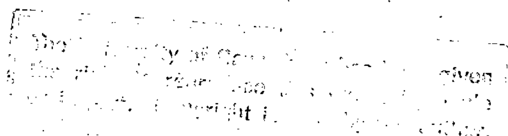
A HIGH LEVEL DISC CONTROLLER

by

Barry Feyder

Submitted to the University of Cape Town in fulfilment of the requirements for the degree of Master of Science in Applied Science.

September 1979.



The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

University of Cape Town

ACKNOWLEDGEMENTS

My supervisor, Professor M.G. Rodd, for his help and encouragement.

The South African Council for Scientific and Industrial Research for their financial assistance.

To my family, colleagues and friends for their support and encouragement.

ABSTRACT

Since the emergence of the digital computer in the late 1940's, computer architecture has been largely dictated by the requirements of mathematicians and scientists. Trends have thus been towards processing data as quickly and as accurately as possible. Even now, in the age of large scale integration culminating in the microprocessor, internal structures remain committed to these ideals. This is not surprising since the main users of computers are involved with data processing and scientific computing.

The process control engineer, who turned to the digital computer to provide the support he required in his ever increasing strive towards automation, has had therefore to use these generalized computing structures. His basic requirements however, are somewhat different to those of the data processing manager or the scientific user. He has to contend with an inherent problem of synchoronizing the computer to the real-world timing of his plants. He is far more interested in the response time of the computer to an external occurrence than he is to sheer 'number-crunching' power.

Despite the trends in process control towards distributed computing, even the most advanced systems require a relatively large central processor. This processor is called upon to carry out a wide variety of different tasks most of which are 'requested' by external events. Multiprogramming facilities are therefore essential and are normally effected by means of a real-time operating system. One of the prime objectives of such a real-time operating system is to permit the various programs to be run at the required time on some priority basis. In many cases these routines can be large - thus requiring access to backing storage.

Traditionally the backing store, implemented by a moving-head disc for example, is under the control of the real-time operating system. This can have serious consequences. If real-time requirements are to be met, transfer to and from the disc must be made as rapidly as possible. Also, in initiating and controlling such transfer, the computer is using time

which otherwise could be available for useful, process-orientated work.

With the rapid advancement of digital technology, the time is clearly right to examine our present computer architecture. This dissertation explores the problem area previously discussed - the control over the bulk storage device in a real-time process-control computer system. It is proposed that a possible solution lies in the development of an intelligent backing-store controller. This essentially combines the conventional low-level backing store interface with a special purpose processor which handles all file routines. This dissertation demonstrates how such a structure can be implemented using current technology, and will evaluate its inherent advantages.

INDEX

Page No.

CHAPTER 1 : INTRODUCTION 1

1.1 Organisation of Tasks in a Process-Control Computer System 1
1.2 Organizational Strategies 6
1.3 The Proposed Approach 12
1.4 Hard-Soft Trade 13
1.5 Hardware Implementation of the Filing System . . . 14
1.6 An Alternate Disc Controller 15
1.7 Summary 18

CHAPTER 2 : REQUIREMENTS OF THE HIGH LEVEL DISC CONTROLLER 20

2.1 Essential Filing System Requirements 21
2.2 File System Structure 22
2.3 Implementation of the Filing System in the High Level Disc Controller 23
2.4 Filing System Response 30
2.5 Reliability of Data on Disc 33
2.6 Free Time 35
2.7 Requirements of the Disc Drive 35
2.8 Conclusion 36

CHAPTER 3 : THE HARDWARE STRUCTURE OF THE HIGH LEVEL DISC CONTROLLER 38

3.1 The Essential Hardware Requirements of the High Level Disc Controller 38
3.2 The Central Processing Unit 39
3.3 Architecture for the High Level Disc Controller . . 42
3.4 The Microinstruction Format 44
3.5 The Central Processor Array Field of the Microinstruction 48
3.6 The Next Address Selection Field of the Microinstruction 52

Page No.

3.7 The Control of External Logic 60
3.8 Microinstruction Cycle Time 63
3.9 System Timing 67
3.10 Local Memory 68
3.11 The Computer Interface 69
3.12 Disc Drive Interface 74
3.13 The Console 80

CHAPTER 4 : MEASURING THE HIGH LEVEL DISC CONTROLLERS PERFORMANCE 83

4.1 Measurement of the Experimental Task 86
4.2 Implementation of the Experimental Task 88
4.3 Execution of the Experimental Task 95
4.4 The Varian Interfaced to the High Level Disc Controller 95
4.5 The Varian Interfaced to the Low-Level Disc Controller 96
4.6 Effects of the Assumptions Made 97
4.7 Summary 98

CHAPTER 5 : EVALUATION AND CONCLUSIONS 104

5.1 Evaluation of the High Level Disc Controller . . . 104
5.2 Summary 109

APPENDIX A : A REVIEW OF FILE HANDLING TECHNIQUES A-1

A.1 File Directories A-1
A.2 Storage of Data on Disc A-6
A.3 Improving Data Reliability on Disc A-9

APPENDIX B : THE MICROPROGRAM ASSEMBLER B-1

B.1 Introduction B-1
B.2 Microinstruction Format B-1
B.3 Operation B-5

Page No.

APPENDIX C : HARDWARE C-1

C.1 Comparison between the Intel 3000 and Advanced Micro Devices 2900 Central Processing Elements . . . C-1

C.2 The Central Processor Array - Functional Description C-2

C.3 The MIXED field C-8

C.4 Microinstruction Fetch Propogation Delay C-9

C.5 Microinstruction Execution Propogation Delay C-11

C.6 Port Timing C-11

C.7 Local Memory Timing C-13

APPENDIX D : INPUT-CONTROL AND OUTPUT-CONTROL SIGNALS...... D-1

D.1 The Central Processor Array Control Signals D-2

D.2 The Console Control Signals D-3

D.3 Local Memory Control Signals D-4

D.4 HLDC/Disc Drive Interface Control Signals D-5

D.5 The HLDC/Varian Interface D-17

APPENDIX E : THE HIGH LEVEL DISC CONTROLLERS CIRCUIT DIAGRAMS E-1

APPENDIX F : MEASUREMENT OF THE EXPERIMENTAL TASK F-1

NOMENCLATURE

(i) Unless otherwise indicated, all numbers are decimal. Octal numbers are of the form n_8, where n is an octal number.

(ii) k = kilo-word = 2^{10} = 1024 words.

(iii) One word contains 16 binary digits (bits) of data unless otherwise specified.

(iv) TRUE = HIGH = + 5 Volts; FALSE = LOW = 0 Volts.

CHAPTER 1

INTRODUCTION

Computers have had an impact on almost every sphere of man's environment. One particular area where the computer is being used in an increasing extent is the industrial sector. This environment extends from simple data logging functions to the complex environment of process control. The demands made on the computer in this environment are increasing. Increased demands mean that the computer capabilities must increase. The solutions to these increased demands are many, and include distributed processing, multi-processing and more sophisticated hardware and software.

In the process control environment, there are many different functions that the computer must perform. Many of these functions may have to be performed in accordance to the "real-world time" rather than the computer's own time. The system must be able to cope with a large number of different tasks. Tasks may be initiated by a real-time clock, by other tasks currently in execution, by a process-operator, or by signals from the process being controlled. Many of these tasks will be time-critical, and will have to be performed within a certain time period. Multiprogramming is necessary in such a system to allow tasks to be active simultaneously. A currently active task can therefore be suspended to allow for the execution of some other task. The suspension of a task may arise through various reasons. A task would be suspended if it required the use of a peripheral which was currently being used. The processor would execute some other task until the peripheral became free. A task would be suspended if a task of higher priority is initiated. Execution of the suspended task would only begin again after all higher priority tasks had been executed. It is evident that a management system is necessary to control the organization and execution of these various tasks.

1.1 Organisation of Tasks in a Process-Control Computer System

Rodd[1] indicates three essential functions which the process control computer must be capable of performing.

(i) The scheduling of a variety of tasks in such a way as to effect overall control of the process.

(ii) The initiation of a specified task, indicated by any of the following occurrences :

(a) A signal derived from a process.
(b) A request from another task, currently being executed.
(c) A signal from one of the computer system's own external or internal peripherals.

(iii) The temporary suspension of a task currently in execution, following the request of a task with higher priority.

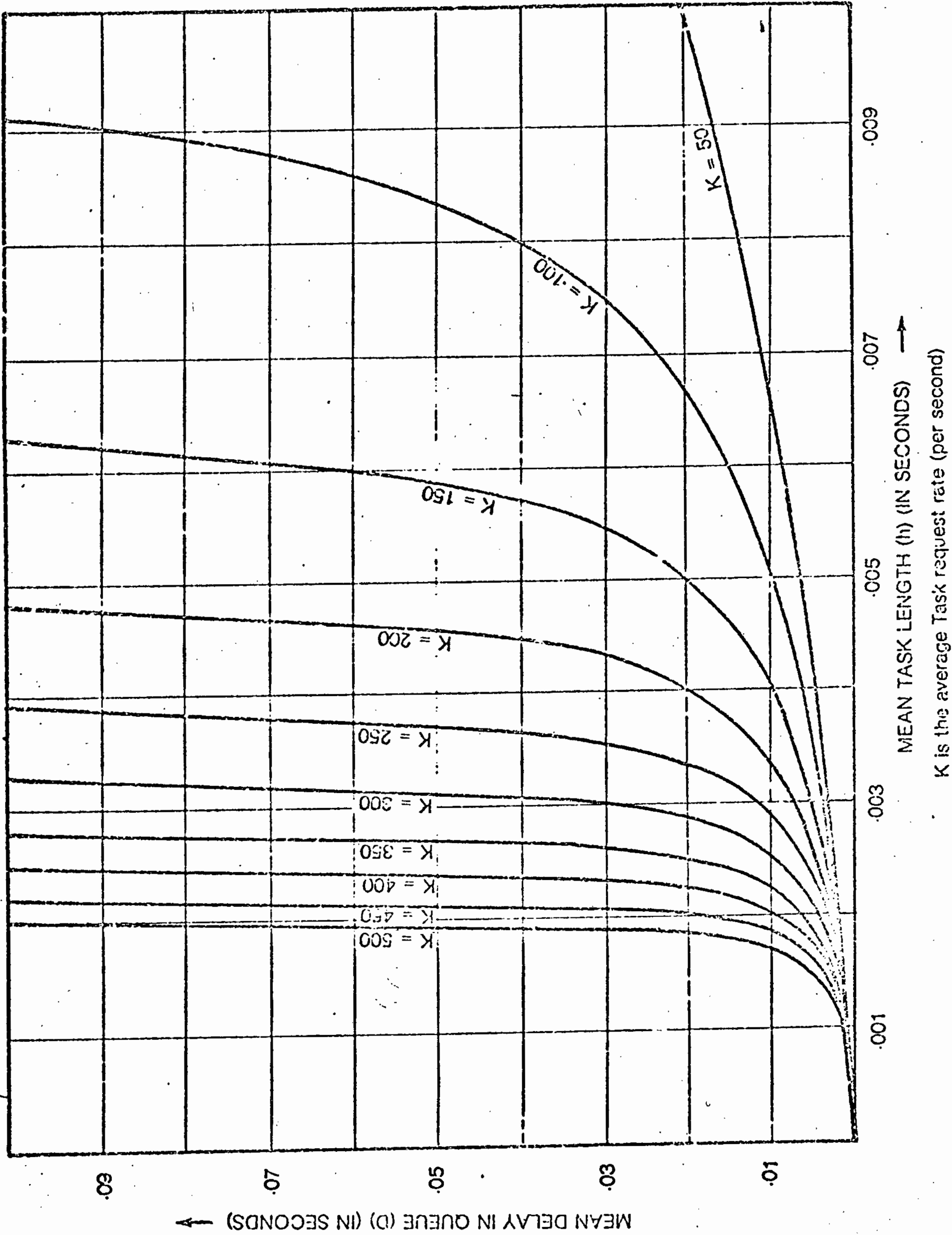
Requests, as outlined above, arrive at the computer, and their corresponding tasks are executed as soon as possible, under the guidance of a chosen scheduling policy. One or a few processors must handle many requests. Queues of tasks waiting for execution will consequently occur. The resulting characteristics of the queue are of the most fundamental importance in the organizational structure. If a particular computer organization cannot handle the request rate, the queue may continuously expand resulting in infinite delay time-in-queue. In most industrial applications there will be deadlines for certain tasks. If a task is not completed by the deadline, the process will collapse. Queue characteristics give a clear indication of the delays which may be expected, and will be the basis in deciding whether a particular organizational structure will be capable of controlling the process.

Rodd[2] gives a detailed statistical analysis of tasks in a multiprogrammed computer system based on traffic theory, and has formulated a queue characteristic of the fundamental importance to the system designer in a multiprogramming environment. That is :

The average delay, D, of task in a queue is given by $D = \frac{h}{1-Kh}$

where, K is the average request rate, and
h is the average task execution time.

This expression relates the delay which can be expected by a task in a queue in a multiprogramming system, to the demands made on that system (i.e. average request rate, and average task execution time). The average delay in a queue can therefore be plotted as a function of these demand characteristics. The resulting graph (Graph 1.1)[3] is a family of curves, each curve being formu-



Graph 1.1 Average Delay in Queue as a Function of Task Characteristics.

lated from different average request rates and average execution times.

When analysing a particular process control environment, one must look at the average delay times which are permissible. This is dependent on the process being controlled, that is, different processes have different time-constants. By estimates of average task arrival rate and average task execution in a system, the system designer can see if a proposed computer organization will adequately meet the permissible average delay times. Estimates can also be made on the amount of system expansion possible. This analysis will be used in proposing and analysing possible organizational strategies.

The analysis has been made on the following assumptions about the characteristics of requests and the servicing of tasks :

(i) Requests for tasks will arrive randomly at the processor, and their corresponding task execution times will be random. This assumption is reasonable in most process control environments.

(ii) The time required to cause the suspension of a current task and to start the execution of a new task will be instantaneous. In practice this is not the case, and this time is most significant. (The process of changing from one task to another is called task-swopping. The time required to perform a task swop is the task-swop time).

(iii) Tasks will be serviced on a first-come-first-served basis. This assumption is not generally true, as tasks will be serviced according to their prescribed task priorities.

The effect of task-swopping and task priorities on the above analysis will now be discussed.

Rodd[4] shows that if all tasks are held simultaneously in the computer's main memory, the task-swop time can be as high as 500 microseconds, whilst if backing store is used, task-swop time can be in the order of milliseconds. Every task which comes into execution requires a task swop. The task-swop time must therefore be indicated in the average task execution time. In the above equation, h will now be given as the sum of the actual task execution time and the task-swop time.

Another important factor influencing the above analysis is task priorities. Tasks are allocated priorities in order to ensure that they are executed within specified times. When a task is requested that is of higher priority than the task currently being executed, the latter must be suspended. The higher priority task will then begin execution. The suspended task is then placed back into the queue. The nett effect is that each task suspension will cause a task swop. A system having a large number of different priority tasks will spend much of its time performing task swop activities on tasks which are not fully completed.

Task swops will be most significant in those systems requiring backing store. This point is expanded below.

Backing Store

Most industrial-control computers require a large storage space to store all the tasks required by the system, as well as data which will be collected about the plant performance. The tasks could all be stored simultaneously in main memory. Task swops would then be efficiently accomplished. There are, however, serious disadvantages with this method of storage in those industrial-control computer systems requiring a large main memory. Whilst memory prices are falling, high-speed memory still remains comparatively expensive. Fast memory is between 10 and 1000 times more expensive per bit than slower backing store memory (such as disc or drum)[5]. A large main memory will require a large word size to efficiently address that memory. (One method of addressing a large main memory with a small word size is to use base registers. The number of base registers required for any address is a function of both word size and main memory size. Task execution times will be increased, as base register modifications will be continuously required. Consequently, efficient addressing is achieved by a large word size). Program expansion, which is fundamental to many industrial computer applications, will become difficult, as main memory size and the computer's word size may consequently have to be expanded. Such a system is inherently inflexible and costly.

The alternate method is to multiplex main memory. That is, only a small number of tasks are kept in main memory with the remainder of the tasks on

a backing storage device. Tasks would then be brought into main memory when required. Tasks will remain in main memory until a task has to be brought off the backing store device into main memory, and no main memory space is available. One or more tasks would then have to be written back to the backing store device (if necessary) to make main memory space available for the new task. The algorithm controlling which task has to be moved to backing store is complicated, and would be handled by a management system.

Lengthy task-swop times for tasks residing on backing store can be expected for the following reasons :

(i) A large amount of processing is required to initiate a backing store transfer (this point is discussed in detail in Chapter 2).

(ii) In most small computer systems, there is only one main memory access path[6]. Data will be typically transferred between the backing store device and main memory by 'cycle stealing'. That is, a main memory cycle is taken up between two successive processor instructions. This effectively slows down the main processor.

(iii) When a task transfer from backing store is initiated, there may be no other task in main memory ready for execution. Consequently, the task transfer from backing store cannot be overlapped with the execution of a task in main memory. In this situation, the task-swop time must be considered as being the time from the initiation of the task transfer, to the time the task is fully in main memory and ready to execute - as backing store is comparatively slow, this time will be significant.

The above discussion highlights the importance of minimizing the non-productive function of task-swopping. This point is expanded later.

1.2 Organizational Strategies

When a system designer is evaluating a particular computer system, he will do it in terms of the average delay curves. The designer will know the plant parameters, and will thus be able to calculate the permissible average task delay time. For example, consider a process which will have an average delay of 0,04 seconds. If the maximum task arrival rate is 200 tasks per second,

then the average task execution time must be less than 0,0045 seconds (refer to Graph 1.1). If the average task execution of a particular computer system was in excess of 0,0045 seconds, then a different computer system must be considered. A solution could be to obtain a faster computer, but the faster computer may not be sufficiently fast to get the average task execution time into the required limits. Different organizational approaches to solving the problem must then be considered. If fewer tasks were allowed to enter the system, then a greater average execution time would be allowed. For example, if the task arrival rate could be lowered to 150 tasks per second, then the average execution time of 0,0057 seconds is allowable. Another approach would be to decrease average task execution time. In the above example, if the task execution time could be lowered to be less than 0,0045 seconds, then the process receiving 200 tasks per second could be adequately controlled. Consequently, two approaches can be made :

(i) Reduce the average request rate.
(ii) Reduce the average task execution time.

1.2.1 Reducing Average Request Rate

One method of reducing the task arrival rate is to distribute the computer system. Distributed systems are particularly suited to those industrial control environments which have a high degree of process (and hence task) independence, and where only a small amount of communication between the different controlling processors (nodes) is necessary. In certain process control applications, however, certain processes may have a high degree of dependence on each other, and a large information flow between these processes may be necessary. Trade-offs will exist between moving a large amount of data between different nodes or having only one node controlling the different processes. More simply stated, not all industrial control applications lend themselves to a high degree of distribution.

Another method which could be used is to use a multiprocessor system. Having more than one processor in the system means that the average task arrival rate at any one processor will be proportionally lower. That is, the average task arrival rate at a processor in a two processor system will be half that of a one processor system. There are a number of inherent problems with this approach[7]. One of the problems is that like the

distributed processing system, a large amount of communication between tasks may be required (communication in the multiprocessor system will occur by means of a common memory). A large amount of the system processing resources will go into handling this task communication and in handling the specialized management functions required in such a system. Consequently, with the reduction in average task arrival rate, the average task execution time will increase. In this instance, the improvement in system performance by adding an extra processor may be small, and the percentage improvement will reduce drastically as the number of processors increases. The cost of the multiprocessor system is high, and whilst it may be suited for certain process control applications, it is necessary to investigate if alternate computer organizational strategies are possible. (One of the arguments used in favour of the multiprocessor system is reliability. The argument is that if one processor in the system fails, the system will continue to function although in a degraded state. This argument is suspect, as it makes assumptions about software reliability and hardware failures which are not always true. Producing fail safe hardware and fault free software is very expensive (if not impossible), and better reliability and cost factors may be obtained by duplicating a simpler organizational structure).

1.2.2 Reducing Average Execution Time

As mentioned above, the time required to execute any task can be divided into two parts :

(i) The time required to perform the productive part of the task - that is, those instructions which perform the prescribed function of the task.

(ii) The time required to perform the non-productive part of the task. This time will be spent performing task swops.

These two points will now be discussed in detail.

(i) First consider the problems of reducing the time spent on the productive part of tasks. One method of reducing the execution time of a task would be to reduce the number of instructions needed to be executed to perform that task. The best way to reduce the number of instructions would be to write programs in

machine code. This solution is rarely considered, as machine code programs are not only more difficult to write than high level language programs, but are more difficult to modify and debug, and hence tend to be less reliable. The labour costs involved in machine code programming are higher than when writing in a high level language[8]. This solution to the problem is thus not recommended. What is recommended is that the correct language choice be made when writing programs (for example, CORAL 66 may be chosen instead of FORTRAN, if such a choice is available), and that careful programming, with certain functions performed in machine code, will help reduce program execution time. Another solution is to have a machine which is architecturally more suited to a real-time language such as CORAL 66. Such a machine will have better execution times for this high level language than a conventional computer, thus helping to satisfy some of the discrepancies between the inefficiencies of a high level language on the one hand, and the advantages of a high level language on the other hand[9].

Task execution can also be reduced by implementing certain commonly executed software functions in hardware. For example, a hardware floating-point arithmetic unit will greatly improve task execution time if a large amount of floating-point arithmetic is necessary.

(ii) Secondly, consider the problems in reducing the time spent on task swops. It has been mentioned above that task-swop times will be most significant in a system having backing store. Improving the average task execution time in such a system will most naturally be achieved by reducing the average task-swop time of tasks from the backing store device.

A file system is necessary in a system having backing store. The file system (filing system) is that part of the operating system which organizes and controls backing store. (This definition is narrower than the general definition of the filing system. A more general definition can be given to include those systems (such as Multics) where the user thinks he is using a very large main memory, all movements between main memory and

backing store being invisible to him[10, 11]. In such a system, the file system is generally defined as the total memory management system).

One method of reducing task-swop times is to delegate most of the file system to a separate processor. This processor can be called a backing-store processor. The backing-store processor will take a large processing load off the main processor, allowing the main processor to spend more time performing useful processing functions. The backing-store processor will generally be a small computer (minicomputer or microcomputer), with the filing system functions performed in software. A possible backing-store processor configuration with the backing-store processor interfaced to a disc drive is illustrated in Figure 1.1.

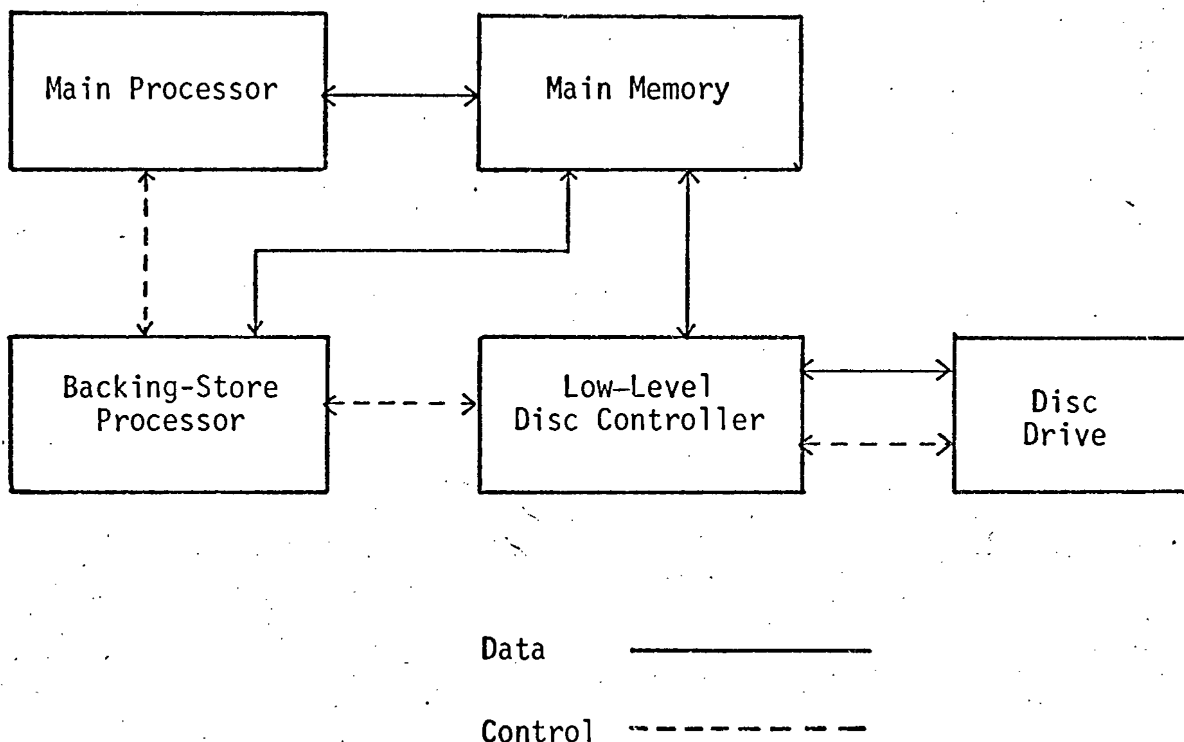


Figure 1.1 A Possible Backing-Store Processor Configuration.

If a task is required which is currently on backing store, the central processor will initiate the task transfer to main memory by passing a short message to the backing-store processor. This message will include the task name, the main memory address where the first word of the task is to be loaded, and the number of words of the task to be loaded. The backing-store processor will do the necessary processing to begin the data transfer to main memory (by cycle stealing). The central processor will at the same time, be executing some other task in main memory, if there is a task in memory waiting for execution. Bringing a task from backing store into main memory, and the execution of some other task already in main memory can therefore be done in parallel. Task-swop times will therefore be the normal swop times between tasks in main memory; the time required to execute those instructions necessary to initiate the transfer from backing store; and the time used for cycle stealing when transferring data to main memory via a direct memory access facility. The net effect is to reduce the average task execution time for tasks being brought off backing store.

The above discussion assumes that the backing-store system will be fast enough to ensure that the next highest priority task will be in main memory waiting for execution before the completion of the currently active tasks. This is not generally true in a real-time environment. The random nature of task arrivals; the fact that certain tasks must meet certain deadlines; the fact that tasks must be transferred from backing store according to task priorities; and the fact that task-swop times from the backing-store system to main memory may be longer than the execution times of certain tasks, will require the backing store system to be as fast as possible. If the backing store system is too slow, the following undesirable situations may occur :

(a) all tasks currently residing in main memory may either be completed or suspended, with a queue of tasks waiting to be transferred from backing store to main memory;
(b) a task may be in execution, but a queue of higher priority tasks may be waiting for transfer from backing store into main memory; or

(c) a task requiring to meet a certain deadline may be requested. The time spent bringing that task off backing store into main memory may mean that the task deadline cannot be met.

Consequently, if the backing-store system is too slow, the system's response to the environment it is controlling may be inadequate. The cause of this undesirable situation being inadequate task transfer speeds from backing store. A possible solution to this problem is to use a high-speed backing store device such as "bubble" memory or fast fixed-head disc or drum. In certain systems, even the introduction of these high-speed devices may prove inadequate. A high-speed backing store device controlled by a backing-store processor forms an expensive system. The expense of such a system may be unnecessary.

1.3 The Proposed Approach

The problems involved in organizing an industrial-control computer system have been discussed. It has been shown that there are many ways of producing an efficient computer organization. It is impossible, however, to propose a general computer organizational structure which is suited to all process control environments. Each application will be different, and will require different solutions.

The demands on the computer system when backing store is used is significant. It has been indicated that the response to the environment in a process-control computer system having backing store can be improved by :

(i) minimizing swop times of tasks from backing store to main memory, and
(ii) minimizing the interference backing store operations have on 'useful' processing operations.

The above discussions have indicated methods which could be used in attaining these goals. Introducing a multiprocessor or backing-store processor will allow one processor to perform backing store operations whilst another processor performs useful processing operations. Writing programs in

machine code will reduce task size[12], and minimize swop times. Introducing a high-speed backing store device will have the same effect.

Cost is of great importance to any system's organizational solution. Certain of the above methods are inherently costly to implement. There are other equally important factors which must be considered. In the real-time process control environment, factors such as system reliability and system upgradability must be considered.

Before any alternate structures which may help in attaining the above two goals can be considered, it is necessary to examine the hardware/software relationship in the real-time process-control computer.

1.4 Hard-Soft Trade

Essentially, functions in a computer system can be implemented in hardware or software or firmware. The decision as to which functions are to be performed in one of the above is fundamental to the design of any computer system. A hard-soft trade is the trade between hardware and firmware, hardware and software, and firmware and software. It is necessary to examine the reasons for performing the hard-soft trade in a real-time process-control computer system.

(i) Improving System Performance

One of the most common reasons for trading software for hardware or firmware in a real-time system is to improve system performance[13]. For example, improved performance will be achieved by including a hardware floating-point arithmetic unit, or a fast Fourier transform processor[14]. Memory management can be moved from software to hardware, as is illustrated by the SYMBOL project[15]. This greatly improves system performance.

Certain operating system functions require the frequent manipulation of tables and queues. Implementing certain of these functions by means of microinstructions can greatly reduce system overhead. A microprogram sequence used in place of a normal subroutine can save several memory cycles. A few examples illustrate this point. The

interrupt-handling and scheduling functions can be implemented by a special microprogrammed processor. Such a system will greatly reduce task-swop times[16]. The Scientific Control Corporation (SCC) 6700 system updates the associative map (used in segmentation) from the segment and page tables by a special microprogrammed processor[17].

(ii) Improve Reliability

Good system reliability is of fundamental importance in any real-time system. Software failure occurs from the accidental overwriting of programs and of introducing faults into the software by frequent software modification. Committing the more critical functions to more secure locations such as firmware or hardware will improve system reliability[18]. There is an inevitable tradeoff between security and high executive overheads when the security functions are performed in software. The GEC 4080 reduces these overheads by implementing all the 'innermost' executive functions in firmware (by a microprogram called Nucleus)[19].

Reducing software complexity by moving certain functions to hardware will aid in improving system reliability.

(iii) System Upgradability

The life-span of a computer system is largely dependent on its ability to meet changing demands. Many process control systems are subject to continuous expansions and modifications. The systems adaptability is achieved by assigning hardware functions to software or firmware. Emulators are an excellent example of the hard-soft trade. The ability of one machine to emulate another is of great value, as it allows software developed on one machine to run on another. The emulation could be performed by software, but this would be slow. Microprogramming helps to solve this problem. For example, the Burroughs B1700 computer system has no machine language, allowing any definable language to be implemented[20]. The B1700 is a hardware-orientated structure which is totally microprogrammed.

1.5 Hardware Implementation of the Filing System

The above discussion of the hard-soft trade has indicated that the hardware

approach is suited to the real-time environment. One method of reducing swop times from backing store using hardware is illustrated by the SYMBOL project[21]. SYMBOL demonstrated that a high-level language, virtual-memory time-sharing system could operate entirely in hardware. An all-hardware memory management was a key aim of the SYMBOL project. Virtual memory for the system is derived from a small core memory and a larger drum memory (managed by a drum controller and a page table that maps virtual memory addresses into core memory addresses). The memory management (which includes the routines to organize and control backing store) is performed entirely by hardware. The major objection to the strictly hardware approach is the lack of flexibility.

The filing system functions could be performed by a hardware unit which is microprogrammed. This is the approach adopted by the Scientific Control Corporation (SCC) 6700 system. The SCC-6700 uses separate computers to control input/output[22]. These computers are called I/O processors. All I/O processors are identical. Each I/O processor has its own memory and is microprogrammed, and each I/O processor has a subset of the instruction set of the CPU. I/O processors can also execute normal input/output instructions, and can contain the allocation and control programs to handle the set of devices attached to it. The I/O processor in charge of the backing storage system (i.e. disc and drum) is called the Auxiliary Memory Control Processor. This processor controls a special high-speed channel to the system main memory. This system is illustrated in Figure 1.2. Basically, a task can be created to bring information off backing store. The task will be run on the Auxiliary Memory Control Processor.

1.6 An Alternate Disc Controller

The SYMBOL and SCC-6700 systems perform most of the processing required by the filing system (in either hardware or firmware units), thus resolving most of the disadvantages of the software backing-store processor. These systems are designed around large computer systems with large, general purpose operating systems.

The area of interest is that of real-time process control. A small computer, interfaced to a backing-store unit, and being controlled by an easy-to-use, fast, reliable operating system, usually forms the processing system for

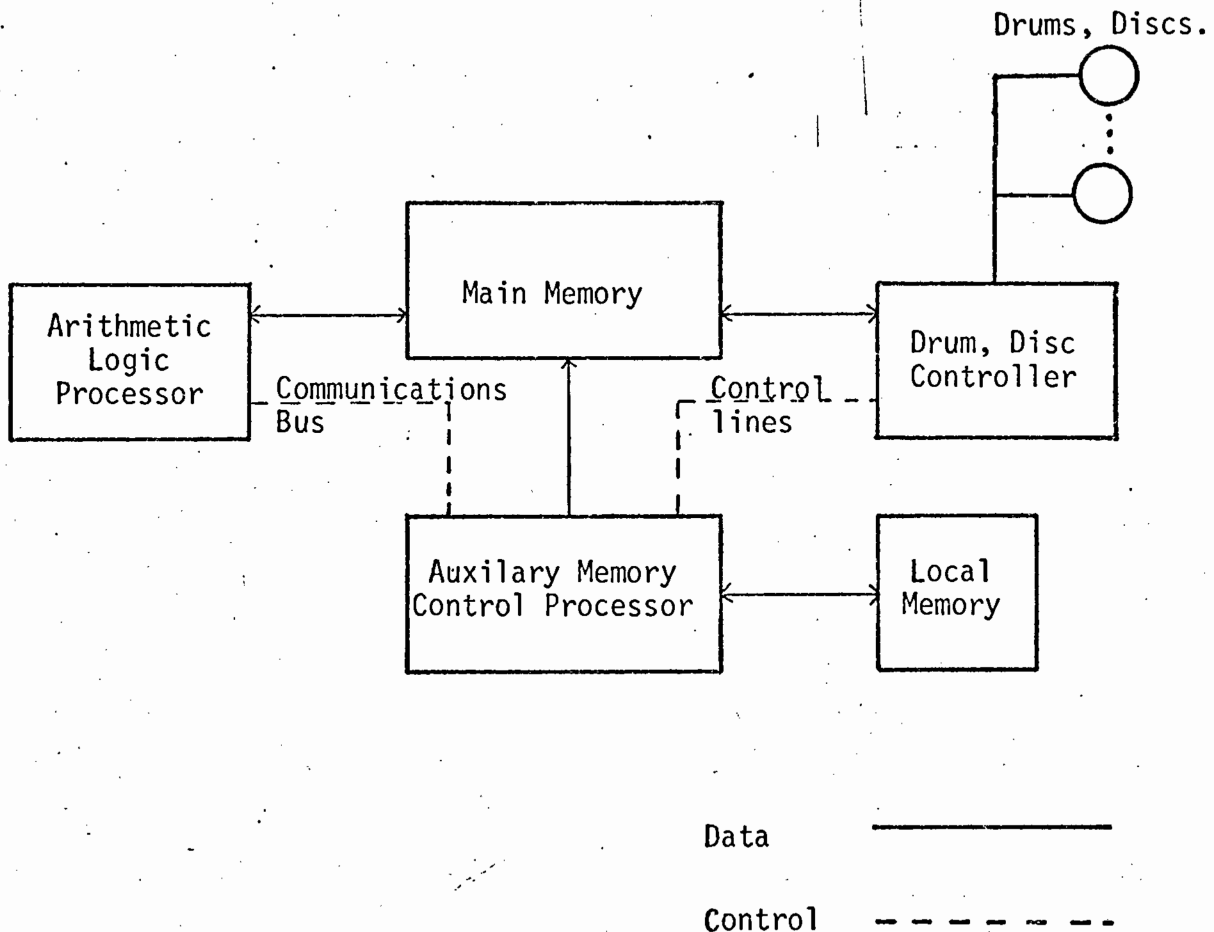


Figure 1.2 The SCC-6700 Architecture

this environment[23]. It must be investigated if the above methods can be adapted to such an environment.

One area which has not been fully investigated is the placement of more processing power in the low-level device controller.

The basic function of a low-level controller is to provide an interface between the processor in charge of the backing store, and the backing-store device. The functions of the low-level controller are generally performed by a hardwired unit. A typical low-level controller for a disc drive requires between 150 and 250 circuit elements[24]. A comparable unit which is microprogrammed has been developed by Intel, and only requires 67 circuit elements[25]. Conventionally the low-level controller is hardwired, and hence inflexible. To allow the system designer flexibility in file organization and control, it has therefore been necessary to limit the

low-level controller functions. The typical low-level controller will only provide assistance in performing basic system functions. (In a disc drive these functions include "seek" to a prescribed cylinder on disc, or "read" or "write" a number of contiguous sectors from disc). With the advent of microprogrammable controllers, the philosophy of restricting the device controller to these "low-level" functions comes into question. A microprogrammed controller will allow a large amount if design flexibility which is not available in the hardwired unit, and will still be sufficiently fast to manage a high-speed backing store device. It is clear that this area requires further examination.

This thesis investigates whether it is feasible for the low-level controller to help in the performing of filing system functions. It will be shown that it is possible for the low-level controller to be combined with some (or all) of the filing system functions into one hardware unit which is totally microprogrammed. This hardware unit can be called a High Level Controller. (The unit could possibly be called an "Intelligent Controller", a "Hardware Filing System", or various other names). This thesis sets out to show that a High Level Controller is particularly feasible in a small real-time process control environment. This will be done by building an experimental High Level Controller and interfacing it to a backing-store device. The backing-store device is a moving-head disc drive, and the resulting hardware unit can therefore be called a High Level Disc Controller (HLDC). Figure 1.3 illustrates a possible HLDC configuration.

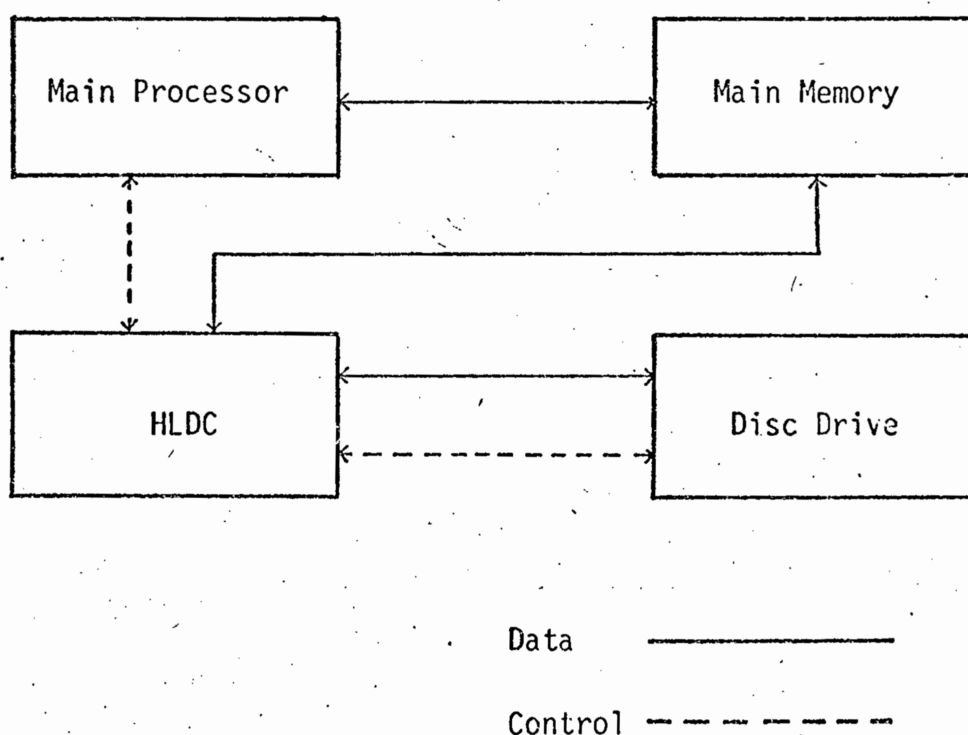


Figure 1.3 Possible High Level Disc Controller Architecture

1.7 Summary

This chapter has shown how the analysis of queue characteristics of tasks in a multiprogramming computer system can help the system designer evaluate different computer organizations for the real-time industrial control environment. The industrial-control computer systems under review are those systems which require backing store (that is, not all tasks will be held simultaneously in the computer's main memory). The backing store device will place large demands on the computer system, resulting in increased task-swop times and consequently increasing the average task execution time. The response of the computer system may then become unacceptable, and consequently different computer organizations must be considered to achieve the desired computer response.

It has been shown that the following two goals are desirable in a system having a backing device :

(i) Minimum swop times of tasks from backing store to main memory.
(ii) Minimum interference of backing store operations on 'useful' processing operations.

The examination of the hard/soft trade has revealed that the hardware or firmware approach is ideally suited to the real-time process control environment. Based on this hard/soft approach, a suitable hardware structure, which is the combination of the conventional low-level controller and the filing system, has been proposed for the real-time process control environment.

References

1. M.G. Rodd, "Organization of Industrial Control Computers", unpublished Ph.D. dissertation, Dept. of Electrical Engineering, University of Cape Town (1976), p. 4.
2. Rodd, pp. 1-9.
3. Rodd, p. 7.
4. Rodd, p. 9 and appendix I, pp. 1-12.
5. Raymond P. Capece, "Memories", Electronics, Vol. 51, No. 22 (October 26, 1978), pp. 126-132.

6. Richard W. Watson, "Timesharing System Design Concepts", McGraw-Hill, Inc. (1970), p. 82.
7. P.H. Enslow Jnr, "Multiprocessor Organisation - A Survey", Computing Surveys Acm, Vol. 9, No. 1 (March 1977), pp. 103-129.
8. J.K. Broadbent, "High Level Language Implementation through Microprogramming. Microprogramming and System Architecture", International State-of-the-Art Report (1975), pp. 337-357.
9. M.C. Sole, "An Optimal Real Time Language Processor", unpublished M.Sc. Thesis, Dept. of Electrical Engineering, University of Cape Town (1978), pp. 1-21.
10. Watson, pp. 187-189.
11. A.C. Shaw, "The Logical Design of Operating Systems", Prentice-Hall, Inc. (1974).
12. B.A. Wichtman, "Evaluating Languages for Real-Time", Infotek State-of-the-Art Report on Real-Time Software, pp. 647-655.
13. R.L. Mandell, "Hardware/Software Trade-offs - Reasons and Directions" AFIPS Conference Proceedings, Vol. 41, FJCC (1972), pp. 453-459.
14. M.H. Corinthios, "A Fast Fourier Transform for High-Speed Signal Processing", IEEE Transactions (August 1971), pp. 843-846.
15. H. Falk, "Hard-Soft Tradeoffs", IEEE Spectrum, Vol. 11, No. 2 (February 1974), pp. 34-39.
16. Rodd, pp. 1-20 and pp. 80-84.
17. Watson, pp. 71-73.
18. Mandell, pp. 453-459.
19. A.O. St. John, D.N. Fish and W. Fingland, "Operating System Design for On-Line Control", pp. 189-197.
20. W.T. Wilner, "Design of the Burroughs B1700", AFIPS Conference Proceedings, Vol. 41, FJCC (1972), pp. 489-497.
21. Falk, pp. 34-39.
22. Watson, pp. 98-99.
23. W.F.C. Purser and D.M. Jennings, "The Design of a Real-Time Operating System for a Minicomputer. Part 1", Software - Practice and Experience, Vol. 5 (1975), pp. 147-167.
24. Intel Data Book, Intel Corporation, Santa Clara, California (1976), pp. 750-751.
25. Ibid.

CHAPTER 2

REQUIREMENTS OF THE HIGH LEVEL DISC CONTROLLER

Before formulating a possible hardware structure for the High Level Disc Controller (HLDC), the fundamental processes to be performed by this structure must be determined. Basically, the HLDC must :

(i) perform the functions specified by the host computer (Figure 2.1 illustrates the host computer/HLDC interface). The functions specified by the host computer are the filing system functions.

(ii) perform the functions specified by the disc drive.

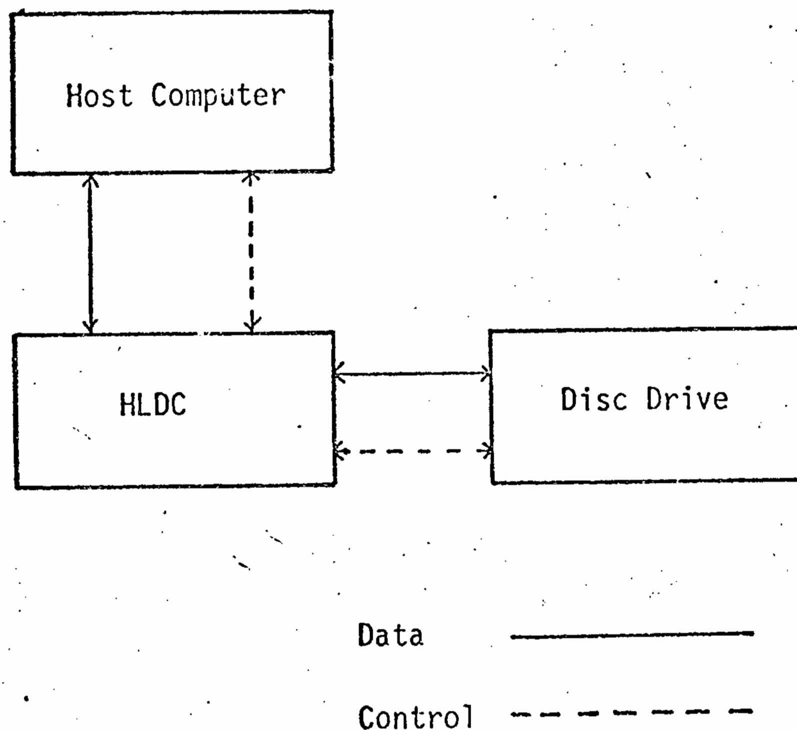


Figure 2.1 HLDC/Host Computer Interface

The text which follows begins with a review of the essential filing system requirements.

2.1 Essential Filing System Requirements

A file is a collection of related information with a name[1]. Files are stored in a computer system as a string of elements : bits, characters, computer words, etc., and may reside in main memory or on backing store (e.g. disc, drum, magnetic tape, paper tape, etc.). The set of routines handling backing store is called a filing system. One of the goals of the filing system is to make the user independent of the system hardware. The user works with logical records, whilst the backing store devices are more naturally organized into physical records. A file is made up of one or more logical records, which may be of arbitrary length or structure. The physical records can be made up of one or more logical records. If the logical record is smaller than the physical record, more than one logical record will usually be packed into the physical record to increase I/O transfer rates and storage efficiency.

The filing system should satisfy the following basic user requirements[2] :

(i) The user should be able to access files by symbolic name;
(ii) The user should be able to create, change and delete files;
(iii) The user should be provided with file backup in case a file is accidentally deleted or corrupted.
(iv) The user should be able to control who has access to his files, and the type of access allowed (e.g. read, write and execute).

The above filing system functions must be implemented so that the following design goals can, as far as possible, be attained :

(i) The filing system must minimize average response time to any request made on it.
(ii) The filing system must maximize the storage efficiency of backing store.
(iii) The filing system must maximize the reliability of the information stored on backing store.

Tradeoffs will inherently occur in attempting to fulfil the above three goals. System reliability will usually require redundancy. This redundancy occurs by adding check information to files in the form of cyclic redundancy codes, etc. Consequently, good reliability will tend to reduce storage efficiency and system response. Tradeoffs will also occur between storage efficiency and response. That is, the methods of storing data which allow maximum system response may result in poor storage efficiency.

Before examining how the above user requirements can be implemented, it is necessary to discuss the file system's data structures.

2.2 File System Structure

An examination of the basic data structures required by the filing system will help indicate which of these data structures can be managed by the HLDC. This will, in turn, indicate which filing system functions can be performed by the HLDC, and which functions should be performed by the host computer.

2.2.1 File Directories

Files are accessed by symbolic name. Consequently the filing system must translate the symbolic file name to the address where the file is stored. The filing system performs this translation by means of a file directory. That is, the file directory stores symbolic file names and corresponding physical addresses. Appendix A.1 discusses file directories in detail. In certain systems (notably those systems under the control of large time-sharing operating systems, or virtual memory operating systems) the user will not know on which backing store device his file is stored[3,4]. In such a system, the symbolic file name is used together with the file directory to locate the name or number of the backing-store device, as well as the address of the file on that device. In a system where there is only one backing-store device, or a system where the user must specify the device name or number, then the file directory function of indicating the backing-store device name or number is redundant. The above distinction between the two types of file directory functions is important, as it indicates what portion of the filing system can be performed by the HLDC.

As mentioned in Chapter 1, the system under review is a small computer system controlled by a fast, reliable operating system. It can be assumed that the operating system will require the user to specify the backing-store device name or number[5]. In such a system the HLDC can handle the file directories and it will be shown that the HLDC can perform most of the filing system functions.

It is useful, at this stage, to briefly investigate the types of filing system functions which could be performed by the HLDC if it was to be interfaced to a more generalized type of system. As mentioned above, in certain computer systems, the user will not know on which device his file is stored. In this situation, the file directory is used to determine the device name. The file directories in such a system are most efficiently handled by a centralized processor. The HLDC would still be useful in this environment, as it will be able to perform much of the organization and control of data structures required when storing data on disc.

The remainder of the text assumes that the HLDC will perform the file directory management.

2.2.2 File Storage

The filing system must resolve the following two issues :

(i) How to keep track of the locations and proper sequencing of physical records on the disc; and

(ii) How to keep track of the space which is not being used (free space), so that this space can be used for new files and the expansion of old files.

A review of some of the methods storing data on disc is given in Appendix A.2. The method of storing data that is recommended is the index block method. This method allows efficient disc accesses whilst maximizing data storage. Appendix A.2 gives a description of the index block method, as well as a description of how free space maps are used to indicate used and unused data sectors.

2.3 Implementation of the Filing System in the High Level Disc Controller

It is now necessary to examine how the file system will implement the above

mentioned user requirements. This examination will be in terms of a computer system interfaced to an HLDC. Four major functions can be recognized in this system when performing the file system functions.

(i) Buffering of information transmitted between backing store and main memory.
(ii) Reserving and allocating of files.
(iii) Protection of the files dedicated to one user from interference by another user.
(iv) Handling of communication between HLDC and host computer.

The following discussion will indicate how these functions can be performed.

2.3.1 Buffering

The information on disc will generally be required to be transferred in two different ways :

(i) When a task is initiated, the information necessary for the execution of this task may be on disc. In this situation, the host computer will request that the information relating to the task be swopped into an area of main memory so that the execution the task can begin. Examples of this type of transfer would be the transfer into main memory of parts of the operating system, or an applications program or users program which is to be run.
(ii) A task may require to use the information on disc as data. Examples are a compiler using a source program as data, or an applications program storing information about plant performance in a data base. In this situation, the task may only request one or a few words of data at a time. Buffering is therefore necessary in this situation. A buffer is a storage area used to give a better match between the central-processor speeds and I/O speeds[6]. In the case of disc the buffer size would generally be the size of a sector (i.e. 1 physical record size). For example, if a program is writing to disc, it may be writing one word at a time. This data, under the control of a special task, would be stored in a buffer in fast memory. When the buffer becomes full, a request will be made to transfer the contents of this buffer to disc.

One of the problems to be solved is whether buffering should be performed by the HLDC or by the host computer. If buffering is performed by the HLDC (i.e. the HLDC would require fast memory to contain such buffers), then the host computer would have to initiate an information transfer every time it required one or a few words of data. The currently executing task in the host computer may then have to be blocked to wait for the HLDC to respond to this request. In this situation, a large amount of processing time will be taken up by the initiation of each data transfer. That is, the host computer will require to send certain information about the file to the HLDC (e.g. file name, etc.), and a task swop will have to be performed. Alternatively, if the host computer does the buffering, a large amount of main memory space may be required to store the buffers, since a number of tasks will generally be accessing disc in parallel, and each task will require its own buffer. Added to this is the processing required to allocate and manage these buffers. Buffering methods are discussed in more detail in the reference Watson[7].

Point (i) above raises the question whether certain tasks may be allowed to by-pass parts of the filing system. This would have the advantage that information could be very quickly accessed. For example, if the task knows the address of the data on disc, it would then be unnecessary to access the file directory. This would minimize disc accesses and would consequently allow fast response to a request for a task transfer. One disadvantage with this method is that certain information would require fixed locations on disc. This would introduce complexity into the filing system. Another disadvantage with this method is that file security and integrity may be degraded. A possible solution is to keep certain parts of the file directory permanently in fast memory (refer example in Appendix A.1). Such a system will allow fast access whilst maintaining file security, flexibility and simplicity.

2.3.2 Communication between the HLDC and Host Computer

The general file system must cope with a wide variety of I/O devices. To create a design which is as conceptually simple as possible, the designer generally isolates as many device-dependant characteristics as possible in separate routines. These routines are often called device drivers. The

device drivers are then interfaced to more general routines which are device-independant[8]. The set of routines which communicate directly with the HLDC can be called the HLDC driver.

There are six basic file commands which any filing system must respond to. These are the commands "create", "delete", "open", "close", "read" and "write"[9]. It is necessary to prevent the HLDC receiving any of these commands in parallel. This point can only be thoroughly explained in terms of the HLDC's hardware. Basically, testing if the HLDC is ready to receive a command and transmitting a command to the HLDC cannot be performed by one instruction. The situation that could occur, therefore, is that immediately after the test operation on the HLDC's readiness is performed, the task performing this test may be suspended. Some other task may now test if the HLDC is ready. Finding it ready, this task could issue a command. There is a finite time between the issuing of a command to the HLDC being able to receive another command. During this period of time the HLDC is no longer ready. If the originally suspended task starts executing whilst the HLDC is not ready, problems may occur. This is a classical problem. The general solution to this problem is to channel all requests through global routines[10]. There will be six such global routines associated with the HLDC. These routines can be called Open, Close, Read, Write, Create and Delete[11]. The task requesting the file operation will make a call to the appropriate global routine. This routine will, in turn, communicate with the HLDC driver. (Communication would occur by means of normal task communication methods. A discussion of these methods is given in the literature[12,13]). The HLDC driver, by using semaphores, will ensure that the above mentioned situation does not occur.

To initiate a file command, the HLDC driver will first test if the HLDC is ready to receive a command, and will then pass a table of all the necessary information to the HLDC. This table of information can be called the filename block, and will generally include file name, file access control information, etc. The contents of the filename block will depend on the file operation to be performed. (Physically, the HLDC driver will only pass the address of the filename block to the HLDC, and the HLDC will access the information via a direct memory access facility).

When the file operation is complete, the HLDC will communicate this completion to the HLDC driver (probably by means of an interrupt). Since a number of commands may be active in the HLDC at any one time, it must be resolved how the HLDC will indicate to the host computer which command has been completed. This problem, together with certain other communication problems, will be discussed in Chapter 3.

2.3.3 File Operations

When the filing system receives a file command, it will perform the corresponding file operation (i.e. create, delete, open, close, read or write). It is essential that the filing system perform any file operation as quickly as possible.

Create and Delete

When a file is initially created (i.e. a create command received from the host computer), the symbolic file name, access control information, etc., will be placed in the file directory. When writing to the file for the first time, an index block will be reserved for this file. The address of this index block will then be placed into the file directory, thus associating the symbolic file name to a physical address on disc. When the file is deleted, the index block will be used to reset the appropriate bits in the free space map. The file name will then be removed from the file directory.

Since the HLDC will manage the file directory, index blocks and free space map, it will perform the create and delete operations. These two operations will be initiated as follows. The task requesting that a file be created (deleted) would communicate this request to the global routine Create (Delete). The information passed to Create (Delete) would be the file name, access control information, the expected size of the file, etc. Create (Delete) would pass this information to the HLDC driver. The HLDC driver would set up the filename block and would pass the filename block to the HLDC. (Physically, the task would set up a buffer containing information such as, the file name etc. The pointer to this buffer would then be passed to Create and then to the HLDC driver. Information may be added to this buffer by Create (Delete) or the HLDC driver to get the filename block into

the correct form so it can be received by the HLDC).

At this stage, the task which initiated the file command would become blocked. (Tasks can be in one of three states[14]: ready to be executed; blocked whilst waiting for I/O or some other task to finish; or running). The task would only be placed into the ready queue once the HLDC had signalled the end of the create operation to the host computer.

Open and Close

When a file is opened, the file directory will be searched and the index block address obtained. The address of the index block, together with certain other information about the file (i.e. file name, whether a character or binary file, access control information, etc.) will be placed into a table. This table can be called the file-control block. Whenever an open file request is received, it must be checked whether a file-control block already exists for that file. File-control blocks will be linked together. If the file can be accessed by many different tasks (e.g. if the file is open for read), the file-control block will contain an open counter which will be incremented and decremented by open and close requests respectively. When the counter reaches zero, the file-control block is relinquished. If the file can only be accessed by one task (e.g. if the file is open for write), then a request to open an already opened file will be rejected.

A task wishing to open a file stored on disc would enter the global routine Open. The request to open this file would then be passed to the HLDC driver, and then to the HLDC in a similar fashion to create and delete. The HLDC will check through the list of file-control blocks to see if a block exists for this file. If no block exists, the HLDC will create the appropriate file-control block. If a block does exist, then the HLDC will either increment the file-control blocks open counter, or will queue the open request. Before a request to read or write a file can be made, the file must first be opened. This will establish the file-control block.

The question now arises when, and how, buffers will be allocated. In terms of the discussion on buffers above, certain tasks will not require the allocation of buffers. In this situation, it may be desirable to issue one

command which would open a file, transfer some, or all, of the file into main memory, and then, possibly, close the file again. When buffers are required, it will be assumed that they will be allocated during read and write commands.

When a task is finished with a file, the file must be closed. Closing the file will free the resources assigned to the file in both the host computer and the HLDC.

Read and Write

Data is written to disc by issuing a write command, and is read from disc by issuing a read command. Since the disc is a random access device, files should be allowed to be used in a random fashion. That is, the user should be able to access any logical record in the file. This is generally done by specifying a location relative to the start of the file, and the number of words, characters or bits to be transferred.

When a task issues a read command, the following situation will occur. The task will enter the global routine Read. Initially, there will be no buffers assigned to this task. Assuming the host computer manages the buffering of data from disc, the read routine will obtain buffers from a buffer pool. The size of each buffer requested will generally be equal to one disc sector. The read routine will put a request to the HLDC driver to transfer data to main memory. This request will contain the number of words or sectors to be transferred, the buffer (i.e. main memory) addresses, and possibly the start address, relative to the beginning of the file, that the read must commence from. This data, as well as other information such as file name etc., will be placed into the filename block, and will then be transferred to the HLDC by the HLDC driver. The HLDC, on receiving a read request, will search through the list of file-control blocks. Using the information stored in the file-control block, the data transfer will be made. One important point which must be mentioned, relates to the fact that more than one task could be accessing one file for read. The problem here is to keep track of which part of the file any one of these tasks is reading from. This could either be done by storing certain information about each task accessing the file in the file-control block, or this information could be stored in the filename block.

A similar procedure will occur when writing data to a file. The major difference here is that a file may be open to only one task when writing. The HLDC, when writing data to disc, will require to access the free space map to reserve sectors for the data written. The HLDC will also update the index block for each sector written to disc. When a file is closed, the index block will be written to disc by the HLDC.

2.4 Filing System Response

As mentioned in Section 2.1, the filing system must respond to a request made on it as quickly as possible. That is, the filing system must minimize average response time.

2.4.1 Queues of Requests

The most significant time in data transfers to or from disc is the time taken to perform seek operations. It is therefore necessary to minimize the number of seeks and the seek distances so as to reduce the average transfer times to and from disc. In the multiprogramming environment, a queue of requests for file operations may occur. This queue would be managed by the HLDC, as the HLDC will know, by examination of the file directories and index blocks how to minimize seek times. Denning[15] and Teorey[16] examine the problems of minimizing seek times in detail.

2.4.2 Fast Storage Requirements of the HLDC

The HLDC will require access to fast storage for various reasons :

(i) It is desirable to have the currently referenced index block of each open file in the HLDC in fast memory. When data is written to disc, the index block can then be rapidly updated. During read operations, the address of the next sector to be read from disc can also be rapidly located if the index block is in fast memory.

(ii) As mentioned in Appendix A.2, 1250 words will be required to house the entire free space map for one disc drive. The 1250 words will be stored on disc.

During write and delete operations, a number of words in the free space map will require modification. These words will usually be contiguous or close to each other (refer Section 3.4.3). Consequently, when the free space map is to be modified, it is desirable to have the area of the map which is currently being modified in fast memory and hence avoid unnecessary disc accesses.

The entire 1250 words of the free space map may be kept continuously in fast memory. The free space map would then only be written back to disc when the system closes down. The advantage of this method is that accesses to the free space map will be efficiently accomplished. The disadvantage being that it is expensive in terms of fast memory usage.

An alternative method is to keep only part of the free space map in fast memory. The minimum amount would be one sector (128 words) of the map. When the map is to be updated, the appropriate part of the map would be brought into memory if it is not already there. If the current sector of the map has been updated, and a new sector of the map is to be brought into memory, the current sector must first be written to disc. This method has the disadvantage of increasing the number of seek operations when the map is to be accessed. The greater the percentage of the map kept in local memory, the greater the probability that the part of the map currently held in memory will be the desired part of the map, consequently reducing access times.

If the HLDC is in an environment which has a large amount of modifications to disc files, then as much as possible of the free space map should be held in fast memory. If, on the other hand the system has to manage a large but reasonably static data base, it would only be necessary to hold a small part of the map in fast memory.

When a file is expanded, it is desirable that the file be in contiguous sectors to allow maximum transfer rates. Checking

through the free space map for contiguous sectors can be a lengthy operation. Another map could be used giving the number of free sectors on each particular cylinder (the sectors may not, however, be contiguous). This map would be kept continuously in fast memory. In a system having 200 cylinders, the map would be 200 words in length. This map would aid searches into the free space map for free sectors.

(iii) As mentioned in Appendix A.1, it is desirable to have as much of the file directory in fast memory as possible to maximize the filing system's response.

(iv) The filename block (Section 2.3.2), the file-control block (Section 2.3.3) and the information associated with queues (Section 2.4.1) should be stored in fast memory.

2.4.3 Improving Transfer Rates between Disc and Computer

There are various methods of improving transfer rates, and three methods are discussed below :

(i) Both storage and transfer efficiency can be improved by packing character files. The routines to pack and then unpack these files would be performed by the HLDC.

(ii) Tasks requiring frequent access could be placed near the centre of the disc, whilst those tasks requiring infrequent access could be moved to the edges of the disc. Disc accesses would then be reduced by concentrating these accesses to a small area. A record of the number of accesses a task has had, and the date the task was created could be stored in the file directory. A routine in the HLDC could then attempt to perform the above reorganization on the basis of this information.

(iii) When files are expanded, parts of the file will often become scattered throughout the disc (remember the index block method of storing files is being used). This is particularly true when storage rates approach a maximum. Scattered files will cause slower transfer rates. For example, consider a common disc operation of transferring a number of sectors of data to the host computer's main memory. Assume three sectors of data are to be transferred. To transfer three sectors of data contained on different cylinders is a lengthy operation. If the data is

stored in three contiguous sectors in one cylinder, however, the transfer can be handled quickly and efficiently.

To realize maximum disc efficiency in terms of transfer rates and access time, the disc should be reorganized periodically to make files as contiguous as possible. The critical decision is when, and how often, to perform the reorganization. In those systems which are not continuously on-line, the reorganization of the disc can be done as an off-line operation. If the HLDC is interfaced to a system which is continuously on-line, which is the case with certain real-time systems, file reorganization could possibly be performed during the HLDCs idle time (free time).

2.5 Reliability of Data on Disc

As mentioned in Section 2.1, the filing system must maximize the reliability of the information stored on disc. The integrity of data on disc is put at risk by the probability of hardware failure, software malfunction and physical damage. There are various methods of improving data reliability. Software malfunction is reduced by placing the filing system in firmware. That is, by placing the filing system functions in secure locations (i.e. hardware), the file handling routines cannot be inadvertantly modified. This is important, since the modification of parts of the file system routines could cause wholesale corruption throughout the data base. This is one of the advantages of the HLDC over the conventional "software" filing system.

The disc drive, because of its mechanical nature, is the most unreliable component in the backing store system. Consequently it must be investigated whether or not the disc drives reliability can be improved, and what the overheads of such improvement will be.

To minimize the possibility of corruption of data on disc, it is often necessary to incorporate error detection and correction procedures. The greater the reliability required by the system, the greater the demands these procedures will place on the system in terms of both response time and storage efficiency (i.e. error detection relies on the existence of

redundancy in the data base). A detailed discussion on the methods of ensuring that the correct sector has been located, and that the correct data has been written and read is given in Appendix A.3.

The system designer must assume that corruption of data will occur, and consequently file backup must be provided. Data backup will generally be onto a slow backing store device such as magnetic tape. The recovery of data from such a device is slow, and if a time-critical task becomes corrupted, the recovery procedure will generally not be fast enough to enable the task to meet its real-time deadline. Consequently, it may be necessary to ensure that a duplicate copy of such a file is available, and can be transferred fast enough to meet its real-time deadline. The file may be duplicated by keeping a copy of the file on the same disc drive, on a different disc drive controlled by the same HLDC, or even on a different backing store subsystem.

Data on disc can only be accessed through the file directory, and therefore reliability of the file directory is important. The above discussion on time-critical tasks may make it necessary for a duplicate copy of the file directory to be maintained on a fast backing store device. The duplication of the file directory has certain disadvantages associated with it. A system which requires a large number of file directory modifications, will have to perform these modifications on two file directories instead of only one, with the resulting increase in disc seeks and data accesses.

On the plus side, if only part of the file directory is to be held in fast memory, two well placed file directories will help reduce the average file access time. Any file directory access will then choose the file directory closest to the current head position.

As mentioned above, it must be assumed that system failure will occur. In this situation, it is necessary to have a restart procedure which will assure that there is no inconsistency within the file control information (i.e. index blocks, free space map and file directories). Minor errors in the control information can provoke wholesale corruption throughout the data base.

2.6 Free Time

The HLDC will not be continuously active performing commands from the host computer. There are useful housekeeping functions which can be performed during this free time. The types of activities which could be performed during this free time have been indicated above, and include :

(i) the reorganization of a file;
(ii) the maintenance of a duplicate file directory; and
(iii) allowing the movement of frequently used files towards the centre of the disc, to minimize disc accesses.

The major problem in performing any free time operation is that a request on the file system may occur whilst the free time operation is in progress. This necessitates that free time operations be as short as possible to ensure that the request can become active as quickly as possible. It must also be ensured that free time processing does not become a liability to the system. Careful analysis of the system functions and careful choice of a free time algorithm will undoubtedly improve system performance.

2.7 Requirements of the Disc Drive

The disc drive to be interlaced to the HLDC is the CalComp CD1 disc drive. The requirements made on the HLDC by this disc drive are well defined and the hardware requirements can be easily determined. The CalComp CD1 disc drive requires that the HLDC transmit control signals and data to the disc, and receive control signals and data from the disc to perform the following disc drive operations :

(i) a disc seek,
(ii) a head select,
(iii) a data transfer to disc (write data),
(iv) a data transfer from disc (read data),
(v) the power up operation for disc,
(vi) the monitoring of error conditions.

The order that the signals are sent and received to perform any of the above

operations is fully specified by the disc drive, and the flowcharts specifying this order are given in the reference[17].

One of the functions which is usually performed by the low-level controller is that of initializing or formatting the disc. Every disc to be used on a system will first have to be formatted. The formatting routine will generally write header information onto each sector of the disc plus other necessary information. Formatting is discussed in further detail in Chapter 3.

2.8 Conclusion

The analysis of the filing system and disc drive functions has highlighted the processing requirements of the HLDC. The hardware structure necessary to implement these requirements is the subject of the next chapter.

References

1. Richard W. Watson, "Timesharing System Design Concepts", McGraw-Hill, Inc. (1970), pp. 187-189.
2. Ibid.
3. Ibid.
4. A.C. Shaw, "The Logical Design of Operating Systems", Prentic-Hall, Inc. (1974).
5. W.F.C. Purser and D.M. Jennings, "The Design of a Real-Time Operating System for a Minicomputer. Part 1", Software - Practice and Experience, Vol. 5 (1975), pp. 147-167.
6. Watson, pp. 216-218.
7. Ibid.
8. Watson, p. 215.
9. Purser, pp. 157-161.
10. Purser, pp. 147-167.
11. Ibid.
12. Ibid.
13. Watson, pp. 135-156.
14. Watson, pp. 137-143.
15. P.J. Denning, "Effects of Scheduling on File Memory Operations", AFIPS Conf. Proc., Spring Joint Computer Conf. Vol. 34 (1969), pp. 657-664.

16. T.J. Teorey, "Properties of Disc Scheduling Policies in Multiprogrammed Systems", AFIPS Cont. Aoc., Vol, 41, Part 1 (1972), pp. 1.12.

17. CD1 Disc Drive, CalComp Field Engineering Service Handbook, California Computer Products, Inc., (1971).

CHAPTER 3

THE HARDWARE STRUCTURE OF THE HIGH LEVEL DISC CONTROLLER

The discussion in Chapter 2 has indicated the types of functions that the High Level Disc Controller (HLDC) must perform, and a possible architecture for the HLDC will be developed in terms of this discussion.

