ORGANIZATION
OF
INDUSTRIAL CONTROL COMPUTERS

By Michael G. Rodd

Submitted to the University of Cape Town

in fulfilment of the requirements

for the Degree of

Doctor of Philosophy.

September 1976

STATEMENT.

The author declares that the theories proposed in this thesis are his original work, developed independently, and that it has not been submitted for a degree at any other university.

ABSTRACT

The efficient use of industrial control computers is recognized as an organizational problem akin to the traffic-switching problem in communications.

A systematic approach to this problem is proposed, based on theory developed for the handling of telephone traffic.

The application of the approach indicates that it is necessary to re-evaluate traditional hardware/software relationships. A change in these relationships is desirable, since multiprogrammed computers spend too much time in handling their own organization. This situation is compounded in time-critical industrial process-control applications.

It is proposed that the solution lies in the use of a flexible hardware operating system, working in close relationship with a conventional minicomputer. The unit proposed to implement this function, termed a microcontroller, makes use of the new bipolar microprocessor elements and provides a high-speed, flexible control unit, adaptable to user requirements. To retain a high degree of flexibility the microcontroller is microprogrammable. In essence, the unit executes the principal functions of a real-time operating system, acts as a pre-processor for all incoming requests, and ensures a high rate of task-switching.

This system is applied to a series of configurations, each selected to demonstrate, quantitatively, the value of the technique in real applications. Comparisons are made between real-time control configurations based on the software-implemented approach and the identical configurations based on this system. The proposed strategy is shown to result in a better and more economical industrial controller.

The wider implication for any aspect of organization is that "bigger" is not necessarily "better". Successful management implies effective use of facilities, rather than a proliferating structure.

ACKNOWLEDGEMENTS

The author wishes to express his sincere appreciation to the people without whose help this dissertation would not have been written. In a project of this nature, the list of those involved would be impossible to complete, but particular mention must be made of the following:

- His supervisor, Professor N.C. de V. Enslin, Head of the Department of Electrical Engineering, University of Cape Town, for his guidance, enthusiasm and interest.
- His co-supervisor, Professor K. MacGregor, Head of the Department of Computer Science, University of Cape Town, for his interest, and help in unravelling the many obscurities in the field of Computer Science.
- Professor J.L.N. Besseling, Department of Electrical Engineering, University of Cape Town, for his constructive scrutiny of the final draft of the text.
- His colleagues in the Department of Electrical Engineering, for their support and tolerance.
- Mr H.F. Weehuizen, who prepared suitable versions of the "BASIC" interpreter program.
- The workshop staff of the Department of Electrical Engineering, who assisted in the construction of various items of hardware.
- Mr K.A. Behr of the Cartographic Unit, Department of Geography, University of Cape Town, for his invaluable help in the presentation of the sketches and circuit diagrams, and for undertaking all the necessary photographic work.
- Mrs J.F. Davis and Mrs L. Moore for typing the final MS, and for Mrs Davis's careful scrutiny of the text.
- His wife, Sue, for the typing and correction of the draft copies of the dissertation, and for her unstinting support and encouragement throughout the years which this work has taken to complete.

Finally, the author wishes to acknowledge the provision, by the University of Cape Town, of the facilities and financial support which made this work possible.

INDEX

Page No.

INDEX TO FIGURES, TABLES AND CIRCUIT DIAGRAMS.

FIGURES:

TABLES:

CIRCUIT DIAGRAMS:

GRAPH:

NOMENCLATURE.

(i) Unless otherwise indicated, all numerical values expressed are decimal numbers. Octal numbers are denoted $x_{(8)}$.

(ii) 1 Kiloword (1 Kword) of memory capacity is equivalent to 1024 words.

(iii) 1 Word contains 16 Bits (Binary Digits) of data, unless otherwise specified.

(iv) 1 Byte contains 8 Bits.

(v) 1 mSec $\equiv$ 1 millisecond $\equiv 10^{-3}$ seconds.

(vi) 1 μSec $\equiv$ 1 microsecond $\equiv 10^{-6}$ seconds.

(vii) All logic symbols used are in accordance with the recommendations of IEC 117 {78}.

(viii) $[x]$ implies the largest integer smaller than or equal to x.

CHAPTER I

INTRODUCTION

"The thing can be done," said the Butcher, "I think.
The thing must be done, I am sure.
The thing shall be done! Bring me paper and ink,
The best there is time to procure."

Lewis Carroll, *The Hunting of the Snark*
(Fit the Fifth)

This dissertation is concerned with organization. Contemporary society is founded on the organization of systems, created to provide for the needs of man's communal life. Organizations exist in all walks of life, from the multinational United Nations to a nursery school's P.T.A. The precise meaning of an "organization" is difficult to formulate. Any definition must take into cognizance the fact that it deals with empirical phenomena, and that the world has an unfortunate way of not allowing itself to fit into clean classifications. The most apt attempt at definition comes from the world of social science: an organization is "both a process and a condition. As a condition, it is the structure of the various units in society in their interrelationships with each other. As a process, it is the development of coordination among the various units of society." {1}

As society evolves towards greater complexity, the theory of organizations occupies an increasingly significant place in modern science {2}. As a system, of any description, becomes larger, linking more and more component parts, the interrelationships between the constituents take on new facets and greater significance. An efficient organization benefits those for whom it was created; an inefficient one leads ultimately to chaos.

The engineer stands in the privileged position of being trained to deal with the problems of systems. He might be an expert in automatic control theory, concerned with the efficient organization of industrial equipment, but the basic principles embodied in the interaction between electrical and mechanical entities can equally be applied to the control of a sociological system. The principles of cause and effect, all too well-known to the social scientist, are the basis of industrial control systems. The engineer is well aware of the fact that his control strategy has limitations. Go beyond the design limits, and instability can result. Introduce noise into a system not designed to cope with it, and unpredicted results will occur.

Neglect a feedback path or a communication channel, and sectors of the system will operate totally independently. A close examination of many sociological systems will reveal similar characteristics.

The introduction of the electronic computer has had an immense impact on society. In almost every aspect of life, this product of man's powers of innovation is exerting an influence. One particular area in which the computer's capacity to perform high-speed, repetitive tasks is used to an increasing extent, is the industrial sector. Ours is an era of ever-increasing industrial capacity, resulting in a need for higher production, more constructive use of labour, and increased manufacturing complexity. Industrialists look to the computer to aid them in achieving these goals.

The first industrial use of computer control was in 1958, when Texaco connected a digital computer directly to a chemical plant, with the aim of process optimization. Since then, an exponentially increasing number of industries, both large and small, have adopted this basic strategy in a variety of ways, from simple data-logging functions to highly sophisticated direct-digital-control. The demands made on the controlling computer have increased from the execution of relatively simple tasks to an extent where total control of complex processes is invested in the computer. The reason for this is very simple - greater productivity.

As the demands increase, the capabilities of the computer must expand likewise. While dramatic technological advances have resulted in faster, more reliable machines, there is, seemingly, a limit to what a single computer is capable of performing. The solutions offered have been many - multiply the complexity of the computer; add hardware; develop more sophisticated software; introduce additional, parallel processors. The computer has become not only a component of a large system, but also a complex system in itself.

1.1 The Organizational Problems of Process-control Computers

Early computer systems presented few organizational problems to their users. Each job that the computer was required to perform was requested in a sequential manner. As one job was completed, the next was initiated and allowed to run its course; a strictly single-user, batch-processing procedure. Organization was totally managed by the operator.

As demands increased, computer technologists realised that the available hardware was not being exploited to the full, and the concept of multiprogramming was born. The software was organized in such a way that many jobs could be processed virtually "simultaneously", each being allocated a certain amount of processing time. Sometimes execution of a job would have

to be suspended, pending the availability of a particular peripheral. In such a case, the system would set aside this job and continue with another, instead of merely marking time while waiting.

As computers began to find their place in industrial applications, the use of multiprogramming became essential. In a typical industrial control application, there are many different functions to be attended to by the computer. An additional complication is the need to take account of "time". The process under control will normally be required to operate according to its own time requirements, and so the computer must be synchronized to "real-world time", and not to its own "computer time".

The requirements of an on-line process-control computer begin to emerge. The system has to cope with a large number of different jobs (or tasks) and must be able (in most cases) to relate the execution of these jobs to real-time. The initiation of the individual tasks will be related to a variety of situations. Tasks may be "requested" by external events or by the process operator, or may be activated by other tasks, in accordance with an overall operational strategy. Additional requirements may also be introduced. Modern man typically squeezes the last ounce of capability out of a system. The process manager will thus require the use of any possible spare computing power for the calculation of production figures, or the compilation of operating statistics, etc.

The organizational problem is evident - given the requirements of a process-control computing system, how are the hardware and software to be arranged, so as to achieve the goal? Can this question be tackled on a rational, scientific basis?

1.2 Analysing the Problem

In any organizational problem, there are two primary factors which must be considered. Firstly, there is the matter of providing the essential facilities necessary to implement the required functions. Secondly, there is the need to administer these facilities. In the case of on-line process-control computers, the essential facilities to be provided are easily defined as: a central processing capability, a storage medium to retain programs and data, and a means of inter-communication for the process, the operator and the computer. The pertinent question is the precise interrelationship between these facilities.

The answer is not a simple one, as each process will have its own particular requirements. It is possible, however, to establish generalized system requirements (based on current industrial experience) and to analyse these, so as to extract guidelines which may be used to formulate broad

organizational principles.

1.2.1 Essential Capabilities of a Process-control Computer System

In broad terms, a computer applied in a real-time, process-control situation must be capable of performing the following functions:

(i) The scheduling of a variety of tasks in such a way as to effect overall control of the process. (At this stage, a "task" is taken to imply a sequence of computer instructions which perform a predetermined system function). Each task will have a certain time which it takes to complete, and will require execution at certain intervals.

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

  (a) A signal derived from the 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, such as an event-time or a real-time clock.

(iii) The temporary suspension of a task currently in execution, following a request for a task with a higher priority in the predetermined, pre-emptive, priority hierarchy.

1.2.2 Methodology

The organizational problems of an industrial control computer are not unique. The situation has its parallels in many aspects of society. In broad terms, the underlying problem is simple to formulate - the processing of information, originating from a multiplicity of sources, using limited facilities. The system objective is efficiency. Consider the administrative centre of any large undertaking. The centre is created to provide for the functional needs of the rest of the organization. Each sector of the organization (including the administrative centre itself) will require certain tasks to be undertaken by the centre. Information is fed directly to the centre, and responses requested. The precise way in which the centre is organized to handle the incoming information and subsequently to process it, will determine the efficiency, assessed in terms of the benefit to the system's users. The key lies in the way in which the centre is organized.

The problem of information reception and processing is seen at its most acute in telephone exchanges. A typical exchange might have some tens of thousands of subscribers. Each subscriber wishes to be able to communicate at will with any other subscriber. This is technologically feasible

but economically unfeasible. The solution adopted is to provide a limited switching capacity, statistically designed to provide the most efficient service within sensible economic and practical limitations. Typically, an analysis is made of average traffic loads, and switching (or processing) capacity is provided to meet these requirements. In order to provide a scientific basis for this design procedure, much research has gone into the study of the characteristics of communication traffic.

The close parallel between this traffic organization problem and the organizational problems met with in multiprogrammed computer systems is most striking. In the former, the concern is with incoming calls, having random arrival times and random call lengths (or holding times). In the latter, there are incoming requests (which will be largely random in industrial control applications), each resulting in the execution of a task (which may also be regarded as having a random execution time). If it is assumed that, in the case of the traffic problem, there is only one server (or processor), and that, in the case of the computer, task-switching occurs virtually instantaneously, then it is feasible to apply traffic theory to the analysis of industrial control computer performance.

Communication traffic theory is based on a statistical evaluation of incoming call patterns and their resulting call times. An estimate of the resulting queue characteristics is based on these parameters. Essentially, if a call arrives and the processor is not free, the call must be placed in a queue to await subsequent processing. The same situation exists in the control computer. Requests, as outlined in 1.2.1, arrive, and their corresponding tasks are executed as soon as possible, under the guidance of a chosen scheduling policy. (For example, the queue-handling may be based on a first-come, first-served basis, or else priorities may be permitted. In the statistical work which follows, a simple queue of the first-in, first-out type will be assumed.)

The resulting characteristics of the queue are of paramount importance in the organizational structure. Two situations arise. Firstly, the queue might get so long that the subsequent delay time-in-queue tends to infinity, and the system will be deadlocked. Secondly, in most industrial applications there will be deadlines for certain tasks (i.e. certain tasks will have to be executed before a set time), or the process will collapse. While this situation may partially be alleviated by the careful allocation of priorities, the characteristics of the queue give a clear indication of the delays which may be expected.

1.2.3 Applying Traffic Theory to the Analysis of Industrial Control Computers

Appendix L sets out in detail the statistical background relevant to this section. References to the originators of the concepts used are quoted there. The basic premises are reviewed below, and relevant results given.

Consider a time interval T, and assume that within this time-span the average rate of requests is K (per time-span). The requests are characterized by a Poisson distribution. Considering a small interval of time, Δt, within T, the probability of the arrival of one request is $K\Delta t$ (Equation (7), Appendix L).

If h is the mean execution time of the tasks, it is proposed (Appendix L) that the distribution of task lengths may be characterized by a function

$$f(t) = \frac{1}{h} e^{-t/h}.$$

The probability of a task length lying within a time-span of t_o to $(t_o + \Delta t)$ is

$$\frac{1}{h} e^{-t_o/h} \Delta t \text{ (Equation (8), Appendix L).}$$

Using these functions, the probability of exactly one task being completed within a specified time interval, t_o, may be established as $\frac{1}{h}\Delta t$. (Equation (10), Appendix L).

The distribution of requests, and the statistical determination of the length of time which a task takes in execution, permit a prediction of the resulting queue characteristics. It may be shown that the mean queue length, L, is given by $L = \frac{Kh}{1-Kh}$ (Equation (12), Appendix L).

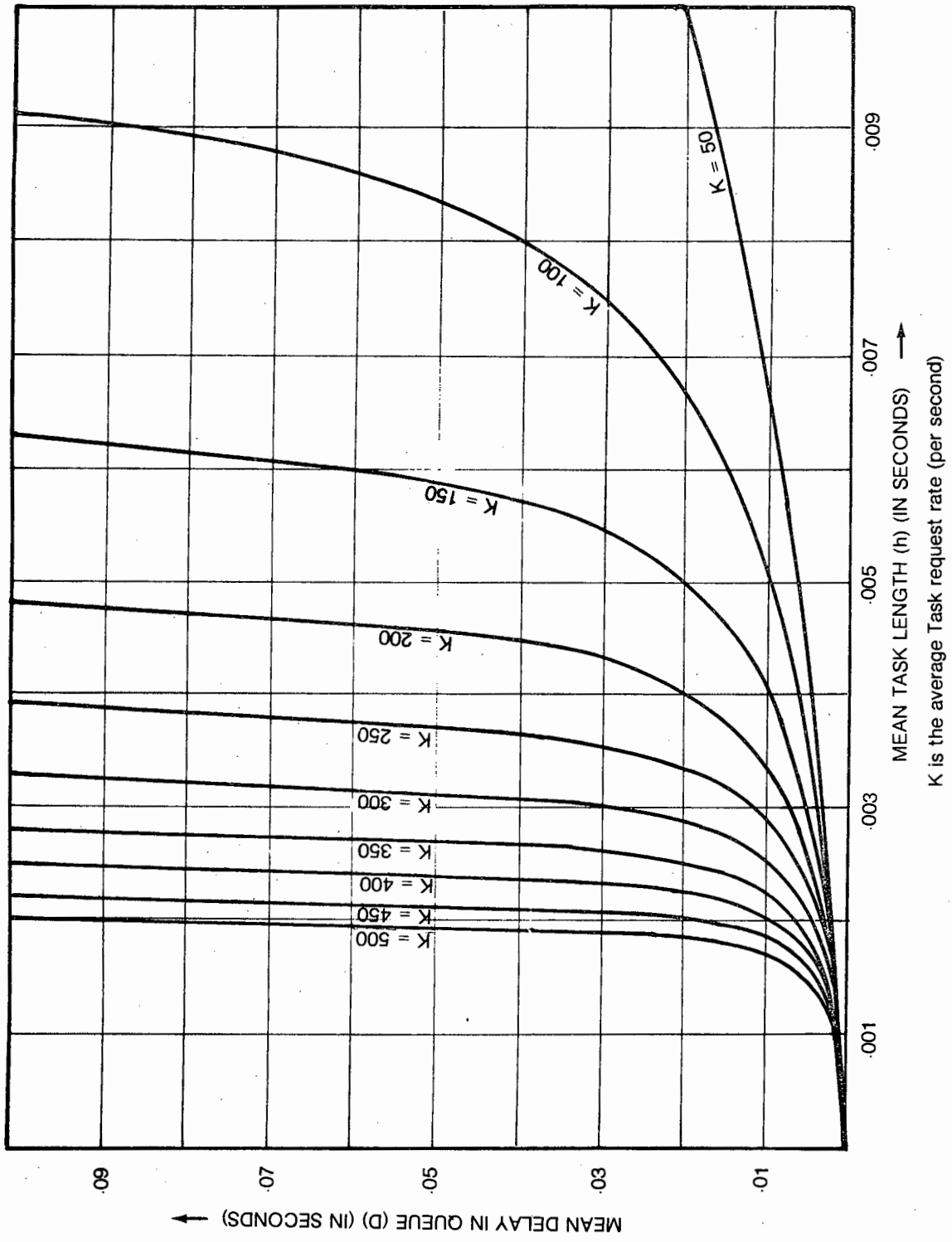
The mean delay, D, of tasks in the queue is given by $D = \frac{h}{1-Kh}$ (Equation (13), Appendix L).

The latter expression is of universal importance. It relates the mean delay directly to the system demand characteristics (i.e. the average request rate, K, and the average task execution time, h). This relationship forms the basis of the argument which follows.

1.3 Organizing an Industrial Control Computer

Graph No. 1 plots the expression derived for the mean delay experienced by a task in a queue which can form in a multiprogrammed computer system. In the most basic of terms, this gives the key to the performance of an industrial control computer, and points the way for proposing a suitable organizational strategy. Certain assumptions are made to permit this analysis, and the influence of these assumptions is discussed in the latter part of this section.

In order to predict the performance of the system, a clear indication must be given concerning the actual, practical process characteristics. These



GRAPH No 1: MEAN DELAY IN QUEUE AS A FUNCTION OF TASK CHARACTERISTICS

relate to the number of task requests to be catered for per time-unit, and their approximate execution times. If the averages of these parameters are applied to the queue delay function, D, the resulting performance may be evaluated.

A most important guide-line arises directly from the expression for D. Since $D = \frac{h}{1-Kh}$, if D tends to infinity, the system will be totally oversaturated, and will collapse. No tasks will ever get out of queue . This arises when $(1-Kh) \rightarrow 0$, i.e. when $Kh \rightarrow 1$.

Thus, the system overload occurs when

(AVERAGE REQUEST RATE) x (AVERAGE EXECUTION TIME) → 1

Of greater practical importance, however, is the necessity to consider the mean task-delay time which is permissible. In a slow chemical process, where time-constants are typically very long, delay times of minutes may be tolerable. In an on-line, computer-controlled rolling-mill, however, the maximum permissible delay times might be milliseconds.

Thus, for example, in the chemical process, average delay times of, say, 0,1 sec. may be permitted. If the average task execution time is, say, 0,005 seconds, 200 task requests per second may be included. In a rolling-mill, however, if the maximum permissible delay time for task execution is only 0,01 sec, and the average task execution time is 0,005 sec, then only 100 task requests per second can be permitted.

Other considerations to be taken into account include the question of priority. In a typical situation, tasks may be allocated priorities in order to ensure that they are executed within specified times. The effect of this allocation on the proposed method of analysis must be considered. If there are relatively few, high-priority tasks, the analysis is still valid, since their influence will be slight. If, however, the majority of tasks are allocated priorities, the picture can change dramatically. In effect, the problem resolves itself by forcing a higher request rate. This is because the arrival of a high-priority request will imply that the current task in execution must be pre-empted (i.e. suspended) and the new one brought into execution. The suspended task will enter the queue and await attention. This may, then, be viewed as an additional request. The situation can be seen to become highly complex and difficult to handle. The references cited in Appendix L suggest possible means of coping with the situation, but no clear-cut method has as yet emerged. This is undoubtedly an important area, and one which requires further attention. For the present, the conclusion is that priority allocation will tend to degrade system performance. Appendix H discusses means whereby a system may be organized so as to ensure in practice that all deadlines are met (assuming that the system is possible to organize in the first place).

The next aspect to be considered is that relating to practical problems met with in actually changing tasks, normally called "task-swopping". The analysis carried out above assumes an infinitely-small swopping-time. In practice (and as is clearly demonstrated in Appendix I) this is far from the reality. If all tasks are held simultaneously in the computer memory (and therefore do not require to be read from a bulk-storage device), the swop-time can be as large as 500 microseconds. In a case where the tasks must be obtained from a bulk-storage device, the time can be milliseconds long. The realistic situation may be catered for within the previous analysis, since each task which comes into execution will require a task-swop. Thus, the task-swop-time may be included within the estimates of mean task execution time. Thus, if the task-swop-time is t_s (which will be approximately constant for all task-swops), then the value previously used for the task execution time, h, will be given by $h = t_s + t_a$, where t_a is the actual average task execution time. The resulting effects are shown in practical cases by the figures given in Appendix I.5.

1.3.1 Organizational Strategies

In determining the actual organization of an Industrial Control Computer System, the designer has several options open to him. Having estimated his process parameters, he may look to the expression relating to average delay times for guidance. The average delay curves will define his starting-point, since he will know from the plant parameters what the permissible average delay time may be. This will give a clear indication of the constraints to be applied. (For example, if the average delay that the process can allow is 0,05 seconds, then a task request arrival rate of 150 per second may be allowed, provided that the average task execution time does not exceed 0,006 seconds).

Of particular interest is the question of suitable action to be taken if it is shown, from the above evaluation, that the desired objectives cannot be met by a conventional computer system. In addition, and this is a widely-experienced phenomenon, if a system is found to be operating poorly, or if there is a need to add certain new features (which require the execution of additional tasks), organizational improvements must be undertaken.

(At this point an illustrative example is worth considering. The author has been involved with the on-line computer control of a submerged arc-furnace. After a year of operation, it was decided to include a further section of the process within the control system. Initial tests on the system indicated that if this additional function were added, the system would be degraded so as to be totally incapable of meeting the overall requirements. A survey of the status quo showed that the system was, in its current form, organized in such a way that a slight increase in the average request rate

meant that the average task delay time became almost infinite. The system was operating at the breakpoint of a curve very close to the K = 150 curve on Graph 1. The problem was then to determine how to reorganize the system to meet the new demands.)

The curves in Graph 1 indicate the various avenues which might be taken, either to achieve a satisfactory system, or to improve an existing, inefficient one. The general objective will be to reduce the average delay times experienced. Two possible approaches may be made.

(i) Reduce the average request rates.

(ii) Reduce the average task execution times.

Each of these should be examined in depth.

1.3.2 Reducing the Average Request Rates

This amounts to reducing the demands made on the processor, and is the obvious solution which emerges from a cursory look at any inefficient system. (It might, however, be the most costly solution.) There are two methods of achieving this goal.

(i) Actual reduction of the demands. In other words, the inputs to the system are restructured so as to lessen the incoming traffic. This can be done by a careful examination of the system supplying the requests. Can, for example, a certain amount of pre-processing be carried out, so that only essential requests arrive at the processor? This solution can be seen in many conventional, large data-processing computers. A smaller processor is included as a "front-end" system, which takes many trivial tasks away from the main machine. A similar situation is emerging in the development of so-called "intelligent terminals". These are computer terminals which can perform a limited amount of processing. Instead of information being sent line-by-line to the computer, a large amount of information is collected by the terminal, verified and edited, and finally sent off as a complete block of "clean" data. (This situation is equally applicable to complex administrative functions and the working of committees, etc. Instead of a large committee being faced with a lengthy agenda and weighty supporting documentation, it is preferable to delegate the initial sifting and processing of information to small, efficient sub-committees. Only crystallized, concise information need then be sent on to the central body.)

In an industrial control computer organization this approach is most valuable. Certain control functions may be distinguished as virtually separate entities, requiring a minimum of communication with a central control computer. These control functions may be delegated to small computers, typically microcomputers, which operate virtually independently of the central machine. The central computer then performs a monitoring function, and will only communicate with the satellite controllers when absolutely necessary. The concept of "distributed" control computers is discussed in a paper by the author, cited as reference {109}.

(This solution was, in fact, adopted in the illustrative example quoted above. The supply of raw materials to the furnace was placed under the control of a microcomputer. The microcomputer then sent back predigested inventory information to the central computer. Any change in the specifications of the raw materials determined by the central computer would then be sent back to the microcomputer, which would implement the necessary alteration to its schedule of tasks. The system designed by the author is described in reference {110}.)

(ii) Decentralization. This embodies the division of labour between more than one computer, each being given a reduced number of tasks, resulting in a feasible overall organization. This is, basically, the classical solution: "If the system can't cope, double up on facilities!" There are inherent disadvantages to this approach. If total decentralization occurs, co-ordination of the workings of the separate entities can become a major problem. Also, in accordance with Parkinson's principles, the workings of each entity will expand to fill its capacity. If, however, the various entities are organized to be in constant communication, and hence (hopefully!) to co-ordinate their workings, the situation can be reached where this information flow between the bodies is so great that more time is spent in performing this function than in executing the specified tasks! For this reason, this approach is not favoured.

1.3.3 Reducing the Average Execution Time

Two aspects must be considered here. Firstly, an attempt should be made to streamline the tasks requiring execution. In many cases, a close examination of the status quo will indicate that certain tasks are unnecessarily

complex. (This, in computers, raises some fundamental points about the philosophy adopted in programming. The use of high-level languages, for instance, will generally lead to relatively inefficient machine-code programs. The question, then, is whether it would not be more expedient to create programs directly in machine-code (a controversial issue!). Or, arithmetical processing, being normally time-consuming, might be reduced by the addition of extra hardware, or by the use of fixed-point arithmetic instead of floating-point arithmetic, or by sacrificing arithmetical accuracy.)

Another relevant matter is the question of task duplication and task combination. It is often possible to identify areas in which tasks are duplicated. Also, instead of requesting separate tasks, two similar tasks may sometimes be condensed into one. Careful and intelligent system analysis can assist here.

