text{ div } N) * 16 + Q) \text{ mod } M \end{aligned}$$

$$\begin{aligned} \text{where } N &= (\text{no.of cylinders} * \text{no.of sectors/track}) / 16 \\ M &= \text{no.of sectors/track.} \end{aligned}$$

Reversely, the bit position of a block within the bit map can be calculated from the disk address H,C,S by;

$$\begin{aligned} P &= (H * N + C * M + S) \text{ div } 16 \\ Q &= (H * N + C * M + S) \text{ mod } 16 \end{aligned}$$

To give an impression of the size of such a bit map, an example of a medium sized Winchester drive is given below.

Capacity	34 Mbyte formatted.
No of heads	4
No of cylinders	1024
No of sectors/track	17
No of bytes/sector	512

Total number of blocks    69632  
 Bit map size required      4352 words

Thus the bit map would occupy 9 blocks on the disk in this particular case.

Note however that when the number of sectors per track is not a power of two, the arithmetic operations involved in mapping between bit map address and disk address becomes quite complex.

Re-allocation.

Blocks being turned free, probably as a result of the deletion of a record or a file, have to be registered as such. This means converting the corresponding disk addresses to bit positions within the bit map and setting these bits to indicate - free blocks. A check to see whether the block turned free was actually occupied is also performed in the process. Refer to figure 5.38.

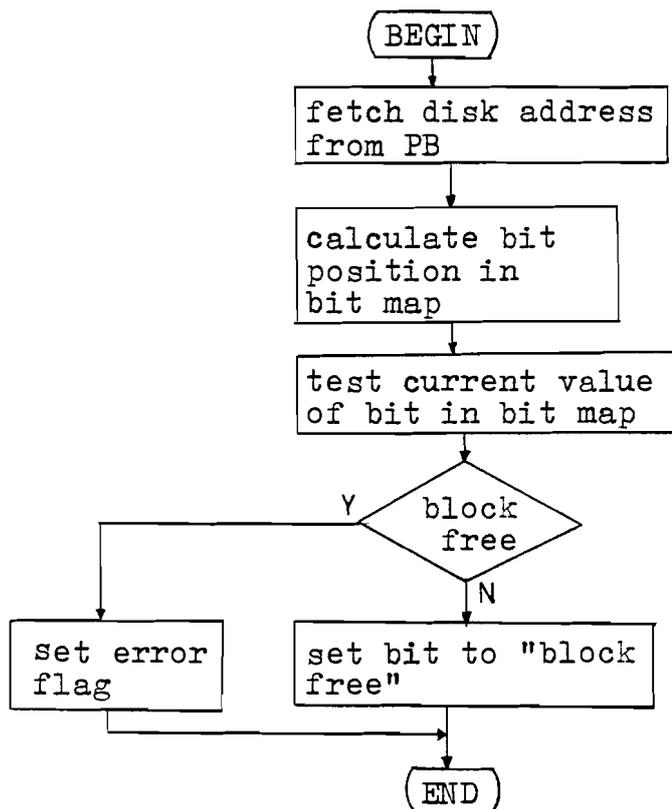


Figure 5.38: Re-allocation.

Allocation.

Allocation of disk space to a file or record is done by request from the command handler. A few things should be considered when designing an algorithm for this purpose.

- Assigned space should be contiguous as much as possible in order to minimize head movements and thus to reduce average access time.
- Space assigned to an existing file for the purpose of extending it should be as close to the old file position on the disk as possible for the same reason.
- Garbage collection, i.e. moving files across the disk to create contiguous free space, should only be done when disk saturation occurs.
- Generally, newly allocated blocks should be at the end of the previous allocation, i.e. the disk should be filled from the lower tracks upwards. This is to prevent tedious fitting in of blocks in small areas previously turned free by the deletion of records, when there is still sufficient contiguous free space on the upper tracks.

In order for the allocation routine to meet these requirements, it requires knowledge of the following information:

1. Number of blocks to be allocated.
2. Position of the last allocated block.
3. File extension or new file.
4. In case of a file extension, position of the last block of the file.

15

∅

NUMBER OF BLOCKS	NEW FILE/ EXTENSION	Request information
CYLINDER ADDRESS		Last allocated block address
HEAD NUMBER	SECTOR NUMBER	
CYLINDER ADDRESS		Last block of file to be extended
HEAD NUMBER	SECTOR NUMBER	
BUFFER POINTER		Pointer to memory space where block list must be stored.
	ERROR FIELD	

Figure 5.39: Allocation routine PB.

ALLOCATION

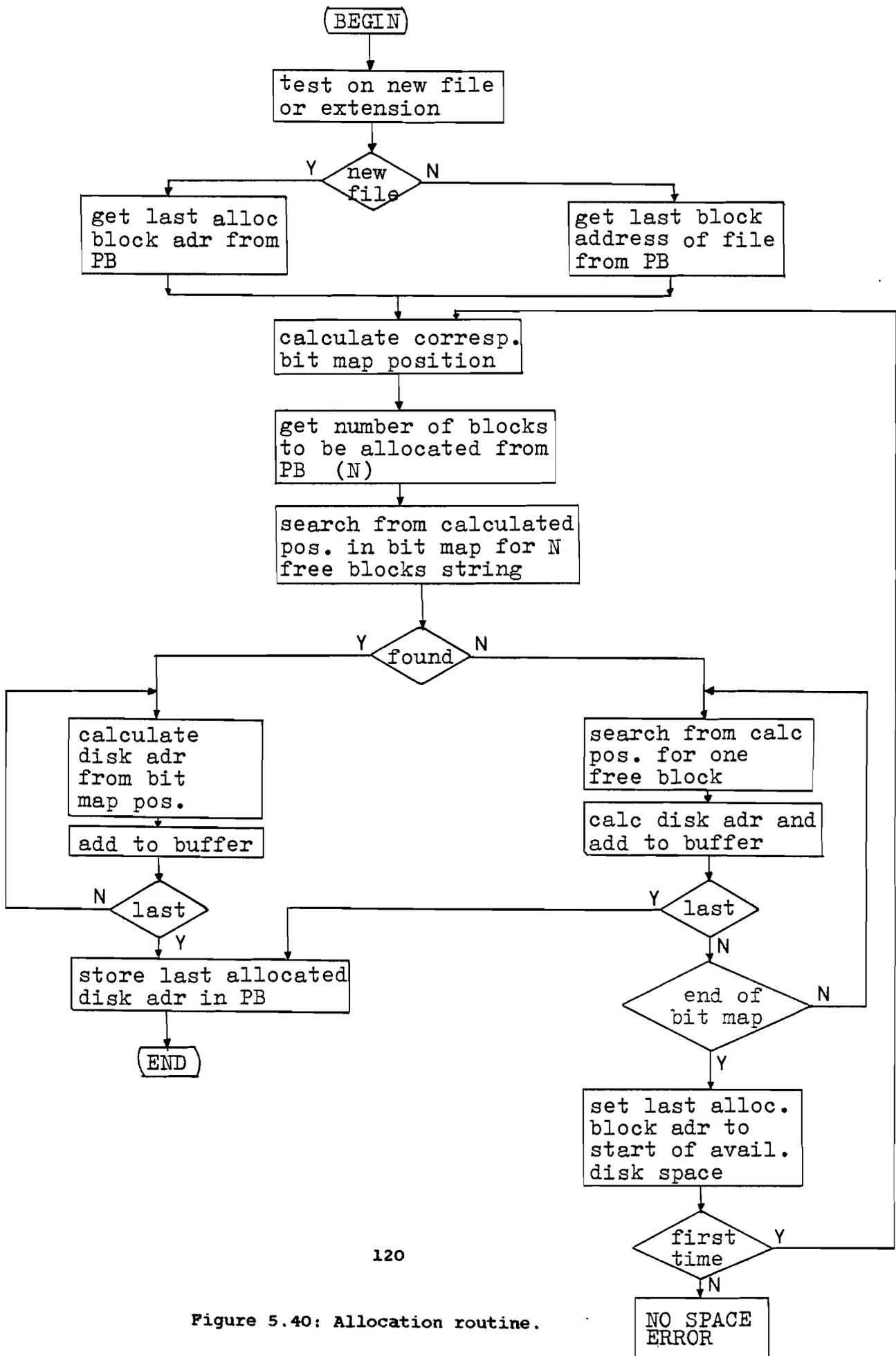


Figure 5.40: Allocation routine.

Calling the allocation routine results in the return of a list of disk addresses which can be assigned to the record or file they where requested for. This list is stored in workspace at a location pointed to by a pointer supplied by the calling process.

Communication between calling process (command handler) and allocation routine is performed through a parameter block. The format of this PB is shown in figure 5.40.

### 5.5.3. File Management.

The File Management section is made up of a database, containing all the lists and tables required for the file access mechanism, and a set of routines that operate on this database. The list and tables mentioned were described in chapter 3 previously. The set of routines described below, offers additional service to the process that wishes to use the information contained in the database.

#### Access right control.

Using the information stored in a file's index node regarding the identity of the file's user and owner in conjunction with the kind of access requested, the access rights can be checked. Three types of access are possible, read only, read/write and execute only. Also, three levels of access authority are distinguished, owner, user and group. Hence the format of the access information field in the I node is as such:

OWNER	USER	GROUP.
exc r/w ro	exc r/w ro	exc r/w ro
x x x	x x x	x x x

#### Index node create routine.

When called by the create routine of the command handler, this service routine configures and stores a new Index node using the information supplied by the calling process.

#### Search file name.

This routine searches through a list or table, depending on the process which called it, and when found returns a pointer to the location where the filename was found. When not, an error message is returned. Normally this search would be linear since no ordering of file names takes place.

#### 5.5.6. Error Handler.

As can be seen in the flowcharts presented in this chapter, several error conditions can occur at various points during program execution. The purpose of the error handling routine is to minimize the effects of these error conditions and to prevent them from obstructing further operation of the controller.

Error conditions are divided into three categories:

1. Recoverable errors.
2. User errors.
3. Fatal errors.

#### 5.6.1. Recoverable errors.

Recoverable errors are those errors whose effects can be eliminated by either trying to obtain the same result through an alternative way or by simply retrying the operation that caused the error.

Recoverable errors frequently occur in communication processes where the loss of some information caused by external interference results in an error condition. By restarting, the same error will usually not occur again. Most of the errors that can be avoided or compensated for this way, are covered locally in the software in the software routines. Refer in particular to the frequent occurrence of time-out loops in the host and disk protocol handler routines. Only after a predetermined number of retries, a fatal error is signalled.

Another type of semi-recoverable error is the one that occurs when requested free disk space is not available, the so called NO SPACE ERROR. Three possibilities exist to overcome this problem:

1. Garbage collection. By searching through the file management table's, all areas of the disk that are not in use are turned free. In the process, all files are rearranged so as to avoid small unused gaps between them.
2. Change volume. Though disk space is usually requested on a specific volume, it could in some cases be diverted to another where more space is available. Note that this has considerable consequences for the file administration information.
3. Return to User. By returning to the user, it becomes his responsibility to create free space by deleting one or more files or records. Thus, a status I/O block is configured and sent to the host.

### 5.6.2. User errors.

A considerable percentage of errors falls into this category. User errors are caused by the issuing of non-executable commands by a user. Upon detection of such an error condition, the controller aborts its current process, typically the execution of a user command, and creates a status I/O block, informing the user of the kind of error he made. Armed with this information, the user can take the appropriate action.

User errors are: FILE NOT OPEN  
NO FILE  
NO RECORD  
NO ACCESS  
NOT EOF  
CREATE  
NO RECOVER  
NOT DELETED  
NO DIRECTORY  
ILLEGAL NAME

### 5.6.3. Fatal errors.

As the name suggests, these kinds of errors result in a situation the controller cannot handle. Generally this means a hardware failure occurred in the controller itself or in the Host's communication section.

Fatal errors are: DEVICE ERROR, controller module malfunction.  
COMMUNICATION ERROR, Time-out occurred during  
DMA transfer or busy wait.  
HOST ERROR, Host does not respond correctly

Upon detection of a fatal error, the controller enters an idle state.

## Chapter 6. CONCLUSIONS AND RECOMMENDATIONS.

Due to the fact that the controller module used in this design was not available at the time, no prototype could be built and tested. Therefore, no experimental data or test results can be presented here. Nevertheless, the design presented in this report is considered to be a feasible solution to the interface problem associated with the application of Winchester Disk Drives.

### 6.1. Conclusions.

The most predominant advantage of the design at hand is the high degree of service it offers to the host computer. The command level supported by this controller related to file manipulation is comparable to that of a high level language like Pascal.

From a designer's point of view, the use of an integral controller module like the MSC 9000 eliminates the traditional design problems of the disk controller unit hardware. The penalty for this is a lower degree of flexibility in respect of interface standards and drive types that can be accommodated. Additionally, a controller conform this concept is inherently expensive compared to those that lack a DOS facility. Thus its application is restricted to small and medium sized mini-computer systems, rather than micro-systems.

The use of the UNIX operating system as a guide-line for designing the controller's DOS proved advantageous since UNIX offers a straightforward file structure and storage organization. Unfortunately it will be impossible to implement a UNIX compatible operating system in the controller since UNIX uses special files for I/O operations. This requires the necessary file organization information to be present at the host which is exactly what should be avoided by using an intelligent controller.

The aspects mentioned above should be considered seriously when deciding for an approach like the one described here.

### 6.2. Recommendations.

Some recommendations can be made to enhance the design for future follow-up of the project.

Firstly, the introduction of the Intel 80186 single chip computer during this project, might prove to be a serious alternative to the co-processor system of 8086 and 8089. Although the DMA controller imbedded in the 80186 is not as powerful as the IOP, it could prove to offer sufficient service for this application.

The suggestion to use programmable logic (FPGA's/PAL's) instead of a controller module as is done here, is repeated. Also, it would be wise to use controller modules that support a standard interface to there host e.g. the SASI interface bus.

Finally, it would be advantaguous to allow the controller direct access to the host's memory. Allthough the idea of buffering in the controller is not a bad idea in itself, the extra work of moving information from this buffer to the host could be avoided by granting the controller direct access to the host memory. A condition for this approach is the rewuiired compatibility between controller and host bus system. In particular a dual port memory system at the host would suit this application.