3.1 The Essential Hardware Requirements of the High Level Disc Controller

The facilities required by the HLDC, based on the discussion in Chapter 2, can now be determined :

(i) Chapter 2 has highlighted the necessity for the HLDC to have access to a large amount of fast memory. This memory will be used for the manipulation of the file directory, the free space map, index blocks and the various system tables. The HLDC could conceivably use main memory. The HLDC would then require a large number of memory accesses. Whilst these memory accesses would by-pass the main processor (i.e. a direct memory access facility would be used), the main processor would be inhibited for a certain amount of time to perform each access (i.e. by cycle stealing)[1]. The time associated with the cycle stealing would interfere with the main processors "useful" processing functions. This, together with the main memory space which would have to be reserved for the filing system functions, makes the use of main memory unattractive.

Fast metal-oxide semiconductor (MOS) random access memory elements are, on the other hand, comparatively inexpensive, and the price of these memory elements continues to fall. This makes it feasible for the HLDC to have a large supply of its own (local) memory.

(ii) An interface between the HLDC and host computer is necessary. This interface must perform the following functions :

(a) The interface must be able to receive file commands from the host computer.

(b) The interface must be able to transmit the completion of commands to the host computer.

(c) The interface must provide fast data transfers between main memory and the disc drive with the minimum amount of interference to the main processor. Transfers will thus be performed by a direct memory access facility.

(iii) An interface between the HLDC and the disc drive is necessary to allow for the communication of data and control signals between these two units.

(iv) The HLDC must have a central processing unit (CPU) which must perform the following functions :

(a) The CPU must supervise and control the above three system elements (i.e. local memory, the disc drive interface and the host computer interface).

(b) The CPU must be capable of performing the file handling functions and the disc drive functions. These functions were discussed in Chapter 2.

To fulfil the above requirements, the CPU must be capable of performing simple arithmetic and have the ability to make logical decisions.

The implementation of the above structure must be done with certain basic design goals in mind. The system must be both fast and reliable, and should be able to adapt to the changing demands of the process control environment.

The central processing unit will now be discussed.

3.2 The Central Processing Unit

The central processing unit (CPU) could possibly be designed using one of the numerous 8-bit or 16-bit microprocessors. These microprocessors are comparatively slow, however, and would generally not be fast enough to fully execute certain critical program sequences necessary to handle

the data transfer rates of a high-speed backing store device[2,3]. In such a system, data transfers between backing store and the host computer would not be routed through the microprocessor. A possible hardware configuration using one of these standard microprocessors is illustrated in Figure 3.1.

Data transfers between the disc drive and host computer would occur via a high-speed data link. This data link would convert serial disc data to parallel data for the host computer. The data link would also have to control the direct memory access transfers to the host computer. Data from disc required by the CPU (i.e. the file directory, index blocks and the free space map) would be routed into local memory via this data link. Similarly, data from the host computer required by the CPU (i.e. the filename block) would also be routed via the data link, and into local memory.

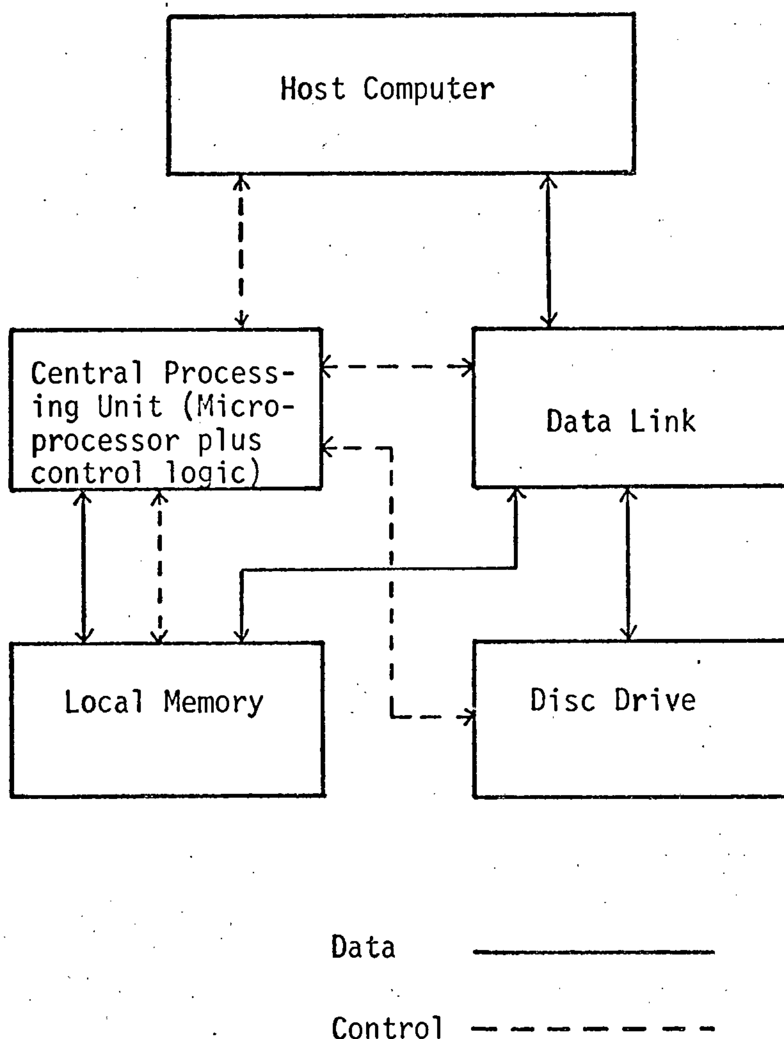


Figure 3.1 Possible HLDC Architecture - Using a Conventional Microprocessor.

The host computer would initiate commands by either routing the commands into local memory via the data link, or transferring the command directly to the CPU. The CPU could communicate the completion of a command in a similar fashion.

Whilst the above method has many desirable features, it has the following disadvantages.

(i) The standard microprocessor instruction set is not fully suited to perform the disc controller functions. Initiation of disc operations (e.g. seek, read and write), will therefore be lengthy;

(ii) Chapter 2 has indicated that a large amount of processing is required to gain maximum efficiency from the disc drive. The standard microprocessor, being comparatively slow, would not give good response.

The central processing unit could be designed from the vast range of medium- and large-scale integrated circuits. The resulting system design would be complex, however, and a large number of integrated circuit elements would be required. System reliability would also be poor, as system complexity will undoubtedly produce hardware bugs which may not be eliminated in the production machine. The cost of such a unit will be high, resulting from lengthy development time and high component count. Finally, such a system will allow little, if any, system flexibility. These factors make this approach unattractive.

The approach which is best suited to the HLDC application is to have a microprogrammed central processing unit. The microprogrammed system could be interfaced to hardwired logic to perform the appropriate HLDC functions. Such a design would again be complex.

The file handling and controller functions of a microprogrammed HLDC are most suitably performed by an array of fast bit-slice microprogrammable microprocessor elements. These bit-slice microprocessor elements can be connected together to form words of almost any length. Most bit-slice processors available allow high-speed operation and the processor cycle times are usually 100 nanoseconds or better. The resulting system will be easier to design and build than the hardwired system, and will be easily

modified to meet the changing requirements of the real-time computer environment. Modifications will now be made by reprogramming the system as opposed to redesigning and rebuilding the system's hardware.

The microprogrammed controller will have a definite speed advantage over any fixed instruction microprocessor. That is, customized microinstructions, which includes the ability to perform a number of operations in parallel, will greatly reduce the number of instructions required to perform a particular task. The reduced number of instructions required, together with the high-speed operation obtainable with a microprogrammed CPU, will produce good system response.

The HLDC's architecture can now be developed.

3.3 Architecture for the High Level Disc Controller

There are various architectures which could be developed for the HLDC. For example, the HLDC could have a hardware configuration similar to the configuration illustrated in Figure 3.1. Instead of a standard microprocessor forming part of the CPU, the CPU would now be microprogrammed. Another possible configuration would be to route all data through the bit-slice microprocessor. This configuration is illustrated in Figure 3.2. The main disadvantage of the system illustrated in Figure 3.1 is the amount of hardware it requires for implementation. (That is, the data link will require a large amount of logic to route data to and from the three memory elements, and the data link would be most efficiently implemented by a microprogrammable microprocessor). The advantage of this system is that file handling functions could be overlapped with data transfers.

The system illustrated in Figure 3.2 performs all the data routing. Whilst file system function cannot be overlapped with data transfers, this architecture will require fewer circuit elements and its implementation will be more easily realized.

It is hypothesized that in a system using a moving-head disc drive, most of the filing system functions which can be overlapped, will be performed during disc seeks. That is, the HLDC will spend much of its time waiting for seeks to be complete so that index blocks, and parts of the file

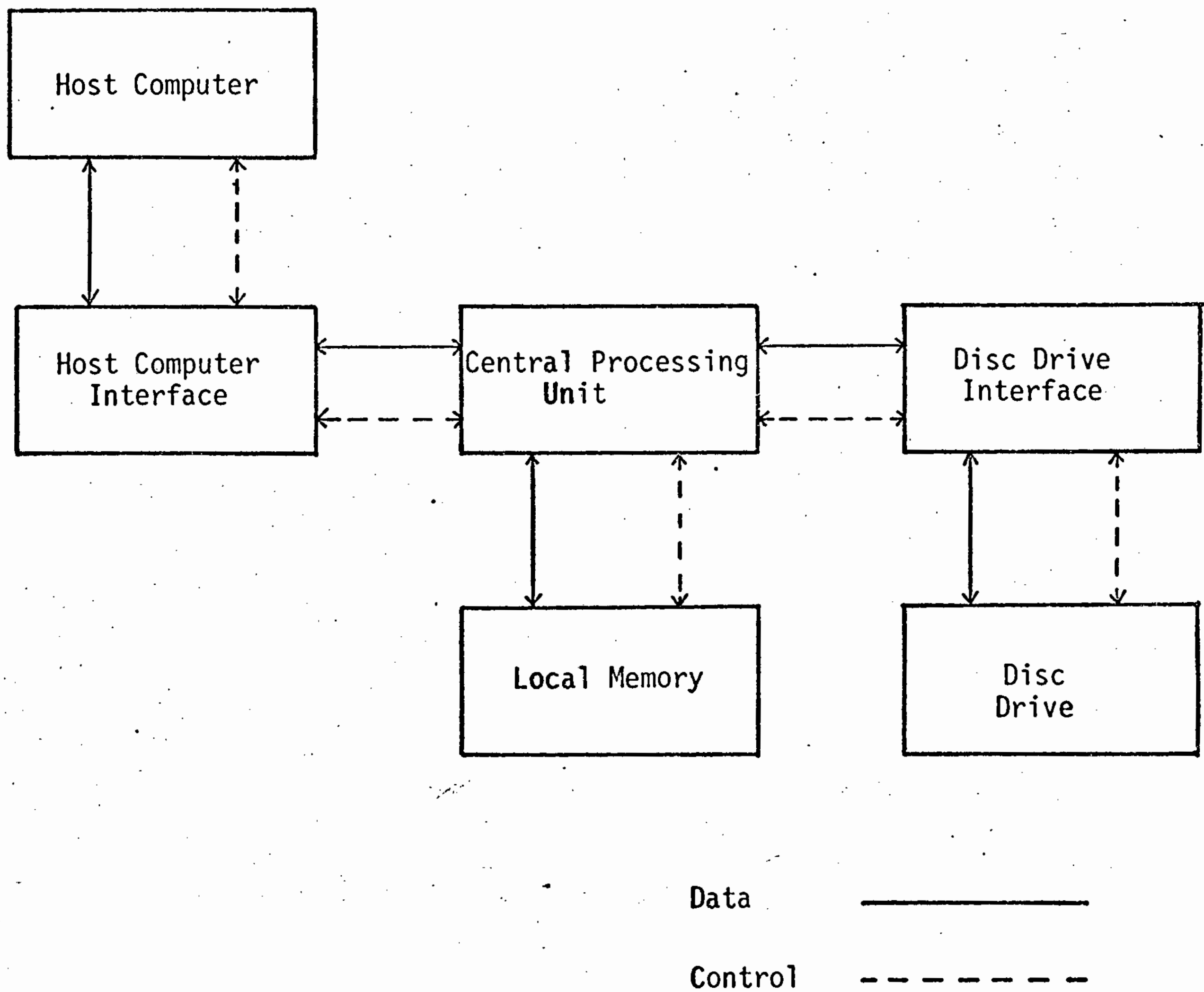


Figure 3.2 Possible HLDC Architecture - Using a Microprogrammed Microprocessor.

directory and free space map can be accessed. The architecture illustrated in Figure 3.2 is therefore chosen. (The above hypothesis may not be true for fixed head disc or drum, as processing may not be able to be fully overlapped with disc or drum rotations).

The HLDC's architecture centres around the CPU (refer Figure 3.2). The central processing unit will comprise of high-speed control memory (microprogram memory or micromemory) and a high-speed bit-slice micro-programmable microprocessor. The central processing unit will provide control signals to the external logic (i.e. local memory, the disc drive

interface and the computer interface), and will provide a central point for gating data to and from the external logic. The micromemory will supply control signals to the central processing array and the external logic.

The remainder of this chapter deals with the hardware structure of the HLDC. The circuit diagrams for the HLDC are given in Appendix E.

3.4 The Microinstruction Format

Since the microinstruction controls the entire system, the microinstruction format is of the most fundamental importance to the system design. The microinstruction must be capable of performing the following three functions in the system :

(i) The microinstruction must provide the control signals to the central processing array (CPA).
(ii) The microinstruction must provide the logic for selecting the next microinstruction's address.
(iii) The microinstruction must control the external logic (i.e. the local memory, disc drive interface and host computer interface).

The microinstruction format is formulated by analysing each of these three system functions in detail. To make the analysing of these functions clearer, the microinstruction format that has been obtained is given in Table 3.1, and the HLDC's architecture is given in Figure 3.3.

Before the three elements (or fields) are discussed in detail, it is necessary to discuss the need for a software development aid.

Microprogram Assembler

The software requirements for the HLDC will be extensive. It is conceivable that the microprogrammer could write the microprogrammes in machine code. In the current system, this would mean coding 35 bit microinstructions. The software for the HLDC would be difficult to write, as well as being difficult to modify and debug. Software tools are therefore necessary.

	Mnemonic	Field
0	MIXED0	MIXED
1	MIXED1	MIXED
2	MIXED2	MIXED
3	MIXED3	MIXED
4	MIXED4	MIXED
5	MIXED5	MIXED
6	MIXED6	MIXED
7	MIXED7	MIXED
8	MIXED8	MIXED
9	MIXED9	MIXED
10	MIXED10	MIXED
11	MIXED11	MIXED
12	F0	OPCODE
13	F1	OPCODE
14	F2	OPCODE
15	F3	OPCODE
16	F4	OPCODE
17	F5	OPCODE
18	F6	OPCODE
19	MADDR0	MUX
20	MADDR1	MUX
21	MADDR2	MUX
22	MADDR3	MUX
23	MADDR4	MUX
24	RE	MCU
25	S1	MCU
26	S0&	MCU
27	MC	MCU
28	FE	MCU
29	PUP	MCU
30	PORT-CONTROL0	PORT-CONTROL
31	PORT-CONTROL1	PORT-CONTROL
32	K-CONTROL0	K-CONTROL
33	K-CONTROL1	K-CONTROL
34	CI	

Next address selection field =
MCU + MUX + MIXED

Central processor array field =
OPCODE + K-CONTROL + CI + MIXED

External logic control field =
PORT-CONTROL + MIXED

Table 3.1 Microinstruction Format

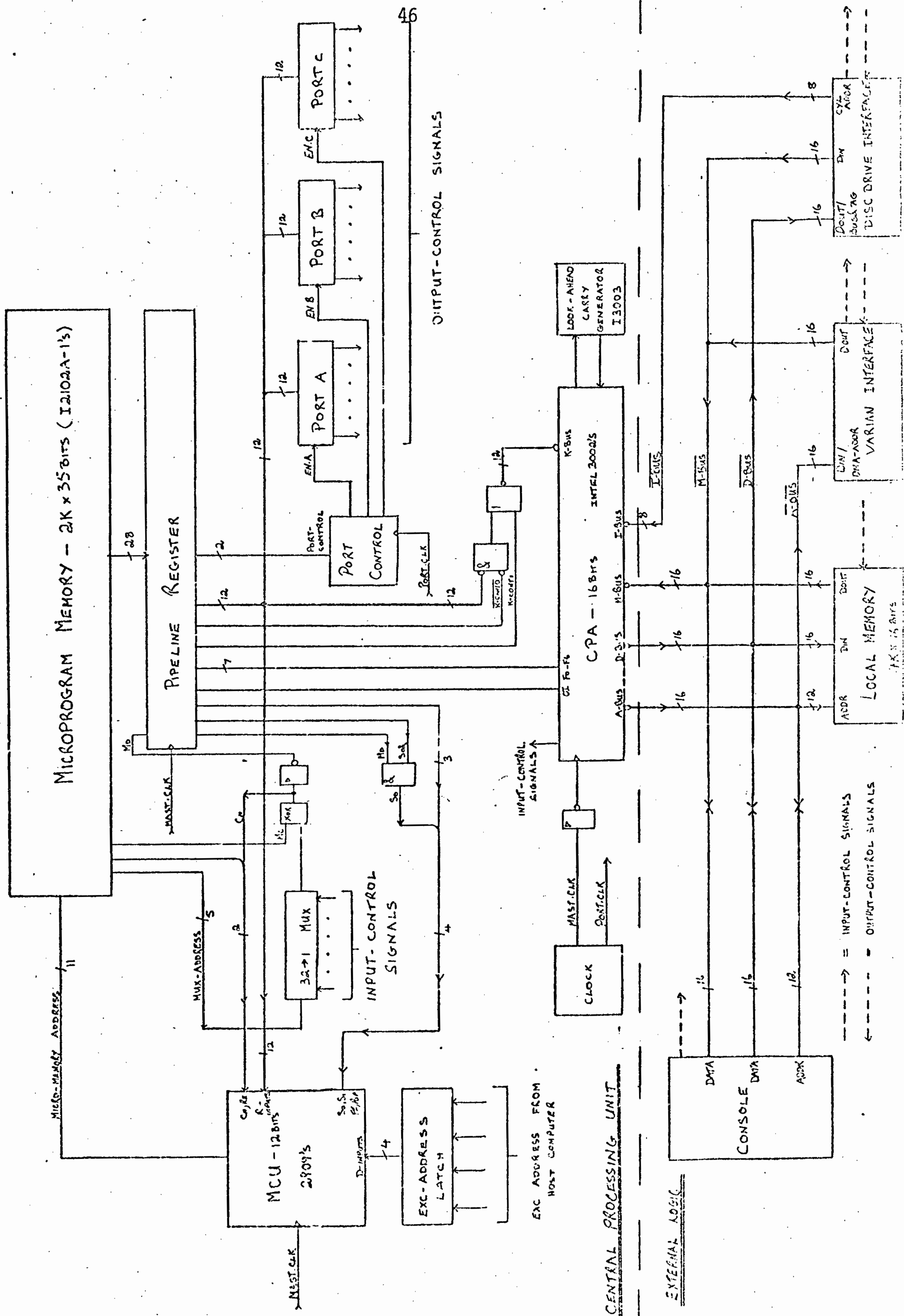


Figure 3.3 Block Diagram of the HLDC

The general software aid for such a system is an assembler. By the nature of microprogramming, each new architecture which is developed will have a different microinstruction format. Most manufacturers therefore offer assemblers which allow the user to define the microinstruction format and the requirements of each microinstruction field. The assembler can then be used by the user to assemble his 'symbolic' microprogrammes. This type of assembler is sometimes called a meta-assembler[4]. High level microprogramming languages, which use a principal similar to the meta-assembler, could also be used as a software design aid[5].

As no suitable software development aid was available when this project was initiated, it was necessary to design and implement a microprogram assembler for the HLDC. The microprogram assembler was developed to incorporate certain features.

(i) Although the HLDC is designed as an experimental machine, it is desirable to develop an assembler which has a certain amount of portability. This feature will allow firmware development to take place on many different computer systems. The assembler was therefore written in standard FORTRAN IV - if a compiler is available, it will generally be a FORTRAN compiler - and was designed to be as machine independent as possible. The assembler's portability is illustrated by the fact that it was developed on a Univac 1106 computer, and also runs on a Varian V77 minicomputer.

(ii) The assembler must be easy to modify. An experimental machine will often require change. This change must be reflected in the assembler. The assembler was therefore coded in a structured fashion (as structured as FORTRAN allows), to allow easy modification.

(iii) The assembler must make the microprogrammer's task as easy as possible. In this respect, the microprogram assembler is similar to any normal assembler. One point is worth mentioning. This is the choice of mnemonics for the various microinstruction fields. A mnemonic, to be useful to the microprogrammer, must convey as much meaning as possible about the function that is to be performed.

Good software documentation is another important programming aid.

A detailed description of the microprogram assembler is given in Appendix B. To emphasize certain hardware features developed, it is necessary to include microinstruction sequences in the text. All program sequences will be in symbolic code (refer Appendix B).

Each of the above three elements (or fields) of the microinstruction will now be discussed in detail.

3.5 The Central Processor Array Field of the Microinstruction

The central processor array (CPA) must perform the following functions.

(i) The CPA must provide a central point for gating data around the system on various system buses. The CPA must supply the local memory address and route data to and from local memory; the CPA must supply the main memory address for direct memory access and route data to and from the host computer; and the CPA must route data to and from the disc drive interface.

(ii) The CPA must be capable of performing the file handling and controller functions specified in Chapter 2. These functions require only simple arithmetic and logic. The CPA must also be able to control branching in micromemory. This will allow microprogram loops to be controlled, and will allow the branching on the result of a certain computation. The ability of the CPA to supply a signal on whether a particular register is zero or non-zero, will provide these capabilities.

3.5.1 Choice of Central Processor Array and Local Memory word sizes

There are two different word sizes which must be considered in the system. One is the processor word size and the other is the local memory word size. (Processor 'word' size is really a misnomer, as word size generally refers to memory word size. The processor word may be larger than the memory word to allow extra memory addressing capabilities). It would seem natural that the processor and memory word sizes be equal to the host computer's word size to allow for easy data transfers between controller and host computer. This is not necessarily true. If the host computer was to be

a 36-bit machine, then a 36-bit processor and a 36-bit memory would be required. All system busses would therefore be 36 bits. The interface to the disc drive would require a 36-bit shift register to allow serial to parallel, and parallel to serial conversions (refer Section 3.12). The hardware requirements of such a system would be large, and possibly unnecessary.

The word size is also a function of the disc speed. Data transfers between the CPA and the disc drive interface in an eight bit system will be eight bits at a time. When writing data to disc, for example, the disc drive interface will perform parallel to serial conversion of these eight bits by shifting one bit of data at a time out to disc. The CPU, in the meantime, must formulate the next data word to be written to disc. Eight bits may not allow the CPU sufficient time to formulate the next data word to be transferred to disc. Furthermore, an 8-bit or even a 12-bit processor interfaced to an 8-bit memory, will not give sufficient direct local memory addressing capabilities.

The choice of these two word sizes is therefore not straight forward, as the word sizes affect :

(i) the number of circuit elements required in the system;
(ii) the ease of the controller's interface to the disc drive and host computer; and
(iii) the amount of local memory which can be directly addressed.

The host computer to be interfaced to the HLDC in this system has a 16-bit word, and the most suitable word sizes, in this instance, is to have both processor and local memory words equal to 16 bits.

3.5.2 The Central Processor Array

The choice of components required for a specific system is dependent on a number of factors. These factors often prevent the ideal specifications of the system being met. The designer must consider local availability and support for the components he requires, as well as the ease at which these components will interface to the rest of the system. An evaluation

of the different bit-slice microprocessor families currently available can be found in the literature[6,7,8].

The Advanced Micro Devices 2900 series appears to be the choice amongst the 4-bit/slice processors. The Intel 3000 series has a 2-bit/slice processor. Appendix C.1 gives a comparison between the 2900 series and 3000 series central processing elements. The Intel 3000 central processing element, the Intel 3002 has been chosen.

The central processing array (CPA) is a 16-bit processor made up of eight 2-bit slice Intel 3002 central processing elements. Basically the CPA is capable of a variety of simple arithmetic and logic operations including logical shifting. A detailed discussion of the CPA is given in Appendix C.2.

The CPA field of the microinstruction is made up of four subfields as follows :

OPCODE

A 7-bit field is required to specify the microfunction to be performed by the CPA. This field is called the OPCODE field and occupies bits 12 to 18 of the microinstruction. These bits are designated F0 to F6 respectively.

CI

A one bit field is required to provide control over the least significant carry-in bit of the CPA. This will allow incrementing to be performed by the CPA. This field is called the CI field, and occupies bit 34 of the microinstruction.

K-CONTROL and MIXED

The CPA must be capable of receiving constants from micromemory for the following three reasons :

(i) Certain data will be placed in local memory at certain fixed locations. (For example, the free space map, the master file directory, and other important information will be placed in local memory starting at fixed locations). Since local memory is to be 4k (refer Section 3.10), twelve bit constants

are required to directly address this memory.

(ii) Constants will be required to set up loop counters.

(iii) Positive and negative constants will be required in certain arithmetic operations.

The range of constants required for these last two operations will be small, and the 12 bits required for addressing purposes will be adequate. Constants will be supplied by micromemory onto the CPA's K-bus.

Constants will only be required in a small percentage of microinstructions, and constants can therefore be multiplexed with the other data supplied by the (multiplexed) MIXED field of the microinstruction (refer Appendix C.3). Since the data on the MIXED field is multiplexed, two bits are required in the microinstruction to specify whether the K-bus is to have an all 0's or all 1's bit pattern, or whether the K-bus inputs are to come from the MIXED field of the microinstruction. This field is called the K-CONTROL (K-bus control) field, and occupies bits 32 and 33 of the microinstruction.

Table 3.2 summarizes the four possible K-control bit patterns, and the corresponding data to be placed on the K-bus.

K-CONTROL 1 (Bit 33)	K-CONTROL 0 (Bit 32)	FUNCTION PERFORMED
0	0	All K-bus inputs zero
1	0	All K-bus inputs one
1	1	All K-bus inputs one
0	1	K-bus inputs come from the MIXED field.

Table 3.2 K-CONTROL

Since the processor is 16 bits wide and the MIXED field only 12 bits wide, the twelfth bit of the MIXED field must be connected to bits 11 to 15 of the K-bus to give the correct magnitude to negative numbers. That is, negative numbers are represented in their 2's compliment form with the most significant bit of the CPA (bit 15) being the sign bit. The most significant bit of the MIXED field (bit 11) will therefore specify the sign for arithmetic operations. The range of numerical values that

can be supplied by the MIXED field for arithmetic operations is therefore between $-2^{11} = -2048$ and $2^{11} - 1 = 2047$.

Since the MIXED field is 12 bits, it allows direct addressing of $2^{12} = 4k$ of local memory.

3.6 The Next Address Selection Field of the Microinstruction

Each microinstruction must contain information to select the next microinstruction address. This information is supplied by a field which can be called the next address selection field or, using conventional terminology, the jump field. The microprogrammer, to implement requirements of the HLDC, requires the ability to perform conditional and unconditional jumps.

(i) Conditional jumps. The selection of the next microinstruction may be dependant on certain signals from the external logic, or on signals indicating the results of computations performed by the central processor array. Examples of these signals include a signal from the disc drive indicating the completion of a seek, or a signal from the central processor array indicating whether a particular register is zero or non-zero. These signals can be called the input-control signals, as they influence the order in which microinstructions are to be executed. A jump which is dependant on any input-control signal can be called a conditional jump.

(ii) Unconditional jumps. A jump which is independant of any input-control signal can be called an unconditional jump.

The method used in selecting the next microprogram address is to use a microprogram control unit (or microprogram sequencer). The microinstruction will control the microprogram control unit (MCU) and the MCU will supply the address of the next microinstruction to be executed to micromemory. There are certain features which are desirable in an MCU :

(i) The MCU must allow the current microinstruction to specify any other microinstruction in micromemory as being the next microinstruction to be executed. This feature is not supplied

with all MCUs. A notable exception is the Intel 3000 series MCU, the 3001[9]. The advantages of this feature are :

(a) it allows the microprogram assembler to be easily constructed. That is, the microinstruction placement problems which occur with the Intel 3001 will not occur with an MCU having the above feature; and

(b) wasted micromemory space will not occur. Wasted micromemory space occurs with a microsequencer such as the Intel 3001 because certain areas of memory become inaccessible[10].

In this instance, the full jump address will be supplied to the MCU from the jump field of the microinstruction.

(ii) The MCU must have the ability to step through micromemory one microinstruction at a time without having to supply the next microinstruction address. This requires that the MCU have a microprogram counter which can be incremented either automatically or under microprogram control. This feature has two advantages :

(a) it will free that part of the microinstruction field required for supplying the microinstruction address for other operations; and

(b) with the MCU under microprogram control, the microprogram may easily perform a conditional jump. One arm of the jump is the next microinstruction address supplied by the MCU. The other arm of the jump is an address supplied by the microinstruction.

(iii) Subroutining is as desirable a feature when writing microprograms as it is when writing ordinary computer programs. The ability to easily implement subroutines is therefore desirable. This requires that the MCU be able to store the subroutines return address. To allow subroutine nesting, the MCU should have a stack to store the return addresses. Ideally this stack should be large enough to enable the programmer to be reasonably unrestricted in the number of subroutine nestings he requires. The minimum stack should be able to store in the region of ten return addresses.

3.6.1 The Microprogram Control Unit

It would seem that since the Intel 3000 series central processing elements are used, the logical choice for the microprogram control unit (MCU) would be the 3000 series MCU, the 3001. Some of the problems encountered when using the 3001 have been discussed above, and these problems are further discussed in the literature[11,12]. An evaluation of the different microprogram control units currently available can also be found in the literature[13,14].

The Advanced Micro Devices Inc. 2900 series microprogram control unit, the 2909, fulfills the requirements of the MCU specified above, and this is the microprogram sequencer chosen. Part of the jump field of the microinstruction will control the 2909. This sub-field can be called the MCU-control field.

The 2909[15] will now be discussed in detail.

Each 2909 is a 4-bit wide address controller, and the devices are cascadable so that two devices allow addressing up to 256 words of microprogram memory. Three of these 2909's are used, and microprogram memory can therefore be easily expanded to 4k. The block diagram of the 2909 MCU is illustrated in Figure 3.4.

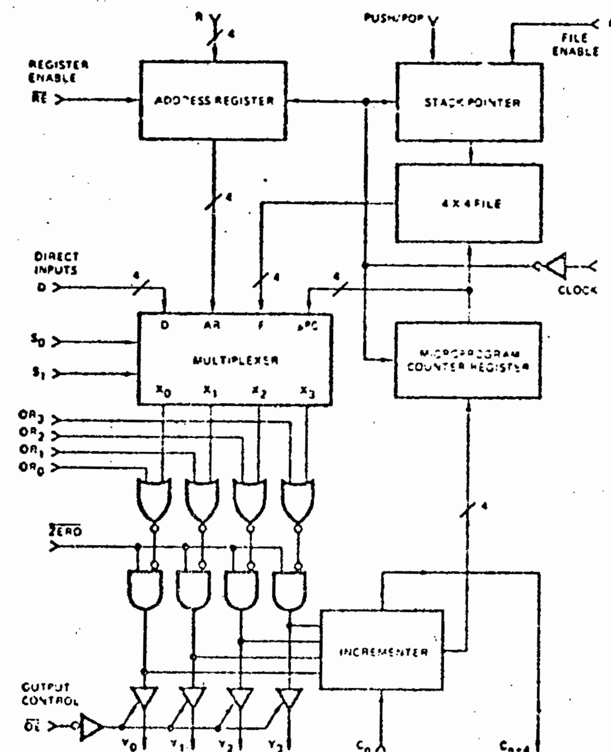


Figure 3.4 Block Diagram of the 2909 Microprogram Control Unit

Each 2909 contains a four-input multiplexer that is used to select either the address register, direct inputs, microprogram counter, or file as the source of the next microinstruction address. The multiplexer is controlled by the S0 and S1 inputs (refer Table 3.3).

Address Selection

OCTAL	S_1	S_0	SOURCE FOR Y OUTPUTS	SYMBOL
0	L	L	Microprogram Counter	μPC
1	L	H	Address register	AR
2	H	L	Push-Pop stack	STK0
3	H	H	Direct inputs	D_i

Output Control

OR_i	$\overline{ZERO}$	$\overline{OE}$	Y_i
X	X	H	Z
X	L	L	L
H	H	L	H
L	H	L	Source selected by S_0 S_1

Z = High Impedance

Synchronous Stack Control

$\overline{FE}$	PUP	PUSH-POP STACK CHANGE
H	X	No change
L	H	Increment stack pointer, then push current PC onto STK0
L	L	Pop stack (decrement stack pointer)

Table 3.3 2909 Multiplexer Select Codes

The 2909 contains a microprogram counter which is composed of a 4-bit incrementer followed by a 4-bit register (microprogram register). The incrementer has a carry-in and carry-out such that MCU's can be easily cascaded together using the ripple carry configuration. The maximum delay time from carry-in to carry-out for one 2909 is 18 nanoseconds, hence the maximum propogation delay time across the three 2909's is 54 nanoseconds. The least significant carry-in to the MCU array is under microprogram control, allowing the microprogram counter to be used in one of two ways. When this least significant carry-in is HIGH, the microprogram register is loaded on the next clock cycle with the current microinstruction address plus one. If this least significant carry-in is LOW, the current microinstruction address is not incremented, and

the microprogram register is loaded with the same microinstruction address on the next clock cycle. Thus the same microinstruction can be executed any number of times. The latter configuration is used in synchronizing the controller with signals from either the disc drive or host computer. When the appropriate signal goes TRUE (or FALSE), thus the carry-in will go HIGH.

The address register (AR) of each 2909 consists of four D-type, edge-triggered latches with a common clock enable. New data is entered into the address register from the external R-inputs on the clock LOW-to-HIGH transition. The R-inputs of the MCU are connected to the MIXED field of the microinstruction. (The reason for this will be discussed in Section 3.6.3) When a branch is required in the microprogram, the address register will be loaded from the MIXED field, and the address register will then be selected as the next microinstruction address.

The HLDC can receive 16 different commands from the host computer. Physically, a command arrives as a 4-bit address on a bus called the EXC-address bus. Four of the D-inputs of the MCU are connected to the four EXC-address lines from the host computer (as illustrated in Figure 3.3). The EXC-address bus allows the host computer to select 1 of 16 possible functions. The least significant D-input line is tied HIGH, the next four D-input lines are tied to the EXC-address latch, whilst the remaining 7 lines are tied LOW. When the D-inputs are selected as the next micro-instruction address, a jump will be made to an odd location between 0 and 31 depending on the EXC address. That is, a jump will be made to the microprogram memory, 1, 3, 5, ..., 29 or 31, corresponding to an EXC address 0, 1, 2, ..., 14 or 15 respectively.

The last source available to the multiplexer is a four word deep pop/push file (stack). The stack is used to provide the return address when a subroutine jump is made. The file contains a built in stack pointer (SP) which always points to the last file word written (called STKO). The stack pointer operates as an up/down counter with separate pop/push (PUP) and file-enable (FE) inputs. The file operations performed by pop/push and file-enable are summarized in Table 3.3. Since the stack is four words deep, only four micro-subroutines can be nested. This is the only severe restriction of the 2909.

The jump field of the microinstruction can now be determined. The jump field can be divided into three sub-fields as follows :

(i) The input-control signals select field;
(ii) the microinstruction address field; and
(iii) the MCU-control field.

These three sub-fields will now be discussed in detail.

3.6.2 The Input-control Signal Select Field

As mentioned above, conditional jumps are dependant on certain input-control signals. It is necessary to provide a means of monitoring these signals and to allow the selection of the next microinstruction to be based on these signals. There are twenty input-control signals in the HLDC which the microinstruction must be capable of monitoring. The microinstruction must select the appropriate input-control signals and use the state of these signals to modify the output of the MCU-control field, and so cause the appropriate next microinstruction address to be selected. Each of the input-control signals could be masked by a bit in the microinstruction. Twenty signals would require 20 microinstruction bits to perform the masking. The required branch in the microprogram logic would then be performed by the outcome of a logical operation performed on the twenty microinstruction bits and the 20 input-control signals. Whilst this method may be necessary in a system requiring the simultaneous examination of a large number of different inputs, it is not necessary in the HLDC which requires the examination of only one input-control signal at a time. The method used is to input the twenty signals to a 32 to 1 multiplexer. The output of the multiplexer is gated with certain of the MCU-control bits of the microinstruction (refer Section 3.6.4) to provide the desired address being selected by the MCU. The appropriate signal to be selected by the 32 to 1 multiplexer is obtained by supplying a five bit address to the multiplexer. These five bits are supplied by the microinstruction. This five bit field can be called the input-control signal select field or the multiplexer control field (MUX field).

The MUX field occupies bits 19 to 24 of the microinstruction. Inputs to the multiplexer are named MUX0, MUX1, ..., MUX31. The five bit multiplexer address selects the appropriate input as follows: address 0 selects MUX0, address 1 selects MUX1, and so on. MUX0 is tied LOW, and MUX1 is tied HIGH, the remaining multiplexer inputs, MUX2 to MUX31 being tied to the appropriate input-control signal. MUX0 and MUX1 are used in unconditional jumps, and MUX2 to MUX31 are used for conditional jumps. Table B.6 in Appendix B summarizes the twenty multiplexer input-control signals with their corresponding mnemonics. Only 22 of the 32 multiplexer lines are currently used. Twenty lines are used for the input-control signals, and two lines are used for unconditional jumps. Consequently the HLDC can accommodate ten more input-control signals with only a minimal amount of hardware modification.

3.6.3 The Microinstruction Address Field

As mentioned above, the microinstruction must be able to supply a jump address to the MCU. Micromemory is expandable up to 4k, and therefore twelve bits are required for the jump address. The next microinstruction to be executed will generally be the microinstruction in the following micromemory location. In this situation, the twelve bit jump address field will not be required, as the next address will be supplied from the MCU's microprogram counter. The jump address can therefore be placed in the twelve bit (multiplexed) MIXED field of the microinstruction.

3.6.4 The MCU-Control Field

Whilst the ability to perform only two jumps, one conditional and the other unconditional, will adequately satisfy the HLDCs needs, it will not allow easy and efficient programming. Consequently, to allow for easy and efficient programming, eleven jumps have been formulated. These jumps are discussed below.

Conditional jumps

The CPU must be able to syncronize itself with the external logic. For example, the CPU may have to wait for the falling edge of the index pulse before continuing with a particular program. The best way to achieve this

syncronization is to force the MCU to continuously fetch and execute the microinstruction which tests this condition. As soon as the condition goes TRUE (or FALSE), the MCU will fetch the next microinstruction, causing syncronization to occur. The two mnemonics associated with this type of conditional jump are INCT (increment if TRUE) and INCF (increment if FALSE). The following example illustrates the use of this jump. Assume a program is required to syncronize itself with the falling edge of the index pulse. The index input-control signal is selected by the multiplexer address INDEX (refer Table B.6 in Appendix B). The following program sequence would cause the required syncronization :

```
*    INCT    INDEX    #    Wait for the index pulse if it has not arrived.
*    INCF    INDEX    #    Wait for the falling edge of the index pulse.
```

Another two conditional jumps are useful :

(i) If condition TRUE, select the information in the MIXED field as the next microinstruction address. If condition FALSE select the microprogram counter as the next microinstruction address. The mnemonic associated with this conditional jump is JMPT.

(ii) If condition FALSE jump to the address in the MIXED field. If condition TRUE jump to the microprogram counter address. The mnemonic associated with this conditional jump is JMPF.

For example, to test if the drive is unsafe, the following program sequence may be used (the unsafe condition from disc is examined by an UNSAFE in the MUX field) :

```
*    JMPT    UNSAFE    ERROR
```

<u>Unconditional jumps</u>

There are seven unconditional jumps which have been formulated. A brief description of each of these jumps with an associated jump mnemonic is given below :

JNC - fetch the next microinstruction.

JMP - jump to the address in the MIXED field.

JMPD - jump to the address specified by the host computer. This jump was elaborated in Section 3.6.1 above.

JSR - jump to subroutine. The subroutine address being specified in the MIXED field.

JSRD - jump to subroutine. The subroutine address being specified by the host computer.

RTS - return from subroutine.

HLT - continuously fetch and execute the current microinstruction.

Note that only four jumps JMPT, JMPF, JMP and JSR use the MIXED field of the microinstruction. This field remains free when using any of the other jumps.

The format of the MCU-control field can now be determined. As there are only eleven jumps in the microinstruction, only four bits are required to perform the appropriate jump. If only four bits are used in the MCU-control field, jumps would then be in an encoded form, and each jump would have to be decoded to allow the appropriate jump to be performed. The extra logic required for decoding makes this method unattractive. The jumps can be performed by using six bits in the microinstruction field with a minimum amount of extra logic. Consequently the latter method is used. The six MCU-control bits are designated PUP, FE, MC, S0&, S1, and RE, and occupy bits 29 to 24 of the microinstruction respectively.

The signals PUP, FE, S1, and RE are connected directly to the MCU through the pipeline register. MC and S0& are gated with the output of the multiplexer to obtain the MCU signals Cn and S0. The next address selection logic is illustrated in Figure 3.3. Table 3.4 shows how the appropriate jumps are formed by the six microinstruction bits.

3.7 The Control of External Logic

The signals for controlling the external logic (i.e. local memory, disc drive interface and computer interface) are derived from micromemory. These signals can be called output-control signals. There are 26 output-control signals which are required in the current system, and they are summarized in Table B.7 in Appendix B. Each of the 26 signals could be controlled by

MNEMONIC	DESCRIPTION	PUP	FE	MC	SO&	S1	RE	MUX
INC	Increment μPC. Select μPC as next μI address	0	0	1	0	0	0	ONE
JMP	Load AR and select AR as next μI address	0	0	0	1	0	1	ONE
JMPD	Select D-inputs as next μI address	0	0	0	1	1	0	ONE
JSR	Increment μPC. Increment stack pointer and push μPC onto stack (STKO). Load AR and select AR as next μI address	1	1	0	1	0	1	ONE
JSRD	Increment μPC. Increment stack pointer and push μPC onto Stack (STKO). Select D-input as next μI address	1	1	0	1	1	0	ONE
RTS	Jump to STKO. Decrement stack pointer (ie. POP stack)	0	1	1	0	1	0	ONE
HLT	Select μPC as next μI address (Note: Do not increment μPC)	0	0	1	0	0	0	ZERO
INCT	If condition TRUE increment μPC. Select μPC as next μI address	0	0	1	0	0	0	ADDR
INCF	If condition FALSE increment μPC. Select μPC as next μI address	0	0	0	0	0	0	ADDR
JMPT	Load AR. If condition TRUE select AR as next μI address. If condition FALSE increment μPC and select μPC as next address	0	0	0	1	0	1	ADDR
JMPF	Load AR. If condition FALSE select AR as next μI address. If condition TRUE increment μPC and select μPC as the next μI address	0	0	1	1	0	1	ADDR

Notes: 1. μPC = microprogram counter, μI = microinstruction, and AR = address register.

2. The MUX address ZERO selects a LOW, and the MUX address ONE selects a HIGH. ADDR selects the appropriate multiplexer input condition.

Table 3.4 Jumps

a separate microinstruction bit. Whilst this has the advantage of allowing any of the 26 signals to be active simultaneously, it is expensive in terms of the logic it requires (i.e. fast micromemory is expensive), and is unnecessary in the HLDC. Usually only one output-control signal is required in any one microinstruction. When more than one signal is required, the signals will be related. For example, a signal for controlling the disc interface will never be required with a signal controlling the host computer interface. This field can consequently be multiplexed as follows. Only 12 bits will be used for controlling the external logic. Each bit will provide three possible output-control signals. The appropriate output-control signal being selected by two other bits in the microinstruction. That is, there are three output ports. Each port supplies a maximum of 12 different signals to the external logic. The ports are named PORT A, PORT B and PORT C. The signals to the external logic will only be required in a small percentage of microinstructions. These signals can therefore be multiplexed with the other data in the MIXED field. The two bits that select the appropriate port are called the port control bits, and occupy bits 30 and 31 of the microinstruction. (Bit 30 is called PORT.CONTROL0, and Bit 31 is called PORT.CONTROL1).

The port control bits enable the appropriate port by causing one of the signals EN.A, EN.B and EN.C to go HIGH (refer Table 3.5). These three signals are AND'ed with each of the bits in the MIXED field to form 36 different port signals (output-control signals). Since only 26 of these output-control signals are currently used, the remaining ten signals allow for system expansion.

PORT.CONTROL1	PORT.CONTROL0	PORT SELECTED	SIGNAL THAT GOES HIGH
0	0	NONE	NONE
0	1	PORT A	EN.A
1	0	PORT B	EN.B
1	1	PORT C	EN.C

Table 3.5 Port Control

3.8 Microinstruction Cycle Time

The microinstruction cycle time directly affects system performance - the faster the HLDCs clock, the better the response of the backing store device. Consequently, factors affecting the microinstruction cycle time will be discussed in detail.

3.8.1 Microprogram Memory

The microprogram memory (micromemory) contains the microinstructions. Each microinstruction is 35 bits wide, and therefore micromemory is a 35 bit wide memory. The amount of micromemory required by the HLDC will depend on the sophistication of the filing system. The system currently has 1k of micromemory. The maximum amount of micromemory which can be supported is 4k. As the current system is designed as an experimental machine, random access memory (RAM) has been used. RAM allows microprograms to be modified. This is essential for hardware and software development.

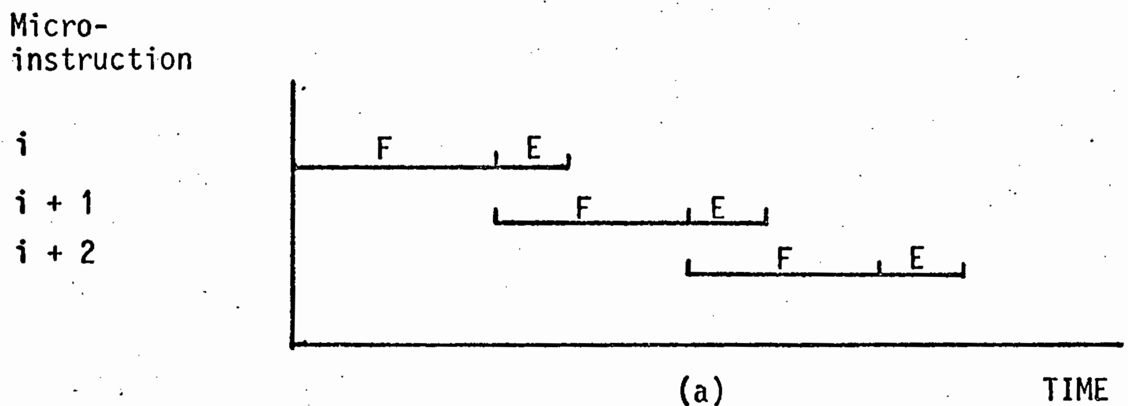
The access time of micromemory will directly affect the microinstruction cycle time. Consequently the system should be designed using high-speed bipolar RAM (such as the Intel 3106A with an access time of 60 nanoseconds[16]). High-speed bipolar RAM is expensive, however, and uses a great deal of power. Metal-oxide semiconductor (MOS) memory components are cheaper than bipolar memories, but are considerably slower. The cheaper MOS memories are suitable for the experimental machine. The memory elements used are the Intel 2102A-1 MOS memory elements with a 200 nanosecond access time. The production machine would use high-speed bipolar read-only or random-access memory.

3.8.2 Pipelined Architecture

One method of improving the microinstruction cycle time is by overlapping the execution of the current microinstruction with the fetching of the next microinstruction. This is called pipelining. Figure 3.5(a) illustrates how the pipelined architecture will help improve system performance. In

a non-pipelined architecture, the fetching of the next microinstruction to be executed does not begin until the execution of the current microinstruction is complete. This is illustrated in Figure 3.5(b).

The pipeline architecture is implemented by placing edge-triggered D-type latches between the micromemory outputs and the external circuitry. These latches are together called the pipeline register. The pipeline register holds the present microinstruction whilst the next microinstruction is being fetched. Certain bits of the microinstruction cannot be input to the pipeline register. These are the bits of the microinstruction used in calculating the next microinstruction, and which are clocked by the MCU. The bits used in calculating the next microinstruction which are not clocked by the MCU (e.g. S0 and S1) must go through the pipeline register.



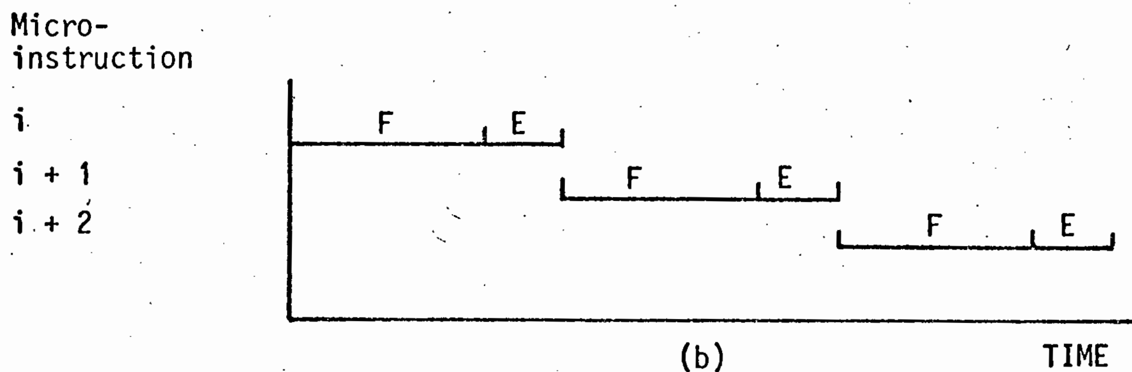


Figure 3.5 (a) Pipelined Architecture (Parallel Fetch and Execute)
(b) Non-Pipelined Architecture (Serial Fetch and Execute)

There are two disadvantages in using pipelining. The first disadvantage is that the hardware is more complicated in a pipelined system. The second disadvantage is that a conditional jump, using an output from the CPA, cannot be done in one microinstruction. That is, the CPA cannot set up a condition in the same microinstruction that this condition is to be tested. For example, if it is necessary to test whether a particular register (say register R6) is zero or non-zero, two microinstructions are required as follows :

```
i     : TZR(R6)
i + 1 : NOP     JMPT     CO    AGAIN
```

Whilst the input-control signal CO is being generated by microinstruction i, the next microinstruction i + 1 is being fetched. This problem only occurs with the two synchronous input-control signals CO and SIGNBIT.

3.8.3 Calculation of the Microinstruction Cycle Time

The most commonly used method of calculating cycle times is by increasing the clock rate until the system no longer works. Whilst this method may be useful in the production stage of the machine, an analysis of propogation delays in the system which determine the microinstruction cycle time is important for two reasons :

(i) The HLDC will not always be operating in a non-hostile environment. Calculation of typical and maximum microinstruction cycle times will, therefore, give the user some indication of the system performance to be expected in a real-world situation.

(ii) The analysis will highlight bottlenecks in the system. This is essential in a prototype system, since system improvements can then be made, if possible, resulting in better microinstruction cycle times, and hence better system response.

The microinstruction cycle time is determined by examining the propogation delays in the system associated with the fetch and execution of each microinstruction.

(i) The next-address selection field of the microinstruction determines the next microinstruction address. Consequently the information in this field must be used in every microinstruction. The delays associated with the next-address selection field determine the microinstruction fetch propogation delay. A detailed analysis of the microinstruction fetch propogation delay is given in Appendix C.4. This analysis shows that the maximum fetch propogation delay, t_{FM}, is 355 nanoseconds, and the typical fetch propogation delay, t_{FT}, is 265 nanoseconds.

(ii) The central processor array and the external logic control fields perform the microinstruction execution. The external logic control field will only be required in a small percentage of microinstructions. Consequently, the delay associated with the execution of this field is not considered when calculating the microinstruction execution delay, and is analysed separately. If an output-control signal is required to be present for longer than one microinstruction cycle, this will be done by either triggering a monostable or by using two (or more) microinstructions. The microinstruction execution time is therefore determined by the CPA field. The microinstruction execution propogation delay is analysed in Appendix C.5. This analysis shows that the maximum execution propogation delay, t_{EM}, is 150 nanoseconds, and the typical execution propogation delay, t_{ET}, is 100 nanoseconds.

As the HLDC has a pipelined architecture, the microinstruction cycle time is the maximum of the microinstruction fetch and the microinstruction execution propogation delays. In the present system, this will give a maximum microinstruction cycle time of 355 nanoseconds, and a typical microinstruction cycle time of 265 nanoseconds.

The microinstruction cycle time chosen will depend on environmental factors. Since the HLDC is currently situated in a non-hostile environment, the propogation delay time will be in the region of t_{FT}, and the system can successfully run at this clock rate. A clock period of 280 nanoseconds has been chosen for the analysis of the system, as this will be a more realistic figure to ensure stable operation. The clock (MAST.CLK) HIGH time is set to 70 nanoseconds, and the clock LOW time is set to 210 nanoseconds.

3.8.4 Improvements in Microinstruction Cycle Time

Having located the maximum propogation delay path in the system, system

improvements can be proposed. It must be realized that by improving one path of delay in the system, another path may become the most significant delay path.

The most significant factor in the microinstruction cycle time is the speed of microprogram memory. This is to be expected, and improvements to system performance will most naturally be achieved by using faster memory. For example, by replacing the Intel 2102A-1 MOS memory elements used in the system by the faster Intel 3106A bipolar memory elements with a maximum access time of 60 nanoseconds, the microinstruction cycle time will be reduced to :

t_{FM} = 210 nanoseconds
t_{FT} = 165 nanoseconds.

These times are still greater than the microinstruction execution cycle time. Setting the microinstruction cycle time to 190 nanoseconds results in a 30% improvement in cycle time, and consequently better system performance. This is the microinstruction cycle time which could be expected in a production machine using high-speed read-only memory.

3.9 System Timing

3.9.1 Glitches

It is necessary to ensure that glitches (spikes), which will inevitably occur in the system, will be rendered harmless. To ascertain whether or not any glitches could be harmful, it is necessary to look at the timing associated with the three microinstruction fields.

(i) The next address selection field. The data in this field is initially clocked into the MCU or pipeline register. The data will only be clocked into the MCU and pipeline register when the data is again stable at the inputs of these two elements. Consequently no timing problems will be associated with this field.

(ii) The central processor array field. Microfunctions are supplied to the CPA on one clock edge, and the data is gated into the CPA registers on a different clock edge. Glitches will not be of sufficient duration to cause any timing problems.

(iii) The external logic control field. The port data and port control bits are logically AND'ed together to form the output-control signals. These output-control signals directly control the

external logic. That is, these signals will set and clear latches, will gate data onto the M-bus and will provide the write signal for local memory. If the output-control signals are not clocked, propogational delays in the system may cause unwanted output-control signals. This problem is analysed in detail in Appendix C.5. The resulting solution is to have a clock (called PORT.CLK) which is out of phase with the systems master clock (MAST.CLK). PORT.CLK clocks certain of the output-control signals and thus prevents glitches from occurring.

3.9.2 Critical Timing Paths

It is essential to examine all the possible time-critical paths in the system to ensure that no timing problems will occur. The time-critical paths associated with the external logic will be discussed below in the relevant sections on the external logic.

The remainder of this chapter deals with the external logic.

3.10 Local Memory

As mentioned in Chapter 2, a large amount of local memory is required. The amount of memory required is dependant on the process control environment, and the type of response required. Basically, the more local memory available, the better the system response will be (refer Chapter 2). Since a twelve bit constant can be gated into the CPA from micromemory, 4k of local memory is directly addressable.

The HLDC has been provided with 4k of 16 bit memory. Since local memory accesses are comparatively infrequent, fast local memory is unnecessary. The memory elements used can therefore be the Intel 2102A-2 static random access memories, with a 250 nanosecond access time[17].

An analysis of local memory timing is essential for programming purposes, and a detailed analysis is given in Appendix C.7. This analysis is only true in a system which has a microinstruction cycle time of 280 nanoseconds, and uses the Intel 2102A-2 memory elements for the local memory. If the microinstruction cycle time was to be improved, local memory timing would have to be reanalysed.

3.11 The Computer Interface

The computer interfaced to the HLDC is the Varian 620 minicomputer. The input/output (I/O) facilities provided by the Varian minicomputer allow three basic modes of operation[18]. The HLDC/Varian interface has been provided with these modes of operation, and Appendix D.5 discusses how the input-control and output-control signals are used to perform each operation. These operations are first discussed below, and then it will be shown how communication between the Varian and HLDC could occur.

(i) I/O Instructions. I/O instructions are executed by the Varian's main processor. Each I/O instruction has associated with it a device address. Each peripheral controller attached to the system has one or more device addresses. The HLDC has one device address. This is address 14_8 (octal 14). Each I/O instruction directed to the HLDC must therefore specify this address. There are four types of I/O instructions, and these will now be discussed.

(a) External Control. The external control instructions (EXC and EXCE) can be used to specify a maximum of 16 different modes of operation. For example, the external control command could be used to instruct the HLDC to perform a particular file operation (i.e. read, write, create, delete, open or close).

(b) Program Sense. The program sense instruction (SEN) is used to test the status of a specific device condition, and, if a true condition, a program jump is made. If a false condition is detected, the next instruction in the sequence is executed. For example, the sense instruction could be used to test whether the HLDC is ready to receive a command; whether the HLDC is out of action (possibly through a disc fault); etc.

Each device address has associated with it a maximum of eight sense lines. A sense instruction specifies the device address plus the sense line it is testing. The HLDC currently supplies three sense lines to the Varian,

but this can easily be expanded to eight if required.

(c) Single-word input transfer. There are five instructions which provide single-word input transfers[19]. The single-word input transfer instructions are used to input data from the peripheral controller. For example, the HLDC could set up its status in a register. The data in this register could then be transferred to the Varian by one of the single-word input transfer instructions.

(d) Single-word Output Transfer. There are three single-word output transfer instructions[20]. A single-word output transfer instruction could be used for transferring a main memory address to the HLDC. The HLDC could then use this address for a DMA transfer.

(ii) Cycle-Stealing I/O. Cycle-stealing I/O allows a word of data to be transferred directly between main memory and the peripheral controller by means of a direct-memory-access (DMA) feature. In the Varian, each data word transferred by DMA will inhibit the main processor for a total of 3,15 micro-seconds[21]. This is the time required to transfer a word of data directly between main memory and the peripheral device. DMA allows data transfer rates of up to 202 000 words per second[22].

(iii) Interrupts. An interrupt from the HLDC will force the currently executing program in the Varian to branch to a memory-address location specified by the HLDC. For example, on the completion of a file operation, the HLDC could issue an interrupt to the Varian.

3.11.1 Communication between the HLDC and the Varian

The functions to be performed by the HLDC/Varian interface were outlined in Section 3.1. There are three functions to be performed, and the following discussion indicates various ways these functions may be performed.

Data Transfers

It is conceivable that data transfers between Varian and HLDC could take place using the various I/O instructions. This method of communication is not practical for two reasons. Firstly, this method would result in poor data transfer rates. Transfer rates using this method would typically be 30 000 words per second, as opposed to data transfer rates of up to 202 000 words per second via DMA[23]. Secondly, main processor utilization would suffer, as data transfers would be directed through the main processor. Transfers will therefore occur via DMA. DMA transfers occur by trap-in and trap-out requests[24]. These requests can either be supplied directly from the HLDC or from an I/O option on the Varian called the Buffer Interlace Controller (BIC). The advantage of using the BIC is that the hardware interface will be easily accomplished. The disadvantage in using the BIC is that the DMA start and end addresses must be set up in the BIC under program control. This is unsuitable for the HLDC. In the multiprogramming environment, the HLDC will be servicing a large number of different requests. Each request will require data transfers to or from different areas in main memory. Consequently, setting up the currently desired address in the BIC would disrupt the main processor. A better method is therefore to initiate the trap-in and trap-out requests directly from the HLDC.

File Command Initiation

There are many ways in which file commands can be transferred to the HLDC :

(i) One method of implementing command transfers is to supply the HLDC with an interrupt structure. Such a structure will have many desirable features, but will also have certain disadvantages. In the HLDC, an interrupt structure would complicate both hardware and firmware. Interrupts will also reduce system reliability[25]. If interrupts were to be used, the interrupts would not be serviced during certain operations. For example, if the HLDC is reading data off the disc, then the servicing of an interrupt may cause data to be lost. The following discussion will show that there are other satisfactory methods of initiating a file command, and consequently no interrupt mechanism is supplied.

(ii) Another method of communicating commands is to use the 'polling' method of communication which was used in first and second generation computer systems[26]. This approach, in a somewhat modified form, is again being performed by many computer systems. Examples include the Berkeley Computer Corporations timesharing system[27], and the Central Data Corporation (CDC) 6000 and CDC-7600 series computers[28]. There are basically two ways in which this could be done in the HLDC.

(a) The first method requires that a location (or locations) in main memory be set aside entirely for file command transfers. A file command could be requested by the Varian placing the address of the filename block in this location. (Chapter 2 discusses the filename block). The filename block would specify the file operation to be performed (amongst other things). The HLDC would access this memory location via DMA to see if a command transfer is requested, and would indicate that it had received the command by either setting this location to a particular value, or some other (fixed) location to a particular value. The HLDC would examine this location at regular intervals to see if a file command is requested. To allow efficient response time to file commands to be achieved, this location would have to be frequently examined. The cycle-stealing associated with the examination of this location would cause a small amount of system degradation.

(b) In the second method, the Varian initiates a file operation by executing a single-word output transfer instruction. The data transferred to the HLDC by this instruction would be the address of the filename block. The HLDC, on finding that an output transfer instruction had been executed, would gate the data associated with this instruction into the central processor array. The advantages of this method is that it will not degrade the main processor's performance, and it is easily implemented. Consequently, this is the method that is recommended.

File Command Completion

When a file command is complete, the HLDC must communicate the completion to the Varian. As a number of file operations may be active in the HLDC at any one time, the HLDC must indicate which file operation is complete. The methods of indicating the completion of a command can be performed in a similar way to the initiation of a command. For example, the HLDC could set one of the words in the filename block to a particular value. The disadvantages associated with this method is that the main processor would now have to inspect the appropriate word in all the filename blocks at various intervals.

The most efficient method is to issue an interrupt to the Varian. This is the method which would most commonly be used, as most real-time operating systems are interrupt driven. The question still arises as to how the Varian would know which file operation is complete. One method would be for the Varian to input the filename block address of the completed file operation from the HLDC. It would do this by means of a single-word input transfer I/O instruction. The Varian, using the filename block address, will be able to determine which file operation is complete.

The above discussion has shown various ways in which the communication between HLDC and Varian can occur.

3.11.2 Physical Structure of the HLDC/Varian Interface

There are two physical parts to the HLDC/Varian interface:

(i) the interface which resides at the Varian; and
(ii) the interface which resides at the HLDC.

The basic design goal of the HLDC/Varian interface has been to make the interface at the HLDC as machine independent as possible (i.e. as independent of the Varian as possible), whilst making the interface at the computer as simple as possible. More simply stated, the interface at the HLDC must accomplish as much as possible whilst remaining machine independent. This design goal will allow the HLDC to be easily interfaced to different computers.

The HLDC/Varian interface is also used for transferring microprogrammes from the Varian to micromemory. This facility is essential in the prototype machine to allow easy microprogramming. Appendix B discusses how the microprogram transfer between Varian and HLDC is performed.

3.12 Disc Drive Interface

The disc drive interfaced to the HLDC is the CalComp DC1 disc drive. The disc drive specifications, and the flow charts for the various disc drive operations are given in the reference[29].

The HLDC must be able to transmit and receive signals and data from the disc drive. Some examples of the signals received from the disc are: index and sector pulses, signals indicating that the drive is online, etc. Examples of signals that must be transmitted to disc from the HLDC are signals instructing the drive to perform a seek, to switch a read/write head on for writing, to select the appropriate read/write head, etc.

Reading data from disc presents a special problem. Disc speeds are a function of power supply voltages and frequencies. Since these voltages and frequencies are subject to variations, disc speeds are variable. Data coming off the disc cannot, therefore, be read by a fixed clock. One method which can be used is to write alternate data bits and clock bits to the disc drive. When data is read off the disc, the clock bits will be used to update a variable frequency oscillator, and subsequently allow the prediction of the next data bit. Problems of recording data on disc are discussed in the literature[30,31]. The variable frequency oscillator (VFO) circuitry can be called the read clock. A fixed frequency clock is also required for writing data to disc. This clock can be called the write clock.

Since the read clock and write clock circuitry, together with the circuitry necessary to supply and receive control signals from the disc drive, was available from a Telefile disc controller, this circuitry is used in the HLDC.

Another problem to be resolved in the HLDC/drive interface is how data transfers between the CPA and disc drive will occur. Data transfers between the disc drive and the HLDC occur in serial form, and data transfers between the HLDC and the host computer occur in parallel form. It is therefore necessary to provide a means of converting data from serial to parallel form, and vice versa. The method used is to provide a 16-bit shift register. During write operations, 16-bit data is loaded into the register, and is clocked out to disc using the write clock. During read operations, data is clocked into this register by the read clock, and is then gated into the central processor array.

The control of the above operations are performed by certain input-control and output-control signals. The control of the disc drive is discussed in terms of these input- and output-control signals in Appendix D.4.

3.12.1 Disc format

The necessity for initializing (formatting) the disc was previously mentioned in Section 2.7. All discs to be used by the HLDC must first be formatted by the HLDC. The HLDC's format routine would perform various functions :

(i) A possible format for each sector on disc and a discussion on this format in terms of the input- and output-control signals is given in Appendix D.4. The format routine would first set up each sector according to the format illustrated in Figure D.3.

(ii) The format routine will then do successive read and write operations on the data area of each sector using different bit patterns. This operation will isolate any sectors which are unusable.

(iii) The free space map (refer Chapter 2) will be constructed during the format routine. The bits in the free space map corresponding to the unusuable sectors found by (ii) above will be set. The free space map will then be written to an area of disc.

(iv) A sector on disc will be set up to be the first sector of the master file directory (MFD). It must be decided whether or not a number of tracks will be reserved for the file directory, and if so, where these tracks will reside on disc.

One problem which must be solved is how the HLDC will know where the first sector of the MFD is stored on disc when the HLDC is initially started up. A possible solution to this problem is to store the first sector of the MFD at a specified sector on disc. The address of this sector is stored in micromemory. If the specified sector is corrupted, the HLDC could then store the data at the first uncorrupted sector after the specified sector. At system startup, the HLDC would realize that the specified sector was corrupted, and would then inspect the following sectors until the MFD was found. A duplicate copy of the MFD will generally be kept. At system startup, the two copies of the MFD could be compared against each other to ensure that the MFD is uncorrupted.

3.12.1 Critical Timing

Data transfers to and from disc occur at a rate which is determined by the read and write data clocks. Consequently, it must be determined whether there will be any timing problems associated with transferring data to or from disc. The read and write operations present different problems, and they will be discussed separately.

Transfers from Disc to Main Memory (Reading)

There are basically two methods of transferring data from disc to main memory :

(i) Data can be transferred by first reading one, or a number of sectors from the disc drive to local memory, and then from local memory to main memory. There are, however, certain inherent disadvantages of transferring data using this method. Consider, for example, the typical data transfer from disc to main memory of a number of contiguous sectors of information. This transfer could be done as follows. The information could

be transferred from disc to local memory one sector at a time, and then this sector could be transferred immediately from local memory to main memory. The problem of this type of transfer is one of response time. That is, whilst the HLDC is transferring data from local memory to main memory, the next sector would have passed under the read/write heads, causing a disc revolution to be wasted. Another possible method of performing the transfer is to transfer all the data into local memory, and then from local memory to main memory. The problems here would be that local memory requirements would be excessive, and response time would not be maximized.

(ii) Data can be transferred directly from the disc drive to main memory without a detour to local memory. This method of data transfer has certain inherent advantages. Since data is transferred directly to main memory, no demands will be made on local memory space, and the data transfer time will be maximized. There are, however, certain timing problems which are associated with this type of transfer. When reading, the current word of data must be processed before the next word of data is received. The data-clock, for both read and write operations, will have a period of approximately 800 nanoseconds (refer Appendix D). Consequently, a word of data (16-bits) will be received by the central processor array from the disc drive every 16 x 0,8 = 12,8 microseconds. (Note that the HLDC, with a microinstruction cycle time of 280 nanoseconds, can execute 45 microinstructions in 12,8 microseconds). The HLDC will therefore have to read a word of data into the CPA, update the cyclic redundancy check (CRC) word, and output the data word to main memory via direct memory access in 12,8 microseconds.

The program sequence in Figure 3.6 illustrates how a sector of data would be transferred from the disc drive to main memory. This program illustrates that only seven microinstructions are required to perform the transfer of one word of data. Timing problems may occur, however, due to the time required to perform

```
#  DISC TO MAIN MEMORY TRANSFER.
#  Check if a word of data is ready to be input from the disc drive,
#  and set up the DMA output address in the VAR.DOUT latch.
AGAIN    *              INCT     EQ.16       VAR.DOUT
#  Input the disc data from the VAR.DIN latch to the accumulator.
        ACM(A)          *                    DISC.DIN
#  Perform the exclusive-or of the accumulator and register R1.
#  Set up the DMA.IN latch to initiate a DMA transfer to main memory.
        XNR(R1)         *                    DMA.IN
#  Decrement the T register, and clear the EQ.16 latch
        SDR(T)          *                    CLR.EQ16
#  If a word of data is ready to be transferred off disc before the DMA
#  transfer is complete, then the appropriate action must be taken.
MORE     *              JMPT     EQ.16       ACTION
#  Test if the T register is zero.  Check if the DMA
#  transfer is complete, if not complete, jump to location  MORE.
        TZR(T)          JMPF     DMA.FIN     MORE
#  R0 to MAR;  R0 + 1 to R0.  If the T register is non-zero,
#  perform the loop again.
        LMI(R0)+        JMPT     CO          AGAIN
```

NOTES: (i) The number of words to be transferred is initially set up in the T register;

(ii) the main memory address for the DMA transfer is set up in register R0.