A second possibility relates back to an earlier point, the question of the time taken to change tasks. It will be recalled that the average execution time of tasks was taken to include swop-time. It is possible (and this dissertation bears out this statement) to reduce the actual time involved. Initially, one method seems the most obvious. If all tasks are stored in main memory instead of in a backing bulk-storage device, it will be inherently (and, in practice, up to 10x) faster to effect a task-swop. With the rapid decrease in main memory costs, and with the introduction of mini-computers with large word-size (and, hence, greater addressing capabilities) this improvement can be achieved. This also recalls previous comments on the production of shorter, more efficient programs.

Secondly, with suitable hardware, it is physically possible to reduce the swop-time. This will be discussed in depth later.

1.4 The Way Ahead

The problems involved in organizing an industrial control computer system have been analysed above. The analysis has indicated that many avenues may be taken to achieve a powerful, yet efficient, system.

Of great importance in any system's organizational solution is the cost. Many of the methods proposed above are inherently costly to implement. For example, a dual-processor system will imply double the initial outlay. A multiprocessor, distributed system will require the acquisition of many computing systems. (This might, however, be counterbalanced by the reduction in the cost of routing large amounts of information).

This thesis proposes to tackle the problem from an unconventional standpoint. It has been mentioned above that:

(i) Preprocessing of information is desirable, and

(ii) Swop-time reduction is important.

A further point, the question of management, should be introduced.

Any system, not withstanding the adoption of ideal organizational methods, requires supervision. In computers, this is exercised by an "operating system". In the analogous telephone switching situation, this supervision is achieved, in the main, by hardware. In computers, it has classically been implemented by software. Would it not be possible to bring together these three ideals - pre-processing, swop-time reduction, and hardware supervision?

This dissertation argues that this ideal is possible. What is required is a close examination of the fundamental principles on which classical control computers are based, followed by the adoption of an integral hardware/software approach, utilizing current technology to the full.

1.5 The Relative Roles of Hardware and Software in Computer Systems

A hardware-software trade-off establishes the division of responsibility for performing system functions among the software, firmware and hardware {5}. This must form one of the most essential components in the definition of a computer's organization. Three types of trade-off are apparent:

(i) Software-to-Hardware Trade-off

This is a matter of deciding which functions are to be performed by hardware and which by software. It may be thought of as virtually establishing the machine-code instruction repertoire of the system. The more powerful the instructions are made, the more complex will be the hardware that is required to implement them. A simple example is the computation of a square root. A computer with a relatively unsophisticated instruction set would have to perform this function by a series of primitive "add"- and "shift"-type instructions, obeying some well-established algorithm. A suitable hardware unit, however, could be included in the computer so that a simple, primitive instruction, "SQRT", would activate the unit, and the result be automatically calculated. This approach may be seen in the emergence of units which perform such tasks as hardware floating-point arithmetic and memory mapping.

Historically, this trade-off has been governed by economic considerations. Logical techniques have been available to implement a wide range of arithmetical operations by means of hardware. Because of the high costs of hardware, and an inherent decrease in reliability, advances in this area were restricted. With the technology now available, however, the trade-off may be examined in a far more rational light.

(ii) Software-to-Firmware Trade-off

Many modern computers are microprogrammed by a variety of methods generally based on the original Wilkes model {6}. This leads to the feasibility of implementing often-used functions by micro-coded sub-routines in the computer, responding to specific macro-instructions. In many cases, this might not speed up the overall computational time, but it does lead to fewer memory-fetch cycles, and to greater ease of programming. Since the microprogram has access (in most cases) to far deeper control of the computer's facilities, however, it can lead to greater efficiency and the implementation of functions not otherwise possible. The use of this trade-off is finding its place in the development of firmware to execute complex functions such as often-used language-processor statements.

(iii) Firmware-to-Hardware Trade-off

This area of trade-off is analogous to the software-to-hardware situation. It is a question of how complex the possible microinstructions are to be: whether to use a series of firmware statements with relatively simple hardware, or more powerful microinstructions with more complex hardware. The versatility of the resulting hardware is the major factor to be considered here.

As far as total system costs are concerned, during the past two decades the proportions spent on hardware and software have been reversed. In the early 1960's, hardware accounted for over 50% of total system cost. Today the software accounts for well over 50% of the total, and this percentage is rapidly rising {7}.

The reasons for examining trade-offs may be summarised as follows:

(a) Improvement in System Performance

This may be demonstrated by considering two well-known system-improvement methods. Firstly, in many applications the use of the Fast-Fourier algorithm is required (e.g. in signal-processing). In a classically-structured computer, the time required to process such an algorithm is lengthy and, in the on-line processing of data, may be prohibitively long. The addition of a hardware unit, such as that described in {8}, however, makes feasible "almost" real-time operation. Secondly, the addition of special Input/Output processors can relieve the central computer of much time-consuming monitoring, formatting and general control of peripherals. In the minicomputer area, many manufacturers {9} are realising the potential of including greater intelligence in their peripheral controllers.

(b) Reduction of Software Complexity

This is attractive from the viewpoint of both the manufacturer and the end user. For the manufacturer, the production of a computer with more powerful, less complex software implies easier implementation of the essential system software which must be written in order to market his product. At the user's end, the simpler a machine is to program, the more attractive it will be. This is clearly borne out by the inroads being made into the traditional computer market by the so-called "user-orientated" programmable calculators, in which the user can control sophisticated hardware by instructions which are simple to learn and understand {10}.

Another area in which software complexity may be reduced by introducing additional hardware is that of automatic memory management. This relieves the user of the problems of limiting his program to available memory.

(c) System Reliability

Regular software failure occurs in the accidental overwriting of programs, and software faults are introduced by frequent changes. Committing often-used routines to hardware will lead to greater security. This, however, does raise the problem of a reduction in flexibility, which must always be taken into account when considering committing functions to hardware.

For the designer, such hardware implementation will impose a far higher degree of verification: a routine committed to hardware is inherently more difficult (and costly!) to correct than the corresponding software routine. It is worthwhile remembering, at this point, the famous statement of Dijkstra {11}:

> "Program testing can be used to show the presence of bugs, but never to show their absence!"

(d) Improvement of Fault-monitoring Aids

Checking for errors by means of software leads to high overheads in terms of the additional routines which must be introduced. In many cases, the fault monitoring may be implemented by hardware {12}. Memory protection is one example of this approach which has been widely adopted {13}. Another example is the use of error detection and correction in many modern computer systems, particularly in the memory organization {14}.

(e) System Expansion and Updating

The life-span of a computer system is largely related to its ability to adapt to new situations, i.e. to extend to new applications which may demand new facilities. If the machine is microprogrammed, new features may be relatively simply implemented. From a marketing standpoint, the ability of one machine to emulate another can be of great value. This can allow software from the second machine to be executed on the first. Such an emulation can be achieved by suitable software, but with an obvious execution-speed penalty. Microprogramming can largely solve this problem. In addition, in considering a whole family of computers from one manufacturer, if the machines are microprogrammed, upward expansion of the family (i.e. the inclusion of additional features) becomes easier to implement.

From the discussion, so far, it becomes clear that hardware-software trade-offs are of vital consideration in computer architecture. In the computer industry, however, the design engineer is not always free to examine the situation rationally. The high cost of research and development implies that manufacturers are reluctant to make major modifications to their tried and trusted systems. So minor enhancements are the order of the day, with as much salvaging of current hardware and software as possible. Thus, major changes in architecture tend to come from new entries into the computer market. An interesting example is the emergence of the now famous Intel Corporation which was founded by designers who found their previous employers (a very large semiconductor manufacturing company) reluctant to launch out into a revolutionary area, which promised a great future but involved a high risk factor {15}.

1.6 Methodology in Evaluation of Hardware-Software Trade-Offs

It is apparent that an integral hardware-software approach is necessary to formulate the architecture of a computer system. Contemporary systems can usefully be described as a 'superposition' of physical and logical media {4}. The physical medium is the hardware which forms the host environment in which the logic (i.e. the software) operates and implements the desired task. It is thus obvious that the hardware and software form a structure which is a single entity and which sets out to solve the specified problems. Any examination of the hardware-software trade-offs, therefore, must consider the overall picture. The objects of an evaluation, then, may be stated as follows:

(i) The simplification of design, construction and maintenance.

(ii) The creation of an environment which is more amenable to the testing and maintenance of major system software.

(iii) The production of a machine language which is more procedure-orientated.

(iv) The production of a system which lends itself to a high utilization of available hardware.

(v) The design of a modular system which will lend itself to simple modification to meet changing circumstances.

It is interesting to note that as technology tends to reach the limit of its physical capabilities (after all, electrical propagation cannot exceed the limits of the speed of light), the designer's object must be to achieve a higher throughput, not in terms of mere speed, but in overall system performance. This, in essence, is summed up by (iv) above (high utilization of hardware), and is the object of parallel processing of information.

Barsamian and De Cegama [4] have produced an interesting attempt to reduce the trade-off problem to mathematical terms. Their approach is to model a system mathematically, and their conclusions indicate that more effective systems can be designed by increasing the logical complexity of the hardware, and by the extensive use of microprogramming.

1.7 Current Trends Towards Software Replacement

Current trends towards the replacement of software appear to be taking two directions. The first is that in which certain select categories of software are implemented either by microprogrammed routines or by a specialised hardware unit. A typical example is in the execution of floating-point, arithmetical functions. The second, and more dramatic, area of interest is that of the design and implementation of integral computer systems. Here, the concepts of the von Neumann structure are adapted to produce essentially hardware-orientated systems.

A classic example of the second situation is the development of the "SYMBOL" system at the Fairchild Laboratories [16]. The Fairchild researchers were convinced that the potential computing power of hardware had not been fully explored, and decided to create an experimental working system to investigate the limits of the functional capabilities of computer hardware. Six years of basic design and implementation led to the installation of a prototype system at Iowa State University. The primary object of this whole exercise was to show that a high-level, general-purpose procedural language, and a large sector of the time-sharing operating system, could be implemented in

hardware, while achieving a significant increase in overall computational rate. A general-purpose language was created, based on current high-level languages. The language would have to perform all essential operations on application problems, without being cluttered with machine-dependent operators. No major restrictions were placed on the language or on the hardware developed to match it. The resulting system contained a wide range of features implemented solely in hardware. These included Dynamic Memory Allocation, Automatic Virtual Memory Management, Hardware-implemented Time-sharing Supervision, and Direct Hardware Compilation.

The experience on this system led to two important conclusions: firstly, that a high-level, general-purpose procedural language and a high percentage of the time-sharing operating system can effectively be implemented directly in hardware; secondly, the use of such a system showed that significantly fewer hidden rules exist to plague the user! It must also be pointed out that the system did have one major disadvantage - lack of flexibility. This is simply because it is easier to modify software than hardware. The implication is that computers which are designed along the lines proposed by the SYMBOL system will tend to be special-purpose. The final solution, then, appears to lie somewhere between the two extremes.

This middle-of-the-road solution leads to the concept of a powerful, hardware-orientated structure which is totally microprogrammed. Two examples of this structure are worth discussing. The Burroughs B1700 system {17} was designed with this in mind. It is basically a powerful, hardware-implemented system with total variability, giving the appearance of having no inherent structure. This implies that any definable language may be implemented. In essence, the B1700 premise is that "the effort needed to accommodate definability from instruction to instruction is less than the effort wasted from instruction to instruction when one system is used for all applications." Important features of this system include: Microcode execution directly out of main memory (this is normally buffered via a fast-access cache memory), microprocedures which are re-entrant and re-usable (each processor in the multiprocessor structure including a 32-deep stack, stack operations being automatic), microprograms which are not limited in size, and concurrent hardware execution of various sections of the interpreters.

A second example is the emergence of powerful hand-held calculators. These are strictly microprogrammed computers, appropriate micro-coded sequences being selected by means of keystrokes. In essence, such calculators are totally hardware-orientated computers, made possible by the technology of microcomputers.

1.8 Hardware Implementation of Operating Systems

It has been mentioned (in section 1.7 above) that one of the features of a typical computer system which appears to be particularly amenable to hardware implementation is that of the operating system.

This area is currently being investigated by various research bodies: a typical example is the work being carried out at Purdue University, based on a Micro-data 1621 [18]. The features of this system include the firmware implementation of typical functions such as input/output channel allocation, file opening, CPU control, etc.

From the summary of a workshop sponsored by the IEEE on the interaction of operating systems and computer architecture [18], it becomes clear that one area which has not been widely investigated is that of hardware-controlled scheduling of a single processor. This area appears very attractive, offering as it does the replacement of necessary but repetitive software by relatively simple hardware. The concept becomes increasingly viable when applied to the area of real-time process control. A minicomputer is usually used in such environments to achieve a high degree of reliability, reasonable arithmetical accuracy, and rapid response. Users of such systems do not normally want to be concerned with complex, difficult-to-use operating systems. The objective is in the application. Typically, such a system will require sophisticated interrupt-handling capabilities. In terms of a software-based operating system, this requirement imposes a high overhead in terms of central processor computing time which must be made available to deal with the interrupts. The solution seems to be to delegate these functions, that is, the interrupt handling, plus the associated scheduling problems, to a hardware unit which will leave the central processor free to deal with the process-control tasks.

It may rightly be argued that a better solution would be to implement the process-control system by means of a multi-computer arrangement. This could effectively achieve the objects mentioned above. Two points tend to detract from this - firstly, the problems of having an additional computer to program and maintain and, secondly, the cost involved. As will be seen in the system proposed and evaluated in this thesis, a relatively low-cost, reliable unit may be used instead of the second computer. It is also shown that by implementing the "operating system" controller on a microprogrammed basis, the "operating system" may be suitably modified to meet changing circumstances. This reduces the disadvantage of a hardware-based system in so far as lack of versatility is concerned.

1.9 Summary

"Theories of rational choice generally have not made a distinction between continuation of an existing program of action and change in program of action. In these theories, the chooser is simply confronted with two (or more) alternatives of action and required to select the better of the two." {2}

The designer of an industrial control computer system is faced with many fundamental problems. The most significant is the determination of the computer system organization. He has generally to be guided either by the computer vendor (who, naturally, has vested interests), or by past experience. In the latter case, the relevant technology is so new that suitable experience is not always available, or even particularly relevant. Vast ranges of hardware and software are available - how are these best organized into an efficient system?

This chapter has shown that a scientific approach may be adopted. The theoretical backing for this approach comes from the world of communication, specifically the study of traffic in a switching system. It has been shown how this approach permits the system designer to evaluate the projected organizational strategy, and how the desired objectives may best be met. Of fundamental importance are the conclusions which may be reached by a close study of the relationships between the expected delays occurring within a system, and the demands made on that system. These conclusions clearly indicate the steps which may be taken to provide efficient organization. While these studies are directed towards industrial computer control, the conclusions are universally applicable to systems in general.

The actual management function within a computer system is performed by the operating system. It has been shown that it is indeed the management function which determines the overall system efficiency. If the operating system can:

(i) Limit the amount of information directed towards the central processing facility (i.e. pre-process);

(ii) Switch between tasks in a very rapid manner, and

(iii) Generally function in a direct and unobtrusive fashion,

then efficient organization will result.

To meet these goals, it is necessary to examine the basic structure of current industrial control computers. The essential roles of the hardware and the software of a computer have been considered, and the trade-offs made during the formulation of the architecture of a system, reviewed. Then, based on an integral hardware/software approach, a suitable management technique has been proposed.

CHAPTER II

DEFINITION OF A REAL-TIME OPERATING SYSTEM AMENABLE TO HARDWARE IMPLEMENTATION

"Just the place for a Snark! I have said it twice:
That alone should encourage the crew.
Just the place for a Snark! I have said it thrice:
What I tell you three times is true."

Lewis Carroll, *The Hunting of the Snark*
(Fit the First)

Chapter I has provided an insight into the direction to be taken by the investigation set out in this thesis, namely, the replacement of a major section of a conventional, software-implemented, real-time operating system by means of hardware. Before setting out to define a possible hardware structure, the fundamental processes to be performed by this structure must be determined. It must be stressed, however, that the approach is an integral one, and that the final specifications of the system must be determined while taking into account the potential, and the limitations, of hardware.

In order to provide an orderly discussion of the structure of real-time operating systems, the text which follows begins with a brief review of fundamental operating-system concepts. These are then treated in the context of real-time operating systems, with the emphasis on contemporary systems, as used in minicomputer applications. These discussions provide the background information from which may be developed a suitable structure amenable to hardware implementation.

2.1 Operating Systems - A Background

In broad terms, an operating system is that essential part of a computer system which controls the "flow" of "jobs" through the computer and its associated peripheral equipment. The term "job" is used very loosely in this context. It may be more strictly defined, according to Denning [19], as a "process", being the abstraction of the activity of a processor. Essentially, a "program"

is a sequence of instructions, whereas a "process" is the sequence of actions performed by the program. "A program is static: a process is dynamic." {20}

The term "process" is equivalent to a "task" {21}. In view of the use of "task" by most authorities working in the Real-Time Operating System Field, this term will be used throughout this dissertation.

An overall computer system may be defined completely by the control functions it makes available to its users. The essential control functions of a system are to initiate a task, to control the flow of the task as it progresses along its prescribed path within the system, and finally to remove the task. Within this system, certain control functions may be designated. These may include: the allocation of hardware resources to tasks as and when required, the provision of access to the system's software resources, the control of error conditions which may arise during execution, the provision of protection and security where necessary and, in the case of a system in which many tasks may be simultaneously active, the provision of synchronization and communication between tasks. These management functions are classically implemented by means of an operating system.

The degree of complexity of the operating system varies according to the needs of the application. In this work, the concern is with real-time operating systems used in minicomputers, with particular reference to process-control applications. In such situations, the application requires an emphasis on speed of processing, and core storage usage. The system is normally constructed from the "bottom up", i.e. it provides the bare essentials required, with expansion of facilities where necessary. The user cannot tolerate unnecessary overheads in terms of features which are not of direct use to him.

The priority in all real-time systems is multi-programming. The concept here, according to Dijkstra {22}, is one in which many tasks operate in parallel. Unless the computer system incorporates actual parallel processors, of course, as is now being implemented in the design of many large mainframe systems, the real meaning is that the tasks are permitted to use the facilities of the system in response to various "signals". Such signals may include interrupts, internal real-time clock indications, or internal rescheduling determined by the operating system. This scheduling of tasks may be based on various systems, such as:

i) A "round robin" system, used in early time-sharing, in which each task is allocated a certain period of time.

ii) A priority queue system, in which the processor, when available, will execute the task with the highest priority.

iii) A shortest-elapsed-time and shortest-processing-time system, in which the tasks that have used the shortest processing time since arrival, or which have the least execution

time, are executed first. In terms of actual application of real-time operating systems, the scheduling is normally governed by the priority of tasks, since certain tasks which are of great importance (based, say, on an urgent interrupt request), must be executed regardless of the present scheduling arrangements. (A classic example, and a useful test of the system's capability, is the time that the system takes to respond to a power-fail condition).

A feature of current minicomputer real-time operating systems is a foreground-background mode of operation [23]. In such a situation, the real-time control functions are run as, say, foreground tasks. These are essential tasks which must be executed in real time in response to interrupts, forced scheduling, etc. When the computer is free, that is, has no more essential tasks to execute, it falls back onto non-time-critical, low-priority tasks, which are held as background tasks. These normally include the compilation of reports, language processing, editing, etc. The important aspect to consider is the speed at which the computer system can switch back to foreground operation when required.

2.2 The Structure of a Minicomputer Real-Time Operating System

A real-time operating system will generally consist of an executive program and a suite of utility programs. These utility programs, which are themselves tasks in terms of the definition, would normally be concerned with file-handling, input-output, etc. As mentioned previously, in order to optimize the system, these tasks are kept to a minimum. In general, the utility tasks may be treated as similar in nature and execution to the user's tasks. The executive is therefore of prime importance in this work. It is precisely the executive which forms the heart of the operating system, and it is the executive which is, in fact, replaced by hardware in the system proposed here.

.2.1 The Role of the Executive

The executive is the element which performs the management function. In broad terms, the executive is required to perform the following functions:

i) To schedule the tasks: in plain words, "to decide what must be done next". In practice, the scheduling will be based on certain constraints which resolve the question of priority. The user, when configuring a system, will decide on the priority of each task, when it must be executed, whether it may be superseded by another task, and how it may be requested. A task may be initiated at a specified time, by an external interrupt, by another task, or simply whenever the processor is free. It should also be possible for the priority of tasks to be dynamically changed, if and when necessary.

ii) Arising from (i), to handle the interrupt structure. A variety of interrupt conditions, such as external signals, real-time clock signals, etc., may arise. The interrupt handling may be greatly simplified by introducing hardware techniques, such as automatic interrupt vectoring [24] and interrupt masking. In a process-control situation, the interrupt capability of the modern computer is widely exploited to handle alarm signals, to synchronize external process events, and generally to provide a means of external control over the system. It is worth remembering this fact, since the handling of interrupts in practical applications is virtually a component of the operating system, and is a critical area in the total system. This point will be reiterated in the chapters below.

iii) To communicate between processes. The executive must provide the means of linking tasks, for invariably one task will require access to another. For example, a specific task might require that data be sent to a peripheral, which will have a suitable handling task associated with it. The requesting task will require the executive to set up the second task and to arrange the passage of information between them. This data transference is usually implemented by means of a buffer, either in-core or on a back-up storage device. The first task will write the information into the buffer and the second task (when instructed to do so by the executive) will access this data. This process of inter-task communication presents interesting problems. Dijkstra [22] presents the semaphore as the basic means of signalling between two tasks - a semaphore being essentially a stop/go flag, capable of being tested and/or set by more than one process. The semaphores are acted upon by two operations, the so-called P- and V-operations. The P-operation, or "stop" operation, sets a semaphore to a stop state, and will suspend a calling task if it has been stopped. The V-operation will start a task which has been stopped when the semaphore occurred. It will also set the semaphore to a "go" condition if no process was stopped. This amounts to code words which relate to the communicating processes, and which will hold up both tasks and determine the task to be continued. Reference [25] deals in detail with the use of this concept and its wider applications.

2.2.2 Implementation of a Suitable Real-Time Operating System

In order to implement a real-time operating system, it is necessary to provide a concise mechanism for the specification of tasks. One convenient approach is the allocation of process control blocks (or PCB's). A PCB contains all relevant information pertaining to a task in a one-to-one fashion, i.e. each task will be allocated its own PCB. Data held in this block includes all information which the executive needs in order to schedule or initiate a task. So a typical PCB will contain priority information, current status of the task (whether active, suspended or in a waiting queue, etc.), and linking information relevant to the initiation of the task (such as required contents of processor registers, overflow indicators, etc.). With the information held in the PCB, the executive is in a position to know when to execute that task, and what parameters must be set up to initiate the actual processing of the task.

The basic functions of the executive are typically implemented by means of sets of sub-routines, or what have come to be known as "Virtual Instructions". These virtual instructions may be regarded as forming the instruction repertoire of the executive in the same way as the machine-code instructions form the repertoire of the actual computer. Typical Virtual Instructions may include:

i) Interrupt Handler. This would react to interrupts, identify the source, and take the appropriate action. It might be used to achieve a variety of operations, such as the suspension of a current task which has requested (via an interrupt) access to another task and is waiting for that task to become available; the entering of an interrupt-requested task into a queue to await attention; the immediate activation of a high-priority task; and the control of interrupt-masking.

ii) Control of Semaphores. These virtual instructions will implement Dijkstra's {22} idea of semaphores. They operate on a semaphore specified by a requesting task, to perform the P- (stop) and V- (go) operations on two tasks.

iii) a) Send Operation. This will, if necessary, attach a buffer to a queue, and activate one task if another has been suspended.

b) Receive Operation. This will, if necessary, take a buffer from a queue, and suspend a task.

Essentially, these operations control the stopping of a task, the activation of a new task, and the associated buffer handling. They are also used to control queues, for both tasks and buffers.

With these ideas in mind, the path is clear to move on to describe the essential constituents of the executive. These may be summarised as follows:

i) Initialization. Means must be provided to set the executive to its initial condition. This will involve the setting of the contents of PCB's, buffers, queues, etc. The system must be inherently self-initializing so that recovery is rapid in the event of forced restarting. In practice, this will require the designer to determine a suitable initial starting-point for the system. In the case of failure, the system would also have to be capable of initialization to a point at which it is suitable to return the process to the pre-failure situation. In process-control applications, particular attention must be paid to this problem, taking into account the operating strategies of the plant being controlled.

ii) Interrupt Handling. As pointed out above, this must be able to identify interrupts, activate specified processes, and fit in with the general scheduling policy. It must take particular care of priority specifications.

iii) Scheduling. This is obviously the central controlling mechanism, and the concepts are the same as outlined in Section 2.2.1. It is interesting to note, however, that according to Purser and Jennings {26}, "Contrary to experience on large machines, where there are big overheads on changing processes, and a finely-tuned scheduling algorithm can be very important, many mini-systems, particularly core-resident ones, are insensitive to scheduling strategy, provided that no obvious stupidities are committed". (It is, however, the author's opinion, based on observation of some current minicomputer real-time operating systems, that many "obvious stupidities" are, in fact, regularly perpetrated!)

iv) Virtual Instructions. As outlined previously, these are privileged instruction sequences which are accessed only by the executive itself, or (in certain cases) by global sub-routines. In practice, they are usually entered by means of interrupts, operator commands, or certain trap instructions from an executive's run-stream. It is worth noticing the wide use of such virtual instructions in many contemporary operating systems (for example, the system commands in Data General RDOS {28}).

The question of re-entrancy and interruptability of virtual instructions (and, in fact, of most component parts of the executive) is worth considering. In general, the question is largely related to the hardware structure of the computer being used. The availability of adequate stacking facilities will make it far simpler to provide for re-entrancy and interruptability. It is also largely a function of the ability of the programmer!

While each section of the executive can be clearly identified, certain real-time operating systems make extensive use of global sub-routines. These are created chiefly to ease the task of the users of the system, and are sub-routines which may be employed to control complex operations. For example, a global sub-routine such as "READ" may be included. A call to this sub-routine would allow the user to read information from a particular device. It would thus access the necessary Virtual Instructions and set the required semaphores, etc., to achieve this action; that is, it sets up the necessary "run-stream" to the executive.