## Chapter 7. ACKNOWLEDGEMENTS.

### 7. Acknowledgements.

I would like to take this opportunity to express my gratitude to ir. M.P.J. Stevens and ir. J.P. Kemper who supervised this graduation project and whose technical support and advise was highly appreciated. Furthermore I would like to thank Prof. ir. A. Heetman under which authority this graduation project took place.

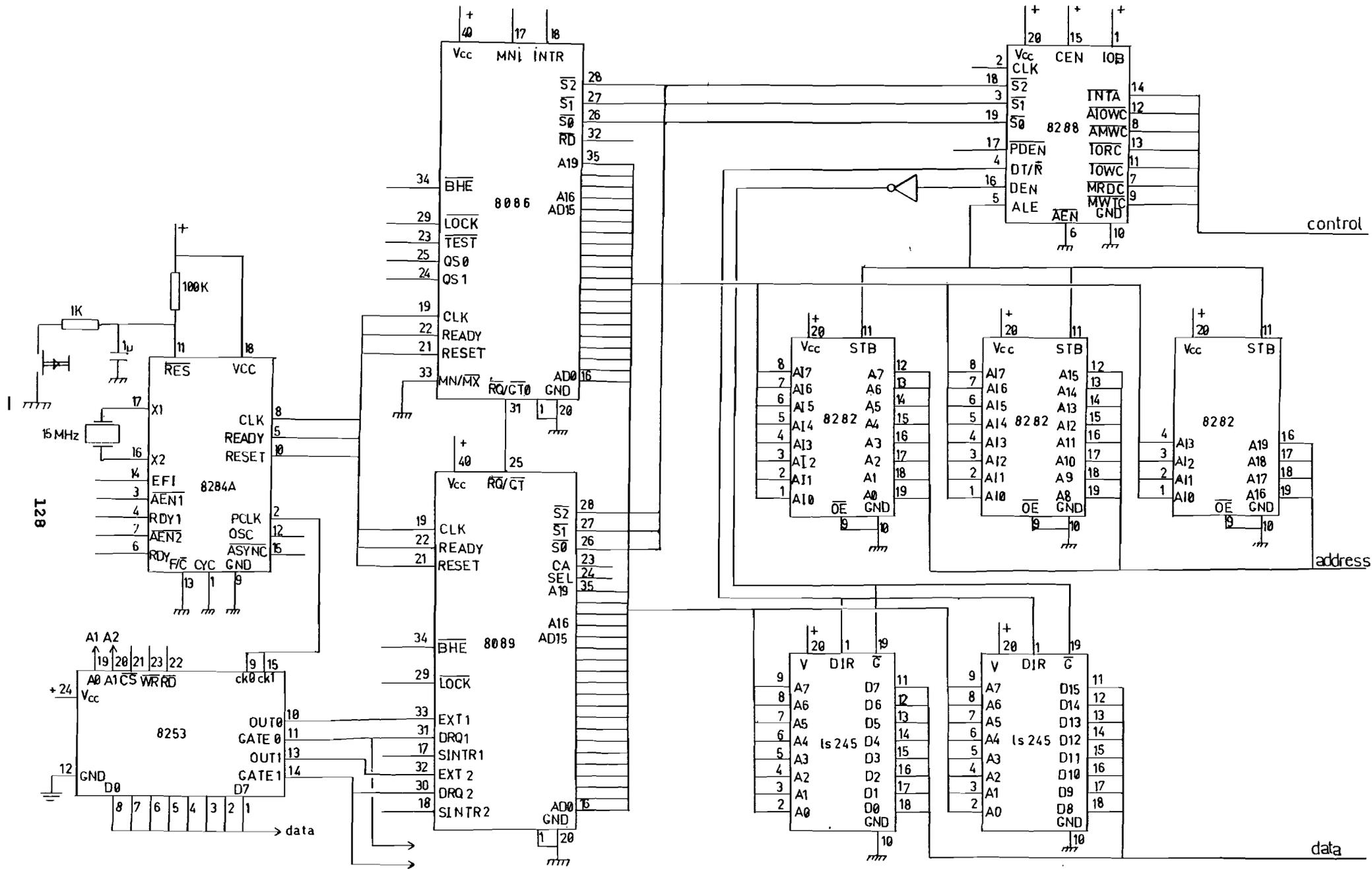
I greatly enjoyed the pleasant atmosphere in which I could work in the Digital Systems division of the Department of Electro-technical Engineering at the Eindhoven University of Technology. For this atmosphere I would like to thank all the staff members of the Digital Systems division as well as my fellow students.

In particular I would like to mention Ing L. van Bokhoven who showed great interest in the project and whose technical advise was highly appreciated.

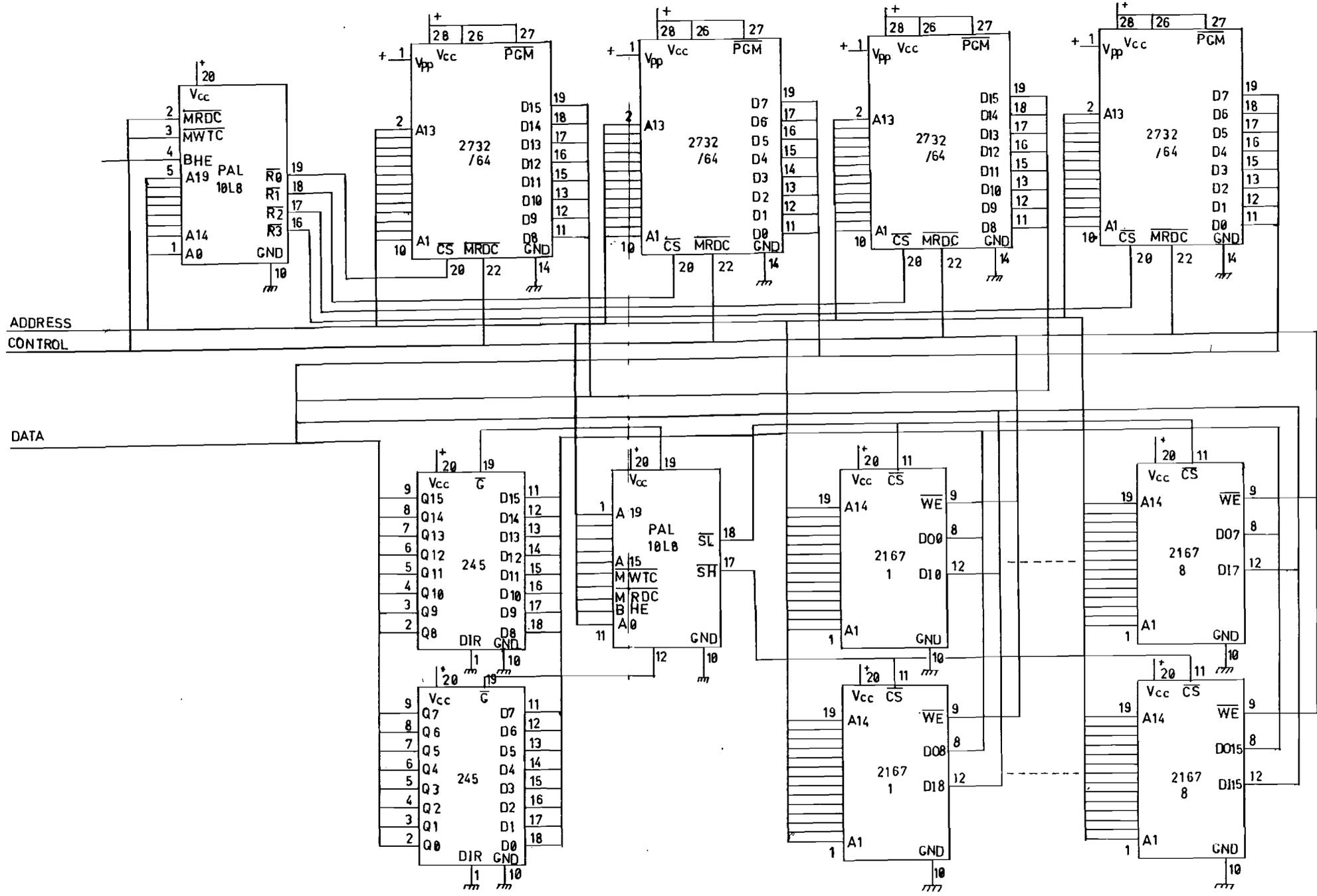
Finally I would like to mention Mr. T. Suters who, by his physical presence at the opposite side of my desk, proved to be a constant source of inspiration.

LITERATURE REFERENCE LIST.

1. Sumner, Don  
Compact controller can run any Winchester disk drive.  
Electronics, March 24, 1981, p 155.
2. Warren, Carl  
Disk drives.  
EDN vol 26, August 19, 1981, p 106.
3. Loring, Steve / Morgan, Bill.  
Floppy disk controller adapts to data format and drive.  
Electronics, August 28, 1980, p 171.
4. Neiman, Bruce.  
Winchesters set fast pace for controller designers.  
Electronic Design, May 10, 1980, p 179.
5. Teja, Edward R.  
8 inch hard disk storage.  
EDN, February 5, 1980, p 105.
6. Weller, Larry.  
Small Winchesters rolling in Volume.  
Electronics, February 10, 1981, p 97.
7. Hemenway, Jack.  
An 8 inch rigid disk interface improves micro-system capabilities.  
EDN, March 5, 1980, p 147.
8. Mahon, Douglas K.  
Micro-Winchester fits right into systems of the future.  
Electronic Design, March 31, 1981, p 129.
9. Scooros, Ted.  
Single board controller interfaces hard disk and back-up media.  
Electronics, May 19, 1981, p 160.
10. UNIX time sharing system.  
UNIX programmers manual.  
Seventh edition, Volume 2A/2B, January 1979.
11. Matick, Richard.  
Computer Storage Systems & Technology.  
John Wiley and Sons, New York 1977.
12. Meesters, A.J.C.  
Universele Diskcontroller.  
ECB 847, December 1981.

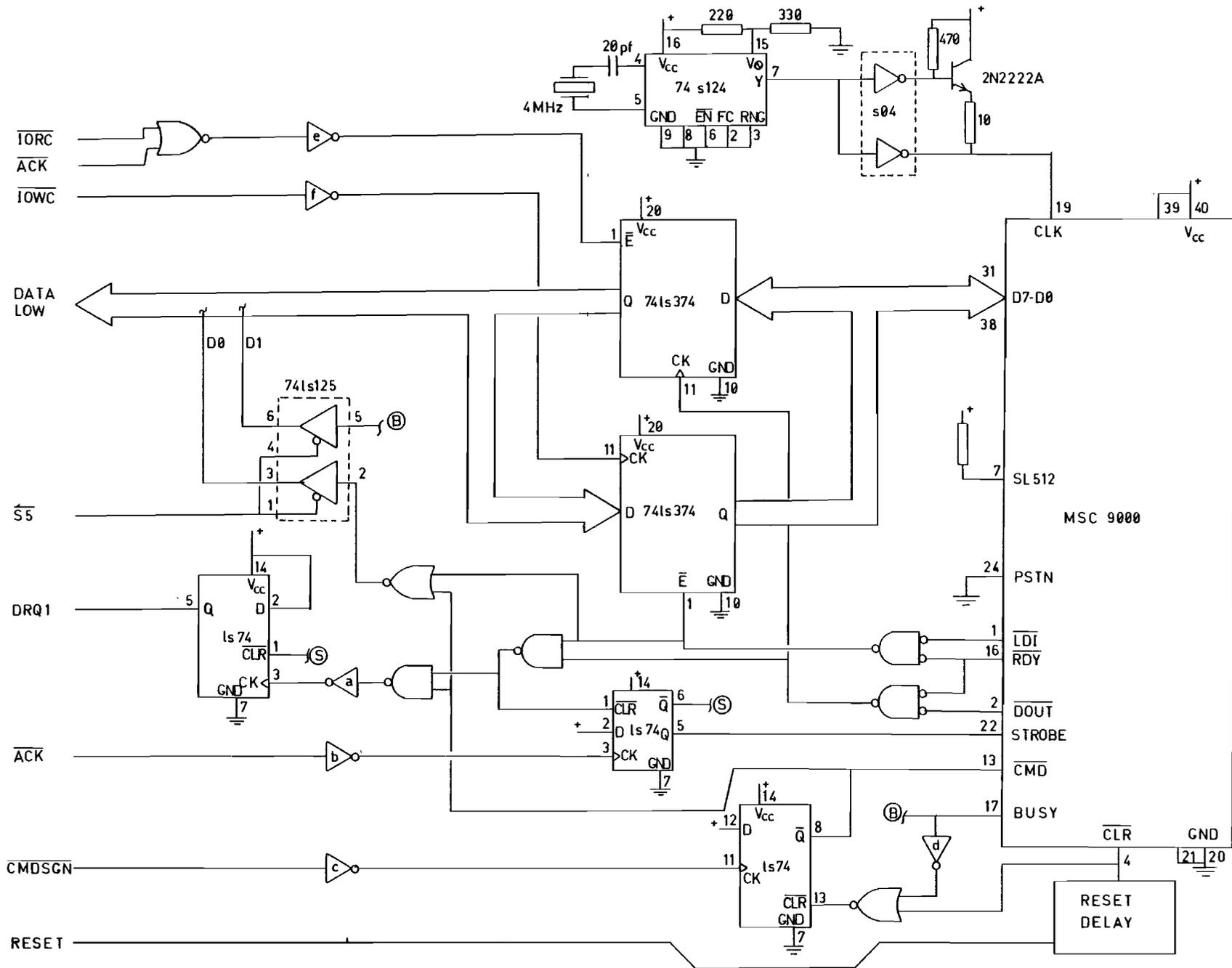


Appendix A. HARDWARE CIRCUIT DIAGRAMS.