(iii) It will be assumed that the check word will be a simple exclusive-or (XOR) of all the data words. The check word being stored in register R1.

Figure 3.6 Disc to Main Memory Transfer

a DMA data transfer. In the Varian computer, DMA transfers cannot occur after certain instructions[32], and the time required to perform the DMA transfer could therefore be in excess of the 12,8 microseconds. Consequently, if the HLDC transfers data directly from disc to main memory whilst the Varian is executing a random program sequence, timing problems may occur. The HLDC must therefore ensure that the word of data is processed within the specified time period. This can be done by alternatively checking the EQ.16 and DMA.FIN input-control signals as illustrated in Figure 3.6. If the EQ.16 signal becomes TRUE before the DMA.FIN signal, then this means that the next word of data is available for transfer before the transfer of the current word is complete. If this situation occurs the transfer of the sector may have to be aborted.

A possible method for overcoming this problem is as follows. The Varian can be forced into a special program loop when data is to be transferred. This loop could be entered into by an interrupt from the HLDC. The Varian would continuously loop, executing NOP instructions. When the data transfer is complete, the HLDC can either interrupt the Varian or set one of the Sense latches to allow the Varian to exit from this loop. The main disadvantage of this method is that the main processor will be idle whilst data transfers between the HLDC and Varian are in progress.

The above discussion has indicated the disadvantages associated with transferring data first to local memory and then to main memory, and the disadvantages associated with transferring data directly from disc to main memory. One possible method of transferring data from disc to main memory is to use a combination of the above two methods as follows. The HLDC, to perform a data transfer will not interrupt the host computer, but will transfer data from disc to main memory directly, until a situation occurs where a word of data cannot be transferred within 12,8 microseconds. If this situation occurs, the next word of data received from disc would then be written to local memory. From here on, the data transfer would occur by forming a first-in-first-out queue in local memory of data to

be transferred to main memory. Data from disc would be added onto one end of the queue, whilst data to main memory would be removed from the other end of the queue. A state of equilibrium would again be met when the queue becomes empty. The maximum queue size would possibly be restricted to one sector. This method, or a variation of this method, will help maximize the HLDCs performance.

Transfers from Main Memory to Disc (Writing)

If data is written directly from main memory to the disc drive, the timing associated with DMA transfers could cause timing problems to occur. If a DMA data transfer is too slow, the word of data which was to be written to disc will not be received in time to perform such writing. If this situation was to occur, the entire sector of data would have to be rewritten. There are certain methods which can be employed to overcome this problem. Possibly the best solution would be to first transfer an entire sector of data from main memory to local memory. This sector could then be written to disc whilst the next sector of data is being transferred from main memory to local memory via DMA.

3.13 The Console

A necessary development tool for both hardware and software in a prototype machine is a versatile console facility. A console is also useful in a production machine as a debugging aid for both hardware and software.

The conventional console design, with all its switches and displays, has lost favour in many modern computer designs, and the conventional console is being replaced by a virtual console[33]. In a virtual console system, all functions normally carried out by the conventional console, are carried out by a program in read only memory (ROM). If the virtual console design were to be used in the High Level Disc Controller, these functions would reside in microprogram memory, and would be accessed from the host computer via a teletype or a video display unit. This approach is very attractive for a production machine, as system cost would be greatly reduced. That is, the rows of switches and displays with their controlling logic is very expensive, and most of this expense can be eliminated by using the virtual

console design. The virtual console design is, however, unsuitable for a prototype system. The conventional console design is therefore used.

The HLDC's console can perform the following functions :

(i) Read from or write to any local memory or micromemory location.
(ii) Step through or run a microprogram from any micromemory address.
(iii) Reset the system.

References

1. In many computer systems, memory is divided up into modules (or boxes) to provide better transfer capability between main memory and all processors requiring access (Richard W. Watson, "Timesharing System Concepts", McGraw-Hill, Inc. (1970), pp. 79-82). In most small computer systems, however, there is only one access path (Watson, p. 82), and data is generally transferred between the backing store device and main memory by cycle stealing.
2. Intel Data Book, Intel Corporation, Santa Clara, California (1976), p. 750.
3. M.P.J. Stevens and A.M.G. Claessen, "A Universal High Speed Micro-programmable I/O-Controller", Microcomputer Architecture, Euromicro (1977), pp. 250-257.
4. V.M. Powers and J.H. Hernandez, "Microprogram Assemblers for Bit Slice Microprocessors", Computer, Vol. 11, No. 7 (July 1978), pp. 108-120.
5. P.W. Mallett and T.G. Lewis, "Considerations for Implementing a High Level Microprogramming Language Translation System", Computer, Vol. 8, No. 8 (August 1975), pp. 40-52.
6. M.G. Rodd, "Organization of Industrial Control Computers", unpublished Ph.D. dissertation, Dept. of Electrical Engineering, University of Cape Town (1976), appendix D, pp. 1-5.
7. W.T. Adams and S.M. Smith, "How Bit-Slice Families Compare: Part 1, Evaluating Processor Elements", Electronics, Vol. 51, No. 16 (August 3, 1978), pp. 91-98.
8. Andrew Colin, "Intel 3000 and AM 2900 Microprocessors - a comparison". Microprocessors, Vol. 1, No. 5 (June 1977), pp. 287-292.
9. 3001 Microprogram Control Unit, Intel Corporation, Santa Clara, California (1975), pp. 1-14.

10. Colin, pp. 287-292.
11. Colin, pp. 287-292.
12. M.C. Sole, "An Optimal Real-Time Language Processor", unpublished M.Sc. thesis, Dept. of Electrical Engineering, University of Cape Town (1978), pp. 75-77.
13. Rodd, appendix D, pp. 1-5.
14. W.T. Adams and S.M. Smith, "How Bit-Slice Families Compare : Part 2, Sizing up the Microcontrollers", Electronics, Vol. 51, No. 17 (August 17, 1978), pp. 96-102.
15. <u>M2900 Bipolar (TTL) Processor Family</u>, Motorola Semiconductor Products, Inc. (1976), pp. 18-26.
16. Intel Data Book, pp. 115-118.
17. Intel Data Book, pp. 46-49.
18. <u>Varian 620/L Computer Handbook</u>, Varian Data Machines, Irvine, California (1971), chapter 11, p. 3.
19. Varian, chapter 11, pp. 16-17.
20. Varian, chapter 11, p. 17.
21. Varian, chapter 8, p. 10.
22. Varian, chapter 11, p. 3.
23. Varian, chapter 11, p. 3.
24. Varian, chapter 11, pp. 22-28.
25. M.G. Rodd, "Is your Interrupt Really Necessary?", Dept. of Electrical Engineering Research Review, Vol. 2, No. 3 University of Cape Town (April 1978), pp. 70-72.
26. Watson, p. 120.
27. Watson, pp. 123-126.
28. Panel discussion on Input/Output Control Techniques, C.A.R. Hoare and R.H. Perrott (ed.), "Operating System Techniques", A.P.I.C. Studies in Data Processing No. 9, Academic Press (1972), pp. 218-223.
29. <u>CalComp Field Engineering Service Handbook</u>, Calcomp CD1 Disc Drive, California Computer Products, Inc., Anaheim, California (1971), p. 23.
30. William I. Girdner and Wallace H. Overton, "Reading and Writing on the Fast Disc", Hewlett-Packard Journal (1972), pp. 12-14.
31. D.J. Kalstrom, "Simple Encoding Schemes Double Capacity of a Flexible Disc", Computer Design (September 1976), pp. 98-102.
32. Varian, chapter 11, p. 25.
33. Sole, p. 70.

CHAPTER 4

MEASURING THE HIGH LEVEL DISC CONTROLLERS PERFORMANCE

It has been proposed that the replacement of the low-level disc controller by the HLDC in certain real-time process-control computer systems will be beneficial. This benefit, it has been claimed, is due to improved system reliability and system performance when using the HLDC - the HLDC still allowing the computer system to meet the changing demands of the real-time process-control environment. It is necessary to determine to what extent the HLDC has achieved these goals.

The evaluation of system reliability and performance presents different problems. In this chapter, measurements relating to system performance will be obtained, and the HLDC will be evaluated in Chapter 5.

There are various methods of evaluating system performance. These methods include simulation and mathematical modelling. The danger with using any of these techniques is that certain aspects of the system may be overlooked[1]. The approach adopted is to compare a computer system interfaced to the HLDC with a similar system interfaced to a low-level disc controller.

The process-control computer system under review will generally use a minicomputer, and the filing system functions will generally be performed by this computer - as opposed to a backing-store processor. Consequently, the following analysis will be based on a computer interfaced to a low-level disc controller - the computer performing the filing system functions; and a computer interfaced to the HLDC - the HLDC performing the filing system functions.

What is of real interest to the system designer is the improvement that will be achieved in the computer systems response to the environment it is controlling when the HLDC is used. This is directly related to the average amount of disc usage a process-control computer will require. Consequently the improvement in the systems performance from one system to another will be variable. A possible method of gauging the improvements

to be expected when the HLDC is used would be to develop a sophisticated filing system and implement it on the HLDC. The HLDC could then be interfaced to a number of real-time process-control computer systems. Measurements of system performance will then be made when the HLDC is used, and when the low-level disc controller is used. In the latter case, the host computer performs the filing system functions. The two filing systems - the host computers and HLDCs - would have to be comparable for a justifiable comparison between the two systems. Analysis of each computer system interfaced to the low-level controller and then the HLDC would be done over a period of time, and data would be collected. The data would then be used as a basis for evaluating the improvement in performance when the HLDC is used. This method of analysing the HLDCs performance would possibly be the best method of analysis, as it would show the improvement in the computer systems response when different demands are made on the backing store device. This method is not, however, feasible in the context of this dissertation.

The system designer will generally know the backing store usage, and the effect the backing store is having on the computer systems response to the environment. Consequently, by knowing the relative improvement in the backing stores performance when the HLDC is used, the improvement in the systems performance can be calculated. The evaluation of the HLDC's performance can therefore be made on this basis.

One possible method of evaluation is that particular functions performed by the HLDC be compared against the same functions performed by a conventional system. For example, the time taken to find a file name in a file directory using the HLDC could be compared against the corresponding time required in a system interfaced to a low-level disc controller. This type of analysis is not meaningful, as it will not show how different functions relate to each other. For example, if the computer responds too slowly to a low-level controller's request, a full disc revolution may be wasted.

Another possible method of evaluation is to first look at how a file operation or file operations (i.e. read, write, open, close, etc) will

be performed in the multiprogramming environment. These operations could be measured in the system interfaced to an HLDC, and in a system interfaced to a low-level controller. The problem here is choosing a file operation or group of file operations which can be considered as being typical. The type of operations which will generally be required will be dependant on the system. For example, certain systems may have a large turnover in files, and in this instance the file operations "create" and "delete" will be extensively used.

What is of real interest, however, is how the HLDC will respond to time-critical tasks. In the general process control environment, time-critical tasks associated with disc will generally require information to be read off disc into main memory. Examples include the transfer from disc into main memory of parts of the operating system necessary for the execution of a time-critical task; of data which is required by a time-critical task; or of parts of an applications program associated with a time-critical task. Appendix A.1 discusses in more detail the types of information which will generally be foreground (time-critical) information. A typical situation would be that a task is initiated, but the information necessary for the execution of this task is on disc. In this situation, the operating system will create a task to bring the required information off disc into main memory. Chapter 2 indicated the type of functions which would be associated with such a task. For example, this task would enter the global routines Open, Read and Close to respectively open, read and close the appropriate file. These routines, would, in turn, communicate with the appropriate device drivers to perform file directory searches, data transfers off disc, etc. To make the following discussion easier, two assumptions will be made.

(i) In Chapter 2 it was mentioned that a special file command could be used to open a file, read the entire file into main memory, and then close the file again. It will be assumed that this is the file operation which will be performed to bring the required information off disc.

(ii) Chapter 2 indicated that a number of different tasks will be associated with the performing of a file operation. These tasks include queue handling tasks, disc driver tasks,

etc. It will be assumed that all the functions necessary to perform the above file operation will be performed by one task only. The only effect this assumption will have on the processing time of the task will be the time associated with the scheduling of the various different tasks (i.e. the task-swop times).

Consequently there will only be one task which is of interest. This task will perform all the operations necessary to bring the desired data off disc. This task will be referred to as the experimental task.

Using this experimental task as the basis for analysing the system will have certain advantages. Firstly, this task will be closely related to a typical time-critical task in a multiprogramming environment, and secondly, the full range of file system functions will be performed when executing this task. Consequently, obtaining measurements about this task will allow a meaningful comparison between the HLDC and the conventional low-level disc controller. The experimental task will therefore be used as the basis for evaluating the HLDC. Two questions immediately arise :

(i) What measurements relating to this task will be required for the analysis?

(ii) How will the experimental task be implemented?

4.1 Measurement of the Experimental Task

Chapter 1 specified that two goals are desirable when accessing data on backing store.

(i) A minimum amount of useful processing time must be taken up by the backing store system; and

(ii) The response to a backing store request must be as fast as possible.

These two goals can be restated for the experimental task to indicate what measurements must be made. This is done by examining the execution of this

task in more detail.

As mentioned in Chapter 2, any task in the system can be thought of as having three different states[2] :

(i) ready to be executed;
(ii) blocked whilst waiting for I/O or some other task to finish; and
(iii) running (executing).

The experimental task will, at different times, be in each of the above three task states. Whenever this task initiates an I/O operation, it will block itself, and will remain blocked until the I/O operation is complete. The task will then be placed in the queue of ready tasks, and will be rescheduled according to a scheduling algorithm.

When the task is executing, it will remain executing until it is complete; until it initiates an I/O operation (it will then block itself); or until it is suspended (according to the scheduling policy).

The following times associated with the experimental task are of interest :

(i) The run time of the task.
(ii) The task-swop times associated with swopping the processor to the experimental task. When a task enters into run mode, a task swop is required.
(iii) The cycle stealing associated with the experimental task. Cycle stealing data transfers will generally occur when the task is blocked.
(iv) The task's blocked times. The task will become blocked when it initiates an I/O operation.

The task's execution time is the sum of the run, swop and cycle stealing times. The task's response time is the length of time that the task is 'alive'. This includes the run, swop and blocked times of the task. It is assumed that the cycle stealing associated with this task will occur whilst the task is blocked (i.e. whilst some other task is in the run

state). There are, however, a number of elements in the task's response time which will make the response time variable. Firstly, the task may be suspended through the initiation of a higher priority task, thus increasing the response time; secondly, cycle stealing may occur whilst the task is in the run mode - this will also increase response time; and thirdly, there will be a number of queues in the system - the queue length and average delay-time-in-queue will affect the response time. The above three factors are variable and difficult to gauge, and these factors will be ignored for the time being. The effect that these factors will have on the response time will be discussed in Section 4.7 below. What is of interest, however, is the 'fixed' part of the response time. These are the elements of the response time which are directly associated with the task. The response time will therefore be considered as being the sum of the task's run, swop and blocked times.

The above mentioned goals can now be restated as follows :

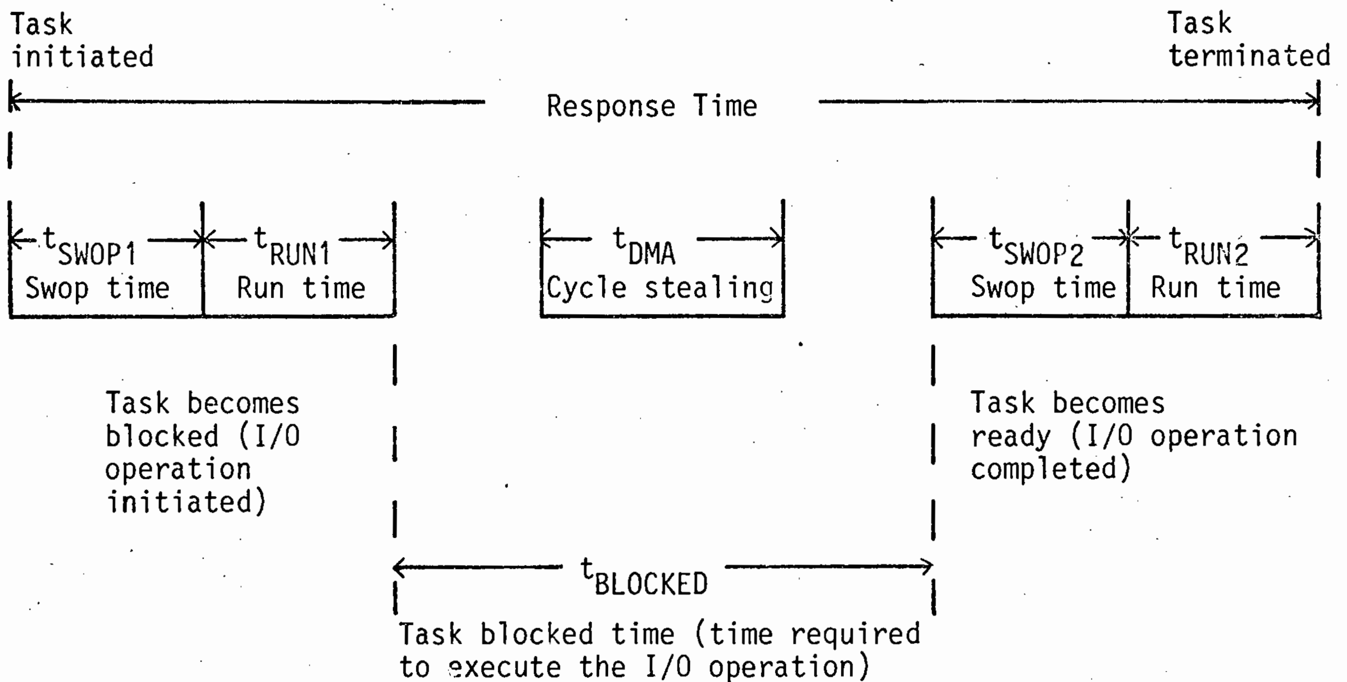
(i) The task's execution time must be minimized; and
(ii) the task's response time must minimized.

Figure 4.1 illustrates a task which is blocked only once during its life.

The task's blocked time will be the most significant factor in the task's response time because, in a moving head disc system, seeks and data transfers will be orders of magnitude greater than the processing associated with the task. It is desirable to isolate the device dependant characteristics in the analysis, as this will allow estimates to be made of the type of improvements in response time which could be expected by interfacing the HLDC to faster devices. This will also indicate the type of improvement in response time that can be expected with an improved microinstruction cycle time.

4.2 Implementation of the Experimental Task

Ideally, response times and task execution times would be measured by running the experimental task under the control of a real-time operating



t_{RUN} = the total run time of the task = $t_{RUN1} + t_{RUN2}$

t_{SWOP} = the total swop time of the task = $t_{SWOP1} + t_{SWOP2}$

$t_{BLOCKED}$ = the total blocked time of the task

Task execution time = $t_{RUN} + t_{SWOP} + t_{DMA}$

Task response time = $t_{RUN} + t_{SWOP} + t_{BLOCKED}$

Figure 4.1 The Life of a Task

system interfaced first to the HLDC, and then to a low-level controller. Configuring such an experimental situation would be difficult, requiring a large amount of programming. It will be shown below that meaningful measurements can be obtained by a much simpler method, and consequently the above method is not used.

A possible method for obtaining measurements on the experimental task is that an experimental environment be developed for this task. The experimental environment would be closely related to the environment in which a typical process-control computer's filing system runs. The experimental task would then be executed in this environment and measurements taken. The essential components of such an environment would be :

(i) A computer system.
(ii) A low-level disc controller.
(iii) The HLDC.
(iv) A file-system structure.
(v) Other necessary data structures such as the filename block.

The advantage of the above configuration (that is, an experimental task running in an experimental environment), is that measurements can be easily taken. Consequently this is the method of evaluation that will be used. The essential components of the experimental environment will first be chosen, and the processing to be performed by the experimental task will then be discussed. At each stage the experimental file situation will be compared with the real-world file situation. Where assumptions have been made, the possible effects of these assumptions on the measurements to be taken will be discussed.

4.2.1 System Components

As mentioned in Chapter 1, the process control environment under review will generally use a minicomputer to perform the process control functions. Consequently the computer used for the experimental environment can be a Varian 620 minicomputer.

Measurements of the task's performance can therefore be made in the following two systems :

(i) A Varian 620 minicomputer interfaced to the HLDC - the HLDC performing all the filing system functions.

(ii) A Varian 620 minicomputer interfaced to a low-level disc controller such as the Telefile disc controller[3]. In this configuration, the Varian will perform the filing system functions.

4.2.2 File-System Structure

The file-system structure to be accessed by the experimental task is based on the file-system structure discussed in Chapter 2. Any assumptions which are made about this structure will be mentioned. The structure is now discussed (refer Figure 4.2) :

(i) The index block method is used (refer Appendix A.1 for a discussion on the index block method). The first sector of the index block will contain the access control information and other relevant information about the file. This information will occupy the first twenty words of this sector.

(ii) The information to be transferred from disc to main memory is held in a file called EXP (for experimental). EXP is made up of four information sectors. This number has been arbitrarily chosen. The average number of sectors requiring transfer will be dependant on the process control environment. Since transfer times are device dependant, the transfer times will be isolated in the analysis. The four sectors and the index block will reside on contiguous locations in one cylinder. This is a reasonable assumption, as the filing system will generally produce this situation (refer Section 2.4.3).

(iii) The file directory is to have a two-level hierarchical structure. In this instance the pointer to EXP will be in a sub-directory called the Applications File Directory. The pointer to the Applications File Directory is in the master file directory.

(iv) The entire master file directory (MFD) will be held in fast memory. Since most computer systems have a small and frequently accessed MFD, the MFD is generally held in fast memory.

(v) The Application File Directory will have the following structure :

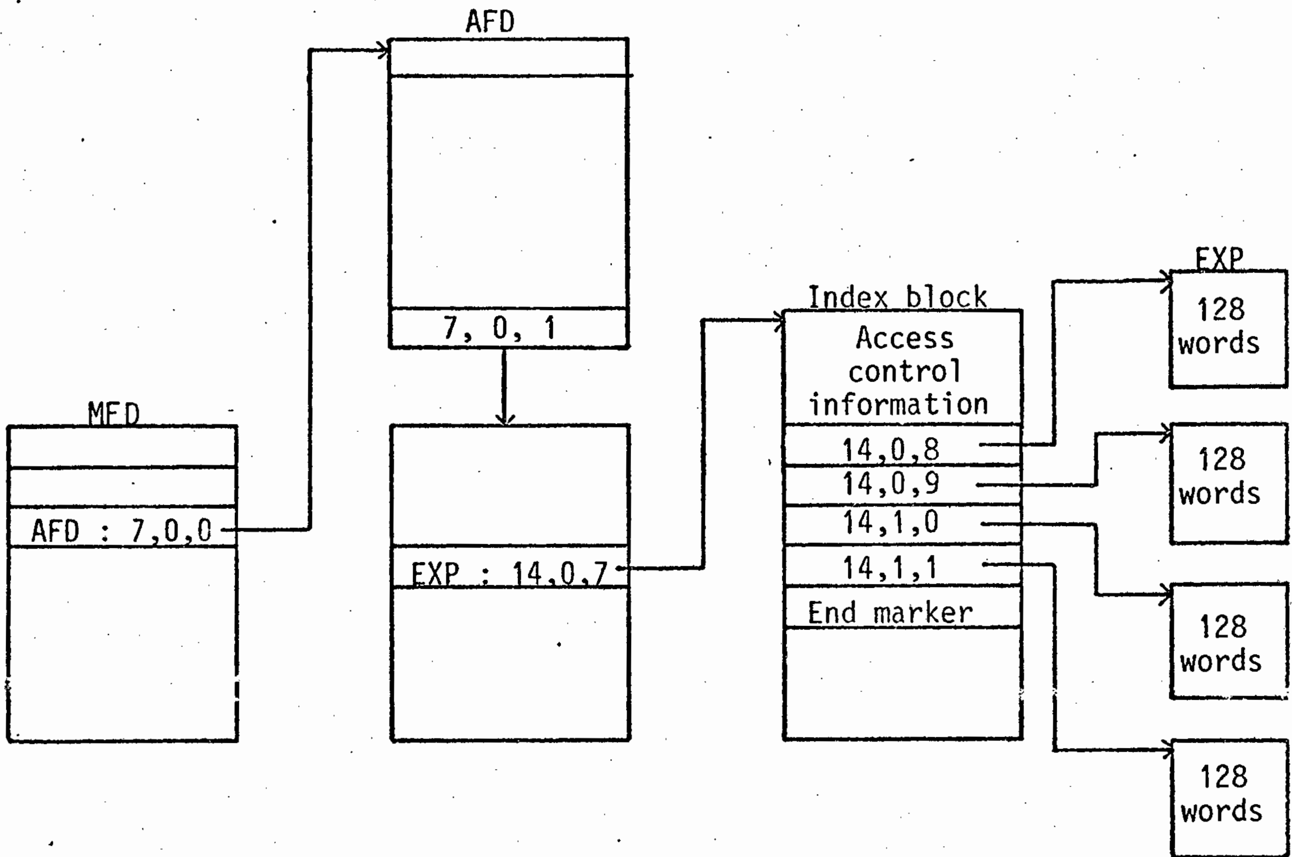
(a) The required sector of the Applications File Directory (AFD) will be on disc. The probability of the required sector of a sub-directory being on disc is dependant on many factors. These include the size of the file directory and the amount of fast memory space available to house the file directory. The general situation is that one or a number of disc accesses are required to obtain the desired sector of the directory.

(b) The AFD will be stored on contiguous locations on disc, and one sector of the file directory will be brought into fast memory at a time to be searched.This will be the general method of storing and accessing the file directory.

(c) Each file name can be a maximum of six characters. It will be assumed that the characters will not be packed, and therefore two characters will occupy one word. The file name will be stored in the file directory with a pointer to the appropriate index block. Consequently each entry in the file directory will be four words in length. The last entry in the file directory will be a pointer to the next sector of the file directory.

Since there are 128 words per sector, each sector will contain 31 file names plus the pointer to the next sector of the directory, if required. EXP will be the fiftieth file name in the AFD, and therefore EXP will be in the second sector of the AFD.

The above file structure is illustrated in Figure 4.2.



NOTES:

(i) The read/write heads are to be initially positioned at cylinder 0. Seek distances are unimportant, as seek times will be isolated in the analysis.

(ii) Address is given as: cylinder, track, sector.

(iii) The end marker is an all 1's bit pattern.

Figure 4.2 The Experimental File-System Structure

4.2.3 Other Data Structures

A filename block must initially be set up in main memory. In the case of the filing system functions being performed by the HLDC, the filename block will be transferred to local memory by means of direct-memory access (DMA). The following information will generally be stored in the filename block :

(i) The total number of words in the filename block. The number of words will be variable and will depend on the file operation to be performed.

(ii) An error word. This word is reserved for the filing system. If the transfer is successful, this word will be set to an all 1's bit pattern. If the transfer is unsuccessful, the word will be set to one.

(iii) The file command. In this instance a read command (RD) is required.

(iv) The file directory name. In this instance the file directory name is the Applications File Directory (AFD).

(v) The file name. In this instance the file name is EXP.

(vi) The access control information. Since a read operation is required, only the read key must be specified. The read key in this instance is READKY.

(vii) The main memory address that the data must be transferred to. This is the start address of the transfer. There may be a number of such addresses, each address having associated with it the number of words or sectors to be transferred, and the maximum number of words to be transferred.

(viii) The number of words or sectors to be transferred, or whether the entire file is to be transferred. The entire file is specified by an all 1's bit pattern. The transfer of a number of sectors is specified by the most significant bit (bit 15) being set.

(ix) The maximum number of words to be transferred. This is used as a protection facility to prevent the inadvertant overwriting of main memory.

Before the execution of the experimental task can begin, the above file-

system and data structures must be set up. That is, EXP's index block and EXP's data blocks must be written to the appropriate sectors on disc, and the filename block must be written into main memory (i.e. the Varian's memory). When the HLDC is used, the master file directory must be loaded into the HLDC's local memory. When the Varian is used, the master file directory must be loaded into main memory.

4.3 Execution of the Experimental Task

It must be ensured that the execution of the experimental task in the experimental environment will be similar to the execution in a typical multiprogramming environment. In the experimental environment, task swops will not occur, but after initiating an I/O operation, the program will loop waiting for an interrupt from either the HLDC or the low-level controller. In the multiprogramming environment, however, when a task initiates an I/O operation, the task will become blocked. When the task again enters run mode, a task swop is required. Rodd[4] shows that task swops can be lengthy. For example, in a system using a microprogrammed controller, task-swop times of 45 microseconds can be expected for tasks held in main memory[5], whilst in a more general system, task-swop times in the region of 200 microseconds can be expected for tasks residing in main memory[6]. It will be assumed that whenever the experimental task initiates an I/O operation, the task will become blocked, and when an interrupt is received indicating the completion of the I/O operation, a task swop of 200 microseconds will be required.

The experimental task will be activated from the run switch on the Varians console, and will be complete when the Varian enters halt mode. The flowchart associated with the file system routine which performs the required file operation is illustrated in Figure 4.3. The entry point of the flowchart will be different for the two configurations. The appropriate entry will be discussed in Section 4.5 for the HLDC, and Section 4.6 for the low-level controller.

4.4 The Varian Interfaced to the High Level Disc Controller

Section 3.11.1 discussed various ways of initiating file commands. It will

be assumed that file commands will be initiated by transferring the filename block address to the HLDC. This address is transferred by a single-word output transfer instruction. Initially the HLDC will be continuously cycling in an idle loop testing if an I/O command has been initiated by the Varian. This loop is illustrated in Figure 4.4. When a single-word output transfer command is received, the HLDC will transfer the filename block into local memory. In this instance, the file operation specified by the filename block will be to read data into main memory. When the data transfer is complete (or if an error has occurred), the HLDC will send an interrupt to the Varian indicating the completion of the file operation. The functions to be performed by the HLDC are illustrated in the flowchart in Figure 4.3.

The experimental task is initiated by the run switch on the Varian console, and the task will be complete when the Varian enters halt mode. The experimental task is illustrated in the flowchart in Figure 4.5 and the time required to perform the task is discussed in Appendix F.

As mentioned above, two measurements associated with the experimental task are required :

(i) The task's execution time. $t^H_{EXECUTION}$, the task's execution time when the HLDC is used, is found to be equal to 2085 microseconds (refer Figure F.1).

(ii) The task's response time. $t^H_{RESPONSE}$, the task's response time when the HLDC is used, is found to be equal to 73195 microseconds (refer Figure F.1). The main component of the response time is the task's blocked time. The blocked time in this instance is the time required by the HLDC to execute the file command. The components of the task's blocked time are illustrated in Figure F.2.

4.5 The Varian Interfaced to a Low-Level Disc Controller

With the Varian interfaced to a low-level controller, the experimental task must perform all the filing system functions. Consequently the flowchart illustrated in Figure 4.3 is the flowchart for the experimental

task. The experimental task is again initiated from the run switch on the Varian's console, and is terminated when the Varian enters halt mode. The time required to perform the experimental task is discussed in Appendix F.

Two measurements associated with the experimental task are required :

(i) The task's execution time. $t^{L}_{EXECUTION}$, the task's execution time when the low-level disc controller is used, is found to be equal to 7465 microseconds (refer Figure F.3).

(ii) The task's response time. $t^{L}_{RESPONSE}$, the task's response time when the low-level disc controller is used, is found to be equal to 79935 microseconds (refer Figure F.3).

4.6 Effects of the Assumptions Made

Certain assumptions have been made to produce the above experimental file situation. The effect the alteration of these assumptions will have on the execution and response times of the experimental task in the two systems will now be discussed.

(i) A far greater amount of processing will be associated with the execution of a file operation in a sophisticated filing system than has been the case with the experimental file situation. The effect of the extra processing on the two systems will be as follows :

(a) The task execution time when using the low-level controller will increase, whilst the task execution time when using the HLDC will remain the same (that is, when the HLDC is used, most of the extra processing will be performed by the HLDC).

(b) The relative improvement in task response of the HLDC over the low-level controller should increase. This is because the measurements obtained show that the HLDC performs processing much more efficiently than the host computer. Therefore, the problems of wasted disc revolutions (refer to Appendix F) will more than likely increase in the latter

system (i.e. the host computer interfaced to the low-level controller).

(ii) Assumptions have been made about the way the file directory and the information files will be stored, and on the number of sectors requiring transfer. If the master file directory was on disc and not in fast memory, or if more sectors of the sub-directory (Applications File Directory) had to be searched, this would improve the response time of the HLDC relative to the low-level disc controller (refer Appendix F). If, however, many more data sectors required transfer, this would reduce the relative improvement of the HLDC in both response time and task execution time. The improvement in response time would be reduced since the data transfer time would become the most significant component in the response time; and the improvement in execution time would be reduced as cycle stealing associated with the data transfers would become the most significant factor in task execution time - the data transfer time and cycle stealing time being the same for both systems.

The above discussion shows that there are many variables dictating the degree of improvement in performance to be expected when using the HLDC. The actual improvement to be expected is dependant on a particular process control environment.

4.7 Summary

This chapter has obtained measurements on task execution and task response times when the HLDC replaces a low-level disc controller. These measurements were obtained by creating an experimental file situation - the components of this file situation were chosen so that it would be closely related to a typical time-critical process-control file situation. The measurements obtained will be used as the basis for evaluating the HLDC's performance.

References

1. Richard W. Watson, "Timesharing System Design Concepts", McGraw-Hill, Inc. (1970), pp. 237-239.
2. *Ibid.*, pp. 137-143.

3. Operation and Maintenance Manual, DC-16 Disc Drive Controller, Telefile Computer Products, Irvine, California (September 1970).

4. M.G. Rodd, "Organization of Industrial Control Computers", unpublished Ph.D. dissertation, Dept. of Electrical Engineering, University of Cape Town (1976), appendix I, pp. 1-12.

5. Ibid., appendix I, pp. 7-9.

6. Rodd, appendix I, p. 9.

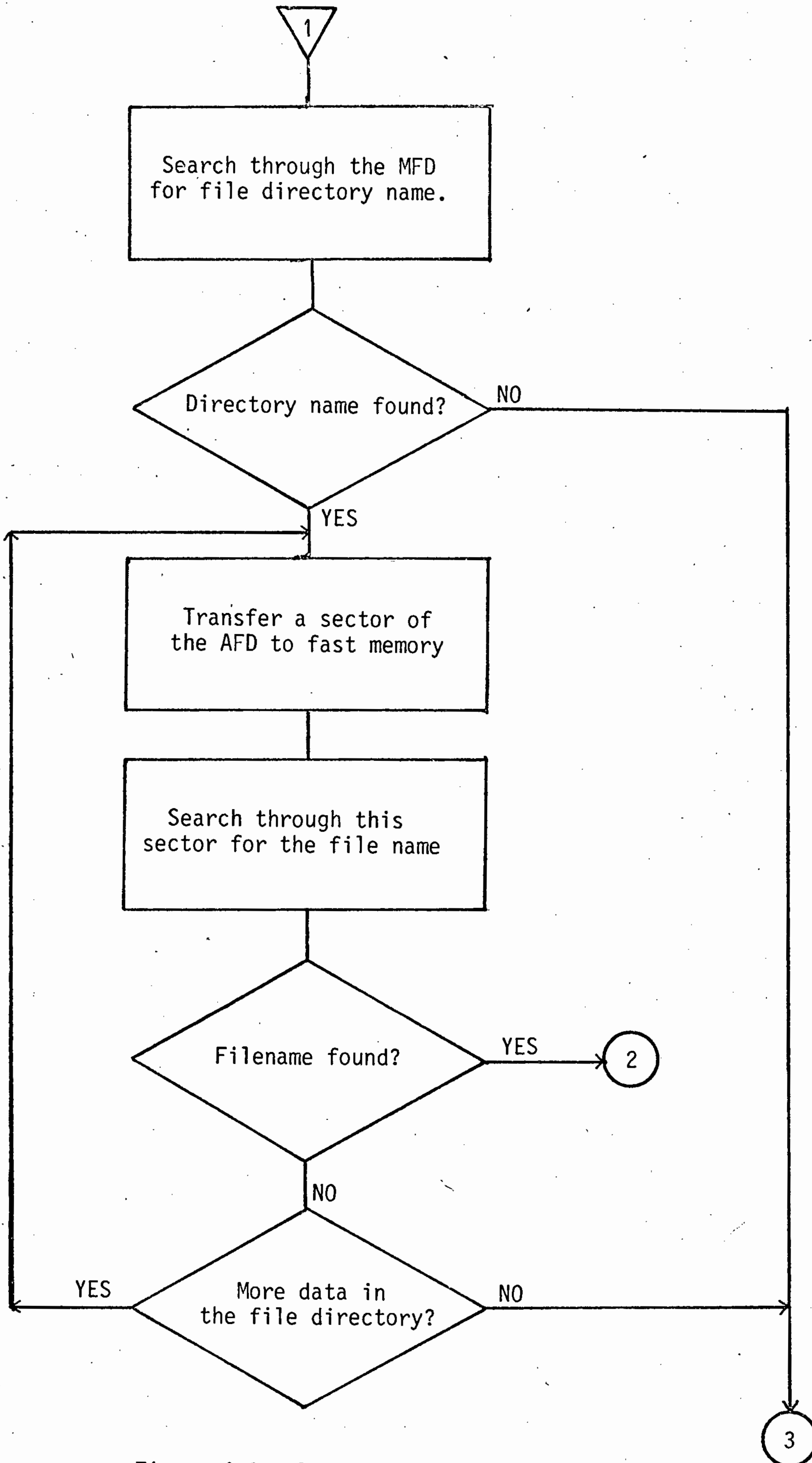


Figure 4.3 Flowchart of the File Operation

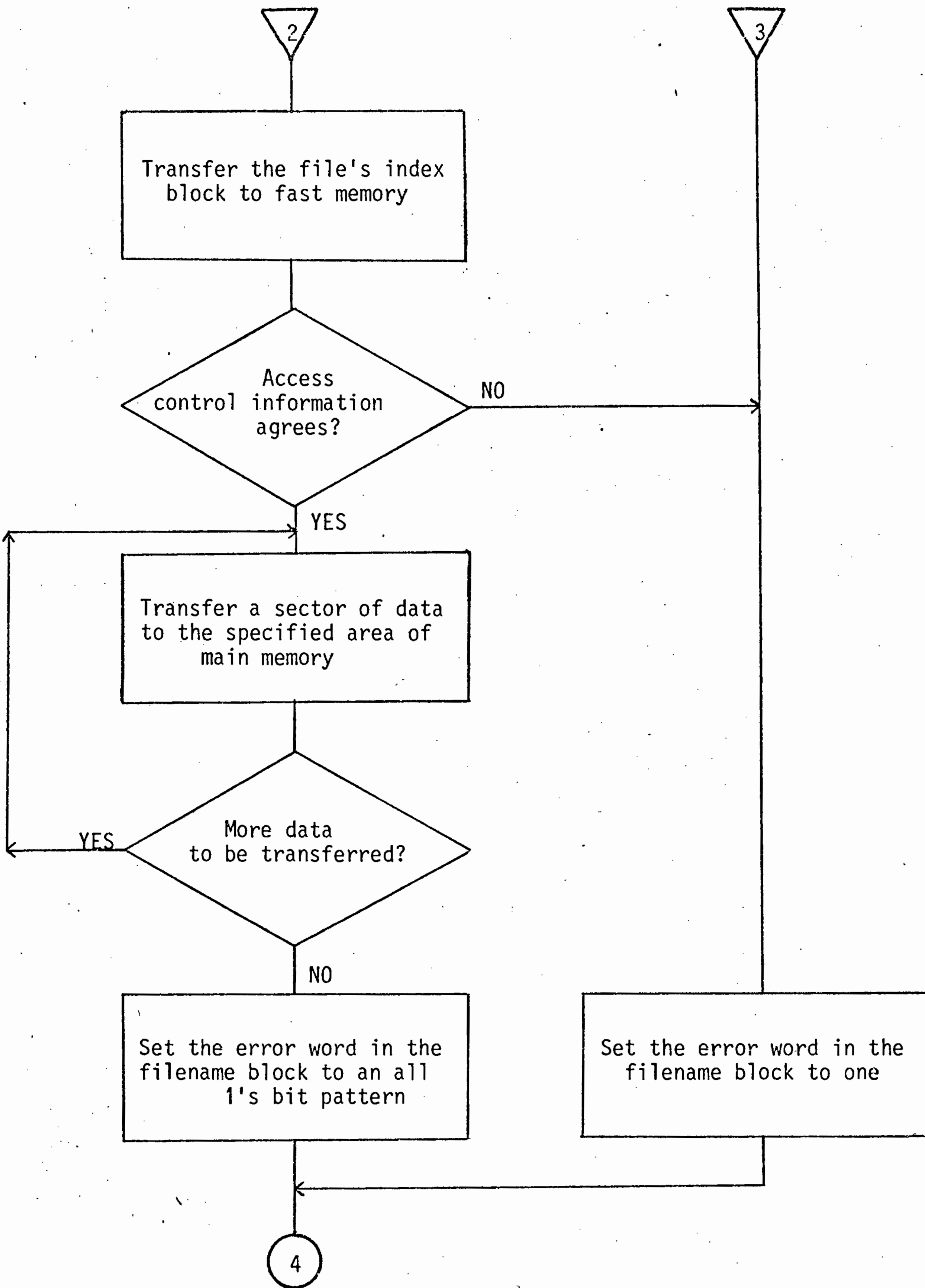


Figure 4.3 continued

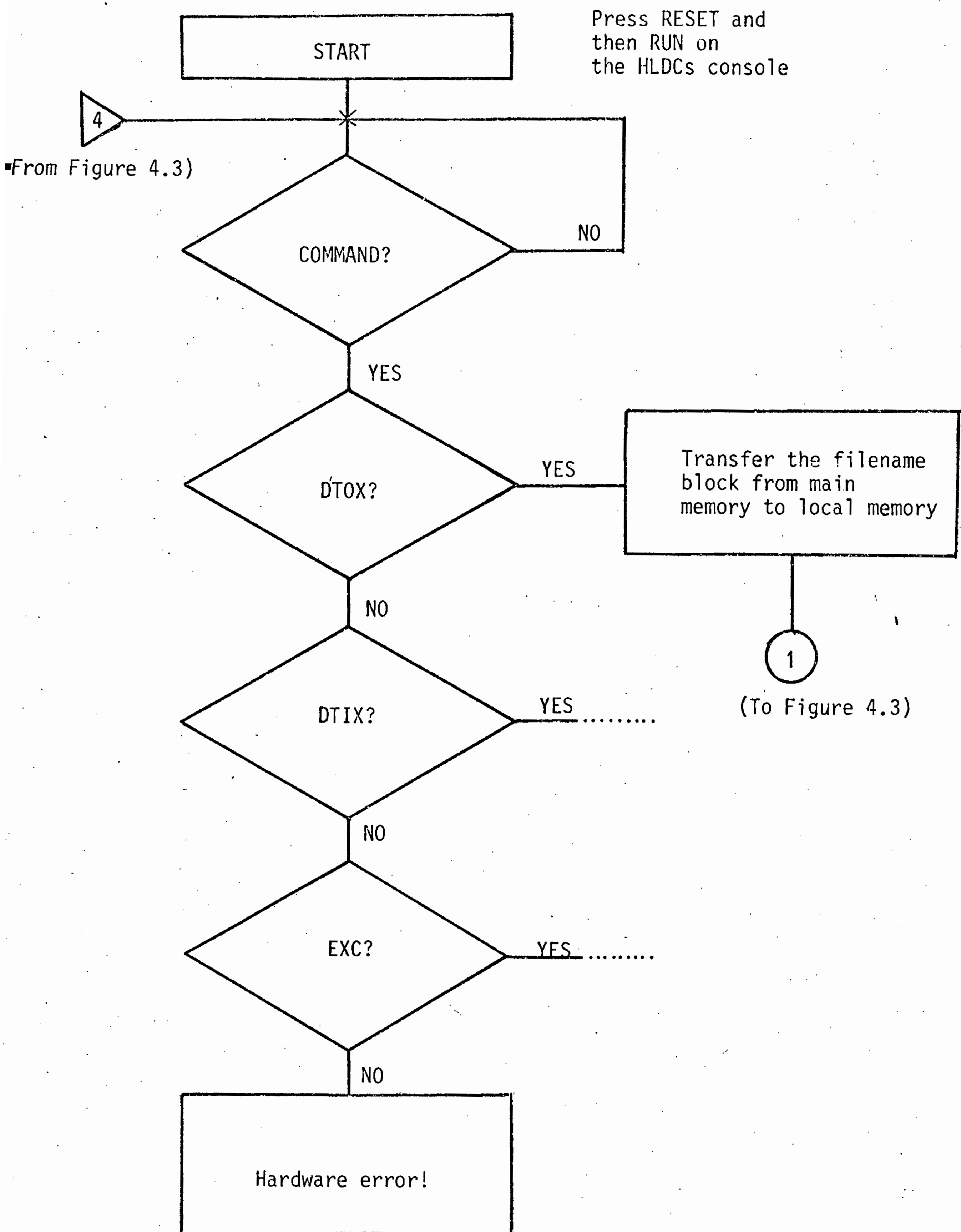


Figure 4.4 Flowchart of the HLDCs 'Wait' Loop

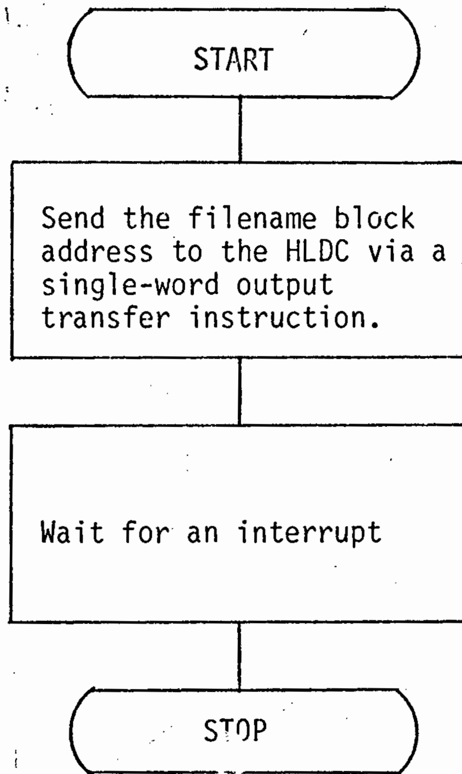


Figure 4.5 Flowchart for the Experimental Task when the HLDC is used.

CHAPTER 5

EVALUATION AND CONCLUSIONS

It must now be determined whether the HLDC will offer any improvements to the process-control computer system.

5.1 Evaluation of the High Level Disc Controller

System Efficiency

The analysis in Chapter 4 has provided the following measurements on the experimental task :

(i) The task execution time when the HLDC is used, $t^{H}_{EXECUTION}$, is 2085 microseconds.

(ii) The task response time when the HLDC is used, $t^{H}_{RESPONSE}$, is 73195 microseconds.

(iii) The task execution time when the low-level disc controller is used, $t^{L}_{EXECUTION}$, is 7465 microseconds.

(iv) The task response time when the low-level disc controller is used, $t^{L}_{RESPONSE}$, is 79935 microseconds.

The experimental situation discussed in Chapter 4 was developed on the basis of file operations performed in the general process-control computer system. Consequently, whilst the above measurements are based on a particular set of circumstances, it is expected that deductions made on the basis of these measurements will be valid for the general process-control computer system.

As mentioned in Chapter 1, the response to the environment by a computer system requiring backing store will be improved by :

(i) improving the response time to backing store requests; and

(ii) reducing the percentage of total processing time that the file system's operations will consume.

The response time of backing store is especially important when time-critical information is involved. It is often necessary to know the maximum possible response time to a request made on a particular backing store

device. This will indicate whether a faster backing store system is required or whether a particular task must be permanently resident in main memory. In terms of the experimental task, the following improvement in response time, calculated as a percentage, has been obtained when the low-level disc controller is replaced by the HLDC :

$$\frac{t^{L}_{RESPONSE} - t^{H}_{RESPONSE}}{t^{L}_{RESPONSE}} \times 100 = 8{,}4\%$$

A large portion of this improvement results from fact that the host computer will not respond to certain information coming off disc quickly enough, thus causing unwanted delays (refer Chapter 4 and Appendix F).

It is useful to estimate what improvement will be possible by interfacing the HLDC to a faster backing store device. Since the device dependant characteristics have been isolated in the analysis in Chapter 4, measurements relating to a faster backing store device can be easily calculated. The type of device that the HLDC could be interfaced to is a fixed head disc with double the data rate of the current disc drive. The measurements relating to such a device are given in brackets in Figures F.1, F.2 and F.3 in Appendix F. The resulting response times being :

$$t^{H}_{RESPONSE} = 34980 \text{ microseconds, and}$$
$$t^{L}_{RESPONSE} = 39685 \text{ microseconds.}$$

The percentage improvement when the HLDC replaces a low-level disc controller in such a system will therefore be :

$$\frac{39685 - 34980}{39685} \times 100 = 11{,}9\%$$

It would appear, therefore, that the improvement in response time will become increasingly more significant when a low-level controller is replaced by a "high-level" controller when a high-speed backing store device such as fast fixed-head disc or drum is used. This is to be expected, since when these faster devices are used, processing must be

even quicker to prevent delays through unwanted disc (drum) revolutions.

One of the advantages of the HLDC over a "software" backing-store processor is in system response. The backing-store processor, whilst providing the same reduction in task execution time as the HLDC, will only have a similar (or even slower) task response time to a backing store request as the process-control computer.

The other factor that is of interest, is the effect the reduction in the task's execution time will have on the computer's response to the environment. Before this can be determined, it is first necessary to calculate the percentage reduction in the experimental task's execution time when the low-level disc controller is replaced by the HLDC. That is,

$$\frac{t^{L}_{EXECUTION} - t^{H}_{EXECUTION}}{t^{L}_{EXECUTION}} \times 100 = 72\%$$

What is of real interest is not only the improvement in the average task execution time of filing system tasks, but of the average task execution time of all tasks in the system. This is a function of what percentage of the total processing time is used in performing filing system functions. If the filing system tasks represent 5% of the total processing time in a system having a low-level disc controller, then the percentage improvement when this controller is replaced by the HLDC is :

$$5 \times \frac{72}{100} = 3{,}6\%$$ ..... assuming a 72% improvement in the filing system's average task execution time.

That is, the HLDC will improve the system's response, this improvement becoming increasingly more significant as the disc usage increases.

Reliability

The measurement of system reliability is difficult if not impossible. One element in the system's reliability which can be calculated is that of component reliability[1]. The reliability figures of modern integrated circuit elements are high, and the probability of system failure through

component failure in a computer system is generally small. The measurement of software failure, however, involves the entire experience and history of the computing industry. One need only examine the increasing amount of cost going into software maintenance. Software maintenance costs currently amount to over 40% of the total software cost, with an expected cost of 60% by the mid 1980's[2]. (Software costs currently amount to nearly 70% of the total cost for a computer system[3]). It was mentioned in Chapter 1 that one of the major sources of software errors was the inadvertant overwriting of memory locations, and failure introduced into the software by modification. The placement of the filing system functions into firmware will prevent the problem of the program being overwritten. On the question of software modification, one could justifiably argue that firmware is in a sense itself software, and is therefore subject to program errors through program modification. A system such as the HLDC will undoubtably be modified to meet the requirements of different operating systems, and of the general changing process control environment. When modifications take place in a system such as the HLDC, the reliability can be improved by subjecting the systems to a large amount of automatic testing in the off-line situation.

It is claimed, on the basis of the above discussion, that the hardware implementation of the file system will be more reliable than an equivalent software implemented file system.

Main memory utilization

Chapter 2 mentioned that a significant amount of memory space is required to obtain good response from the backing store system. When the process-control computer performs the file handling functions, it will require main memory space to store parts of the file directory and free space map, and the index blocks of files currently open on the system. Added to this is the main memory space required to store part, or all, of the filing system routines. Contrasting to the main memory requirements when the process-control computer performs the filing system functions, the HLDC will only require a minimal amount of main memory space - this space will be used to house the programs necessary to initiate the

required file operations. The HLDC will, in effect, provide memory expansion by freeing that part of main memory required by the filing system. This point is important when estimating the cost of the HLDC.

Practical Value of the HLDC

In a real-world application, the microprograms would be permanently in read-only microprogram memory. If program modification is required, this could be easily accomplished[4]. If the microprogram memory has a suitable structure, new program sequences could be generated by the microprogram assembler running in the associated host computer.

Programming the HLDC is not an easy task. The microprogrammer must have a deep insight into the structure he is manipulating, and must fully understand the overall operation of the system. A programming aid, such as the microprogram assembler for the HLDC, does alleviate the programmer's task to some extent, as does good system documentation. Programming a system such as the HLDC will certainly be more difficult and more expensive than programming a conventional computer system.

A possible solution to this problem is that the manufacturer producing a system such as the HLDC produce software for this system. The manufacturer could produce a general filing system which would then be tailored to different users needs. The disadvantage here is that "general" software will generally not produce the highly efficient programs that are required[5]. The manufacturer could, therefore, produce a number of different hardware based filing systems which are tailored for various probable applications[6].

An important aspect in any system is that of cost. When considering a unit such as the HLDC, the user will weigh up the cost of the unit against the benefits that this unit will bring to the computer system. The difference between the hardware cost of the HLDC and the cost of a low-level controller such as Intel's micropramned "low-level" disc controller[7] will be in the extra memory the HLDC requires. As mentioned above, the HLDC will provide memory expansion by freeing part of main memory. This

factor will help neutralize the extra hardware cost of the HLDC. The major cost factor in the system will most probably result from developmental costs, which include software development costs. These costs become less significant in a product as the demand for that product increases. (That is, the development costs would then be recovered over a large quantity of items).

The discussion on the HLDC has centered on the process control environment. It is of interest to examine whether the HLDC could be of value in a more general computer environment. It was mentioned in Section 2.2.1 that in certain systems, the HLDC would only perform a small portion of the filing system functions. The HLDC would, however, still provide similar advantages to those mentioned above - that is, improved system response and reliability. One factor that could be improved by the HLDC is that of information security. Certain information stored on backing store may require a great degree of security against being either read or modified. Only a certain group of users would be allowed access to this information in a particular system. The system would identify these users by means of a series of pass-words. The problem is that the identification program could itself be modified if stored in software. By performing the identification program in hardware (possibly by the HLDC), security would be enhanced.

5.2 Summary

Chapter 1 shows that the computer system's response to the environment it is controlling is of major importance in a process-control computer system. In order to control an industrial process, many parameters must be monitored and controlled, usually under the supervision of a single computer. The computer system generally requires the use of one or more backing store devices. A backing store device often degrades the computer's response to the process being controlled.

The aim of this dissertation has been to produce a method which will help to improve the response times of those computer systems requiring backing store. On the basis of the hardware/software approach discussed in Chapter 1, an organisational strategy for improving the effectiveness of the backing store system was proposed.

It was claimed that by incorporating the file system functions and low-level controller functions together into one hardware unit, which is totally microprogrammed, certain system factors could be improved.

In order to demonstrate the validity of these claims, a fully operational system was designed and built. The hardware requirements of this system were based on a typical filing system discussed in Chapter 2. In Chapter 3, possible architectures for the system were analysed, and the design chosen was discussed in detail. The system was then built, and was interfaced to a moving-head disc drive - the resulting system being called a High Level Disc Controller. An assembler was written for the HLDC to allow software development to be easily accomplished. In this chapter and the previous one the HLDC was evaluated, and it has been shown that a system such as the HLDC will improve the effectiveness of certain industrial-control computer systems.

References

1. G.C. Hendrie and R.W. Sonnenfeldt, "Computer Reliability", ISA Journal, Vol. 10, No. 1 (January 1963), pp. 51 - 56.
2. T.G. Rauscher, "A Unified Approach to Minicomputer Software Development", Computer (June 1978), pp. 44-54.
3. Ibid.
4. L.H. Jones and R.E. Merwin, "Trends in Microprogramming : A Second Reading", IEEE Trans. Computers, Vol. 23, No. 8 (August 1974), pp. 754-758.
5. S. Ribeiro, "Talking to the Minicomputer", IEEE Trans. Industrial Elect. & Control Inst., Vol. 18, No. 2 (May 1971), pp. 67-72.
6. M.G. Rodd, "Organisation of Industrial Control Computers", unpublished Ph.D. dissertation, Dept. of Electrical Engineering, University of Cape Town (1976), pp. 78-79.
7. Intel Data Book, Intel Corporation, Santa Clara, California (1976), pp. 750-758.

APPENDIX A

A REVIEW OF FILE HANDLING TECHNIQUES

A.1 File Directories

The file directory is the data structure used to map symbolic file names to physical locations. The file directory is searched according to file name. Generally the file directory has a hierarchical structure. For example, if a user accesses the system by means of a user identification, then each file the user creates could be entered into the file directory as a combination of the users identification and the users file name. When searching for a particular file name, the file directory would first be searched according to users identification, and then according to the users file name. Organizing the file directory on a hierarchical basis will have certain advantages :

(i) A comprehensive file name will help prevent name clashes (i.e. a user wanting to use a file name already in use).
(ii) A hierarchical search will help in fast file accesses. This point will be expanded later.

A possible file directory structure is illustrated in Figure A.1. Circles represent sub-directories and squares represent information files. The file directory will be the set of all the sub-directories. (Information files are not part of the file directory, but pointers to these information files will be part of the file directory).

To access an information file, the symbolic file name will be used to provide the path down the 'tree' from the root directory (master directory) to the address of the information file.

Whilst the minimum information contained by the file directory about each information file is the symbolic file name plus its physical addresses, the file directory will usually contain administrative and access control information as well. The file directory will therefore contain the following types of information about each file, usually stored as a table of contiguous computer words[1] :

[1] Richard W. Watson, "Timesharing System Design Concepts", McGraw-Hill, Inc. (1970), pp. 202-203.

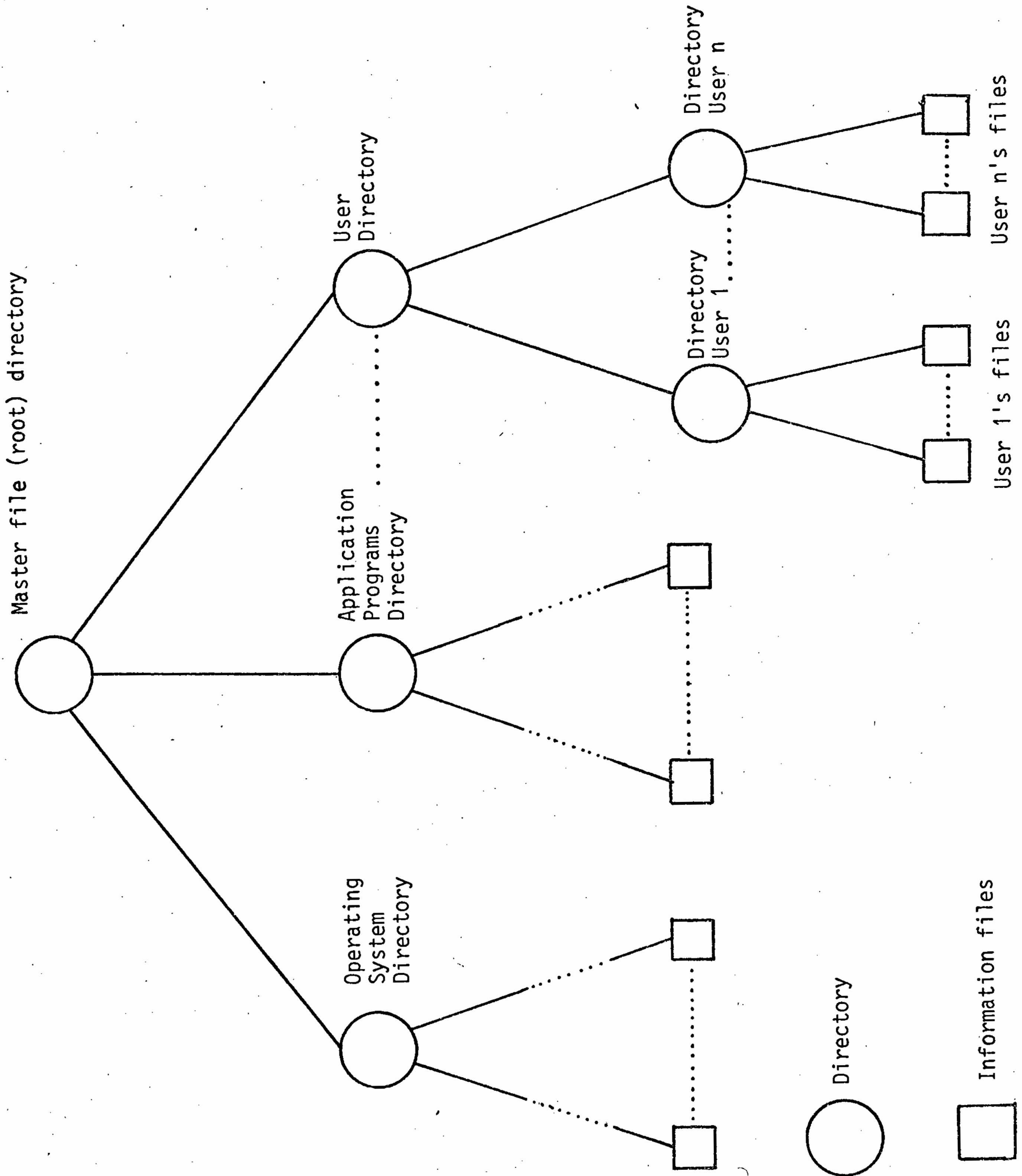


Figure A.1 File directory structure

(i) The file name as a character string or a pointer to such a character string.

(ii) Type of file, i.e. binary, character or other type as defined in a particular system.

(iii) Access protection (possibly a pointer to a list of other users who have access to the file and the kind of access which they are allowed such as read-only, write-only, read/write, or execute-only).

(iv) Information indicating directly where the file is stored or a pointer to further tables containing such information.

(v) Other information such as date of definition, frequency of use, or additional facts felt to be useful by an installation or system designer.

The file directory will be stored on disc. To aid searches into the file directory, part or all of the file directory is read into fast memory when the system is started up. If the entire file directory is in fast memory, searches through the file directory will be quick. The file directory will often be very large, however, and may be continuously expanding. Fast memory requirements in this situation will be extensive and unpredictable. The general method is to keep only a small part of the file directory in fast memory. When a directory search is required, the search will begin in fast memory. Parts of the file directory will then be brought off disc into this memory area until the search for the file address is complete.

The directory could be stored as a linear list. This structure may be adequate if the directory is to be small. If the directory is large, however, a large number of disc accesses would then generally be required before the file's address is obtained. A more general method of storing the file directory, to allow better access times, is to store it according to its tree structure. Figure A.2 illustrates how the file directory could be stored on disc. If, for example, user 1's file ABC is to be accessed, the following information must be received by the filing system :

(i) That a users file is required. This will direct the filing system to the user directory;

(ii) That the user is user 1. This will direct the filing system to user 1's directory; and

(iii) That the file name is ABC. The file directory will then search through the list of user 1's files for the file name ABC.

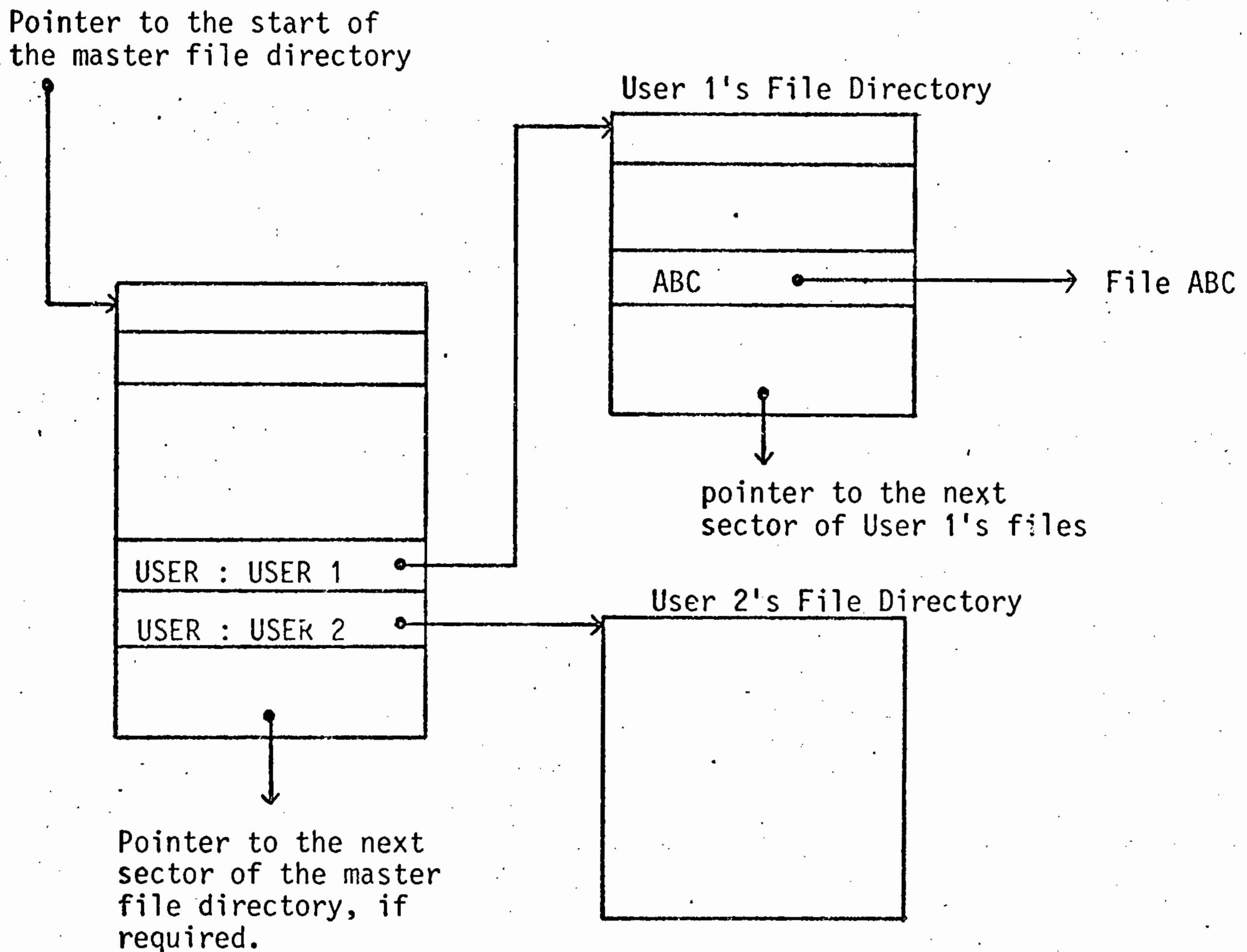


Figure A.2

Locating an information files address may still require a number of disc accesses. That is, although the master file directory will generally reside constantly in fast memory, the appropriate sub-directory required will usually be on disc, thus requiring a disc access.

One method of storing the file directory which could be used in the real-

time process control environment is discussed below.

A.1.1 A Method for Storing the File Directory in a Real-Time Process-Control Computer System

The process-control computer system will store the following types of information on disc :

(i) Operating system routines.
(ii) Compilers, interpreters, text editors, assemblers and debugging routines.
(iii) Application routines.
(iv) User files (both program and data files).
(v) General data files (e.g. information collected about the process being controlled).
(vi) Service routines.

Access to any of the above information will be through the file directory. The above six categories of information can be divided up into foreground and background information. Foreground information is that information associated with time-critical functions, and in the process control environment, foreground information will be the information associated with the control of the process. This information will include :

(i) Some of the application programs;
(ii) Those parts of the operating system and those data files associated with time-critical tasks; and
(iii) possibly language interpreters (e.g. BASIC).

Background information will include :

(i) Users files. Users files will generally be those programs in the developmental stage. When these programs are complete, they will usually be entered into the application programs library.
(ii) Compilers, text editors, debugging routines and assemblers. These processors will be used when developing user programs.
(iii) Certain applications programs. An example of a background application program would be a program used to prepare

reports on the systems operations.

(iv) Service routines. These routines will generally be run when the process-control computer is off-line.

Two sub-directory structures could thus be developed. One for foreground information, and the other for background information.

All file accesses would specify whether a file is foreground or background. The background sub-directory may be large, whilst the foreground directory may be much smaller. The foreground directory could thus be kept entirely in fast memory. The background sub-directory being multiplexed with a small area of fast memory. Having the entire foreground file directory in fast memory would greatly improve transfer speeds of time-critical information.

A.2 Storage of Data on Disc

Three possible methods of storing data on disc will now be discussed :

(i) In method one, sectors are chained to each other by pointers from one sector to the next. In such a system, if a user reserves a certain number of sectors, say 600, and only 50 of these sectors are currently required, the filing system will only allocate 50 sectors to the user. The remaining 550 sectors being allocated when required. Whilst the method utilizes disc storage space efficiently, data can only be accessed sequentially, and hence this method does not take advantage of the random access characteristics of the disc drive.