2.3 Definition of the Real-Time Operating System to be Implemented in Hardware.

Bearing in mind the basic concepts discussed as embodied in the design of real-time operating systems, the features of the system which it is proposed to implement in hardware may be defined. In undertaking this task, it is worth drawing attention to an observation made by Robert McClure at the 1975 conference of the IEEE Computer Society. In a discussion on the outlook for hardware, firmware and technology, Dr McClure observed that there is a positive trend towards "the construction of simpler systems, a recognition that operating systems need not, or should not, be as complex as they have been" {27}. It is the author's contention that in many cases operating systems, and especially those developed for low-cost minicomputer systems, have grown in an unco-ordinated fashion, with modifications and additions to meet new situations. For example, a review made by the author, over the last three years, of the Data General RDOS (Real-Time Disc Operating System) {28} has clearly indicated that the system as presently marketed, was not totally conceptualised or designed from the start. As one reads the specifications, it becomes obvious that RDOS consists of extensive modifications and extensions, many in direct response to features found in competitive systems. The classic reply to this statement is, "We learn and improve" {29}.

Thus, while it is important to use past experience gained, and to apply the relevant, highly-developed theory when a new system is to be defined, the requirements should be appraised without bias. The final system adopted must (especially in situations where bottom-up methods are to be applied) be tailored to meet the precise demands of the range of applications. The system developed in this project is extremely simple but, as will be demonstrated, it fully meets the requirements of the application.

It must also be pointed out that this thesis does not propose a revolutionary approach to the theory of operating systems, but develops a new method of implementing an operating system. Thus, the operating system developed in what follows is by no means a fully comprehensive system, but comprises a basic structure, tailored to hardware implementation, which (a) shows that the overall philosophy is viable, and (b) provides a basis for further development.

One should remember, too, Barron's comment: "An elephant is a mouse with an IBM-operating system" {21}.

2.3.1 The Desired Operating System Features

The functions which the real-time operating system must be capable of performing may be summarised as follows:

(i) The total scheduling of multiple tasks. The tasks may be real-time dependent and priority-based, or non-time-critical, background tasks. The operating system must also be capable of dynamic rescheduling.

(ii) Multiple Interrupts. The interrupts are to be fully vectored to increase processing speed.

(iii) Intertask communication and synchronization, including buffer-passing between tasks.

(iv) Control of all input/output operations.

(v) Communication between user tasks and the executive.

(vi) Parallel task processing.

(vii) True foreground/background operation.

.3.2 Structure of the Proposed Operating System

Two factors play important roles in defining the structure of the operating system proposed in the studies which follow. Firstly, the operating system must be versatile enough to demonstrate the variety of differing applications to which the concept proposed here may be applied. The second and, from the engineering point of view, more important factor is that the operating system is developed with the hardware structure (discussed in the next chapter) very much in mind. While this does lead to the adoption of some rather unusual features, the ultimate aim is to achieve a high degree of hardware/software interaction efficiency.

The proposed operating system is essentially an event-driven system, since the user does not have to execute special system calls to effect rescheduling of the defined tasks. In the system, the "tasks" are to include all user tasks as well as all required input/output routines. The operating system will ensure the maintenance of the highest-priority task capable of being in execution. This task, i.e. the highest-priority "ready" task, will continue in execution until one of several conditions arises:

(i) The task terminates its own operation.

(ii) The task is suspended while it awaits the completion of an event which it has requested, such as an input/output operation.

(iii) A higher-priority task becomes capable of being executed.

Tasks are regarded as being in one of three possible states:

(i) Active - in execution.

(ii) Suspended - awaiting attention (possibly in view of priority, or while waiting for another task to be completed, or while being delayed for some other reason).

(iii) Dormant.

Each task is assigned a unique identification number and a priority. The task identifier will include a pointer which indicates where that task's "Process Control Block" (PCB) is held in memory. The PCB holds all relevant information pertaining to that task, including whether the task is interruptable or not. This is an important consideration, as even some low-priority tasks may not be interruptable once their execution is started. This rather anomalous situation often arises in process-control applications, where a sequence of commands to a particular piece of equipment must be completed once started. Obviously, such conditions must be very carefully controlled so as not to hold up high-priority tasks for any significant length of time. Other information held in the process control block is related to starting the actual execution of the task, i.e. the re-entry address, register contents, overflow indications, etc.

In order to cope with tasks being suspended or awaiting execution, a so-called "wait-queue" is introduced. Each task which is waiting for execution is placed in this queue, the queue being organised in such a way that it is always arranged in priority order. When any task is placed in the queue, the entry is made in the appropriate priority position. The top entry in the queue will thus always be the next task to be executed. In order to simplify the detection of the presence of tasks in this queue, a flag, denoted the "wait-for-service" flag, is defined. The value of the WFS flag indicates the number of entries in the queue.

An additional table is introduced, the Time-Dependent Tasks Table (TDT). This table contains the identification of all tasks which are to be activated at specific time intervals, or after certain delays. Again, a flag (the TDT flag) is used to indicate the presence of entries in this table. Such an entry will indicate the task's identification and the time at which it must be activated.

The actual re-scheduling of tasks is initiated by interrupts to the operating system. The interrupts are divided into four classes:

(i) Class I: Real-Time Clock Interrupts. These are generated by the system's real-time clock(s).

(ii) Class II: Process Interrupts. These are interrupts generated either by the actual physical process under control, or by the computer's own set of peripherals.

(iii) Class III : Computer-driven Interrupts. These are interrupts which are created by user tasks or by system programs in which they are used for intertask synchronization. In user tasks they may indicate end-of-task conditions, or express requests for communication with other tasks.

(iv) Class IV : Power Fail/Restart Interrupts. These are generated by peripheral circuits to indicate a power-failure occurrence, and the subsequent power-on condition.

The principles outlined above form the basis on which the proposed operating system is developed. In classical terms, they may be thought of as forming the "executive" of the operating system. As mentioned previously, it is this part of the operating system which is dealt with in this work. Other facilities encountered within real-time control systems, such as memory allocation and file management, are regarded as falling into the category of support software, and are not included in the executive itself. In practical terms, they are implemented as system tasks in the work which follows, with only the necessary bit-maps, memory protection registers, etc., being held in the executive.

2.4 Summary

This chapter has covered a brief review of operating systems in general, and of the requirements and principles of real-time operating systems in particular. On the basis of these requirements, the fundamental definition of a specific, real-time operating system has been developed, and a suitable structure proposed to implement these concepts. The proposal must be viewed with direct reference to its ultimate object, the implementation of a practical, real-time operating system in hardware.

CHAPTER III

THE DEVELOPMENT OF A SUITABLE HARDWARE STRUCTURE

"For the Snark's a peculiar creature, that won't
Be caught in a commonplace way.
Do all that you know, and try all that you don't:
Not a chance must be wasted today!"

Lewis Carroll, *The Hunting of the Snark*
(Fit the Fourth)

It has been claimed above that the real-time operating system previously discussed is amenable to hardware implementation. Can, in fact, a physical structure be developed which will permit such a concept to become a reality?

It may correctly be suggested that the most expedient way of achieving these goals is by means of a second computer, working in a multi-process or configuration. It must not be forgotten, though, that the area of interest here is in small, and somewhat dedicated, systems. A system proposed and developed during the past decade by Texas Instruments Inc. does lean towards the desired goals. This system was the TI ASC 'Super-computer' {30}, which consisted of a central memory, a central processor, a peripheral processor and a host of peripherals. Here, the operating system was processed within the peripheral processor. In this so-called "super-computer", however, hardware complexity (and the resultant cost) brings the system into the large-computer area. As has been discussed above, in process-control applications, the object is simplicity and efficiency. The solution should not lie simply in multiplying the complexity of the overall system by adding additional, complex processors to perform the various tasks required. Instead, the facilities available to the computer architect, in terms of past experience and new technologies, should be carefully studied, and new avenues sought.

3.1 The Essential Hardware Requirements

The facilities required to effect a real-time operating system in hardware may be reduced to the following essential elements (see Fig. 1):

(i) A simple but fast data-processing system. The functions to be performed will generally be centred around the scheduling operations, thus requiring the capacity to handle queues and make logical decisions.

(ii) The processor must have available a limited amount of high-speed storage, in which it can hold process-control blocks, pointers, stacks and constants. In order that the processor may carry out most of its tasks without interfering with the central minicomputer, it is desirable that this memory should be totally independent of the computer's own memory.

(iii) An interface to the minicomputer to perform the interchange of information between the hardware controller and the minicomputer.

(iv) A sophisticated interrupt hierarchy. Such interrupts may be from real-time clocks, from an external process under control, from peripheral devices, or from within tasks being executed. Any hardware-implemented operating system must be able to deal with these interrupts and to respond accordingly. This is really the most powerful argument for removing the operating system from the computer - the ability to deal with multiple interrupts without any penalty in available computing time.

In order to implement these elements in hardware, there are two important factors which must be kept in mind: speed and simplicity. A third factor which should also be considered is the anomalous question of flexibility. As has been pointed out, the user of a real-time operating system will usually require a system optimised for his specific situation. It would accordingly be desirable to retain a degree of flexibility, so that particular features may be eliminated, or additional features added, as required.

3.2 The Proposed Approach

With the new technologies commercially available, the development of the hardware system may be approached in a variety of ways. In principle, the total system discussed in 3.1 could be implemented by familiar and well-understood microprocessor components. One parameter that would suffer, however, is that of speed. Microprocessors are generally based on

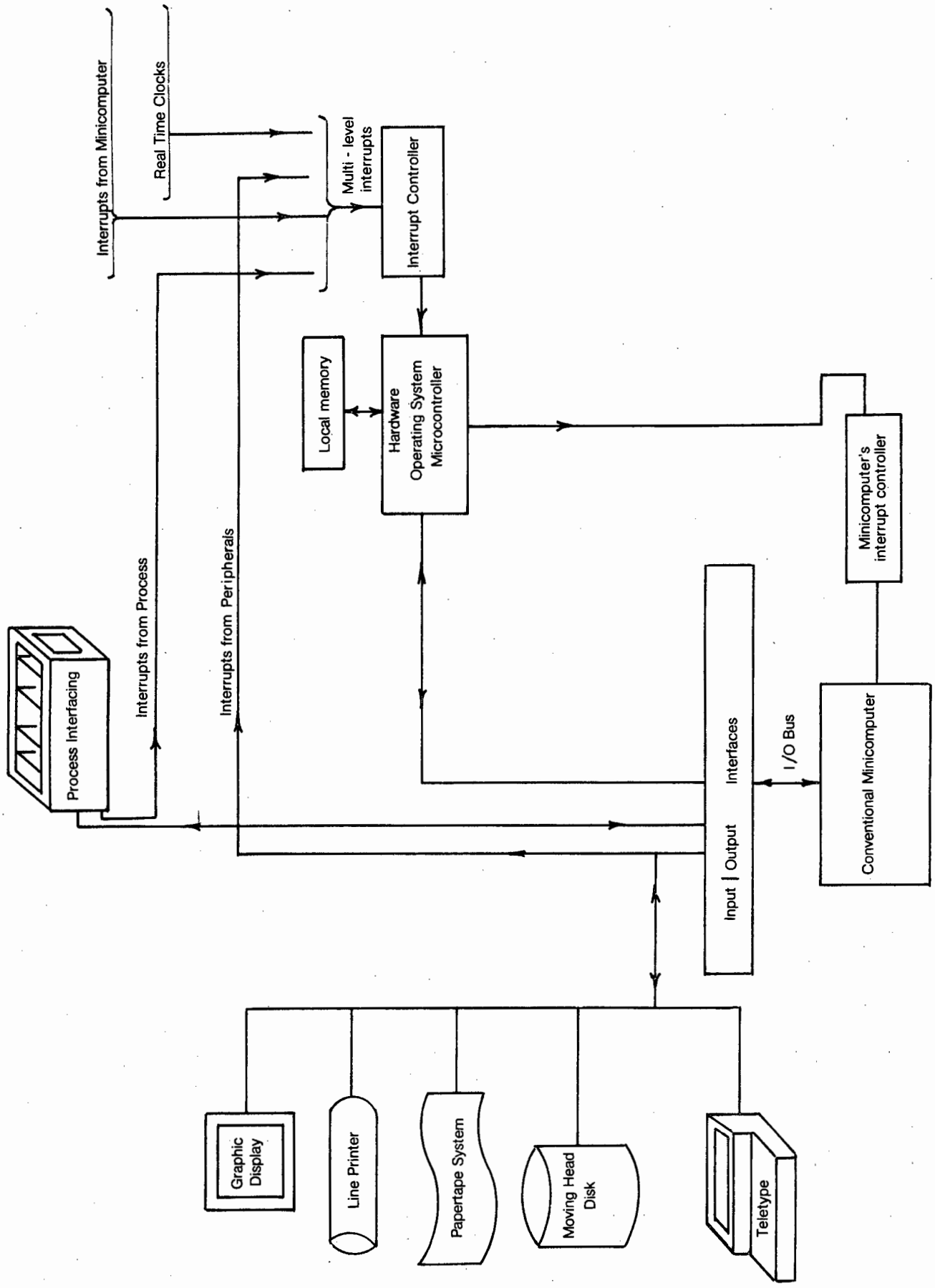


FIGURE 1 : SYSTEM BLOCK DIAGRAM

n- or p-channel MOS technology, which is inherently slow in terms of execution speed. Also, the designer is somewhat restrained by the available instruction set {31}. It is a question of starting with a generalised structure, while pursuing the goal of a structure which is very specific.

A classical approach would, of course, be to tackle the problem, armed with the vast range of large- and medium-scale integrated circuit elements currently available, and to design the complete unit from scratch. This would be a formidable task, with many pitfalls and subsequent delays - all too well-known to the experienced digital designer.

A close look at current technologies reveals a compromise between these two extremes, which fully embodies the over-riding philosophy of this text, i.e. the exploitation of state-of-the-art techniques in a practical situation.

The technology which it is proposed to use is that of "Bit-Slice Microprocessor Sets" {32}. These sets essentially provide the component parts of a computing system with virtually limitless word-size. Since the components are based on bipolar technology, they are capable of high-speed operation, resulting generally in overall cycle-times in the 100 nanosecond range. A critical review of these components is given in Appendix D.

It must be pointed out here that, in fact, the term "microprocessor sets" is rather a misnomer. The component parts of the sets are merely high-density, large-scale integrated circuits, which may be combined to produce a microcomputer. Their use is by no means limited to this type of application, and they should be thought of as circuit elements, to be used where appropriate.

The feature which makes the bit-slice components of particular interest when considering their use in specialised hardware, is their potential micro-programmability. The question of inherent flexibility, in terms of the desired hardware facilities discussed earlier in this chapter, would seem to be answered by this feature, i.e. the creation of a microprogrammable logical unit.

Thus, the underlying structure of the hardware unit may be defined. It will be organised as a microprogrammable, logical device, based on bit-slice architecture.

3.2.1 Microprogramming

Microprogramming, as a technique in digital computer design, is very much in vogue today, but had its origins in the early 1950's. Dr M. V. Wilkes {6} said then that microprogramming was superior to existing architectural techniques, since it offered a greater degree of flexibility in specifying a computer's instruction repertoire, while also resulting in considerable simplification in the logic. Although some microprogrammed computers were build at that time, the techniques were not generally applied

until the mid-1960's. In general, the high cost of utilizing such techniques tended to outweigh the benefits. Recent technological developments, particularly in the solid-state memory field, have led to the current growth in usage. Appendix A briefly describes the principles of microprogramming.

Of central importance in the implementation of a microprogrammed structure [33] is a fast storage medium, in which the microinstructions are stored. The access-time of this memory will limit the total microcycle-time of the computer. In an elementary system, each microinstruction is fetched from the micro-code store, and is then executed. Various techniques have been proposed which will effectively increase this microcycle time, and these will be discussed later in this chapter.

A further current development, and one which is applied to some contemporary systems [17] is the concept of dynamic microprogramming. This embodies the possibility of altering the actual microinstructions to meet changing requirements. In the Burroughs B1700 computer [17], this concept is exploited to permit the emulation of various differing instruction sets.

While the application of microprogramming to commercial computer systems is increasing [34], notably in large systems, such as the IBM System 370 series, the large-computer manufacturers have tended to discourage user-microprogramming. At the other end of the computer industry, mini-computer manufacturers have used user-microprogrammability as a selling feature [14] [35]. From the user's point of view, however, the difficulties encountered in producing microprograms have in many cases led to great disappointment [34].

3.2.2 Bit-Slice Architecture

The past few years have seen numerous entries into the bit-slice microprocessor field. Generally, the component sets offered are similar, with one major variation - the number of bits-per-slice. Intel [32], one of the first entries into the race, opted for a 2-bit-wide slice, with extra input/output port facilities. Advanced Memory Devices [36], on the other hand, exemplify the other approach, that of 4-bit-wide slices, with fewer input/output ports. In other respects, as with all competing technological innovations, there are various pros and cons. From a user's point of view, the question is more one of local availability, with acceptance of any specific disadvantages. The question of "family completeness" must also be considered, the object being to reduce the number of additional elements required to interconnect the component parts of a total system.

3.3 The Design of the Microcontroller

In the discussions which follow, the hardware unit used to implement the real-time operating system will be termed the "Microcontroller".

As has previously been discussed, the attraction of using bit-slice,

microprogrammable architecture is the flexibility which is given to the designer and to the ultimate user. The proposed system has as main objectives two factors - speed and simplicity. Simplicity here has two facets, in terms of implementation as well as of user modifications to the microcoding. As a result, the final system proposed is relatively simple to re-code and meets the essential requirements in terms of processing capability, and yet occupies only a single input/output slot in a conventional minicomputer!

The rest of this chapter deals with the development of each major section of the microcontroller. Full details of the actual circuitry are given in Appendix C. Figure 2 shows a block diagram of the unit.

3.3.1 The Format of the Microinstruction

The principal decision which has to be made by the designer of a microprogrammable logic system, is the format of the microinstruction [37]. Essentially, this will determine the resources which each microinstruction can control. Generally, microinstructions can be classified as vertical or horizontal, although in practice a compromise is normally made. Vertical microinstructions are similar to normal machine-code instructions, effecting, as they do, single primitive operations. They will normally consist of one operation code and one or two operands. Horizontal microinstructions will control the operations of many resources operating in parallel. Typical word-lengths would be 12-24 for vertical microinstructions, and up to 64 (and possibly more) for horizontal microinstructions. It is obvious that horizontal microprogramming offers many attractions in so far as higher utilization of resources is concerned, although it will necessitate a more complex structure, so as to permit parallel operation. As contemporary examples of each type of structure, the HP 2100 computer series [38] is strictly vertically-designed, with a 24-bit word, while at the other end of the scale the Varian V-70 series [35] offers a 64-bit microinstruction and is essentially horizontally-microprogrammed.

On reviewing the requirements of the proposed microcontroller, it becomes apparent that the features required are limited because of the singleness of purpose. Added to this, the requirement of simplicity in user-programmability leads to the choice of a microinstruction format which may be classified as having a vertical structure.

The designation of fields within the chosen microinstruction is shown in Figure 3. In broad terms, this means that within one vertical micro-instruction, the following operations may be specified:

(i) Control of the Central Processor Array.

(ii) Control of Next Address Generation.

(iii) Control of "flags" to and from the Microsequencer Unit and the Central Processor Array. (Full details of these "flags" may be found in Appendix C).

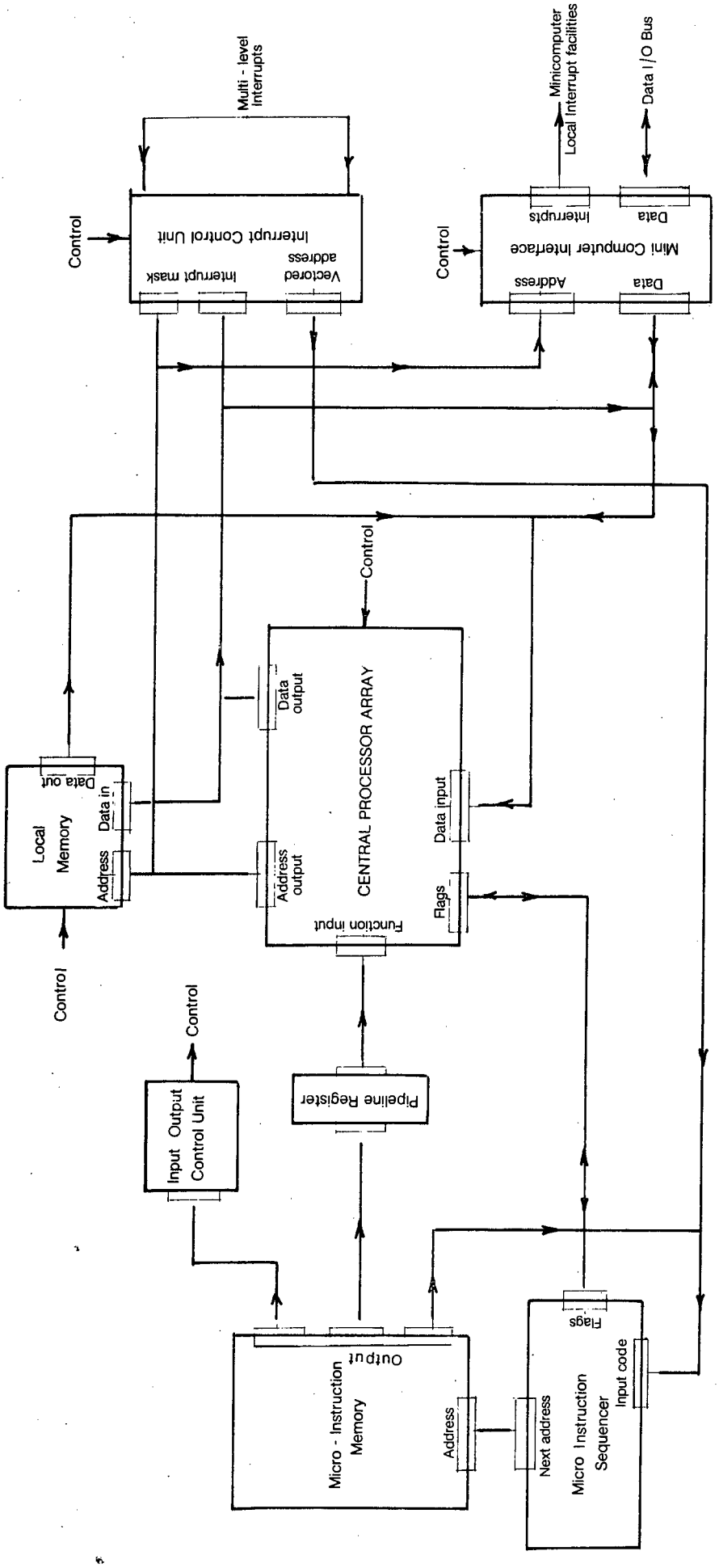


FIGURE 2 : MICROCONTROLLER BLOCK DIAGRAM

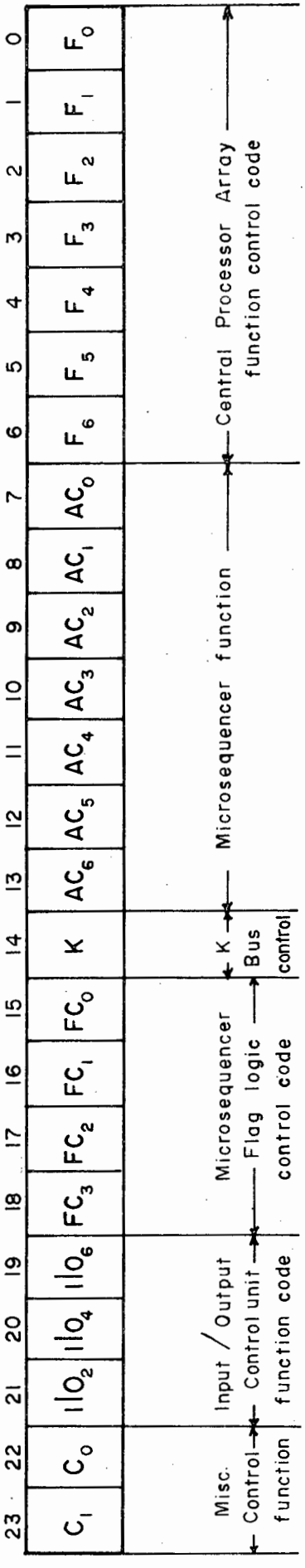


FIGURE 3: THE MICROINSTRUCTION WORD FORMAT

(iv) Control of input/output functions, including the local memory, the interface to the minicomputer, and the interrupt control unit.

3.3.2 The Microprogram Sequencer

The major function performed by the Microprogram Sequencer is the determination of the order in which instructions are to be fetched from the microinstruction store. The identification of the particular location to be accessed in the microinstruction store is usually based on certain prevailing conditions. These will, in practice, include:

(i) Present Address.

(ii) Condition of input flags, which may have been set by the Central Processor Array, or by control bits in the previously-accessed microinstruction.

(iii) Control inputs, normally specified in the previously-accessed microinstruction.

A most desirable functional feature of the Sequencer is that the total amount of microinstruction storage required should be minimised for any given application, as far as is consistent with speed and ease of use [39].

The microcontroller under discussion here makes use of the Intel microinstruction sequencer unit [40]. An interesting problem that arises from the use of such a single-chip, integrated circuit sequencer is the classic problem of addressing a large address space, with a limited number of address bits [39]. In minicomputers, the problem arises from the use of relatively short word-lengths, typically 16- or 18-bit. Thus, in a single-word instruction, only 8 to 10 bits could be allocated to specification of operand addresses. To allow the computer to address a larger memory space, a variety of techniques has emerged. These include indirect addressing, the use of paging registers, and relative addressing techniques. The problem is greatly compounded when designing a single-chip sequencer, where it is necessary to keep the number of input/output connections to a minimum. It is also desirable to reduce the number of next-address control bits which must be included within a micro-instruction.

A typical microprogram storage unit will contain between 2^8 and 2^{10} microinstructions. This will require the development of at least an 8- to 10-bit microinstruction storage address. The microinstruction sequencer used in the proposed microcontroller adopts a two-dimensional addressing system [32]. Essentially, this scheme permits the specification of an address in a 2^9-word address space, using only 7 control bits. The concept is based on the organisation of the address space into a two-dimensional array, with 32 rows and 16 columns, that is, $512 = 2^9$ storage locations. Each microinstruction supplies 7 bits to the microsequencer unit. In most cases either the column or the row address bits will be altered, and the other bits in the microinstruction

will be used to specify the required operation. (Table No. 1 defines the microinstruction sequencer instructions available for the unit used in the microcontroller).