129



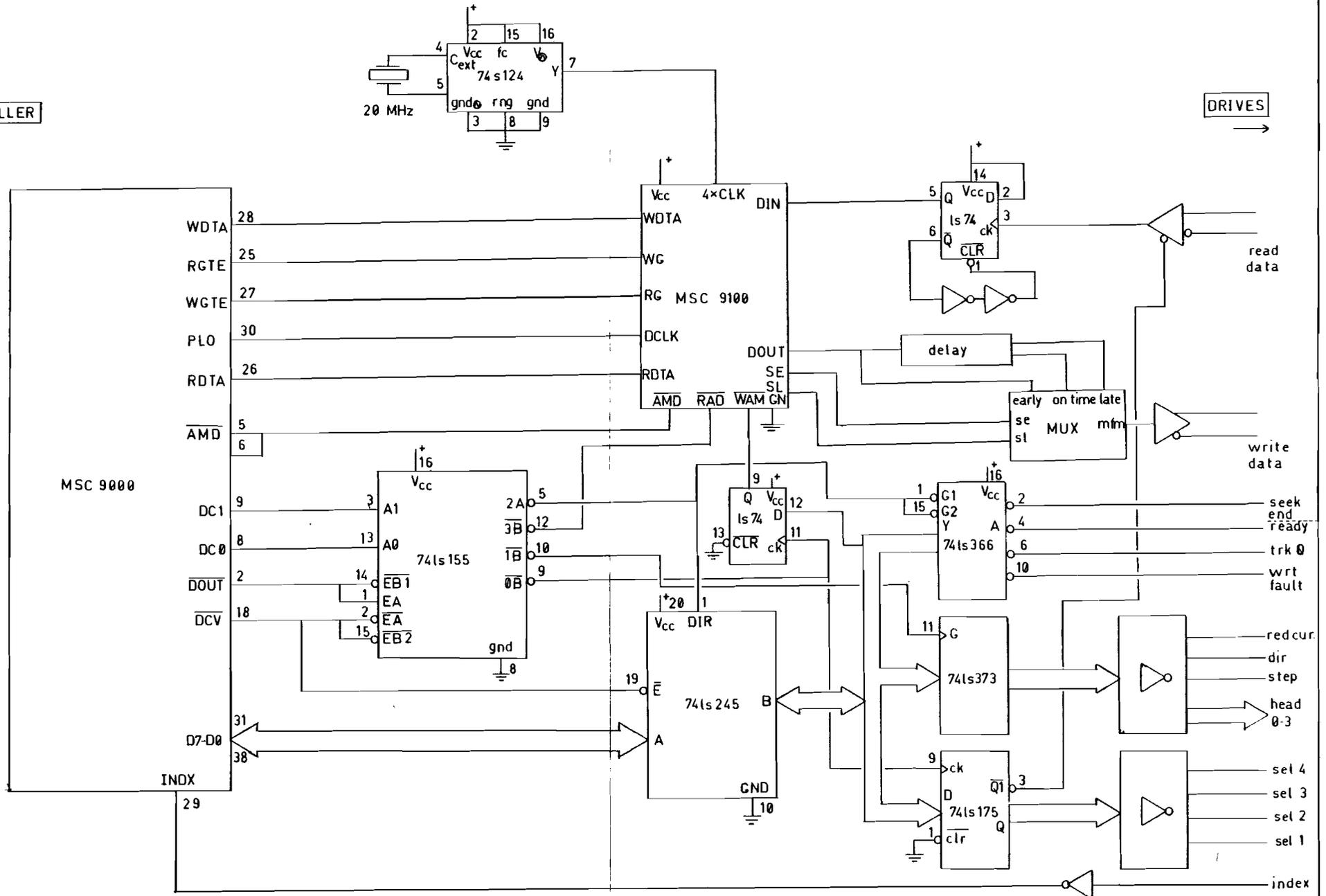


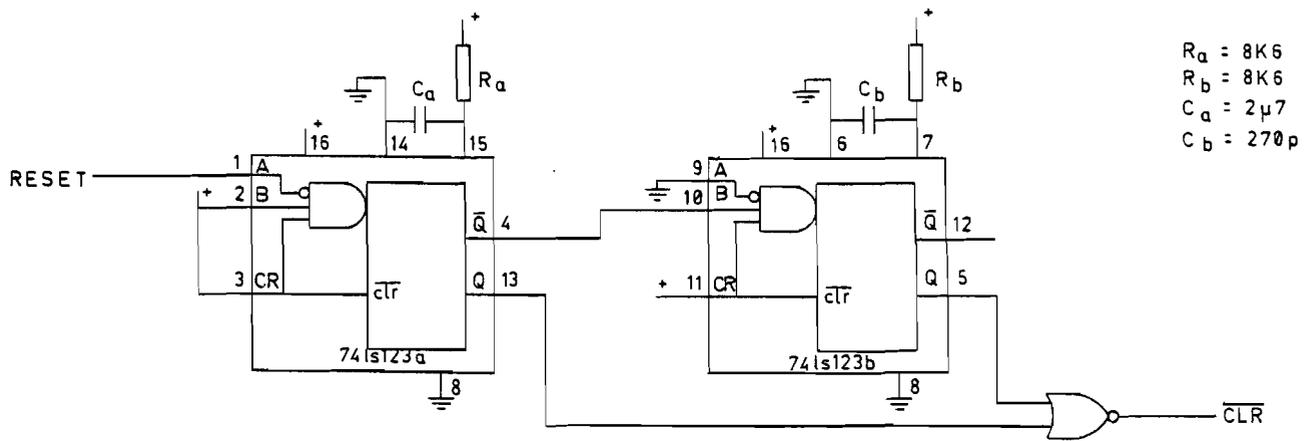
131

CONTROLLER ←

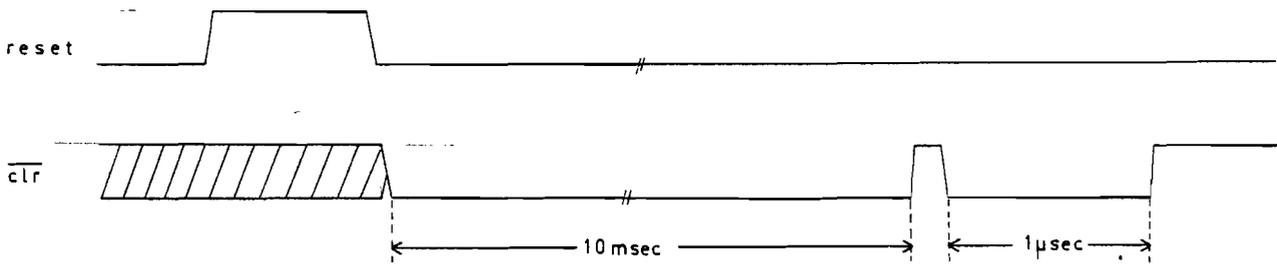
DRIVES →

132

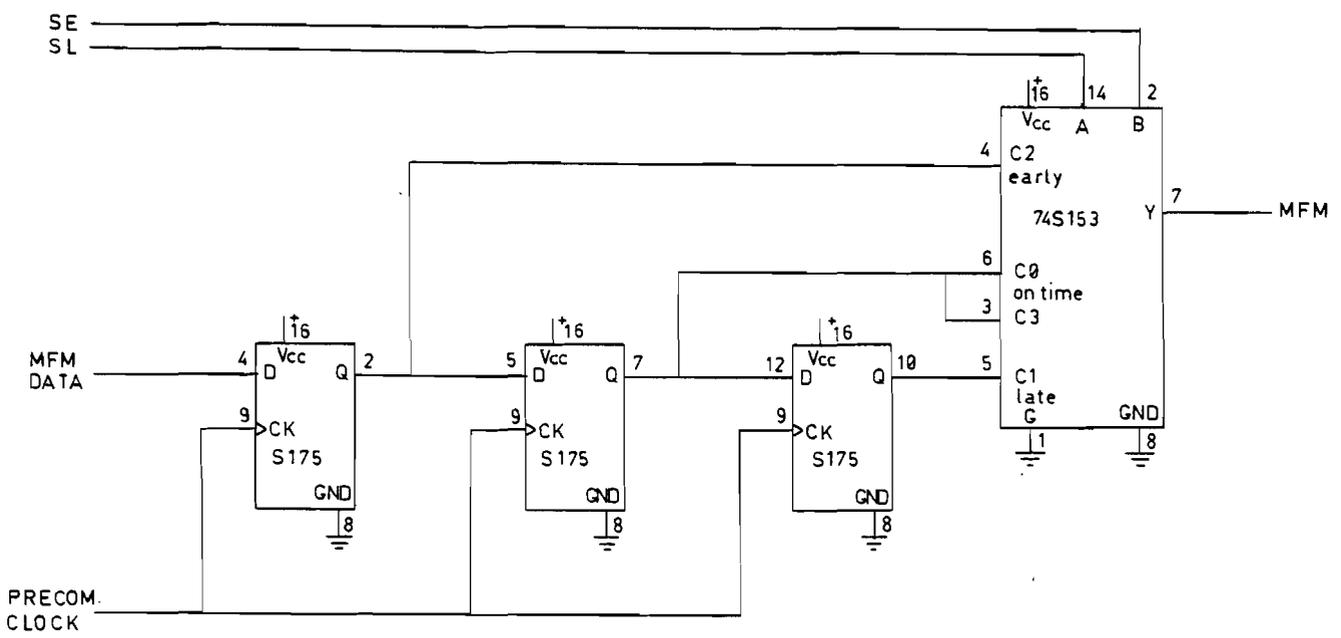




$R_a = 8K6$   
 $R_b = 8K6$   
 $C_a = 2\mu7$   
 $C_b = 270p$



MIDDELHOVEN UNIVERSITY TECHNOLOGY	A6 RESET DELAY CIRCUIT	H.v.Eijkelenburg 1982



MIDDELHOVEN UNIVERSITY TECHNOLOGY	A8 PRECOM. DELAY LINE	H.v.Eijkelenburg 1982

# MSC-9000 SERIES DISK ORIENTED I/O PROCESSORS PRODUCT SPECIFICATION

The MSC-9000 is a series of modules which incorporate 75% of the circuitry required to interface to small Winchester disk drives. There are two modules, each with identical features but with functional differences to accommodate either the Memorex 101, or Seagate Technology ST-506 disk drive. The functions incorporated within each module of the MSC-9000 series allow high level tasks to be communicated with it, achieving sophisticated control of the disk drive with minimum additional circuitry. Signals are provided allowing the easy implementation of simple interface circuits to control up to 4 disk drives. The MSC-9000 simplifies and handles most of the burdens associated with implementing a disk drive controller.

There are twelve separate commands which the Module will execute. Each of these commands requires multiple 8 bit bytes to fully specify the task.

Seek	Read Sector	Write Long
Recalibrate	Write Sector	Status Report
Diagnostic	Read Long	Format Track
Set Interleave	Write Alt. Sector	Write Check

## FEATURES

- Alternate sectoring for defect skipping.
- Variable interleave for data transfer rate tuning.
- Sophisticated error correction ensures data integrity, detecting up to 22 bit burst-errors and correcting up to 11 bit burst-errors.
- Full Sector Data buffer allows flexible data rates without affecting data transfer integrity.
- Automatic position verification to ensure data base integrity.
- Self-diagnostics assures easy maintainability.
- Proven technology for attaining high reliability.
- Compact Module ensures easy integration into stringent space restrictions.
- Low power consumption (4W typical).
- High level modularity ensures high maintainability.
- Universality assures easy integration for most system requirements.
- Supports up to 4 disk drives.
- Automatic Retries

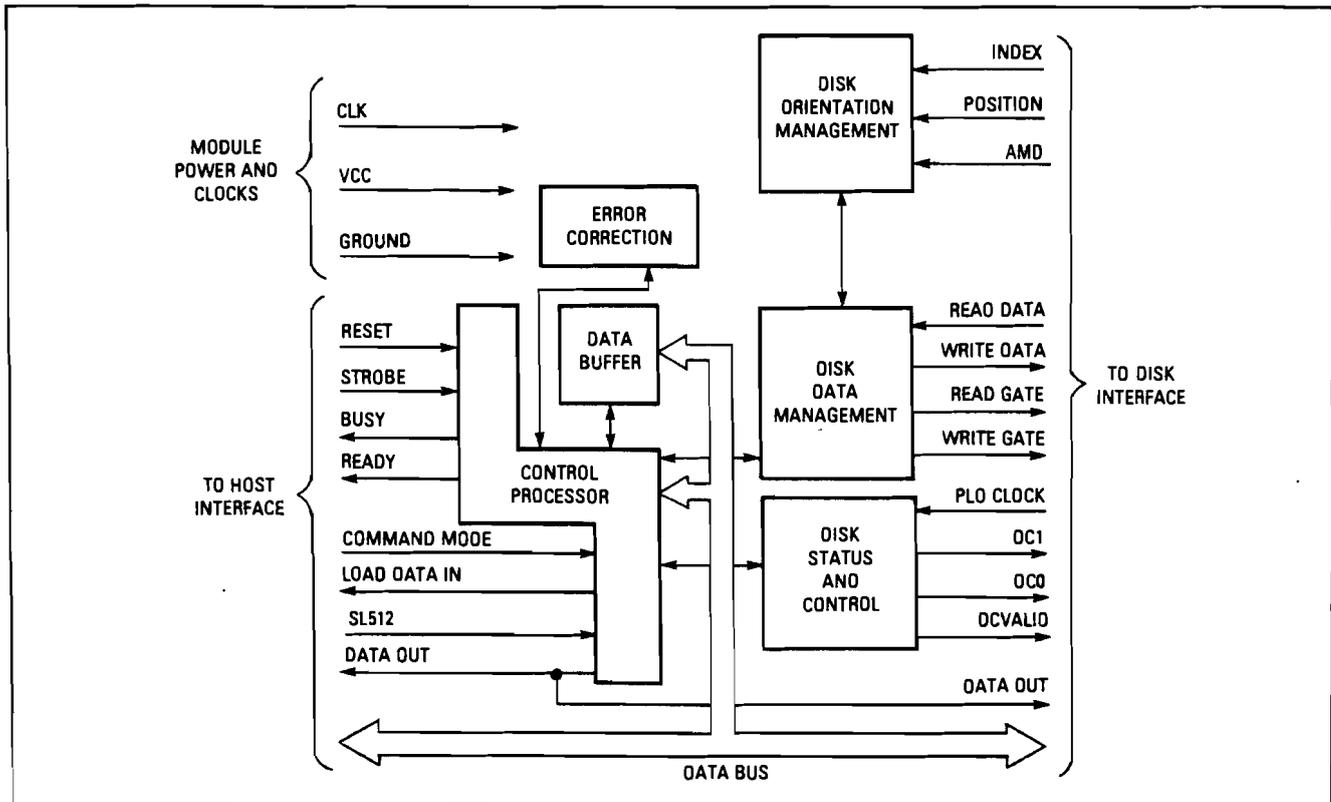


Figure 1

$\overline{\text{LDI}}$	○ 1	40 ○	VCC
DOOUT	○ 2	39 ○	VCC
UNUSED	○ 3	38 ○	DTA7
$\overline{\text{CLR}}$	○ 4	37 ○	DTA0
$\overline{\text{AMD}}$	○ 5	36 ○	DTA1
AMD	○ 6	35 ○	DTA6
SL512	○ 7	34 ○	DTA5
DC0	○ 8	33 ○	DTA2
DC1	○ 9	32 ○	DTA3
UNUSED	○ 10	31 ○	DTA4
UNUSED	○ 11	30 ○	PLO
UNUSED	○ 12	29 ○	INDX
$\overline{\text{CMD}}$	○ 13	28 ○	WDTA
UNUSED	○ 14	27 ○	WGTE
UNUSED	○ 15	26 ○	ROTA
RDY	○ 16	25 ○	RGTE
BSY	○ 17	24 ○	PSTN
DCV	○ 18	23 ○	UNUSED
CLK	○ 19	22 ○	STB
GND	○ 20	21 ○	GND