(ii) Method two uses the same principal as the above method, except that the data for a file must occupy continuous sectors. Random access would be achieved by the processor performing some simple arithmetic. The major problem with this method is file expansion. If the user reserves 600 sectors of which only 50 are used, the filing system has two possible choices. Firstly, it can reserve all 600 sectors of which 550 sectors will be unused; or secondly, it could reserve 50 sectors only, and if file expansion occurs, the data would have to be written to a contiguous area

of disc of at least 600 sectors in length. The first method is wasteful in storage space, and the second method will provide slow data access in a system having a large number of file modifications (i.e. data will be continuously shunted around the disc). A further difficulty occurs when a file is deleted, as large gaps will be scattered around the disc.

(iii) The method that is best applicable to the general filing system is to chain the physical blocks together by means of index blocks[2]. An index block is nothing more than a table of contiguous words, the i'th of which has the address of the i'th physical record in the file sequence. Several index blocks may be required, and they will be chained together by using one word of the index block to contain the address of the next index block in the sequence. The address of the first index block is maintained in the file directory. This concept is illustrated in Figure A.3.

The advantage of the above methods (i) and (ii) in organizing files is in the ease of implementation. The disadvantages being that both methods lack versatility, and do not allow efficient use of the disc. Method (iii), the index block method, whilst requiring more programming and more housekeeping, will help maximize storage efficiency and system response. A further advantage of the index block method is that administrative functions can be easily accomplished. These functions include, returning blocks no longer required to free space; locating blocks required for random access in a large file; adding blocks to or deleting blocks from the middle of a file; and checking the file for consistency after some malfunction.

[2] _Ibid._, pp. 192-195.

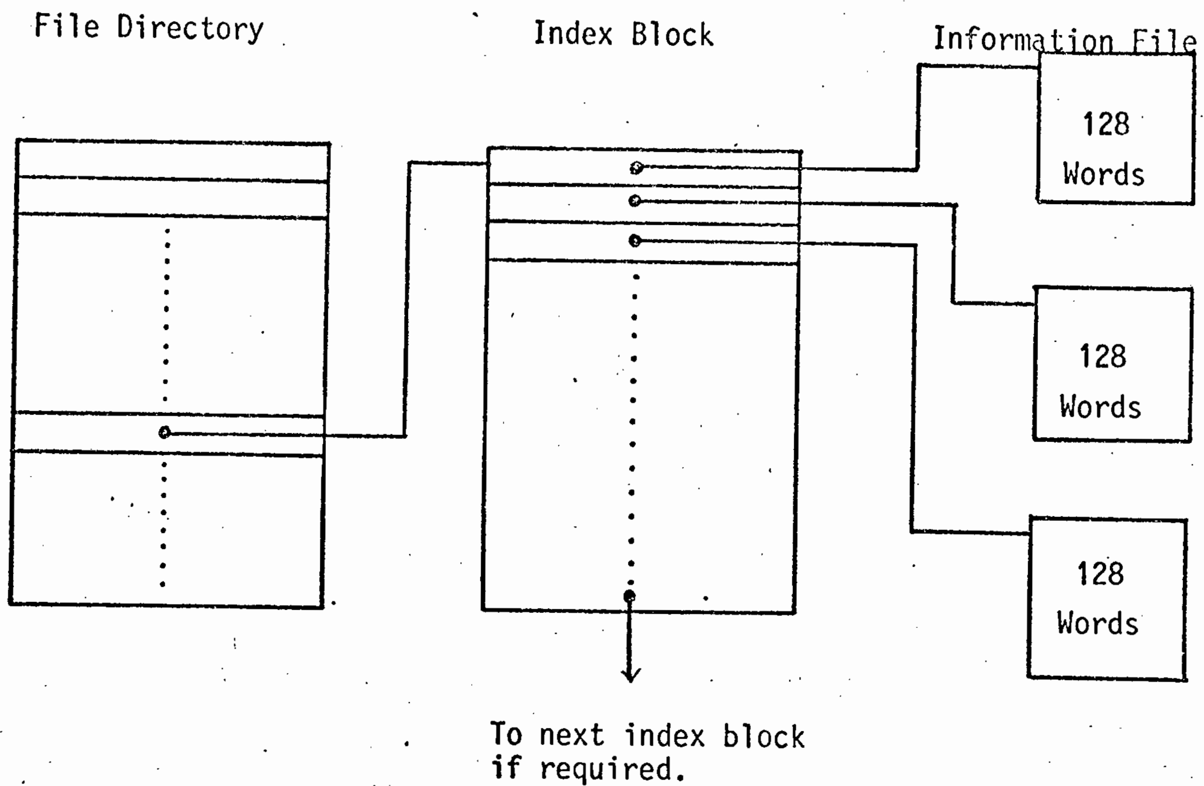


Figure A.3 Data Organisation for a File

A.2.1 Free Space Maps

The above discussion has shown how physical records can be chained together to form a file. It is necessary to have a method of keeping track of the unused physical records - i.e. free space. Various methods are possible. One method is to chain free sectors together. This method is simple to implement, and requires little storage space, but is slow when the access of more than one sector is required. Another method is to use a bit map (free space map)[3]. Each bit in the map represents one sector. A bit is set if a sector is currently being used and unset if the sector is free. The number of words required for the bit map depends on the number of sectors on the disc. For example, in the disc drive to be used, there are 200 cylinders with 10 sectors/track and 10 tracks/cylinder. Hence there are 20 000 sectors, and consequently 20 000 bits or 1250 sixteen bit words are necessary to house the entire bit map. The bit map would be stored on disc and ten sectors of 128 words/sector would be necessary to store the entire map.

[3]Watson, pp. 195-196.

A.3 Improving Data Reliability on Disc

The HLDC must be concerned :

(i) that the correct data is obtained for read/write operations;
(ii) that the correct data is written; and
(iii) that the correct data is again reread.

By sacrificing a certain amount of storage efficiency and/or system response, the HLDC can monitor these three functions to see if the disc drive has performed them correctly. To ascertain whether such a sacrifice is necessary, each of these three operations will be evaluated in terms of the following five points :

(i) the significance of the failure of the operation;
(ii) the reliability of the operation;
(iii) a method of improving the operations reliability;
(iv) the resulting reliability using this method; and
(v) the overheads in terms of response time and storage efficiency in using this method.

A.3.1 Sector Check

(i) If the incorrect sector is located for a write operation, valuable data may be lost. If the incorrect sector is located for a read operation, the incorrect program sequence may be performed with catastrophic results.
(ii) When locating a given sector, the drive will have to :

(a) perform a seek operation. This involves the physical movement of the read/write heads from one cylinder to another;
(b) perform a head select; and
(c) send sector pulses to the HLDC, so that the HLDC can perform a counting operation on these sector pulses to locate the desired sector.

The mechanical nature of the disc drive makes this operation highly unreliable, and a method of improving the

reliability is therefore desirable.

(iii) The conventional method of improving the reliability of locating a particular sector is as follows.

Before a disc can be used for normal read/write operations, it must first be initialized (formatted). In the formatting procedure, header information is written on every sector of the disc. The header information will contain the current cylinder, head and sector addresses. When a sector is located during a normal read/write operation, the header information (actual sector address) is compared against the expected sector address. If they do not correspond, the HLDC must take the appropriate action. To further improve reliability, the header may be followed by a cyclic redundancy check (CRC) word. When the HLDC reads the header, it will calculate the CRC for this header, and then compare it against the CRC stored with the header. Again, if they disagree, the appropriate action must be taken.

(iv) The above method will almost totally eliminate the problem of reading from or writing to the incorrect sector.

(v) The overheads of this method can only be calculated in terms of the format structure still to be developed. A possible format is illustrated in Figure D.3 in Appendix D. A header will require in the region of twenty extra words per sector. Twenty words per sector is about 10% of the disc space. If the space used to store the header was to be used for data, a 15,5% increase in data storage would be possible. A header is therefore expensive in terms of both storage efficiency and transfer rates. Since this method has the advantage of eliminating the incorrect sector being chosen, it is generally used.

A.3.2 Read Check

(i) If the incorrect task is read from disc, then the incorrect program sequence will be performed with possibly catastrophic results.

(ii) The reliability of the read operation involves more than the reliability of the read/write heads. Data may become corrupted on disc in a variety of different ways. For example, a head crash may occur, the disc pack may become damaged, etc.

(iii) The most commonly used method of checking whether data has been read correctly is to write a cyclic redundancy check (CRC) word (or words) with the data when the data is being written. When the data is read, a CRC is formulated from the read data, and compared against the written CRC. If these two CRC's compare, then it is assumed that the data is correct. If they do not compare, then the sector is reread. If the CRC's continuously disagree for a specified number of reads, it is assumed that the data on that sector is corrupted and hence irrecoverable. The sector must then be checked to see if it is permanently corrupted (i.e. by a series of successive read/write operations).

(iv) A cyclic redundancy check character will give the data read a great degree of reliability.

(v) The overheads in terms of response time and storage space are negligible, as only one extra word of data has to be written to or read from disc with each data transfer.

A.3.3 Write check

(i) If data is written to disc incorrectly, then that data will be lost. When reading the data again, the CRC's will not compare and the data loss will be discovered.

(ii) Data will generally be written incorrectly through some malfunction in the read/write heads.

(iii) To check if data has been written correctly, the data is first written and then reread. Each word of the read data is checked against the word of the written data (which will be stored in memory) to see if there are any discrepancies.

(iv) This method eliminates the problem of data being written incorrectly.

(v) Whilst this method does not require any extra storage space, it is very expensive in terms of response time. Every time data is written to disc, at least one full disc revolution is required before the data will have been checked.

The above method of improving data reliability will only be used in exceptional cases.

APPENDIX B

THE MICROPROGRAM ASSEMBLER

B.1 Introduction

Chapter 3 indicated the necessity for an assembler to be written for the High Level Disc Controller (HLDC). The assembler for the HLDC is written in FORTRAN IV, and is currently operational on the Varian V77 mini-computer at the Electrical Engineering Department at the University of Cape Town. Flow charts and program listings for the assembler are available from the author.

B.2 Microinstruction Format

Each microinstruction is capable of performing the following 3 functions :

(i) The control of the central processor array;
(ii) the selection of the next microinstruction address; and
(iii) the control of the external logic.

Each microinstruction is made up of seven different fields :

(i)	MIXED field	bits 0 to 11
(ii)	OPCODE field	bits 12 to 17
(iii)	MUX field	bits 18 to 23
(iv)	MCU field	bits 24 to 29
(v)	PORT-CONTROL field	bits 30,31
(vi)	K-CONTROL field	bits 32,33
(vii)	CI field	bit 34

These seven fields perform the above three functions as follows :

(i) The MIXED, OPCODE, K-CONTROL and CI fields together perform the control of the central processor array;
(ii) the MIXED, MUX and MCU fields together perform the selection of the next microinstruction address; and
(iii) the MIXED and PORT-CONTROL fields together perform the control of the external logic.

The MIXED field of the microinstruction is a multiplexed field, and can be used in any one of the three microinstruction functions, if required. The K-CONTROL field will determine if the MIXED field is required for the control of the central processor array; the MCU field will determine if the MIXED field is required for the selection of the next microinstruction address; and the PORT-CONTROL field will determine if the MIXED field is required to control the external logic.

There are many ways in which the HLDC's assembler can be described. Possibly the best, and quickest way for an experienced programmer to learn a computer language is to study a program written in that language, and to reference a number of notes which help describe the language. This is the method adopted. The microprogram assembler is given in its Backus-Naur Form (BNF) in Table B.1. A discussion on how a source program is developed using BNF is given in Table B.2. Examples of symbolic microprograms are given throughout the text. The remainder of this section should be studied together with these examples.

It is necessary that the format of the symbolic microinstruction be chosen to make the microprogrammers task as easy as possible. The format of the microinstruction chosen, given here in BNF, is as follows :

{<label>} {<processor> {<jump> {<mixed>}}} {'#' <comment>}

The symbolic microinstruction must, of course, completely specify all seven microinstruction fields. <processor> will determine the OPCODE, K-CONTROL and CI fields of the microinstruction. <jump> will determine the MCU and MUX fields of the microinstruction. <mixed> will determine the value to be placed in the MIXED field. One field must still be determined, however, and this is the PORT-CONTROL field. By having all the port mnemonics as reserved words (refer Table B.7), the port mnemonics specified by <mixed> will be used to determine the PORT-CONTROL field.

The remainder of this section will be in the form of notes, and will further describe the symbolic microinstruction.

B.2.1 <label>

Labels must start in column 1, and the first character of the label must be either an alphabetic character or a full-stop. Labels may be of any length, but only the first 12 characters are used. That is, labels must be unique in the first 12 characters. If a label is duplicated, then the duplicated label name is displayed on the system output device. This occurs during pass 1 of the assembly. Also, a label name must not be the same as any port name (Table B.7 lists all the port names).

Since no facility has been provided for linking external subroutines together, it has been found necessary to introduce the feature of internal and external labels. Any label beginning with a full-stop is an external label. The advantage of having the two types of labels is that :

(i) Internal labels are only unique within an external block. Consequently the same label names can be used in different external blocks. An external block consists of all the source statements between two external labels. The first source statement in the program being the beginning of the first external block, and the last source statement in the program being the end of the last external block.

(ii) An external label can be referenced from anywhere within the program.

An example of the use of internal and external labels is given below :

```
A1        *   JMPT   UNSAFE   A3
A2        *   JSR             .SUB1.
          *   HLT
A3        *   JMP             .ERROR0
.SUB1.    *   JMPT   UNSAFE   A3
A1        *   RTS
A3        *   JMP             .ERROR1
.ERROR0   *   HLT
.ERROR1   *   HLT
```

Note that internal label names can be repeated in different external blocks. External label names may not be repeated.

B.2.2 <processor>

(i) The default <processor> field is a NOP. If the character * is used in the <processor> field, a NOP will be assumed.

(ii) The default <register> is the accumulator (A). If no <register> field is present, the accumulator will be assumed.

(iii) If the character + appears, then the least significant carry-in bit of the central processor array is set.

(iv) The K-CONTROL field of the microinstruction is fully specified by <opcode>.

(v) Tables B.3 and B.4 list the opcode mnemonics and their corresponding functions.

B.2.3 <jump>

(i) Table B.5 summarizes the different conditional and unconditional jumps, and Table B.6 summarizes the different input-control signals.

(ii) The default jump is INC, and INC is assumed if this field is blank, or if the character * is used in this field.

B.2.4 <mixed>

(i) The port (output-control) mnemonics are summarized in Table B.7, and Appendix D discusses the output-control signals together with the input-control signals.

(ii) There are certain constants which are often used when disc accesses are required. Mnemonics for these constants are supplied by the assembler, and are summarized in Table C.8. These mnemonics, like the port mnemonics, are reserved words.

(iii) Examples of valid <port> fields :

(a) DRIVE1, DRIVE2, DRIVE4

(b) BUS.EN, TAG.EN

(iv) Example of an invalid <port>field :

DRIVE1, BUS.EN .... DRIVE1 and BUS.EN are of different port types (refer Table B.7)

(v) <digit> must be less than 2^{12} = 4096. If a minus is placed in front of an octal or decimal number, then the 2's compliment of that number is formed.

B.3 Operation

The microprogram assembler for the High Level Disc Controller runs on a Varian V77 Minicomputer, under the control of the Vortex 1 operating system. There is a copy of this assembler at the University of Cape Town. The name given to the assembler is MICRAS (Microprogram Assembler). An understanding of the Vortex 1 operating system is desirable for the following discussion.

I/O Devices and Logical Units

The Vortex 1 operating system allows the use of symbolic names for both I/O devices and for logical units. The logical units used by MICRAS are the PI, BO, LO, SO and SI, and the user must assign these logical units to the appropriate I/O devices before the execution of MICRAS begins. The logical units are used as follows :

PI (processor input) : The source program is input from the PI.
BO (binary output) : The binary output of the assembler.
LO (listing output) : All program listings are directed to the LO.
SI (System input) : All operator commands are made from the SI.
SO (System output) : All program commands are received on the SO.
I/O device assignments to the above logical units:
PI - will usually be the disc partitions XA or XB, or magnetic tape MO or M1.
BO - BO will always be papertape PT00.
LO - will usually be the line printer LP00, but may be SO.
SO and SI - will always be the backround terminal TY20.

B.3.1 Using the Assembler

When the assembler is loaded, it will initially respond with :

1, 2, E, O, P

on the SO. The user responds by typing in one of these characters.

1 means pass 1
2 means pass 2
E means End
O means Output Options
P means Filename

Source Program

A source program is a set of source statements which ends with an END card, where :

(i) the 1st character of END begins in column 1.
(ii) There is no other data on the END card.

A source program is defined in its Backus-Naur Form in Table B.1.

B.3.1.1 Pass 1 (1)

When pass 1 is requested, the source program is entered from the PI. For example, if the PI is set to XA, and if the filename is set to FORMAT (refer P option below), then the source program input will be the file FORMAT on the disc partition XA. Pass 1 builds up the symbol table, and all duplicate labels are listed on the SO during pass 1. When pass 1 is completed (i.e. when an END has been received on the PI), the program will again respond with:

1, 2, E, O, P

B.3.1.2 Pass 2 (2)

As in pass 1, the source program is entered from the PI for pass 2. Pass 2 does the syntax checking of each source statement and obtains the appropriate microcode. Outputs are made to the LO and/or BO as requested by the output options.

At the end of pass 2, the number of errors found in pass 2 will be listed on both the LO and SO. The assembler will then respond with:

1, 2, E, O, P

It is useful having pass 1 and pass 2 separated so that pass 2 can be executed as many times as desired without pass 1 being re-executed. For example, the user may first wish to check if his program has any errors in it. He will therefore execute pass 2 using the error output option. If no errors are found, he may then execute pass 2 with the listing option and/or the binary listing option.

B.3.1.3 END (E)

When the user wishes to exit from the processor, an E must be typed.

B.3.1.4 Listing Options (O)

The assembler supplies four different types of output. These are :

(i) Symbolic listing (L) on the LO;
(ii) Binary output (B) on the BO;
(iii) Error listing (E) on the LO; and
(iv) Symbol table listing (S) on the LO.

All these outputs occur during pass 2 of the assembly, and hence the listing options required must be set up before pass 2. The listing outputs are set up by typing an O on the SI in response to a :

1, 2, E, O, P

The assembler will then respond with :

*, B, D, E, L, N, S

where
B (binary listing) - Sets B. All other options remain unchanged.
D (default listing) - Sets L and resets B, E and S.
E (error listing) - Sets E and resets L. B and S unchanged.
L (long listing) - Sets L and resets E. B and S unchanged.
N (no listing) - resets B, E, L and S.
S (symbol table listing) - Sets S. All other options unchanged.

When one of the above characters is typed in on an SI, the assembler again responds with

*, B, D, E, L, N, S

This enables the user to set up all the listing options he requires. To get out of this mode, the user types a * on the SI.

B.3.1.5 <u>File Name (P)</u>

When a P is typed in response to

1, 2, E, O, P

the processor will respond with

FILE NAME =

The appropriate filename is then entered on the SI. The default filename is DATA.

B.3.2 <u>Output Formats</u>

As mentioned above, the assembler supplies four different types of output :

(i) <u>Symbolic Listing</u>

A symbolic listing is output to the LO as follows :

Record number (decimal)	Symbolic Record	Current location counter value (octal)	Object code if any (octal)

If an error is found in the symbolic code, then the error number (refer Table B.9) is listed <u>above</u> the erroneous record.

(ii) <u>Binary Listing</u>

If a binary listing is specified, then data is output in unformatted form to the BO, which will be papertape (PT00). The papertape serves as the input to the microprogram loader residing in the Varian (refer Section B.3.5).

(iii) Error Listing

If an error listing is specified, only the erroneous records in the program are output. The output has the following form :

Record number (decimal)	Symbolic Record	Error number (refer Table B.9)

(iv) Symbol Table Listing

If a symbol table listing is requested, it will occur after the symbolic listing or error listing on the LO device. The symbol table listing is a listing of all the labels used in the source program together with their corresponding octal location values. Labels are listed in the order they are received.

B.3.3 Error Handling

If an error is found in a source code statement, object code is still produced for that statement. The object code being the default microinstruction :

```
NOP(A)    INC
```

A source program containing errors can therefore be assembled and then loaded into the HLDC. The erroneous locations could then be changed manually via the console.

B.3.4 A Typical Terminal Session

The following is an example of a typical terminal session with the assembler. A user wishes to assemble a source program held in a file called FORMAT on the disc partition XA. A symbolic listing and a symbol table listing are required on the line printer, and a binary listing is required on papertape. The user inputs all his commands on the SI, and receives all outputs on the SO. The underlined commands are those commands typed by the user.

```
/ASSIGN PI = XA, BO = PT00, LO = LP00
/PLOAD, MICRAS
1, 2, E, 0, P
P
FILENAME =
FORMAT
1, 2, E, 0, P
0
*, B, D, E, L, N, S
S
*, B, D, E, L, N, S
B
*, B, D, E, L, N, S
*
1, 2, E, 0, P
1
1, 2, E, 0, P
2
0 ERRORS
1, 2, E, 0, P
E
MICRAS STOP.
```

B.3.5 Loading the Microprogram into Micromemory

The binary output of MICRAS can now be loaded into the HLDC's micromemory from the Varian 620 minicomputer by using the load program MICLD (Microprogram loader). The procedure for loading the data into micromemory is as follows :

(i) Switch the HLDC's MM.EN switch ON.
(ii) Ensure that the HLDC's RUN switch is OFF.
(iii) The start address of MICLD is 0100, hence the program counter (P-register) on the Varian must be set to 0100.
(iv) First press SYSTEM RESET and then RUN on the Varian.
(v) When the program has been read into micromemory, switch the MM.EN switch OFF.

The microprogram is now loaded into the HLDC's micromemory. To run the microprogram :

(i) Press SYSTEM RESET on the Varian's or the HLDC's console.
(ii) Switch the HLDC's RUN switch ON.

<source program> ::= {<record> <carriage return>}$_0^\infty$ 'END' <carriage return>

<record> ::= {<label>} {<statement>} {'#' <comment>}

<label> ::= <alphabetic> <character>$_0^\infty$ | '.' <character>$_0^\infty$

<alphabetic> ::= 'A'|'B'|'C'| ..... 'Y'|'Z'

<character> ::= any printing character

<statement> ::= <processor> {<jump> {<mixed>}}

<processor> ::= <opcode> {'(' <register> ')'} {'+'}|'*'

<opcode> ::= one of the central processor array mnemonics specified in Table B.3 and Table B.4

<register> ::= 'R0'|'R1'|..... 'R9'|'A'|'T'

<jump> ::= <unconditional>|<conditional> <multiplexer>|'*'

<unconditional> ::= one of the unconditional jumps specified in Table B.5

<conditional> ::= one of the conditional jumps specified in Table B.5

<multiplexer> ::= one of the multiplexer signals (input-control signals) specified in Table B.6

<mixed> ::= <porta> {','<porta>}$_0^\infty$|
<portb> {','<portb>}$_0^\infty$|
<portc> {','<portc>}$_0^\infty$|

<porta> ::= one of the port signals (output-control signals) of port type A specified in Table B.7

<portb> ::= one of the port signals of port type B specified in Table B.7

<portc> ::= one of the port signals of port type C specified in Table B.7

<digit> ::= {'-'} '0' <octal>$_1^4$|{'-'} <decimal>$_1^4$
..... digit $\leq 2^{12}-1 = 4095$

<octal> ::= '0'|'1'| ..... |'7'

<decimal> ::= '0'|'1'| ..... |'9'

<comment> ::= <character>$_0^\infty$

<carriage return> ::= carriage return on the input device

Table B.1 The HLDC's Microprogram Assembler in Backus-Naur Form

The following description of Backus-Naur Form is based on discussion by Maurer[1] and by Intel Inc.[2] The syntax for a language is its structure, or the form and order in which its elements may appear. The BNF syntax description notation is essentially a set of rules, each of which defines how a particular syntactic entity is to be constructed. There are two different classes of syntactic entities in BNF. One class is referred to as terminal symbols, and the other as non-terminal symbols. Terminal symbols are from symbols which may appear in a program written in the language that the BNF describes.

Terminal symbols appear between inverted commas (refer Table B.1). Non-terminal symbols and symbols which may not appear in a program, and are used in the BNF to build more complex structures in an understandable manner. Non-terminal symbols are bracketed by < >.

Notes:

(i) { } means optional.

(ii) | is used to separate alternatives.

(iii) For ::= the words "is defined as" should be substituted.
For example,
<letter> ::= 'A'|'B'|'C'|'D'
would be read as
"the non-terminal symbol <letter> may be written as any of the terminal symbols : A or B or C or D"

(iv) To specify that a particular sequence may be repeated a certain number of times, the notation $< >_i^j$ is used where i and j are integers with $i \leq j$
For example
<comment> ::= <character>$_0^\infty$
<character> ::= any printing character
means that <comment> can be replaced by any number of printing characters, including no characters.

Table B.2 Backus-Naur Form Description Notation

[1] W.D. Maurer, "An Introduction to BNF", BYTE, Vol. 4, No. 1 (Jan 1979), pp. 116-125.

[2] Intel Series 3000 Microprogramming Manual, Intel Corporation, Santa Clara, California (1976), appendix A, pp. 1-4.

Similarly, the sequence

<digit> ::= <number>$_1^2$

<number> ::= '4'|'5'|'6'

then <digit> can be replaced by one of the following :

4, 5, 6, 44, 45, 46, 54, 55, 56, 64, 65 or 66.

Table B.2 Backus-Naur Form Description Notation (continued)

ROUP	R-GROUP	MICRO-FUNCTION		MNEMONIC
0	I	$R_n + (AC \wedge K) + CI \to R_n, AC$		K01
	II	$M + (AC \wedge K) + CI \to AT$		K02
	III	$AT_L \wedge (\overline{I_L \wedge K_L}) \to RO$ $[AT_L \wedge (I_L \wedge K_L)] \vee [AT_H \vee (I_H \wedge K_H)] \to AT_L$	$LI \vee [(I_H \wedge K_H) \wedge AT_H] \to AT_H$	K03
1	I	$K \vee R_n \to MAR$	$R_n + K + CI \to R_n$	K11
	II	$K \vee M \to MAR$	$M + K + CI \to AT$	K12
	III	$(\overline{AT} \vee K) + (AT \wedge K) + CI \to AT$		K13
2	I	$(AC \wedge K) - 1 + CI \to R_n$	(see Note 1)	K21
	II	$(AC \wedge K) - 1 + CI \to AT$	(see Note 1)	K22
	III	$(I \wedge K) - 1 + CI \to AT$	(see Note 1)	K23
3	I	$R_n + (AC \wedge K) + CI \to R_n$		K31
	II	$M + (AC \wedge K) + CI \to AT$		K32
	III	$AT + (I \wedge K) + CI \to AT$		K33
4	I	$CI \vee (R_n \wedge AC \wedge K) \to CO$	$R_n \wedge (AC \wedge K) \to R_n$	K41
	II	$CI \vee (M \wedge AC \wedge K) \to CO$	$M \wedge (AC \wedge K) \to AT$	K42
	III	$CI \vee (AT \wedge I \wedge K) \to CO$	$AT \wedge (I \wedge K) \to AT$	K43
5	I	$CI \vee (R_n \wedge K) \to CO$	$K \wedge R_n \to R_n$	K51
	II	$CI \vee (M \wedge K) \to CO$	$K \wedge M \to AT$	K52
	III	$CI \vee (AT \wedge K) \to CO$	$K \wedge AT \to AT$	K53
6	I	$CI \vee (AC \wedge K) \to CO$	$R_n \vee (AC \wedge K) \to R_n$	K61
	II	$CI \vee (AC \wedge K) \to CO$	$M \vee (AC \wedge K) \to AT$	K62
	III	$CI \vee (I \wedge K) \to CO$	$AT \vee (I \wedge K) \to AT$	K63
7	I	$CI \vee (R_n \wedge AC \wedge K) \to CO$	$R_n \overline{\odot} (AC \wedge K) \to R_n$	K71
	II	$CI \vee (M \wedge AC \wedge K) \to CO$	$M \overline{\odot} (AC \wedge K) \to AT$	K72
	III	$CI \vee (AT \wedge I \wedge K) \to CO$	$AT \overline{\odot} (I \wedge K) \to AT$	K73

2's complement arithmetic adds 111 . . . 11 to perform subtraction of 000 . . . 01.

R_n includes T and AC as source and destination registers in R-group 1 micro functions.

Standard arithmetic carry output values are generated in F-group 0, 1, 2 and 3 instructions.

MBOL	MEANING
K, M	Data on the I, K, and M busses, respectively
. LI	Data on the carry input and left input, respectively
, RO	Data on the carry output and right output, respectively
R_n	Contents of register n including T and AC (R-Group I)
AC	Contents of the accumulator
AT	Contents of AC or T, as specified
AR	Contents of the memory address register
, H	As subscripts, designate low and high order bit, respectively
+	2's complement addition
−	2's complement subtraction
$\wedge$	Logical AND
$\vee$	Logical OR
$\overline{\odot}$	Exclusive-NOR
$\to$	Deposit into

Table B.3 Central Processor Array Mnemonics - Unspecified K-bus Microfunctions.

F-GROUP	R-GROUP	K-BUS = 00 MICRO-FUNCTION	MNEMONIC	K-BUS = 11 MICRO-FUNCTION	MNEMONIC
0	I	$R_n + CI \to R_n, AC$	ILR	$AC + R_n + CI \to R_n, AC$	ALR
	II	$M + CI \to AT$	ACM	$M + AC + CI \to AT$	AMA
	III	$AT_L \to RO$ $AT_H \to AT_L$ $LI \to AT_H$	SRA		–
1	I	$R_n \to MAR$ $R_n + CI \to R_n$	LMI	$11 \to MAR$ $R_n - 1 + CI \to R_n$	DSM [?]
	II	$M \to MAR$ $M + CI \to AT$	LMM	$11 \to MAR$ $M - 1 + CI \to AT$	LDM [?]
	III	$\overline{AT} + CI \to AT$	CIA	$AT - 1 + CI \to AT$	DCA
2	I	$CI - 1 \to R_n$ (See Note 1)	CSR	$AC - 1 + CI \to R_n$ (See Note 1)	SDR
	II	$CI - 1 \to AT$ (See Note 1)	CSA	$AC - 1 + CI \to AT$ (See Note 1)	SDA
	III	(See CSA above)	–	$I - 1 + CI \to AT$	LDI
3	I	$R_n + CI \to R_n$	INR	$AC + R_n + CI \to R_n$	ADR
	II	(See ACM above)	–	(See AMA above)	–
	III	$AT + CI \to AT$	INA	$I + AT + CI \to AT$	AIA
4	I	$CI \to CO$ $0 \to R_n$	CLR	$CI \vee (R_n \wedge AC) \to CO$ $R_n \wedge AC \to R_n$	ANR
	II	$CI \to CO$ $0 \to AT$	CLA	$CI \vee (M \wedge AC) \to CO$ $M \wedge AC \to AT$	ANM
	III	(See CLA above)	–	$CI \vee (AT \wedge I) \to CO$ $AT \wedge I \to AT$	ANI
5	I	(See CLR above)	–	$CI \vee R_n \to CO$ $R_n \to R_n$	TZR
	II	(See CLA above)	–	$CI \vee M \to CO$ $M \to AT$	LTM
	III	(See CLA above)	–	$CI \vee AT \to CO$ $AT \to AT$	TZA
6	I	$CI \to CO$ $R_n \to R_n$	NOP	$CI \vee AC \to CO$ $R_n \vee AC \to R_n$	ORR
	II	$CI \to CO$ $M \to AT$	LMF	$CI \vee AC \to CO$ $M \vee AC \to AT$	ORM
	III	(See NOP above)	–	$CI \vee I \to CO$ $I \vee AT \to AT$	ORI
7	I	$CI \to CO$ $\overline{R_n} \to R_n$	CMR	$CI \vee (R_n \wedge AC) \to CO$ $R_n \overline{\oplus} AC \to R_n$	XNR
	II	$CI \to CO$ $\overline{M} \to AT$	LCM	$CI \vee (M \wedge AC) \to CO$ [?] $M \overline{\oplus} AC \to AT$	XNM [?]
	III	$CI \to CO$ $\overline{AT} \to AT$	CMA	$CI \vee (AT \wedge I) \to CO$ $I \overline{\oplus} AT \to AT$	XNI

NOTES:

1. 2's complement arithmetic adds 111 . . . 11 to perform subtraction of 000 . . . 01.
2. R_n includes T and AC as source and destination registers in R-group 1 micro-functions.
3. Standard arithmetic carry output values are generated in F-group 0, 1, 2 and 3 instructions.

SYMBOL	MEANING
I, K, M	Data on the I, K, and M busses, respectively
CI, LI	Data on the carry input and left input, respectively
CO, RO	Data on the carry output and right output, respectively
R_n	Contents of register n including T and AC (R-Group I)
AC	Contents of the accumulator
AT	Contents of AC or T, as specified
MAR	Contents of the memory address register
L, H	As subscripts, designate low and high order bit, respectively
+	2's complement addition
–	2's complement subtraction
$\wedge$	Logical AND
$\vee$	Logical OR
$\overline{\oplus}$	Exclusive-NOR
$\to$	Deposit into

Table B.4 Central Processor Array Mnemonics - All-Zero and All-One K-bus Microfunctions.

Unconditional jumps

Jump mnemonic	Action taken
INC	Fetch the next microinstruction from micromemory.
JMP	Jump to the address in the MIXED field of the microinstruction.
JMPD	Jump to the address specified by the EXC-address latch. The external control commands and the associated jump addresses are summarized in Table B.5(a).
JSR	Jump to the address in the MIXED field of the microinstruction. Store the return address on the stack.
JSRD	Jump to the address specified by the EXC-address latch (refer JMPD above). Store the return address on the stack.
RTS	Return from subroutine. That is, jump to the address at the top of the stack.
HLT	Continuously execute the current microinstruction.

Conditional jumps

Jump mnemonic	Action taken
INCT	Continuously execute the current microinstruction whilst condition FALSE. When condition goes TRUE, then INC.
INCF	Continuously execute the current microinstruction whilst condition TRUE. When condition goes FALSE, then INC.
JMPT	If condition TRUE then JMP (i.e. jump to the address in the MIXED field). If condition FALSE then INC.
JMPF	If condition TRUE then INC. If condition FALSE then JMP.

Table B.5 Jumps

External Control

Instruction	Micromemory address accessed by JMPD or JSRD
EXC 0014	1
EXC 0114	3
EXC 0214	5
EXC 0314	7
EXC 0414	9
EXC 0514	11
EXC 0614	13
EXC 0714	15
EXCE 0014	17
EXCE 0114	19
EXCE 0214	21
EXCE 0314	23
EXCE 0414	25
EXCE 0514	27
EXCE 0614	29
EXCE 0714	31

Table B.5(a) External Control Instructions and their associated Micromemory Addresses

Multiplexer input line	Mnemonics	Description
0	ZERO	Reserved for the assembler (used for unconditional jumps).
1	ONE	Reserved for the assembler (used for unconditonal jumps).
2	CO	CO is the most significant carry-out of the CPA.
3	EXC	EXC, DTOX or DTIX will go TRUE if an external control, single-word output-transfer or single-word input transfer instruction is respectively executed by the Varian.
4	DTOX	
5	DTIX	
6	DMA.FIN	DMA.FIN is set on the completion of a DMA transfer or interrupt.
7	spare	
8	COMMAND	COMMAND is the logical OR of EXC, DTOX and DTIX.
9	FORMAT	FORMAT is TRUE when the consoles format switch is ON.
10	SIGNBIT	SIGNBIT is the most significant bit of the CPA's accumulator.
11	spare	
12	spare	
13	SYNC	SYNC is used to indicate that the data from disc is syncronized.
14	EQ.16	EQ.16 will go TRUE when a word of data is ready to be transferred from the drive, or when the drive is ready to receive a word of data.
15	spare	
16	spare	
17	SEEKRDY	SEEKRDY, INDEX, SECTOR, SEEKERR, ONLINE, UNSAFE, WRITE.CUR, UNTSEL and ATTEN signals come directly from the disc drive.
18	INDEX	
19	SECTOR	
20	SEEKERR	

Table B.6 Input-Control Signals

Multiplexer input line	Mnemonics	Description
21	spare	
22	ONLINE	
23	UNSAFE	
24	spare	
25	spare	
26	WRITE.CUR	
27	SGLDEN	SGLDEN will be TRUE if the drive is a single density drive and will be FALSE if the drive is double density.
28	UNTSEL	
29	ATTEN	
30	spare	
31	spare	

Table B.6 Input-Control Signals (Continued)

Microinstruction Bit Set	Mnemonic	Port Type	Description
0	READ.LM	A	LM address from MAR; LM data onto M-bus.
1	WRITE.LM	A	LM address from MAR; LM data from AC.
2	VAR.DIN	A	Data from Varian onto M-bus.
3	DISC.DIN	A	Data from disc onto M-bus.
4	DRIVE1	A	DRIVE1, DRIVE2 and DRIVE4 are
5	DRIVE2	A	used in selecting 1 of 8 possible
6	DRIVE4	A	drives. Drives are numbered 0 to 7.
7	DRIVE.CLR	A	Clear the above 3 latches. Selects drive 0.
8	SECTOR.NO	A	Sector number onto M-bus.
9	spare	A	
0	spare	B	
1	INTERRUPT	B	Enable the INTERRUPT latch.
2	DMA.OUT	B	Enable the DMA.OUT latch.
3	DMA.IN	B	Enable the DMA.IN latch.
4	VAR.DOUT	B	Gate data from the MAR into the VAR.DOUT latch.
5	READY	B	READY, EOF or ERR sets the
6	EOF	B	corresponding Sense latch.
7	ERR	B	This latch can be tested by a Sense (SEN) command from the Varian.
8	CLR.COMMAND	B	Clear the command latches.
9	spare	B	
0	READ.DATA	C	Enable/disable the READ.DATA latch.
1	READ.CLK	C	Enable/disable the READ.CLK latch.
2	CLR.EQ16	C	Clear the EQ.16 latch.
3	DISC.DOUT	C	Gate the data from the MAR into the DISC.DOUT latch.
4	WRITE	C	Enable/disable the WRITE latch.

Table B.7 Output-Control Signals

Microinstruction Bit Set	Mnemonic	Port Type	Description
5	SYNC.EN	C	Enable the SYNC latch.
6	SYNC.CLR	C	Clear the SYNC latch.
7	spare	C	
8	BUS.EN	C	Allow Bus latches to be enabled/disabled from AC.
9	TAG.EN	C	Allow Tag latches to be enabled/disabled from AC.

Notes: LM = Local memory
MAR = memory address register
AC = accumulator

Table B.7 Output-control Signals (Continued)

Constant Name	Constant Value	Microinstruction Bit Set
BUS0	128	7
BUS1	64	6
BUS2	32	5
BUS3	16	4
BUS4	8	3
BUS5	4	2
BUS6	2	1
BUS7	1	0
CONTROL	256	8
CYL	512	9
HEAD	1024	10
DIFF	2048	11

Table B.8 Special Constants

Error Number	Cause of Error
5	A label found in Pass 2 which is not in the symbol table. Causes of error: (i) Pass 1 has not been done; or (ii) an external label references a label which is not in the program.
10	Invalid opcode mnemonic. Refer Table B.3 and Table B.4 for valid opcode mnemonics.
20	Construction of CPA field incorrect.
30	Invalid register name.
50	Invalid jump name.
60	A conditional jump is specified, but there is no MUX field.
70	A conditional jump is specified, but the MUX field is invalid. Table B.6 for valid MUX mnemonics.
80	Unrecognisable MIXED field. Causes of error: (i) If a jump address, the label does not occur in the current external block. This error will only occur with internal lables - refer Error 5 above; or (ii) if an output-control signal is required, the signal name is invalid. Refer Table B.7 for valid output-control signal mnenomics.
85	There is an illegal character after the MIXED field. Comments must begin with a #.
90	The constant value in the MIXED field contains illegal characters. If the first digit is a 0, then the number must be octal (i.e. cannot contain the digits 8 or 9). Note that all labels must start with an alphabetic character or a fullstop.

Table B.9 Pass 2 Errors

Error Number	Cause of Error
95	A label has the same name as an output-control signal. This is illegal. Table B.7 summarizes the output-control signal mnemonics.
100	The number in the MIXED field is not in the range $0 \leq \text{Number} \leq 2^{12}$ 4096.
110	Illegal syntax for the output-control signal mnemonics.
130	The MIXED field contains two output-control signal mnemonics which are of different port types. Only output-control signals of the same port type can be used together.

Table B.9 Pass 2 Errors (Continued)

APPENDIX C : HARDWARE

C.1 Comparison between the Intel 3000 and Advanced Micro Devices 2900 Central Processing Elements

The 2900 series central processing element is the 2901[1], and the 3000 series central processing element is the 3002[2].

Both processors have similar operating speeds (in the region of 100 nano-seconds). The main difference in the processors offered is the number of bits/slice. The 3002's 2 bit/slice allows the processor to have more I/O port facilities than its 4 bit/slice counterpart, the 2901. The 3002 has 3 input ports (K, I and M ports) and 2 output ports (A and D ports).

The 3002 has the following advantages over the 2901.

(i) The large number of I/O ports provided by the 3002 allows the external logic to be easily interfaced to the processor, thus making the hardware design simpler than would be the case with the 2901 with its single input port and single output port.
(ii) Data manipulation in the 3002 is therefore much easier than in the 2901.

The 2901 has the following advantages over the 3002.

(i) If the 2901 is used as opposed to the 3002, only half the number of CPE's will be required for a particular application.
(ii) The 2901 has more registers than the 3002 - 17 vs. 13.
(iii) Although the 3002 has a richer instruction set than the 2901, many basic instructions are missing. For example, in the 3001, subtraction can only be done by complimenting and adding, and register to register transfers require two microinstructions.

In summary, the 3002 is better in data manipulation than the 2901, but the 2901 is better in number crunching. As the HLDC will require a large amount of data manipulation (including the gating of constants into the central processor array from micromemory), the Intel 3002 is chosen.

[1] M2900 Bipolar (TTL) Processor Family, Motorola Semiconductor Products, Inc. (1976), pp. 18-26.

[2] 3002 Central Processing Element, Intel Corporation, Santa Clara, California (1975), pp. 1-16.

C.2 The Central Processor Array - Functional Description

The central processing array (CPA) is a 16 bit processor made up of eight 2-bit slice Intel 3002 central processing elements. The functional organization of a central processing element (CPE) is illustrated in Figure C.1. Typical CPE configurations to form CPA's are illustrated in Figure C.2 and Figure C.3.

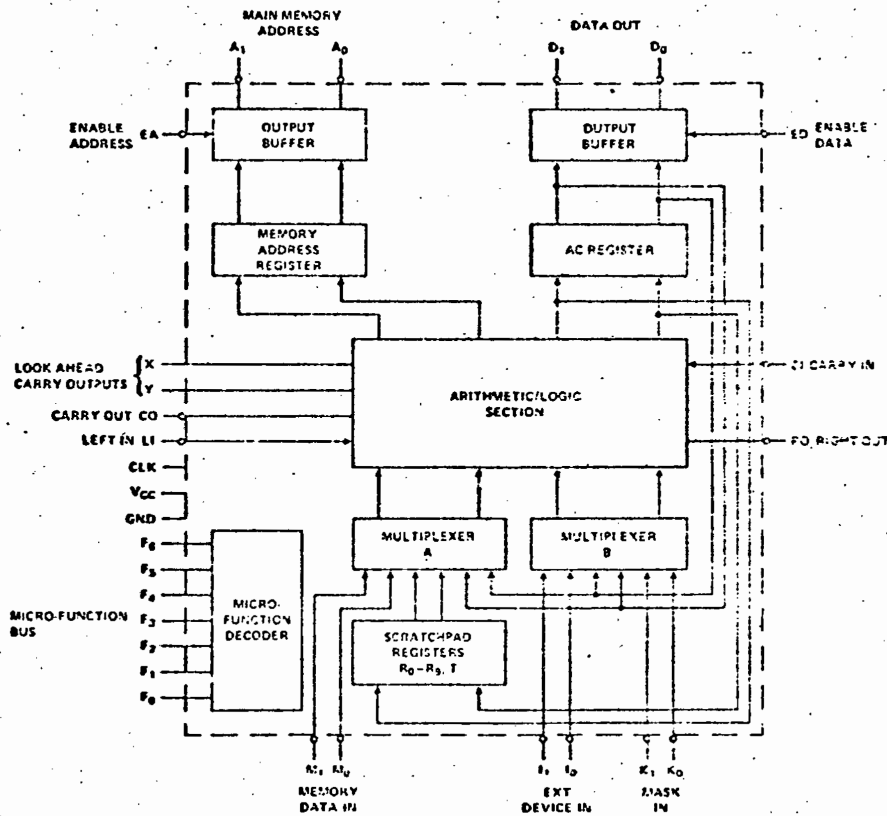


Figure C.1 3002 Block Diagram

The arithmetic and logic section (ALS) of the CPA is capable of a variety of simple arithmetic and logic operations including logical shifting. In order to effect these operations, certain input/output lines are provided for communicating carries and shifts between CPE's. Arithmetic operations are implemented by using the status on the carry-in (CI) and carry-out (CO) lines. Shifting is implemented by the provision of left-in (LI) and right-out (RO) lines. Shift operations are enabled by connecting the right-out lines of the CPE to the left-in lines of the next CPE as illustrated

MEMORY ADDRESS BUS (2N LINES)
DATA BUS TO MEMORY (2N LINES)
CLOCK
F_4-F_6
A D
F_4-F_6 CLK
CO CI
3002
LI RO
F_0-F_3
I M K
3002
3002
CARRY FROM 3001
F_0-F_3
CARRY TO 3001
MICROPROGRAM CONSTANT/MASK INPUT BUS (2N LINES)
DATA BUS FROM MEMORY (2N LINES)
EXTERNAL DATA BUS (2N LINES)

Figure C.2 Ripple-Carry Configuration (N 3002 CPE's).

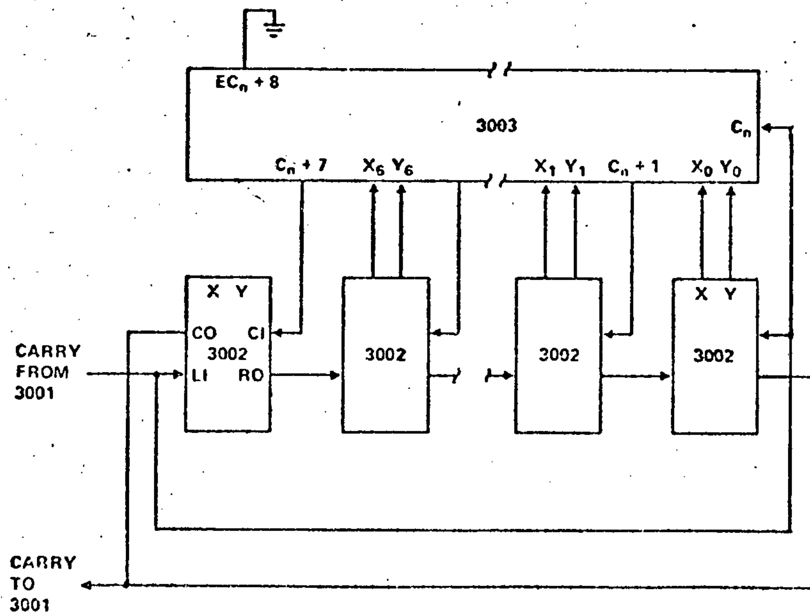


Figure C.3 Carry Look-Ahead Configuration (16 Bit Array).

in Figure C.2. Carries can be performed in one of two ways :

(i) The carry-out (CO) of one CPE can be connected to the carry-in of the next CPE to provide for the normal ripple carry propogation. Figure C.2 illustrates this configuration.

(ii) The look-ahead carry outputs X and Y of each CPE, and the carry-out line of each CPE can be fed into a ripple carry generator (Intel 3003) to produce the appropriate carry-ins for the CPE's as illustrated in Figure C.3.

The typical propogation delay time between carry-in and carry-out for a CPE is 14 nanoseconds, whilst the maximum propogation delay time is 25 nanoseconds. The ripple carry configuration will consequently produce a typical delay time of 112 nanoseconds for the 16 bit CPA, whilst the maximum delay time for the array will be 200 nanoseconds. This necessitates the use of the high-speed look-ahead carry generator (Intel 3003) which will give a typical propogation delay of 13 nanoseconds from carry-in to the outputs, with the maximum delay being 40 nanoseconds. It is desirable to have the least significant carry-in of the CPA under program control to allow for incrementing in the CPA.

There are seven microfunction bus (F-bus) inputs to each CPE, designated F0 to F6. Encoded microfunctions are applied to the CPE on this F-bus, the microfunctions being decoded internally in the CPE by a microfunction decoder. The decoder selects the arithmetic logic function to be performed, generates the scratch-pad register address, and controls the multiplexers. The result of the operation is deposited into the appropriate registers. There are 13 registers which are made up of 11 scratch-pad registers (R0 to R9 and T), the accumulator (AC) and the memory address register (MAR).

The CPA is capable of performing a large number of functions. These functions are summarized in Table C.1. The microfunction to be performed is determined from the function group (F-group) and register group (R-group) selected by data on the F-bus. The F-group is specified by the F-bus bits F4, F5 and F6, and the R-group is specified by the F-bus bits F0, F1, F2 and F3. The F-group and R-group formats are summarized in Table C.2.

F-GROUP	R-GROUP	MICRO-FUNCTION		MNEMONIC
	I	$R_n + (AC \wedge K) + CI \rightarrow R_n, AC$		K01
0	II	$M + (AC \wedge K) + CI \rightarrow AT$		K02
	III	$AT_L \wedge (\overline{I_L \wedge K_L}) \rightarrow RO$ $[AT_L \wedge (I_L \wedge K_L)] \vee [AT_H \vee (I_H \wedge K_H)] \rightarrow AT_L$	$LI \vee [(I_H \wedge K_H) \wedge AT_H] \rightarrow AT_H$	K03
	I	$K \vee R_n \rightarrow MAR$	$R_n + K + CI \rightarrow R_n$	K11
1	II	$K \vee M \rightarrow MAR$	$M + K + CI \rightarrow AT$	K12
	III	$(\overline{AT} \vee K) + (AT \wedge K) + CI \rightarrow AT$		K13
	I	$(AC \wedge K) - 1 + CI \rightarrow R_n$		K21
2	II	$(AC \wedge K) - 1 + CI \rightarrow AT$	(see Note 1)	K22
	III	$(I \wedge K) - 1 + CI \rightarrow AT$		K23
	I	$R_n + (AC \wedge K) + CI \rightarrow R_n$		K31
3	II	$M + (AC \wedge K) + CI \rightarrow AT$		K32
	III	$AT + (I \wedge K) + CI \rightarrow AT$		K33
	I	$CI \vee (R_n \wedge AC \wedge K) \rightarrow CO$	$R_n \wedge (AC \wedge K) \rightarrow R_n$	K41
4	II	$CI \vee (M \wedge AC \wedge K) \rightarrow CO$	$M \wedge (AC \wedge K) \rightarrow AT$	K42
	III	$CI \vee (AT \wedge I \wedge K) \rightarrow CO$	$AT \wedge (I \wedge K) \rightarrow AT$	K43
	I	$CI \vee (R_n \wedge K) \rightarrow CO$	$K \wedge R_n \rightarrow R_n$	K51
5	II	$CI \vee (M \wedge K) \rightarrow CO$	$K \wedge M \rightarrow AT$	K52
	III	$CI \vee (AT \wedge K) \rightarrow CO$	$K \wedge AT \rightarrow AT$	K53
	I	$CI \vee (AC \wedge K) \rightarrow CO$	$R_n \vee (AC \wedge K) \rightarrow R_n$	K61
6	II	$CI \vee (AC \wedge K) \rightarrow CO$	$M \vee (AC \wedge K) \rightarrow AT$	K62
	III	$CI \vee (I \wedge K) \rightarrow CO$	$AT \vee (I \wedge K) \rightarrow AT$	K63
	I	$CI \vee (R_n \wedge AC \wedge K) \rightarrow CO$	$R_n \overline{\oplus} (AC \wedge K) \rightarrow R_n$	K71
7	II	$CI \vee (M \wedge AC \wedge K) \rightarrow CO$	$M \overline{\oplus} (AC \wedge K) \rightarrow AT$	K72
	III	$CI \vee (AT \wedge I \wedge K) \rightarrow CO$	$AT \overline{\oplus} (I \wedge K) \rightarrow AT$	K73

NOTES:

1. 2's complement arithmetic adds 111 . . . 11 to perform subtraction of 000 . . . 01.
2. R_n includes T and AC as source and destination registers in R-group 1 micro-functions.
3. Standard arithmetic carry output values are generated in F-group 0, 1, 2 and 3 instructions.

SYMBOL	MEANING
I, K, M	Data on the I, K, and M busses, respectively
CI, LI	Data on the carry input and left input, respectively
CO, RO	Data on the carry output and right output, respectively
R_n	Contents of register n including T and AC (R-Group I)
AC	Contents of the accumulator
AT	Contents of AC or T, as specified
MAR	Contents of the memory address register
L, H	As subscripts, designate low and high order bit, respectively
+	2's complement addition
−	2's complement subtraction
$\wedge$	Logical AND
$\vee$	Logical OR
$\overline{\oplus}$	Exclusive-NOR
$\rightarrow$	Deposit into

Table C.1 Microfunctions

FUNCTION GROUP	F_6	5	4
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

REGISTER GROUP	REGISTER	F_3	2	1	0
I	R_0	0	0	0	0
	R_1	0	0	0	1
	R_2	0	0	1	0
	R_3	0	0	1	1
	R_4	0	1	0	0
	R_5	0	1	0	1
	R_6	0	1	1	0
	R_7	0	1	1	1
	R_8	1	0	0	0
	R_9	1	0	0	1
	T	1	1	0	0
	AC	1	1	0	1
II	T	1	0	1	0
	AC	1	0	1	1
III	T	1	1	1	0
	AC	1	1	1	1

Table C.2 Function and Register Group Formats.

F-GROUP	R-GROUP	K-BUS = 00 MICRO-FUNCTION	MNEMONIC	K-BUS = 11 MICRO-FUNCTION	MNEMONIC
0	I	$R_n + CI \rightarrow R_n, AC$	ILR	$AC + R_n + CI \rightarrow R_n, AC$	ALR
	II	$M + CI \rightarrow AT$	ACM	$M + AC + CI \rightarrow AT$	AMA
	III	$AT_L \rightarrow RO \quad AT_H \rightarrow AT_L \quad LI \rightarrow AT_H$	SRA		–
1	I	$R_n \rightarrow MAR \quad R_n + CI \rightarrow R_n$	LMI	$11 \rightarrow MAR \quad R_n - 1 + CI \rightarrow R_n$	DSM
	II	$M \rightarrow MAR \quad M + CI \rightarrow AT$	LMM	$11 \rightarrow MAR \quad M - 1 + CI \rightarrow AT$	LDM
	III	$\overline{AT} + CI \rightarrow AT$	CIA	$AT - 1 + CI \rightarrow AT$	DCA
2	I	$CI - 1 \rightarrow R_n$ (See Note 1)	CSR	$AC - 1 + CI \rightarrow R_n$ (See Note 1)	SDR
	II	$CI - 1 \rightarrow AT$ (See Note 1)	CSA	$AC - 1 + CI \rightarrow AT$ (See Note 1)	SDA
	III	(See CSA above)	–	$I - 1 + CI \rightarrow AT$	LDI
3	I	$R_n + CI \rightarrow R_n$	INR	$AC + R_n + CI \rightarrow R_n$	ADR
	II	(See ACM above)	–	(See AMA above)	–
	III	$AT + CI \rightarrow AT$	INA	$I + AT + CI \rightarrow AT$	AIA
4	I	$CI \rightarrow CO \quad 0 \rightarrow R_n$	CLR	$CI \vee (R_n \wedge AC) \rightarrow CO \quad R_n \wedge AC \rightarrow R_n$	ANR
	II	$CI \rightarrow CO \quad 0 \rightarrow AT$	CLA	$CI \vee (M \wedge AC) \rightarrow CO \quad M \wedge AC \rightarrow AT$	ANM
	III	(See CLA above)	–	$CI \vee (AT \wedge I) \rightarrow CO \quad AT \wedge I \rightarrow AT$	ANI
5	I	(See CLR above)	–	$CI \vee R_n \rightarrow CO \quad R_n \rightarrow R_n$	TZR
	II	(See CLA above)	–	$CI \vee M \rightarrow CO \quad M \rightarrow AT$	LTM
	III	(See CLA above)	–	$CI \vee AT \rightarrow CO \quad AT \rightarrow AT$	TZA
6	I	$CI \rightarrow CO \quad R_n \rightarrow R_n$	NOP	$CI \vee AC \rightarrow CO \quad R_n \vee AC \rightarrow R_n$	ORR
	II	$CI \rightarrow CO \quad M \rightarrow AT$	LMF	$CI \vee AC \rightarrow CO \quad M \vee AC \rightarrow AT$	ORM
	III	(See NOP above)	–	$CI \vee I \rightarrow CO \quad I \vee AT \rightarrow AT$	ORI
7	I	$CI \rightarrow CO \quad \overline{R_n} \rightarrow R_n$	CMR	$CI \vee (R_n \wedge AC) \rightarrow CO \quad R_n \overline{\oplus} AC \rightarrow R_n$	XNR
	II	$CI \rightarrow CO \quad \overline{M} \rightarrow AT$	LCM	$CI \vee (M \wedge AC) \rightarrow CO \quad M \overline{\oplus} AC \rightarrow AT$	XNM
	III	$CI \rightarrow CO \quad \overline{AT} \rightarrow AT$	CMA	$CI \vee (AT \wedge I) \rightarrow CO \quad I \overline{\oplus} AT \rightarrow AT$	XNI

NOTES:

1. 2's complement arithmetic adds 111 . . . 11 to perform subtraction of 000 . . . 01.
2. R_n includes T and AC as source and destination registers in R-group 1 micro-functions.
3. Standard arithmetic carry output values are generated in F-group 0, 1, 2 and 3 instructions.

SYMBOL	MEANING
I, K, M	Data on the I, K, and M busses, respectively
CI, LI	Data on the carry input and left input, respectively
CO, RO	Data on the carry output and right output, respectively
R_n	Contents of register n including T and AC (R-Group I)
AC	Contents of the accumulator
AT	Contents of AC or T, as specified
MAR	Contents of the memory address register
L, H	As subscripts, designate low and high order bit, respectively
+	2's complement addition
–	2's complement subtraction
$\wedge$	Logical AND
$\vee$	Logical OR
$\overline{\oplus}$	Exclusive-NOR
$\rightarrow$	Deposit into

Table C.3 All-Zero and All-One Microfunctions

The K-bus is an integral part of each microfunction and the ability of the K-bus to mask inputs to the ALS increases the versitility of the CPA. For example, the K-bus can be used during arithmetic operations to mask portions of the field being operated upon. The main function of the K-bus is that of supplying constants to the CPA from the microprogram. In this instance, the K-bus will contain the constant values to be gated into the CPA. The microfunctions most commonly used, however, are those microfunctions where the K-bus is all zeroes or all ones. These microfunctions are summarized in Table C.3.

C.3 The MIXED field

Certain data supplied by the microinstruction is multiplexed together to form one microinstruction field. This field is called the MIXED field. The advantage of this organization is that it reduces the number of memory elements required (fast memory is expensive), and it allows simple implementation. The number of extra bits required by not multiplexing this field can be easily calculated. The MIXED field supplies the following three types of data :

(i) Constants to the CPA;
(ii) Micromemory addresses to the MCU; and
(iii) Data to the 3 output ports.

The current microinstruction format uses 16 bits for the MIXED field, and for the control of data on the MIXED field. A comparative system which does not multiplex the above data will require 12 bits for constants to the CPA, 12 bits for addresses to the MCU, and 36 bits for the output-control signals. Consequently, this system would require a microinstruction which is 44 bits wider. Such a system will require an extra 44 memory elements per 1k of micromemory (i.e. the Intel 2102A-1 memory elements are used). Extra components will also be required for the pipeline register.

The speed advantage of the non-multiplexed system over the multiplexed system would be negligible. An analysis was made on a program used to format the disc. This program occupies about 1k of micromemory, and makes extensive use of the MIXED field (i.e. the format routine must perform the full range of disc drive operations). The analysis showed no more than 5

microinstructions would be saved if a non-multiplexed microinstruction format was used.

C.4 Microinstruction Fetch Propogation Delay

Data necessary in the microinstruction fetch operation is initially clocked into either the pipeline register or the microprogram control unit on one of the clock edges. The microinstruction fetch propogation delay is obtained by asking the following question : What is the minimum time which must be allowed before the data is again stable at the inputs of the pipeline register and microprogram control unit? This period of time is the microinstruction fetch propogation delay time, since after this period of time, data can again be clocked into the pipeline register and microprogram control unit.

The path where propogation delays are significant in the fetch cycle is illustrated in Figure C.4 (the path being illustrated by the use of broad lines). Using this path, the microinstruction fetch propogation delay can be calculated. The microinstruction fetch propogation delay, t_{FETCH}, will be the sum of the propogation delays of the individual components in the path. (The propogation delay associated with the conductors between the components is minimal, and can be ignored[3]).

t_{FETCH} = sum of the propogation delay times of the components:
74128, 2102A-1, 74S04, 74S04, 74150, 74S00, 74S86, 74S04,
74S175, 74S08, 2909, 74LS368.

Each of these components will have associated with it a typical and maximum propogation delay time. These times are dependent on the environment that the components are operated in. The most significant factors affecting these propogation delays being temperature and power supply fluctuations. Two propogation delay times can therefore be associated with t_{FETCH}. They are t_{FM} (the maximum fetch propogation delay time) and t_{FT} (the typical fetch propogation delay time). Table C.5 gives the maximum and typical propogation delays for the components. The most significant component propogation delay is the delay associated with the micromemory elements. The Intel 2102A-1 memory elements are used for the micromemory. These memory elements are 1k by 1 bit with a guaranteed cycle time of 200

[3]Garry Fielland, "Series 3000 System Timing Considerations", pp. 31-38.

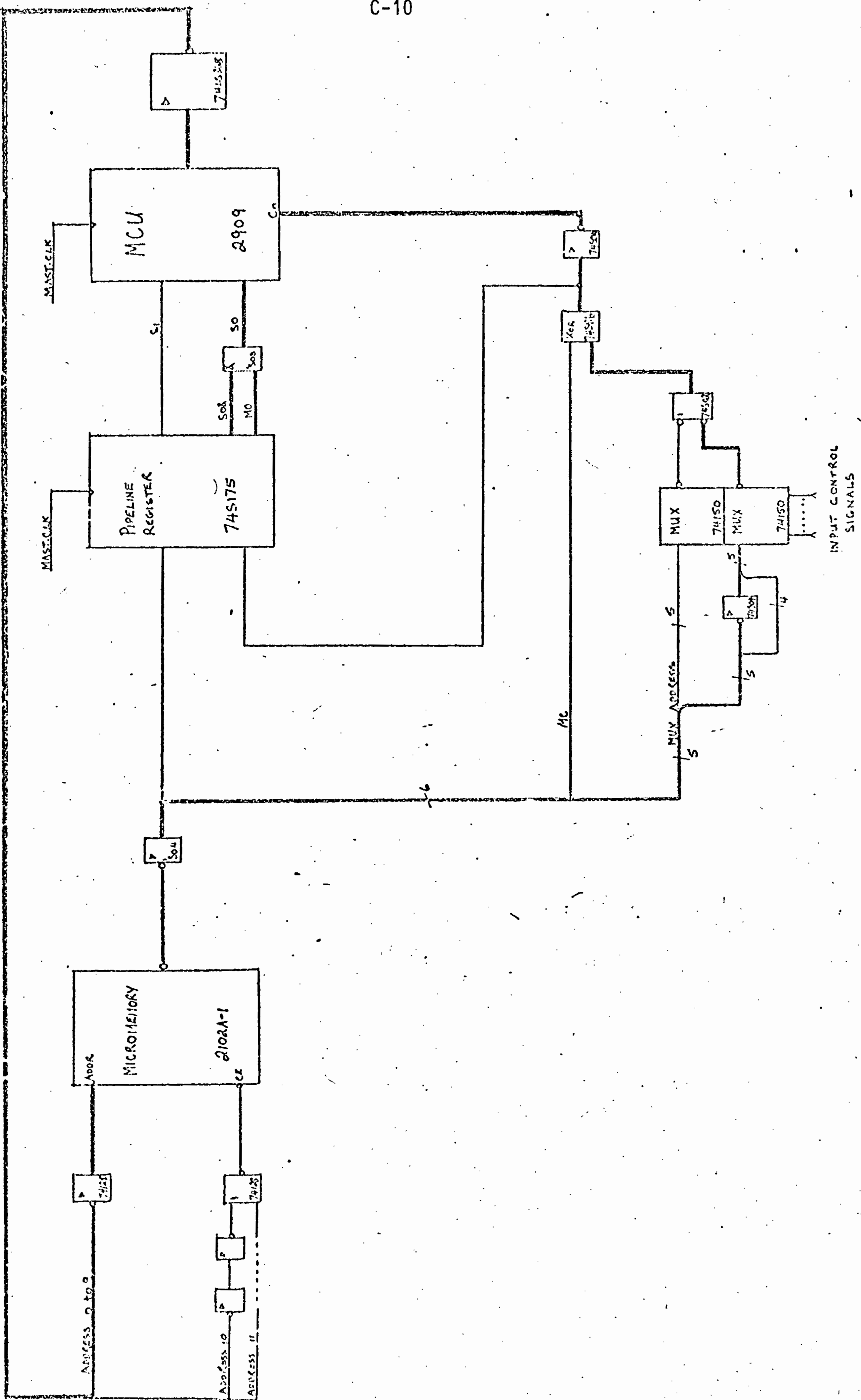


Figure C.4 Microinstruction Fetch Circuitry - Broad Lines Indicate the Path of Maximum Delay.

nanoseconds over the specified temperature range[4]. As this is a guaranteed figure, the average component will perform considerably better, especially in a non-hostile environment, and a typical access time in the region of 130 to 150 nanoseconds can be expected.

The maximum and typical propogation delay times can now be calculated.

t_{FM} = 9 + 200 + 5 + 5 + 35 + 5 + 10,5 + 5 + 17 + 7,5 + 40 + 18
= 357 nanoseconds.

t_{FT} = 6 + 150 + 3 + 3 + 23 + 3 + 7 + 3 + 11,5 + 5 + 40 + 12
= 266,5 nanoseconds.

C.5 Microinstruction Execution Propogation Delay

The only delay which must be considered in calculating the microinstruction execution propogation delay, is the delay associated with the CPA field of the microinstruction. The CPA is interfaced to a look-ahead carry generator (Intel 3003) to minimize propogation delays from carry-in to carry-out. The maximum propogation delay path from carry-in to carry-out when using the look-ahead carry generator is found to be 150 nanoseconds. The typical propogation delay being 100 nanoseconds. This is the most significant propogation delay associated with the CPA field.

C.6 Port Timing

The output-control signals can be classified into three different groups :

(i) The output-control signal for writing to local memory (WRITE.LM);
(ii) The output-control signals for gating data onto the M-bus (VAR.DIN, DISC.DIN, READ.LM and SECTOR.NO);
(iii) The remaining output-control signals either load or clear latches.

Propogation delays in the system can cause glitches to occur. Consider the following program sequence :

```
j     : K11(A)   *   1
j + 1 : *        *   SYNC.EN
```

The timing diagrams for this program sequence is illustrated in Figure C.5.

[4]M.C. Sole, "An Optimal Real Time Language Processor", unpublished M.Sc. Thesis, Dept. of Electrical Engineering, University of Cape Town (1978), p. 58.