This scheme obviously has the desired advantage of reduced number of input/output connections and a smaller number of bits to be held within an individual microinstruction. It does, however, present the programmer with a very difficult task. This is the placement of the microinstructions within such a two-dimensional space. Observation of Table No. 1 demonstrates the limitations on movement within this space. In addition, certain functions of the sequencer are dependent on the specification of particular destination locations. (An example of this is the enabling of the Interrupt Control Unit, which can only occur when a jump to row 0, column 15, is executed).

This placement problem is at present receiving much attention. The work by J. F. Wakely et al [39] of Stanford University is representative of the attempts being made in this direction. Even this work has shown that the complete automation of the task is extremely complex and, in fact, the technique proposed in Ref [39] still requires an interactive procedure, guided by a relatively skilled operator. Current experience gained by the author on this subject is discussed in Appendix F.

One emergent fact, however, is that when utilizing such a micro-program sequencer, many conventional programming techniques have to be abandoned. Normally, a machine-code programmer is accustomed to writing instructions in sequential fashion. The structure of the sequencer discussed here does not lend itself to such a definite technique, and the programmer must place the instructions according to the particular instruction-type being executed, and the ultimate target address. The technique is to a certain extent similar to programming a multi-address computer with no control (program) counter [41].

An additional system requirement is the automatic vectoring of interrupts. The importance of this lies in the need for rapid processing of asynchronous events - a basic requirement of a real-time operating system. What is needed, therefore, is the ability to specify when an incoming interrupt may be acknowledged, and subsequently to permit the particular interrupting device to impress its vectored address onto the microprogram sequencer. The method adopted in the microcontroller is to activate the interrupt control unit by means of a control line, whose status is set to an active state when a specific jump instruction is encountered in a microprogram. This line is used to activate the interrupt control unit, which will respond if it has a waiting interrupt request, by sending the corresponding interrupt address to the Microsequencer Unit. In order to achieve this, the actual next-address generated by the sequencer is withdrawn by a hardware disabling of the

MNEMONIC	DESCRIPTION	FUNCTION CODE 6	5	4	3	2	1	0	NEXT ADDRESS 8	7	6	5	4	3	2	1	0
JCC	Jump in current column	o	o	d_4	d_3	d_2	d_1	d_0	d_4	d_3	d_2	d_1	d_0	m_3	m_2	m_1	m_0
JZR	Jump to zero row	o	1	o	d_3	d_2	d_1	d_0	o	o	o	o	o	d_3	d_2	d_1	d_0
JCR	Jump in current row	o	1	1	d_3	d_2	d_1	d_0	m_8	m_7	m_6	m_5	m_4	d_3	d_2	d_1	d_0
JFL	Jump/test F-latch	1	o	o	d_3	d_2	d_1	d_0	m	d_3	d_2	d_1	d_0	m_3	o	1	f
JCF	Jump/test c-flag	1	o	1	o	d_2	d_1	d_0	m_8	m_7	d_2	d_1	d_0	m_3	o	1	c
JZF	Jump/test z-flag	1	o	1	1	d_2	d_1	d_0	m_8	m_3	d_2	d_1	d_0	m_3	o	1	z
									ROW					COLUMN			

ADDRESS CONTROL FUNCTION

MNEMONIC	DESCRIPTION	CODE FC_3	FC_2	FC_1	FC_0
SCZ	Set c-flag & z-flag to f	x	x	o	o
STZ	Set z-flag to f	x	x	o	1
STC	Set c-flag to f	x	x	1	o
HCZ	Hold c-flag & z-flag	x	x	1	1
FFO	Force FO to o	o	o	x	x
FFC	Force FO to c-flag	o	1	x	x
FFZ	Force FO to z-flag	1	o	x	x
FFI	Force FO to 1	1	1	x	x

FLAG CONTROL FUNCTION

KEY
d_n - data on address contol line
m_n - current address bit n
f - contents of F-latch
c - " " c-flag
z - " " z-flag
FO - flag output

TABLE 1 : MICROSEQUENCER CONTROL CODES

sequencer's output lines. Overall, this means that each interrupt to the system will direct the microcontroller to execute a microinstruction at a specific, predetermined address in the microinstruction store, and automatic vectoring of interrupts is achieved.

3.3.3 The Central Processor Array

The central processor array in the microcontroller is similar in functional operation to that of the arithmetic and logical unit of a conventional von Neumann-structured computer. Within the microcontroller configuration proposed here, the central processor array is required to execute the following primitive operations:

(i) To perform arithmetical and logical operations as required by algorithms implementing the operating system functions.

(ii) To address and route data to and from the local microcontroller memory.

(iii) To route data to and from the input/output interface to the minicomputer.

(iv) To communicate with the interrupt control unit for the purpose of disabling interrupt lines (normally termed "masking").

(v) As a result of logical and arithmetical operations, to provide for flag conditions to be routed to the microinstruction sequencer. Such flag bits would include overflow conditions, register status and negative sign indications.

Since it is desirable that these operations should be executed as expediently as possible, it will be necessary to have available a comprehensive array of registers. As well as the storage of immediate operands, these registers will be required to hold stack pointers, memory address pointers and often-used system constants.

The design proposed to meet these requirements is based on Central Processor Elements developed by Intel Corporation {40}. In order to achieve highly-efficient arithmetical processing, a look-ahead carry unit is included in the array. The resultant design makes available 11 general-purpose registers, a memory address register and a single accumulator.

The logical operation of the central processor array is determined by an 8-bit control code, which forms part of the microinstruction word. Table 2 defines the possible operations available, and the corresponding microcodes.

It is worth noting at this point that a concept of "conditional clocking" {42} was introduced into the design of the central processor

MNEMONIC	CODE BITS 6	5	4	3 ← 0	K	LOGICAL DEFINITION	EXPLANATION
ILR	0	0	0	REGN	0	R(n)+CI→R(n), AC	Move contents of Reg(n) to Accum
ACM	0	0	0	AT1	0	M+CI→AT	M-Bus to Accum or T-reg
SRA	0	0	0	AT2	0	AT_L→RO AT_H→AT_L LI→AT_H	Shift right accum or T-reg
ALR	0	0	0	REGN	1	AC+R(n)+CI→R(n), AC	Add Reg(n) to Accum
AMA	0	0	0	AT1	1	M+AC+CI→AT	Add M-Bus to Accum or T-reg
LMI	0	0	1	REGN	0	R(n)→MAR R(n)+CI→R(n)	Move Reg(n) to Memory Address Reg
LMM	0	0	1	AT1	0	M→MAR M+CI→AT	M-Bus to Mem Add Reg & Accum or T-reg
CIA	0	0	1	AT2	0	$\overline{AT}$+CI→AT	Complement Accum or T-reg
DSM	0	0	1	REGN	1	R(n)−1+CI→R(n) 11→MAR	Decrement R(n). Set Mem Add Reg to -1
LDM	0	0	1	AT1	1	M−1+CI→AT 11→MAR	M-Bus to Accum or T-reg
DCA	0	0	1	AT2	1	AT−1+CI→AT	Decrement Accum or T-reg
CSR	0	1	0	REGN	0	CI−1→R(n)	Set Reg(n) to zero or all ones
SDR	0	1	0	REGN	1	AC−1+CI→R(n)	Move contents of Accum to Reg(n)
LDI	0	1	0	AT2	1	I−1+CI→AT	I-Bus to Accum or T-reg
INR	0	1	1	REGN	0	R(n)+1→R(n)	Increment Reg(n)
ADR	0	1	1	REGN	1	AC+R(n)+CI→R(n)	Add Reg(n) to Accum
AIA	0	1	1	AT2	1	I+AT+CI→AT	Add I-Bus to Accum or T-reg
CLR	1	0	0	REGN	0	0→R(n) CI→CO	Set Reg(n) to zero
ANR	1	0	0	REGN	1	CI∨(R(n)∧AC)→CO R(n)∧AC→R(n)	Logical AND Reg(n) with Accum. OR the result with CI & set CO accordingly
ANM	1	0	0	AT1	1	CI∨(M∧AC)→CO M∧AC→AT	as for ANR except replace Accum by M-Bus
ANI	1	0	0	AT2	1	CI∨(I∧AT)→CO I∧AT→AT	" " " " " " " I-Bus
TZR	1	0	1	REGN	1	CI∨R(n)→CO R(n)→R(n)	Test status of Reg(n)
LTM	1	0	1	AT1	1	CI∨M→CO M→AT	" " " M-Bus. Move M-Bus to Accum or T-reg
NOP	1	1	0	REGN	0	CI→CO R(n)→R(n)	No-Operation
ORR	1	1	0	REGN	1	CI∨AC→CO R(n)∨AC→R(n)	Test status of Accum. OR Accum with Reg(n)
ORM	1	1	0	AT1	1	CI∨AC→CO M∨AC→AT	" " " " " " " M-Bus
ORI	1	1	0	AT2	1	CI∨I→CO I∨AT→AT	" " I-Bus. OR Accum or T-reg with I-Bus
CMR	1	1	1	REGN	0	CI→CO $\overline{R(n)}$→R(n)	Complement Reg(n)
LCM	1	1	1	AT1	0	CI→CO M→AT	Complement of M-Bus to Accum or T-reg
XNR	1	1	1	REGN	1	CI∨(R(n)∧AC)→CO R(n)⊕AC→R(n)	Set CO to the OR of CI & the AND of Reg(n) & the Accum. Excl. NOR Reg(n) & Accum
XNM	1	1	1	AT1	1	CI∨(M∧AC)→CO. M⊕AC→AT	as for XNR except replace Reg(n) by M-Bus
XNI	1	1	1	AT2	1	CI∨(AT∧I)→CO I⊕AT→AT	" " " " " " " I-Bus

SYMBOL	MEANING
I,K,M	Data on the specified Input Bus
CI,LI	" " " Carry Input or Left Input
CO,RO	" " " Carry Output or Right Output
R(n)	Contents of specified Register
AC	" " the Accumulator
AT	" " AC or T-register
MAR	" " the memory address register
L,H	As subscripts, low & high order bits
+	2,s complement addition
−	" " subtraction
∧	Logical AND
∨	Logical OR
→	Deposit into

REGISTER GROUP	REGISTER	CODE BITS 3	2	1	0
REGN	R(0)	0	0	0	0
	R(1)	0	0	0	1
	R(2)	0	0	1	0
	⋮	⋮	⋮	⋮	⋮
	R(9)	1	0	0	1
	T	1	1	0	0
	AC	1	1	0	1
AT1	T	1	0	1	0
	AC	1	0	1	1
AT2	T	1	1	1	0
	AC	1	1	1	1

TABLE 2: CENTRAL PROCESSOR ARRAY INSTRUCTION CODES

array. This facility, under the control of additional bits in the micro-instruction word, effectively permits the non-destructive testing of register status. (See Appendix C.8 for further details).

3.3.4 The Input/Output Control Unit

Three bits of the microinstruction word are dedicated to the control of input/output functions and certain central processor array operations. The following classification may be made:

(i) Local memory read/write functions.

(ii) Peripheral device read/write functions.

(iii) Central Processor Array operation, including conditional clocking.

An important problem which the input/output control unit must deal with is the synchronization of the microcontroller with the local memory and the peripheral devices. In such an input/output operation, the address of the particular device is generated by the central processor array, and the input/output unit develops the control signals to activate the device in question. There will normally be a finite delay before the device has completed the specified task and program execution can continue. This delay may be greater than the normal microcontroller cycle-time. Two solutions are commonly adopted to overcome this problem: firstly, a "sense-ready" type of operation, whereby the processor will continually interrogate the device as to its availability. When the device has completed its specified operation, the processor detects this fact, and continues normal execution. This system has proved simple to implement, and is used in the design of many computers {43}. A second solution is for the processor to initiate an input/output cycle, and to complete the cycle only when a sensing device informs the processor that the specified operation has been completed. This effectively allows for the extension of the processor's cycle-time. Provided that the expected response will not delay the processor for too long a time, this approach has great value in terms of simplicity of programming. The technique may be extended by being used in conjunction with an integrated interrupt-oriented structure, in which slow peripheral devices need not cause undue delays. For example, data to be transferred to a peripheral device may be placed in a buffer store. The actual transfer to the external device will take place when it indicates its availability by means of an interrupt. This system is used in the PDP 11 range of computers {24}.

The second method is used in the organisation of the microcontroller. The idea is that all input/output operations executed by the system will inhibit the system's master clock. When the external device is ready, or has completed its operation, a signal is sent to the control unit, which releases its hold on the system clock, and the execution of the program may continue.

3.3.5 The Local Memory

The necessity for the system to have storage for tables, stacks, bit-maps, etc., has been mentioned before. Two methods of providing this facility were considered when designing the microcontroller. The first method is to make use of the minicomputer's memory. This does, however, deviate from one of the primary objectives of the concept proposed here: to release the minicomputer from as much as possible of the operating system functions. In addition, in terms of speed, even using direct access to the minicomputer memory, the microcomputer will be restricted to the speed of the minicomputer. (This arises because of the method of implementation of Direct Memory Access in the minicomputers used in this work - see Appendix B). So the second solution is to provide a limited amount of high-speed cache memory, to which the microcontroller has ready access.

It was found in the microcontroller which was developed that 256 words of local memory were adequate. This number might at first sight appear restrictive, but it does provide sufficient storage for a relatively large suite of tasks (see Chapter IV). Any subsequent expansion of the memory presents no problem, as a full 16-bit address may be developed by the central processor array.

3.3.6 The Interrupt Control Unit

The interrupt control unit is of great importance in the overall system philosophy. It must be capable of accepting multi-level interrupts, effecting the interruption of the current microsequence, and delivering the appropriate vectored addresses to the microinstruction sequencer unit. In addition, it must be capable of holding masking information, so as to prevent interrupts below a specified level from being passed on to the microsequencer unit.

The resulting interrupt unit was developed, based on the Intel interrupt control unit elements {40}. The system is theoretically expandable to any number of levels of interrupt. However, with the method of interrupt vectoring adopted, and within the limits of the microinstruction addressing scheme, a maximum of 64 levels may be employed. This limitation arises because the vector address generated acts only on the row address, and will always have the column address 15. So, as 1 Kiloword of microinstruction memory is provided, 64 possible column addresses exist. In the prototype system developed, 24 levels were, in fact, found to be adequate. When an incoming interrupt is accepted, an Interrupt Strobe Enable signal (generated by the microinstruction sequencer unit) is awaited. When this occurs, the interrupt vector is made available to the system, and the system responds as outlined in Section 3.3.2. The required masking level is handled by the execution of an output-to-peripheral microinstruction, the data being supplied by the central processor array.

It must be mentioned that there is another possible means of handling the interrupts. This is essentially a non-vectored system, but it does have certain advantages in that it has a common target address for all interrupts. A sequence of microinstructions is started at this target address. This sequence must input the level of the interrupt to the central processor array and set the required mask level. Execution of the microprogram may then continue according to the level indicated. The main disadvantage of this system is the extra time required to perform the interrupt identification procedure. Both alternatives were investigated, and in practice the first approach proved more suited to the requirements of the system. From the point of view of hardware simplicity, no real advantage is gained by adopting either procedure, as the hardware required to implement either is very similar.

3.3.7 Interfacing the Microcontroller to the Minicomputer : The Input/Output Interface

From a philosophical viewpoint, this section of the system provides much interest. It is the meeting-point of two seemingly-incompatible entities. The design philosophy has two main objectives: firstly, to achieve an effective means of synchronization between the two systems and, secondly, to achieve this goal at the best possible cost/performance ratio.

The overall system must be capable of two basic functions:

(i) The bi-directional transfer of data between the microcontroller and the minicomputer.

(ii) The creation and acceptance of interrupts to and from the minicomputer.

In order to outline the actual implementation of these functions, it is essential to discuss the basic method of synchronization and communication adopted.

The approach used is unconventional but proves to be most efficient, requiring a low overhead in terms of necessary support software. Firstly, consider the transfer of data from the microcontroller to the minicomputer. The procedure commences with the microcontroller sending an interrupt to the minicomputer, after which the microcontroller is free to continue with any background tasks, or simply to idle. This interrupt is generated by an output-to-peripheral microinstruction. On accepting this interrupt, the minicomputer will trap to a short,handling routine. It might, however, be held up by the execution of non-interruptable instructions, an unfortunate feature of some minicomputers. The handling routine commences by sending a return interrupt to the microcontroller, and then enters a short delaying routine, the length of which is determined by the maximum time that the microcontroller will ever take to respond to an interrupt. The microcontroller then places the data to be transferred onto its output bus, and the system clock is held up, as described in 3.3.4, until the minicomputer accepts the

data. For the transmission of short blocks of information, the process is continued without further interrupts being needed. In effect, this amounts to the microcontroller's being synchronized, for the period of transfer, to the minicomputer.

Transfer in the reverse direction occurs in a similar fashion. In the case of the minicomputer's requesting a data transfer, the minicomputer signifies its requirement by effecting an interrupt to the microcontroller, and awaiting a response. The microcontroller replies by means of an interrupt. Transfer in this direction has one additional complication. This arises because the minicomputer used has a control structure dissimilar to the microcontroller's. The minicomputer will send data without waiting for the data to be accepted. The problem is overcome by introducing a buffer register between the two systems: the minicomputer writing the data into the buffer, and the microcontroller removing it.

The technique adopted might at first sight appear hazardous. A more conventional approach would be to provide a full buffer memory. Either system could then write all the data to be transferred into this buffer, either under program control or by direct access means, and the other device would remove the data in its own time. It is a question of compromise - extensive hardware, or finely-tuned algorithms based on timing relationships. The method proposed and utilized has great simplicity, but it does require careful programming, especially of the minicomputer. A slight measure of relief may be obtained by introducing a means whereby the minicomputer "senses" when the communication channel is available for a transfer to occur. This system was investigated. Experimental results showed that, in fact, because of the precise timing which may be determined, the method finally adopted is perfectly adequate and requires a minimum (albeit carefully designed) of software in the minicomputer.

As a general principle, however, it must be accepted that the most suitable method would be to use a minicomputer with a dual-port memory structure [35], whereby both minicomputer and microcontroller have access to a common block of memory. For a system without this facility, the proposed approach has merits in terms of cost-effectiveness, which is a primary goal of the structure being presented.

In order to implement the interrupt system discussed above, relatively straightforward hardware is required. Interrupts from the minicomputer are effected by the use of the minicomputer's external command instructions (see Appendix B), which are decoded by the input/output interface, and are sent to the interrupt control unit. Interrupts from the microcontroller are decoded, since they are created by output-to-peripheral microinstructions, and are supplied directly to the minicomputer's interrupt module. This interrupt module [44] is itself vector-based, and so the minicomputer has a direct means of

relating the interrupt received to the function requested by the microcontroller.

3.3.8 The Microinstruction Memory

The microinstruction memory, often termed the "control store", and abbreviated in this text to the "micromemory", is the unit in which the actual microinstructions are stored. The format of this memory must, of course, conform to the capabilities of the microinstruction sequencer unit. As has been pointed out in Section 3.3.1, the sequencer unit can address directly 512 words of micromemory. This may be extended further by introducing additional "banks" of micromemory, each consisting of 512 words. The bank-switching, or selection of bank, can be achieved by the introduction of additional bits in the microinstruction word. The setting of these bits informs the micromemory as to which bank is being accessed. In the system discussed in this text, two banks of 512 words, each word consisting of 24 bits (the microinstruction word-length), were installed. Bits 22 and 23 of the microinstruction word were used to control the selection of bank. In practice, it was found that only one bank was ever required, thus somewhat relieving the problem of additional placement of microinstructions.

The key factor in the design of the micromemory is access-time. In an elementary, microprogrammable computer configuration, this access-time has a direct influence on the effective microcycle-time. This may be demonstrated by considering a complete microcycle. The cycle commences with the address (generated by the microinstruction sequencer) being made available to the micromemory. The micromemory will supply the corresponding microinstruction. The time taken for this microinstruction to become available is, of course, a direct function of the access-time (or, more specifically, the read-time) of the micromemory used. Once the microinstruction is available, the actual processing of that microinstruction can proceed. This means that the duration of each microinstruction processing cycle must include the read-time of the micromemory. The maximum cycle-time of the central processing array is thus degraded. The method of "pipelining" the system overcomes this problem.

The principle is to overlap the two cycles, the current microinstruction being held in a so-called "pipeline" register. While the rest of the system is operating on this microinstruction, the microinstruction sequencer generates the next address and supplies this address to the micromemory. The next microinstruction may then be accessed, ready for entry in the pipeline register at the start of the next cycle.

This technique thus permits a shorter overall processing cycle. It does, however, result in the introduction of some rather unfortunate microprogramming problems. These may be demonstrated by considering the execution of a conditional jump microinstruction. In an un-pipelined configuration, the conditional jump is totally specified within one microinstruction. The function code to the central processor array will be set up to test the required condition.

The central processor array will send the resulting conditional flag bit to the microinstruction sequencer. This bit may immediately be included in the determination of the next address.

In a pipelined organisation, this sequence is not possible in view of the overlapping of access and processing cycles. If the central processor array function code required to test a certain condition is held in microinstruction n, say, the next address control function which can make use of the result of the testing must be included in microinstruction n + 1. In terms of the development of microprograms, this is an additional problem to be borne in mind. In addition, it can have a detrimental effect on the overall system performance. In view of this two-stage implementation of conditional jumps, the second word is difficult to utilize for central processor array functions. This is because the next operation to be processed following a conditional jump normally depends on the result of the conditional test. As in most programming situations like this, a careful review of the instruction flow often reveals that a nominal "housekeeping" type of operation can be executed by the necessary second word.

In most contemporary, microprogrammed computers, the microinstruction memory, or control store, will consist of two sections. The first is a read-only memory in which are stored the microinstruction sequences required to implement the macroinstruction set of the computer. The second form of memory, normally offered as an optional facility, is user-programmable memory, termed "writable control store". This gives the user, via suitable control logic, facilities for developing his own microinstruction sequences. Such a configuration is typified by the computers described in {35} and {38}.

There have, however, been many examples of structures using dynamically-programmed microinstruction memory. This concept was discussed in Chapter I. The approach permits the user to reprogram the micromemory completely, to meet changing circumstances, such as those encountered in the emulation of various computer structures. Conceptually, this is a very powerful technique which appears to have great advantages in the design of the microcontroller. This concept will be discussed in Chapter V but, at this stage, the flexibility achieved provides full justification for the adoption of the strategy, especially in the development and evaluation periods.

3.3.9 Miscellaneous Hardware Facilities

The ultimate objective of the proposed microcontroller is to include it as a relatively inexpensive option which may be incorporated with a minicomputer's complement of optional hardware. It should be as simple to include within the system as, say, a hardware floating-point arithmetic unit. In principle, the proposed structure meets these requirements. Provided that the microinstructions are defined, there is no reason for any external intervention, beyond the controls provided by the minicomputer control signals and the external interrupt generators. The key words in this statement are "provided that the microinstructions

are defined. Two practical situations must be considered: firstly, development of the microinstruction set, and secondly, the possibility of dynamic microprogramming.

During the development and evaluation phases discussed in Chapter IV, it is obviously essential to have full access to the microinstruction memory, and to have full control over the operation of the microcontroller. This leads to the inclusion of the monitor features described in Appendix E. In a dedicated operation, such as on-line process control, the microinstructions would be permanently stored in read-only memory, and all monitor facilities would be removed.

In order to provide for dynamic modification of the contents of the microinstruction memory, the memory must be programmable. Addresses and data are supplied to the microinstruction memory via the minicomputer. This feature is simple to implement, and provision was made for it in the development of the system. It is also desirable to be able to monitor the operation of the microcontroller when developing microinstruction programs [38]. For this reason, a monitor similar to that proposed is valuable as a removable facility.

4 Review of the Microcontroller Hardware

The most surprising aspect of the hardware, when analysed in the context of the facilities offered, is its relative simplicity! This will be less surprising to the experienced user of microprocessors, except where the overall system instruction-time is concerned. Even with the restrictions which had to be applied in certain areas of the prototype system because of the non-availability of certain components, the effective cycle-time of the microcontroller is approximately 300 nanoseconds. (Appendix K discusses the prototype microcontroller specifications). This gives an effective performance improvement factor of at least 5 over a conventional minicomputer. This type of comparison is not strictly valid, however, as the microcontroller is dedicated to a specific task, does not offer the versatility of a computer, and should not be thought of in this light. The point is that it has been shown that the hardware facilities necessary to perform the tasks of real-time operating systems can, in view of current technology, be produced in a simple and straightforward fashion.

CHAPTER IV

A BASIS FOR EVALUATION

"That's exactly the method," the Bellman bold
In a hasty parenthesis cried,
"That's exactly the way I have always been told
That the capture of Snarks should be tried!"

Lewis Carroll: *The Hunting of the Snark (Fit the Third)*

A structure has been proposed which, it has been claimed, will permit the implementation of a real-time operating system in hardware, external to a conventional minicomputer. In order to prove the validity of such a technique, a thorough evaluation of its application in operational computer systems must be undertaken. The object will be not merely to show that the concept is functional, but also to demonstrate clearly, in quantitative terms, the advantages of adopting this philosophy in real-world applications. As has been stressed throughout this dissertation, the particular field of interest is in on-line process-control computer applications. It is thus in such areas that experimental situations must be created, and overall system performance evaluated.

4.1 Evaluation and Measurement Techniques

"Progress in designing and applying information handling systems has outraced progress in evaluating their performance" {45}.

In general, the evaluation of a computer system's performance is of the utmost importance when considering its suitability for a specific application {46}. The objects of an evaluation might be to maximize the throughput of a system, to process a given load for a minimum cost, or to achieve any number of other objective functions. Inherent is the principle of performing the evaluation with a specific objective in mind. This principle is, of course, not applicable only to the computer world, but is true of any aspect of science and technology. In order to evaluate any physical system, a frame of reference must be established, within which the evaluation procedure is to be performed.

In the broadest of terms, evaluation is the ascertaining of the value of an object. On the other hand, measurement is the ascertaining of its extent {47}. Measurement is therefore a valuable, but not always essential, part of the process of evaluation. In other words, evaluation might necessarily include certain aspects which are not readily, or even feasibly, measured. These may include, for instance, such factors as the reliability and security offered by a particular computer system.

Appendix G describes various classical techniques which may be used to evaluate the performance of computer systems. Often-used techniques include "benchmarks", the use of "synthetic programs", mathematical modelling

and simulation. The question which must be carefully examined, however, is that of deciding which technique, or combination of techniques, is most applicable to the computer system concerned. In fact, it must be decided whether any of the classical techniques are applicable, or whether new strategies must be proposed. In order to determine an evaluation strategy for the system of concern in this work, therefore, it is necessary to review the principles on which the proposed concept is based, and to consider the application of the concepts in practical situations.