No	Symbol	Name	I/O	Function
1	$\overline{\text{LDI}}$	Load Data In	○	Active low output. When $\overline{\text{RDY}}$ is active this signal should be used to Gate Host data onto the data bus for input to the module.
2	DOOUT	Data Out	○	Active low output. Indicates the data bus contains valid data output from the module to be stored in the host interface when $\overline{\text{RDY}}$ is active, or the disk interface when $\overline{\text{DCV}}$ is active.
4	$\overline{\text{CLR}}$	Clear	↓	Active low input. Clears the Module.
*5/6	$\overline{\text{AMD}}$	AM DETECT	↓	Active low input. Address mark detect from data separator.
7	SL512	Select 512	—	512 Data Bytes per Sector when = 1, 256 when = 0.
8	DC0	Disk Control 0	○	Active high output disk interface control signals. These signals are encoded for which disk status or control information is on the data bus when $\overline{\text{DCV}}$ is active. See Table 1.
9	DC1	Disk Control 1	○	
13	$\overline{\text{CMD}}$	Command Mode	↓	Active low input. Requests the Module to enter command mode to enable giving it the command descriptor on the data bus. Command mode will only be entered when module is not busy ( $\text{BSY} = 0$ ). This signal can be resst anytime after the module accepts the first byte, it must be inactive before the module returns to Not Busy.
16	RDY	Ready	○	Active low output. Signifies when data can be transferred and Strobe can be activated.
17	BSY	Busy	○	Active high output. Signifies when module is processing (or acting on) command descriptor given to it. When low signifies idle and ready to accept command descriptor.
18	DCV	Disk Control Valid	○	Active low output. Indicates when the disk control signals DC0 and DC1 contain valid control signals to define the use of the Data Bus. See Table 1.
19	CLK	Clock	↓	4 MHz Clock input.
20, 21	GND	Ground	—	D.C. Power Ground
22	STB	Strobe	↓	Active high input. Strobes data in or out of module. May only be set when $\overline{\text{RDY}}$ is active.
**24	PSTN	Position	↓	Active high input. Pulse for defining the rotational position of the disk. Sector Pulse in MSC-9016 from Memorex 101.
25	RGTE	Read Gate	○	Active high output. Read Gate for disk drive.
26	RDTA	Read Data	↓	Active high input. NRZ serial read data from disk drive or data separator.
27	WGTE	Write Gate	○	Active high output. Write Gate for disk drive and data separator.
28	WDTA	Write Data	○	Active high output. Write Data for disk drive or MFM encoding circuit.
29	INDX	Index	↓	Active high input. Index pulse from disk drive.
30	PLO	PLO Clock	↓	Active high input. PLO clock from disk drive or data separator for Strobing read and write data.
31-38	DTAX	Data X		Active High Tri-State Bidirectional 8 bit Data Bus. NOTE: This bus should only be driven from outside the module when $\overline{\text{RDY}}$ and $\overline{\text{LDI}}$ or $\overline{\text{DCV}}$ are active. See Table 1.
39, 40	VCC	Power		+5 Volt D.C. Power.

\*Not used on MSC-9016 for MRX101—Pull up to VCC through 1K $\Omega$  Resistor for -9016.  
\*\*Not used on MSC-9056 for ST-506—Tie to ground for -9056.

**Note: Unused Pins Must Be Left Open.**

- CODE '0' SEEK**—The seek command moves the heads to the specified absolute cylinder. The disk must be formatted.
- '1' READ ONE SECTOR**—The read one sector command transfers one sector of data from the disk to the host system. During this command, the Module positions the heads (implied seek), verifies the ID field, transfers the sector data to the module, and checks and corrects data errors prior to transferring the data to the host.
- '2' WRITE ONE SECTOR**—The write one sector command transfers one sector of data from the host to the disk. During this command, the Module transfers one sector of data from the host to the internal buffer, positions the heads (implied seek), verifies the ID field and if proper, writes the data to the disk.
- '3' FORMAT ONE TRACK**—The format one track command initializes all ID and data fields for the specified track. The data field is initialized with a preset pattern of B6DB6D. The head parameter is used to select the head of the disk prior to formatting, the cylinder parameter is used to write the ID field.
- '4' RECALIBRATE**—The recalibrate command positions the heads at track 00 on the disk.
- '5' STATUS**—The status command sends one byte of status information to the host. The status represents the results of the previous task.
- '6' READ LONG**—The read long command transfers one sector plus the four ECC (Error Correction Code) Bytes from the disk to the host. During this command, the module positions the heads (implied seek), verifies the ID field, transfers the sector data to the module, and then transfers the sector plus ECC Bytes to the host. The module does not try to correct the data field if an ECC error occurs during the read. This command can be used to verify the ECC function of the module.
- CODE '7' WRITE LONG**—The write long command transfers one sector plus four ECC Bytes from the host to the disk. During this command, the module transfers one sector plus the ECC Bytes from the host to the internal buffer, positions the head (implied seek), verifies the ID field, and writes the data and ECC to the disk. This command can be used to verify the ECC function of the module. The appended ECC polynomial should be  $(X^{32} + X^{23} + X^{21} + X^{11} + X^2 + 1)$ .
- '8' WRITE ALTERNATE SECTOR**—This command is used to reassign a defective sector to the alternate location on the track. During this command the module formats the entire track and initializes all data fields with B6DB6D. The specified Sector is assigned as defective and the alternate sector is assigned as a replacement. Note that if the rest of the track already contains data it must be preserved by reading all sectors and re-writing all sectors after the alternate assignment (this command) is executed.
- '9' SET INTERLEAVE**—This command transfers a block of data into the Module which represent the interleave table of logical to physical assignments. After CLEAR or diagnostic command, the Module defaults to no interleave until this command is executed.
- 'A' WRITE CHECK**—The command is identical to the READ Sector command, without any data transferred. It can be used to verify the previously written data can be read without ECC errors.
- 'B' DIAGNOSTIC**—This command causes the Module to execute a self-test; checking its internal processors, data buffers, ECC circuitry, and Program memory. A STATUS command can then be performed which will give the results. If this command does not complete within 2 seconds the Module has a defect which prevents communication. This command will leave the module in a clear state.

---

NOTE: Four retries will automatically be invoked if an error is encountered during any of the following operations:

- Position Verification (Seeking)
- Target Sector Verification
- Hard ECC Error on Read Sector
- Write Alternate Sector

TABLE I—Data Bus Contents with  $\overline{DCV} = 0$

DC1, DC0

	00 Output Head and Control (DOUT = 0)	01 Output Drive Select (DOUT = 0)	10 Input Status	*11 Reset AM Detect (DOUT = 0)
<b>MSC-9016</b>				
D <sub>0</sub>	Head 0	Select 0	Track 00	N/A
D <sub>1</sub>	Head 1	Select 1	Write Fault	N/A
D <sub>2</sub>	Head 2	Select 2	<u>Seek End</u>	N/A
D <sub>3</sub>	Head 3	Select 3	<u>Drive Ready</u>	N/A
D <sub>4</sub>	Step			
D <sub>5</sub>	Direction Select			
D <sub>6</sub>	Fault Clear			
D <sub>7</sub>				
<b>MSC-9056</b>				
D <sub>0</sub>	Head 0	Select 0	Track 00	None
D <sub>1</sub>	Head 1	Select 1	Write Fault	None
D <sub>2</sub>	Head 2	Select 2	<u>Seek End</u>	None
D <sub>3</sub>	Head 3	Select 3	<u>Drive Ready</u>	None
D <sub>4</sub>	Step	Write Address Mark		None
D <sub>5</sub>	Direction Select			None
D <sub>6</sub>				None
D <sub>7</sub>	Reduce Current			None
*This function is not used in the MSC-9016				

All commands are sent to the Module using 8 Bytes of information over the Module Data Bus (BIT 0 = LSB) in the following format.

- Byte 1 Command Code
- Byte 2 Drive Select (one of four)
- Byte 3 Cylinder Address—Most Significant Byte
- Byte 4 Cylinder Address—Least Significant Byte (Maximum numbers of cylinders = 1024)
- Byte 5 Head Address
- Byte 6 Sector Address
- Byte 7 Sector Count Hi
- Byte 8 Sector Count Low

If a command does not require the full 8 Bytes, then Pad Bytes must be included to maintain the 8 Byte message length. After the task is sent to the Module, data will be transferred (in or out) when the task is executed.

The Task Bytes are represented as follows:

Byte	Task Bytes	D7	D6	D5	D4	D3	D2	D1	D0
1	Command Code	1	0	0	0	2 <sup>3</sup>	2 <sup>2</sup>	2 <sup>1</sup>	2 <sup>0</sup>
2	Drive Select (one of four)	0	0	0	0	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>
3	Cylinder Address (MSB)	C <sub>15</sub>	C <sub>14</sub>	C <sub>13</sub>	C <sub>12</sub>	C <sub>11</sub>	C <sub>10</sub>	C <sub>9</sub>	C <sub>8</sub>
4	Cylinder Address (LSB)	C <sub>7</sub>	C <sub>6</sub>	C <sub>5</sub>	C <sub>4</sub>	C <sub>3</sub>	C <sub>2</sub>	C <sub>1</sub>	C <sub>0</sub>
5	Head Address	0	0	0	0	H <sub>3</sub>	H <sub>2</sub>	H <sub>1</sub>	H <sub>0</sub>
6	Sector Address	S <sub>7</sub>	S <sub>6</sub>	S <sub>5</sub>	S <sub>4</sub>	S <sub>3</sub>	S <sub>2</sub>	S <sub>1</sub>	S <sub>0</sub>
7	Sector Count Hi	0	0	0	0	0	0	0	0
8	Sector Count Low	0	0	0	0	0	0	0	1

1 = Retries can be inhibited by setting bit 7 in command byte.

COMMAND	CMD CODE	DRV NBR	CYL MSB	CYL LSB	HD ADR	SCT ADR	SCT HI	SCT LOW	Data Bytes	
									Number of Bytes	In/Out of Module
SEEK	0	X	X	X	---	---	---	---	---	---
READ	1	X	X	X	X	X	X	X	N	OUT
WRITE	2	X	X	X	X	X	X	X	N	IN
FORMAT	3	X	X	X	X	---	X	X	---	---
RECALIBRATE	4	X	---	---	---	---	---	---	---	---
STATUS	5	---	---	---	---	---	---	---	1	OUT
READ LONG	6	X	X	X	X	X	X	X	N+4	OUT
WRITE LONG	7	X	X	X	X	X	X	X	N+4	IN
WRITE ALT.	8	X	X	X	X	X	---	---	---	---
SET INTERLEAVE	9	---	---	---	---	---	---	---	S	IN
WRITE CHECK	A	X	X	X	X	X	X	X	---	---
DIAGNOSTIC	B	---	---	---	---	---	---	---	---	---

(X) represents a task byte is required. (---) represents a Pad Byte is required.

(N) is the number of Data bytes per Sector which can be selected to be 256 or 512.

(S) is the logical to Physical interleave table with the number of bytes equal to the number of Sectors per track.

Byte	Interleave Table Contents
0	Physical location of logical Sector 0
1	Physical location of logical Sector 1
.	.
.	.
20	Physical location of logical Sector 20
.	.
.	.
37	Physical location of logical Sector 37

STATUS CODE TABLE	
CODE	MEANING
00	No Error
01	Invalid Command
02	Drive Not Ready
03	Seek Timeout (2 Seconds)
04	Invalid Track 00 indication from disk drive
05	All ID Fields Bad on Track
06	Target Sector Not Found
07	No Sector Found and ID ECC Error on Target Sector
08	Position Error (Seek Error)
09	Defective Module or Support Signals
0A	Drive Fault Active
0B	Index/Sector Timeout
0C	Command Parameter Error
0D	Uncorrectable ECC Error
1X	Correctable ECC Error
20	Write Alternate Error
21	Invalid Alternate Sector Assignment
22	Alternate Already Assigned
23	Direct Access to Alternate Sector
24	Defective processor
25	Defective Buffer Memory
26	Defective ECC Circuitry
27	Defective Program Memory
28	Illegal Sector Pulse during diagnostic
29	Illegal Interleave Table Parameter

NOTE: X is the length of the burst error which can be 1 to B to note if the correction span was 1 to 11 bits.

Sectors Per Track	Module Type	256 Bytes/ Sector	512 Bytes/ Sector
	MSC-9016	38	21
	MSC-9056	32	17

T<sub>a</sub> = 0°C to +50°C; V<sub>CC</sub> = +5V ±5% unless otherwise specified.