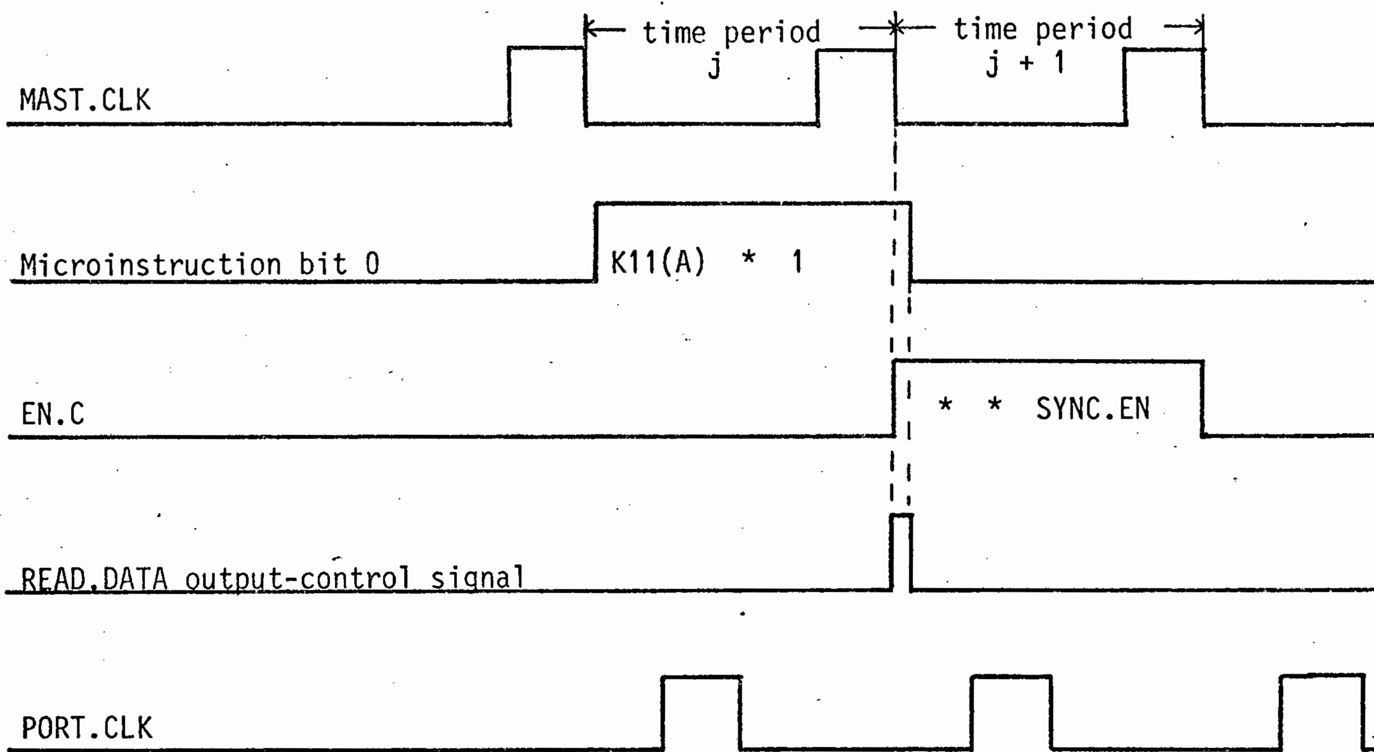


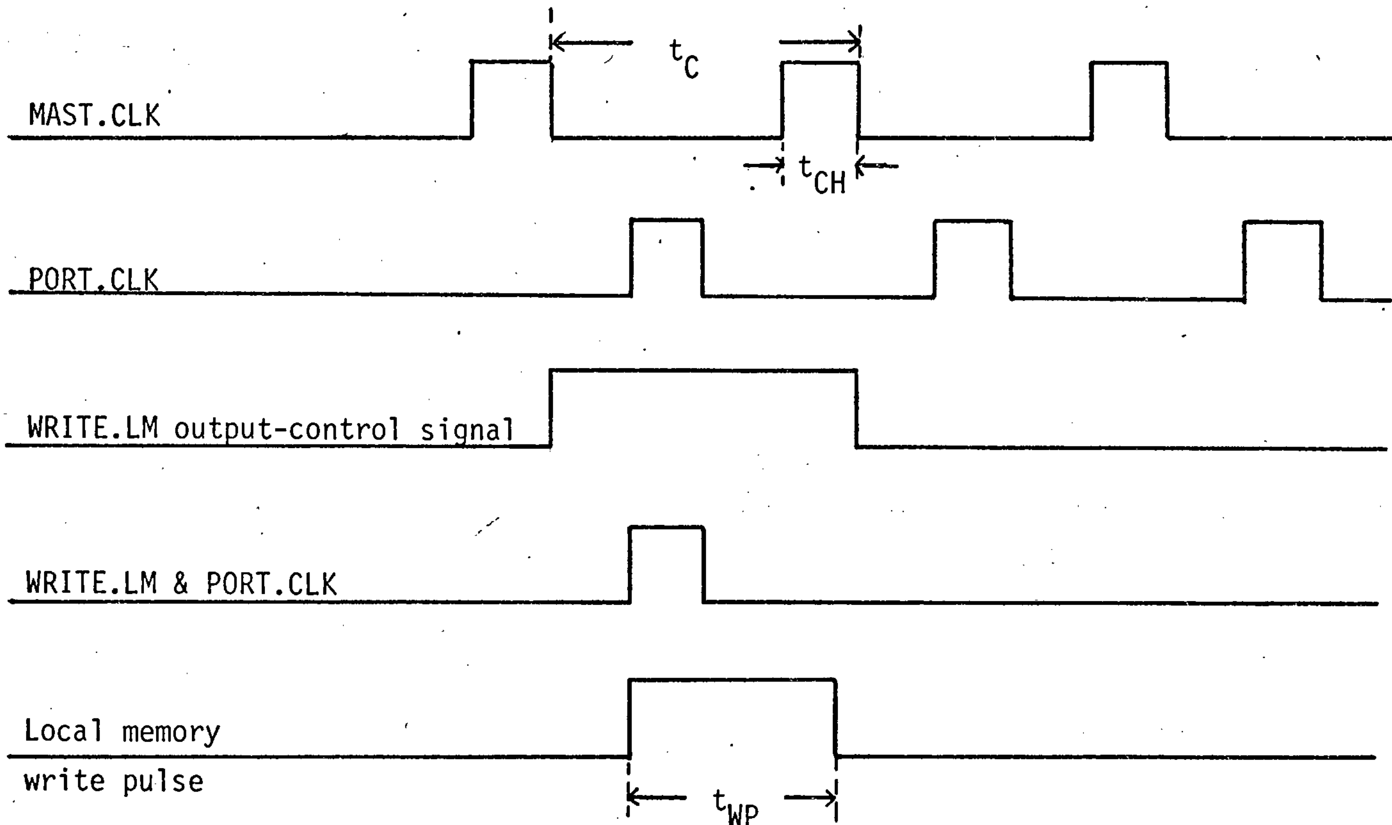
Figure C.5 Glitch Causing Unwanted Output-Control Signal

The SYNC.EN output-control signal will cause EN.C to go HIGH. EN.C logically AND'ed with bit 0 of the microinstruction is the READ.DATA signal. This unwanted signal (glitch) would be no more than 10 nanoseconds. This is sufficient, however, to set or clear most of the latches in the system. To prevent this situation occurring, the output-control signals are clocked using a clock, called PORT.CLK, which is out of phase with the systems master clock (MAST.CLK). Only those output-control signals which set or clear latches are clocked.

Data gated onto the M-bus is only gated into the CPA on the rising edge of a MAST.CLK pulse, and hence those output-control signals which gate data onto the M-bus (i.e. VAR.DIN, DISC.DIN, READ.LM and SECTOR.NO) must not be gated with PORT.CLK.

The output-control signal WRITE.LM presents a special problem. The WRITE.LM signal must have a duration of at least 180 nanoseconds. Since WRITE.LM AND'ed with PORT.CLK will only have a duration of 70 nanoseconds, a different method must be used. The WRITE.LM signal must be protected, as a glitch may cause the corruption of data in memory. The solution is as follows. The signal WRITE.LM is logically AND'ed with PORT.CLK to provide the required

protection. The protected WRITE.LM signal then triggers a 200 nanosecond monostable to provide a write pulse of adequate duration. The timing diagram in Figure C.6 illustrates this situation.



Clock Period (t_C) = 280 nanoseconds
Clock HIGH (t_{CH}) = 70 nanoseconds
t_{WP} = 200 nanoseconds

Figure C.6 Local Memory Write Timing

C.7 Local Memory Timing

The local memory read and write propogation delay times are analysed separately below.

C.7.1 Writing to Local Memory

An analysis of local memory write timing is necessary for the following reasons :

(i) When writing to a memory element, the address of the location to be written to must be set up before the write signal can be applied to that element. If the write pulse occurs before the appropriate address is stable, corruption of a memory location may occur. Consequently, it must be investigated whether any program sequence may inadvertently cause the corruption of a memory location.

(ii) When writing to a memory element, the memory device used will require that the data, address and write signal all be present for a certain amount of time. If this situation does not occur, the required data may not be written to the memory location. It is therefore necessary to ensure that the program sequence for writing data to local memory will in fact write this data.

When writing to local memory, the memory address register (MAR) must be set up with the memory address, and the accumulator (AC) must be set up with the data to be written to this address. Data is then written to memory by the WRITE.LM output-control signal. The WRITE.LM signal is logically AND'ed with PORT.CLK, the resulting signal triggers a 200 nanosecond monostable, as illustrated in Figure C.6.

C.7.1.1 Corruption of Data

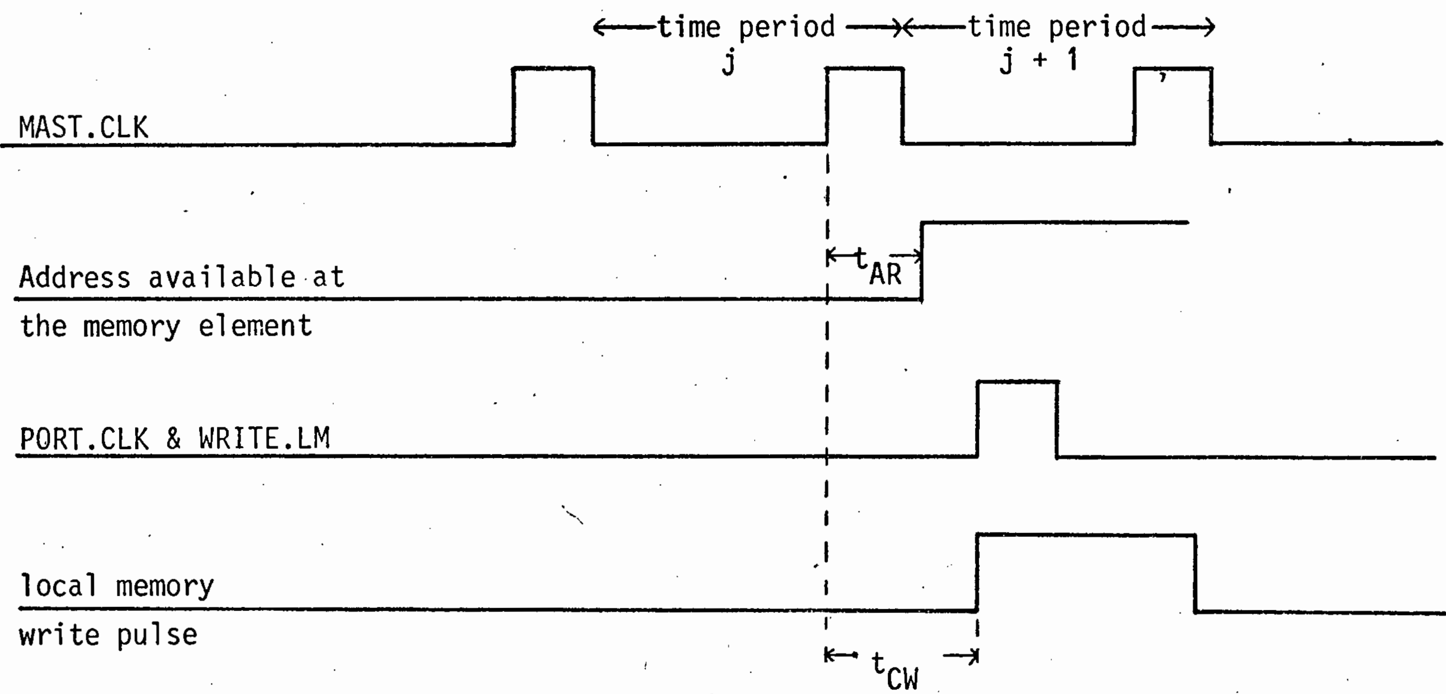
It must be ascertained how many microinstructions the MAR must be set up before the memory write signal, WRITE.LM is given. WRITE.LM cannot occur during the same microinstruction which sets up the MAR (since the write pulse will be triggered by PORT.CLK before the address is gated into the MAR by MAST.CLK).

The first case to be considered is therefore as follows. The MAR is set up by the current microinstruction, and the following microinstruction supplies the write signal (WRITE.LM) from an output port. An example of such a program sequence would be:

```
j       :  K11(R0)   *   60         # 60 to MAR
j + 1   :  *         *   WRITE.LM   # Write data in AC to LM address 60
```

To see whether or not this program sequence will cause corruption, an analysis of the delays in the system affecting the write operation must be made. This analysis must find whether or not the address will be set up in the memory address register before the write operation begins. Physically, the address will only be gated into the CPA's memory address register on the rising edge of the clock during time period j. The timing diagram in Figure C.7 illustrates this situation. The write signal, on the other hand, will be triggered on the rising edge of PORT.CLK. The rising edge of PORT.CLK occurring 140 nanoseconds after the rising edge of MAST.CLK.

The above program sequence will therefore only be valid if t_{AR} (the memory address propogation delay from the rising edge of the clock to the address being stable at the address inputs of the memory element) is less than 140 nanoseconds.



Clock period = 280 nanoseconds
Clock HIGH = 70 nanoseconds
$t_{AR} = t_{DL} + t_{74368} + t_{74124} + t_{AW}$

t_{ARMAX} = 100 nanoseconds
t_{ARTYP} = 72 nanoseconds
t_{CW} = 140 nanoseconds

Figure C.7 Local Memory Write Timing - Setting up the Local Memory Address

The memory address, on its way from the CPA to the memory element, will be subject to certain delays. These delays are due to the components in its path (i.e. a 74368 and a 74124); the propogation delay associated with the component the address has originated from (i.e. the 3002 CPE); and the component the address terminates in (i.e. the 2102A-2 memory element). The path the address takes is illustrated in Figure C.8. Consequently,

$t_{AR} = t_{DL} + t_{74368} + t_{74124} + t_{AW}$

t_{DL} and t_{AW} are defined in Table C.4, and the propogation delays for t_{DL}, t_{74368}, t_{74124} and t_{AW} are given in Table C.4. Therefore, t_{ARMAX} (the maximum propogation delay time for t_{AR}) =

50 + 18 + 12 + 20 = 100 nanoseconds, and

t_{ARTYP} (the typical propogation delay time for t_{AR}) =

32 + 12 + 8 + 20 = 72 nanoseconds.

Both t_{ARTYP} and t_{ARMAX} are less than 140 nanoseconds. The write pulse therefore only becomes active after the address inputs to the 2102A-2 are stable, and the above program sequence will not cause the corruption of data in memory.

C.7.1.2 Checking if Data Written Correctly

To ascertain whether or not the data will be written correctly, two things must be determined.

(i) It must be determined at what stage the address and data will be at when a write pulse is given. The above analysis has shown that the microinstruction must be set up in the MAR at least one microinstruction before the WRITE.LM output-control signal is given. This ensures that the address will always be available at the memory element before the write begins.

It must be determined if the data to be written to the memory element will be stable at the memory element before the write begins. The WRITE.LM signal cannot occur in the same microinstruction that sets up the accumulator. It will be shown that if the

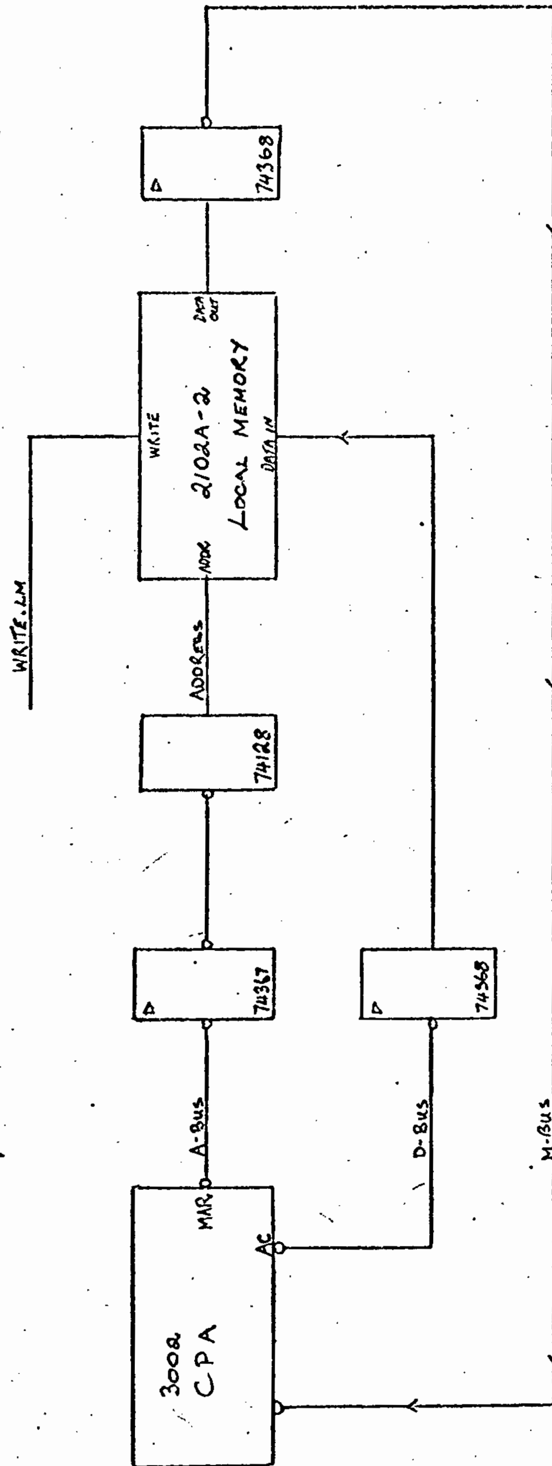


Figure C.8 Local Memory Circuitry Indicating Read/Write Propogation Delay Paths.

data to be written to local memory is set up in the current microinstruction (that is, by modifying the accumulator), and the write command is given in the following microinstruction, then the data will be valid at the memory element before the write begins. An example of such a program sequence is :

```
j      :   ILR(RO)
j + 1  :   *        *   WRITE.LM
```

Let t_{DR} be the propogation delay from the rising edge of the clock to the data being stable at the data inputs of the memory element (refer Figure C.9).

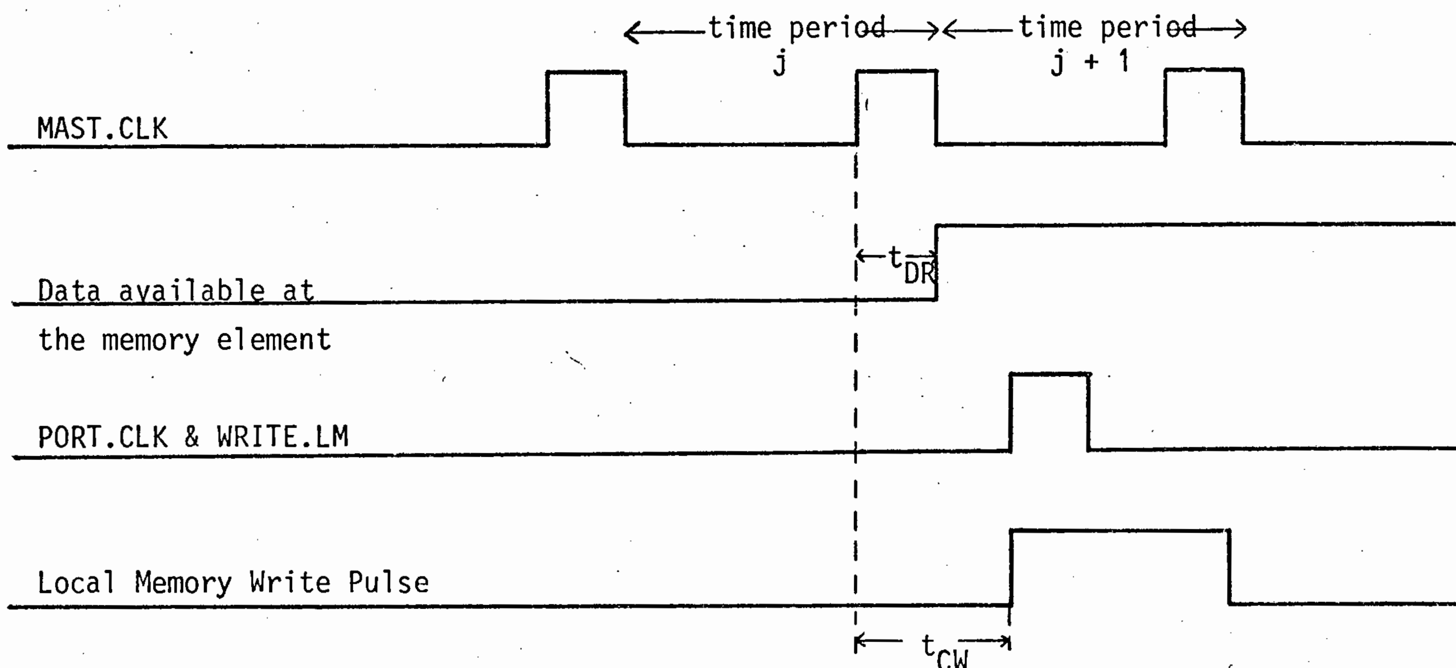
Referring to Figure C.8, it is found that

$$t_{DR} = t_{DL} + t_{74368}$$

t_{DL} is defined in Table C.4, and the propogation delays for t_{DL} and t_{74368} is given in Table C.4. Therefore,

t_{DRMAX} (the maximum propogation delay for t_{DR}) = 50 + 22 = 72 nanoseconds, and

t_{DRTYP} (the typical propogation delay for t_{DR}) = 32 + 10 = 42 nanoseconds.



Clock HIGH = 70 nanoseconds

$t_{DR} = t_{DL} + t_{74368}$

t_{DRMAX} = 72 nanoseconds

t_{DRTYP} = 42 nanoseconds

t_{CW} = 140 nanoseconds

Figure C.9 Local Memory Write Timing - Setting up the Local Memory Data

Since t_{DRMAX} is less than 140 nanoseconds, the data will be ready before the write begins. The timing diagram in Figure C.9 illustrates this situation.

In summary, both data and address will always be ready before a write begins.

(ii) It must now be seen if the write pulse given by only one

* * WRITE.LM

microinstruction will be long enough to cause the data to be correctly written. This is a function of the memory element's access time. Since the 2102A-2 memory elements are used, the write pulse must be available for at least 180 nanoseconds after both address and data are stable[5]. Since the write pulse will be available for 200 nanoseconds after both address and data are available, data will be written correctly.

Note that no modification should occur to either the accumulator or memory address register during a WRITE.LM microinstruction. This will allow the write pulse sufficient time to settle. The memory address register or accumulator can safely be modified on the following microinstruction.

C.7.2 Reading from local memory

When reading data from local memory, it must be ensured that the correct data is gated into the CPA. Again, this is done by analysing the system's timing for read operations. In a read operation, the address of the data be read is set up in the MAR, and the data is gated from the memory element onto the M-bus, and then into the CPA. The path the address has to travel from the CPA to the memory element, and data has to travel from that element to the CPA is illustrated in Figure C.8. The output-control signal READ.LM is used to gate data from the memory element onto the M-bus. During the same microinstruction, data will be gated into the CPA by the appropriate CPA microfunction. The output-control signal READ.LM cannot

[5] Intel Data Book, Intel Corporation, Santa Clara, California (1976), pp. 46-49.

be performed by the same microinstruction that sets up the MAR. The program sequence which must therefore be investigated is as follows. The MAR is set up during the current microinstruction and the data is gated into the CPA during the next microinstruction. An example of such a program sequence would be :

```
j     : K11(R0)   *   60       # 60 to MAR
j + 1 : ACM(A)    *   READ.LM  # Contents of address 60 to AC.
```

Let t_D be the propogation delay from the rising edge of the clock to the time data is ready for use at the CPA. t_D can be broken up into three propogation delays (refer Figure C.8) :

(i) The propogation delay from the rising edge of the clock to the address being stable at the address inputs of the memory element. This is the definition of t_{AR} (refer Figure C.7), and it was found that t_{ARMAX} = 100 nanoseconds and t_{ARTYP} = 72 nanoseconds.

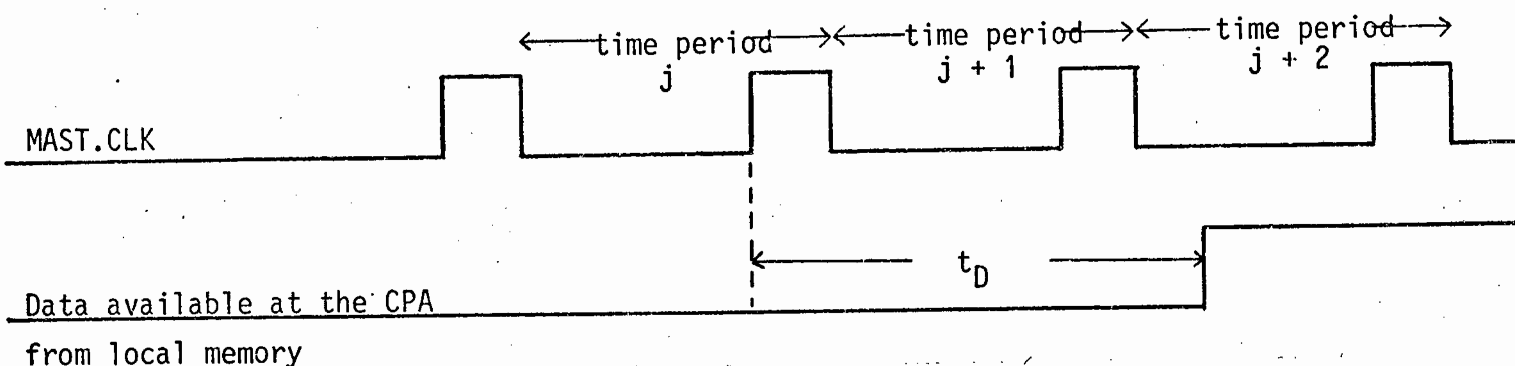
(ii) The propogation delay from the address inputs of the memory element to the data being stable at the output of that element. This is the memory elements propogation delay. The memory element is the 2102A-2, and it has a 250 nanosecond access time.

(iii) The propogation delay from the data being stable at the outputs of the memory element until it is available for use by the CPA. Referring to Figure C.8, it is seen that this is equal to $t_{74368} + t_{DS}$. t_{DS} is defined in Table C.4.

Consequently $t_D = t_{AR} + t_{2102A-2} + t_{74368} + t_{DS}$. Therefore t_{DMAX} (the maximum propogation delay for t_D) =

100 + 250 + 17 + 50 = 417 nanoseconds, and t_{DTYP} (the typical propogation delay for t_D) =
72 + 250 + 11,5 + 30 = 363,5 nanoseconds.

Data will only be ready at the inputs of the CPA during time period j + 2. The timing diagram in Figure C.10 illustrates this situation.



$t_D = t_{AR} + t_{2102A-2} + t_{74368} + t_{DS}$
t_{DMAX} = 417 nanoseconds
t_{DTYP} = 363,5 nanoseconds

Figure C.10 Local Memory Read Timing - Setting up to the Local Memory Data

The microinstruction executed during time period j + 1 is

ACM(A) * READ.LM

Data will therefore be gated into the CPA on the rising edge of the clock during this time period. The maximum time which is therefore allowed for data to become stable is one clock period which is 280 nanoseconds. Since t_{DTYP} is greater than 280 nanoseconds, this program sequence will not work.

It must be determined how many microinstructions after the address has been set up in the MAR, the data can be gated into the CPA. t_{DMAX} is 417 nanoseconds. This is less than twice the clock period which is 560 nanoseconds. Consequently, the microinstruction which gates data into the CPA must occur two or more microinstructions after the address has been set up in the MAR. The minimum program sequence would therefore be :

```
j     : K11(R0)    *    60
j + 1 : *
j + 2 : ACM(A)     *    READ.LM
```

C.7.3 Summary

The results of the above analysis are summarized below :

(i) When writing to local memory, the write pulse may only

occur one or more microinstructions after the memory address has been set up in the memory address register (MAR). The minimum program sequence being of the form :

```
K11(RO)     *     60        #  Modify MAR.
*           *     WRITE.LM  #  Write to Local memory.
```

(ii) When writing to local memory, the write pulse may only occur one or more microinstructions after the data has been set up in the accumulator. The minimum program sequence being of the form :

```
ILR(RO)                     #  Modify the accumulator.
*           *     WRITE.LM  #  Write to local memory.
```

(iii) When reading from local memory, the microinstruction which gates data into the CPA may only occur two microinstructions after the address has been set up in the MAR. The minimum program sequence being of the form :

```
K11(RO)     *     60        #  Modify the accumulator.
*                           #  Wait 1 microinstruction.
ACM(A)      *     READ.LM   #  Gate data into the CPA.
```

Propogation delay times (nanoseconds)	TYPICAL	MAXIMUM
74S00	3	5
74S02	3,5	5,5
74S04	3	5
74S08	5	7,5
74LS367	10	22
74LS368	12	18
74S86	7	10,5
74128	6	9
74150	23	35
74S175	11,5	17
2909		40
2102A-1		200
2102A-2		250
t_{DL}	32	50
t_{DS}	30	50
t_{AW}		20

Definition of Terms

t_{DL} = Progation delay in the CPA from the rising edge of the clock to the A-bus and D-bus outputs. This propogation delay is specified by the Intel 3002 CPE[6].

t_{AW} = The address to write set up time. That is, the write signal must only be present at the local memory t_{AW} nanoseconds after the address is stable. This propogation delay is specified by the Intel 2102A-2 memory[7].

t_{DS} = The data set-up time required from the inputs of the CPA to the rising edge of the clock. This propogation delay is specified by the Intel 3002 CPE[8].

Table C.4 Propogation Delay Times

[6] 3002 Central Processing Element, pp. 9-10.

[7] Intel Data Book, pp. 46-49.

[8] 3002 Central Processing Element, pp. 9-10.

APPENDIX D

INPUT-CONTROL AND OUTPUT-CONTROL SIGNALS

The CPU communicates with the external logic by means of input-control and output-control signals. Input-control signals are input to the 32 to 1 multiplexer, and output-control signals are the outputs from the three output ports.

Input-control signals are received from :

(i) the central processor array;
(ii) the disc drive interface;
(iii) the host computer (Varian) interface; and
(iv) the console.

Output-control signals are transmitted to :

(i) local memory;
(ii) the disc drive interface; and
(iii) the host computer interface.

When a conditional jump is performed, it is the input-control signals which determine the next address to be selected. Output-control signals perform the actual control of the external logic.

It is desirable to discuss the operation of the HLDC in terms of these input-control and output-control signals. Such a discussion is essential for the microprogrammer, as the microprogrammer will need to know how each input and output signal is used, and how it will affect the system. The input and output signals are dependant on each other, and consequently these signals must be discussed together. These signals will therefore be discussed in terms of the following five elements in the system :

(i) the central processor array;
(ii) the disc drive interface;
(iii) the host computer interface;
(iv) local memory; and
(v) the console.

D.1 The Central Processor Array Control Signals

There are two signals associated with this central processor array. These are the two input-control signals CO and SIGNBIT. Signals generated by the central processor array must be handled in a special fashion because the HLDC has a pipelined architecture. That is, whilst the current microinstruction is being executed, the next microinstruction is being fetched. Consequently, an input-control signal generated by the central processor array during the current microinstruction, cannot be used in the fetching of the next microinstruction. This situation is illustrated in Example D.1. In this example, the information in the CPA field of the microinstruction, TZR(R3), sets up the input-control signal CO, which is only tested in the following microinstruction:

```
    *    JMPT    CO    LOOP
```

It is only when a central processor array input-control signal is being tested that this consideration is necessary. All other input-control signals in the system being asyncronous.

CO

CO (carry-out) is used when testing whether or not a particular register is zero. For example, assuming a loop has to be executed 100 times. A register (say R3) would be used as a loop-counter. At the end of every cycle of the loop, R3 would have to be decremented and tested to see if it is equal to zero. A possible microprogram sequence to perform this is illustrated in Example D.1.

```
    CLR(R3)
    K11(R3)     *           99      # Set loop-counter equal to 99
LOOP .
    :
    DSM(R3)                         # Decrement loop-counter
    TZR(R3)                         # and test if zero
    *         JMPT   CO     LOOP    # if non-zero, loop
```

Example D.1

The central processor array command TZR(R3) will perform the logical OR of all the bits in R3. The result of this operation being indicated by the most significant carry-out of the CPA. This carry-out signal is the CO input-control signal. Consequently, CO will be FALSE (LOW) only if R3 is zero.

SIGNBIT

The most significant bit of the accumulator (bit 15) is input to the multiplexer, and is the input-control signal SIGNBIT. This signal is useful for testing if a number is positive or negative. Consider the following microinstruction :

```
     *    JMPT    SIGNBIT    NEGATIVE
```

If the most significant bit of the accumulator is set, SIGNBIT will be TRUE, and a jump is made to location NEGATIVE. If SIGNBIT is FALSE, the next microinstruction is fetched for execution.

D.2 The Console Control Signals

There is only one signal associated with the console. This is the input-control signal FORMAT. The FORMAT input-control signal is TRUE when the consoles format switch is ON, and the FORMAT signal is FALSE when the format switch is OFF. The FORMAT input-control signal has two uses :

(i) The FORMAT signal can be used as a data protection facility for the disc. For example, it can be used to avoid the corruption of a disc pack through inadvertant formatting. To prevent this situation from occuring, it can be ensured that formatting will only begin if the format switch is ON.

(ii) The FORMAT signal can be used as a breakpoint in a microprogram. This facility is useful when debugging a particular microprogram. Breakpoints can be set up throughout the program, and a breakpoint would occur by using the format switch with the following microinstructions :

```
A   :   *   INCT   FORMAT
            :
            :
B   :   *   INCF   FORMAT
```

When the format switch is OFF, the program will proceed as far as microinstruction A. By switching the format switch ON, the program will proceed to microinstruction B.

D.3 Local Memory Control Signals

There are two signals associated with local memory. These are the output-control signals READ.LM and WRITE.LM. There are certain timing problems associated with reading from and writing to local memory. These timing problems are discussed in Appendix C.7.

READ.LM

The READ.LM output-control signal will cause data from local memory to be gated onto the M-bus. The appropriate local memory address being supplied by the CPA's memory address register. As mentioned in Appendix C.7, the microinstruction which gates data into the CPA can only occur two or more microinstructions after the address has been set up in the memory address register. An example of a program sequence to read data from local memory address 60 into the accumulator (AC) is as follows :

```
K11(RO)    *    60         # 60 to MAR
*
ACM(A)     *    READ.LM    # contents of address 60 to AC
```

WRITE.LM

The WRITE.LM output-control signal will write data to local memory. The data to be written to local memory is set up in the accumulator, and the local memory address is set up in the memory address register. The following example illustrates how the contents of the accumulator can be written to local memory address 60 :

```
K11(RO)    *    60          # 60 to MAR
*          *    WRITE.LM    # AC to local memory address 60
```

D.4 HLDC/Disc Drive Interface Control Signals

The disc drive interfaced to the HLDC is the CalComp CD1 disc drive. Communication between the HLDC and the disc drive takes place by means of the following input-control and output-control signals.

Input-control Signals: SYNC, EQ.16, INDEX, SECTOR, ONLINE, UNSAFE, WRITE.CUR, SEEKRDY, SEEKERR, SGLDEN, UNTSEL, ATTEN.

Output-control Signals: DRIVE1, DRIVE2, DRIVE4, DRIVE.CLR, DISC.DIN, DISC.DOUT, SECTOR.NO, READ.DATA, READ.CLK, CLR.EQ16, WRITE, SYNC.EN, SYNC.CLR, BUS.EN, TAG.EN.

There is also a group of constants which is set up by the assembler and which is used in the interface.

Constants: BUS0, BUS1, ... BUS7, CONTROL, CYL, HEAD and DIFF.

The following discussion should be read in conjunction with Figure D.1 which illustrates, in block diagram form, that part of the disc drive interface which interfaces to the central processing unit.

Drive Select Signals

The four output-control signals DRIVE1, DRIVE2, DRIVE4 and DRIVE.CLR are used for selecting 1 of 8 possible disc drives which could be connected to the HLDC. The HLDC currently has connections for only two disc drives. These two drives are called drive 0 and drive 1. Associated with the output-control signals DRIVE1, DRIVE2, and DRIVE4 are three latches (refer Figure D.1). These three latches are cleared when a DRIVE.CLR output-control signal is given. Since all three latches are cleared by a DRIVE.CLR signal, this signal selects drive 0. To select a drive other than drive 0, a DRIVE.CLR signal must first be given, followed by the appropriate combination of the output-control signals DRIVE1, DRIVE2 and DRIVE4. That is, drives 1 to 7 are selected by using one of the following microinstructions :

Figure D.1 Block Diagram of the Disc Drive Interface.

```
*    *    DRIVE1                    # Select drive 1
*    *    DRIVE2                    # Select drive 2
*    *    DRIVE2, DRIVE1            # Select drive 3
*    *    DRIVE4                    # Select drive 4
*    *    DRIVE4, DRIVE1            # Select drive 5
*    *    DRIVE4, DRIVE2            # Select drive 6
*    *    DRIVE4, DRIVE2, DRIVE1    # Select drive 7
```

BUS.EN and TAG.EN Output-control Signals

BUS.EN and TAG.EN are used in outputting bus signals and tag signals to the disc drive. Associated with the bus and tag signals are two sets of latches called the bus latches and tag latches respectively (refer Figure D.1). There are 8 bus latches and they are called BUS0, BUS1, ... BUS7. When a bus latch is enabled, the corresponding bus signal to the disc drive goes HIGH. Similarly, there are 4 tag latches, and they are called CONTROL,CYL, HEAD and DIFF. The bus latches are enabled or disabled by the contents of the accumulator logically AND'ed with the BUS.EN output-control signal. Similarly, the four tag latches are enabled/ disabled by the contents of the accumulator logically AND'ed with the TAG.EN output-control signal. The accumulator is set up to the desired value by using the special constants which have the same names as their corresponding latches.

The following program sequence illustrates how the latches BUS1, BUS4 and CONTROL are first enabled and then disabled.

```
# Enable BUS1, BUS4 and CONTROL
  CLR(A)
  K11(A)     *     BUS1, BUS4, CONTROL
  *          *     BUS.EN, TAG.EN
# Disable BUS4
  CLR(A)
  K11(A)     *     BUS4
  *          *     BUS.EN
# Disable BUS1 and CONTROL
  CLR(A)
  K11(A)     *     BUS1, CONTROL
  *          *     BUS.EN, TAG.EN
```

UNTSEL

When the HLDC selects a disc drive, the UNTSEL input-control signal for that drive will go HIGH. This input-control signal is useful when more than one drive is being used. When the HLDC selects a different drive from the currently selected drive, the HLDC must check if the UNTSEL signal is HIGH.

ATTEN

The input-control signal ATTEN, like UNTSEL, is useful when more than one disc drive is attached to the HLDC. When the selected drive requires the attention of the HLDC, the ATTEN line will go HIGH. For example, the ATTEN line will go HIGH after the completion of a disc seek-operation (whether successful or unsuccessful).

ONLINE

When a disc drive goes online, the ONLINE input-control signal will go HIGH. The appropriate drive is selected by the drive select signals mentioned above, and only one drive will be online at any one time. The online signal will go LOW if an UNSAFE input-control signal is received from the disc drive.

UNSAFE

When an unsafe condition occurs, the UNSAFE input-control signal will go HIGH, and the select lock indicator on the disc drive will light up. The drive power must then first be turned off and then on again. The UNSAFE signal should be examined at those places where a disc drive unsafe condition may occur.

WRITE.CUR

The WRITE.CUR input-control signal will go HIGH when writing, and will go LOW again when writing ceases. There is a delay between instructing the drive to stop writing and the falling edge of WRITE.CUR. This delay makes

it necessary for the HLDC to be able to examine the WRITE.CUR signal.

SEEKRDY and SEEKERR

When a seek is in progress, the signal SEEKRDY will go LOW. When the seek is completed, SEEKRDY will go HIGH. If an error is encountered during a seek, the signal SEEKERR will go HIGH.

INDEX, SECTOR AND SECTOR.NO

The CalComp CD1 disc drive sends out 20 sector pulses per track. The HLDC requires only 10 sectors per track, and hence hardware is used to select every second sector pulse.

The timing of the input-control signals INDEX and SECTOR is illustrated in Figure D.2[1].

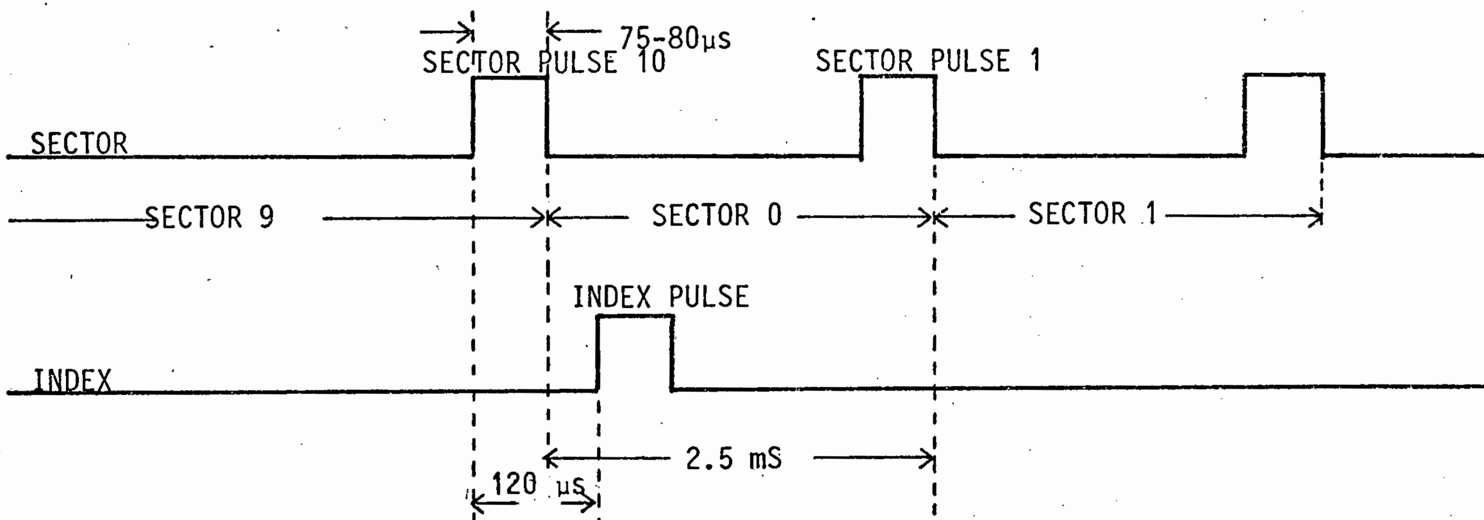


Figure D.2

[1]CalComp Field Engineering Service Handbook, CD1 Disc Drive, California Computer Products, Inc., Anaheim, California (1971), p. 23.

The HLDC should wait for the falling edge of the sector pulse before beginning to write or read. This is done by using two microinstructions as follows :

```
*    INCT    SECTOR
*    INCF    SECTOR
```

This will ensure that read and write operations begin at the same place each time. There are a number of methods which could be used in accessing the desired sector :

(i) The HLDC can wait for the index pulse, and can then count the sector pulses until the desired sector is under the read/write heads. The problem with this method is that an unnecessary disc revolution may be required. For example, if the read/write heads are currently at sector 2, and sector 6 is required, the HLDC, not knowing the current sector position, would first wait for the index pulse, and then count off 6 sector pulses.

(ii) One method of overcoming this problem is to perform a software count on the sectors. This method is used by Intels microprogrammed 'low-level' disc controller[2]. The problem in using this method in the HLDC is that it would be necessary to continuously check if a sector pulse has been received.

(iii) The method used is to have a decade counter. This counter is cleared by the index pulses. The counter increments on each falling edge of the sector pulse. When the tenth sector pulse is received, the counter, being a decade counter, resets itself to zero. Figure D.2 illustrates the sector numbers associated with the counter (sectors being numbered 0 to 9).

The value of the decade counter is inspected by means of the SECTOR.NO output-control signal. This signal gates the value of the counter onto the M-bus. The following example illustrates how the decade counter could be used.

[2] Intel Data Book, Intel Corporation, Santa Clara, California (1976), pp. 750-758.

Assume that it is necessary to access a particular sector, say the sector whose sector number stored in register R0. This could be done by the following program sequence :

```
# Syncronize with the beginning of the sector.
AGAIN     *          INCT       SECTOR
          *          INCF       SECTOR
# Check if the wanted sector minus the current sector equals zero.
          ILR(R0)
          ACM(A)     *                    SECTOR.NO
          TZR(A)
# If zero, then the required sector.
          *          JMPT       CO        O.K.
# If non-zero, then try the next sector.
          *          JMP                  AGAIN
O.K. # the required sector has been found.
```

It can be rightly argued that a possible method for the HLDC to obtain the current sector address, would be to read the current sectors header. The main use of the header is to improve the reliability of data on disc by ensuring that the incorrect sector is not accessed. This was discussed in Appendix A.3. Using the header to obtain the sector address would not, however, reduce the discs reliability to any appreciable extent. This is because, firstly, the reading of the header is a reasonably reliable operation (compared with the disc seek to obtain the header in the first place), and, secondly, the header is followed by a header cyclic redundancy check word.

The advantage of using the decade counter over this latter method is that the sector address can be obtained at any stage. If the latter method is used, the sector address can only be obtained at the beginning of a sector. In the HLDC, it may be desirable to know the current sector address immediately, so that it can be ascertained what operation to perform next.

SGLDEN

The rate at which data is transferred to/from the disc drive is dependent

on the discs rotational speed and recording density. The disc drive interfaced to the system is the CalComp CD1 disc drive. A bit of data can be written to this drive every 400 nanoseconds[3]. The clock used for writing data to disc or reading data from disc comes from one of two different sources :

(i) A fixed clock when writing. The system uses a 5 megahertz clock which is divided down to get the correct frequency for the disc drives. A bit must be written to the CalComp CD1 disc drive every 400 nanoseconds, and hence the write clock operates at a frequency of 2,5 megahertz when writing to this drive.

(ii) A variable frequency oscillator (VFO) when reading. The VFO is continuously updated by the clock bits from disc, and is then used to predict the next clock or data bit from disc.

A disc drive can be set up as single density or double density. Bit transfer speeds for single density drives will be at 400 nanoseconds/bit (800 nanoseconds for clock bit plus data bit). Bit transfer speeds for double density drives are at 200 nanoseconds/bit (400 nanoseconds for data plus clock). Drives are set up as single density or double density by connecting the appropriate jumpers on Board 6 as illustrated by the circuit diagram of Board 6 in Appendix E.

By inspecting the SGLDEN input-control signal, the HLDC can ascertain whether the selected drive is single density or double density. If SGLDEN is TRUE, then the drive is single density. If SGLDEN is FALSE, the drive is double density. Currently only single density drives can be supported by the system.

WRITE

The WRITE output-control signal is used to enable/disable the WRITE latch. Initially this latch is cleared (disabled) by a system reset. The WRITE latch is then successively enabled or disabled by means of the WRITE output-control signal. The use of this latch will be discussed later.

D.4.1 Reading and Writing Data

The following discussion on reading and writing data should be read in conjunction with Figure D.1. Data transfers between the CPA and the disc drive interface are in parallel form. Data is transferred from the drive interface to the CPA on the M-bus when reading, and from the CPA to the drive interface on the D-bus when writing. Transfers between the disc drive interface and the disc drive are in serial form. The 16 bit shift register (Figure D.1) is used to convert data from serial-to-parallel form, and vice versa. This shift register is clocked by the data-clock. The data-clock being a variable frequency oscillator (VFO) when reading and fixed 5 megahertz oscillator when writing. Data syncronization is achieved by a modulo-16 (MOD16) counter (i.e. the counter counts from 0 to 15 and back to 0 again). During write operations (the WRITE latch enabled), when the modulo-16 counter reaches 15, the 16 bit shift register will automatically be loaded from the accumulator. During read operations (the WRITE latch disabled), the 16 bit DISC.DIN latch will automatically be loaded from the shift-register when the modulo-16 counter reaches 15. The modulo-16 counter also enables the EQ.16 latch when it equals 15 (for both read and write operations). The output of this latch is the input-control signal EQ.16. The HLDC uses this signal during both read and write operations to syncronize itself with the disc data.

Reading

When reading, the HLDC will monitor the EQ.16 input-control signal. When this signal goes TRUE, the HLDC will gate the data from the DISC.DIN latch into the CPA by using the output-control signal DISC.DIN. This signal gates the data from the DISC.DIN latch onto the M-bus. The HLDC must then CLR.EQ16 latch with the CLR.EQ16 output-control signal. The following program sequence will read data off disc, and store the data in consecutive locations in local memory. The number of words to be read off disc is initially set up in the T register, and the start address of local memory is initially set up in register R0.

```
AGAIN    *         INCT     EQ.16
         ACM(A)    *                  DISC.DIN
         SDR(T)    *                  WRITE.LM
         TZR(T)    *                  CLR.EQ16
         ILR(R0)   JMPT     C0        AGAIN
```

Writing

When writing, the HLDC will again monitor the EQ.16 input-control signal. When EQ.16 goes TRUE, the HLDC will store the next data word to be written to disc in the accumulator, and will clear the EQ.16 latch with the CLR.EQ16 output-control signal.

D.4.2 Disc Format

A disc format, based on the Telefile low-level disc controllers disc format[3], is illustrated in Figure D.3. The different fields which make up the format for each sector will now be discussed. This discussion should be read in conjunction with Figures D.1 and D.3.

(i) Data Preamble

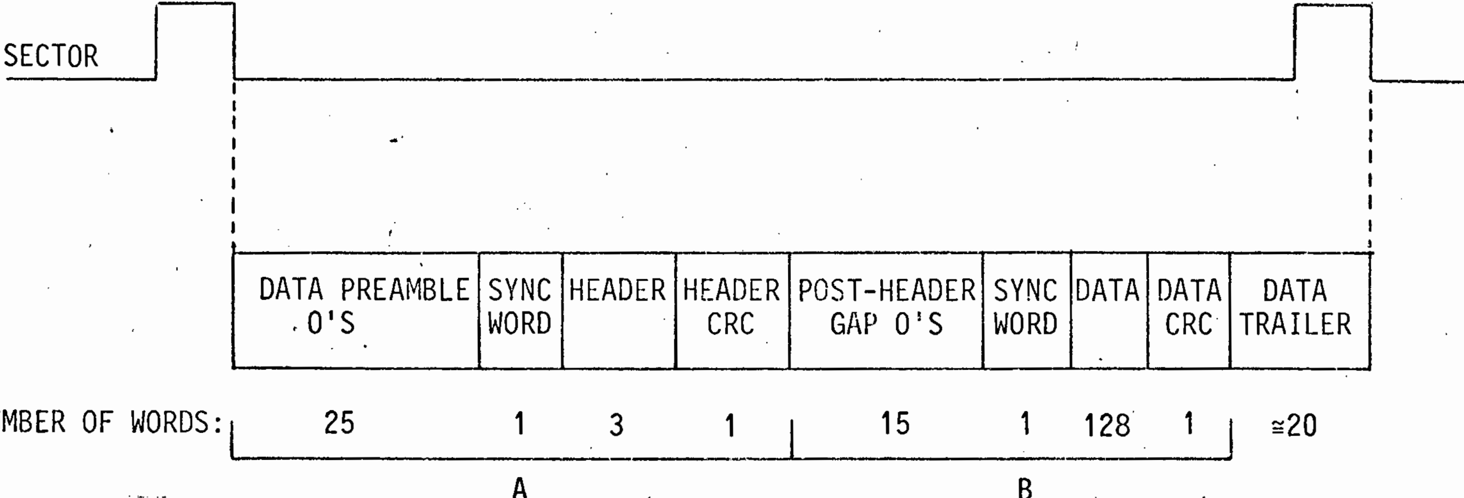
The data preamble consists of an all zeroes bit pattern, and should be between 20 and 25 words in length. The data preamble is used in conjunction with the two output-control signals READ.DATA and READ.CLK to select the variable frequency oscialltor (VFO) as the data-clock (DATA.CLK) during read operations. These two output-control signals are used to enable/disable the READ.DATA and READ.CLK latches respectively (refer Figure D.1). Initially these two latches are disabled by a system reset. The latches are then first enabled and then disabled by the successive application of the appropriate one of the two output-control signals. The enabled READ.DATA latch will select the appropriate phase of the VFO, and this latch should be enabled at about word 5 of the data preamble. The enabled READ.CLK latch will select the VFO as the data-clock, and this latch must be enabled no sooner than 12 words after the READ.DATA latch has been enabled. That is, at about word 17 of the data preamble.

If the write clock is to be selected as the data clock, both the READ.DATA and READ.CLK latches must be disabled.

(ii) Syncronizing Word

A syncronizing word is necessary to syncronize the modulo-16 counter with data from disc. Syncronization will only occur when the SYNC latch has

[3] Products Specification, DC-16 Disc Drive Controller, Telefile Computer Products, Irvine, California (June 4, 1971), p. 4.



Notes:

(i) The total number of words possible at the specified environmental conditions can be calculated as follows. One bit of information can be written to disc every 400 nanoseconds. Consequently, one bit of data plus one bit of clock is written to disc every 800 nanoseconds. As illustrated in Figure D.2, it takes 2,5 milliseconds for the read/write heads to pass over one sector.

Hence, at the correct supnly conditions :

$$\frac{2{,}5 \times 10^{-3} \text{ seconds/sector}}{800 \times 10^{-9} \text{ seconds/bit}} = 3125 \text{ bits/sector}$$

= 195 words/sector .... 16 bits/word.

(ii) The total number of words used is 175. This allows for a 10% leeway in disc speeds.

(iii) The syncronization word (SYNC word) is 2.

(iv) The HEADER contains the cylinder, track and sector addresses. The header can either be stored as three separate words or as one word.

(v) All data in part A of the sector will be written during the format routine, and this data will remain unchanged. The data in part B of the sector will be changed with each new write operation.

Figure D.3 Sector Format

been enabled by the output-control signal SYNC.EN. When the SYNC latch is enabled, a non-zero data bit will cause the modulo-16 counter to be cleared, and the SYNC.REC latch to be enabled. The output of the SYNC.REC latch is the input-control signal SYNC. The HLDC inspects the SYNC input-control signal during data syncronize operations, and when this signal goes TRUE, the HLDC will receive data when the EQ.16 input-control signal next goes TRUE. The HLDC must clear the SYNC and SYNC.REC latches with a SYNC.CLR output-control signal after the SYNC input-control goes TRUE.

(iii) Header and Header CRC

Before read or write operations take place, the HLDC inspects the header to see if the correct cylinder, track and sector have been chosen.

(iv) Post-header

When data is to be written to a formatted disc, the HLDC must first perform a read operation to inspect the header information. Having found that the correct sector has been selected, the write operation can commence. Changing from the VFO (read) clock to the fixed 5 megahertz (write) clock will take a certain amount of time, and will upset the modulo-16 counters timing. It is therefore necessary to write about 10 to 15 words of zeroes followed by a syncronizing word during write operations. When reading, the HLDC will start searching for the zero words between the fifth and tenth words after the header CRC. After locating the zero words, the HLDC will enable the SYNC latch, and the modulo-16 counters timing will subsequently be syncronized with the data.

(v) Data and Data CRC

With the above format, data should be no more than 140 words to allow for variations in disc speeds. The data is followed by a CRC which is calculated by the HLDC.

(vi) Post-data Gap

When writing, enough space must be allowed for the data and data CRC to

accommodate the maximum disc speeds possible. Consequently, there should always be a post-data gap after the data CRC word.

D.5 The HLDC/Varian Interface

As mentioned in Chapter 3.11, the HLDC/Varian interface allows three basic modes of I/O communication between the Varian and HLDC.

(i) The interface allows communication to occur by means of the full range of the Varians I/O instructions.
(ii) The interface allows data to be transferred between the Varian and HLDC via direct memory access (DMA).
(iii) The interface allows the HLDC to interrupt the Varian.

Associated with the above three modes of communication are the following input- and output-control signals.

Input-control Signals : EXC, DTOX, DTIX, COMMAND and DMA.FIN.
Output-control Signals : READY, EOF, ERR, VAR.DIN, VAR.DOUT, CLR.COMMAND, DMA.IN, DMA.OUT and INTERRUPT.

The above three modes of communication, together with their associated control signals, will now be discussed. This discussion should be read in conjunction with certain of the circuit diagrams and block diagrams illustrated in Appendix E. The diagrams which should be referenced are :

(i) Block diagram of the Computer Interface - This diagram is reproduced in this Appendix as Figure D.4.
(ii) DMA and Interrupt Control - Board 5 Page 2.
(iii) Varian Interface (Sense) - Board 5 Page 3.

D.5.1 I/O Instructions

As mentioned in Chapter 3.11 there are basically four types of I/O instructions which can be executed by the Varian :

(i) Program Sense
(ii) External Control

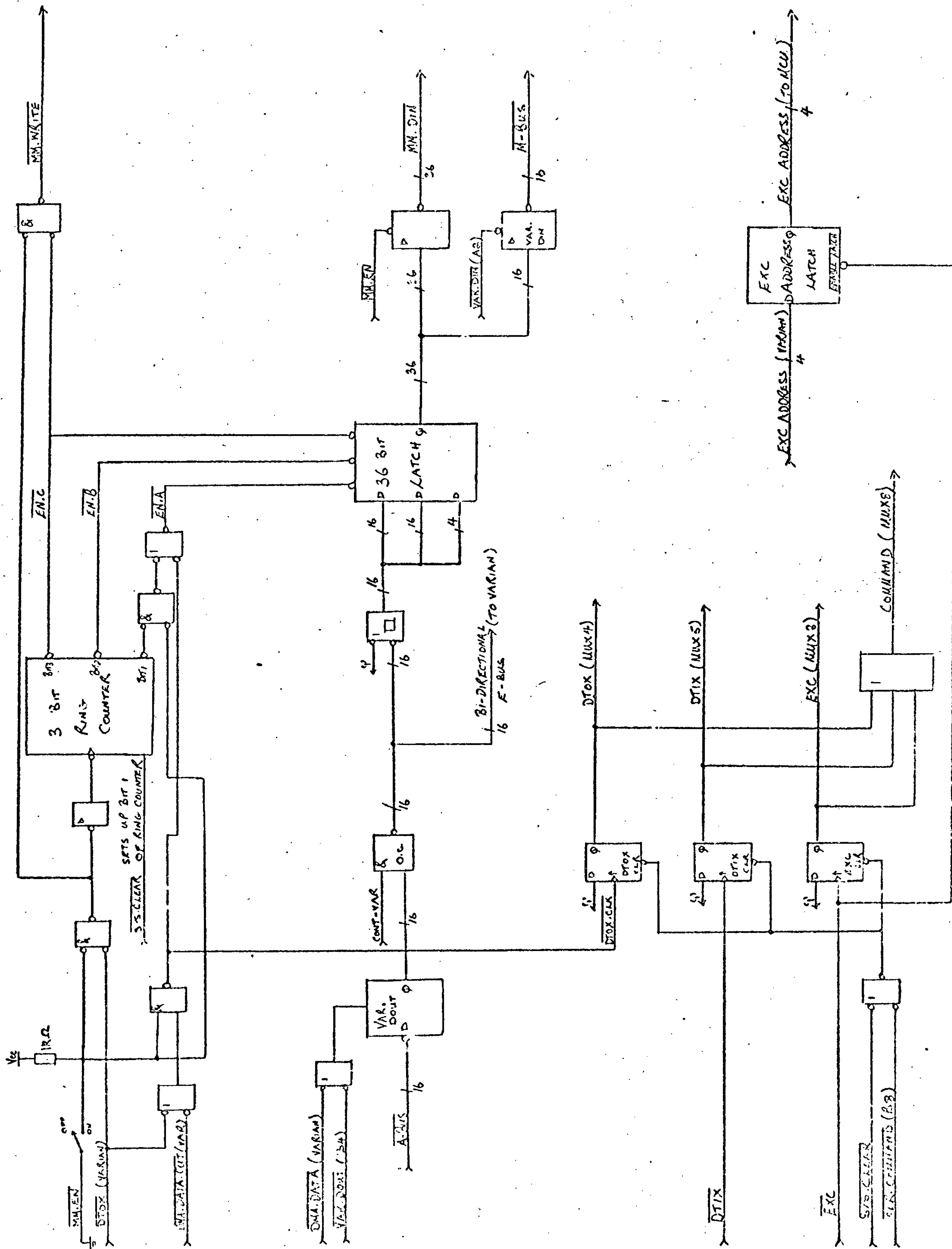


Figure D.4 Block Diagram of the Computer Interface.

(iii) Single-word output transfer
(iv) Single-word input transfer

Associated with these four types of I/O instructions are a number of input- and output-control signals.

Input-control Signals : EXC, DTOX, DTIX, and COMMAND.
Output-control Signals : READY, EOF, ERR, VAR.DIN, VAR.DOUT and CLR.COMMAND.

The above four types of I/O instructions and their associated control signals will now be discussed.

(i) Program Sense (READY, EOF and ERR)

The HLDC indicates its current status to the Varian by means of three Sense latches. These latches are called the READY, EOF and ERR latches, and are set up by the input-control signals READY, EOF and ERR respectively (refer Appendix E, Board 5 Page 3). The Varian can examine any one of these three latches by means of a Sense command.

SEN 0014 examines the READY latch,
SEN 0114 examines the EOF latch, and
SEN 0214 examines the ERR latch.

(ii) External Control (EXC)

When an external control instruction is executed by the Varian, a 4-bit address is gated from the E-bus into a latch called the EXC-address latch. The output of this latch being an input to the microprogram control unit (refer Figure D.4). The external control instruction will also set a latch called the EXC latch. The output of this latch being the input-control signal EXC. The HLDC will periodically inspect the EXC signal, and if the signal is TRUE, the HLDC will take the appropriate action. This will generally be a jump to a microprogram routine specified by the contents of the EXC-address latch. The jump would be made using the JMPD or JSRD jump commands (refer Appendix B).

(iii) <u>Single-word Output Transfer</u> (DTOX and VAR.DIN)

When a single-word output transfer instruction is executed by the Varian, data will be gated from the E-bus into a latch called the VAR.DIN latch. A latch, called the DTOX latch, will also be set. The output of the DTOX latch is the DTOX input-control signal. The HLDC will periodically inspect the DTOX signal to see if a data transfer has occurred. On finding the signal TRUE, the HLDC will gate data from the VAR.DIN latch onto the M-bus by means of the VAR.DIN output-control signal (refer Figure D.4).

(iv) <u>Single-word Input Transfer</u> (DTIX and VAR.DOUT)

Before a single-word input transfer instruction can be executed by the Varian, the appropriate word of data must first be set up in a latch called the VAR.DOUT latch. Data is gated from the CPA's memory address register into the VAR.DOUT latch by means of the VAR.DOUT output-control signal (refer Figure D.4).

<u>HLDC/Varian Communication</u>

The following discussion will show how the above input- and output-control signals are used to perform the required HLDC/Varian communication.

To check if the Varian has initiated an I/O command (i.e. external control, single-word output transfer and single-word input transfer), the HLDC must periodically examine the EXC, DTOX and DTIX input-control signals. Since the operation of examining whether or not a command has been initiated will be frequently performed, it is desirable to test this occurance by means of only one microinstruction. This is done by means of the input-control signal COMMAND. COMMAND is the logical OR of the input-control signals EXC, DTOX and DTIX. The HLDC will periodically check the status of the COMMAND input-control signal. If this signal is TRUE, the HLDC will then check the DTOX, DTIX and EXC input-control signals to ascertain which Varian I/O instruction has been executed.

The HLDC can only receive one command at a time. That is, the amount of time from the COMMAND signal going TRUE to the HLDC inspecting the COMMAND signal will be variable, and will be dependent on the microprogram routine

the HLDC is currently performing. For example, if the HLDC is currently transferring a sector of data to or from disc, it will probably only inspect the COMMAND signal after the sector of data has been fully transferred. During this period of time, the Varian must not initiate another I/O command, as this may cause an I/O command to be lost. For example, if the Varian executes a second single-word output transfer instruction before the first single-word output instruction has been processed by the HLDC, the data in the VAR.DIN latch will be overwritten, causing the first command to be lost.

The following method of communication between the HLDC and Varian will ensure that the above situation will not occur.

The Varian must initially check if the HLDC is ready to receive a command. It would do this by checking whether or not the READY latch is set by means of a 'SEN 014' I/O instruction. If the HLDC is ready, the Varian will execute the appropriate I/O instruction (i.e. external control, single-word output transfer or single-word input transfer). This instruction will set up the appropriate latches. In the case of an external control instruction, the EXC and EXC-address latches will be set up; in the case of a single-word output transfer instruction, the DTOX and VAR.DIN latches will be set up; and in the case of a single-word input transfer instruction, the DTIX latch will be set up. The I/O instruction will, at the same time, clear (reset) all the Sense latches. Clearing the Sense latches will ensure that the HLDC is no longer ready.

The HLDC, when it next inspects the COMMAND input-control signal, will see than an I/O command has been initiated. If the EXC latch is set, the HLDC will perform a jump to the address specified by the EXC-address latch. If the DTIX latch is set, the HLDC will know that the Varian has received the data word that was set up in the VAR.DOUT latch. If the DTOX latch is set, the HLDC will gate the data from the VAR.DIN latch into the CPA.

Once the command has been processed (for example, when the data in the VAR.DIN latch has been gated into the CPA), then the three command latches (i.e. EXC, DTOX and DTIX latches) will be cleared using the CLR.COMMAND

output-control signal. The appropriate Sense latch (e.g. the READY latch) can then be set by the HLDC to indicate to the Varian that the HLDC is ready to receive another command.

Example D.2 illustrates how this communication will work for a single-word output transfer command.

D.5.2 <u>Direct Memory Access</u>

Direct memory access (DMA) transfers between the HLDC and the Varian use the following input- and output-control signals (refer Appendix E, Board 5, Page 2).

<u>Input-control Signals</u> : DMA.FIN
<u>Output-control Signals</u> : VAR.DOUT, VAR.DIN, DMA.OUT and DMA.IN

<u>DMA Transfers from Varian to HLDC</u>

When transferring a word of data from the Varian to the HLDC via DMA, the appropriate address must first be set up in the CPA's memory address register, and then output to the VAR.DOUT latch by means of the VAR.DOUT output-control signal. The output-control signal DMA.OUT is then given. This signal sets the DMA.OUT latch which requests a DMA transfer from the Varian to the HLDC. This signal also causes the DMA.FIN input-control signal to go FALSE. When the DMA transfer is complete, the DMA.OUT latch will be reset, and the DMA.FIN signal will go TRUE. This signal indicates the completion of the DMA transfer. On completion of the transfer, the DMA data, which is automatically gated into the VAR.DIN latch, can be gated onto the M-bus by a VAR.DIN output-control signal. The following example illustrates how one word, stored in main memory address 100, can be transferred into the CPA's accumulator (AC) via DMA.

```
    CLR(A)
    K11(A)        *              100
# Set up the VAR.DOUT latch with 100.
    *             *              VAR.DOUT
# Request a DMA transfer from Varian to HLDC.
    *             *              DMA.OUT
```

```
# Wait for the transfer to complete.
     *              INCT     DMA.FIN
# Input the DMA data into the accumulator.
     ACM(A)         *                        VAR.DIN
```

DMA Transfer from HLDC to Varian

When transferring a word of data from the HLDC to the Varian via DMA, the appropriate main memory address must first be set up in the CPA's memory address register, and output to the VAR.DOUT latch by means of a VAR.DOUT output-control signal. The data to be output to this latch is then set up in the memory address register, and a DMA.IN output-control signal is given. This signal sets up the DMA.IN latch which requests a DMA transfer from the HLDC to the Varian. This signal also causes the DMA.FIN input-control signal to go FALSE. When the DMA transfer is complete, the DMA.IN latch will be reset, and the DMA.FIN signal will go TRUE. This signal indicates the completion of the DMA transfer. The following example illustrates how a word of data stored in register RO can be transferred to the main memory address 200 using DMA.

```
     CLR(A)
     K11(A)         *                        200
# Set up the VAR.DOUT latch with 200
     *              *                        VAR.DOUT
# Transfer the contents of Register RO to the memory address register
# and request a DMA transfer from HLDC to Varian
     LMI(RO)        *                        DMA.IN
# Wait for the transfer to complete
     *              INCT        DMA.FIN
```

D.5.3 Interrupts

The HLDC initiates an interrupt by means of the following input- and output-control signals (Refer Appendix E, Board 5, Page 2).

Input-control Signals : DMA.FIN
Output-control Signals : VAR.DOUT and INTERRUPT

To initiate an interrupt, the interrupt address must first be set up in the CPA's memory address register, and then output to the VAR.DOUT latch by means of a VAR.DOUT output-control signal. The output-control signal INTERRUPT is then given. This signal sets the INTERRUPT latch which requests an interrupt. When the interrupt has been acknowledged by the Varian, the INTERRUPT latch will be reset. This will cause the DMA.FIN input-control signal to go TRUE.