4.2 Design Objectives of the Hardware-Based Real-Time Operating System

Chapter I discussed the rationale for proposing a radical deviation from the normally-accepted organisation of a small computer system used in real-time process-control applications. Two major factors emerged: it is desirable, firstly, to simplify user control over the overall operation of the system and, secondly, to improve overall system efficiency. From an evaluation point of view, these two aspects present completely different problems. The first point, that is, simplification of use, is subjective and may be meaningfully evaluated only by use in suitable applications. The question of efficiency improvement does lend itself to objective evaluation and to the application of relevant measurement techniques.

In the determination of efficiency, a certain degree of circumspection must always be exercised. The efficiency of any physical system may, in fact, only be evaluated on a comparative basis. Efficiency is best defined as the ratio between useful work performed and the total energy expended. In computer terms, what is meant by "useful work"? From the viewpoint of the process-control engineer, the most meaningful definition of efficiency would appear to be the ratio between useful "process-control work" performed, and the total work which the computer installed is capable of performing. In practice, then, a certain amount of potential computing power is available, which may be expressed, for example, in terms of arithmetical ability, memory and peripheral equipment. How much of this power is actually used to control the process, and how much is used to keep the computer system itself operative?

This definition of efficiency has deliberately been expressed in vague terms. The problem which must always be faced is simply that of estimating the potential power of an object which is strongly application-dependent. There is obviously no clear solution to this problem. A reverse approach seems more feasible. In the analysis of any computer system or program, the parameter which is of particular importance is time - the time required to perform different functions or the time spent waiting to perform them [48]. Thus, performance analysis should primarily consist in attempting to answer the question, "Where does the time go?" Given the answer, a subjective judgement can then be applied, to decide whether the amount of wasted time is acceptable.

These ideas point to an approach which may be applied to the evaluation of the proposed hardware-based, real-time operating system. What are the practical advantages to be gained by adopting this philosophy, as compared with a classically-structured real-time process-control computer? Certain advantages will, naturally, be subjective, but in terms of efficiency (as discussed above), clear indications should be established.

The evaluation procedure can be divided into two components. Firstly, the impact of the proposed technique on the computer and its peripherals must be evaluated. Secondly, the effects of such a computer configuration on an overall process-control application must be considered.

Within each division certain key factors emerge, each of which forms an integral part of the evaluation procedure. These may be summarised as follows:

(i) Effects on the computer and peripherals:
  (a) Memory utilization
  (b) Central processor availability and utilization
  (c) Peripheral availability and utilization

(ii) Effects on the overall application system:
  (a) System response times
  (b) User interaction and control
  (c) System security and repeatability.

It is important, at this stage, to stress that these considerations will be largely dependent on the actual computer system to which the hardware-based operating system is applied. What is important, however, is the evaluation of the overall effect of the proposed strategy on a typical conventional computer system.

4.3 Techniques Employed in the Evaluation Procedure

"Measurement yields the greatest fidelity with respect to actual configurations; the analytic approach is most attractive because it can economically provide general insights into computer operations. Much effort has been directed towards both analytical modelling and performance measurement, but too often these efforts have been decoupled" {49}.

The basic philosophy adopted in the development of the evaluation procedures is to look towards practical engineering applications, rather than the use of elaborate hypothetical situations. This concept underlines the fact that the original approach to the subject was based on user requirements, rather than on an abstract attempt to revolutionize the computer industry! Experimental situations had to be created, oriented towards typical applications met with in real-time process-control. This approach, therefore, ruled out many conventional techniques available to the computer engineer, as discussed in Appendix G. Of these techniques, the use of simulation is probably the only one applicable, but it was decided not to make use of this method, as it seemed unlikely that additional information could be obtained as to system

performance, beyond that which was readily available from the experimental situations described in this chapter.

It must also be remembered that in experimental situations, conditions must be capable of being fully controlled, so that meaningful conclusions can be reached. Therefore, while it would be valuable to apply the techniques outlined in an existing industrial process-control situation, the problems which would be encountered in monitoring the system fully, would preclude such an exercise. (Problems would also arise in obtaining access to such an application!)

The evaluation strategy adopted was thus to create a series of experimental environments, based on practical situations, which would permit the making of meaningful measurements. A set of situations was adopted so that a variety of facets of the approach could be investigated, while a controlled environment was still maintained. Each experimental configuration was deliberately kept at a relatively primitive level, so that the required close control mentioned above could be realised. The actual computer systems used in the experiments were configured in such a way that suitable hardware and software "probes" could readily be "inserted" into the system, permitting the making of accurate timing measurements. For example, particular single-word instructions {50} could be included within a program, having only an infinitesimally delaying effect on that program, but resulting in external signals, made available to the monitoring equipment. Thus, the execution-time of any particular program could be measured by including these instructions (or "hardware probes") at the start and the finish of the minicomputer instruction sequence. The intervals between the external signals generated by these instructions could then be measured.

4.4 The Experimental Configurations

Three experimental environments were suitably configured so as to permit the evaluation of the hardware-based, real-time operating system-concept in slightly different situations. The essential structure of the real-time operating system proposed in Chapter III remained the same throughout the experiments. The major system change which was required for each configuration, lay in the "scheduling algorithm" to be applied. Thus, while the basic concepts of the proposed operating system (such as the use of process control blocks, wait-queues and time-dependent tasks) remained the same throughout, the requirements of each configuration led to minor modifications in the "links" between these fundamental ideas. This meant that microsequences, such as those required to cause a task-change (known as a task "swop"),being effectively the "Virtual Instructions" (see Chapter II), needed to be microcoded once only. To change the configuration, relatively

few microinstructions had to be altered. This, of course, clearly demonstrates the potential power of a dynamically-microprogrammed system.

The principles of scheduling have been mentioned in Chapter II. The need for scheduling is to decide "what to do next". In other words, the algorithm must select a task to be executed, then await the occurrence of an event (such as an external interrupt), decide what is indicated by that event, and take the necessary action. The scheduling algorithm is, in effect, the strategy which is employed by the operating system to determine the order in which tasks are to be executed. The subject of scheduling algorithms has proved of great interest to the computer scientist, and many philosophies have emerged. Appendix H contains a survey of this subject, with particular reference to scheduling algorithms proposed for use in real-time, process-control applications. In the following descriptions of the actual experimental configurations, the scheduling algorithms will be discussed individually.

4.4.1 Experimental Configuration I : A "Time-slicing" System

One of the oldest methods adopted in large, time-sharing computer systems is that of "time-slicing", originally proposed by Strachey in 1959 {51}. Here, the operating system allocates each task under its control a small slice of processor time. The simplest algorithm used to implement this technique is based on a "round-robin" system. Each task in turn is given a (normally) fixed amount of processor-time, usually referred to as a "quantum" or "time-slice". Thus, the first task will commence execution, and will be allowed to run for a quantum of time. At the end of that time, it will be removed from execution (or "swopped out"), and the next task will commence execution. The time taken to implement the swopping of tasks is normally called the "swop-time". In terms of overall processor-utilization, this swop-time is really non-productive, in so far as user-output is concerned; it accounts for the major overhead encountered in time-sharing systems in general. The length of a quantum is important, since this will determine overall system utilization. A larger quantum will lead to high utilization of the processor, but could result in intolerably-long response-times for the users. A short quantum means a lower utilization of the processor but better user-response. Reference {23} discusses this problem in depth. It should also be noted that in some advanced systems a variable quantum is used {52}.

To demonstrate the use of the microcontroller in implementing a time-slicing operating system, a suitably-representative computer system was configured. It provided an ideal starting-point for the evaluation phases of the overall concept presented in the dissertation. The fundamental concepts of the proposed real-time operating system were applied, but with a relatively

simple scheduling algorithm. As the quanta permitted for the successive execution of each task were fixed, it was relatively easy to monitor important system parameters such as swop-time and processor utilization.

The configuration consisted of the following:

(i) The microcontroller, interfaced to a minicomputer, as described in Appendix C.

(ii) The creation of four tasks, all of which were simultaneously held in the minicomputer memory, so that no back-up storage provision was required. The four tasks were:

(a) The editing of source programs, using a teletype terminal.

(b) The control of a paper-tape reader, and the verification of data read from the device.

(c) The output of data onto a paper-tape punch.

(d) The output of a single stream of data to a digital counting device, at a rate determined by the task. (This task, which might appear rather trivial, was specifically chosen for evaluation purposes. The number of counts per second read by the counting device gives a clear indication of the time allocated by the operating system to that task).

(iii) A real-time clock, which was applied to the microcontroller's interrupt control unit, and used to determine the quantum of time to be allocated to each task. The clock was variable, so as to allow for modification of the quanta.

As well as the minicomputer programs required to implement each task, communication control routines necessary to interface with the microcontroller had to be developed in accordance with the principles in section 3.3.4. These communication routines were common to all the experimental situations.

4.4.2 Experimental Configuration II : A Foreground-Background, Disc-Based System

The principle of a foreground-background mode of computer system operation was mentioned in Section 2.1. The essential idea is that the tasks which are under the control of the operating system are divided into two sets. One set, the so-called foreground tasks, are those which are normally time-critical, and to be executed at definite times, or when requested by specific occurrences (such as an incoming interrupt signal).

The background tasks are non-time-critical, and are to be executed whenever possible. This means that when the operating system has no more foreground tasks requiring execution, it will commence execution of a background task. In a typical process-control situation, the foreground tasks would implement the actual control over the process, and the background tasks might be the compilation of new programs or the preparation of reports.

In most real-time computer applications, it will be necessary to have fast back-up storage facilities available (such as a moving- or fixed-head disc, or magnetic drum). This need arises because the amount of computer storage available will not normally be sufficient to contain all the required tasks and related software. The tasks must therefore be stored on the back-up device, and brought into the computer memory as and when required. (The decision as to which tasks should be held in the computer's memory at any one time is of great importance, and has been widely discussed. In the present context, a simple approach was adopted. Reference {53} gives a valuable introduction to the subject, and discusses the broader philosophical aspects).

The second experimental configuration was aimed at applying the hardware-based real-time operating system within the framework discussed above. The difficulty was to prevent the experimental situation from becoming too complex (and hence too difficult to control and evaluate), and a relatively primitive configuration was therefore adopted. This consisted, firstly, of one foreground time-critical task: the sampling of a signal applied to an analogue-to-digital converter. The sampling rate was determined by an interrupt signal, sent to the microcontroller's interrupt control unit. In addition, two background tasks were created. These consisted of two BASIC-language Interpreter programs {54}, each working in conjunction with a separate console device, a video display unit. This permitted two simultaneous users to carry out the creation and execution of BASIC programs. In view of the memory requirements of a BASIC Interpreter, it was necessary to store these programs on a moving-head disc system, and to bring alternate copies of the Interpreters into the computer memory.

It should be noted that this is inherently an inefficient method of providing time-sharing facilities for two users requiring access to the same program. Strictly, the Interpreters should be re-entrant, so that only a portion of each user's program would need to be stored on the disc. Such an Interpreter was not available, however, so the method adopted was to place on the disc the total extent of the computer's memory needed for each user, including the users' programs and a copy of the BASIC Interpreter. This did have the advantage, however, that either user's requirements could be changed:

one could, say, have a source-program editor available instead of the BASIC Interpreter.

Thus far, three separate tasks have been described: the two BASIC tasks and the analogue-to-digital conversion task. It will be recalled from Section 2.3.2. that one of the operating-system principles adopted was that of regarding all peripheral-handling procedures as individually-defined tasks. To effect the storage on the disc of the two BASIC Interpreters and the subsequent recalling of each, therefore, a fourth task had to be specified. This was a disc-handling task, which controlled the storing of data on, or the loading of data from, the moving-head disc system. The actual disc-area to be used in any transfer was determined by a "bit-map", held in the process control block transferred from the microcontroller when this task was requested. The length of each of the BASIC Interpreter programs was limited so as to be contained within one "cylinder" on the disc drive and hence reduce the time required for disc-access. (One "cylinder" contains 20,000 16-bit words and may be accessed by an initial head-movement only {55}).

There were, unfortunately, some practical limitations resulting from the actual physical configuration used. The most serious lay in the fact that the disc system available is relatively slow, and has an average time of 220 milliseconds to transfer the specified amount of data to or from the minicomputer. Thus, it will take an average time of 0,5 seconds to complete a full swop in user programs. This problem led to an inherently slow system from the viewpoint of the two users, but it could not be overcome within the software and hardware then available.

A second, and equally serious, problem arose from the nature of the design of the interfacing system used to control the disc drives {55}. Data is transferred to the disc under direct memory access (see Appendix B for a description of this method), which occurs at a specified transfer rate. On a request for the execution of the foreground task during such a transfer, two possible courses of action could be taken. Firstly, the foreground-task-request could be delayed until the disc-transfer was complete. This was undesirable in view of the underlying principle of foreground-background operation. The second approach was to abort the disc-transfer, and re-start it completely, once the foreground task had been attended to. This method was adopted in practice, even though it caused a further degrading of user response.

The scheduling algorithm needed to control the system as configured above was similar in nature to the "round robin" one outlined in Section 4.4.1. The two background tasks had equal priority, and they were swopped at a rate determined by a real-time interrupt signal. When this interrupt occurred,

the microcontroller had to select the disc-handling task and effect a "task-swop-out" operation. Having completed this, the disc-handling task was again selected, so as to "swop in" the next background task. The interrupt requesting the execution of the analogue-to-digital task could occur at any time. When this happened, any current task (either a disc-handling task or one of the background tasks) was suspended, and the foreground task brought into execution.

As previously discussed, it is in fact only in the scheduling algorithm that a difference occurred between the real-time operating system used in the first experimental situation, and the second one. Otherwise, all microinstruction sequences were the same. From the minicomputer, the communication routines required to interface to the microcontroller were also identical to those used previously. The foreground task to control the analogue-to-digital converter and the disc-handling tasks were resident in the minicomputer's memory. The overall system was essentially interrupt-driven, each task being requested by the occurrence of an interrupt. In addition to the two real-time signals used to implement the background-task time-slicing, and the analogue-to-digital sampling interval, there were two other interrupt "classes" of importance. Firstly, there were those used to control the suspension and activation of tasks, and to synchronise the transfer of process control block information between the microcontroller and the minicomputer (in accordance with Section 3.3.4). Secondly, in the case of the sampling task, the microcontroller had to be informed when this had been completed. This was achieved by the completed analogue-to-digital task causing an interrupt to be sent to the microcontroller (see Section 3.3.7). At this point, the relevance of structuring the microcontroller in a multi-level, vectored-interrupt manner becomes obvious!

4.4.3 Experimental Configuration III : A Multi-Task, Real-Time Control System

The two preceding configurations led up to this final system. The object of this last exercise was to create an environment which was, as far as possible, representative of an on-line, real-time process-control application. The configuration was by no means as complex as a typical, industrial system, but it did have the essential component attributes. As has been stressed throughout, the experimental configurations were designed to be of practical significance, while still being simple enough to permit objective evaluation.

A five-task system was designed, each task being of specific importance. The tasks may be described as follows, while Figure 4 shows the system block diagram.

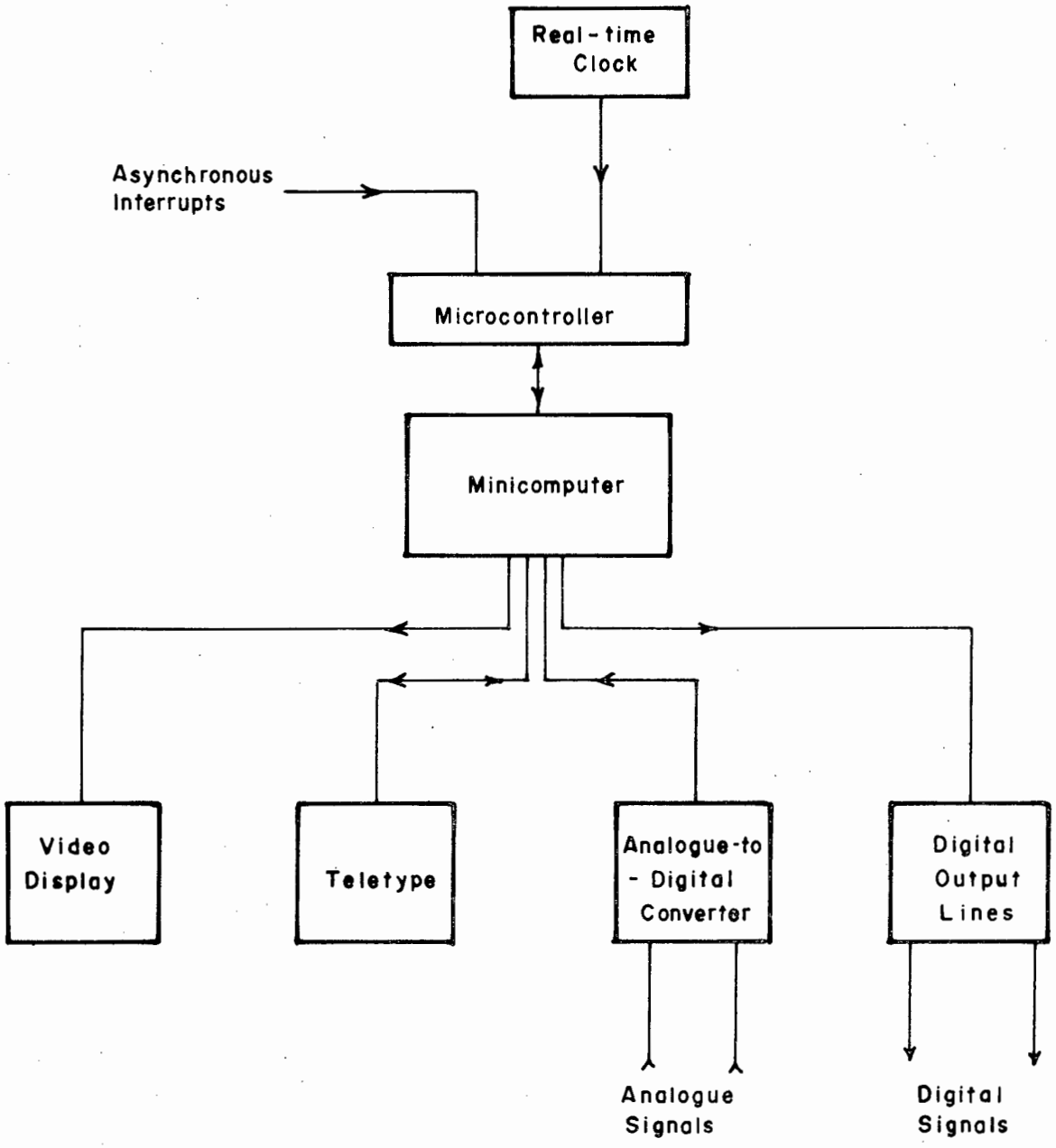


FIGURE 4 : THIRD EXPERIMENTAL CONFIGURATION

(i) Task 1: Determination of the Average Value of an Analogue Signal

This task was designed to sample an incoming voltage signal by means of an analogue-to-digital converter. The sampling rate was determined by an external clock signal, applied to the microcontroller's interrupt control unit. The task performed an averaging of the sampled values over a period of 100 times the sampling rate. Having calculated this average, the task requested the execution of Task 2.

(ii) Task 2: Output of a Digital Value at a Predetermined Time

This task is important, since it shows the possibility of passing information between two tasks. The object was to take a value calculated by Task 1, to scale this data, and to send the resulting value to an external device. In the actual configuration, the external device was a video display unit, where in a practical, on-line application the data would be sent to perform a specific function in the process under control. In other words, in a closed-loop control situation, Task 1 would, say, sample various process parameters. Task 2 would send a suitably-calculated parameter back, for example, to the input to the process, in other words, "closing" the control loop.

(iii) Task 3: Response to an Asynchronous Event

The object of this task was to demonstrate the ability of the system to cope with totally asynchronous, or non-periodic, real-time requirements. The task was requested by means of a random interrupt signal, applied to the interrupt control unit. Once in execution, the task observed the current value sampled by Task 1, and caused a signal to be sent to a digital data-output unit. The logical state of the signal was determined by the corresponding value obtained from Task 1. This also required, therefore, the inter-task exchange of data.

(iv) Task 4: A Background Task

An in-core background task, consisting of the execution of a BASIC Interpreter program, was selected as the fourth task. This task was, therefore, executed at any time when the processor was free.

(v) Task 5: Power Fail/Restart Control

Such a facility is naturally of great importance in any practical situation. The task was required to provide for an orderly shut-down procedure, in the event of a power-failure, of both the microcontroller and the mini-computer. On subsequent power-restoration, a suitable start-up procedure had to be implemented, so as to resume action from the point at which shut-down occurred. (It should be noted that in many practical situations, in fact, the start-up point must be closely examined, as it might require a certain degree of re-initialization of the process under control, or operator intervention might be required).

It may be seen that the five tasks outlined above, although limited in complexity, are truly representative of an on-line control application {56}. In all, they provide for the foreground processing of information obtained from the "system under control", and the supply of information back to the system, in both a periodic and a non-periodic mode. They also provide for background processing, and cater for power-failure occurrences.

From the point of view of the scheduling, limited as it was, the configuration provided for a clear evaluation of the problems encountered in real-time applications. Appendix H discusses these problems, and indicates methods which could be used in the design of the scheduling algorithm. In the first place, it was obvious that the power-fail routines should have the highest priority, and the background tasks the lowest. Other constraints to be applied were determined by the physical requirements. For example, the sampling of information via the analogue-to-digital converter had to be implemented by means of a "non-interruptable" task. This arose from the nature of the converter used, as once the converter had been activated, the task had to wait for a specified time for the conversion to be completed before the input of the data could occur. Any interruption in this sequence could well have caused a loss of data. It was decided that Task 2 was of relatively low priority, and could be interrupted. The major decision was to decide the relative priorities of Task 1 and Task 3. On the basis of the theory developed in Section H.2, the shorter of these tasks was given the higher priority, and in practice this was Task 3. So, in all, the priority assignment was:

Priority level 5 - Task 5 (highest)
Priority level 4 - Task 3
Priority level 3 - Task 1
Priority level 2 - Task 2
Priority level 1 - Task 4

The actual microsequences required to implement the overall system are best understood by observing the flow-charts given in Figures 5, 6 and 7. Included in these flow-charts are the algorithms used to effect the "swopping" of tasks - the suspension of the current task, the obtaining of its current process control block (PCB) data, the sending of the next task's PCB data, and the activation of this next task. Figure 8 shows the local memory structure.

In order to simplify the understanding of the flow-charts, and to stress the "interrupt-driven" nature of the operating system implemented, the interrupts which are received by the microcontroller are grouped into four distinct classes, viz:

Class 1 - Real-time clock interrupts
Class 2 - Interrupts from the process. (This includes the asynchronous interrupt to control Task 3).
Class 3 - Interrupts from the minicomputer, including those necessary for communication synchronization, "end-of-task" indications, and requests for other tasks (such as the request by Task 1 for the execution of Task 2).
Class 4 - Power fail/restart interrupts.

In essence, the microcontroller idled until an interrupt occurred, and then determined the appropriate action to be taken, based on the vectored address of that interrupt.

4.5 Conclusions

This chapter has introduced the subject of evaluating the performance of a computer system. It has been seen that the procedures to be adopted are heavily dependent on the application. In order to provide a mechanism for evaluating the concepts presented in this dissertation, an evaluation strategy was adopted, based on the application of the hardware-based real-time operating system in a series of practical situations. In order to undertake this exercise effectively, suitable hardware and software had to be developed. It was of great importance that the design should be aimed at producing situations which were representative of real-world problems, yet capable of being analysed in a meaningful and quantitative way. In addition,

to show the virtues of the concepts on which this thesis is based, it was essential that the applications should be designed in such a way that the results obtained could be compared with those which would result from the use of conventional techniques.

Finally, of key importance in each configuration was the scheduling algorithm. These have been discussed on the foundation laid in Appendix H.