Parameter	Symbol	Min	Limits Typ(1)	Max	Unit	Test Conditions
Input Low Voltage	V <sub>IL</sub>	-0.5		0.8	Volts	
Input High Voltage	V <sub>IH</sub>	2.5		V <sub>CC</sub> + .5	Volts	
Output Low Voltage	V <sub>OL</sub>			0.45	Volts	I <sub>OL</sub> = Max
Output High Voltage	V <sub>OH</sub>	2.4		V <sub>CC</sub>	Volts	I <sub>OH</sub> = Max
V <sub>CC</sub> Supply Current	I <sub>CC</sub>		0.8	1.4	Amps	

**Note 1:** Typical values for T<sub>a</sub>=25°C and nominal V<sub>CC</sub>.

**OUTPUT CURRENT (MIN)**

Signal	High Level I <sub>OH</sub> (μA)	LOW Level I <sub>OL</sub> (mA)
LDI, DOUT	160μA	2.5mA
RDY	200	3.2
DCV	400	8.0
BUSY, WGTE, WDTA	400	2.8
RGTE	400	5.6
D <sub>0</sub> -D <sub>7</sub>	80	.90mA
DC0, DC1	140	1.36

**INPUT CURRENT (MAX)**

Signal	High Level I <sub>IH</sub> (μA)	Low Level I <sub>LIL</sub> (mA)
CMD, AMD, SL512	10	10(μA)
CLR	90	1.7
D <sub>0</sub> -D <sub>7</sub>	120	.88
PSTN, INDX	50	2.0
CLK	160	2.1
STB	80	1.5
PLO	60	1.2
RDTA	40	.8

T<sub>a</sub> = 25°C; V<sub>CC</sub> = 0V OUTPUT CAPACITANCE-50pf max

**Absolute Maximum Ratings\***

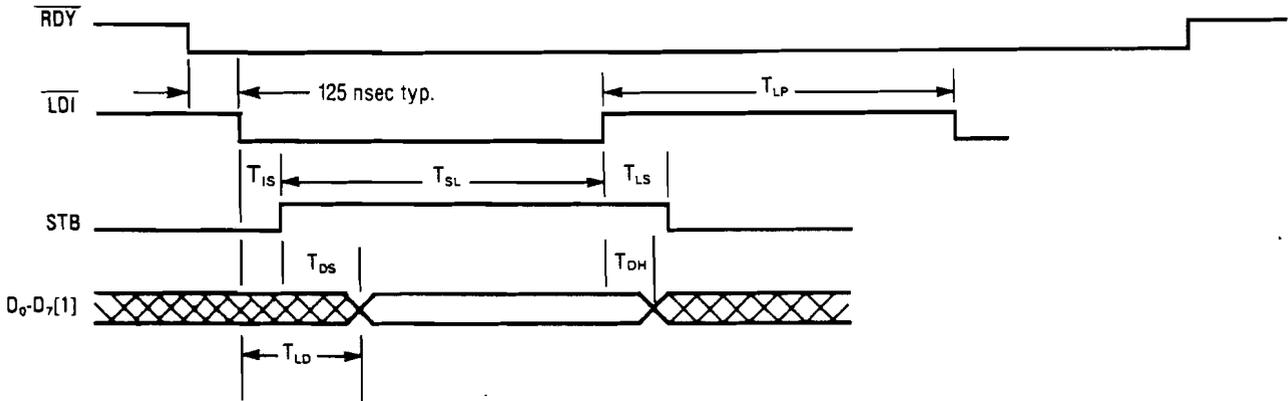
\*T<sub>a</sub> = 25°C

Operating Temperature	0°C to +50°C
Storage Temperature	-40°C to +125°C
All Input Voltages	-0.5 to +7 Volts
All Output Voltages	-0.5 to +7 Volts
Supply Voltage V <sub>CC</sub>	-0.5 to +7 Volts

**Comment:** Stress above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

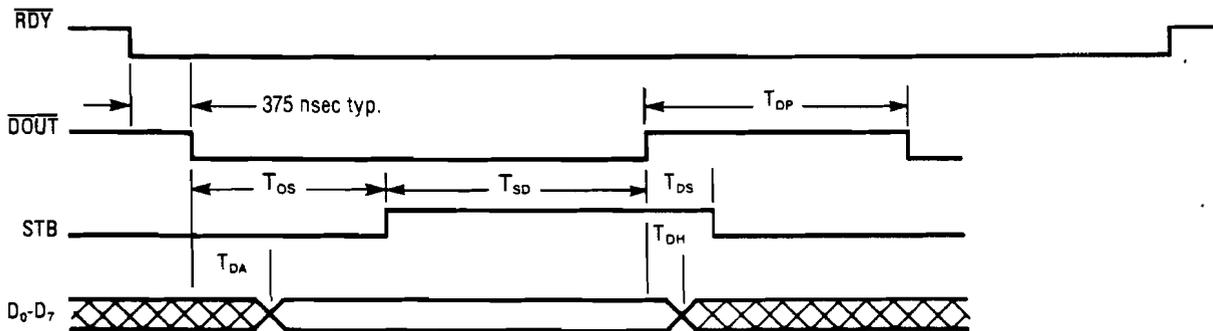
NOTE: Failure to observe conventional precautions during storage, handling, and usage, in order to avoid exposure to excessive voltages or static charges may affect device reliability.

Host Data Input To Module



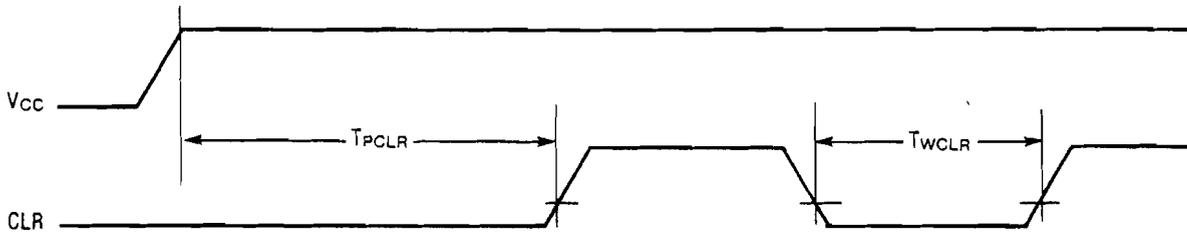
		MIN	TYP	MAX	UNITS
T <sub>LD</sub>	LDI LOW TO DATA ENABLE	0			NS
T <sub>DS</sub>	DATA SETUP FROM STROBE			100	NS
T <sub>DH</sub>	DATA HOLD FROM LDI HIGH	5		115	NS
T <sub>SL</sub>	STB HIGH TO LDI HIGH	580		1235	NS
T <sub>LS</sub>	LDI HIGH TO STB LOW	0		210	NS
T <sub>LP</sub>	LDI HIGH TO LDI LOW	248		748	NS
Max	DATA INPUT RATE (DISK WRITE)			957	KB/ sec.
T <sub>IS</sub>	LDI LOW TO STB HIGH	0			NS

Host Data Output From Module



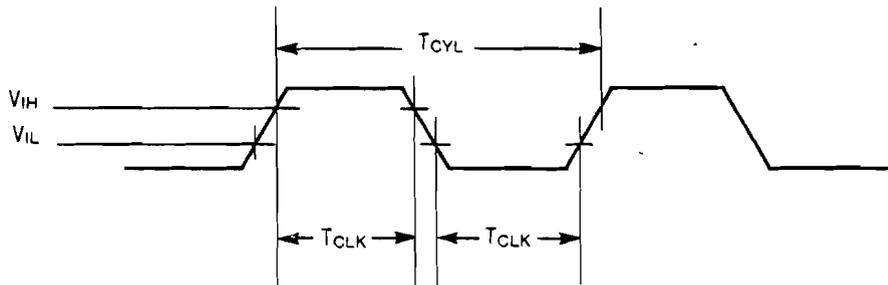
		MIN	TYP	MAX	UNITS
T <sub>DA</sub>	DATA VALID FROM DOUT LOW			75	NS
T <sub>OS</sub>	DOUT LOW TO STB HIGH	0			NS
T <sub>SD</sub>	STB HIGH TO DOUT HIGH	560		1175	NS
T <sub>DH</sub>	DATA VALID FROM DOUT HIGH	32			NS
T <sub>DS</sub>	DOUT HIGH TO STB LOW			270	NS
T <sub>DP</sub>	DOUT HIGH TO DOUT LOW	485		915	NS
Max	DATA OUTPUT RATE (DISK READ)			873	KB/ sec.

Reset Timing



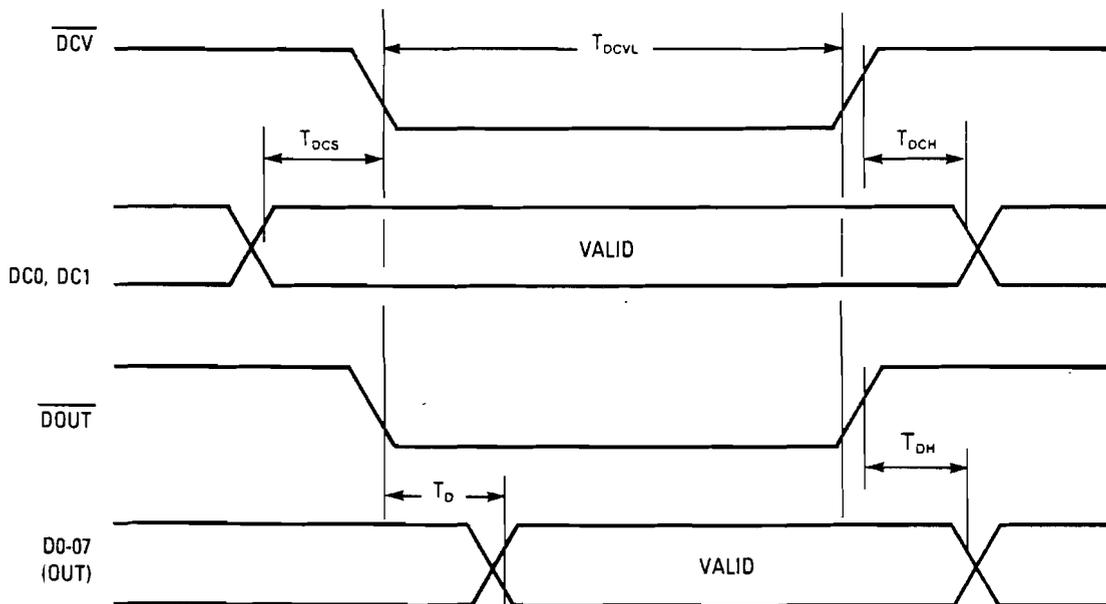
T<sub>PCLR</sub> = 10 MILLISEC (MIN)  
 T<sub>WCLR</sub> = 1 μSEC (MIN)

Clock Timing



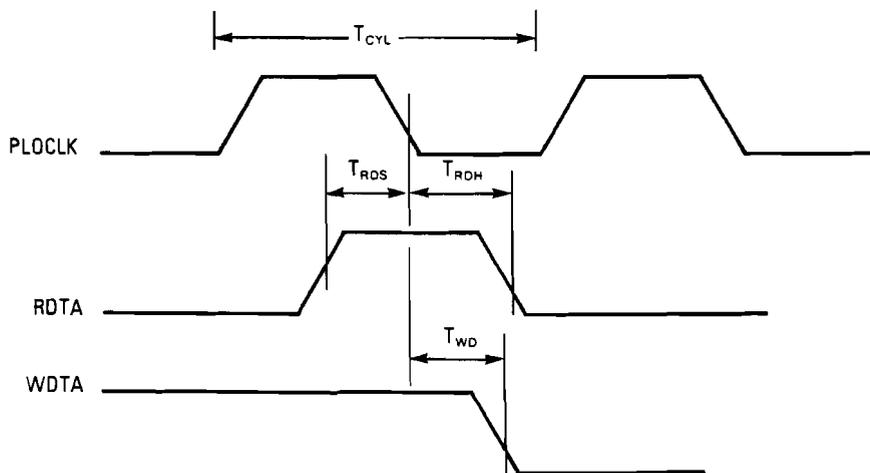
T<sub>CLK</sub> = 110 NSEC (MIN)  
 T<sub>CYL</sub> = 250 NSEC ± 1%  
 V<sub>IH</sub> = V<sub>CC</sub> - 0.8V  
 V<sub>IL</sub> = 0.8V

Disk Control Timing



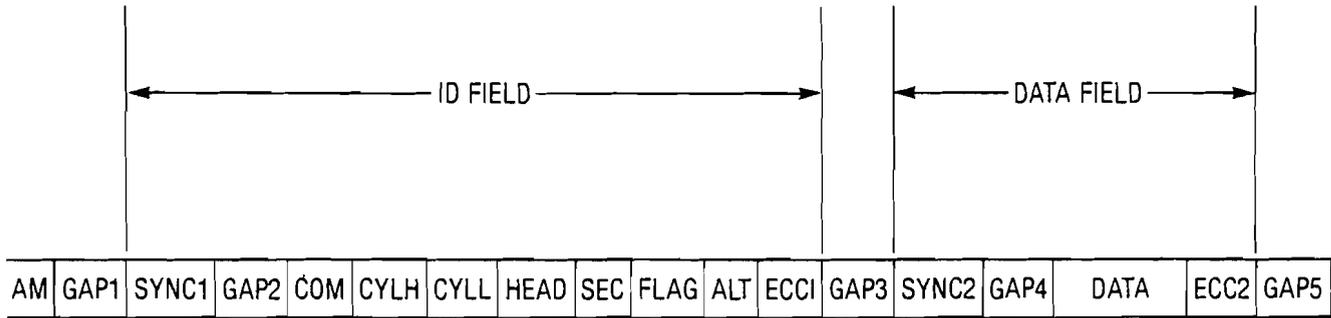
- $T_{DCVL} = 625 \text{ NS}$
- $T_{DCS} = 212 \text{ NS (MIN)}$
- $T_{DOCH} = 51 \text{ NS (MIN)}$
- $T_D = 30 \text{ NS (MAX)}$
- $T_{DH} = 52 \text{ NS (MIN)}$

Disk Data Timing



- $T_{CYL} = 192 \text{ NS (MIN) (50\% \pm 10\% DUTY CYCLE)}$
- $T_{RDS} = 20 \text{ NS (MIN)}$
- $T_{RDH} = 20 \text{ NS (MIN)}$
- $T_{WD} = 55 \text{ NS (MAX)}$