The following example illustrates how an interrupt to main memory address 100 is given by the HLDC.

```
    CLR(A)
    K11(A)        *              100
# Set up the VAR.DOUT latch with 100
    *             *              VAR.DOUT
# Request an interrupt
    *             *              INTERRUPT
# Wait for interrupt to be accepted (if necessary)
    *             INCT     DMA.FIN
```

Varian program: Output the data in the accumulator to the HLDC by means of a single-word output transfer I/O instruction.

```
AGAIN      SEN       014, READY
           NOP
           JMP       AGAIN
READY      OAR       014
```

HLDC program: The following microprogram sequence illustrates the part of the microprogram associated with receiving a word of data from the Varian.

```
.LOOP.     *         INCT        COMMAND
           *         JMPT        DTOX          .DTOX.
           *         JMPT        DTIX          .DTIX.
           *         JMPT        EXC           .EXC.
           *         HLT                                # Hardware error
.DTOX.     # Gate data from the VAR.DIN latch into the CPA
           LMM(A)    *                         VAR.DIN
# Clear the DTOX latch
           *         *                         CLR.COMMAND
# Indicate to the Varian that the HLDC is again ready
           *         *                         READY
                                 .
                                 .
                                 .
           perform processing associated with DTOX
                                 .
                                 .
                                 .
           *         JMP                       .LOOP.
```

Example D.2

APPENDIX E

THE HIGH LEVEL DISC CONTROLLERS CIRCUIT DIAGRAMS

The HLDC consists of 11 boards. Four boards are housed in a chassis called card cage A, six boards are housed in a chassis called card cage B, and the remaining board resides in the Varian chassis. The boards are numbered 1 to 11, and the functions performed by each of these boards is as follows :

Card Cage A

Board 1 : Console, Local Memory and Microprogram Memory Interfaces
Board 2 : Microprogram Memory
Board 3 : Central Processing Unit
Board 4 : Computer and Disc Drive Interfaces

Varian Chassis

Board 5 : Varian 620/L Minicomputer Interface

Card Cage B

Board 6 : CalComp CD1 Disc Drive Interface
Board 7 : VFO Single Density
Board 8 : Read/Write/Clock
Board 9 : Drive Multiplex Transmitter/Receiver
Board 10 : Drive Multiplex Transmitter/Receiver
Board 11 : Drive Simplex Transmitter/Receiver

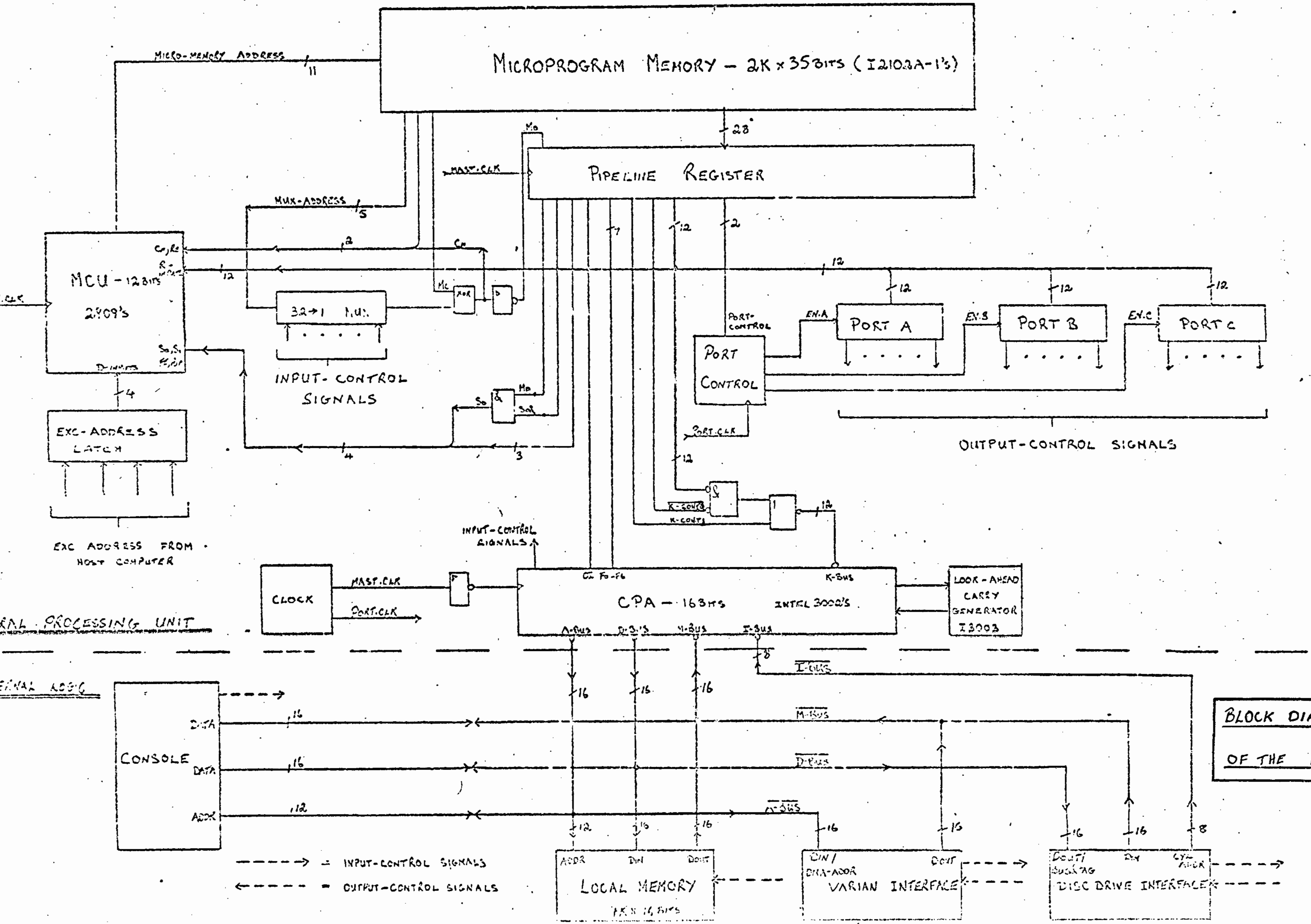
It was mentioned in Chapter 3.12 that certain of the Telefile disc controller circuits have been used in the HLDC. These circuits are boards 7 to 11. The circuit diagrams for these boards are not given in the text, but are available from the reference[1]. The wiring diagrams for these boards are, however, given in this Appendix, as are the circuit diagrams for boards 1 to 6 together with the board layouts and wiring diagrams. Block diagrams are given at certain places in this Appendix to make the circuit diagrams easier to follow.

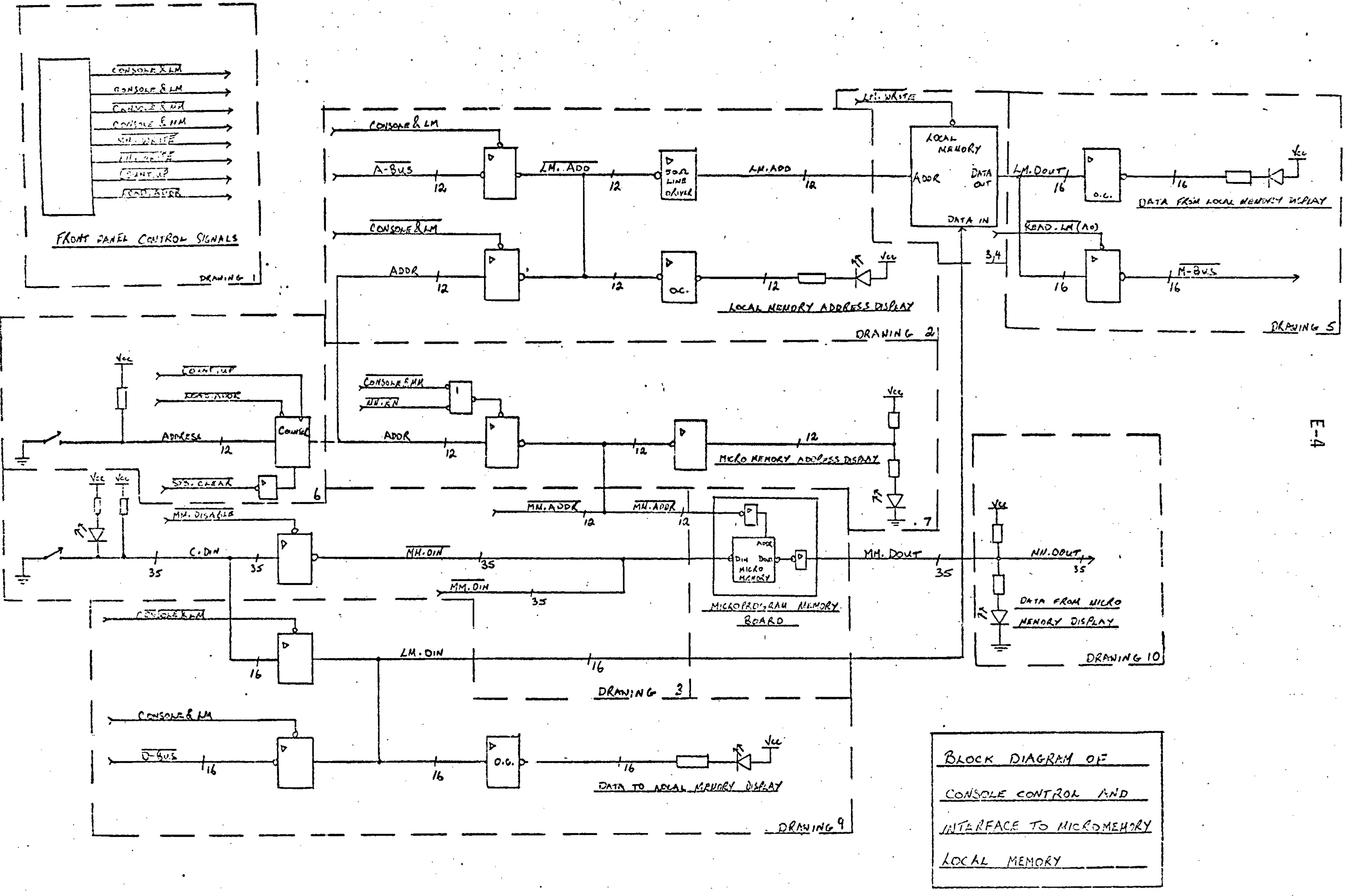
[1] Operation and Maintenance Manual, DC-16 Disc Drive Controller, Telefile Computer Products, Irvine, California (September 1970).

Index to the Circuit Diagrams

Page No.

Block Diagrams of the HLDC E-3
Console, Local Memory and Microprogram Memory Interfaces . E-8
Clock E-22
Microprogram Memory E-23
Central Processing Unit E-26
Disc Drive Interface E-35
Computer Interface E-42
Varian 620/L Minicomputer Interface E-48
CalComp CD1 Disc Drive Interface E-56
Wiring Diagrams for the Telefile Boards E-59


MICROPROGRAM MEMORY – 2K x 35 BITS (I2102A-1's)
MICRO-MEMORY ADDRESS
PIPELINE REGISTER
MAST.CLK
MUX-ADDRESS
MCU – 12 BITS
2909's
32→1 MUX
INPUT-CONTROL SIGNALS
EXC-ADDRESS LATCH
EXC ADDRESS FROM HOST COMPUTER
PORT CONTROL
PORT-CONTROL
PORT A
PORT B
PORT C
EN.A
EN.B
EN.C
OUTPUT-CONTROL SIGNALS
PORT.CLK
K-CONT
INPUT-CONTROL SIGNALS
CLOCK
CPA – 16 BITS
INTEL 3002's
K-BUS
A-BUS
D-BUS
M-BUS
I-BUS
LOOK-AHEAD CARRY GENERATOR I3003
CONSOLE
DATA
ADDR
LOCAL MEMORY
VARIAN INTERFACE
DISC DRIVE INTERFACE
DIN
DOUT
BLOCK DIAGRAM OF THE HLDC
INPUT-CONTROL SIGNALS
OUTPUT-CONTROL SIGNALS

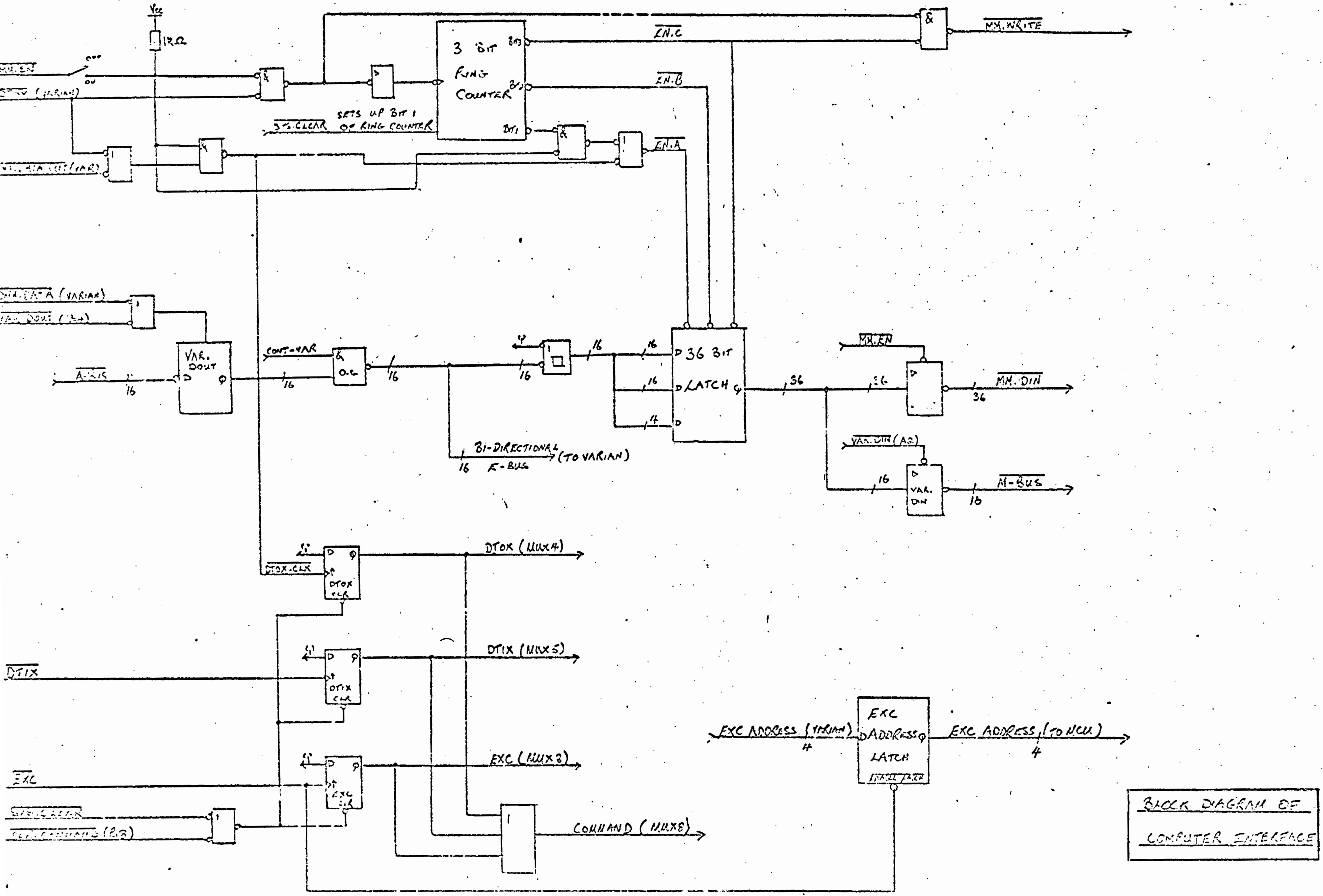


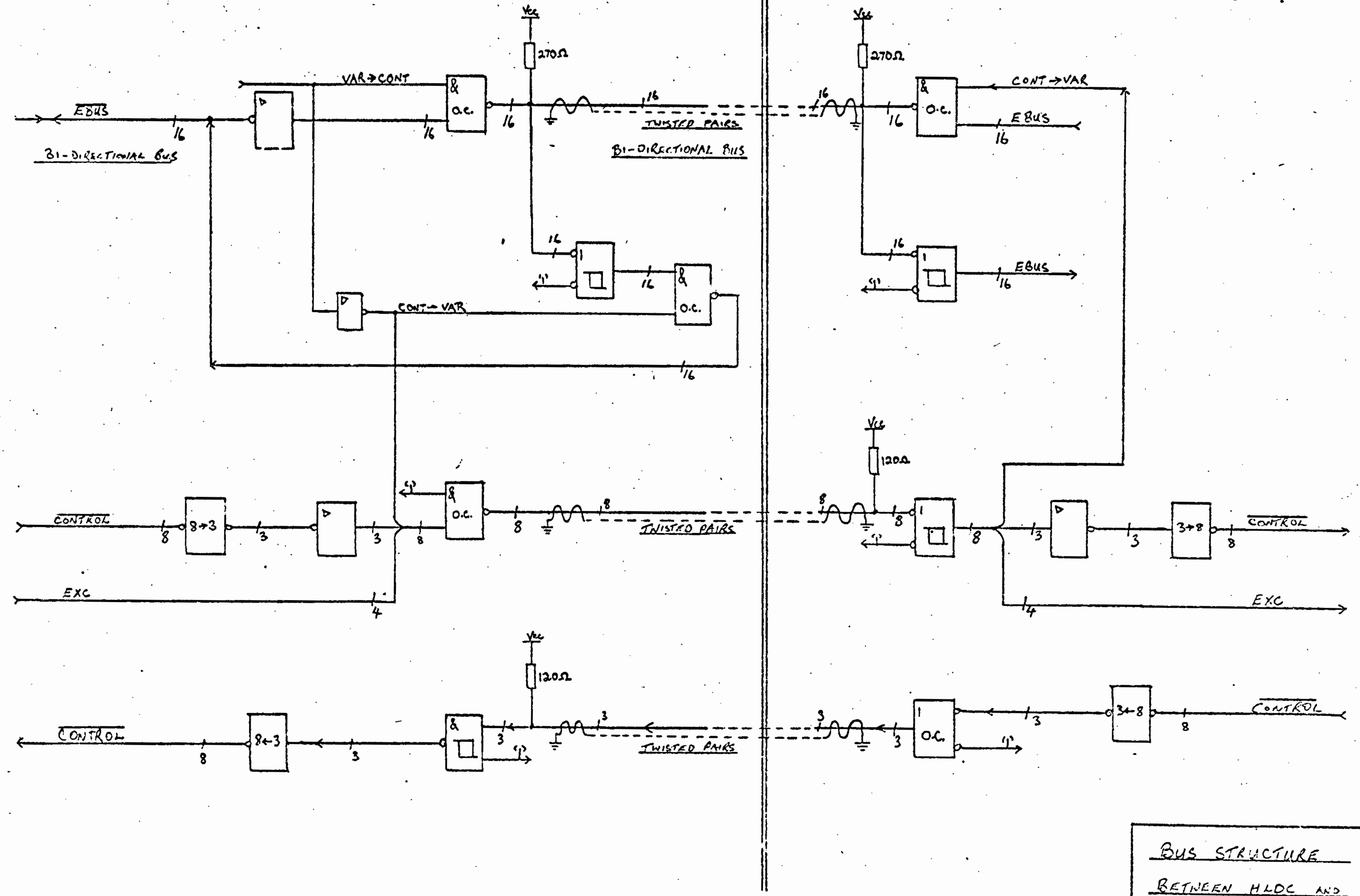
BLOCK DIAGRAM OF
CONSOLE CONTROL AND
INTERFACE TO MICROMEMORY
LOCAL MEMORY
FRONT PANEL CONTROL SIGNALS
DRAWING 1
CONSOLE & LM
A-BUS
LM. ADD
50Ω LINE DRIVER
LM. ADD
LOCAL MEMORY
ADDR
DATA OUT
DATA IN
LM. WRITE
LM. DOUT
O.C.
DATA FROM LOCAL MEMORY DISPLAY
READ. LM (A0)
M-BUS
3,4
DRAWING 5
ADDR
LOCAL MEMORY ADDRESS DISPLAY
DRAWING 2
COUNT UP
ADDRESS
COUNTER
SYS. CLEAR
MM. DISABLE
C. DIN
6
MM. EN
MICRO MEMORY ADDRESS DISPLAY
7
MM. ADDR
MM. DIN
MM. DOUT
DIN
DOUT
MICRO MEMORY
MICROPROGRAM MEMORY BOARD
DATA FROM MICRO MEMORY DISPLAY
DRAWING 10
LM. DIN
DRAWING 3
D-BUS
DATA TO LOCAL MEMORY DISPLAY
DRAWING 9
Vcc

BLOCK DIAGRAM OF

DISC DRIVE INTERFACE

READ.DATA (C0)
READ.DATA
ENRDDA
READ.CLK (C1)
READ.CLK
ENRDCK
SYS.CLEAR
TO DISC
TAG.EN (C8)
TAG LATCHES
TAG LINES
TO DISC
BUS.EN (C9)
BUS LATCHES
BUS LINES
D-BUS
SYS.CLEAR
DRIVE.SELECT A4-A6
DRIVE SELECT LATCHES
SELECT DRIVE
TO DISC
DRIVE.CLK (A7)
DATA.CLK
SYNC.EN (C5)
SYNC LATCH
SYNC.CLR (C6)
SYNC-REQ LATCH
SYNC (MUX13)
DATA.FROM.DISC
DATA.CLK
16 BIT SHIFT REGISTER
DATA.TO.DISC
DATA.TO.DISC
GWRDATA
DISC.DIN (A3)
DATA FROM DISC
DISC.DIN LATCH
M-BUS
DISC.DOUT (C3)
WRITE (C4)
WRITE LATCH
SYS.CLEAR
EQ.16 LATCH
EQ.16 (MUX14)
CLR.EQ16 (C2)
MOD16 COUNTER
CLR.MOD16


3 BIT RING COUNTER
MM.WRITE
EN.C
EN.B
EN.A
Vcc
1kΩ
SETS UP BIT 1 OF RING COUNTER
36 BIT LATCH
VAR. DOUT
CONT-VAR
O.C.
BI-DIRECTIONAL (TO VARIAN)
E-BUS
MM.EN
MM.DIN
VAR.DIN (A0)
VAR. DIN
M-BUS
DTOX (MUX4)
DTOX.CLK
DTOX CLK
DTIX (MUX5)
DTIX
DTIX CLK
EXC (MUX3)
EXC
EXC CLK
COMMAND (MUX8)
EXC ADDRESS (VARIAN)
EXC ADDRESS LATCH
EXC ADDRESS (TO NCU)
BLOCK DIAGRAM OF COMPUTER INTERFACE



BUS STRUCTURE BETWEEN HLDC AND VARIAN

J3 CONNECTOR

	1	2	3	4	5	6	7	8	9	10	11	
A	MM.ADDR.DISP	74193	74368	7407	74128	A6		PINS FOR LM		PINS FOR LM		A
B	LM.ADDR.DISP	74193	74368	7407	74128							B
D	ADDRESS	74193	74368	7405	74128	D6						D
E	MM. DOUT		74368	7405	7401	E6					7405	E
F	MI. DOUT		74367	74367	7404						7405	F
G			74368	74368	74368	G6					7405	G
H	MM. DOUT				74368	H6						H
K			7408	7404	74368							K
L	CONTROLS	7400	7404	74368	7400	L6						L
M	7400	7400	74368	74368	7400							M
N	C.DIN	74368	74368	74368	7404	N6						N
R		74368	74368	74368	74123	7440						R
S	C.DIN	74367	74367	7405	COMPONENTS	74163						S
T	C.DIN	74367	7405	7405	74123	74S124						T
	1	2	3	4	5	6	7	8	9	10	11	

LMO LIES ON TOP OF BOARD 1 & LM1 LIES UNDERNEATH BOARD 1

J1 CONNECTOR

J2 CONNECTOR

CONSOLE, LOCAL MEMORY AND MICROMEMORY INTERFACES

CARD CAGE: A SLOT: 1 BOARD: 1

	1	2	3	4
	CONNECTOR			
BIT 0 A				
BIT 1 B				
BIT 2 C				
BIT 3 D				
BIT 4 E				
BIT 5 F				
BIT 6 G				
BIT 7 H				
	4'TH K	3'RD K	2'ND K	1'ST K

BOARD LAYOUT (ON TOP OF BOARD 1)

Pin	Signal	Pin	Signal
1	$\overline{CE2}$	2	LM.DIN 7
3	LM.DIN 6	4	LM.DIN 5
5	LM.DIN 4	6	LM.DIN 3
7	LM.DIN 2	8	$\overline{CE0}$
9	LM.DIN 1	10	LM.DIN 0
11	$\overline{CE1}$	12	LM.ADD 6
13	LM.ADD 7	14	LM.ADD 5
15	LM.ADD 8	16	$\overline{LM.WRITE}$
17	LM.ADD 9	18	LM.ADD 1
19	LM.ADD 2	20	LM.ADD 3
21	LM.ADD 4	22	LM.ADD 0
23	GND	24	Vcc
25	LM.DOUT 0	26	LM.DOUT 1
27	LM.DOUT 2	28	LM.DOUT 3
29	LM.DOUT 4	30	LM.DOUT 5
31	LM.DOUT 6	32	$\overline{CE3}$
33	LM.DOUT 7	34	—

CONNECTOR

	1	2	3	4
	CONNECTOR			
BIT 8 A				
BIT 9 B				
BIT 10 C				
BIT 11 D				
BIT 12 E				
BIT 13 F				
BIT 14 G				
BIT 15 H				
	4'TH K	3'RD K	2'ND K	1'ST K

BOARD LAYOUT (UNDERNEATH BOARD 1)

Pin	Signal	Pin	Signal
1	$\overline{CE2}$	2	LM.DIN 15
3	LM.DIN 14	4	LM.DIN 13
5	LM.DIN 12	6	LM.DIN 11
7	LM.DIN 10	8	$\overline{CE0}$
9	LM.DIN 9	10	LM.DIN 8
11	$\overline{CE1}$	12	LM.ADD 6
13	LM.ADD 7	14	LM.ADD 5
15	LM.ADD 8	16	$\overline{LM.WRITE}$
17	LM.ADD 9	18	LM.ADD 1
19	LM.ADD 2	20	LM.ADD 3
21	LM.ADD 4	22	LM.ADD 0
23	GND	24	Vcc
25	LM.DOUT 8	26	LM.DOUT 9
27	LM.DOUT 10	28	LM.DOUT 11
29	LM.DOUT 12	30	LM.DOUT 13
31	LM.DOUT 14	32	$\overline{CE3}$
33	LM.DOUT 15	34	—

CONNECTOR

ALL IC'S ARE 2102A-2

LOCAL MEMORY (4K x 16 BITS)

BITS 0–7 OF 4K

LOCAL MEMORY (4K x 16 BITS)

BITS 8–15 OF 4K

CARDCAGE: A SLOT: 1 BOARD: 1

BOARD NAME : CONSOLE, LOCAL MEMORY AND MICRO-MEMORY INTERFACES

CARD CAGE:A SLOT:1 BOARD:1

CONNECTOR J1

PIN	SIGNAL	PIN	SIGNAL
1	MM.DIN21	2	MM.DIN0
3	MM.DIN22	4	MM.DIN1
5	MM.DIN23	6	MM.DIN2
7	MM.DIN24	8	MM.DIN3
9	Polarising Pin	10	Polarising Pin
11	MM.DIN25	12	MM.DIN4
13	MM.DIN26	14	MM.DIN5
15	MM.DIN27	16	MM.DIN6
17	MM.DIN28	18	MM.DIN7
19	MM.DIN29	20	MM.DIN8
21	MM.DIN30	22	MM.DIN9
23	MM.DIN31	24	MM.DIN10
25	MM.DIN32	26	MM.DIN11
27	MM.DIN33	28	MM.DIN12
29	MM.DIN34	30	MM.DIN13
31	Spare	32	MM.DIN14
33	Spare	34	MM.DIN15
35	Spare	36	MM.DIN16
37	Spare	38	MM.DIN17
39	Spare	40	MM.DIN18
41	Spare	42	MM.DIN19
43	Spare	44	MM.DIN20

CONNECTOR J2

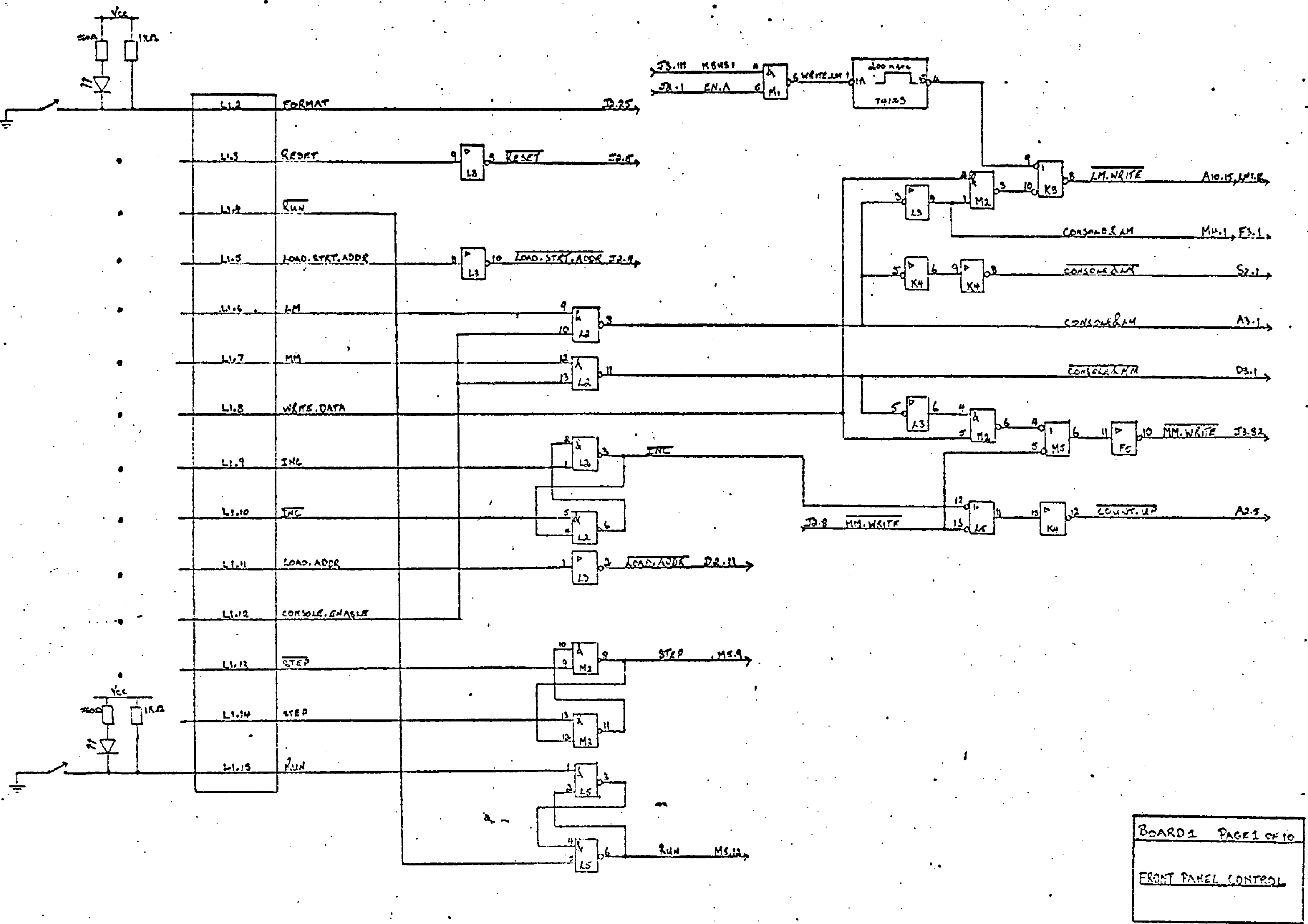
PIN	SIGNAL	PIN	SIGNAL
1	EN.A	2	EN.B
3	EN.C	4	LOAD.STRT.ADDR
5	RESET	6	MCU ADDR.EN
7	SYS.CLEAR	8	MM.WRITE
9	EXC1	10	EXC2
11	EXC3	12	EXC4
13	EXC	14	DTOX
15	DTIX	16	MM.EN
17	DMA.FIN	18	Spare
19	COMMAND	20	SYNC
21	EQ.16	22	SGLDEN
23	UNTSEL	24	ATTEN
25	FORMAT	26	PORT.CLK
27	Spare	28	Spare
29	Spare	30	Spare
21	Spare	32	PORT.CLK
33	Spare	34	Spare
35	Polarising Pin	36	Polarising Pin
37	Spare	38	Spare
39	Spare	40	Spare
41	Spare	42	Spare
43	Spare	44	Spare

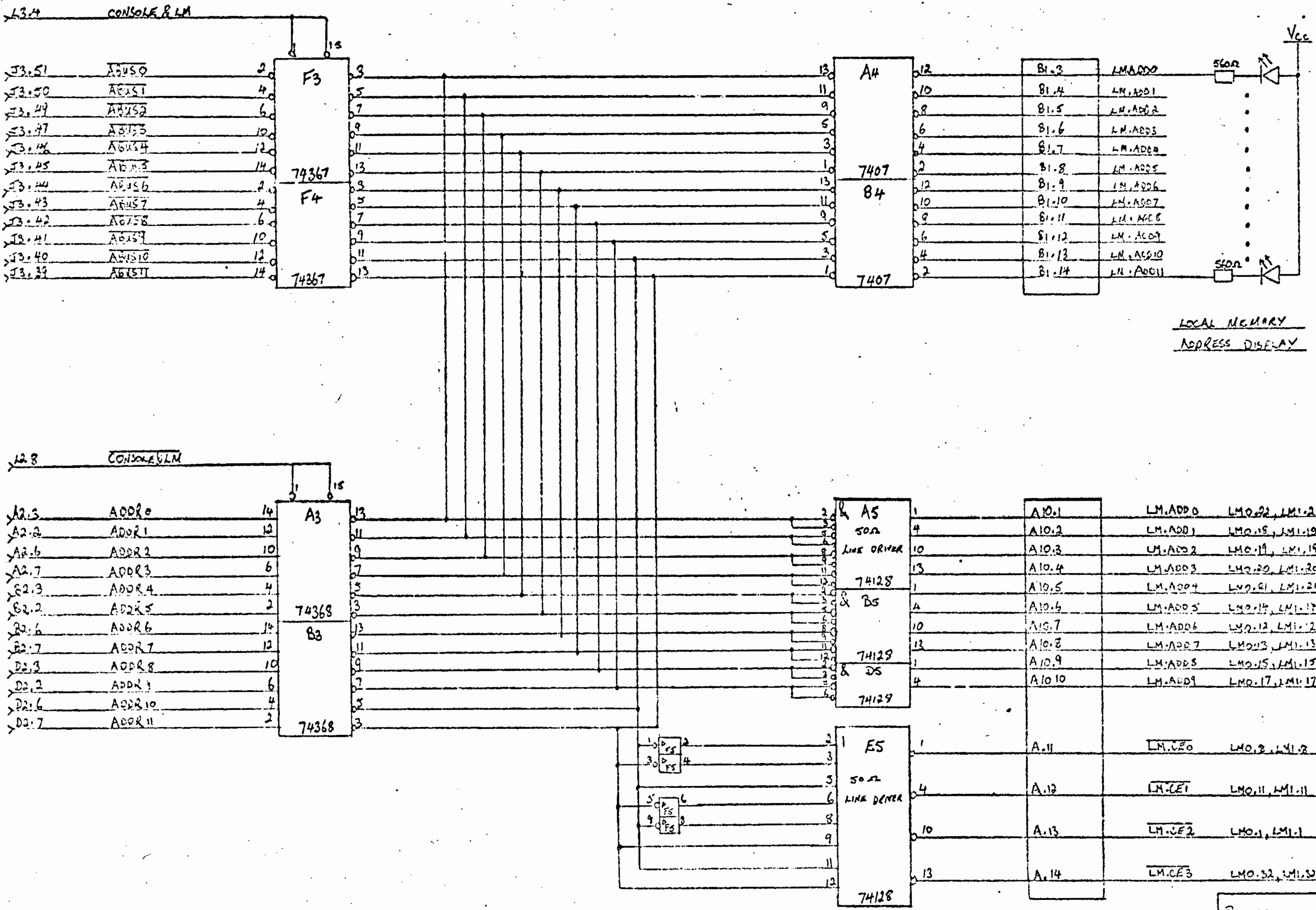
BOARD NAME : CONSOLE, LOCAL MEMORY AND MICRO-MEMORY INTERFACES

CARD CAGE:A SLOT: 1 BOARD: 1

CONNECTOR J3

PIN	SIGNAL	PIN	SIGNAL	PIN	SIGNAL	PIN	SIGNAL
1	GND	2	MM.DOUT0	63	DBUS6	64	DBUS5
3	MM.DOUT1	4	MM.DOUT2	65	DBUS4	66	DBUS3
5	MM.DOUT3	6	MM.DOUT4	67	DBUS2	68	DBUS1
7	MM.DOUT5	8	MM.DOUT6	69	DBUS0	70	MM.ADD0
9	MM.DOUT7	10	MM.DOUT8	71	MM.ADD1	72	MM.ADD2
11	MM.DOUT9	11	MM.DOUT10	73	MM.ADD3	74	MM.ADD4
13	MM.DOUT11	14	MM.DOUT12	75	MM.ADD5	76	MM.ADD6
15	MM.DOUT13	16	MM.DOUT14	77	MM.ADD7	78	MM.ADD8
17	MM.DOUT15	18	MM.DOUT16	79	MM.ADD9	80	MM.ADD10
19	MM.DOUT17	20	MM.DOUT18	81	MM.ADD11	82	MM.WRITE
21	MM.DOUT19	22	MM.DOUT20	83	MAST.CLK	84	MBUS15
23	MM.DOUT21	24	MM.DOUT22	85	MBUS14	86	MBUS13
25	MM.DOUT23	26	MM.DOUT24	87	MBUS12	88	MBUS11
27	MM.DOUT25	28	MM.DOUT26	89	MBUS10	90	MBUS9
29	MM.DOUT27	30	MM.DOUT28	91	MBUS8	92	MBUS7
31	MM.DOUT29	32	MM.DOUT30	93	MBUS6	94	MBUS5
33	MM.DOUT31	34	MM.DOUT32	95	MBUS4	96	MBUS3
35	MM.DOUT33	36	MM.DOUT34	97	MBUS2	98	MBUS1
37	Spare	38	Spare	99	MBUS0	100	GND
39	ABUS11	40	ABUS10	101	KBUS11	102	KBUS10
41	ABUS9	42	ABUS8	103	KBUS9	104	KBUS8
43	ABUS7	44	ABUS6	105	KBUS7	106	KBUS6
45	ABUS5	46	ABUS4	107	KBUS5	108	KBUS4
47	ABUS3	48	GND	109	KBUS3	110	KBUS2
49	ABUS2	50	ABUS1	111	KBUS1	112	KBUS0
51	ABUS0	52	DBUS15	113	ABUS15	114	ABUS14
53	DBUS14	54	DBUS13	115	ABUS13	116	ABUS12
55	DBUS12	56	DBUS11	117	Spare	118	+5VOLTS
57	DBUS10	58	DBUS9	119	Spare	120	+5VOLTS
59	DBUS8	60	DBUS7	121	+5VOLTS	122	GND
61	UNUSED	62	UNUSED				

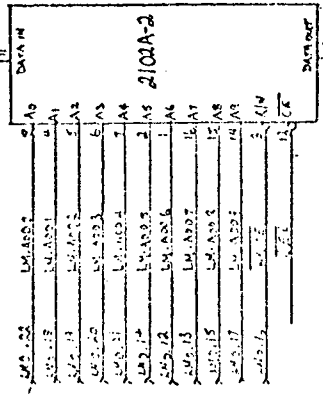

BOARD 1 PAGE 1 OF 10
FRONT PANEL CONTROL
FORMAT
RESET
RUN
LOAD.STRT.ADDR
LM
MM
WRITE.DATA
INC
LOAD.ADDR
CONSOLE.ENABLE
STEP
LM.WRITE
MM.WRITE
COUNT.UP
74123

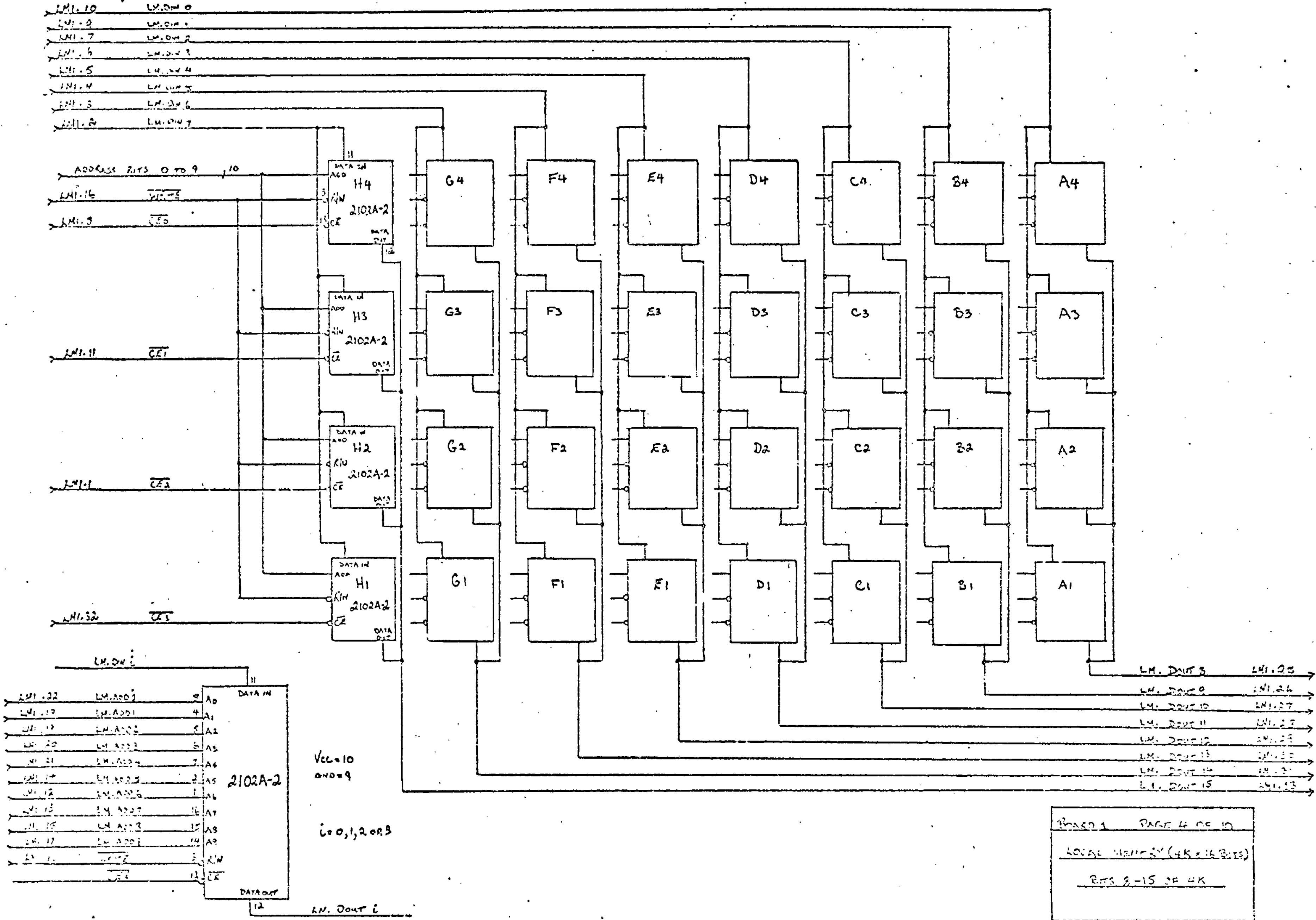

CONSOLE & LM
CONSOLE LM
F3
F4
74367
A4
B4
7407
A3
B3
74368
A5
50Ω
LINE DRIVER
74128
B5
74129
D5
E5
Vcc
560Ω
LOCAL MEMORY
ADDRESS DISPLAY
LM.ADD0
LM.ADD1
LM.ADD2
LM.ADD3
LM.ADD4
LM.ADD5
LM.ADD6
LM.ADD7
LM.ADD8
LM.ADD9
LM.CE0
LM.CE1
LM.CE2
LM.CE3
ADDR0
ADDR1
ADDR2
ADDR3
ADDR4
ADDR5
ADDR6
ADDR7
ADDR8
ADDR10
ADDR11
BOARD 1 PAGE 2 OF 10
LOCAL MEMORY ADDRESS

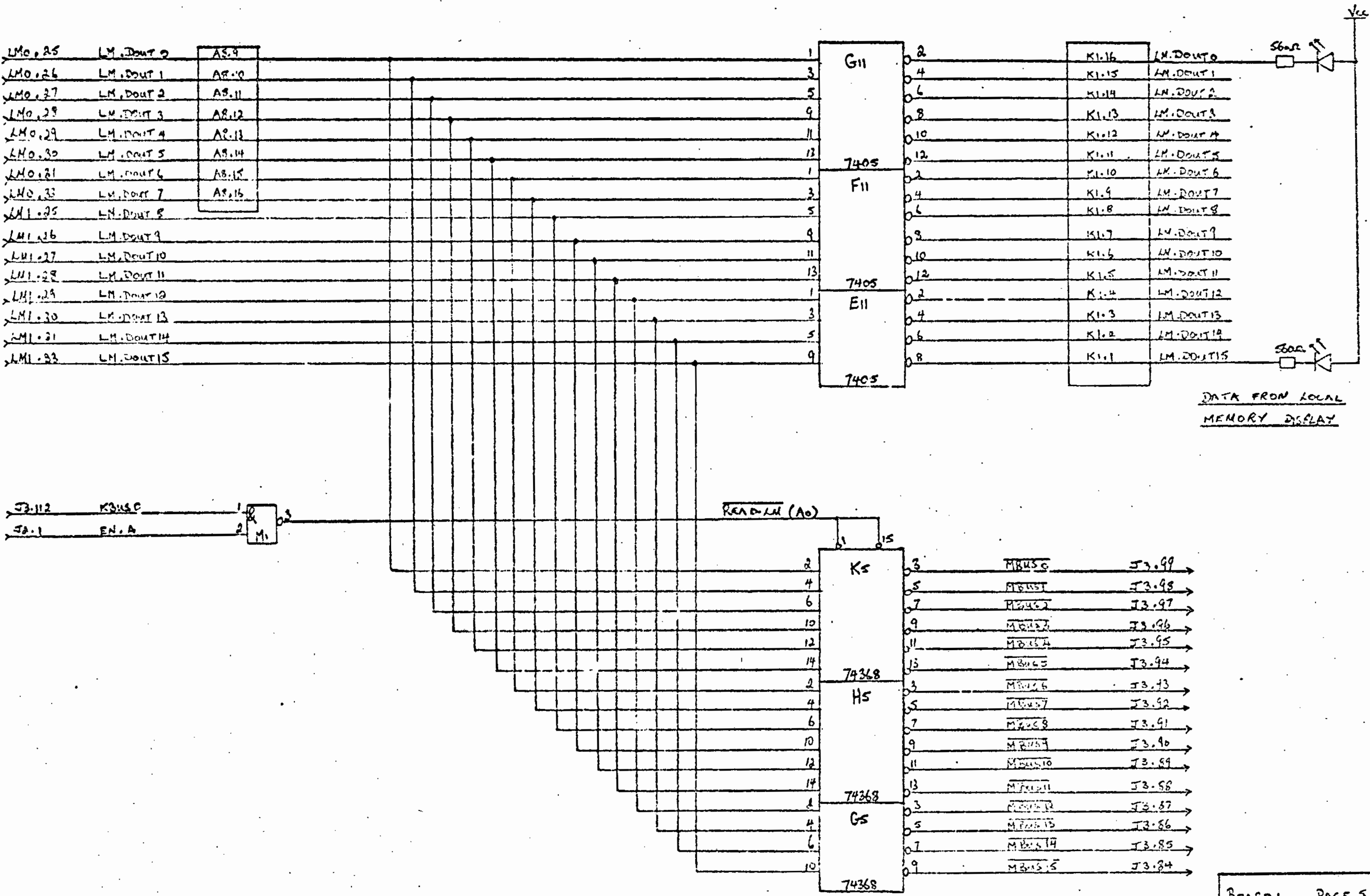
BOARD 1 PAGE 3 OF 10

LOCAL MEMORY (4K x 16 BITS)

BITS 0-7 OF 4K

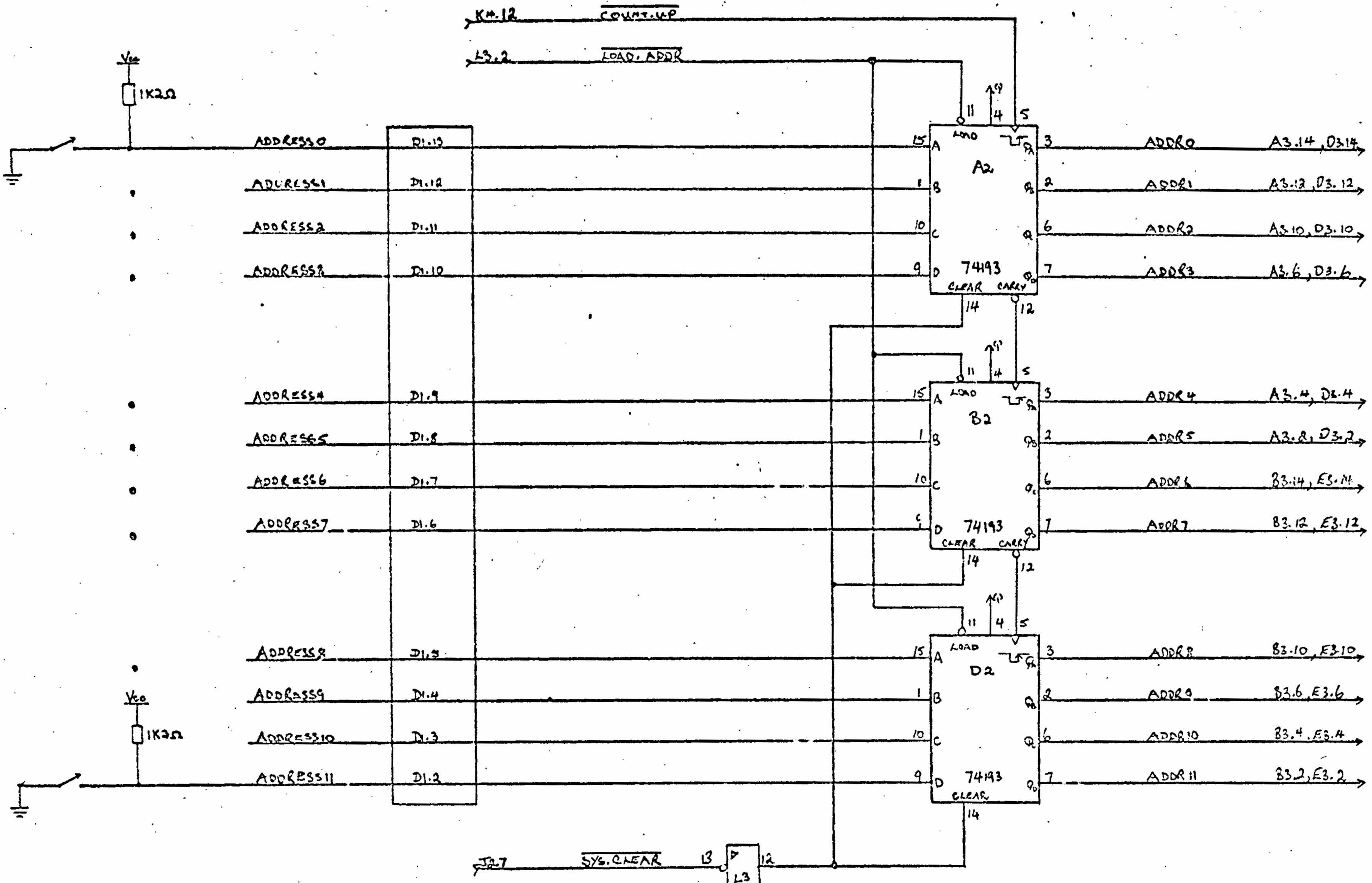


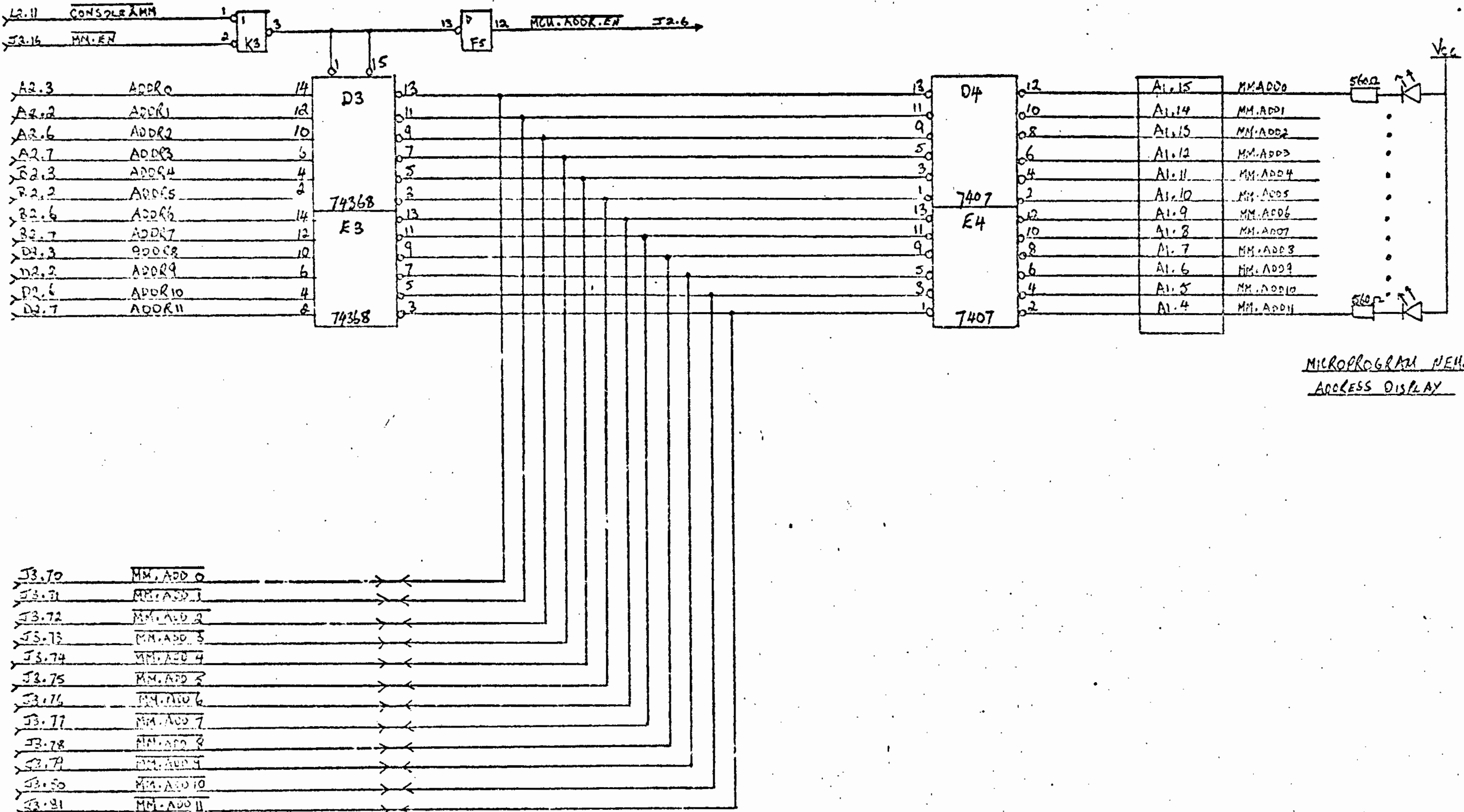

ADDRESS BITS 0 TO 9
H4
2102A-2
H3
2102A-2
H2
2102A-2
H1
2102A-2
G4
F4
E4
D4
B4
A4
G3
F3
E3
D3
C3
B3
A3
G2
F2
E2
D2
C2
B2
A2
G1
F1
E1
D1
C1
B1
A1
2102A-2
Vcc=10
GND=9
i=0,1,2 or 3
BITS 8-15 OF 4K

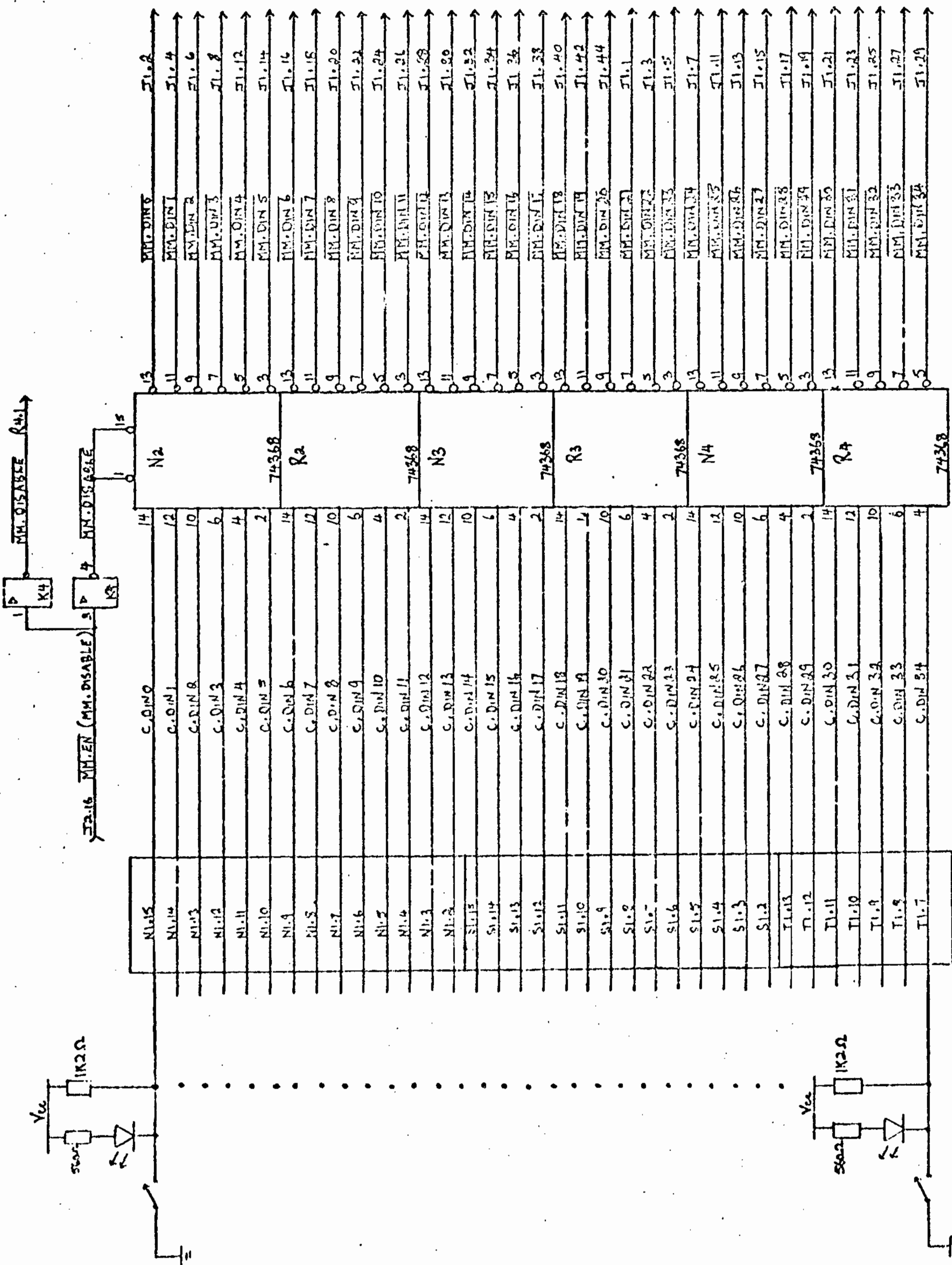


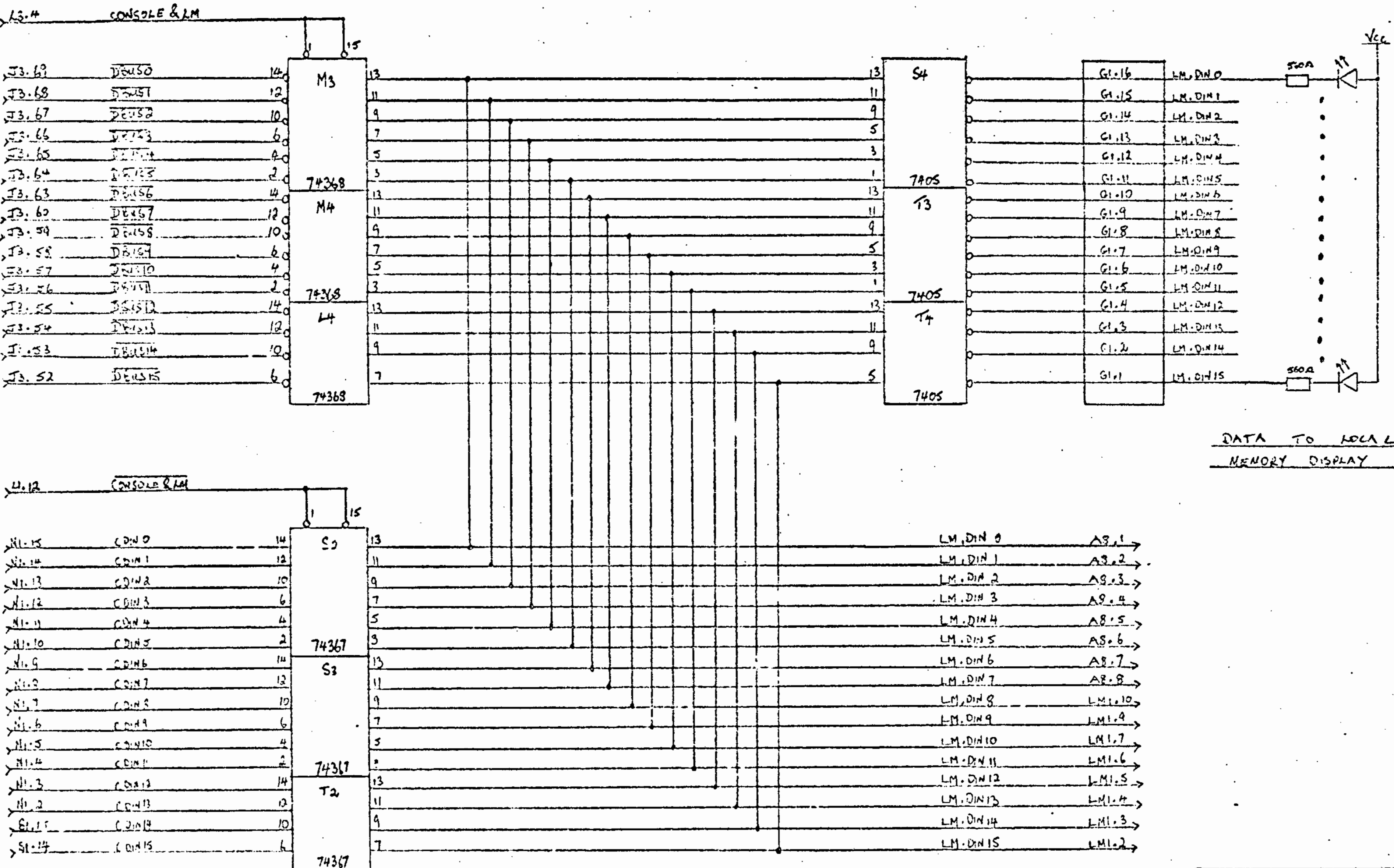
BOARD 1 PAGE 5 OF 10

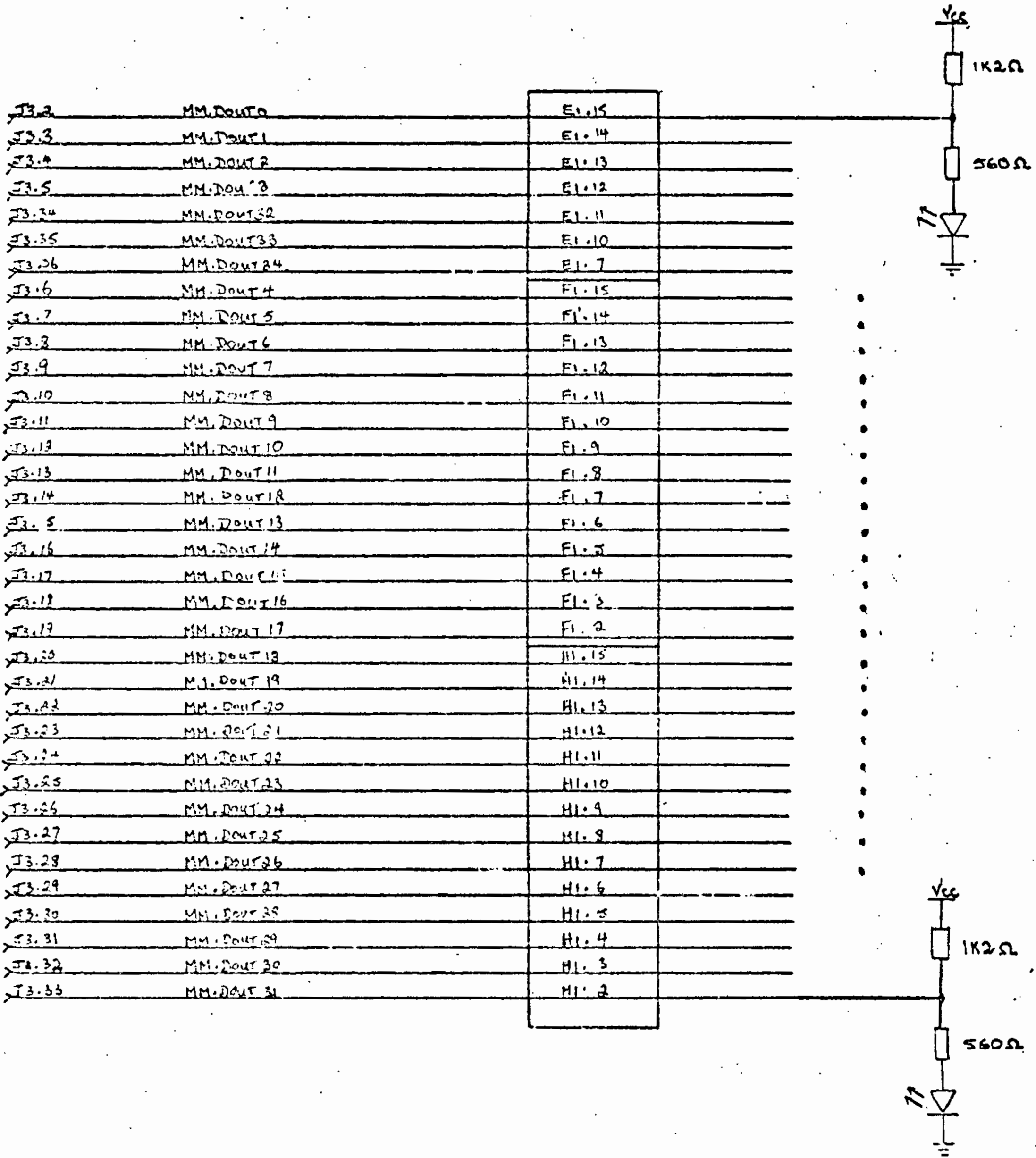
DATA FROM LOCAL MEMORY


KM.12
COUNT.UP
L3.2
LOAD.ADDR
Vcc
1K2Ω
ADDRESS0
ADDRESS1
ADDRESS2
ADDRESS3
ADDRESS4
ADDRESS5
ADDRESS6
ADDRESS7
ADDRESS8
ADDRESS9
ADDRESS10
ADDRESS11
D1.13
D1.12
D1.11
D1.10
D1.9
D1.8
D1.7
D1.6
D1.5
D1.4
D1.3
D1.2
A2
B2
D2
74193
LOAD
CLEAR
CARRY
ADDR0
ADDR1
ADDR2
ADDR3
ADDR4
ADDR5
ADDR6
ADDR7
ADDR8
ADDR9
ADDR10
ADDR11
A3.14, D3.14
A3.12, D3.12
A3.10, D3.10
A3.6, D3.6
A3.4, D3.4
A3.2, D3.2
B3.14, E3.14
B3.12, E3.12
B3.10, E3.10
B3.6, E3.6
B3.4, E3.4
B3.2, E3.2
J2.7
SYS.CLEAR
L3
BOARD1 PAGE 6 OF 10
ADDRESS FROM FRONT
PANEL TO LOCAL MEMORY
AND MICROMEMORY


CONSOLE AMM
MM.EN
K3
F5
MCU.ADDR.EN
J2.6
D3
74368
E3
74368
D4
7407
E4
7407
Vcc
560Ω
MICROPROGRAM MEMORY
ADDRESS DISPLAY
ADDR0
ADDR1
ADDR2
ADDR3
ADDR4
ADDR5
ADDR6
ADDR7
ADDR8
ADDR9
ADDR10
ADDR11
MM.ADD 0
MM.ADD 1
MM.ADD 2
MM.ADD 3
MM.ADD 4
MM.ADD 5
MM.ADD 6
MM.ADD 7
MM.ADD 8
MM.ADD 9
MM.ADD 10
MM.ADD 11
J3.70
J3.71
J3.72
J3.73
J3.74
J3.75
J3.76
J3.77
J3.78
J3.79
J3.80
J3.81
A1.15
A1.14
A1.13
A1.12
A1.11
A1.10
A1.9
A1.8
A1.7
A1.6
A1.5
A1.4
BOARD 1 PAGE 7 OF 10
MICROMEMORY ADDRESS