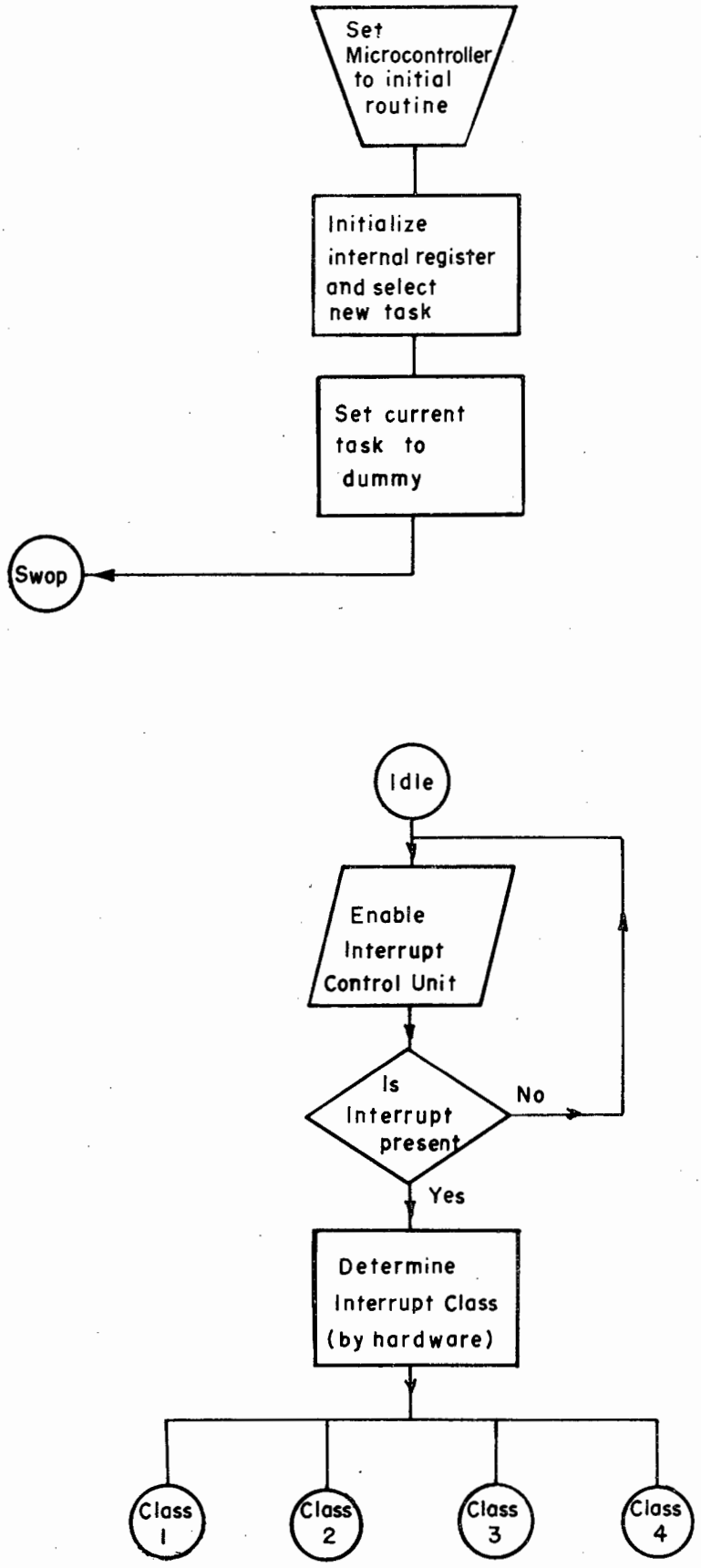


FIGURE 5 : INITIALIZATION AND IDLE ALGORITHMS

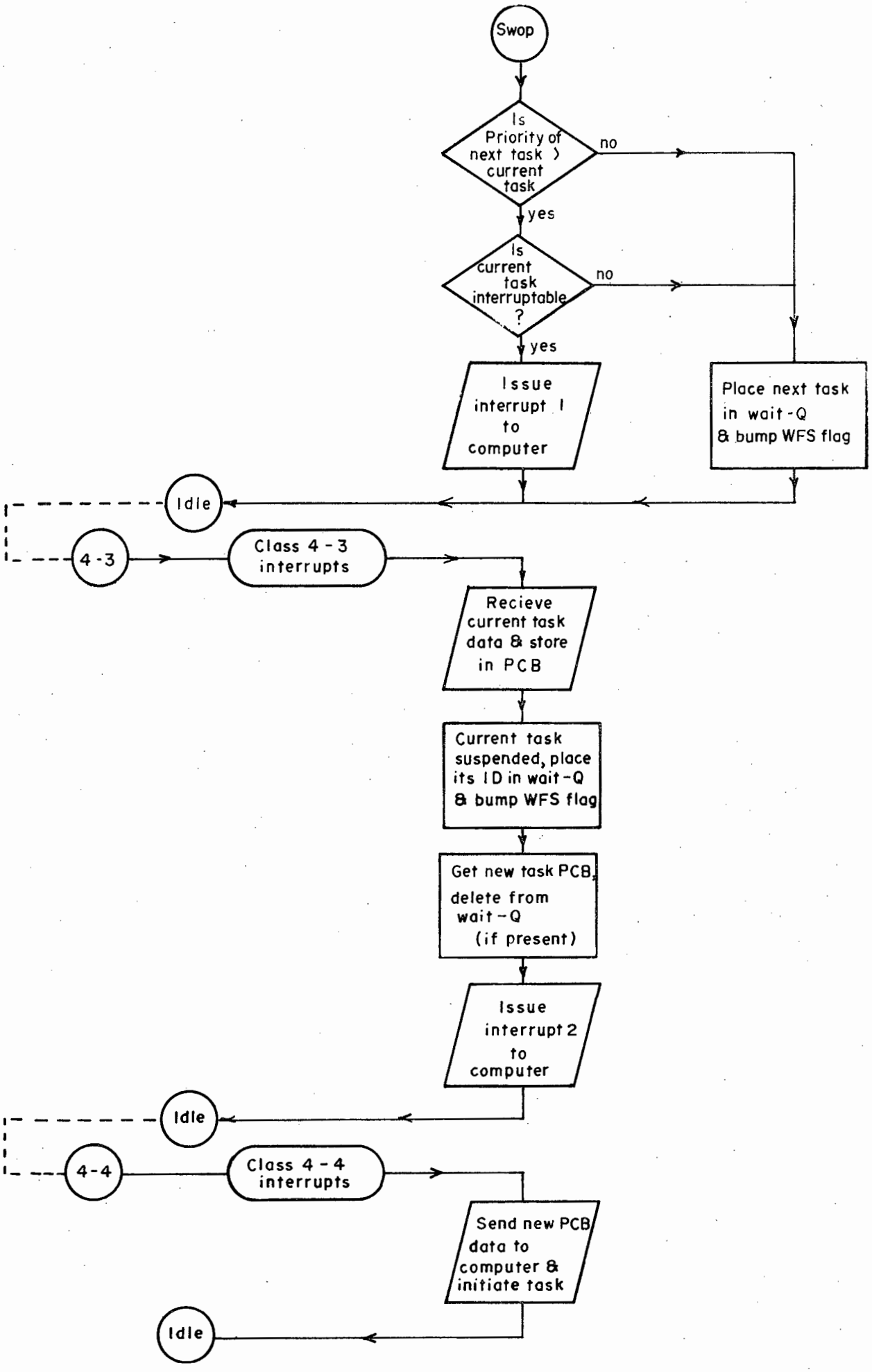


FIGURE 6 : THE SWOP ALGORITHM

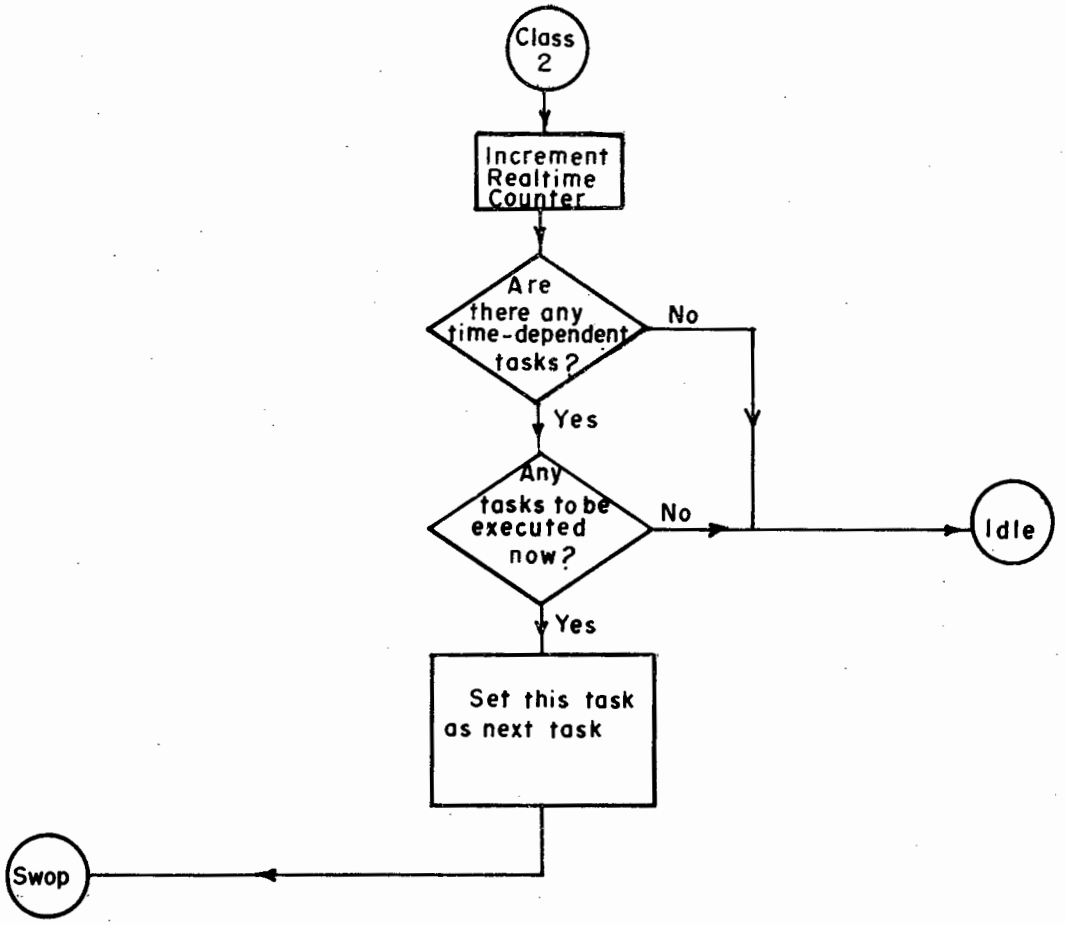


FIGURE 7 (a): REALTIME CLOCK INTERRUPTS

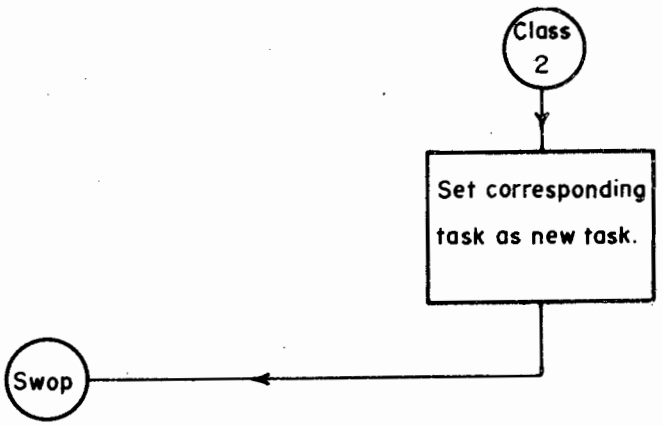


FIGURE 7 (b): PROCESS INTERRUPTS - CLASS 2

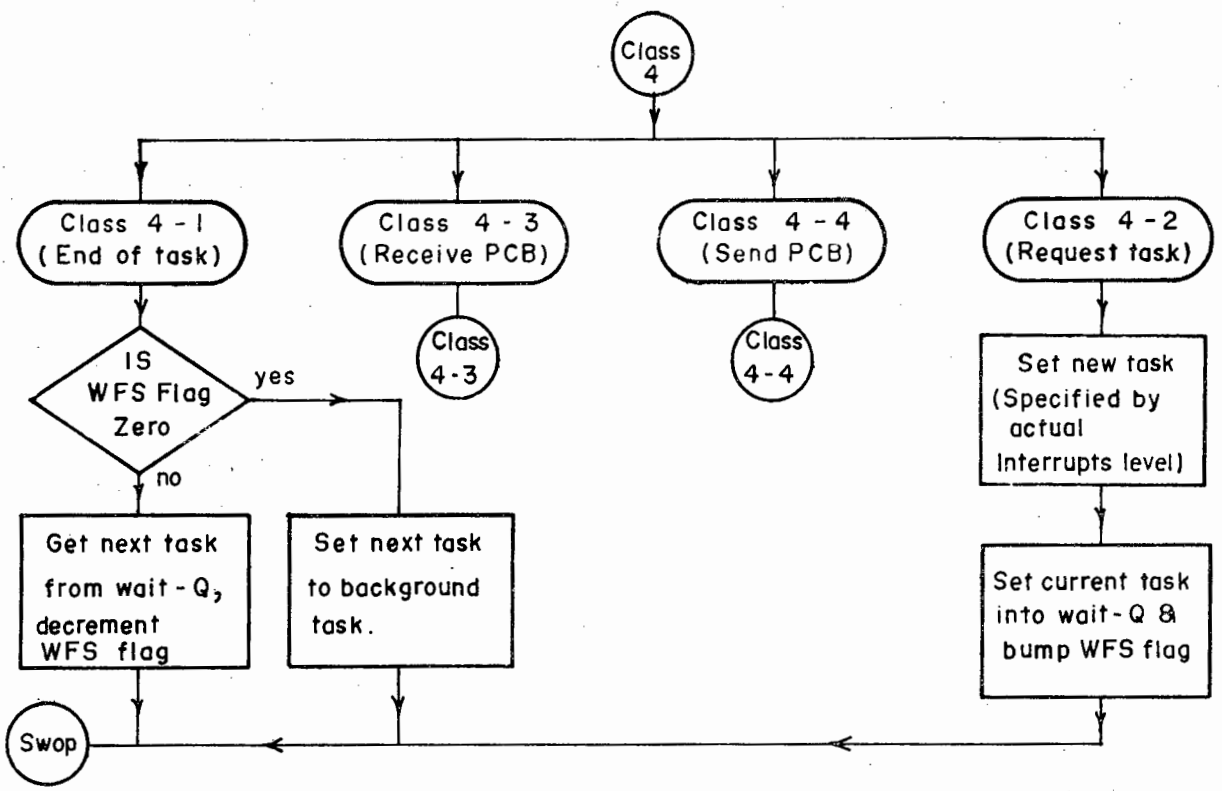


FIGURE 7 (c): INTERRUPTS FROM MINICOMPUTER - CLASS 4

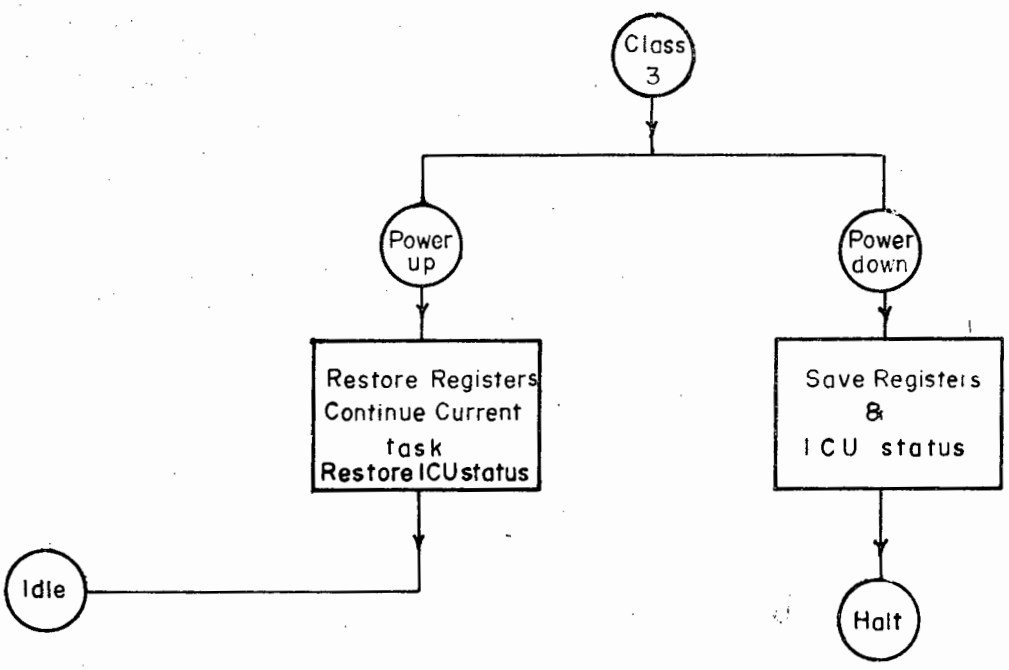


FIGURE 7 (d): POWER FAILURE / RESTART INTERRUPTS - CLASS 3

LOCATIONS	CONTENTS	DESCRIPTION
0	WFS FLAG	No. of tasks in Wait-Q
1	TDT FLAG	No. of time-dependent tasks
2 ↕ 7	SYSTEM CONSTANTS	I/O addresses, etc.
10 ↕ 33	PCB ADDRESS TABLE	Tasks identified by actual position in Table. i.e. location 10 = task1 „ 11 = task2 Priority is also specified within the contents of these locations
34 ↕ 57	WAIT Q	Priority-ordered Queue for all Suspended tasks which are awaiting execution
60 ↕ 77	TDT TABLE	Specifies tasks (& times) which must be executed at a particular time.
100 ↕ 313	PCB's	Each Process Control Block will contain the following information: -register contents for execution -priority, & whether interruptable -disc address (if applicable)
314 ↕ 360	TEMPORARY STORAGE	Used by Microcontroller
361 ↕ 377	SAVE FILE	Used by power-fail microsequence

FIGURE 8 : TYPICAL LOCAL MEMORY MAP

CHAPTER V

ASSESSING THE SYSTEM

> But the Judge said he never had summed up before;
> So the Snark undertook it instead,
> And summed it so well that it came to far more
> Than the Witnesses ever had said!
>
> Lewis Carroll, *The Hunting of the Snark (Fit the Sixth)*

The stage has now been set for assessing the hardware-based, real-time operating system. In Chapter IV, evaluation strategies are discussed, and an objective analysis of the ensuing experimental configurations is contained in Appendix I. There are, in addition, many philosophical and subjective aspects to the evaluation, which are of great relevance in determining the viability of the proposed hardware-based system.

There is one overriding criterion which must be applied in assessing the value of such an innovation: "Does the technique offer its users sufficient advantages to warrant a major revision of the principles on which their existing computer systems are based?" It is important to identify the "users" in question, since, while the broad concepts proposed are widely applicable, this dissertation is primarily concerned with computer applications in on-line process-control. The viability study must, therefore, be concentrated on this area, and relevant comparative studies undertaken.

5.1 Inherent Advantages of the Hardware-Orientated Approach

From a practical viewpoint, any computer-system user should ask the question, "How much of the capital invested in the system will be used for productive purposes, and how much is being consumed in keeping the system operative?" To all intents and purposes, this amounts to the question of overall computer efficiency, as discussed in Chapter IV. The strictly non-productive work results from the apparent need to make use of an operating system. As has been described in Chapter II, the operating system is responsible for overall supervision of the computing system. Conventionally, the operating system is effected within the same hardware structures as the programs which perform the on-line process control functions; a situation akin to the managing director of a company having to share the one available workbench with his artisan! The idea presented in this thesis aims at giving the managing director his own desk but not a whole new factory, as would be the case in a multi-processor configuration.

It should be noted, though, that in many cases the provision of an operating system may not be strictly necessary. This question must be carefully investigated at an early stage of the system design. Such an investigation might reveal, for instance, that a very simple monitoring

program is all that is required. It might be most expedient for the user to develop this program himself, instead of purchasing a full, generalised operating system.

An examination of the structure of a conventional, real-time operating system reveals certain essential features, which must be carried out, but which are strictly non-productive. The most obvious of these are the scheduling of tasks and the supervision of incoming interrupt signals. In a typical, event-driven, real-time operating system (see Chapter II), the occurrence of an interrupt normally requires the computer to suspend its current activity, to examine the interrupt, and to decide what action must be taken. In most cases, the "action" will simply be to "remember" the occurrence and to resume execution of the current task. This is the first area in which the hardware-based operating system has great advantages. The microcontroller is responsible for both the detection of incoming interrupts and the determination of the resultant action which must be effected. Thus, a task which is currently being executed will only be interrupted if execution must be changed to a new task. As is shown in Appendix I, in contemporary, real-time, operating systems, the operation involved in detecting an incoming interrupt and effecting a "task-swop" can take up to 2,5 milliseconds. In the case of the microcontroller, the detection of an interrupt occurrence is extremely rapid, and if a task-swop is required, this may be completed within approximately 173 microseconds of the occurrence of the interrupt.

As well as providing a far higher overall interrupt-response-time, therefore, the microcontroller avoids the wastage of valuable computing time which must occur when the computer has to deal with the interrupts by itself. The faster response-time to interrupts is intrinsically of great virtue. In many process-control applications, it is vital to deal with certain classes of interrupt (for example, an alarm indication) as expediently as possible. Such high-priority interrupts normally have to be catered for outside the conventional, real-time operating systems in order to achieve sufficiently rapid response-times. (A typical example would be the handling of power-failure conditions).

There is an additional factor to be examined, the question of memory utilization. In contemporary real-time process-control operating systems, the operating system is resident in the same memory as the process-control programs. In most systems, in fact, in order to reduce this "core-overhead" (as it is usually referred to), only the bare essentials of the operating system will be resident, the rest being stored on a back-up storage device (such as a magnetic disc), and being brought into the memory as and when required. The resident portion, however, can still be significantly large. In the case of a minimal, real-time operating system such as the Data General RTOS [58],

this amounts to a minimum of 5000 memory locations. For a larger system, such as the Interdata OS/32 {59}, or the Data General RDOS {28}, this will amount to a minimum of approximately 12000 memory locations. In terms of current memory costs this might not be very significant, but it must also be remembered that the user will be restricted by certain memory limitations on the placement of programs.

This "core-overhead" is greatly reduced when adopting the proposed strategy. The only computer memory locations involved are those which are used to store the routines necessary for the passing of process control block information between the computer and the microcontroller. This amounts to a mere 76 locations, giving an obvious advantage to the user, since he may therefore make use of virtually the total available memory capacity.

Utilization of the available peripherals must also be considered. The supervision of peripherals is delegated to the microcontroller, and in practice the interrupt-handling capabilities of this unit are exploited to achieve a higher system efficiency. The improvement results from the high response-times of the microcontroller. The peripherals will be under the control of specific tasks, which are scheduled by the microcontroller in the same way as user tasks. For example, a request by a user task to access a peripheral will be initiated by means of an interrupt to the microcontroller. The microcontroller can initiate the peripheral-handling task, and then bring another task into execution while it waits for the peripheral to become available. When the peripheral indicates its availability by means of an interrupt to the microcontroller, the peripheral-handling task may be completed. A higher overall system efficiency will therefore result. The idea is, of course, similar in nature to the concept of "spooling", which is used in many contemporary computing systems {25}.

5.2 Advantages of the System in Practical Applications

To analyse the practical benefits to a user of the proposed strategy, it is useful to discuss first how a realistic, application-orientated system may be configured. The interaction between the user and the system may then be reviewed in this context.

The third experimental configuration described in Chapter IV and analysed in Appendix I reveals the relative simplicity of applying the strategy to a real-time process-control environment. Very few modifications are required to apply the techniques used in this experiment to a typical industrial situation. The basic scheduling algorithms will remain essentially the same, as will all the other techniques developed. The only modification which will be necessary is the inclusion of any additional tasks which might be required in a typical, real-time process-control application. To make this change, two steps are required. Firstly, the local memory must be expanded to meet the required capabilities. (The local memory map shown in Figure 8 shows that 20 tasks

may be specified within a 256-word memory). Secondly, the scheduling algorithm must be checked to reveal if it is "feasible" for the priorities assigned to each task. It is important to note, however, that no modifications will be required to the actual microsequences stored in the micromemory.

In a real-world application, the microsequences would be permanently written in a Read-only micromemory. Only radical changes (requiring a modification of the overall operating-system philosophy) might imply an alteration of these microsequences. This is itself not necessarily too formidable a task, as the system can be dynamically microprogrammed [34], provided that the micromemory is suitably structured, new microsequences being generated by the associated minicomputer. In most real-time process-control situations, in fact, it is generally experienced that the only regular changes encountered are either in the addition and subtraction of tasks, or in the altering of priorities. This might seem to be a sweeping statement, but it must be remembered that the term "tasks" is, in the context of this work, taken to include the programs which are used to control peripheral equipment, as well as the user's process-control programs. Thus, the addition of a new piece of peripheral equipment will strictly require only the addition of (a) new task(s) to the system. One limitation must, however, be applied, arising from the handling of interrupts. A particular microcontroller configuration will have available a predetermined number of interrupt lines, together with the associated handling routines incorporated within the microsequences. Thus, a particular configuration can handle a specific number of interrupts, and any additional requirements will necessitate a hardware modification and a microprogramming change. The availability of dynamic microprogramming and the simplicity of extending the interrupt control unit, facilitate these changes.

To add or subtract tasks, or to change priority levels, the user has to change the contents of the local memory. This is a straightforward procedure, and may be effected either by a direct exchange of information between the computer and the local memory, or by a data-transfer controlled by a microprogram. The latter facility was provided as a standard feature of the prototype system. A selected interrupt signal was used to initiate a microsequence which transferred the contents of the local memory to the minicomputer, or vice versa. Thus, the contents of the local memory could be rapidly examined and suitably modified, under the control of the minicomputer.

An important point to consider is the allocation of the available minicomputer memory to user programs. In contemporary systems, the operating system normally determines where user programs are to reside in the computer's memory. As mentioned before, the subject of "memory management" is a very broad one, and Reference [53] offers a valuable introduction. This subject

did not escape attention when the structure of the proposed operating system was investigated, and two approaches were explored. Firstly, a conventional approach could be adopted. The operating system implemented in hardware by the microcontroller could be expanded to encompass memory management. Thus, in the case of a disc-based system, the microcontroller would determine where tasks being read from the disc should be placed in the minicomputer's memory. This meant that the microcontroller had to have available within its local memory a "map" of the minicomputer's memory, showing which tasks were stored where. Of course, tasks would have to be programmed in a re-entrant, re-locatable form, so that they could be located anywhere within the minicomputer's memory. Arising from this concept is an area of research which would prove most fruitful to pursue. The techniques known as "virtual memory" structuring have attracted much interest, having first been presented in 1961 by a group of computer scientists, working on the now-legendary "Atlas" computer system at Manchester, England {60}. The ideas were formalised in Denning's paper in 1970 {61}. The principle involved makes available to the computer user a larger memory capacity than that which physically exists. The idea is normally implemented by {61} by storing all the user programs on a back-up disc storage device, the programs being structured into "pages" of suitable lengths, normally 1 Kiloword. When a memory address is specified, suitable hardware {62} is used to modify this "virtual" address to a physically-realisable one (i.e. one which is available within the computer's actual memory-space). The "page" which contains the requested address must be loaded into the computer's memory, if it is not already there, and execution may continue. This amounts to address modification, i.e. the mapping of theoretical (or "virtual") addresses onto physically realisable ones.

The hardware required to implement such a virtual memory system is relatively extensive. In addition to address modification, the hardware must maintain tables which reflect the state of the computer's memory. Also, algorithms will have to be developed to decide which "pages" of memory are to be written onto the back-up storage device, so as to make space available for new "pages", when required. (Obviously, some of these procedures could be effected by programs held within the computer's memory, but with obvious degradation of the overall computer system performance.)

On reviewing the potential capabilities of the microcontroller system developed in this study, the possibilities of utilizing it to implement a virtual memory system are very attractive. With a minimum of modifications to the prototype system hardware, namely, an additional interface to the address bus of the minicomputer and a direct control over the computer's master clock signal, the microcontroller may be used to control, in addition to the functions required to implement the specified real-time operating system,

a virtual-memory configuration. While such a system does not form a central feature of this dissertation, it has been considered in depth and, using a parallel research project {63}, has been shown to be feasible. The facility was not included within the proposed microcontroller because of the architecture of the minicomputer to which the microcontroller was applied. Unlike the PDP-11 {64} (which was used for the studies described in {63}), the Varian 620 range of computers {50} presents an address-bus structure which, without major hardware design, may not readily be accessed. (The Varian 620 computer series has a totally-synchronous memory-addressing scheme, whereas the PDP-11 is representative of computers designed with an asynchronous architecture). Thus, without extremely hazardous redesign or the use of alternate computer systems, the inclusion of virtual memory techniques within the hardware-based, real-time operating system had temporarily to be discounted.