# MSC-9056 FORMAT

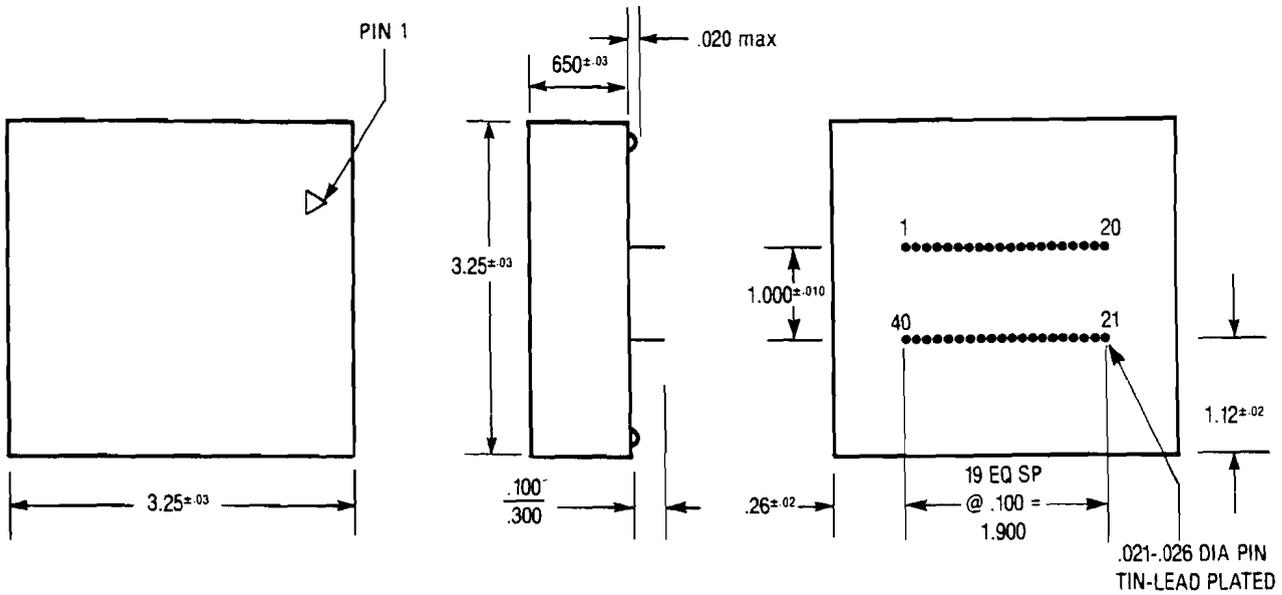


FIELD	FIELD DESCRIPTION	# OF BYTES
AM	ADDRESS MARK	4
GAP1	ZERO BYTE GAP	9
SYNC1	ID SYNC BYTE	1
GAP2	ID ZERO BYTE GAP	2
COM	ID COMPARE BYTE	1
CYLH	CYLINDER HIGH (MSB)	1
CYLL	CYLINDER LOW (LSB)	1
HEAD	HEAD #	1
SEC	SECTOR #	1
FLAG	FLAG BYTE	1
ALT	ALTERNATE SECTOR	1
ECC1	ID ECC BYTES	4
GAP3	ZERO BYTE GAP	16
SYNC2	DATA FIELD SYNC BYTE	1
GAP4	DATA FIELD ZERO BYTE GAP	2
DATA	DATA FIELD	256/512*
ECC2	DATA FIELD ECC BYTES	4
GAP5	INTER-RECORD ZERO GAP	14/43*

256 bytes = 32 sectors/track  
 512 bytes = 17 sectors/track

\*256 BYTE SECTOR/512 BYTE SECTOR

## Mechanical Dimensions



## RECOMMENDED MOUNTING METHODS

- 1.) Wave or Hand Soldered w/ .042 Min. Hole Dia.
- 2.) Sockets w/ Mechanical Mounting to PCB.

Sockets: Augat #8134-HC-6P2

8134-HC-6P3

Cambion #450-3703-01

450-3983-01

## ORDERING INFORMATION

Model	Disk Drives Accommodated
MSC-9016	Memorex-101, Fujitsu 2301
MSC-9056	Seagate Technology ST-506

## Appendix C. WD CONTROLLER MODULES OVERVIEW.

### C.1 Controller hardware.

The layer between the Winchester and the software drivers (see figure 1.1) is formed by the controller hardware. The controller hardware is responsible for executing functions involving data-conversion, AM detection and status and control handling.

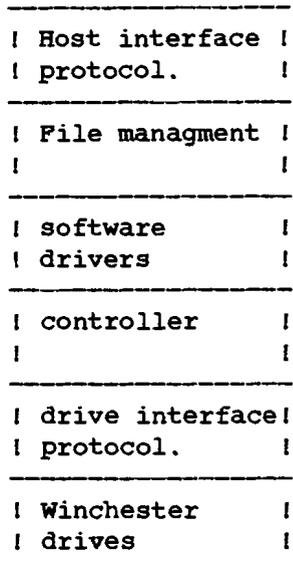


Figure C.1.

Two approaches exist when designing the controller hardware. The first and hitherto most commonly used approach is the use of conventional TTL logic, usually in fairly large quantities. Such a design can be customized to meet all the requirements of the user. The price for this is a large number of chips, high power consumption and little flexibility.

In order to overcome these problems, several manufacturers have come up with functional modules, capable of performing many of the functions required for drive control. Using such modules lowers the amount of external hardware and simplifies controller design. Since this is a fairly recent development, scarce information is available on these modules. Nevertheless an attempt is made to describe the functional features of some of them.

## C.2. MSC 9000.

The MSC 9000, manufactured by Microcomputer Systems Corporation, incorporates 75% of the circuitry required to interface to small Winchester disk drives. Two models exist, each with identical features but with functional differences to accommodate either the Memorex 101 or Seagate Technology 506 disk drive. The functions incorporated within the modules allow high level tasks to be communicated with it, achieving sophisticated control of the disk drive with minimum additional circuitry. Up to four drives can be serviced by one module. Refer also to appendix B.

The following lists all the commands the module is capable to execute.

- Seek Track.
- Recalibrate.
- Diagnostic.
- Set Interleave.
- Read Sector.
- Write Sector.
- Read long.
- Write Alternate Sector.
- Write long.
- Status Report.
- Format Track.
- Write Check.

Commands are transferred in a 8 byte format as such:

- Byte 1: Command Code.
- Byte 2: Drive Select.
- Byte 3: Cylinder Address.
- Byte 4: Cylinder Address.
- Byte 5: Head Address.
- Byte 6: Sector Address.
- Byte 7: Sector Count Hi.
- Byte 8: Sector Count Lo.

Commands that don't require the full 8 bytes like Status report, use pad bytes to maintain the 8 byte length. Sector size is software selectable to 256 or 512 bytes/sector. Error bursts of up to 11 bits can be corrected and 22 bit burst errors will be detected allowing high data integrity.

Data transfers.

The module transfers data in blocks that equal one sector on the disk. Both programmed I/O and DMA transfer techniques may be used. A built-in sector buffer compensates for speed differences between controller and host. Maximum data-rates using DMA are set to 957 Kbytes/sec for input to the module, 873 Kbytes for output.

Unfortunately the data-bus from the module is only 8 bits wide.

Unfortunately the data-bus from the module is only 8 bits wide. Using a 8089 I/O processor would limit the transfer speed to 625 Kbytes/sec since the 8089 requires eight clock cycles to perform a memory to port or port to memory transfer. With a 5 Mhz clock this requires  $8 \times 200 \text{ nsec} = 1.6 \text{ microseconds}$ . A 16 bit wide data-bus would permit maximum transfer rates to be achieved.

#### MSC 9100.

When controlling low-cost drives like the ST506, external circuitry for data-separation and AM detection is required, as mentioned before. The MSC 9100 was developed to be used in conjunction with the MSC 9000 to perform precisely these functions. It comprises a clock generator, NRZ to MFM converter, address mark detector/generator, phase locked loop and MFM to NRZ converter. Control signals for precompensation are also available from this module.

#### Interface.

The floppy extension interface is used for interfacing between drives and controller. A universal bus is available for host communication.

#### C.3 Western Digital 1010.

The Western Digital WD 1010 is a 40 pin package, single chip Winchester controller, capable of controlling an interesting class of low-cost 5.25 and 8 inch Winchester drives. The chip contains MFM encoder/decoder, address mark detector, high speed shiftregisters, write precompensation and write splice avoidance logic. External sector buffering, phase locked MFM read clock and high frequency detection must be added externally.

The command level is not as high as is the case with the MSC 9000. A brief survey is given in the table beneath.

-Restore	: automatic seek to track 000.
-Seek	: seek specific track.
-Read sector	: perform a sector read.
-Write sector	: perform a sector write.
-Scan ID	: scans track for ID field.
-Write format	: writes all necessary address marks and data marks on a track.

Data transfers to and from the disk are made at a 5 Mbit/sec speed which is compatible with most low-cost 5.25 inch drives. The use of an external FIFO sector buffer enables DMA transfers at any desirable speed. (limited by the FIFO and or DMA logic, not by the controller chip.) As with the MSC 9000, the extended floppy interface is supported while the interface to the host is universal.

#### C.4. National Semiconductor chip set.

Only very recently, National Semiconductor Corporation announced a chipset, capable of supporting a large number of different Winchester drives. Depending on the sophistication of the drive to be controlled, one or more of the chips are needed. The four chips in the set are:

- Pulse detector, converting analog pulse signals into digital pulses.
- Data separator, generating a read clock and decoding standard MFM data.
- MFM data encoder.
- Formatter, serializer, deserializer, doing all the read/write formatting as well as serial-to-parallel and vice versa data conversion in combination with error detection and correction.

Combining these chips as required allows the construction of a very flexible controller, supporting most of the existing controller-to-drive interface standards.

A very useful feature of the formatter chip is its possession of a 16 bit wide databus to the host computer, thus allowing a 16 bit microprocessor to interface to the controller. Using a 8089 I/O-processor, 1.25 Mbyte/sec can be transferred between the host and the controller under DMA. The controller itself supports transfer rates to and from the drive at a speed of up to 30 Mbit/sec.

#### C.5. Summary.

Based on the little information that is available about the forementioned components, one can say that it is very cost effective to integrate them in any modern Winchester controller design. It may be considered too soon but judging from the above description, one might consider the National Chipset as one of the better choices. Unfortunately, no technical information is available on it yet.

## Appendix D. CONTROLLER MODULE INTERFACE.

### D.1. Introduction.

The interface hardware between the MSC 9000 controller module and the processor system was described in chapter 4 of this report. From the circuit diagram shown in Appendix A, one can see that this traditional way of connecting a peripheral to a processor system results in a circuit consisting of several Flip-Flops and logic gates. Furthermore, the design of a circuit like the one shown, is rather heuristic, starting out from timing diagrams which show the required sequence of events. In this Appendix, an alternative is described using a Field programmable Logic Sequencer (FPLS) to obtain a "clean", straight-forward design method, resulting in a functional circuit with fewer chips.

### D.2. Interfacing to a peripheral.

Figure D.1. below shows a generally applicable blockdiagram of an interface between a peripheral and a processor or processor system.

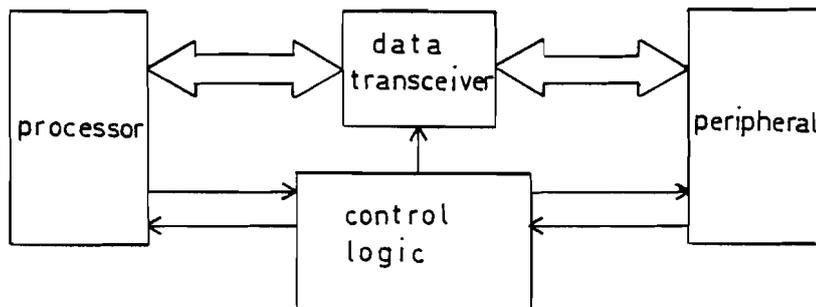


Figure D.1: Processor peripheral interface.

Traditionally, the block designated by "control logic" is built with random logic circuitry. Depending on the complexity of the interface, various chips are required, often not fully used. The object of this FPLS approach, is to implement this control logic block in one FPLS. This would result in a circuit diagram which is almost identical to the block diagram in figure D.1.

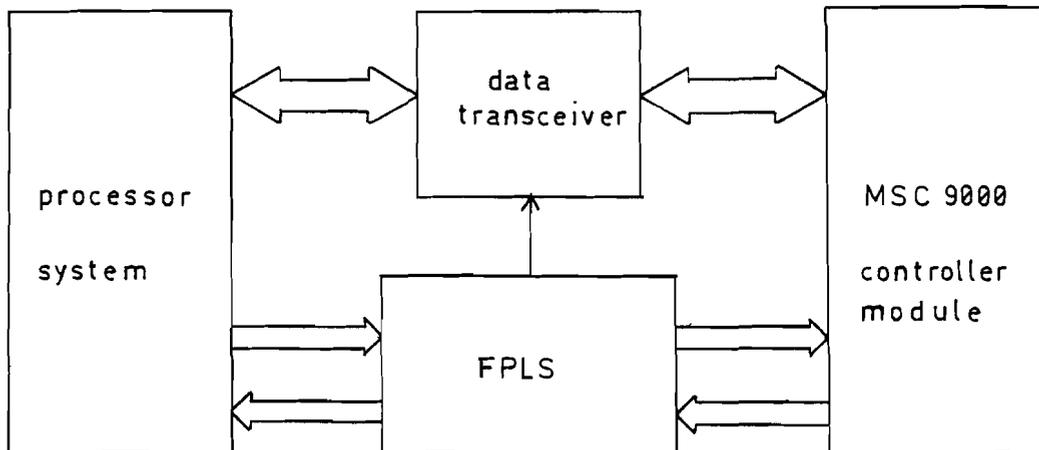


Figure D.2: FPLS based interface.

One important feature of a programmable logic control section should be stressed. A FPLS is a synchronous device and thus requires a clock signal to control its operation. Replacing a pulsed mode, asynchronous interface circuitry like the one shown in Appendix A will therefore have to be clocked at a rate high enough to avoid delay's compared to the asynchronous solution. In this particular case where controller module and processor system each have different clocks, the clock frequency of the FPLS will have to exceed both of them. Since both controller module and processor system only monitor their interface signals at a minimal interval determined by their clock periods, the maximal delay possibly introduced by a FPLS clocked at a higher frequency would be equal to the clock period of either the processor clock or the module clock.

Finally, a asynchronous circuitry has a delay caused by the accumulated propagation delay times of the gates invloved which is usually in the order of 60 nsec. If the FPLS were to be clocked at a frequency of 16 MHz, the same result would be obtained.

The PLO (phase locked oscillator) output, present on the 8284A clock generator of the processor system can be used conveniently for this purpose. Its frequency of 15 MHz will result in state transitions of the FPLS at 66 nsec intervals leading to an average delay of 33 nsec.