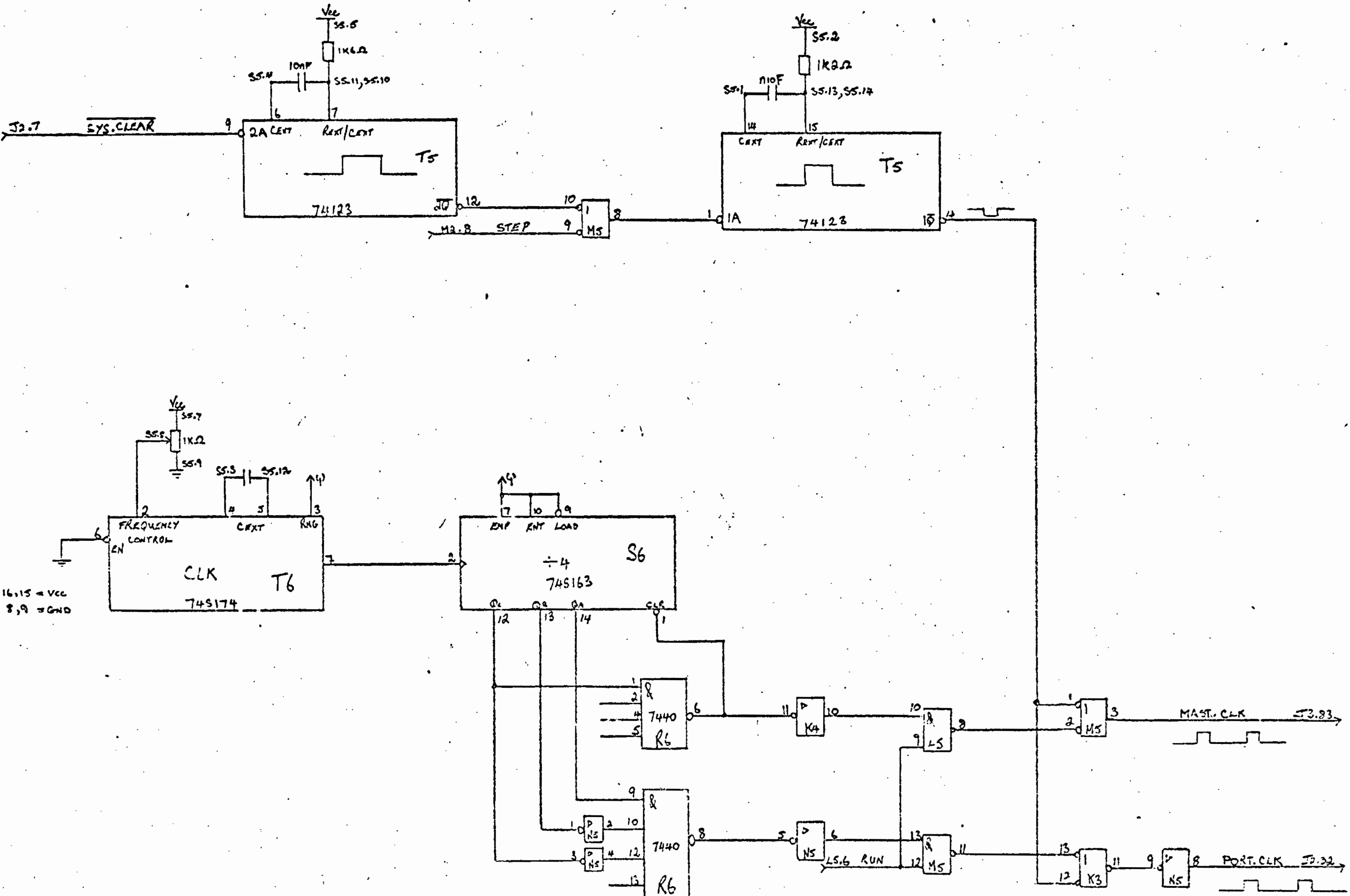

BOARD1 PAGE 3 OF 10
DATA FROM FRONT PANEL TO MICROMEMORY
DATA FROM FRONT PANEL
MM.DISABLE
MM.EN (MM.DISABLE)
N2
R2
N3
R3
N4
R4
74368
Vcc
1K2Ω
560Ω


CONSOLE & LM
M3
M4
L4
74368
S4
T3
T4
7405
Vcc
560Ω
DATA TO LOCAL
MEMORY DISPLAY
S2
S3
T2
74367
LM.DIN 0
LM.DIN 1
LM.DIN 2
LM.DIN 3
LM.DIN 4
LM.DIN 5
LM.DIN 6
LM.DIN 7
LM.DIN 8
LM.DIN 9
LM.DIN 10
LM.DIN 11
LM.DIN 12
LM.DIN 13
LM.DIN 14
LM.DIN 15
DATA FROM FRONT PANEL
BOARD 2 PAGE 9 OF 10
DATA TO LOCAL
MEMORY



DATA TO MICRO
MEMORY DISPLAY

BOARD 1 PAGE 10 OF 10
DATA FROM MICROMEMORY
TO FRONT PANEL DISPLAY


J2.7
SYS.CLEAR
T5
74123
M2.8
STEP
T5
74123
MAST. CLK
J3.33
PORT. CLK
J3.32
LS.6 RUN
T6
CLK
74S174
FREQUENCY CONTROL
FOR T6: 16,15 = Vcc
8,9 = GND
S6
÷4
74S163
7440
R6
7440
R6
BOARD 1
CLOCK

BOARD NAME : MICROPROGRAM MEMORY CARD CAGE : A SLOT : 2 BOARD 2

CONNECTOR J1

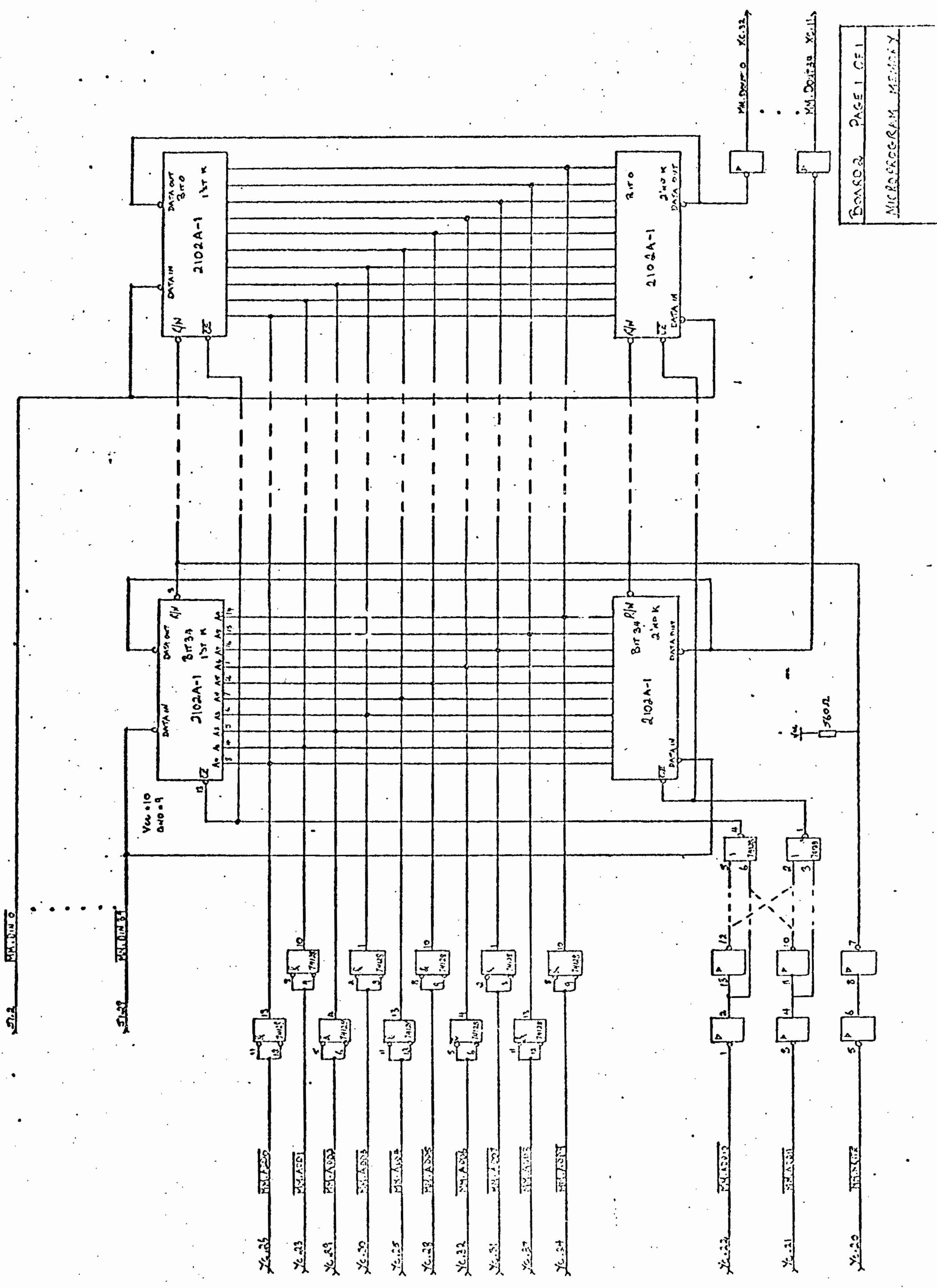
PIN	SIGNAL	PIN	SIGNAL
1		2	
3		4	
5		6	
7		8	
9		10	
11		12	
13		14	
15		16	
17		18	
19		20	
21		22	
23		24	
25	SPARE	26	SPARE
27		28	
29		30	
31		32	
33		34	
35		36	
37		38	
39		40	
41		42	
43		44	
45		46	
47		48	
49		50	

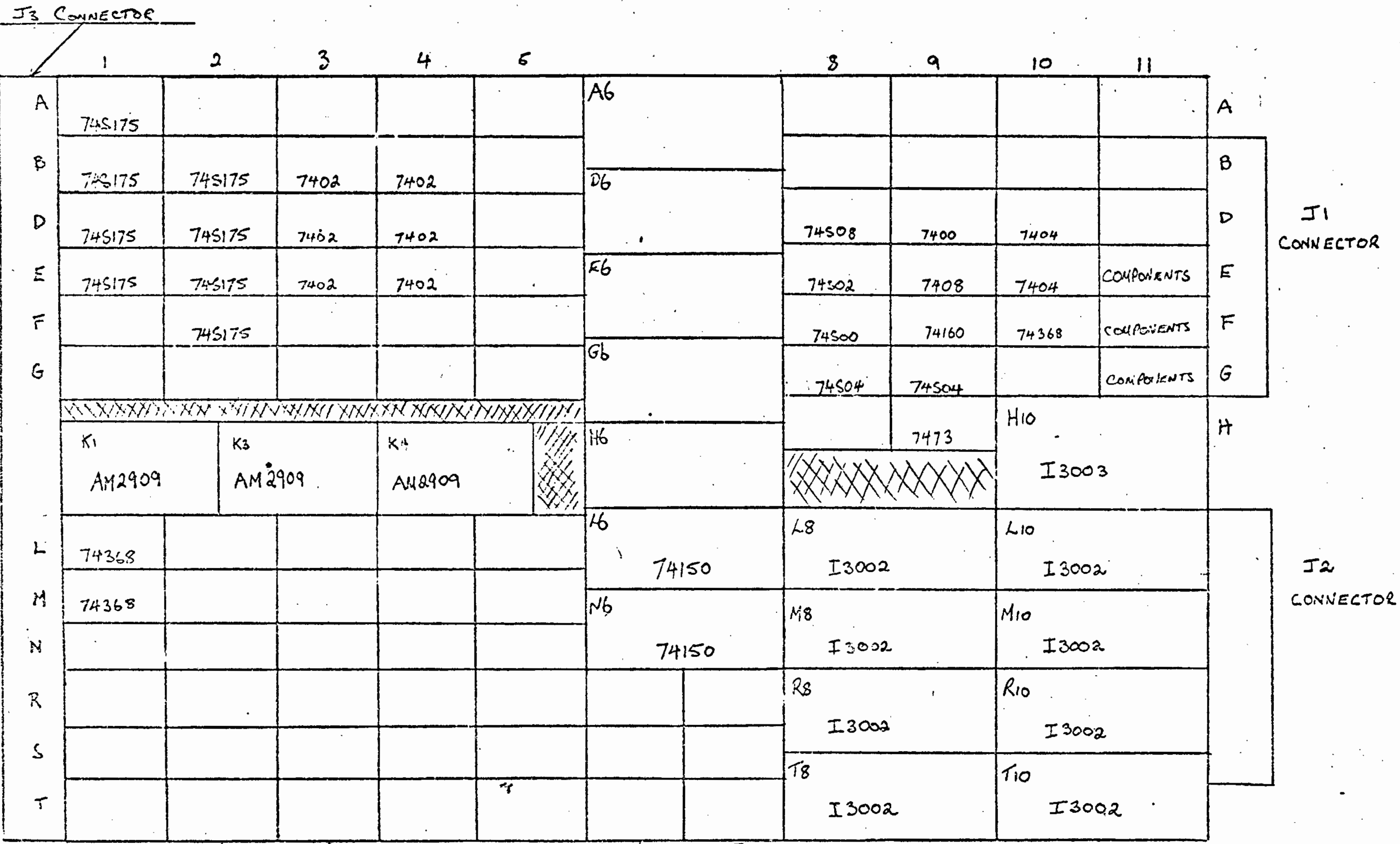
CONNECTOR J2

PIN	SIGNAL	PIN	SIGNAL
1	$\overline{\text{MM.DIN21}}$	2	$\overline{\text{MM.DIN0}}$
3	$\overline{\text{MM.DIN22}}$	4	$\overline{\text{MM.DIN1}}$
5	$\overline{\text{MM.DIN23}}$	6	$\overline{\text{MM.DIN2}}$
7	$\overline{\text{MM.DIN24}}$	8	$\overline{\text{MM.DIN3}}$
9	$\overline{\text{MM.DIN25}}$	10	$\overline{\text{MM.DIN4}}$
11	$\overline{\text{MM.DIN26}}$	12	$\overline{\text{MM.DIN5}}$
13	$\overline{\text{MM.DIN27}}$	14	$\overline{\text{MM.DIN6}}$
15	$\overline{\text{MM.DIN28}}$	16	$\overline{\text{MM.DIN7}}$
17	$\overline{\text{MM.DIN29}}$	18	$\overline{\text{MM.DIN8}}$
19	$\overline{\text{MM.DIN30}}$	20	$\overline{\text{MM.DIN9}}$
21	$\overline{\text{MM.DIN31}}$	22	$\overline{\text{MM.DIN10}}$
23	$\overline{\text{MM.DIN32}}$	24	$\overline{\text{MM.DIN11}}$
25	$\overline{\text{MM.DIN33}}$	26	$\overline{\text{MM.DIN12}}$
27	$\overline{\text{MM.DIN34}}$	28	$\overline{\text{MM.DIN13}}$
29		30	$\overline{\text{MM.DIN14}}$
31		32	$\overline{\text{MM.DIN15}}$
33		34	$\overline{\text{MM.DIN16}}$
35		36	$\overline{\text{MM.DIN17}}$
37		38	$\overline{\text{MM.DIN18}}$
39		40	$\overline{\text{MM.DIN19}}$
41		42	$\overline{\text{MM.DIN20}}$
43		44	
45		46	
47		48	
49		50	

BOARD NAME : MICROPROGRAM MEMORY CARD CAGE : A SLOT : 2 BOARD : 2

X CONNECTOR				Y CONNECTOR			
PIN	SIGNAL	PIN	SIGNAL	PIN	SIGNAL	PIN	SIGNAL
32	MM.DOUT1	32	MM.DOUT0	32		32	$\overline{\text{MM.ADD6}}$
31	MM.DOUT3	31	MM.DOUT2	31		31	$\overline{\text{MM.ADD7}}$
30	MM.DOUT5	30	MM.DOUT4	30		30	$\overline{\text{MM.ADD3}}$
29	MM.DOUT7	29	MM.DOUT6	29		29	$\overline{\text{MM.ADD2}}$
28	MM.DOUT9	28	MM.DOUT8	28		28	$\overline{\text{MM.ADD5}}$
27	MM.DOUT11	27	MM.DOUT10	27		27	$\overline{\text{MM.ADD8}}$
26	N/C	26	GND	26		26	$\overline{\text{MM.ADD0}}$
25	N/C	25	GND	25		25	$\overline{\text{MM.ADD4}}$
24	MM.DOUT13	24	MM.DOUT12	24		24	$\overline{\text{MM.ADD9}}$
23	MM.DOUT15	23	MM.DOUT14	23		23	$\overline{\text{MM.ADD1}}$
22	MM.DOUT17	22	MM.DOUT16	22		22	$\overline{\text{MM.ADD10}}$
21	MM.DOUT19	21	MM.DOUT18	21		21	$\overline{\text{MM.ADD11}}$
20	MM.DOUT21	20	MM.DOUT20	20		20	$\overline{\text{MM.WRITE}}$
19	MM.DOUT23	19	MM.DOUT22	19		19	
18	N/C	18	GND	18		18	
17	N/C	17	GND	17		17	
16	MM.DOUT25	16	MM.DOUT24	16		16	
15	MM.DOUT27	15	MM.DOUT26	15		15	
14	MM.DOUT29	14	MM.DOUT28	14		14	
13	MM.DOUT31	13	MM.DOUT30	13		13	
12	MM.DOUT33	12	MM.DOUT32	12		12	
11	Spare	11	MM.DOUT34	11		11	
10	N/C	10	GND	10		10	
9	N/C	9	GND	9		9	
8		8		8		8	
7		7		7		7	
6		6		6		6	
5		5		5		5	
4		4		4		4	
3		3		3		3	
2		2		2		2	
1		1		1		1	


2102A-1
2102A-1
2102A-1
2102A-1



CENTRAL PROCESSING UNIT (CPU) CARD CAGE: A SLOT: 3 BOARD: 3

BOARD NAME : C.P.U. CARD CAGE : A SLOT : 3 BOARD : 3

CONNECTOR J1

PIN	SIGNAL	PIN	SIGNAL
1	SEEKRDY	2	
3	SEEKERR	4	
5	SECTOR	6	
7	INDEX	8	
9	Polarising Pin	10	Polarising Pin
11	ONLINE	12	
13	UNSAFE	14	
15	Spare	16	
17	WRITE.CUR	18	
19	CAR128	20	
21	CAR64	22	
23	CAR32	24	
25	CAR16	26	
27	CAR8	28	
29	CAR4	30	
31	CAR2	32	
33	CAR1	34	
35		36	
37		38	
39		40	

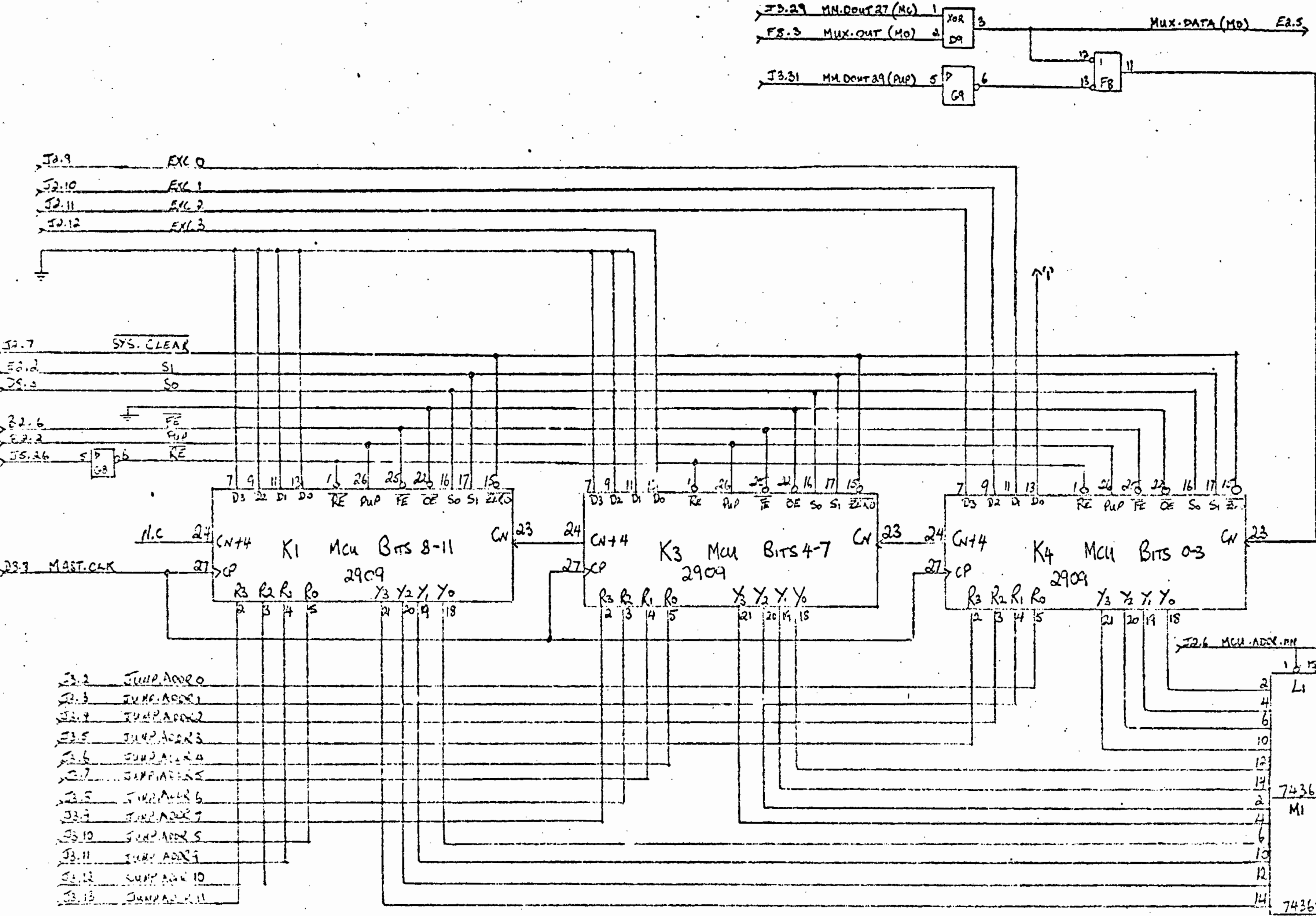
CONNECTOR J2

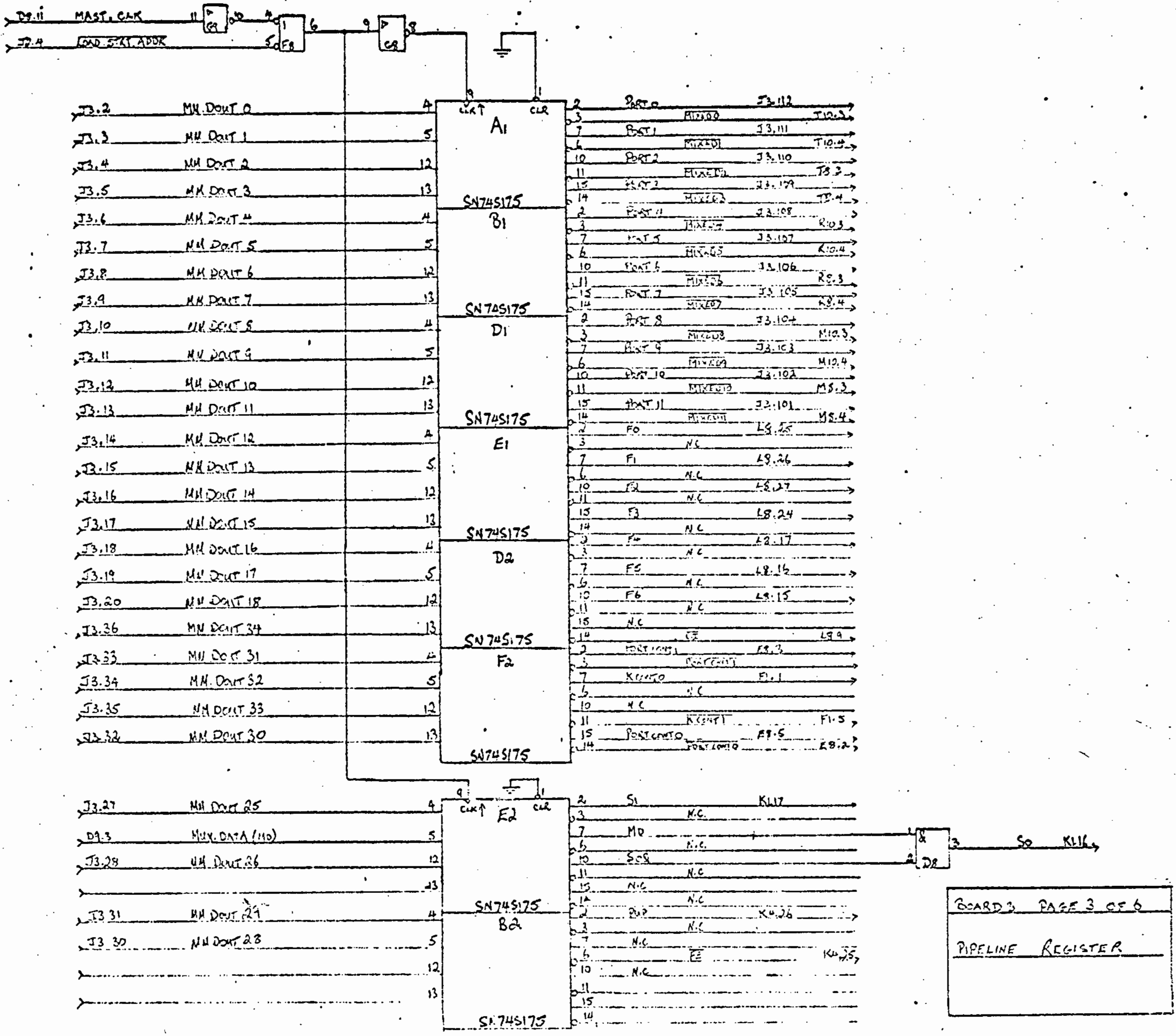
PIN	SIGNAL	PIN	SIGNAL
1	EN.A	2	EN.B
3	EN.C	4	LOAD.STRT.ADDR
5	RESET	6	MCU.ADDR.EN
7	SYS.CLEAR	8	MM.WRITE
9	EXC1	10	EXC2
11	EXC3	12	EXC4
13	EXC	14	DTOX
15	DTIX	16	MM.EN
17	DMA.FIN	18	Spare
19	COMMAND	20	SYNC
21	EQ.16	22	SGLDEN
23	UNTSEL	24	ATTEN
25	FORMAT	26	PORT.CLK
27	Spare	28	Spare
29	Spare	30	Spare
31	Spare	32	PORT.CLK
33	Spare	34	Spare
35	Polarising Pin	36	Polarising Pin
37	Spare	38	Spare
39	Spare	40	Spare
41	Spare	42	Spare
43	Spare	44	Spare

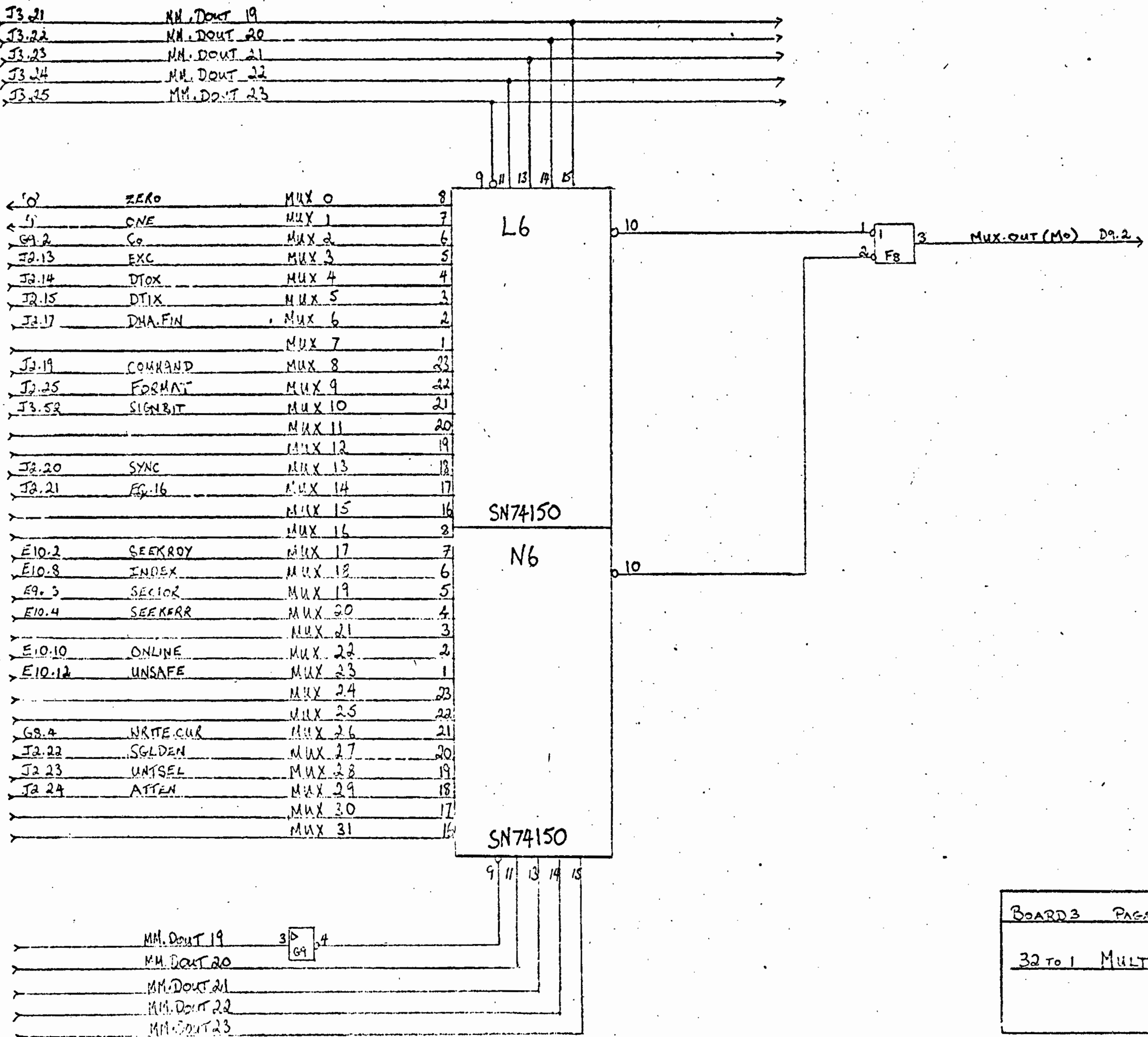
BOARD NAME : CPU CARD CAGE:A SLOT:3 BOARD:3

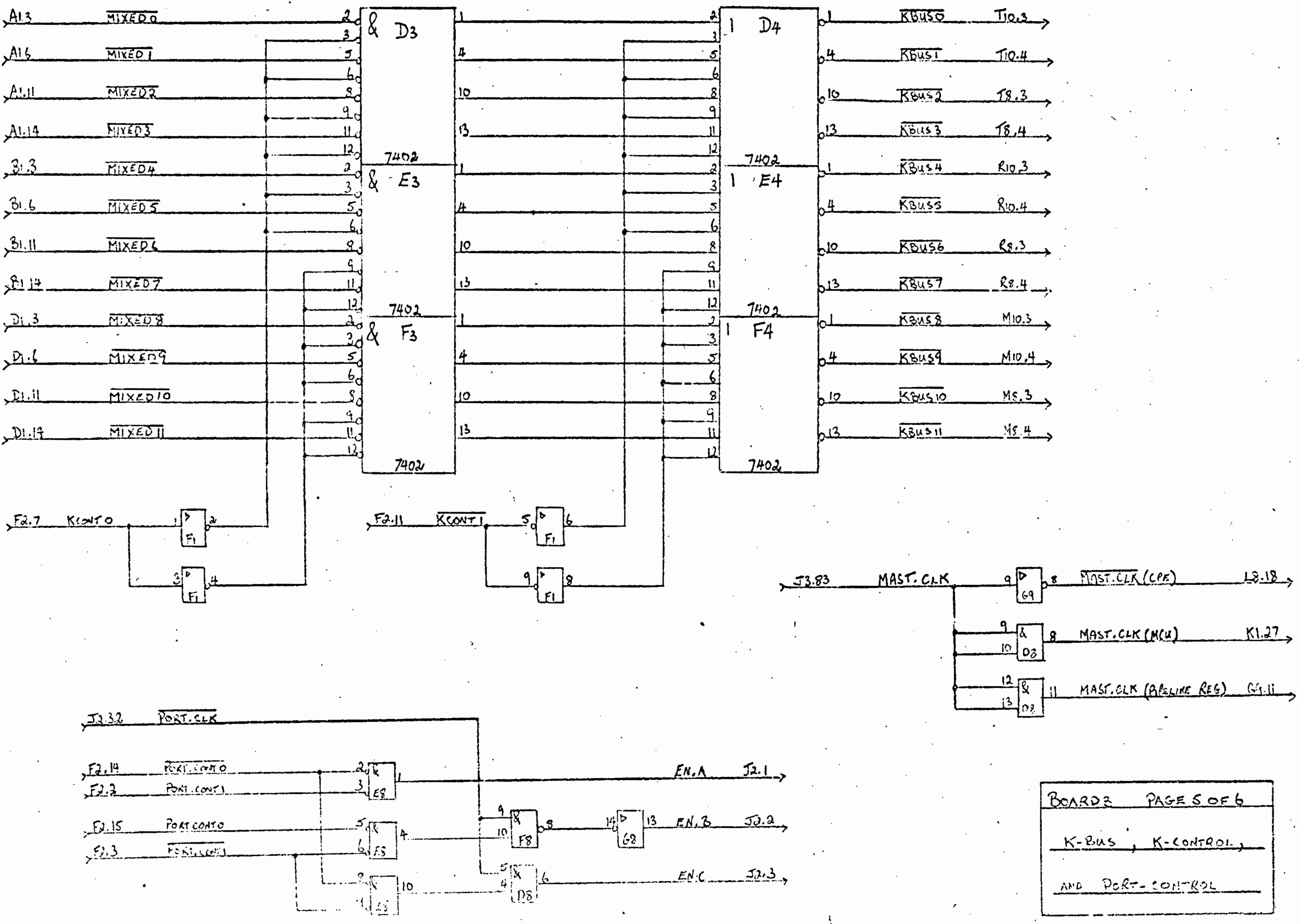
CONNECTOR J3

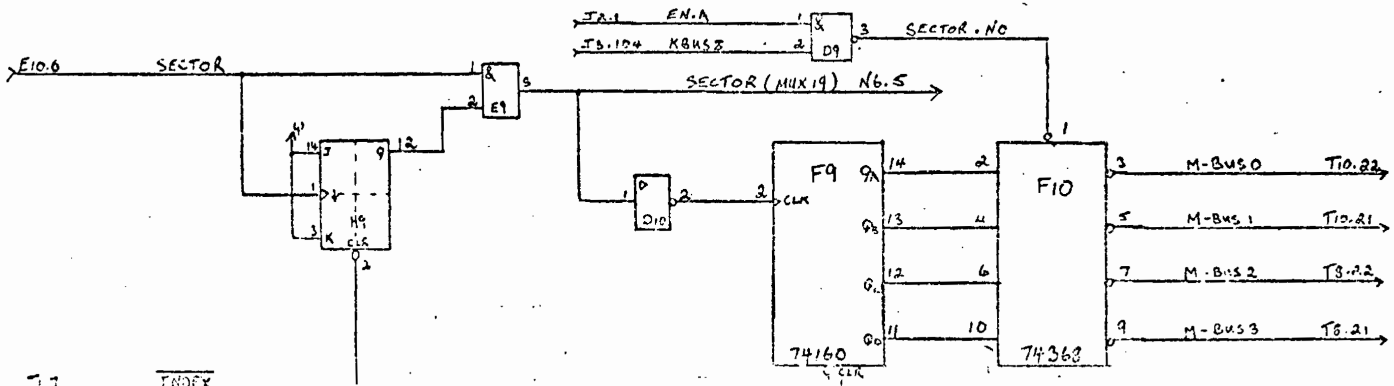
PIN	SIGNAL	PIN	SIGNAL	PIN	SIGNAL	PIN	SIGNAL
1	GND	2	MM.DOUT0	63	DBUS6	64	DBUS5
3	MM.DOUT1	4	MM.DOUT2	65	$\overline{\text{DBUS4}}$	66	$\overline{\text{DBUS3}}$
5	MM.DOUT3	6	MM.DOUT4	67	$\overline{\text{DBUS2}}$	68	$\overline{\text{DBUS1}}$
7	MM.DOUT5	8	MM.DOUT6	69	$\overline{\text{DBUS0}}$	70	$\overline{\text{MM.ADD0}}$
9	MM.DOUT7	10	MM.DOUT8	71	$\overline{\text{MM.ADD1}}$	72	$\overline{\text{MM.ADD2}}$
11	MM.DOUT9	11	MM.DOUT10	73	$\overline{\text{MM.ADD3}}$	74	$\overline{\text{MM.ADD4}}$
13	MM.DOUT11	14	MM.DOUT12	75	$\overline{\text{MM.ADD5}}$	76	$\overline{\text{MM.ADD6}}$
15	MM.DOUT13	16	MM.DOUT14	77	$\overline{\text{MM.ADD7}}$	78	$\overline{\text{MM.ADD8}}$
17	MM.DOUT15	18	MM.DOUT16	79	$\overline{\text{MM.ADD9}}$	80	$\overline{\text{MM.ADD10}}$
19	MM.DOUT17	20	MM.DOUT18	81	$\overline{\text{MM.ADD11}}$	82	$\overline{\text{MM.WRITE}}$
21	MM.DOUT19	22	MM.DOUT20	83	MAST.CLK	84	$\overline{\text{MBUS15}}$
23	MM.DOUT21	24	MM.DOUT22	85	$\overline{\text{MBUS14}}$	86	$\overline{\text{MBUS13}}$
25	MM.DOUT23	26	MM.DOUT24	87	$\overline{\text{MBUS12}}$	88	$\overline{\text{MBUS11}}$
27	MM.DOUT25	28	MM.DOUT26	89	$\overline{\text{MBUS10}}$	90	$\overline{\text{MBUS9}}$
29	MM.DOUT27	30	MM.DOUT28	91	$\overline{\text{MBUS8}}$	92	$\overline{\text{MBUS7}}$
31	MM.DOUT29	32	MM.DOUT30	93	$\overline{\text{MBUS6}}$	94	$\overline{\text{MBUS5}}$
33	MM.DOUT31	34	MM.DOUT32	95	$\overline{\text{MBUS4}}$	96	$\overline{\text{MBUS3}}$
35	MM.DOUT33	36	MM.DOUT34	97	$\overline{\text{MBUS2}}$	98	$\overline{\text{MBUS1}}$
37	Spare	38	Spare	99	$\overline{\text{MBUS0}}$	100	GND
39	$\overline{\text{ABUS11}}$	40	$\overline{\text{ABUS10}}$	101	KBUS11	102	KBUS10
41	$\overline{\text{ABUS9}}$	42	$\overline{\text{ABUS8}}$	103	KBUS9	104	KBUS8
43	$\overline{\text{ABUS7}}$	44	$\overline{\text{ABUS6}}$	105	KBUS7	106	KBUS6
45	$\overline{\text{ABUS5}}$	46	$\overline{\text{ABUS4}}$	107	KBUS5	108	KBUS4
47	$\overline{\text{ABUS3}}$	48	GND	109	KBUS3	110	KBUS2
49	$\overline{\text{ABUS2}}$	50	$\overline{\text{ABUS1}}$	111	KBUS1	112	KBUS0
51	$\overline{\text{ABUS0}}$	52	$\overline{\text{DBUS15}}$	113	$\overline{\text{ABUS15}}$	114	$\overline{\text{ABUS14}}$
53	$\overline{\text{DBUS14}}$	54	$\overline{\text{DBUS13}}$	115	$\overline{\text{ABUS13}}$	116	$\overline{\text{ABUS12}}$
55	$\overline{\text{DBUS12}}$	56	$\overline{\text{DBUS11}}$	117	Spare	118	+5VOLTS
57	$\overline{\text{DBUS10}}$	58	$\overline{\text{DBUS9}}$	119	Spare	120	+5VOLTS
59	$\overline{\text{DBUS8}}$	60	$\overline{\text{DBUS7}}$	121	+5VOLTS	122	GND
61	UNUSED	62	UNUSED				


MUX. DATA (MO)
FOR MCU: 14 = GND
23 = Vcc
6,8,10,12 = GND
SYS. CLEAR
MAST. CLK
K1 MCU BITS 8-11
2909
K3 MCU BITS 4-7
2909
K4 MCU BITS 0-3
2909
74368
BOARD 3 PAGE 2 OF 6
MICROPROGRAM


MAST. CLK
LOAD START ADDR
SN74S175
BOARD 3 PAGE 3 OF 6
PIPELINE REGISTER


J3.21 MM.DOUT 19
J3.22 MM.DOUT 20
J3.23 MM.DOUT 21
J3.24 MM.DOUT 22
J3.25 MM.DOUT 23
'0' ZERO MUX 0
'1' ONE MUX 1
G9.2 C0 MUX 2
J2.13 EXC MUX 3
J2.14 DTOX MUX 4
J2.15 DTIX MUX 5
J2.17 DMA.FIN MUX 6
MUX 7
J2.19 COMMAND MUX 8
J2.25 FORMAT MUX 9
J3.52 SIGNBIT MUX 10
MUX 11
MUX 12
J2.20 SYNC MUX 13
J2.21 FG.16 MUX 14
MUX 15
MUX 16
E10.2 SEEKRDY MUX 17
E10.8 INDEX MUX 18
E9.3 SECTOR MUX 19
E10.4 SEEKERR MUX 20
MUX 21
E10.10 ONLINE MUX 22
E10.12 UNSAFE MUX 23
MUX 24
MUX 25
G8.4 WRITE.CUR MUX 26
J2.22 SGLDEN MUX 27
J2.23 UNITSEL MUX 28
J2.24 ATTEN MUX 29
MUX 30
MUX 31
L6
SN74150
N6
SN74150
F8
MUX.OUT (M0) D9.2
MM.DOUT 19
MM.DOUT 20
MM.DOUT 21
MM.DOUT 22
MM.DOUT 23
G9
BOARD 3 PAGE 4 OF 6
32 TO 1 MULTIPLEXOR


D3
E3
F3
D4
E4
F4
7402
MIXED0
MIXED1
MIXED2
MIXED3
MIXED4
MIXED5
MIXED6
MIXED7
MIXED8
MIXED9
MIXED10
MIXED11
KBUS0
KBUS1
KBUS2
KBUS3
KBUS4
KBUS5
KBUS6
KBUS7
KBUS8
KBUS9
KBUS10
KBUS11
KCONT0
KCONT1
MAST.CLK
MAST.CLK (CPE)
MAST.CLK (MCU)
MAST.CLK (PIPELINE REG)
PORT.CLK
EN.A
EN.B
EN.C
BOARD 3 PAGE 5 OF 6
K-BUS, K-CONTROL,
AND PORT-CONTROL



Vcc
E11 G11
E10 7404
G8 7404

J1.1 SEEKRDY — SEEKRDY (MUX 17) N6.7
J1.3 SEEKERR — SEEKERR (MUX 20) N6.4
J1.5 SECTOR — SECTOR H9.1
J1.7 INDEX — INDEX (MUX 18) N6.5
J1.11 ONLINE — ONLINE (MUX 22) N6.2
J1.13 UNSAFE — UNSAFE (MUX 23) N6.1
J1.15 Spare — N/C
J1.17 WRITE.CUR — WRITE.CUR (MUX 26) N6.21
J1.19 CAR 128 — I-Bus R8.2
J1.21 CAR 64 — I-Bus R8.1
J1.23 CAR 32 — I-Bus R10.2
J1.25 CAR 16 — I-Bus R10.1
J1.27 CAR 8 — I-Bus T3.2
J1.29 CAR 4 — I-Bus T8.1
J1.31 CAR 2 — I-Bus T10.2
J1.33 CAR 1 — I-Bus T10.1
E10.6 SECTOR
INDEX

BOARD 3 PAGE 6 OF 6

SIGNALS FROM DISC

J3 CONNECTOR

	1	2	3	4	5	6	7	8	9	10	11		
A						A6 7416		7476	7476	SWITCH	74368	A	
B								7404	7432		74368	B	J1 CONNECTOR
D						D6 7416		7408	7400		74368	D	J1 CONNECTOR
E						E6 7416		74175	74175		74368	E	J1 CONNECTOR
F								74175	74175	7474	74368	F	J1 CONNECTOR
G						G6 7416		74138	74148	7474	74368	G	J1 CONNECTOR
H	7432	7473	7473	7404	7408	H6 7416		COMPONENTS	COMPONENTS	COMPONENTS		H	
K	7432	7473	74195	74175	COMPONENTS			7438	7438	7438	7438	K	
L	7432	7473	74195	74175	7473	L6		74132	74132	74132		L	J2 CONNECTOR
M	74368	7400	74195	74175	7473			74132	74132	74132	7438	M	J2 CONNECTOR
N	74368	7400	74195	74175	7473	N6		J4 CONNECTOR				N	J2 CONNECTOR
R	74368	7400	74368	74161	7400	7438	7438					R	J2 CONNECTOR
S	7400	7400	74368	7473	7474	7438	7438	J5 CONNECTOR				S	J2 CONNECTOR
T	7400	7400	74368	7474	7474	7438	COMPONENTS					T	
	1	2	3	4	5	6	7	8	9	10	11		

COMPUTER AND DISC DRIVE INTERFACE

CARD CAGE: A SLOT: 4 BOARD: 4

BOARD NAME : COMPUTER AND DISC DRIVE INTERFACES   CARD CAGE:A   SLOT:4   BOARD:4

CONNECTOR J1

PIN	SIGNAL	PIN	SIGNAL
1	MM.DIN21	2	MM.DIN0
3	MM.DIN22	4	MM.DIN1
5	MM.DIN23	6	MM.DIN2
7	MM.DIN24	8	MM.DIN3
9	Polarising Pin	10	Polarising Pin
11	MM.DIN25	12	MM.DIN4
13	MM.DIN26	14	MM.DIN5
15	MM.DIN27	16	MM.DIN6
17	MM.DIN28	18	MM.DIN7
19	MM.DIN29	20	MM.DIN8
21	MM.DIN30	22	MM.DIN9
23	MM.DIN31	24	MM.DIN10
25	MM.DIN32	26	MM.DIN11
27	MM.DIN33	28	MM.DIN12
29	MM.DIN34	30	MM.DIN13
31	Spare	32	MM.DIN14
33	Spare	34	MM.DIN15
35	Spare	36	MM.DIN16
37	Spare	38	MM.DIN17
39	Spare	40	MM.DIN18
41	Spare	42	MM.DIN19
43	Spare	44	MM.DIN20

CONNECTOR J2

PIN	SIGNAL	PIN	SIGNAL
1	EN.A	2	EN.B
3	EN.C	4	LOAD.STRT.ADDR
5	RESET	6	MCU.ADDR.EN
7	SYS.CLEAR	8	MM.WRITE
9	EXC1	10	EXC2
11	EXC3	12	EXC4
13	EXC	14	DTOX
15	DTIX	16	MM.EN
17	DMA.FIN	18	Spare
19	COMMAND	20	SYNC
21	EQ.16	22	SGLDEN
23	UNTSEL	24	ATTEN
25	FORMAT	26	PORT.CLK
27	Spare	28	Spare
29	Spare	30	Spare
21	Spare	32	PORT.CLK
33	Spare	34	Spare
35	Polarising Pin	36	Polarising Pin
37	Spare	38	Spare
39	Spare	40	Spare
41	Spare	42	Spare
43	Spare	44	Spare

BOARD NAME : COMPUTER AND DISC DRIVE INTERFACES   CARD CAGE:A   SLOT:4   BOARD:4

CONNECTOR J3

PIN	SIGNAL	PIN	SIGNAL	PIN	SIGNAL	PIN	SIGNAL
1	GND	2	MM.DOUT0	63	DBUS6	64	DBUS5
3	MM.DOUT1	4	MM.DOUT2	65	DBUS4	66	DBUS3
5	MM.DOUT3	6	MM.DOUT4	67	DBUS2	68	DBUS1
7	MM.DOUT5	8	MM.DOUT6	69	DBUS0	70	MM.ADD0
9	MM.DOUT7	10	MM.DOUT8	71	MM.ADD1	72	MM.ADD2
11	MM.DOUT9	11	MM.DOUT10	73	MM.ADD3	74	MM.ADD4
13	MM.DOUT11	14	MM.DOUT12	75	MM.ADD5	76	MM.ADD6
15	MM.DOUT13	16	MM.DOUT14	77	MM.ADD7	78	MM.ADD8
17	MM.DOUT15	18	MM.DOUT16	79	MM.ADD9	80	MM.ADD10
19	MM.DOUT17	20	MM.DOUT18	81	MM.ADD11	82	MM.WRITE
21	MM.DOUT19	22	MM.DOUT20	83	MAST.CLK	84	MBUS15
23	MM.DOUT21	24	MM.DOUT22	85	MBUS14	86	MBUS13
25	MM.DOUT23	26	MM.DOUT24	87	MBUS12	88	MBUS11
27	MM.DOUT25	28	MM.DOUT26	89	MBUS10	90	MBUS9
29	MM.DOUT27	30	MM.DOUT28	91	MBUS8	92	MBUS7
31	MM.DOUT29	32	MM.DOUT30	93	MBUS6	94	MBUS5
33	MM.DOUT31	34	MM.DOUT32	95	MBUS4	96	MBUS3
35	MM.DOUT33	36	MM.DOUT34	97	MBUS2	98	MBUS1
37	Spare	38	Spare	99	MBUS0	100	GND
39	ABUS11	40	ABUS10	101	KBUS11	102	KBUS10
41	ABUS9	42	ABUS8	103	KBUS9	104	KBUS8
43	ABUS7	44	ABUS6	105	KBUS7	106	KBUS6
45	ABUS5	46	ABUS4	107	KBUS5	108	KBUS4
47	ABUS3	48	GND	109	KBUS3	110	KBUS2
49	ABUS2	50	ABUS1	111	KBUS1	112	KBUS0
51	ABUS0	52	DBUS15	113	ABUS15	114	ABUS14
53	DBUS14	54	DBUS13	115	ABUS13	116	ABUS12
55	DBUS12	56	DBUS11	117	Spare	118	+5VOLTS
57	DBUS10	58	DBUS9	119	Spare	120	+5VOLTS
59	DBUS8	60	DBUS7	121	+5VOLTS	122	GND
61	UNUSED	62	UNUSED				

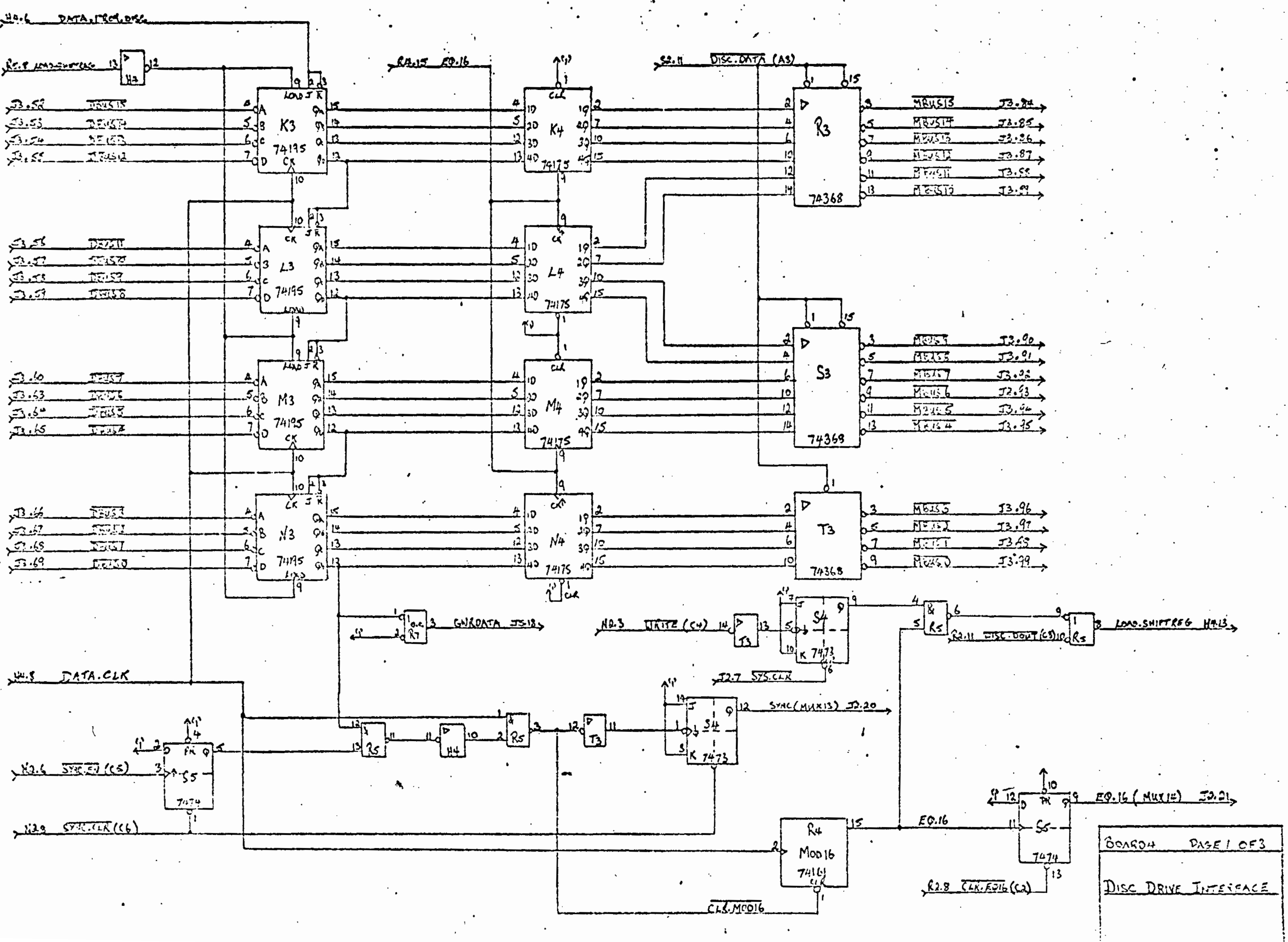
BOARD NAME : COMPUTER AND DISC DRIVE INTERFACES   CARD CAGE:A   SLOT:4   BOARD:4

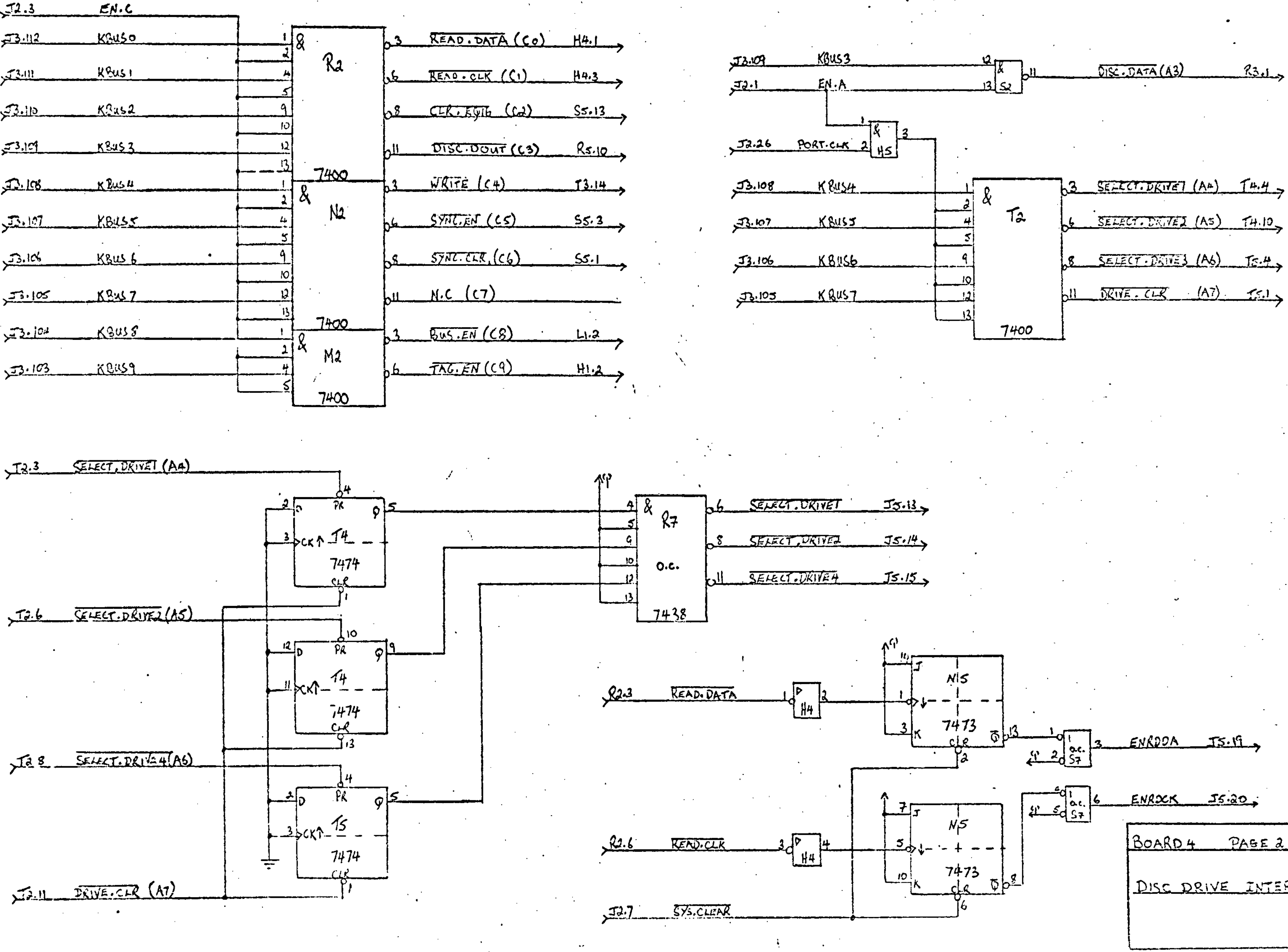
CONNECTOR J4

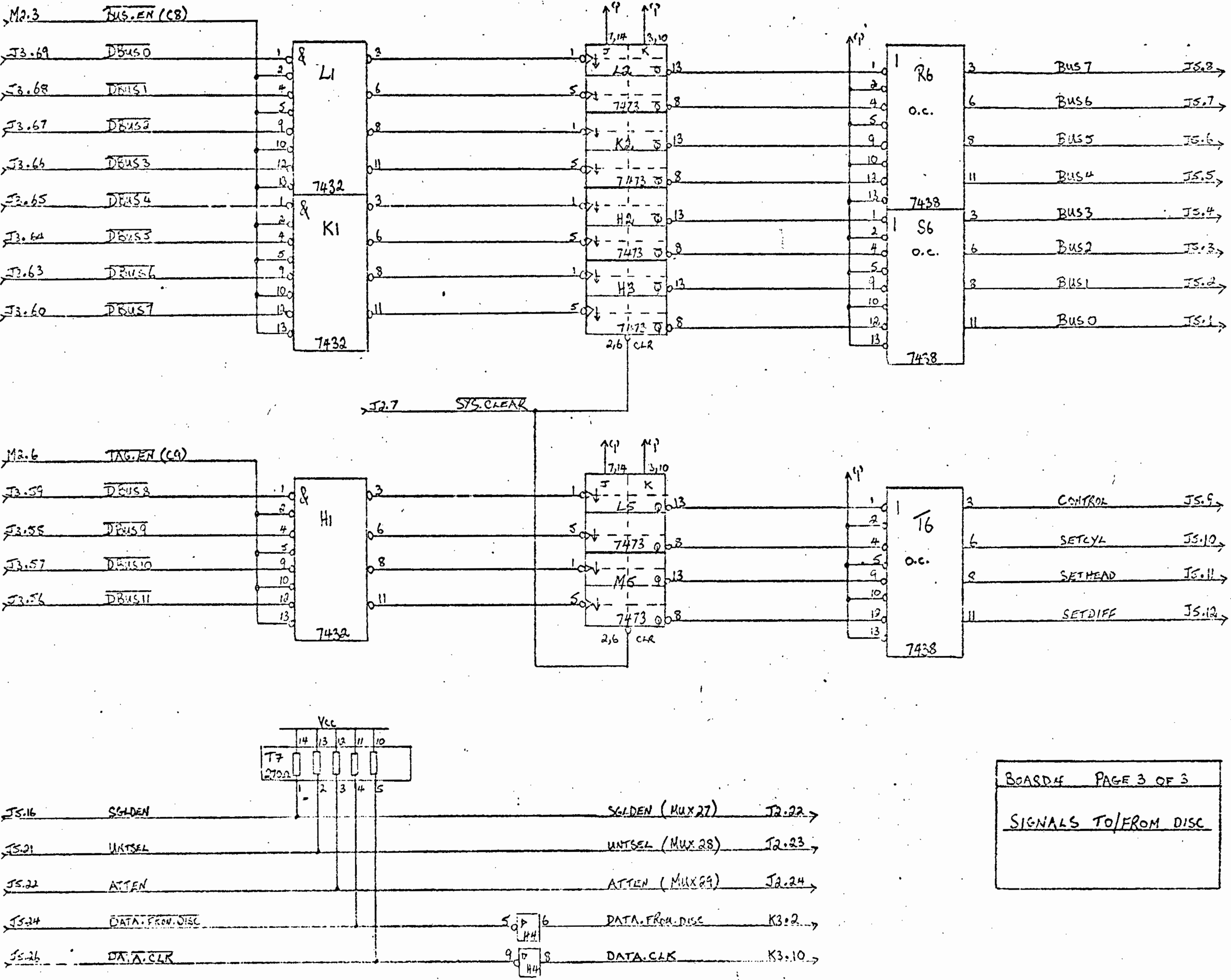
PIN	SIGNAL	PIN	SIGNAL
1	EBUS0	1	
2	EBUS1	2	
3	EBUS2	3	
4	EBUS3	4	
5	EBUS4	5	
6	EBUS5	6	
7	EBUS6	7	
8	EBUS7	8	
9	EBUS8	9	
10	EBUS9	10	
11	EBUS10	11	
12	EBUS11	12	
13	EBUS12	13	
14	EBUS13	14	
15	EBUS14	15	
16	EBUS15	16	GROUND
17	EXC0	17	
18	EXC1	18	
19	EXC2	19	
20	EXC3	20	
21	VAR- CONT0	21	
22	VAR-CONT1	22	
23	VAR-CONT2	23	
24	CONT→VAR	24	
25		25	
26			
27	CONT-VAR0	27	
28	CONT-VAR1	28	
29	CONT-VAR2	29	
30	INTERRUPT	30	
31		31	
32		32	

CONNECTOR J5

PIN	SIGNAL	PIN	SIGNAL
1	BUS0	1	
2	BUS1	2	
3	BUS2	3	
4	BUS3	4	
5	BUS4	5	
6	BUS5	6	
7	BUS6	7	
8	BUS7	8	
9	CONTROL	9	
10	SETCYL	10	
11	SETHEAD	11	
12	SETDIFF	12	
13	SEL.DRIVE1	13	
14	SEL.DRIVE2	14	
15	SEL.DRIVE4	15	
16	SGLDEN	16	
17	GND	17	
18	GWRDATA	18	
19	ENRDDA	19	
20	ENRDCK	20	
21	UNTSEL	21	
22	ATTEN	22	
23	GND	23	
24	DATA.FROM. DISC	24	
25	GND		
26	DATA.CLK	26	
27		27	
28		28	
29		29	
30		30	
31		31	
32		32	


BOARD 4
PAGE 1 OF 3
DISC DRIVE INTERFACE


J2.3 EN.C
J3.112 KBUS0
J3.111 KBUS1
J3.110 KBUS2
J3.109 KBUS3
J3.108 KBUS4
J3.107 KBUS5
J3.106 KBUS6
J3.105 KBUS7
J3.104 KBUS8
J3.103 KBUS9
R2
N2
M2
7400
READ.DATA (C0) H4.1
READ.CLK (C1) H4.3
CLK.ENB (C2) S5.13
DISC.DOUT (C3) R5.10
WRITE (C4) T3.14
SYNC.EN (C5) S5.3
SYNC.CLR (C6) S5.1
N.C (C7)
BUS.EN (C8) L1.2
TAG.EN (C9) H1.2
J3.109 KBUS3
J2.1 EN.A
J2.26 PORT.CLK
DISC.DATA (A3) R3.1
T2
SELECT.DRIVE1 (A4) T4.4
SELECT.DRIVE2 (A5) T4.10
SELECT.DRIVE3 (A6) T5.4
DRIVE.CLR (A7) T5.1
J2.3 SELECT.DRIVE1 (A4)
J2.6 SELECT.DRIVE2 (A5)
J2.8 SELECT.DRIVE4 (A6)
J2.11 DRIVE.CLR (A7)
T4
T5
7474
R7
O.C.
7438
SELECT.DRIVE1 J5.13
SELECT.DRIVE2 J5.14
SELECT.DRIVE4 J5.15
R2.3 READ.DATA
R2.6 READ.CLK
J2.7 SYS.CLEAR
H4
N5
7473
S7
ENRDDA J5.19
ENRDCK J5.20
BOARD 4 PAGE 2 OF 3
DISC DRIVE INTERFACE


BOARD 4 PAGE 3 OF 3
SIGNALS TO/FROM DISC
M2.3 BUS.EN (C8)
J3.69 DBUS0
J3.68 DBUS1
J3.67 DBUS2
J3.66 DBUS3
J3.65 DBUS4
J3.64 DBUS5
J3.63 DBUS6
J3.60 DBUS7
L1
K1
7432
L2
K2
H2
H3
7473
CLR
R6
S6
o.c.
7438
BUS7 J5.8
BUS6 J5.7
BUS5 J5.6
BUS4 J5.5
BUS3 J5.4
BUS2 J5.3
BUS1 J5.2
BUS0 J5.1
J2.7 SYS.CLEAR
M2.6 TAG.EN (C9)
J3.59 DBUS8
J3.58 DBUS9
J3.57 DBUS10
J3.56 DBUS11
H1
L5
M6
T6
CONTROL J5.9
SETCYL J5.10
SETHEAD J5.11
SETDIFF J5.12
Vcc
T7
J5.16 SELDEN
J5.21 UNTSEL
J5.22 ATTEN
J5.24 DATA.FROM.DISC
J5.26 DATA.CLK
H4
SELDEN (MUX27) J2.22
UNTSEL (MUX28) J2.23
ATTEN (MUX69) J2.24
DATA.FROM.DISC K3.2
DATA.CLK K3.10

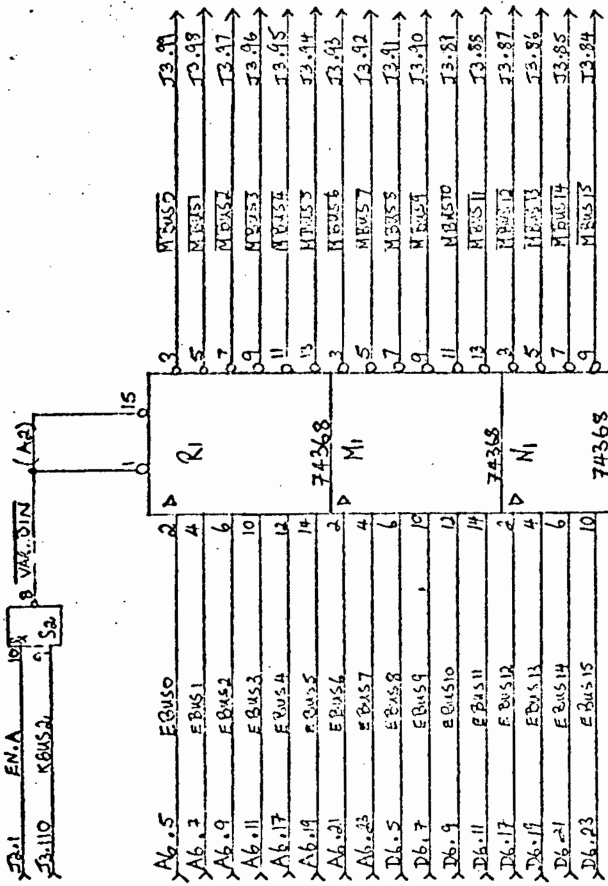
BOARD 4 PAGE 1 OF 6

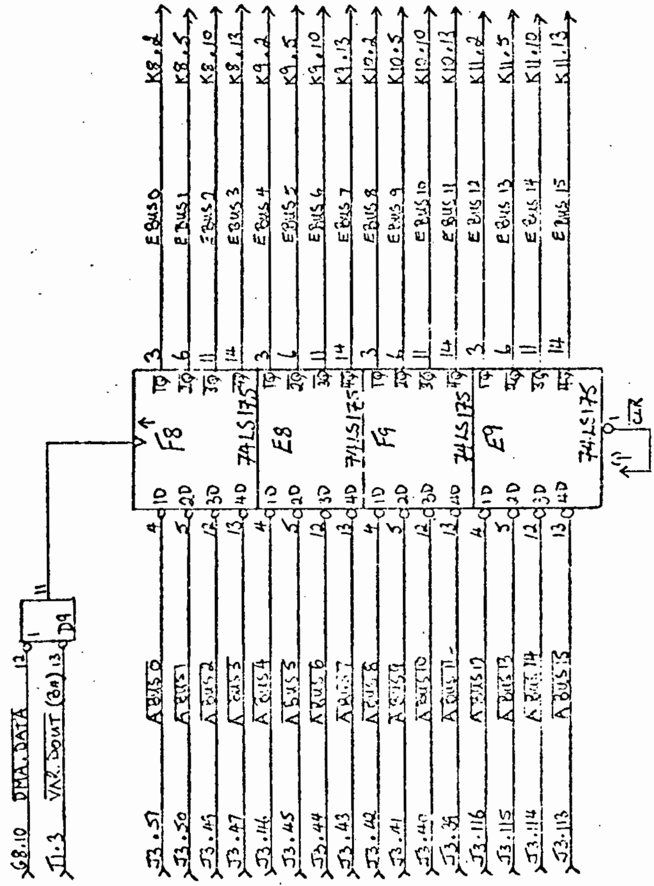
COMPUTER INTERFACE:

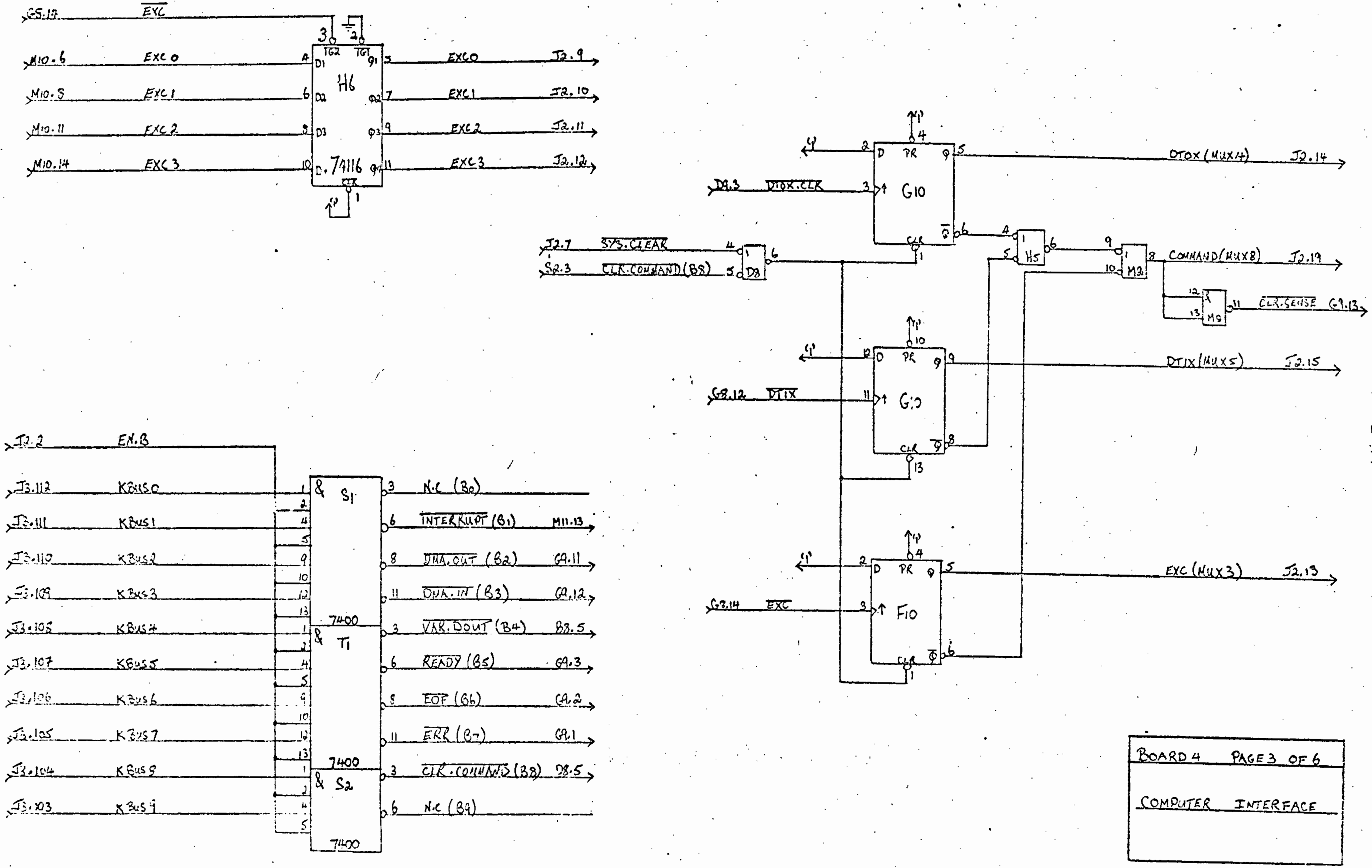
DATA TO MICROMEMORY

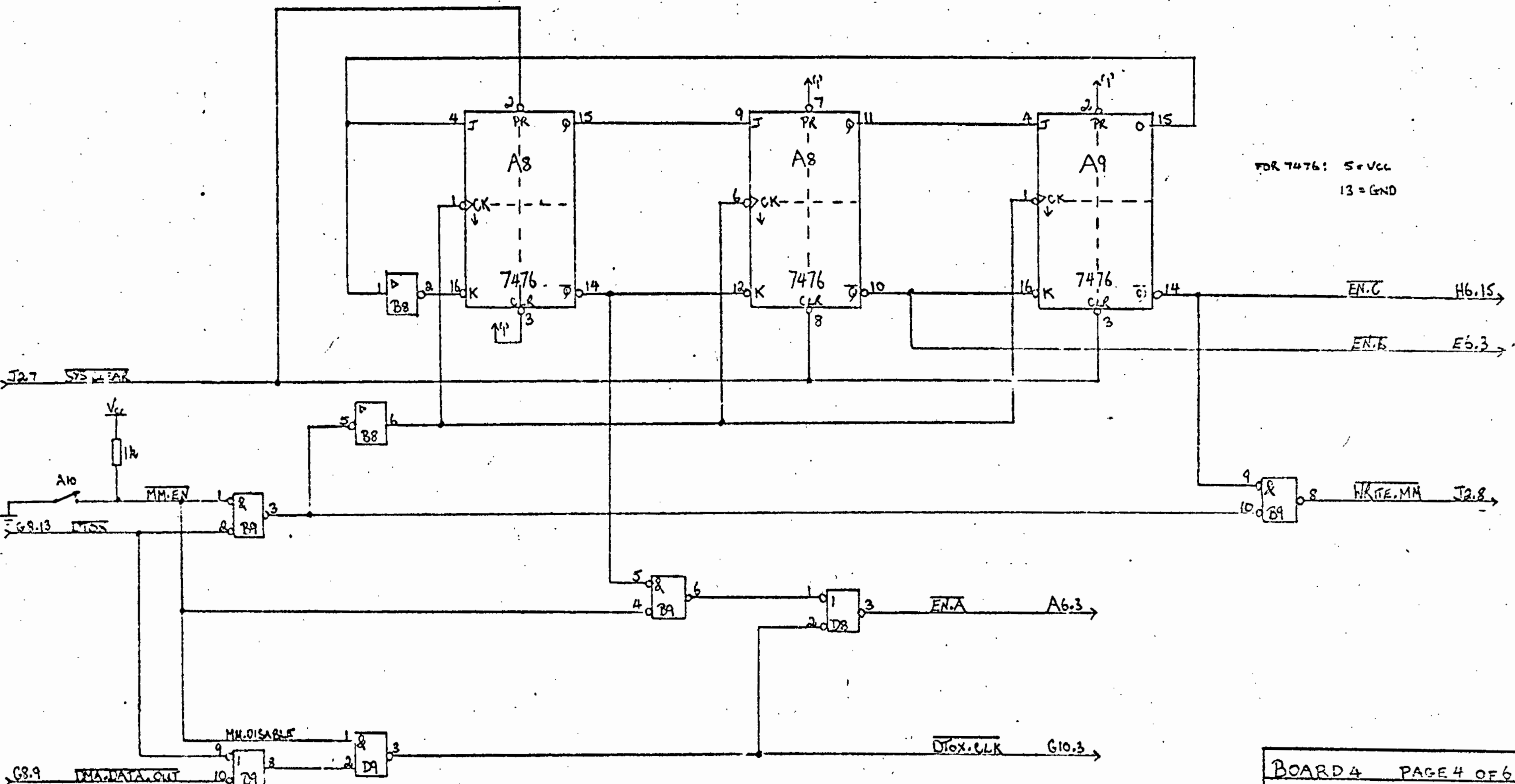
BOARD 4 PAGE 2 OF 6

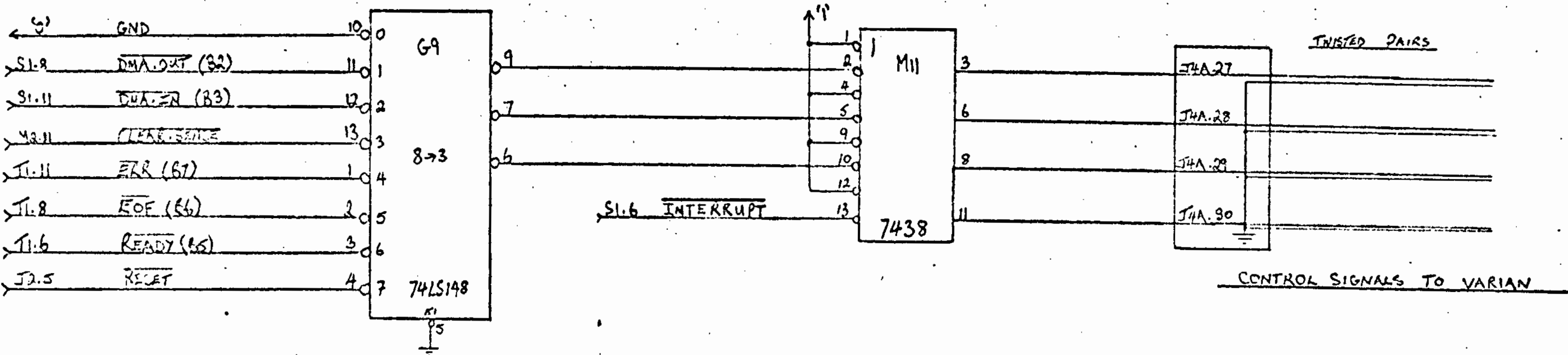
COMPUTER INTERFACE

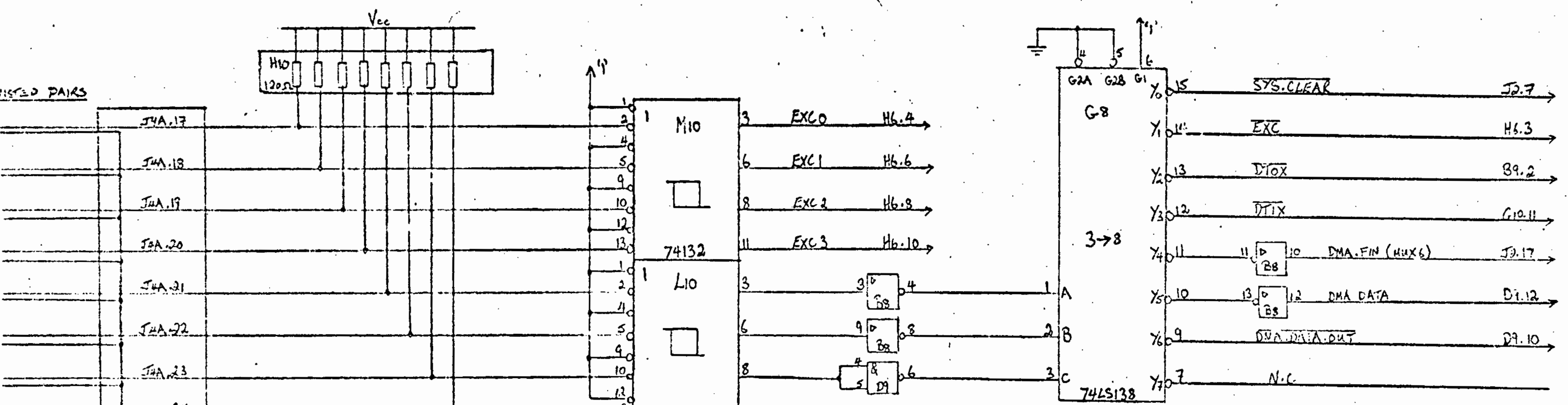


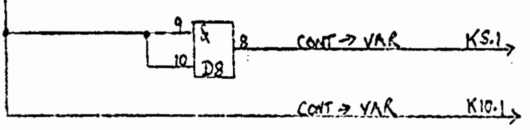


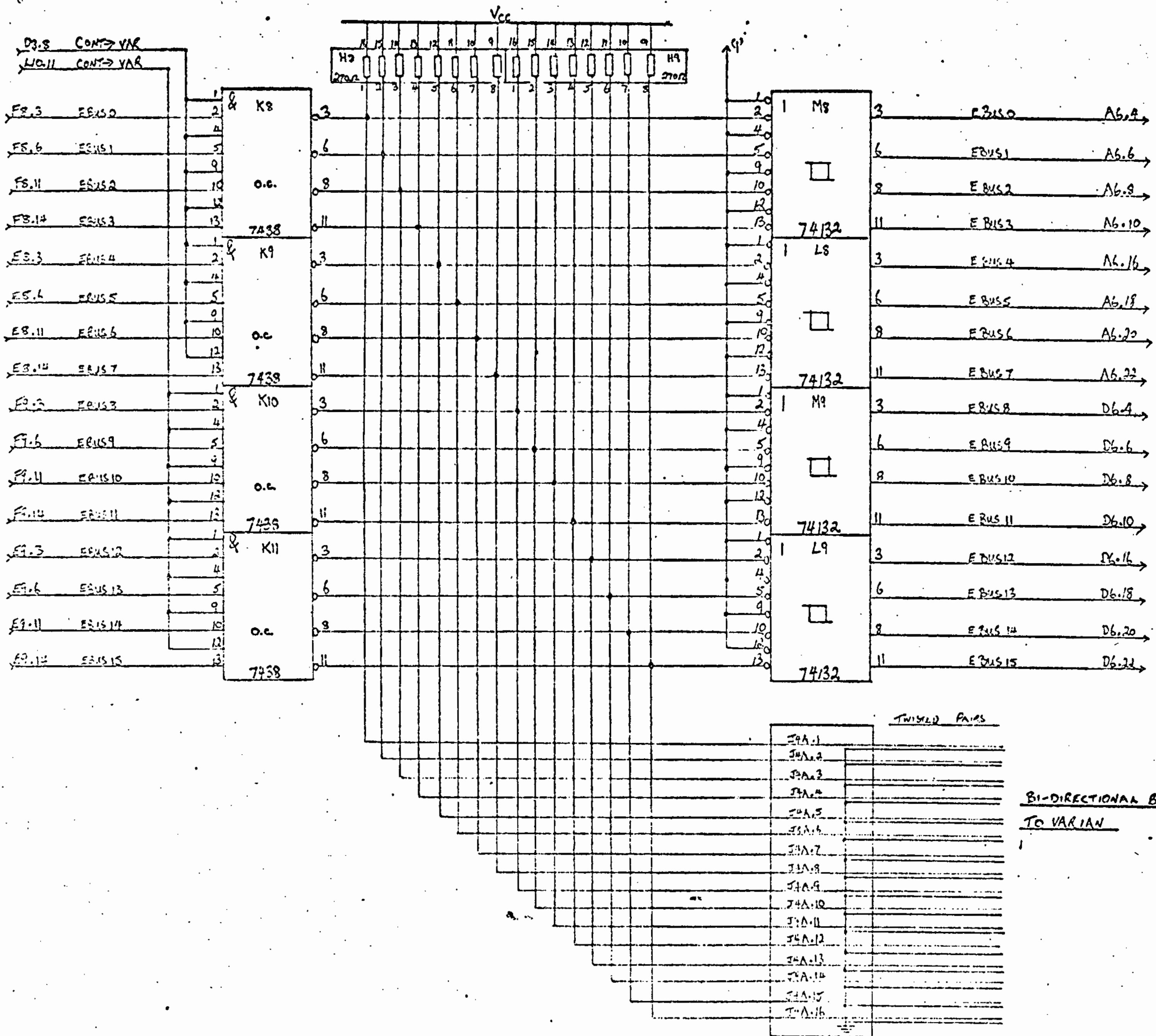

G5.14
EXC
M10.6
EXC 0
M10.8
EXC 1
M10.11
EXC 2
M10.14
EXC 3
H6
74116
EXC0
J2.9
EXC1
J2.10
EXC2
J2.11
EXC3
J2.12
D9.3
DTOX.CLK
G10
DTOX (MUX4)
J2.14
J2.7
SYS.CLEAR
S2.3
CLR.COMMAND (B8)
D8
H5
M2
COMMAND (MUX8)
J2.19
M9
CLR.SENSE
G9.13
G8.12
DTIX
DTIX (MUX5)
J2.15
G8.14
EXC
F10
EXC (MUX3)
J2.13
J2.2
EN.B
J3.112
KBUS0
J3.111
KBUS1
J3.110
KBUS2
J3.109
KBUS3
J3.108
KBUS4
J3.107
KBUS5
J3.106
KBUS6
J3.105
KBUS7
J3.104
KBUS8
J3.103
KBUS9
S1
T1
S2
7400
N.C (B0)
INTERRUPT (B1)
M11.13
DMA.OUT (B2)
G9.11
DMA.IN (B3)
G9.12
VAR.DOUT (B4)
B8.5
READY (B5)
G9.3
EOF (B6)
G9.2
ERR (B7)
G9.1
CLR.COMMAND (B8)
D8.5
N.C (B9)
BOARD 4
PAGE 3 OF 6
COMPUTER INTERFACE


BOARD 4 PAGE 4 OF 6
COMPUTER INTERFACE:
RING COUNTER
FOR 7476: 5 = VCC
13 = GND
A8
A8
A9
7476
EN.C
H6.15
EN.B
E5.3
WRITE.MM
J2.8
EN.A
A6.3
DTOX.CLK
G10.3
MM.EN
MM.DISABLE
DMA.DATA.OUT
G3.9
G3.13
J2.7
A10
Vcc
1k


GND
S1.8
DMA.OUT (B2)
S1.11
DMA.IN (B3)
T1.11
ERR (B7)
T1.8
EOF (B6)
T1.6
READY (B5)
J2.5
RESET
G9
8→3
74LS148
S1.6
INTERRUPT
M11
7438
TWISTED PAIRS
J4A.27
J4A.28
J4A.29
J4A.30
CONTROL SIGNALS TO VARIAN


TWISTED PAIRS
J4A.17
J4A.18
J4A.19
J4A.20
J4A.21
J4A.22
J4A.23
J4A.24
Vcc
H10
120Ω
M10
74132
L10
74132
EXC0
H6.4
EXC1
H6.6
EXC2
H6.8
EXC3
H6.10
G8
3→8
74LS138
SYS.CLEAR
J2.7
EXC
H6.3
DIOX
B9.2
DTIX
G10.11
DMA.FIN (MUX6)
J9.17
DMA DATA
D1.12
D9.10
N.C
CONT→VAR
K5.1
CONT→VAR
K10.1
CONTROL SIGNALS FROM VARIAN


BOARD 4 PAGE 5 OF 6
CONTROL SIGNALS
TO/FROM VARIAN


D3.8 CONT→VAR
H0.11 CONT→VAR
Vcc
H9
270Ω
K8
K9
K10
K11
o.c.
7438
M8
L8
M9
L9
74132
EBUS 0
EBUS 1
EBUS 2
EBUS 3
EBUS 4
EBUS 5
EBUS 6
EBUS 7
EBUS 8
EBUS 9
EBUS 10
EBUS 11
EBUS 12
EBUS 13
EBUS 14
EBUS 15
A6.4
A6.6
A6.8
A6.10
A6.16
A6.18
A6.20
A6.22
D6.4
D6.6
D6.8
D6.10
D6.16
D6.18
D6.20
D6.22
TWISTED PAIRS
BI-DIRECTIONAL BUS FROM CONTROLLER
TO VARIAN
BOARD 4 PAGE 6 OF 6
BI-DIRECTIONAL E-BUS DATA

J3 CONNECTOR

	1	2	3	4	5	6	7	8	9	10	11	
A	7404	7404	7430	7403	7403	A6						A
B	7404	7400	7402	7403	7432	D6		7404	74132	7438	COMPONENTS	B
D	7404	7408	7476	7403					74132	7438		D
E	7404	7432	74148	7403		E6			74132	7438	COMPONENTS	E
F	7403	7430	74138	7400					74132	7438		F
G	7402	7474	74138	7408		G6			74132	7438		G
H	7408	7474	7473	7405		H6			74132	7438		H
K	7400	7400	7473	7473						COMPONENTS	COMPONENTS	K
L						L6						L
M						N6						M
N												N
R												R
S												S
T												T
	1	2	3	4	5	6	7	8	9	10	11	

J1 CONNECTOR (rows B–G)

J2 CONNECTOR (rows L–S)

VARIAN INTERFACE

CARD CAGE: VARIAN SLOT: BOARD: 5

BOARD NAME : VARIAN INTERFACE　　CARD CAGE:VARIAN 620/L SLOT:
BOARD:5

CONNECTOR J1

PIN	SIGNAL	PIN	SIGNAL
1	EBUS0	2	
3	EBUS1	4	
5	EBUS2	6	
7	EBUS3	8	
9	Polari-sing Pin	10	Polari-sing Pin
11	EBUS4	12	
13	EBUS5	14	
15	EBUS6	16	
17	EBUS7	18	
19	EBUS8	20	
21	EBUS9	22	
23	EBUS10	24	GROUND
25	EBUS11	26	
27	EBUS12	28	
29	EBUS13	30	
31	EBUS14	32	
33	EBUS15	34	
35	EXC0	36	
37	EXC1	38	
39	EXC2	40	
41	EXC3	42	
43	VAR-CONT0	44	

CONNECTOR J2

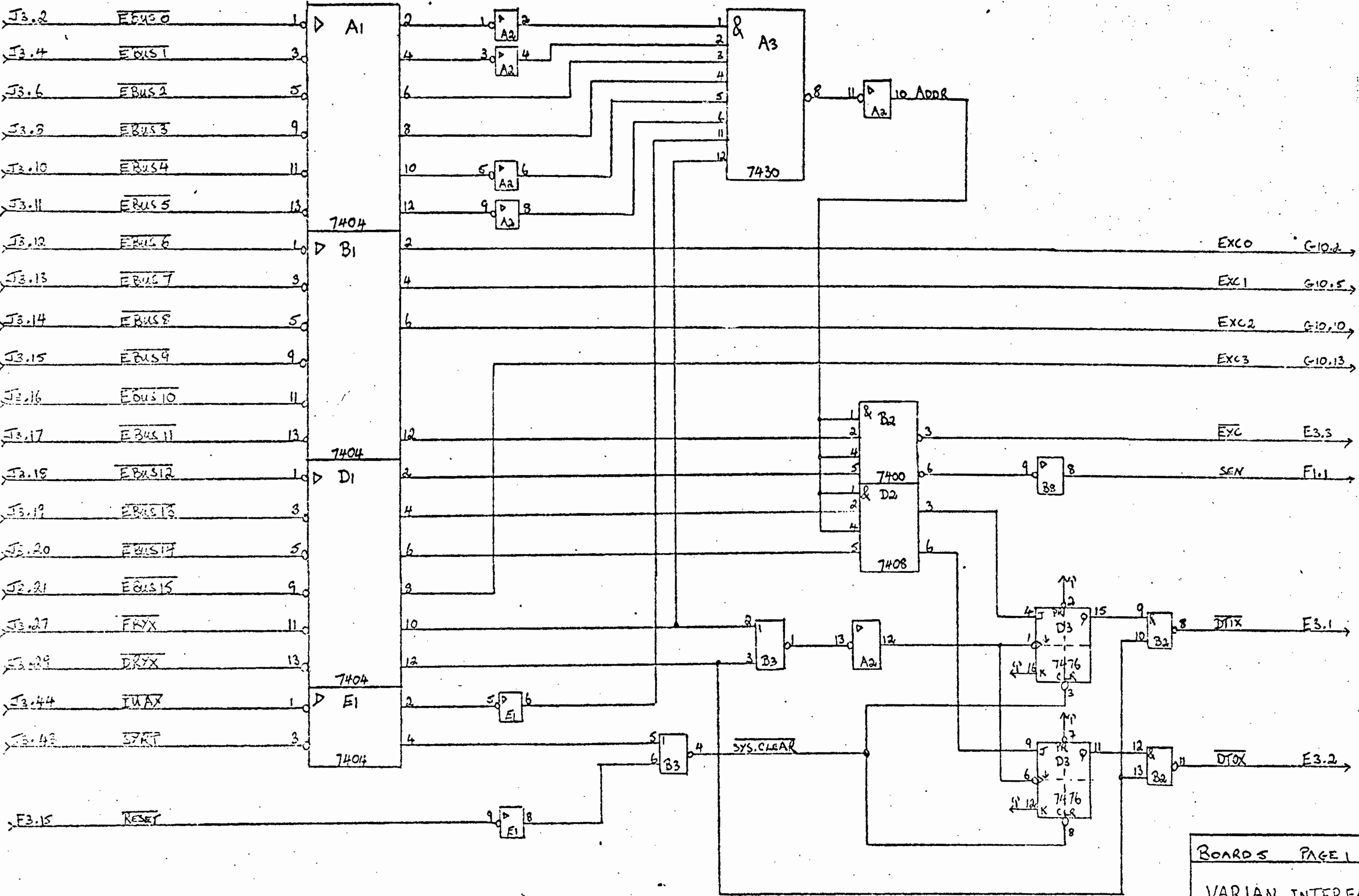
PIN	SIGNAL	PIN	SIGNAL
1	VAR-CONT1	2	
3	VAR-CONT2	4	
5	CONT→VAR	6	
7		8	
9	Polari-sing Pin	10	Polari-sing Pin
11		12	
13	CONT-VAR0	14	
15	CONT-VAR1	16	
17	CONT-VAR2	18	
19	INTERRUPT	20	
21		22	
23		24	GROUND
25		26	
27		28	
29		30	
31		32	
33		34	
35		36	
37		38	
39		40	
41		42	
43		40	

BOARD NAME : VARIAN INTERFACE CARD CAGE: VARIAN 620/L SLOT :

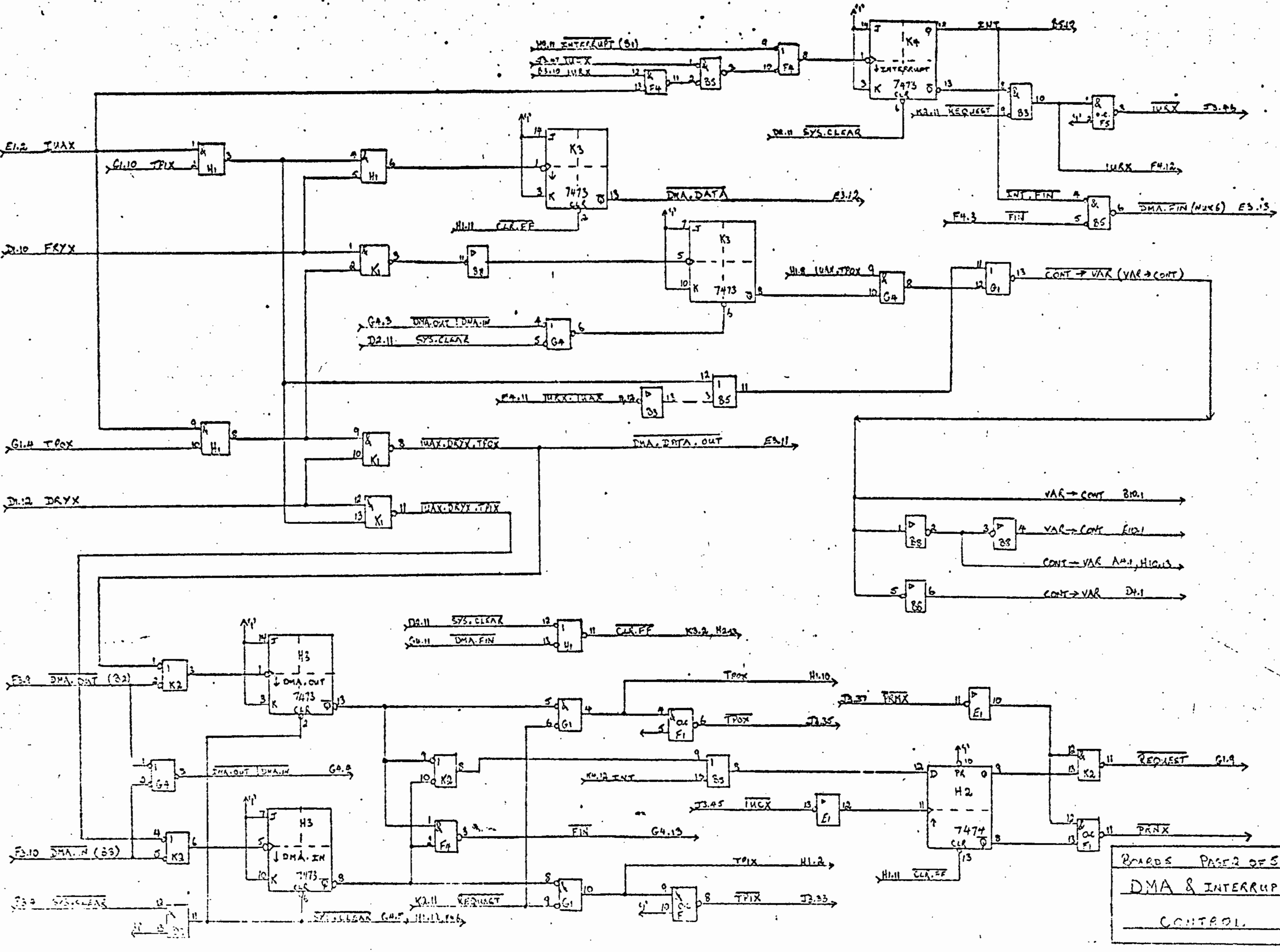
BOARD: 5

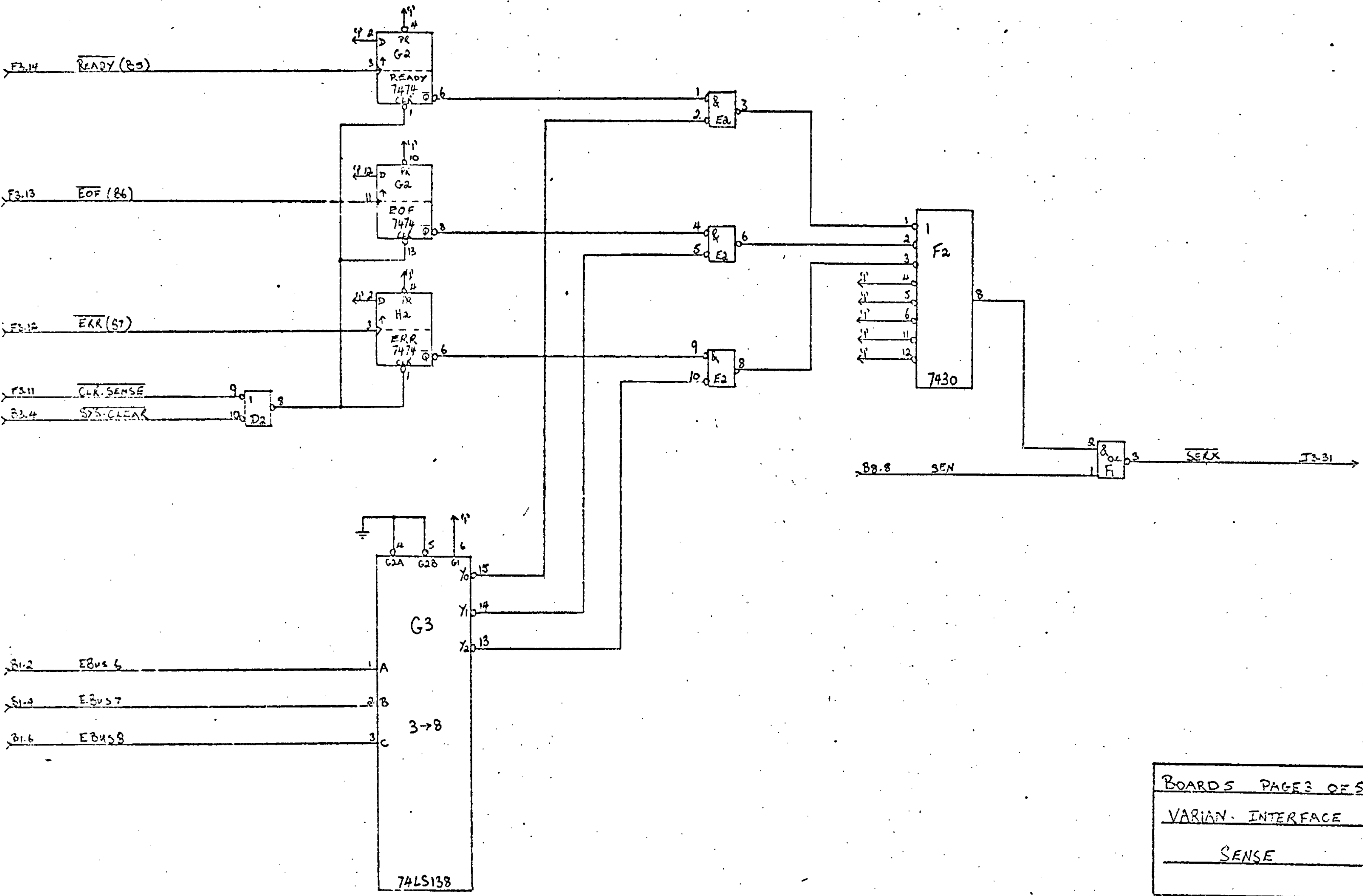
CONNECTOR J3

PIN	SIGNAL	PIN	SIGNAL	PIN	SIGNAL	PIN	SIGNAL
1	GND	2	$\overline{\text{EBUS0}}$	63		64	
3	GND	4	$\overline{\text{EBUS1}}$	65		66	
5	GND	6	$\overline{\text{EBUS2}}$	67		68	
7	GND	8	$\overline{\text{EBUS3}}$	69		70	
9	GND	10	$\overline{\text{EBUS4}}$	71		72	
11	$\overline{\text{EBUS5}}$	12	$\overline{\text{EBUS6}}$	73		74	
13	$\overline{\text{EBUS7}}$	14	$\overline{\text{EBUS8}}$	75		76	
15	$\overline{\text{EBUS9}}$	16	$\overline{\text{EBUS10}}$	77		78	
17	$\overline{\text{EBUS11}}$	18	$\overline{\text{EBUS12}}$	79		80	
19	$\overline{\text{EBUS13}}$	20	$\overline{\text{EBUS14}}$	81		82	
21	$\overline{\text{EBUS15}}$	22	GND	83		84	
23		24	GND	85		86	
25		26	GND	87		88	
27	$\overline{\text{FRYX}}$	28	GND	89		90	
29	$\overline{\text{DRYX}}$	30	GND	91		92	
31	$\overline{\text{SERX}}$	32	GND	93		94	
33	$\overline{\text{TPIX}}$	34	GND	95		96	
35	$\overline{\text{TPOX}}$	36	GND	97		98	
37	PRMX	38	GND	99		100	GND
39		40	GND	101		102	
41		42		103		104	
43	$\overline{\text{SXRT}}$	44	$\overline{\text{IUAX}}$	105		106	
45	$\overline{\text{IUCX}}$	46	$\overline{\text{IURX}}$	107		108	
47	$\overline{\text{IUJX}}$	48		109		110	
49		50		111		112	
51		52		113		114	
53		54		115		116	
55		56		117		118	
57		58		119		120	
59		60		121	+5V	122	GND
61		62					

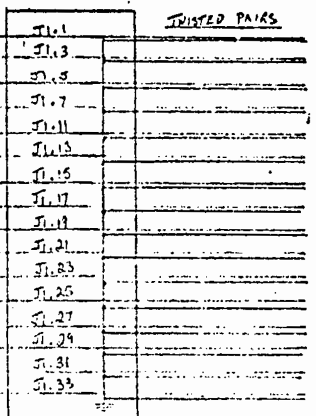


J3.2
EBUS 0
J3.4
EBUS 1
J3.6
EBUS 2
J3.10
EBUS 4
J3.11
EBUS 5
J3.12
EBUS 6
J3.13
EBUS 7
J3.14
EBUS 8
J3.15
EBUS 9
J3.16
EBUS 10
J3.17
EBUS 11
J3.18
EBUS 12
J3.19
EBUS 13
J3.20
EBUS 14
J3.21
EBUS 15
J3.27
J3.44
F3.15
RESET
A1
B1
D1
E1
7404
A2
A3
7430
10 ADDR
B2
7400
D2
7408
B3
D3
7476
E3.3
F1.1
E3.1
E3.2
EXC0
EXC1
EXC2
EXC3
EXC
SEN
DTIX
DTOX
SYS.CLEAR
BOARD 5 PAGE 1 OF 5
VARIAN INTERFACE


BOARDS PAGE 2 OF 3
DMA & INTERRUPT
CONTROL



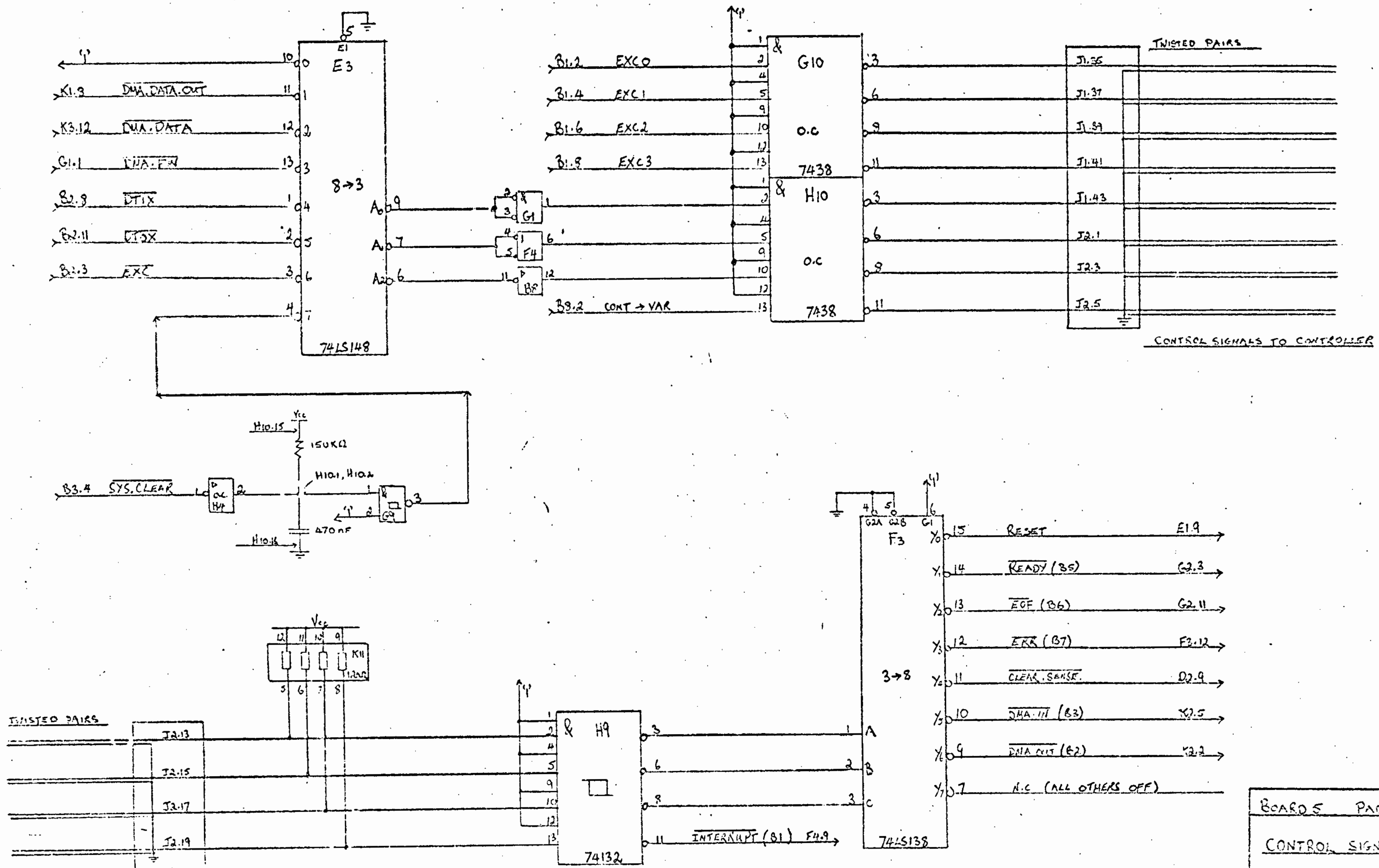
F3.14
READY (B5)
G2
READY
7474
F3.13
EOF (B6)
G2
EOF
7474
F3.12
ERR (B7)
H2
ERR
7474
F3.11
CLK. SENSE
B3.4
SYS. CLEAR
D2
E2
F2
7430
B8.8
SEN
F1
SERX
J3-31
G3
3→8
74LS138
B1.2
EBus 6
B1.4
EBus 7
B1.6
EBus 8
BOARD 5 PAGE 3 OF 5
VARIAN. INTERFACE
SENSE



BI-DIRECTIONAL BUS FROM VARIAN TO CONTROLLER

BOARD 5 PAGE 4 OF 5

BI-DIRECTIONAL E-BUS DATA FROM VARIAN TO CONTROLLER AND CONTROLLER TO VARIAN



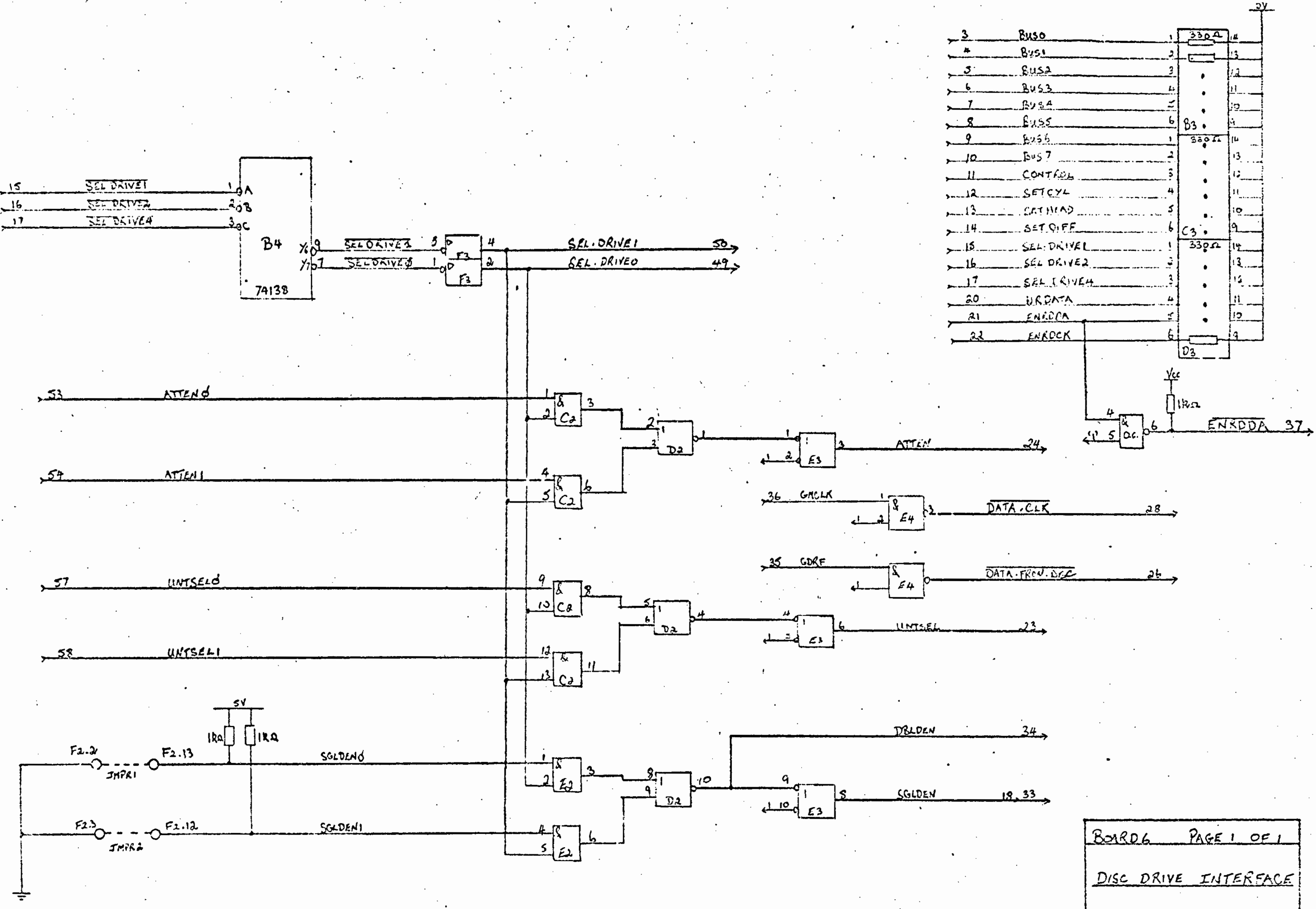
TWISTED PAIRS
CONTROL SIGNALS TO CONTROLLER
E3
8→3
74LS148
DMA DATA OUT
DMA DATA
DTIX
EXC
EXC0
EXC1
EXC2
EXC3
CONT → VAR
G10
O.C
7438
H10
O.C
7438
SYS. CLEAR
150KΩ
470nF
F3
3→8
74LS138
RESET
READY (B5)
EOF (B6)
ERR (B7)
CLEAR SENSE
N.C (ALL OTHERS OFF)
H9
74132
INTERRUPT (B1)
TWISTED PAIRS
BOARD 5 PAGE 5 OF 5
CONTROL SIGNALS
TO/FROM CONTROLLER

	6	5	4	3	2	1	
A							A
B			74138	COMPONENTS	COMPONENTS	CONNECTORS	B
C				COMPONENTS	7408	CONNECTORS	C
D				COMPONENTS	7402	CONNECTORS	D
E			7438	7438	7408	CONNECTORS	E
F				7404	JUMPERS	CONNECTORS	F
G						CONNECTORS	G
H							H
	6	5	4	3	2	1	

DISC DRIVE INTERFACE   CARD CAGE: B   SLOT: 1   BOARD: 6"

BOARD NAME : DISC DRIVE INTERFACE CARD CAGE:B SLOT:1 BOARD:6

PIN	SIGNAL	PIN	SIGNAL
1	+5V	2	+5V
3	BUS0	4	BUS1
5	BUS2	6	BUS3
7	BUS4	8	BUS5
9	BUS6	10	BUS7
11	CONTROL	12	SETCYL
13	SETHEAD	14	SETDIFF
15	$\overline{\text{SEL.DRIVE1}}$	16	$\overline{\text{SEL.DRIVE2}}$
17	$\overline{\text{SEL.DRIVE4}}$	18	SELDEN
19	GND	20	WRDATA
21	ENRDDA	22	ENRDCK
23	UNTSEL	24	ATTEN
25	GND	26	$\overline{\text{DATA.FROM.DISC}}$
27	GND	28	$\overline{\text{DATA.CLK}}$
29		30	
31		32	
33	SGLDEN	34	DBLDEN
35	GRDF	36	GMCLK
37	$\overline{\text{ENRDDA}}$	38	
39		40	
41		42	
43		44	
45		46	
47		48	
49	SELDRIVE0	50	SELDRIVE1
51		52	
53	ATTEN0	54	ATTEN1
55		56	
57	UNTSEL0	58	UNTSEL1
59		60	
61	GND	62	GND


BOARD 6 PAGE 1 OF 1
DISC DRIVE INTERFACE
BUS0
BUS1
BUS2
BUS3
BUS4
BUS5
BUS6
BUS7
SET CYL
SET DIFF
SEL DRIVE1
SEL DRIVE2
SEL DRIVE4
WRDATA
ENRDDA
ENRDCK
330Ω
B3
C3
D3
Vcc
1KΩ
ENRDDA 37
B4
74138
SEL.DRIVE1 50
SEL.DRIVE0 49
ATTEN0
ATTEN1
UNTSEL0
UNTSEL1
SGLDEN0
SGLDEN1
5V
1KΩ
F2.2
F2.13
F2.3
F2.12
JMPR1
JMPR2
ATTEN 24
GMCLK
DATA.CLK 28
GDRF
UNTSEL 23
DBLDEN 34
SGLDEN 18, 33

PCB NAME : DRIVE MULTIPLEX XMTR/RCVR CARD CAGE : B SLOT : 2

PIN	SIGNAL	PIN	SIGNAL
1	+5V	2	+5V
3	+3V	4	+3V
5	-3V	6	-3V
7		8	
9		10	$\overline{\text{SEEKRDY}}$
11	CYL	12	
13	HEAD	14	$\overline{\text{SEEKERR}}$
15	DIFF	16	
17		18	$\overline{\text{SECTOR}}$
19		20	
21		22	$\overline{\text{INDEX}}$
23	CONTROL	24	
25		26	$\overline{\text{ONLINE}}$
27		28	
29		30	$\overline{\text{UNSAFE}}$
31		32	
33		34	
35		36	
37		38	$\overline{\text{WRITE.CUR}}$
39		40	
41		42	
43		44	
45		46	
47		48	
49		50	
51		52	
53		54	
55		56	
57		58	
59		60	
61	GND	62	GND

PIN	SIGNAL	PIN	SIGNAL
XA	N/C	X1	$\overline{\text{SEEKRDY}}$
XB	GND	X2	GND
XC	$\overline{\text{CYL}}$	X3	$\overline{\text{SEEKERR}}$
XD	GND	X4	$\overline{\text{SECTOR}}$
XE	$\overline{\text{HEAD}}$	X5	GND
XF	GND	X6	$\overline{\text{INDEX}}$
XH	$\overline{\text{DIFF}}$	X7	$\overline{\text{ONLINE}}$
XJ	GND	X8	GND
XK	N/C	X9	$\overline{\text{UNSAFE}}$
XL	GND	X10	
XM	N/C	X11	GND
XN	GND	X12	$\overline{\text{WRITE.CUR}}$
XP	N/C	X13	GND
XR	GND	X14	N/C
XS	$\overline{\text{CONTROL}}$	X15	GND

PCB NAME : DRIVE MULTIPLEX XMTR/RCVR   CARD CAGE : B   SLOT : 3

PIN	SIGNAL	PIN	SIGNAL
1	+5V	2	+5V
3	+3V	4	+3V
5	-3V	6	-3V
7		8	
9	BUS0	10	$\overline{CAR128}$
11	BUS1	12	
13	BUS2	14	$\overline{CAR64}$
15	BUS3	16	
17	BUS4	18	$\overline{CAR32}$
19	BUS5	20	
21	BUS6	22	$\overline{CAR16}$
23	BUS7	24	
25		26	$\overline{CAR8}$
27		28	
29		30	$\overline{CAR4}$
31		32	
33		34	$\overline{CAR2}$
35		36	
37		38	$\overline{CAR1}$
39		40	
41		42	
43		44	
45		46	
47		48	
49		50	
51		52	
53		54	
55		56	
57		58	
59		60	
61	GND	62	GND

PIN	SIGNAL	PIN	SIGNAL
XA	$\overline{BUS0}$	X1	$\overline{CAR128}$
XB	GND	X2	GND
XC	$\overline{BUS1}$	X3	$\overline{CAR64}$
XD	GND	X4	$\overline{CAR32}$
XE	$\overline{BUS2}$	X5	GND
XF	GND	X6	$\overline{CAR16}$
XH	$\overline{BUS3}$	X7	$\overline{CAR8}$
XJ	GND	X8	GND
XK	$\overline{BUS4}$	X9	$\overline{CAR4}$
XL	GND	X10	$\overline{CAR2}$
XM	$\overline{BUS5}$	X11	GND
XN	GND	X12	$\overline{CAR1}$
XP	$\overline{BUS6}$	X13	GND
XR	GND	X14	N/C
XS	$\overline{BUS7}$	X15	GND

PCB NAME : DRIVE SIMPLEX　　　　CARD CAGE : B　　SLOT : 4

PIN	SIGNAL	PIN	SIGNAL
1	+5V	2	+5V
3	+3V	4	+3V
5	-3V	6	-3V
7		8	
9	UNTSEL0	10	
11	UNTSEL1	12	
13	ATTEN0	14	
15	ATTEN1	16	
17		18	
19		20	
21		22	
23		24	
25		26	
27		28	
29		30	
31		32	
33		34	
35		36	
37		38	
39		40	
41		42	
43		44	
45		46	$\overline{\text{ENRDDA}}$
47		48	GOWRITE
49		50	SELDRIVE0
51		52	SELDRIVE1
53		54	$\overline{\text{READ1}}$
55	$\overline{\text{READ0}}$	56	
57		58	
59		60	
61	GND	62	GND

PIN	SIGNAL	PIN	SIGNAL
XA	$\overline{\text{UNTSEL0}}$	X1	$\overline{\text{SELDRIVE0}}$
XB		X2	
XC	$\overline{\text{UNTSEL1}}$	X3	$\overline{\text{SELDRIVE1}}$
XD		X4	
XE	$\overline{\text{ATTEN0}}$	X5	
XF	GND	X6	
XH	$\overline{\text{ATTEN1}}$	X7	
XJ	GND	X8	WRITE0
XK		X9	
XL		X10	WRITE1
XM		X11	
XN	READ0	X12	
XP	GND	X13	
XR	GND	X14	
XS	READ1	X15	

PCB NAME : VFO   CARD CAGE : B   SLOT : 5

PIN	SIGNAL	PIN	SIGNAL
1	+5V	2	+5V
3	+3V	4	+3V
5	-3V	6	-3V
7	-15V	8	-15V
9	+15V	10	+15V
11		12	
13		14	
15		16	
17	$\overline{\text{ORDATA}}$	18	
19		20	
21		22	
23		24	
25	VFOCLK	26	
27		28	
29		30	
31	DELDAT	32	
33	SGLDEN	34	
35		36	
37	DBLDEN	38	
39		40	
41		42	
43		44	
45		46	
47		48	
49		50	
51		52	
53		54	
55		56	
57		58	
59		60	
61	GND	62	GND

PCB NAME : READ/WRITE/CLOCK CARD CAGE : B SLOT : 6

PIN	SIGNAL	PIN	SIGNAL
1	+5V	2	+5V
3		4	
5		6	
7	$\overline{\text{READ0}}$	8	$\overline{\text{ORDATA}}$
9	$\overline{\text{READ1}}$	10	
11		12	
13		14	
15		16	
17		18	
19		20	
21	ENRDCK	22	
23		24	RDF
25		26	
27		28	
29		30	MCLK
31		32	
33	DBLDEN	34	GOWRITE
35	SGLDEN	36	
37		38	WRDATA
39		40	DELDAT
41		42	
43	VFOCLK	44	
45	ENRDDA	46	
47		48	
49		50	
51		52	
53		54	
55		56	
57		58	
59		60	
61	GND	62	GND

APPENDIX F

MEASUREMENT OF THE EXPERIMENTAL TASK

In Figures F.1, F.2 and F.3, the times are given in the order in which they occur. For example, consider Figure F.1. The task associated with this figure begins with a task swop (t_{SWOP}). The task will then enter run mode, and will remain running until it becomes blocked (t_{RUN1}). The task will become blocked when an I/O operation is initiated, and will remain blocked until the I/O operation is complete ($t_{BLOCKED}$). During the task's blocked time, a DMA transfer will occur (t_{DMA}). On completion of the I/O operation, an interrupt will 'reawaken' the task. A task swop will then occur (t_{SWOP}), and the task will run until its completion (t_{RUN2}).

Figure F.1 gives the 'life' of the task when the HLDC is used. The different components of the task's life are described in Table F.1.

Figure F.2 gives the task's blocked time when the HLDC is used (during this period of time, the HLDC will perform the file operation) the different components of the blocked time are described in Table F.2.

Figure F.3 gives the task's life when the low-level controller is used. The different components of the task's life are described in Table F.3.

Task life (refer Table F.1 for definitions)	Execution time $t^H_{EXECUTION}$	Response time $t^H_{RESPONSE}$
t_{SWOP}	200	200
t_{RUN1}	18	18
$t_{DMA}/t_{BLOCKED}$	1654	72765 (34550)
t_{SWOP}	200	200
t_{RUN2}	11	11
	≅ 2085	≅ 73195 (34980)

NOTES:

(i) The time in brackets is for a fixed head disc drive with a 2,5 million bits per second transfer rate.

(ii) All times are in microseconds.

Figure F.1 The Life of the Experimental Task when the HLDC is used.

Task blocked time (refer Table F.2 for definitions).	Device independant times	Device dependant times
t_{RUN1}	133 (112)	
t_{WAIT1}		30000 (13625)
t_{RUN2}	172 (117)	
t_{WAIT2}		2330 (2385)
t_{RUN3}	116 (79)	
t_{WAIT3}		30000 (13625)
t_{RUN4}	39 (26)	
t_{WAIT4}		2460 (1225)
t_{RUN5}	26 (18)	
t_{WAIT5}		2475 (1230)
t_{RUN5}	26 (18)	
t_{WAIT5}		2475 (1230)
t_{RUN5}	26 (18)	
t_{WAIT5}		2475 (1230)
t_{RUN6}	14 (10)	
	552 (398)	72215 (34550)

$$t_{BLOCKED} = 552 + 72215 \ (398 + 34550)$$
$$\cong 72765 \ (34950)$$

NOTES:

(i) The microinstruction cycle time is 280 nanoseconds. The device independant times in brackets is the expected execution time with a 190 nanosecond microinstruction cycle time.

(ii) The device dependant times are for the CalComp CD1 moving head disc drive. The times in brackets are for a fixed head disc drive with a 2,5 million bits per second transfer rate.

(iii) All times are in microseconds.

Figure F.2 The Task's Blocked Time when the HLDC is used

Task life (refer Table F.3 for definitions)	Execution Time $t^L_{EXECUTION}$	Response Time $t^L_{RESPONSE}$	
t_{SWOP}	200	200	
t_{RUN1}	268	268	
$t_{DMA1}/t_{BLOCKED1}$	403	30000	(13625)
t_{SWOP}	200	200	
t_{RUN2}	1753	1753	
$t_{DMA1}/t_{BLOCKED2}$	403	3245	(1995)
t_{SWOP}	200	200	
t_{RUN3}	1129	1129	
$t_{DMA1}/t_{BLOCKED3}$	403	30000	(13625)
t_{SWOP}	200	200	
t_{RUN4}	457	457	
$t_{DMA2}/t_{BLOCKED4}$	1613	12045	(5795)
t_{SWOP}	200	200	
t_{RUN5}	38	38	
	≅ 7465	≅ 79935	(39685)

NOTES:

(i) The times in brackets is for a fixed head disc drive with a 2,5 million bits per second transfer rate.

(ii) All times are in microseconds.

Figure F.3 The Life of the Experimental Task when a Low-Level Disc Controller is used.

Notes to Tables F.1, F.2 and F.3

(i) All times associated with the disc drive are for "normal" supply conditions. At the normal supply conditions, the disc speed is 2400 revolutions per minute[1].

(ii) Half a disc revolution takes 12500 microseconds[2].

(iii) When a sector is read from disc to main memory, only the first 175 words of data is accessed. The remaining 20 words of data being used to provide for variations in disc speeds, and for computations to occur (refer Figure D.3 in Appendix D). Since a bit of data and a bit of clock is read off disc every 0,8 microseconds at the normal supply conditions (refer Appendix D.4), 175 words (16 bits/word) will be read off disc in 175 x 16 x 0,8 ≅ 2250 microseconds; and 20 words will be read off disc in 20 x 16 x 0,8 ≅ 250 microseconds.

(iv) Each word of data transferred to main memory via DMA inhibits the Varians main processor by 3,15 microseconds[3].

Consequently :

(a) the transfer of one sector (128 words) to main memory via DMA will inhibit the main processor by 128 x 3,15 = 403 microseconds; and

(b) the transfer of four sectors will inhibit the main processor by 4 x 128 x 3,15 = 1613 microseconds.

[1] CalComp Field Engineering Service Handbook, CD1 Disc Drive, California Computer Products, Inc., Anaheim, California (1971), p. 23.

[2] Ibid.

[3] Varian 620/L Computer Handbook, Varian Data Machines, Irvine, California (1971), chapter 8, p. 10.

t_{SWOP} : When a task enters run mode, a task swop is required. The task-swop time is 200 microseconds (refer Section 4.3).

t_{RUN1} : The time from the task being initiated to the Varian initiating the file command. t_{RUN1} is found to be 18 microseconds.

$t_{BLOCKED}$: The time from the initiation of the file command to an interrupt being received from the HLDC indicating the completion of the file command. $t_{BLOCKED}$ = 72765 microseconds.

t_{RUN2} : The time from the interrupt being received from the HLDC to the task being complete (i.e. the Varian entering halt mode). t_{RUN2} = 11 microseconds.

t_{DMA} : The time taken up by cycle stealing when performing the direct memory access transfers. The HLDC will receive 13 words of the filename block from main memory, and will transfer four sectors of data to main memory. Consequently the DMA transfer time will be 1654 microseconds (refer Note (iv) above).

Table F.1

t_{RUN1} The time from the initiation of the file command (i.e. when the HLDC receives a single-word output transfer command from the Varian), to the initiation of the seek for the appropriate sector of the Applications File Directory (AFD). This time includes :

(i) The transfer time of the filename block from the Varian to the HLDC via DMA.

(ii) The time required to search through the MFD for the file directory name.

t_{RUN1} = 133 microseconds.

t_{WAIT1} The time required to transfer the first sector of the AFD into local memory. This time includes :

(i) The time to perform the disc seek from cylinder 0 to cylinder 7. This takes 15250 microseconds.

(ii) The time from the completion of the disc seek to the required sector of the AFD being under the read/write heads. This is a function of which sector will be under the heads when the seek is complete. On average, half a disc revolution will be required before the required sector is under the heads. Half a disc revolution takes 12500 microseconds (refer Note (ii) above), and it will be assumed that this is the time required before the required sector is under the heads.

(iii) The time from the required sector being under the heads, to this sector being fully transferred into local memory. This is equal to the reading of 175 words of data off disc, which takes 2250 microseconds (refer Note (iii) above).

$$t_{WAIT1} = 15250 + 12500 + 2250 = 30000 \text{ microseconds}$$

Table F.2

t_{RUN2} The time from the transfer of the first sector of the AFD being complete to the initiation of the transfer of the second sector of the AFD. This includes the time required to search through the entire sector of the AFD for the file name. (Note that the file name will not be in the first sector of the AFD). t_{RUN2} = 172 microseconds.

t_{WAIT2} The time required to transfer the next sector of the AFD from disc to local memory. As mentioned in Section 4.3.2, the AFD will be stored in contiguous locations on disc. Consequently, if t_{RUN2} is too long, a full disc revolution will be required before the appropriate sector of the AFD is obtained. To prevent an unwanted disc revolution, t_{RUN2} must be less than 250 microseconds (refer note (iii) above). Since t_{RUN2} is equal to 172 microseconds, an unwanted disc revolution will not occur. t_{WAIT2} = 2330 microseconds.

t_{RUN3} The time from the transfer of the second sector of the AFD into local memory being complete, to the initiation of the disc seek required to obtain the index block. This includes the time required to search through the second sector of the AFD - the file name being in this sector. t_{RUN3} = 116 microseconds.

t_{WAIT3} This is the same as t_{WAIT1} above, except that the search is from cylinder 7 to cylinder 14 - the seek distance still being 7 cylinders. t_{WAIT3} = 30000 microseconds.

t_{RUN4} The time from the transfer of the index block into local memory being complete, to the initiation of the transfer of the first sector of EXP to main memory. This time includes the time required to check the access control information. t_{RUN4} = 39 microseconds.

Table F.2 (continued)

t_{WAIT4}	This time is similar to t_{WAIT2} above. Again, the HLDC must respond quickly enough to ensure that an unnecessary disc revolution does not occur. Since t_{RUN4} is equal to 39 microseconds, which is less than 250 microseconds (refer t_{WAIT2} above), the HLDC will respond quickly enough to prevent an unwanted disc revolution. t_{WAIT4} = 2460 microseconds.
t_{RUN5}	The time from the transfer of a sector to main memory being complete, to the initiation of the transfer of the next sector of data, if required. t_{RUN5} = 26 microseconds.
t_{WAIT5}	This time is similar to t_{WAIT2} above. Since t_{RUN5} is only 26 microseconds, an unnecessary disc revolution will not occur. t_{WAIT} = 2475 microseconds.
t_{RUN6}	The time from the last sector being fully transferred to main memory, to the HLDC issuing an interrupt indicating the end of the file operation. t_{RUN6} = 14 microseconds.

Table F.2 (continued)

t_{SWOP} When a task enters run mode, a task swop is required. The task-swop time is 200 microseconds (refer Section 4.3).

t_{RUN1} The time from the experimental task being initiated, to the task instructing the low-level controller to bring the first sector of the AFD off disc into main memory. t_{RUN1} = 268 microseconds.

$t_{BLOCKED1}$ The time from the Varian instructing the low-level controller to bring the first sector of the AFD off disc into main memory, to an interrupt being received from the low-level controller indicating the completion of this operation. This time includes :

(i) The time to perform a disc seek from cylinder 0 to cylinder 7. This takes 15250 microseconds.

(ii) The time from the completion of the disc seek to the required sector of the AFD being under the read/write heads. This is a function of which sector will be under the heads when the seek is complete. On average, half a disc revolution will be required before the required sector is under the heads. Half a disc revolution takes 12500 microseconds (refer Note (ii) above), and it will be assumed that this is the time required before the required sector is under the heads.

(iii) The time from the required sector being under the heads, to this sector being fully transferred into local memory. This is equal to the reading of 175 words of data off disc, which takes 2250 microseconds (refer Note (iii) above).

$t_{BLOCKED1}$ = 15250 + 12500 + 2250
= 30000 microseconds.

Table F.3

t_{RUN2} The time from an interrupt being received from the low-level controller indicating the end of the transfer of the first sector of the AFD, to the initiation of the transfer of second sector of the AFD. This time includes the time required to search through the entire sector of the AFD for the file name. (Note that the file name will not be in the first sector of the AFD). t_{RUN2} = 1753 microseconds.

$t_{BLOCKED2}$ The time required to transfer the next sector of the AFD from disc to main memory. As mentioned in Section 4.3.2, the AFD will be stored in contiguous locations on disc. Consequently, if t_{RUN1} is too long, a full disc revolution will be required before the appropriate sector of the AFD is obtained. To prevent this unwanted disc revolution, t_{RUN2} must be less than 250 microseconds (refer Note (iii) above). Since t_{RUN2} is equal to 1753 microseconds, a disc revolution will occur before the required sector is of the AFD is obtained. There are, however, two methods which could be used in avoiding a full disc revolution :

(i) A double buffering technique of storing data could be used. Initially the first buffer would be filled with the first sector of the AFD, and then, whilst the second sector of the AFD is being transferred into the second buffer, the first buffer would be searched for the file name. Since the search through a buffer will be complete before a sector is fully transferred from disc, the first buffer area can be used for the following sector of the AFD. This procedure would continue until the file name is found. The disadvantage of this method is that one more sector of the AFD will be transferred to main memory than is required. This will increase the task execution time by the cycle stealing associated with the transfer of one sector (i.e. 403 microseconds).

(ii) Another method would be to store the AFD on alternate sectors on disc as opposed to contiguous sectors. A sector of data would be read into main memory, and the search through this sector would be complete before the next sector of the AFD is under the read/write heads.

Table F.3 (continued)

This method, whilst increasing the task's response time, will not increase the task execution time.

Since the increase in the task's execution time when using method (i) is relatively large, whilst the increase in the task's response time when using method (ii) is relatively small, it is assumed that method (ii) would be used. When using method (ii), $t_{BLOCKED2}$ = 3245 microseconds.

t_{RUN3}	The time from an interrupt being received from the controller to the initiation of the transfer of the index block. This time includes the time required to search through the second sector of the AFD - the file name being in this sector. t_{RUN3} = 1129 microseconds.
$t_{BLOCKED3}$	The time required to transfer the index block from disc to main memory. This I/O operation is similar to $t_{BLOCKED1}$, except that the search is from cylinder 7 to cylinder 14. The seek distance, however, is still seven cylinders. $t_{BLOCKED3}$ = 30000 microseconds.
t_{RUN4}	The time from an interrupt being received from the controller, to the initiation of the transfer of EXP from disc to main memory. This time includes : (i) The check of the access control information. (ii) The calculation of how many sectors can be transferred by one I/O operations. Since the four sectors making up EXP are in contiguous locations on disc, only one I/O operation is required. t_{RUN4} = 457 microseconds.
$t_{BLOCKED4}$	The time required to transfer four sectors of information from disc to main memory. A similar problem occurs with $t_{BLOCKED4}$ that occurred with $t_{BLOCKED2}$ above. That is,

Table F.3 (continued)

since t_{RUN4} is greater than 250 microseconds, a full disc revolution will be required before the first sector of EXP is obtained. A possible solution to this problem is to still store EXP in contiguous sectors, but to leave a sector between the index block and the first sector of the file. The processing associated with the index block could then be performed whilst this sector passes under the read/write heads. The disadvantage of this method is that this sector will probably remain unused. That is, associated with each index block there may be an 'unusable' sector. When using this method, $t_{BLOCKED4}$ = 12245 microseconds.

t_{RUN5}	The time from an interrupt being received by the Varian indicating the end of the data transfer, to the Varian entering halt mode. t_{RUN5} = 38 microseconds.
t_{DMA1}	The time required to transfer one sector of data from disc to main memory via DMA. This is equal to 403 microseconds. (refer Note (iv) above).
t_{DMA2}	The time required to transfer four sectors of data from disc to main memory via DMA. This is equal to 1613 microseconds (refer Note (iv) above).

Table F.3 (continued)