The second approach which could be adopted to overcome the memory allocation problems was that employed in many "in-core" real-time operating systems {58}, and one which may be applied (within limitations) in a disc-based system. This is the allocation of user tasks to fixed and predetermined areas of the available memory space. If a back-up storage device is used, techniques similar to those used in the second experimental configuration (see Section 4.4.2) may be adopted. This means that if a task is to be loaded into the computer's memory, it must be placed in a predetermined space, after tasks which have previously occupied this area of memory have been stored on the back-up storage device (a technique often referred to as "overlaying"). This is, admittedly, not a very efficient system, but it does in practice have certain advantages to the user. It means that programs may be developed in a relatively straightforward fashion, and easily verified.

This system has the implication that the addition and subtraction of tasks must be carried out with a certain degree of care. A memory-map must be physically maintained so as to prevent the destruction of current programs when new tasks are added. In practice, new tasks will normally be prepared in a "relocatable" form (that is, they may subsequently be loaded anywhere in the memory), the length of the task determined and the task finally loaded into an available space.

Returning to the main point of this section, what are the other advantages which the proposed hardware-based system offers to the user? The

user of a process-control computing system does not really care what method is used to implement memory-management, etc. Of great concern to him are matters such as reliability, repeatability and security.

The question of hardware and software reliability is a direct function of the computer system selected, and of the care which is taken in the preparation and verification of the process-control programs. These subjects are well-covered in the relevant literature, and of particular value are References {65} and {66}. It is important to consider whether a hardware-based operating system leads to an improvement in the areas mentioned.

From a hardware point of view, it cannot be denied that reliability is a function of the number of physical components used in a system {65}. This does imply that the additional hardware required to implement the techniques presented will apparently lead to a decrease in reliability. In terms of the inherent reliability of modern integrated-circuit technology, however, this problem is relatively insignificant. If a survey is undertaken of problems which are met in on-line process-control applications, three areas appear to account for the major sources of failure. These are:

(i) The interface between the computer and the process being controlled

(ii) The computer's peripheral equipment, notably such items as discs and printers

(iii) Software errors (or "bugs") {67} {68}

The first two areas above will not be altered by the use of a hardware-based operating system. It is only the third point which must be reviewed. To requote Dijkstra, "Program-testing can be used to show the presence of bugs, but never to show their absence". {11} This is an ever-present problem in the on-line control area.

(An illustration of the type of software problem which arises is the following: at a large, submerged arc-furnace plant, an on-line computer control system had been in use for 18 months. It was noted that about once a month the computer would not respond to an incoming signal which requested a certain action to be effected. All the available diagnostic programs were run, and the system was checked and rechecked, but there was no obvious source of this error. After six months of testing, and in sheer desperation, the computer engineers started examining the real-time operating system. This was the standard system offered by a leading, world-wide minicomputer manufacturer, and had been in use in thousands of installations for over five years. The problem was finally traced to the interrupt-handling routines within the operating system. If two interrupts of equal priority occurred simultaneously, one of them would be disregarded! A chance occurrence, but one which would be extremely difficult to check for in software). {68} The implication is that in a highly-complex operating system, it would be extremely

difficult to ensure that no errors would ever occur under any possible circumstances. Also to be considered is the possibility of sections of the operating system, or even single locations, having their contents changed, either by the operating system itself, or by user programs.

All this discussion leads back to one of the points made in Chapter I, when the advantages of implementing software functions in hardware were dealt with. The hardware-based operating system would normally be permanently stored in a totally non-destructive memory, and no modification could therefore possibly occur. From the viewpoint of latent errors which might exist in the microprogramming of the hardware unit, the extremely low level at which the microsequences must be developed implies a very close relationship between programmer and hardware. So, together with the relative simplicity of the microsequences, the resultant microprogramming can be extensively tested and an error-free (virtually - remember Dijkstra! {11}) system produced. Also, once the microinstructions have been permanently placed in the read-only micromemory, the microcontroller can be subjected to automatic testing, similar to methods currently being used in the verification of microprocessor systems {69}.

It is therefore claimed that a hardware-based operating system can be far more reliable than an equivalent software-implemented one. The basis of this claim lies in the use of non-volatile program storage.
As a corollary to this, with this inherent reliability, the repeatability of the on-line control system would be improved. Repeatability is of prime concern to the user and a major factor in software design. With little or no possibility of the operating system being corrupted during the execution of tasks, and because of the simplicity of the microinstruction procedures, a very high degree of repeatability is to be expected.

"Security" is a term at present receiving much attention in the computer world {70}. The aspect of this subject which is of importance in on-line control is the potential modification of the computer system which could occur under any circumstances, and which would result in the making of erroneous decisions. While the proposed system could not alter the potential security of the computer and its environment, the definite control extended over these by the hardware-implemented operating system is clearly seen to be an immediate advantage. Also of importance to the overall security of the process under control is the rapid response of the operating system to external events, such as alarm conditions.

What is of prime importance in all the above considerations is the clear definition of the operating system when it is implemented in hardware. It is this fact which makes the concept most viable when applied to real-time process-control environments.

5.3 The Disadvantages of the Hardware-Based System

Any practical application of science and technology will be the result of compromise. The ideal solution to any problem is never really possible to achieve. There must always be a balance between cost and practicality, between simplicity and efficiency... the list of compromises is a long one. This dissertation presents an alternative approach which may be used in solving the problems arising from the need to supervise a complex computer system in a particular field of application. The preceding sections of this chapter have discussed the positive attributes of this approach; there must be a similar list of its disadvantages to achieve a balanced view.

The first of these disadvantages is applicable to all proposals which require a major reorganisation of a widely-used technique. This revolves around the "inertia" which exists in an industry committed to producing a specific product in a specified way. The adoption of the technique proposed here would require the computer industry (particularly that section involved in the production of small computers used in process-control applications) to make major changes. Many man-years of software development would have to be written off, and a new form of hardware brought into production. This, fortunately, is not as great a change as might be anticipated. The basic philosophies relevant to the structuring of operating systems are still applicable, albeit in a modified, simpler form. It has also been demonstrated that within the framework of current technology the new hardware required is minimal.

Considering the aspect of cost, any new technique which is introduced will tend to increase the price of the article. This is basically so because research and development costs have to be recovered. In the long term, the extra complexity of the hardware would probably balance out the simplification of the software. In realistic terms, the current purchase price of a sophisticated minicomputer real-time operating system is of the order of R3000. A reasonable cost for the replacement hardware unit would be a similar figure, the basic component cost of the system currently developed amounting to some R900.

The complexity of the hardware-based system must also be considered with regard to versatility. If it is necessary to alter the scheduling algorithm at a level beyond mere priority modification, the user is required to be a competent microprogrammer. The reluctance of users to utilise user-microprogrammable facilities offered in some current computers is indicative of the problems involved [34]. It is by no means an easy task, even using the software aids which are currently available [71]. The microprogrammer must have a very deep insight into the structure which he is manipulating, and must fully understand the overall operation of the system. The micro-

programs produced must be highly efficient, especially if (as is the case in the majority of microprogrammed computers) a limited memory is available for the storage of the microinstructions. Thus, the only practical solution seems to be for manufacturers to produce hardware-based operating systems which are tailored to various probable applications. (This principle is one which users have discussed at length. Most available software, and in fact hardware, systems are general-purpose and, by definition, do not provide the optimal solution to a specific application. Reference {72} discusses these problems in depth). The users would then be restricted to work within the framework provided. The modifications which would, typically, then be available would cover the specification of tasks which are to be included, and of their relative priority, etc.

One practical aspect which is highly relevant is the question of power dissipation. The hardware-based operating system must be capable of operating at high speeds, or many of the key advantages offered by the approach will be lost. This means that high-speed digital circuitry is required. It is an unfortunate fact that the power dissipated by a semiconductor device tends to be a function of the square of the speed of operation. Thus, as is demonstrated in the microcontroller discussed in this work, in order to achieve satisfactory speed performance, the circuits which must be used have extremely high power-requirements and subsequent power dissipation. In the prototype system, the power consumption of the microcontroller was greater than that of the central processor and the memory of the minicomputer. There is no immediate solution to this problem, though the development of similar-speed, non-saturating digital circuitry of the emitter-coupled type {73} might well provide the answer in the near future.

5.4 Conclusion

The objectives of this chapter have been to analyse the proposed hardware-based real-time operating system in terms of its application in on-line process-control. The measurements made when using the prototype microcontroller in an experimental real-time control configuration have demonstrated the material advantages of the technique. It is, however, in the overall system philosophy that the concept may be seen to hold great promise. Taken from a global view, it leads to greater reliability, ensured repeatability and higher security.

The strength of the technique lies predominantly in two aspects: the inherent rapidity of response, and the removal of the operating system from the host computer. The latter advantage provides for a computer management mechanism which is virutally incorruptible, and which imposes a minimal non-productive workload on a processor which has on-line control functions as its prime objective.

CHAPTER VI

CONCLUSIONS

"There is Thingumbob shouting!" the Bellman said.
"He is shouting like mad, only hark!
He is waving his hands, he is wagging his head,
He has certainly found a Snark!"

Lewis Carroll, *The Hunting of the Snark*
(Fit the Eighth)

In January, 1973, the IEEE Computer Society sponsored an extensive workshop on the interaction between operating systems and computer architecture {1}. The need for this interchange of ideas arose from the growing concern among computer technologists about the increasing lack of co-ordination between these two vital components of a computer system. The operating system is probably the most extensive and complicated program that has to be developed for a computer system, but the hardware design of computers rarely takes into consideration the needs of the operating system. Once a computer has been designed, the programmers have to do their best to create an operating system which utilizes the facilities to the greatest advantage. It was noted that the hardware-controlled scheduling of a single processor did not seem to be receiving any attention {18}.

Chapter I of this work provides a statistical means of ensuring efficient organization of industrial control computers, and makes it clear that the provision of bigger and better system facilities will not necessarily result in improved system organization. In order to improve organization, and to provide for maximum utilization of the facilities offered (i.e., efficiency), there are certain clear-cut factors which may be considered. The various aspects of these factors indicate that an unconventional, integral, hardware/software approach to the management of industrial control computers would ensure highly-efficient operation. The engineer involved in the use of computers in the sphere of industrial control requires the availability of a highly-efficient, reliable and simple controlling device. In the early days of on-line control, large, general-purpose computers were used, and the resulting system-complexity led in many cases to great disillusionment. The control engineer then turned to minicomputers (offering, as they do, relative unsophistication and simplicity) to solve his problems. The rapid escalation in the minicomputer industry bears witness to the success of this approach.

Nevertheless, to any engineer, one of the major objects of system-design is to achieve the greatest possible efficiency. In order to implement the supervision of an industrial process, many different parameters must be monitored or controlled, normally under the overall command of a single

computer. To achieve this, sophisticated software must be available to allow for the "scheduling", or selection, of the various programs needed to effect the control functions. The software which carries out this supervision is the real-time operating system. In plain terms, it is that component of a computer's software which supervises the computer and its immediate environment. In a realistic evaluation, it is a necessary evil, being totally unproductive so far as the process itself is concerned.

From these ideas were born the concepts which are discussed in this dissertation. The object is to reduce, as far as possible, this operating-system "overhead". When a close examination is made of the actual requirements of a real-time operating system, certain functions are seen to be particularly important. These are the so-called "executive functions" and, in essence, they decide, on a pre-determined basis, what is to be done and when. A typical process-control computer system is event-driven, that is to say, it responds to certain occurrences (generated by external events, or by the end of timing periods, or within programs being executed by the computer). The operating system must acknowledge these events, and take the necessary action. Such events may occur in rapid succession, or even simultaneously, and so the operating system must be able to select the procedure to be followed on an algorithmic basis. Of importance in many applications is the speed of the response to an event. In commercial applications, speed may not be of primary importance, but the picture changes in process-control situations. The time which a computer takes, say, to respond to an alarm indication could be of major significance: a slow response might lead to catastrophic costs.

Based on these considerations, and taking into account the more usual functions of an operating system, the specifications of a representative real-time operating system were drawn up. The object was then to implement these functions by means of hardware.

6.1 The Hardware Structure

There have been numerous attempts to solve the problems mentioned above. Two of these are of significance: the ASI TI Supercomputer [30] and the "SYMBOL" system [16]. In both of these studies the object was to separate the operating system from the rest of the computing system. Both attempts, however, have a common approach which detracts from their practical value. In both developments, the essential principle adopted was to house the operating system in what amounts to an entirely separate, complete processor. In other words, the final system is virtually a dual- or multi-processor configuration. This is by no means a new approach, but one which is adopted (in principle) in many large computer systems. The concept may be extended into the minicomputer area, and two such computers could be suitably interfaced, one executing the

process-control programs and the other the operating system functions.

This technique has the disadvantage of increasing both complexity and the resulting maintenance and support requirements. In effect, two computers are used to do the work of one.

The proposal put forward by this thesis uses a single, simple hardware unit to implement the operating system. The principle is that this unit may then be included within the minicomputer without seriously increasing the overall complexity. The idea becomes feasible in view of current technological developments, especially in the field of large-scale integration, and specifically with the introduction of ultra-high-speed microcomputer components. The facilities which the "hardware" operating system must provide are relatively simple, but must be effected as expediently as possible. It is also desirable that the hardware unit should be capable of modification, so as to meet changing circumstances.

The proposed hardware unit meets these requirements; it is capable of high-speed operation, and may be modified by the user. The key lies in microprogramming, a technique widely explored by the computer architect in designing the control units of many contemporary computer systems. The resulting system is a microprogrammable unit, capable of exerting full control over the minicomputer with which it is associated, and completely replacing a conventional, real-time operating system. Changes to the operating system structure are effected by alterations to the microprograms stored in the unit.

The hardware unit may be contained within a single computer board, included within the mainframe of a minicomputer. It requires no modifications to the host computer.

6.2 An Assessment of the System's Practical Value

In Chapter I it was seen that two primary factors governed the performance, in terms of average delay, of a system. In order to reduce this average delay, two strategies merited attention. The average number of incoming requests could be reduced, and/or the average processing time of the requested tasks could be shortened. Thus, to utilize a single processor to its maximum, while still providing a highly efficient system, the management of the inherent capabilities is the critical factor to be considered. The proposed solution is aimed at the heart of the problem: ensuring that both the controlling parameters for the system (as mentioned above) are held at a minimum. Using a unique, integral, hardware/software approach, a hardware-implemented management mechanism has been proposed.

In order to demonstrate the validity of the technique, a fully-operational system was applied to a conventional minicomputer. Various experimental environments were created, each being representative of different aspects met with in real-time computer systems. Each situation was fully controlled in order that meaningful evaluations could be made. Of particular interest is the comparison made with classically-structured computing systems. Although it is difficult to achieve a high degree of similarity between the proposed system and a classical one, the results indicate the true value of the technique under consideration. In addition, they pinpoint areas which offer much promise as extensions of the principles embodied in the hardware implementation of a real-time operating system. Of immediate interest is the field of memory management, that is, the use of the available computer memory. The basic technique proposed does, in itself, vastly reduce the amount of memory which must be allocated to the operating system and which is therefore not available to the user's programs. In many applications, however, the memory required to contain the user's programs is far in excess of that which is physically available. Thus, a technique must be adopted to permit management of the available memory in such a way that maximum use is achieved for minimum overheads. This management function is normally implemented by means of a suitable hardware unit, sometimes called a "memory mapping unit" or "memory paging device". With the availability of an "intelligent" controller, such as that proposed for implementation of the operating system, it becomes feasible to include the memory management functions within the same unit. Of particular interest is the use of the hardware operating system in a "virtual memory" structure, as discussed in Chapter V. This places the responsibility for memory allocation right where it truly belongs - within the operating system itself.

Another area of interest is the integration of the hardware operating system within a disc-based structure. As discussed in the previous chapter, in many practical situations it is necessary to have available large, back-up storage devices. These would hold both data files and process-control programs which cannot be stored currently within the computer's own memory, normally because of lack of space. It would be possible to incorporate the hardware required to control the storing of data on, and the accessing of data from, the disc, within the hardware structure used to implement the operating system. It might appear that the suggested demands on this unit are excessive, leading to its overloading and subsequent performance degradation. It must be remembered, however, that the unit, as proposed, is capable of extremely high-speed operation, and (as it is being currently applied) has available large periods of free time. Of course, care must be taken to avoid any overloading of the microcontroller, over and above its primary operating system functions.

6.3 Summary

Any advance in the spectrum of engineering is only of significance if it brings an advantage to the person at whom the innovation is directed. In the case of the strategy proposed in this dissertation, there is undoubtedly a benefit to the user - the process-control engineer. In practice, the structure of the operating system may be clearly defined, and will not require him to become involved in the intricacies of microprogramming, as this may be carried out by the system vendor. The process-control engineer is, then, only required to state the bare essentials relating to the programs which he wishes to be controlled by the operating system. He then has available to him a computer system which is capable of scheduling and supervising the application of programs in an extremely rapid and accurate way. It is virtually impossible for the operating system to be destroyed by accidental over-writing, and no operating system software maintenance is required. The solution proposed is undoubtedly cost-efficient, and leads to a high utilization of the computer system installed.

The fundamental principles on which the proposal is founded are themselves of paramount importance. They involve a scientific evaluation of the expected performance of a system, and indicate a means whereby efficient organization may be achieved. As has been discussed above, the concepts show their true value, not only in the computer world, but also in many other aspects of our organization-orientated existence.

Thus, one may look at any inefficient or ineffectual organization, apply the proposed techniques, and arrive at a scientific and rational solution.

The objectives of this dissertation may therefore be seen as two-fold: firstly, to develop a rationalized approach to the study of organizations, with special reference to industrial control computers; and, secondly,to demonstrate how (based on evidence derived directly from the analysis set out above) a single industrial control processor may be organized in a highly-effective way.

REFERENCES

1. Roucek, J.S. and Warren, R.L., "Sociology - An Introduction", Littlefield, Adams & Co., U.S.A. (1966), p.272
2. March, J.G., "Organizations", John Wiley & Sons, Inc. (1967), pp. 1 & 172.
3. Jensen, E.D., "Hardware vs. Software: The Two Faces of Computers", Computer, Vol. 6, No. 11 (November 1973), p. 15.
4. Barsamian, H., and De Cegama, A., "Evaluation of Hardware-Firmware-Software Trade-Offs with Mathematical Modeling", AFIPS Conference Proceedings, Vol. 38, SJCC (1971), pp. 151-161.
5. Mandell, R.L., "Hardware/Software Trade-Offs - Reasons and Directions", AFIPS Conference Proceedings, Vol. 41, FJCC (1972), pp. 453-459.
6. Wilkes, M.V. and Stringer, J.B., 'Microprogramming and the Design of the Control Circuits in an Electronic Digital Computer", reprinted in "Computer Structures: Readings and Examples", edited by Bell, C.G. and Newell, A., McGraw-Hill, N.Y. (1971).
7. Falk, H., "Computer Report I - Hard-Soft Trade-offs", IEEE Spectrum, Vol. 11, No. 2 (February 1974), pp. 34-39.
8. Corinthios, M.J., "A Fast Fourier Transform for High-Speed Signal Processing", IEEE Trans. Computers, Vol. 20, No. 8 (August 1971), pp. 843-846.
9. Anon., "Universal I/O Interface Cuts Interfacing Costs to Processor Level", Computer Design, Vol. 14, No. 2 (February 1975), p. 18.
10. Wang System 2200, CPU Data Sheets, Wang Laboratories Inc., Mass. (1974).
11. Dijkstra, E.W., "Notes on Structured Programming", EWD 249, Technical University, Eindhoven, The Netherlands (April 1970).
12. Havek, E.A. and Dent, D.A., "Burroughs' B6500/B7500 Stack Mechanism", AFIPS Conference Proceedings, Vol. 32, SJCC (1968), pp. 245-251.
13. Schoeder, M.D., and Saltzer, J.H., "A Hardware Architecture for Implementing Protection Rings", Comm. ACM, Vol. 15, No. 3 (March 1972), pp. 157-170.
14. Anon., 'Minicomputers Offer High Speed Cache Memory", Computer Design, Vol. 13, No. 11 (November 1974), pp. 120-122.
15. Bylinsky, G., "How Intel Won Its Bet on Memory Chips", Fortune (November 1973), pp. 142-186.

16. Rice, R. and Smith, W.R., "SYMBOL - A Major Departure From Classic Software Dominated by von Neumann Computing Systems", AFIPS Conference Proceedings, Vol. 38, SJCC (1971), pp. 575-587.
17. Wilner, W.T., "Design of the Burroughs' B1700", AFIPS Conference Proceedings, Vol. 41, FJCC (1972), pp. 489-497.
18. Browne, J.C. and Howard, J.H. (Jnr.), "The Interaction of Operating Systems and Computer Architecture - A Workshop Summary", Computer, Vol. 6, No. 11 (November 1973), pp. 16-18.
19. Denning, P.J., "Third Generation Computer Systems", Computing Surveys, Vol. 3, No. 4 (December 1971), pp. 175-216.
20. Coffman, E.G. (Jnr.) and Denning, P.J., "Operating Systems Theory", Prentice-Hall, New Jersey (1971), p.8.
21. Barron, D.W., "Computer Operating Systems", Chapman and Hall, London (1971), p. 44.
22. Dijkstra, E.W., "Co-operating Sequential Processes", reprinted in "Programming Languages", edited by Genuys, F., NATO Advanced Study Institute, Academic Press, London (1968), pp. 43-112.
23. McKinney, J.M., "A Survey of Analytical Time-Sharing Models", Computing Surveys, Vol. 1, No. 2 (June 1960), pp. 105-116.
24. Součeк, B., "Minicomputers in Data Processing and Simulation", Wiley-Interscience, U.S.A. (1972), pp. 271-291.
25. Brinch Hansen, P., "Operating System Principles", Prentice-Hall Inc., N.Y. (1973).
26. Purser, W.F.C. and Jennings, D.M., "The Design of a Real-Time Operating System for a Minicomputer - Part I", Software - Practice and Experience, Vol. 5 (1975), pp. 147-167.
27. McClure, R., "Hardware/Software Trend: Think Small", Datamation, Vol. 22, No. 4 (April 1976), p. 109.
28. Real-Time Disc Operating System - User's Manual, Data General Corporation, Mass. (1973).
29. Private Communication with the Managing Director, Perseus Computing and Automation (Pty) Ltd., Pretoria, R.S.A. (1975).
30. Watson, W.J., "The TI ASC- A Highly-Modular and Flexible Super Computer Architecture", AFIPS Conference Proceedings, Vol 41, FJCC (1972), pp. 221-228.
31. Staff Production, "Microprocessors: A Special Edition", Electronics International, Vol. 49, No. 8 (April 15th, 1976), pp. 74-174.
32. Rattner, J., Cornet, J. and Hoff, M.E., "Bipolar LSI Computing Elements Usher in New Era of Digital Design", Electronics, Vol. 47, No. 17 (September 5th, 1974), pp. 89-96.
33. Flynn, M.J. and Rosin, R.F., "Microprogramming: An Introduction and a Viewpoint", IEEE Trans. Computers, Vol. 20, No. 7 (July 1971), p.727-731.

34. Jones, L.H. and Merwin, R.E., "Trends in Microprogramming: A Second Reading", IEEE Trans. Computers, Vol. 23, No. 8 (August 1974), pp. 754-758.

35. Varian 73 Series, Processor Manual, Varian Data Machines, California (December 1972).

36. Mick, J.R., "AM 2900 Bipolar Microprocessor Family", Proceedings, IEEE, Eighth Annual Workshop on Microprogramming (September 1975), pp. 56-63.

37. Agrawala, A.K. and Rauscher, T.G., "Microprogramming: Perspective and Status", IEEE Trans. Computers, Vol. 23, No. 8 (August 1974), pp. 817-837.

38. Hewlett-Packard 2100, Processor Description, Hewlett-Packard Corp., Calif. (July 1972).

39. Wakerly, J.F. et al, "Placement of Microinstructions in a Two-Dimensional Address Space", Proceedings, IEEE, Eighth Annual Workshop on Microprogramming (September 1975), pp. 46-51.

40. Series 3000 Reference Manual, Intel Corporation, Santa Clara, California (1975).

41. Chapin, N., "Computers - A Systems Approach", Von Nostrand Reinhold Company, New York (1971), p. 303.

42. Hoff, M.E., Sugg, J. and Yara, R., "Central Processor Designs Using the Intel Series 3000 Computing Elements", Intel Corporation, Santa Clara, California (1975).

43. Interface Reference Manual, Varian Data Machines, California (May 1969).

44. Priority Interrupt Module, Varian Data Machines, California (November 1969).

45. Calingaert, P., "System Performance Evaluation: Survey and Appraisal", Comm. ACM, Vol. 10, No. 1 (January 1967), pp. 12-18.

46. Lucas, H.C., "Performance Evaluation and Monitoring", Computing Surveys, Vol. 3, No. 3 (September 1971), pp. 79-91.

47. Drummond, M.E.J., "Evaluation and Measurement Techniques for Digital Computer Systems", Prentice-Hall Inc., N.J. (1973).

48. Cantrell, H.N. and Ellison, A.L., "Multiprogramming System Performance Measurement and Analysis", AFIPS Conference Proceedings, Vol. 32, SJCC (1968),pp. 212-221.

49. Chiu, W., DuMont, D. and Wood, R., "Performance Analysis of a Multiprogrammed Computer System", IBM J. R & D, Vol. 19, No. 3 (May 1975), pp. 263-271.

50. Varian 620 I Computer Handbook, Varian Data Machines, California (1969).

51. Strachey, C., "Time-Sharing in Large Fast Computers", Proceedings, International Conference on Information Processing , UNESCO, Paris (June 1959), pp. 336-341.

52. Anderson, H.A. and Sargent, R.G., "Investigation Into Scheduling for an Interactive Computing System", IBM J. R & D, Vol. 18, No. 2 (March 1974), pp. 125-137.

53. Madnick, S.E. and Donovan, J.J., "Operating Systems", McGraw-Hill, U.S.A. (1974).

54. BASIC Language Reference Manual, Varian Data Machines, California (1970).

55. Operation and Maintenance Manual, VDM 620/I - CDS 111 Disc-Drive Interface, Telefile Computer Products, U.S.A. (September 1970).

56. Theis, D.J. and Hobbs, L.C., "Minicomputers for Real-Time Applications", Datamation, Vol. 15, No. 3 (March 1969), pp. 39-53.

57. Smith, C.L., "Digital Control of Industrial Processes", Computing Surveys, Vol. 2, No. 3 (September 1970), pp. 211-241.