### D.3. Field Programmable Logic.

Field programmable logic devices were designed to avoid the need of random logic often required to interface between advanced, highly complex functional modules. These non-trivial ran-

dom logic configurations rely heavily on small and medium scale integrated logic circuits, whose fixed functions lead to relatively high chip counts.

Signetics corporation offers a range of field programmable logic chips, ranging in complexity from FPLG's (Field Programmable Logic Gate), FPLA's (Field Programmable Logic Array) to FPLS (Field Programmable Logic Sequencer). The table below outlines this product line along with some of its features.

FIELD-PROGRAMMABLE LOGIC FAMILY									
Device	Organization	Device	Inputs	Outputs <sup>(1)</sup>	Chip enable (CE) <sup>(3)</sup>	I <sub>CC</sub> (max)	Delay (max)	Availability	Package <sup>(2)</sup>
FPGA	• AND/NAND	82S102	16	9 OC	yes	170 mA	35 ns	now	N
		82S103		9 TS			50 ns	now	N, F
FPLA	• AND-OR/NOR	82S100		8 TS	no		70 ns	4Q79	N
		82S101		8 OC					
FPLS	• AND-OR • Complement array • 6-bit state register • 8-bit output register	82S106		8 OC	yes <sup>(3)</sup>		90 ns	4Q79	N, F
		82S107		8 TS					
		82S104	8 OC						
		82S105	8 TS						

(1) OC = open collector      TS = three-state  
(2) N = plastic              F = Cardip  
(3) CE input may be optionally programmed as preset.

Table D.1.

These gate arrays provide a powerful alternative to random logic, enabling systematic design, yielding power-, cost- and PC board area savings.

FPLS description.

For the problem at hand we will focus on FPLS's, the most powerful member of the gate array family. Basically, a FPLS is a state machine, performing synchronously clocked logic sequences. State machines are known in two forms;

- Moore Machines, in which the output is a function of the present state only.
- Mealy Machines, in which the output is a function of both the present state and the present input.

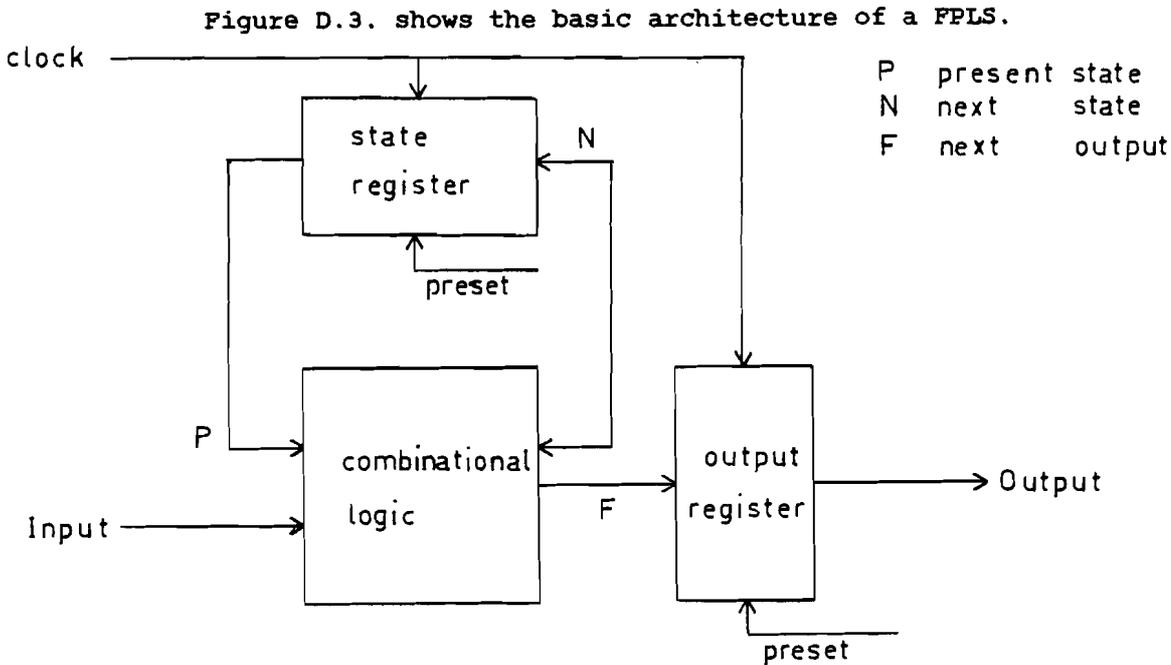


Figure D.3: MEALY machine.

With a FPLS, any logic sequence that can be expressed as a series of transitions between states, triggered by a valid input condition at clock time T can be programmed. The number of states involved depends on the complexity of the process involved and is limited by the width of the FPLS state register. In the case of a 82S105 FPLS, the state register is 6 bits wide so 64 different states can be defined. Expansion of the state register by assigning part of the output register for this purpose is possible at the cost of loosing input and output lines. The maximum number of transitions is limited by the amount of AND gates in the AND array of the FPLS, in this case 48.

Figure D.4. shows a typical state diagram. The state from which the transition occurs is called the present state P, that at which it terminates is referred to as next state N. A transition always causes a change in state but not necessarily a change in the machines output F.

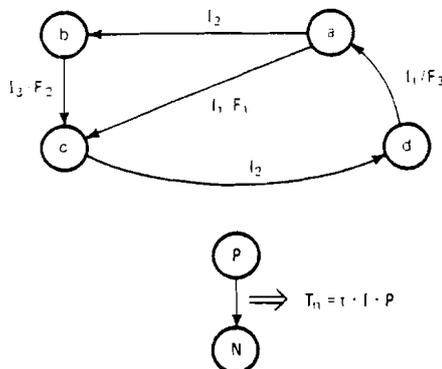


Figure D.4: State diagram.

All states are arbitrarily assigned and stored in the state register which has the next state information from the combinational network as its inputs. State transitions can only occur when the AND function of clock, present state and input condition is true. Figure D.5. shows the architecture of a FPLS.

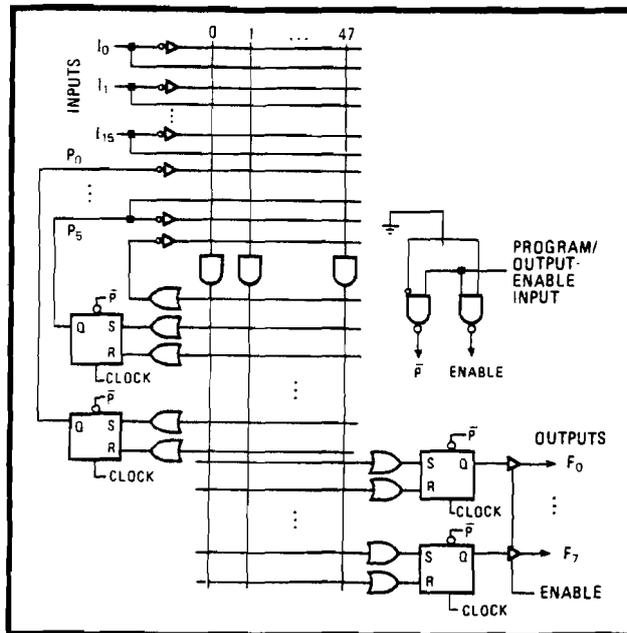


Figure D.5: FPLS architecture.

#### FPLS programming.

Programming of a FPLS can be done by means of a logic programming equipment, similar to a PROM programmer. The input connections necessary to implement the desired logic function are coded directly from the state diagram using a programming table as shown in figure D.7. In this table, the logic state or action of control variables C,I,P,N and F associated with each transition term  $T_n$ , is assigned a symbol which results in the proper fusing pattern of the corresponding link pairs. In the next section, an example of this state table will be shown for the interface function at hand.

#### D.4. Design.

Designing the interface between controller module and processor system using a FPLS, is now basically reduced to programming the FPLS in the correct way. In order to do this, a state diagram is drawn, reflecting the different states in which the interface logic can be, as well as the input conditions that cause a state transition and the output status of the FPLS resulting from the transition. Refer to figure D.6.

In order to be able to draw this diagram, two things are required:

1. Flow chart or Time diagram, showing the protocol of the communication.
2. List of input and output signals involved.

Time diagrams are obtained from the data-sheets supplied by the manufacturer of the peripheral, in this case, refer to Appendix B. From these time diagrams, the correct sequence of events can be deduced.

Below, a list of the signal lines involved along with a block diagram of the interface set-up is given.

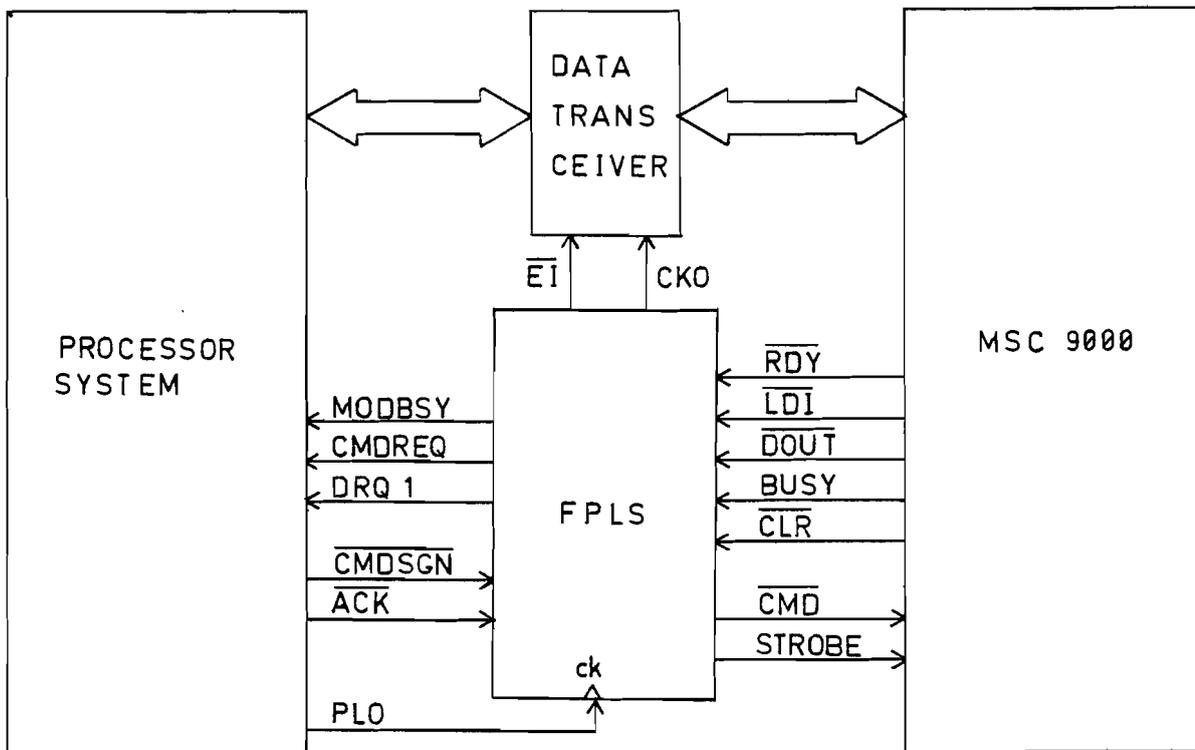


Figure D.8: Signal lines.

FPLS input signals.

CMDSGN: Issues a command cycle start request to the controller.  
ACK : Acknowledge data transfer.  
RDY : Controller ready to accept or sent data.  
LDI : Data input request.  
DOUT : Data output request.  
BUSY : Controller busy.  
CLR : Reset interface.

FPLS output signals.

MODBSY: Module busy indication.  
CMDREQ: Command byte request to processor system.  
DRQ1 : DMA request signal.  
EI : Enable input latch.  
CKO : Clock output latch.  
CMD : Signal command cycle start to controller module.  
STROBE: Data transfer strobe signal.

The input- and output vectors for the state diagram shown in figure D.6. are as follows:

I:	RDY	LDI	DOUT	BUSY	CMDSGN	ACK	CLR
	I6	I5	I4	I3	I2	I1	I0
O:	CMDREQ	MODBSY	DRQ1	CMD	STROBE	EI	CKO
	F6	F5	F4	F3	F2	F1	F0

The format used at the state transition arrows is:

I6	I5	I4	I3	I2	I1	I0	/	F6	F5	F4	F3	F2	F1	F0
INPUT CONDITION								OUTPUT RESULT						

Apparently, the total number of states involved is 13, thus only four bits of the state register are needed. Coding of these states can be done arbitrarily so the binary equivalent of the state numbers shown in figure D.6 was taken. Figure D.7. shows the corresponding programming sheet.

Once the FPLS has been programmed, the hardware of the interface between the MSC 9000 controller module and the processor system will look as is shown by Figure D.9. The difference with the corresponding circuit diagram of Appendix A is obvious.

preset/power-on

0  
initial state

T0: xxxxxx0/0001010

1  
module idle

T1: 1110011/0000010

2  
cmd cycle start

T2: 0010111/1000010

3  
module ready for cmd byte

4  
cmd byte accepted

T3: 0010101/1000100

6  
data i/o mode

T4: 0110111/0000010

5  
next cmd xfr or data xfr

T5: 1111111/0101010

T6: 0011111/0111000

7  
wait for data

T16: 0101111/0111011

10  
data out

T10: 0101101/0101111

T15: 0101111/0111011

12  
next data or terminate

T11: 0111111/0101010

156

9  
next data or terminate

T14: 0011111/0111000

T7: 0011101/0101100

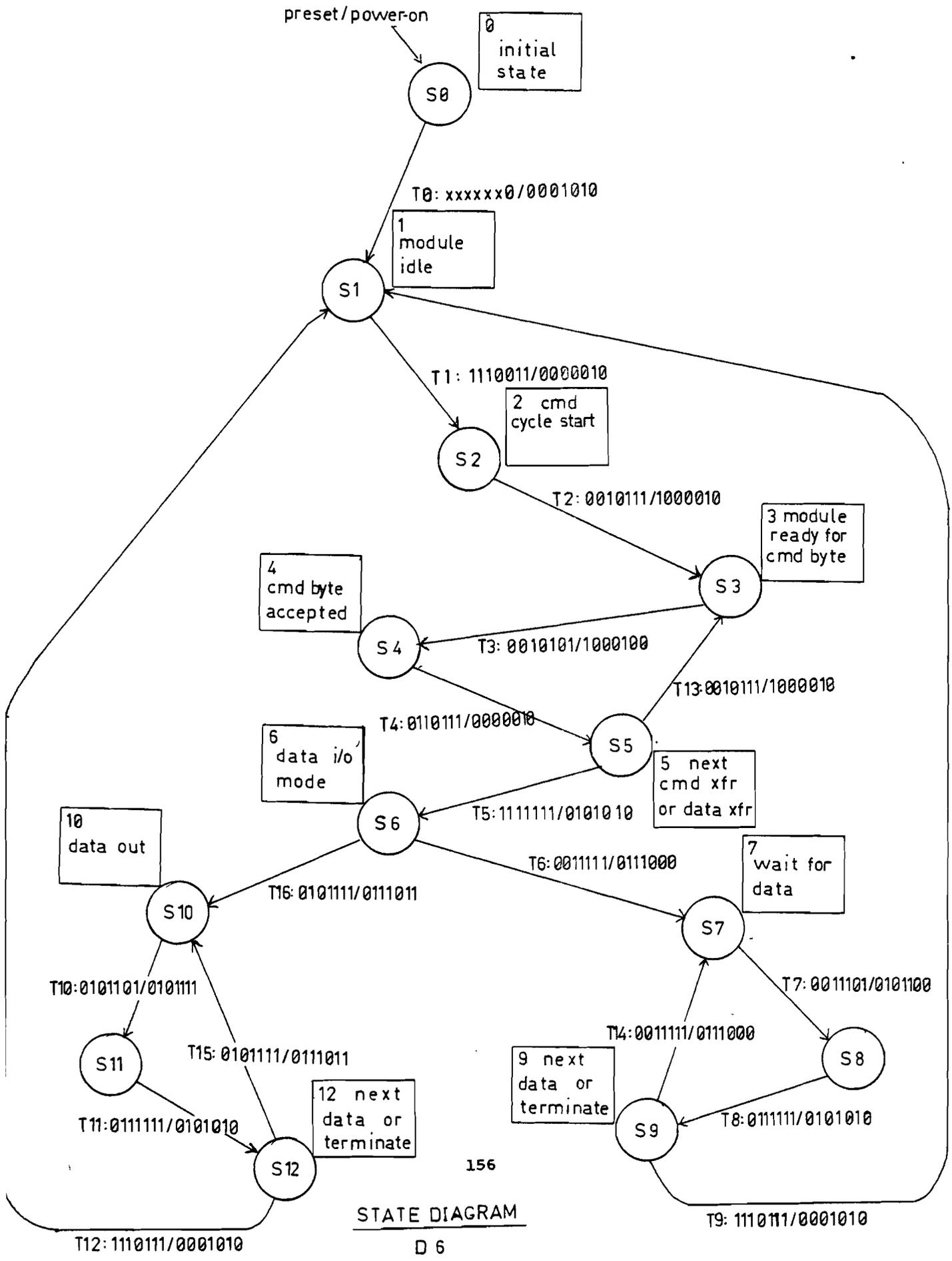
T8: 0111111/0101010

T9: 1110111/0001010

T12: 1110111/0001010

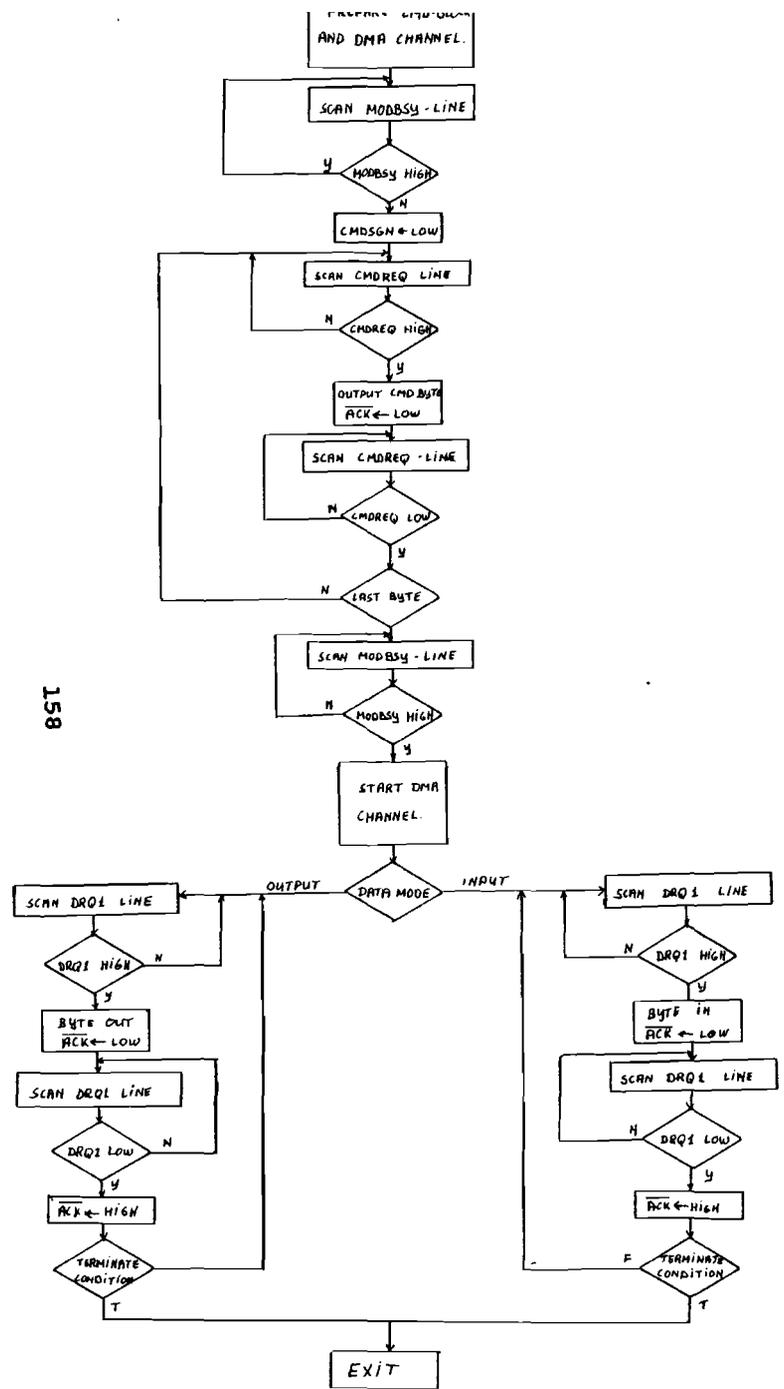
STATE DIAGRAM

D 6



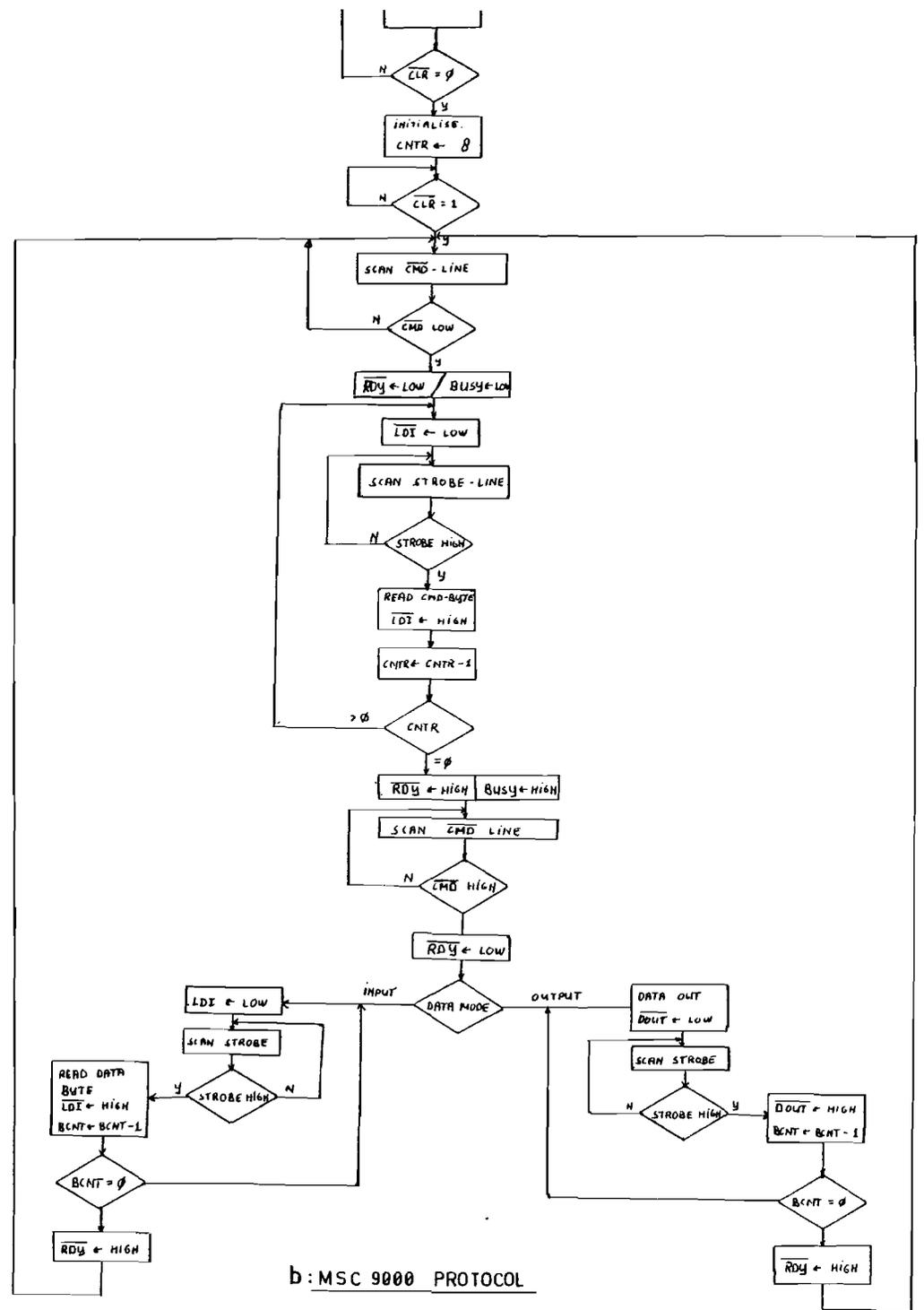


158

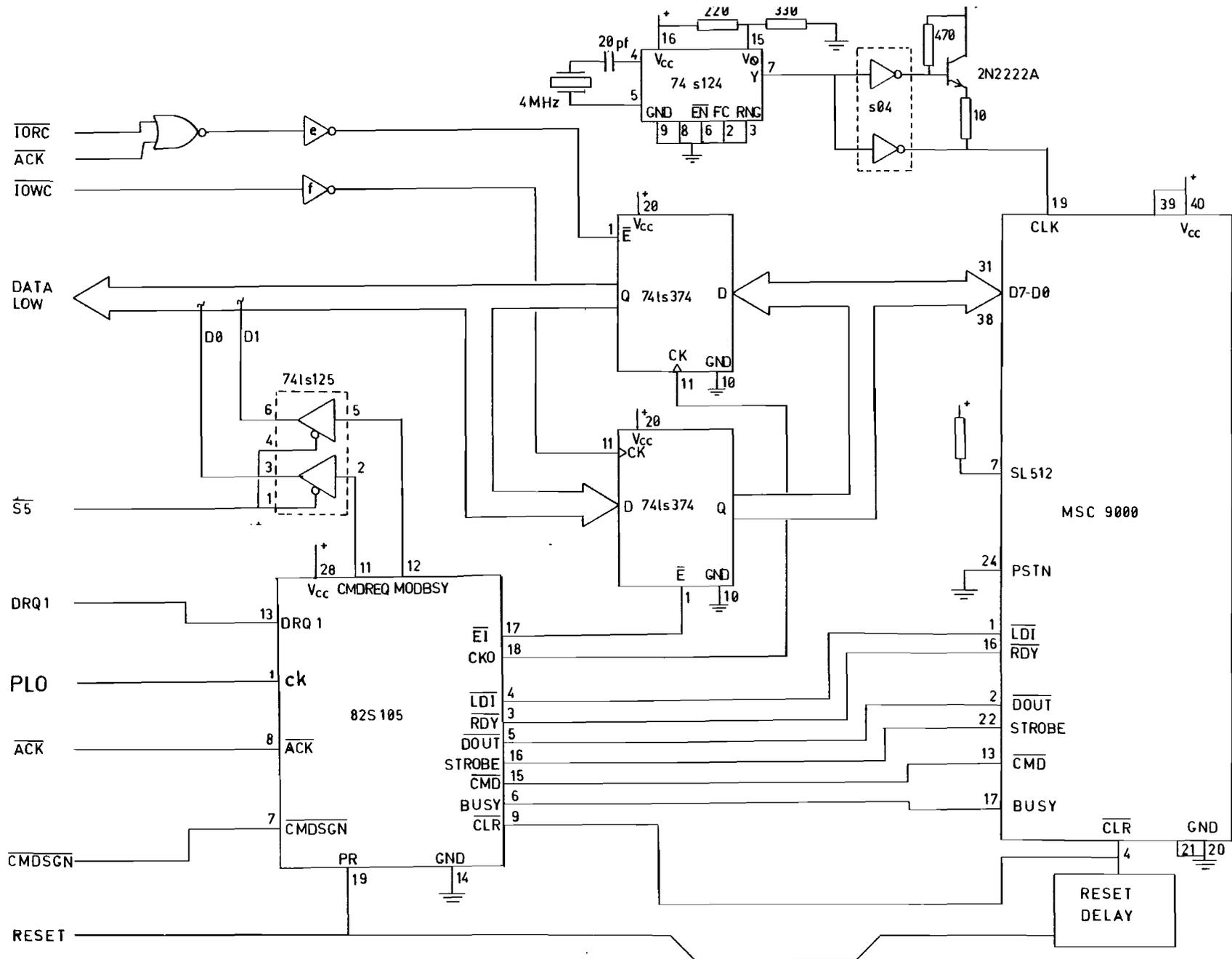


a: PROCESSOR PROTOCOL

D8



b: MSC 9000 PROTOCOL



159