58. Real-Time Operating System - Reference Manual, Data General Corporation, Mass. (1974).

59. Operating System OS/32 MT Manual, Interdata Corporation, Oceanport, New Jersey (1976).

60. Fotheringham, J., "Dynamic Storage Allocation in the Atlas Computer, Including an Automatic Use of a Backing Store", Comm. ACM, Vol. 4, No. 10 (October 1961), pp. 435-436.

61. Denning, P.J., "Virtual Memory", Computing Surveys, Vol. 2, No. 3 (September 1970), pp. 153-180.

62. Gloriosa, R.M. and Chase, T.D., "Design of Virtual Memory for Small Computers", Computer Design, Vol. 12, No. 12 (December 1973), pp. 67-72.

63. Hawinkels, J.H.W. and Rodd, M.G., "PDP 11/20 Virtual Memory", Thesis, University of Cape Town, (1974).

64. PDP 11/20 Processor Handbook, Digital Equipment Corporation, Maynard, Mass. (1972).

65. Hendrie, G.C. and Sonnenfeldt, R.W., "Computer Reliability", ISA Journal, Vol. 10, No. 1 (January 1963), pp. 51-56.

66. Fraade, D. and Copeland, J.R., "An A.B.C. of Buying a Minicomputer", Instrument Practice (March 1971), pp. 155-159.

67. Hix, A.H., "Status of Process-Control Computers in the Chemical Industry", Proc. IEEE, Vol. 58, No. 1 (January 1970), pp. 4-10.

68. Stewart, A.B., Private Communication, National Institute for Metallurgy, Johannesburg (1976).

69. Lucin, W., "Can a User Test LSI Microprocessors Effectively ?", IEEE Semiconductor Test Symposium (1975).

70. Popek, G.J., "Protection Structures", Computer, Vol. 7, No. 6 (June 1974), pp. 22-31.

71. Intel Series 3000, "CROMIS-Cross Microprogramming System", Reference Manual, Intel Corporation, Santa Clara, Calif. (1975).

72. Ribeiro, S., "Talking to the Minicomputer", IEEE Trans. Industrial Elect. & Control Instr., Vol. 18, No. 2 (May 1971), pp. 67-72.

73. Anon., "ECL Bit-Slice Processor Arrives, Setting Top Speed for Bipolar Microprocessors", Electronics Design, Vol. 24, No. 10 (May 1976), pp. 53-55.

74. Husson, S.S., "Microprogramming Principles and Practice", Prentice-Hall, N.J. (1970).

75. Wilkes, M.V., "The Growth of Interest in Microprogramming: A Literature Survey", Computing Surveys, Vol. 1, No. 3 (September 1969), pp. 139-145.

76. Barr, R.G., et al., "A Research-Oriented Dynamic Microprocessor", IEEE Trans. Computers, Vol. 22, No. 11 (November 1973), pp. 976-985.

77. Varian 620L Computer Handbook, Varian Data Machines, Calif. (1972).

78. Graphic Symbols: Logic, STD 095, National Institute for Defence Research, C.S.I.R., Pretoria, R.S.A. (1970). (Based on Recommendations IEC 117).

79. Intel Data Catalogue, Intel Corporation, Santa Clara, Calif., (1975).

80. Mills, T.E. "Schottky TTL vs. ECL for High Speed Logic", Computer Design, Vol. 11, No. 10 (October 1972), pp. 79-86.

81. Torrero, E.A., "Bipolar Bit-Slice Microprocessors Shrink the Size and Cost of Minis and Controllers", Electronics Design, Vol. 24, No. 10 (May 1976), pp. 34-40.

82. Anon., "SBP0400 4-Bit Parallel Binary Processor Elements", Texas Instruments Incorporated, Dallas, Texas (1976).

83. Reyling, G. (Jnr), "Considerations in Choosing a Microprogrammable Bit-Slice Architecture", Computer, Vol. 7, No. 7 (July 1974), pp. 26-29.

84. Scarlett, J.A., "Transistor-Transistor Logic and its Interconnections", Van Nostrand Reinhold, Marconi Series (1972), pp. 217-227.

85. Statland, N., "Methods in Evaluating Computer Systems Performance", Computers and Automation, Vol. 13, No. 2 (February 1964), pp. 18-23.

86. Stimler, S. and Brons, K.A., "A Methodology for Calculating and Optimizing Real-Time System Performance", Comm. ACM, Vol. 11, No. 7 (July 1968), pp. 509-516.

87. Bucholz, W., "A Synthetic Job for Measuring System Performance", IBM Systems J., Vol. 8, No. 4 (April 1969), pp. 309-318.

88. Nutt, G.J., "Tutorial: Computer System Monitors", Computer, Vol. 8, No. 11 (November 1975), pp. 51-61.

89. Svobodova, L., "Computer System Measurability", Computer, Vol. 9, No. 6 (June 1976), pp. 9-17.

90. Arndt, F.R. and Oliver, Capt. G.M., "Hardware Monitoring of Real-Time Computer System Performance", Computer, Vol. 15, No. 4 (July/August 1972), pp. 25-29.

91. Coffman, E.G. and Kleinrock, L., "Computer Scheduling Methods and Their Countermeasures", AFIPS Conference Proceedings, Vol. 32, SJCC (1968), pp. 11-21.

92. Liu, C.L. and Layland, J.W., "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", J. of ACM, Vol. 20, No. 1 (January 1973), pp. 46-61.

93. Martin, J., "Programming Real-Time Computer Systems", Prentice-Hall Inc., N.J. (1965).

94. Holland, F.C. and Merikallio, R.A., "Simulation of a Multiprocessing System Using GPSS", IEEE Trans. on System Science and Cybernetics, Vol. 4, No. 4 (April 1968), pp. 395-400.

95. Coffman, E.G. (Jnr.), et al., "System Deadlocks", Computing Surveys, Vol. 3, No. 1 (June 1971), pp. 67-78.

96. Vortex II Reference Manual, Varian Data Machines, California (1975).

97. "BEST" Overview, Varian Data Machines, California (1973).

98. Shearing, B., "The History and Philosophy of Time Sharing and Multiprogramming - Operating Systems MP/1, MP/2 and MP/3", Computer Caravan, 25-27th September, 1973, U.S.A.

99. Master Operating System (MOS) Reference Manual, Software Handbook, Varian Data Machines, California (1971).

100. Weavind, T.E.F., "The Siemens 320K - Is Minicomputer the Term for a Blockbuster?", Electronics and Instrumentation, Vol. 6, No. 10 (October 1975), pp. 54-58.

101. Process Computers, Siemens 300/76-Bit Systems, Catalogue PR2, Siemens Aktiengesellschaft, W. Germany (1974).

102. Weavind, T.E.F., "Dual Processor Computer", Electronics and Instrumentation, Vol. 7, No. 1 (January 1976), pp. 45-53.

103. Hodges, J.L. & Lehmann, E.L., "Basic Concepts of Probability & Statistics", Holden-Day, U.S.A.(1970).

104. Rubin, M. & Haller,C.E., "Communication Switching Systems", Reinhold Publishing Corp. N.Y., (1966), pp.246-271.

105. Lundkvist, K., "Analysis of General Theory for Telephone Traffic", Ericsson Technics, Vol.11, No.1 (1955), pp.3-32.

106. Cox, D.R. & Smith, W.L., "Queueing Theory", John Wiley & Sons, (1961).

107. von Neumann, J., Barks, A.W. and Goldstine, H.H., "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument", Datamation, Vol. 8, No. 9 (September 1962), pp. 24-31, and Vol. 8, No. 10 (October 1962), pp. 36-41. Originally published by the Institute of Advanced Study, Princeton, New Jersey (1947).

108. Richards, R.K., "Arithmetic Operations in Digital Computers", Van Nostrand, New Jersey (1955).

109. Rodd, M.G. & Potgieter, J.H., "A Microprocessor-Based Weighing and Feed Control System", Proceedings, IFAC 2nd. Symposium on Automation in Mining, Mineral and Metal Processing, Johannesburg, South Africa (1976).

110. Rodd, M.G. & Bloch, G., "A Micro-Processor Based Industrial Control System", Proceedings, SACAC Symposium on Minicomputers and Microprocessors, Durban, South Africa (1976).

111. Syski, R., "Congestion Theory in Telephone Systems", Oliver & Boyd, London (1960), pp 298- 340.

INDEX TO THE APPENDICES

.....
"It next will be right
To describe each particular batch:
Distinguishing those that have feathers, and bite,
From those that have whiskers, and scratch."

Lewis Carroll, *The Hunting of the Snark (Fit the Second)*

Page No.

APPENDIX A : MICROPROGRAMMING : AN INTRODUCTION AND SOME DEFINITIONS

A.1 Philosophy of Microprogrammed Control of Computers

The organization of a conventional computer is illustrated schematically in Fig. A-1(a). Essentially, four major sections may be identified: the memory, the input/output facilities, the arithmetic and logic unit, and the managing director of the system, the control unit. It is precisely the control unit which makes a computer worthy of the name. It provides for overall control of the various sections of the computer, defines their operations, and regulates data flow between them. The total effective operation of the system is defined by the current machine-code instruction which is being dealt with. Each machine-code instruction is fetched from the memory and decoded by the control unit, and the operations defined by the contents of the instruction are implemented.

Following the original concepts presented by Wilkes [6] in the early 1950's, the organisation of a microprogrammed computer structure is illustrated in Fig. A-1(b). The essential difference between these two structures lies in the mode of operation of the control unit. In the microprogrammed structure, a microinstruction memory (or control store) contains sets of primitive operation codes. These sets of codes are termed microinstructions. Each component part of a microinstruction speficies an elementary logical or arithmetic process to be effected in the computer. In order to execute the machine-code instructions mentioned in the first paragraph, a series of micro-instructions must be written into the microinstruction memory. The machine-code instruction will then be used to "point" to the start of the appropriate sequence. Fig. A-1(c) shows a typical system for implementing such a concept. The machine-code instruction will set up a start address in the microinstruction memory address register. This address specifies the first microinstruction in the required sequence. The microinstruction is accessed from the memory and placed in the micro-instruction register. The resulting control signals are sent to the rest of the computer to effect the specified micro-operations. The information necessary for the determination of the next address is contained within the current microinstruction. The resulting next

address is sent to the address register. If the microinstruction is the last to be executed in the current sequence, a control signal will be generated to fetch the next machine-code instruction, to define the next microinstruction sequence to be implemented.

Typically, a modern microinstruction is usually more complex than a machine-code instruction. It will, therefore, require the programmer to have a considerable, detailed knowledge of the hardware of a particular system in order to undertake the actual task of microprogramming. (For example, the IBM 360/40 has a 56-bit microinstruction with 19 distinct fields, most of which are used to define independent hardware operations. The IBM 370/158 has a 72-bit microinstruction with 22 fields!)

Further information on microprogramming may be found in Husson's now-famous book, "Microprogramming Principles" {74}. An excellent review of literature on the subject is found in Ref. {75}.

A.2 Definition of Terms Relevant to Microprogramming

As with most new technologies, the subject of microprogramming and microprogrammable computers comes with a multitude of varying terminologies and definitions. In the current text, the following definitions are adhered to, being similar to those proposed in Ref. {76}.

(i) Assemblerinstruction: a mnemonic instruction that is mapped into one or more machine instructions by means of an assembler program.

(ii) Machine-Code Instruction: a bit-pattern that is interpreted by the executing control unit hardware in a hardwired (conventional) computer.

(iii) Microinstruction: a bit-pattern, normally stored in a micro-instruction memory, which controls the computer hardware at a primitive level. It will define the micro-operations to be effected during one microcycle.

(iv) Microroutine: a sequence of microinstructions.

(v) Micro-operation: a primitive microinstruction action, such as addition or transfer.

(vi) Microprocedure: a sequence of micro-operations, which constitutes a functional whole.

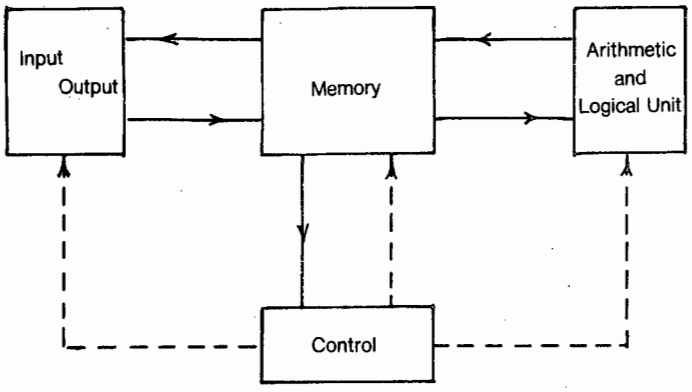
(vii) Microinstruction Memory (or Control Memory, or Control Store): the memory in which the microinstructions are stored.

(viii) Local Memory: a high-speed memory, normally with limited capacity, located within the processing unit and operating synchronously with it.

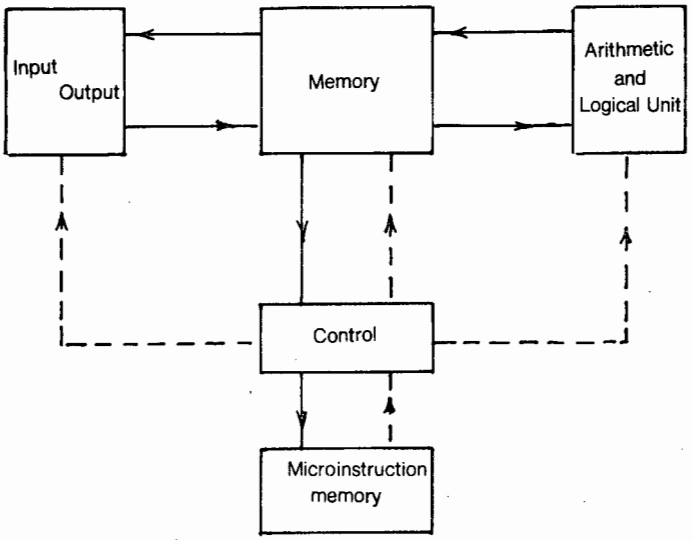
(ix) Microcycle: a cycle of control which performs the fetching and execution of one microinstruction and, normally, the generation of the next microinstruction to be dealt with.

(x) Microinstruction Sequencing: the determination of the chronological order in which the microinstructions are to be executed.

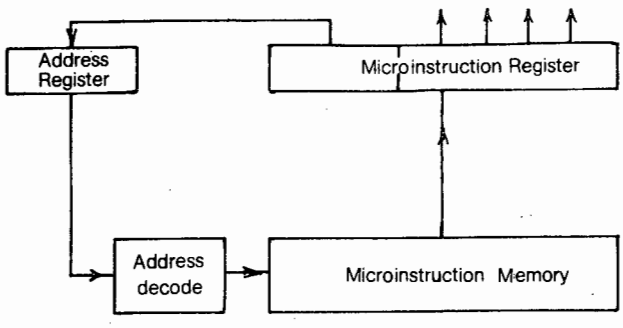
(xi) Microinstruction Sequencer Unit: the logic which performs the actual microinstruction sequencing.



(a) ORGANIZATION OF A CONVENTIONAL COMPUTER



(b) ORGANIZATION OF A MICRO PROGRAMMED COMPUTER



(c) MICROPROGRAM CONTROL STRUCTURE

FIGURE A-1 : COMPARISON BETWEEN A CONVENTIONAL COMPUTER AND A MICROPROGRAMMED COMPUTER

APPENDIX B : DISCUSSION OF THE MINICOMPUTERS EMPLOYED

The minicomputers which were used in the work described in this thesis were both Varian computers, produced by Varian Data Machines of America. For the first and third experimental situations discussed in the text, a Varian 620L-100 machine, with 8 Kilowords of core memory and a minimum of peripherals, was used. For the second experiment, the computer used was a Varian 620I with 20 Kilowords of core memory and a full complement of peripherals. The change in computers was the result of general system availability demands. In both computers, however, the input/output structures are virtually identical, the 620L-100 having a fractionally faster cycle-time, which necessitated a slight modification to the timing algorithms outlined in Section 3.3.7 of the text.

Full details of these computers may be found in references {50} and {77}. Of immediate importance to the completeness of this text is a discussion of the input/output facilities available {43}.

B.1 Input/Output Structure of the Varian 620 Computer Range

The Varian 620 minicomputer has a relatively primitive input/output structure. For both input and output of data, corresponding sets of machine-code instructions are available. These instructions specify the address of the peripheral within the lower 6 bits of the 16-bit word. The source, or destination, of data within the computer will be one of the two accumulator registers, denoted the "A" and "B" registers. A memory cell may also be specified as the source or destination of data, although this requires the use of double-word instructions, the memory address being indicated by the contents of the second word. The 16 input/output lines, referred to as the External Bus, or E-Bus, lines (EB00-EB15), are time-shared between address and data. Two signals, FRYX (Function-Ready) and DRYX (Data-Ready), control the time-sharing. When the FRYX signal is active, the information on the E-Bus lines 0-5 forms an address which must be decoded to determine which peripheral interface is specified. At the same time, other E-Bus lines, in particular EB11-EB15, are used to indicate the type of instruction being executed by the computer (e.g. if the instruction is to effect an output of data, EB13 will be active). These controls permit the total identification of the procedure to be followed by the interface. When DRYX is active, the data transfer may be effected.

Two other types of input/output instruction are available. The first is the EXC, or External Control instruction. This instruction is used to produce a single control signal to a peripheral device, and has a greater address capability than data input/output instructions. In addition to the basic six address bits, the next three bits (bits 6 - 8 in the instruction word) may be used to form a sub-address. These nine bits are placed on the E-Bus when FRYX is active and may subsequently be decoded. This means that within a single basic address eight different signals may be derived. In fact, this may be extended to sixteen, as two different EXC instructions are available, the basic EXC and a second, termed EXC2. The choice between the two is indicated during the FRYX active time, by either EB11 or EB15 respectively becoming active. There is no data transfer during the execution of an EXC command, so DRYX does not become active.

The second additional instruction is the double-word "sense" instruction. When the peripheral device being "sensed" becomes available, the contents of the second word are used as an address at which the execution must continue. If the peripheral is not ready, the second word is ignored. The instruction is similar in operation to the EXC instruction, except that EB12 becomes active to indicate the type of instruction being executed. The decoded address and sub-address are logically-gated with a "ready" line, derived from the peripheral device. The state resulting from this gating is routed back to the processor by means of a control line termed "SERX", or "sense response". The processor tests the status of this line during FRYX active time and, if it is logically true, causes the program counter to be set to the address held in the second word of the double-word instruction. If SERX is not true, the second word is ignored, and the program execution continued from the next word.

A typical peripheral-accessing cycle might therefore consist of the following sequence of instructions:

(i) Send an EXC instruction to activate the peripheral.

(ii) Repeatedly "sense" the status of the peripheral until it becomes "ready".

(iii) When "ready", input or output the data.

B.2 Interrupt Facilities

Appropriate control lines are available in the input/output structure to effect interrupts to the processor. Essentially, the peripheral will inform the processor of its intention to interrupt the current program and will place the desired interrupt address onto the E-Bus lines. The processor will accept this information and trap to that address. Reference { 43 } gives full details of the procedure.

An optional interface produced by the manufacturer of Varian Computers obviates the necessity for the user to design his own interrupt control logic. This interface is termed a "Priority Interrupt Module", or PIM {44 }. This unit makes available to the user eight interrupt control lines which, when activated by incoming signals, will implement interrupts to predetermined addresses. The PIM is priority-structured, and may be activated or de-activated under program control. Any interrupt channel(s) may be masked out. This interface was used within the experimental systems described, to reduce the amount of hardware that had to be custom-designed.

B.3 Direct Memory Access

The Varian minicomputers are capable of implementing direct memory access functions, i.e. the transfer of data between peripheral devices and memory, without program control. These are based on a cycle-stealing principle, where a single DMA transfer may occur after the execution of an instruction, the basic cycle-time of the computer being extended to permit this transfer. Direct memory access techniques could valuably have been used to effect the transfer of data to and from the microcontroller. It was decided, though, that the extra hardware required in the microcontroller input/output interface to implement this facility would not be warranted in an experimental situation. The disc system included in the experimental situation mentioned in the text was, however, interfaced to the computer with DMA facilities so as to allow for the high rate of transfer required.

APPENDIX C: THE MICROCONTROLLER CIRCUITRY.

In this appendix, the design details of the microcontroller are discussed. The logical descriptions should be read in conjunction with the schematic drawings given. It should be noted that, in order to make the drawings relatively simple to understand, all detailed information has been removed. The logic conventions adopted are in close concordance with the logic standards defined by Ref {78 }. Full working drawings are available from the author. In order to produce a self-contained text, a description of the large-scale-integration elements used (the Intel 3000 series microcomputer components) is given in Section C.8.

C.1. The Microinstruction Sequencer and the Central Processor Array: Drawing No.1.

Although the microinstruction sequencer and the central processor array have previously been described as separate entities, they are discussed together so as to give a better overview of the total system operation. The heart of the microinstruction sequencer is the Intel 3001 Microcontrol Unit (MCU). The simplest way of describing the logical oper--ation of this unit is to consider a complete cycle of activity. The initialization of the MCU may be implemented by the use of the secondary (SX) and primary (PX) buses, and the "load start" input. When a signal is applied to this input, the information present on the SX and PX buses is gated directly into the next-address register of the MCU. When the outputs of this register are enabled, this information will be the initial address sent to the microinstruction memory. In the circuits shown, the initial address is entered via the console and appropriate buffer elements. The "load start" signal is obtained directly from the console.

On receipt of this address, the micromemory makes available the corresponding microinstruction. Bits 7 to 13 of this microinstruction comprise the next-address control specifications, and are supplied directly to the MCU. The rest of the microinstruction bits are stored in the pipeline register. Using this next-address control information, together with "carry-in" status (derived from the central processor array), the MCU will form the next micromemory address.

The interpretation of the "carry-in" flag data is controlled by bits 15 to 18. These bits will also control the status of the "carry-out" flag. Table 1 shows the possible operations which may be implemented.

Interrupt vectoring is handled by the sequencer unit. When a "jump to row 0, column 15" instruction is executed by the MCU, the Interrupt Strobe Enable (ISE) output line is set active. The interrupt control unit (See C.4) will respond to this signal if an interrupt has been raised, by sending a signal to the "Enable Row" (ERA) input to the MCU. This disables the next-address row outputs, while leaving the next-address column outputs unaffected. The interrupt control unit will then place the appropriate interrupt vector address on to the row outputs, and hence cause a new address to be sent to the micromemory.

The 16-bit central processor array is composed of 8 Intel 3002 Central Processor Elements, each of which is a 2-bit-wide arithmetic and logical unit. The control over the function which the CPA is to perform is derived from bits 0 to 6 of the microinstruction, via the pipeline register. An Intel 3003 Look-Ahead-Carry Element is used to perform a conventional look-ahead arithmetic function over the 8 central processor elements. The CPE's have available standard look-ahead-carry outputs, denoted X and Y. The "carry-in" and "carry-out" functions are thus derived from the look-ahead-carry element, in conjunction with the "carry-in" and "carry-out" flag bits from the sequencer unit. These bits form the essential communication functions between the sequencer unit and the central processor array. The K-bus control inputs to the CPE's are linked, and their status is set by the contents of bit 14 in the microinstruction.

The various input and output buses on the CPE's are used as follows:

(i) The A-Bus: This makes available the contents of the address registers in the CPE's, and is used to supply addresses to the local memory and to the input/output control unit.

(ii) The D-Bus: This makes available the current value of the accumulator of the CPE's, and forms the sole data output path for the system.

(iii) The M-Bus: Data output from the local memory is supplied to the CPE via this bus. Also, to inject test data into the CPE's, the binary value specified by the data input switches on the console may be placed on to this bus via a set of tri-state latches.

(iv) The I-Bus: Data output from the input/output interface is sent to the CPE's via this bus.

CPA monitoring facilities are provided by latches located on the output lines from the A- and D-buses.

The basic clock signal to the central processor array is not the master system clock, but a special one, denoted the CPA clock, which may be inhibited by a suitable combination of bits 19 to 21, which will allow for "conditional clocking" operations to occur (as discussed in Section C.8.2.)

Note that the A- and D-buses are buffered, in view of the subsequent loading to be placed on these output lines, and may be disabled by means of the EA (Enable Address), or ED (Enable Data) inputs.

C.2 The Microinstruction Memory: Drawing No. 2.

To provide for the possibility of dynamic microprogramming, and to ease initial microprogram development, the microinstruction memory (or micromemory) is composed of 2102-A N-MOS Random Access Memory elements. This yields a maximum read-cycle-time of approximately 300 nanoseconds. These memory elements are arranged as 1024 x 1-bit arrays, so 24 of them will provide 2 banks of 512-word memory. In the normal microcontroller use the memory will, of course, only be operated in the read-mode, and thus the rather lengthy read-write cycle of 2102-A's is not relevant. {79} In the prototype system used, the data to be written into the micromemory is supplied by the input switches on the console, and the read-write control line to the system is set by the console interface (See Section C.7).

To ensure that the memory is non-volatile, its power supply is separated from the microcontroller power supplies. When the microcontroller power fails, a secondary back-up battery supply is brought into operation to retain the contents of the memory.

C.3 The Local Memory: Drawing No. 3.

This comprises 16 x 3106 bipolar Random Access Memory elements, to provide 256 words of storage - the 3106 RAM having a 256 x 1-bit structure. {79} Unlike the micromemory, the local memory must be truly read-write, with fast enough access-times to ensure that the microcontroller's performance is not degraded. Address data and memory data are provided via buffer elements from the central processor array. The memory output data is buffered before being sent, via the M-Bus, to the CPA.

Control of the local memory is effected by decoding the control signals produced by the input/output control unit. (See Section C. 6) The local memory is normally held in the read mode. On the request for a read-write cycle, two monostables are used to provide a Memory Acknow-

ledge (MEM ACK) signal back to the I/O control unit. The first monostable is triggered as soon as the request occurs, and is timed so as to allow the request to be honoured by the memory. At the end of this time, the second monostable produces a suitable MEM ACK signal.

Power-fail protection (as previously discussed in the case of the micromemory) must also be provided.

C.4 The Interrupt Control Unit: Drawing No. 4.

Central to the design of the interrupt control unit are the Intel 3214 ICU elements. Each element provides for 8 interrupt input lines. The elements may be multiplexed to provide for a greater number of interrupt lines. The current level of interrupt is held within the ICU's by a set of current status latches. Any incoming interrupt is encoded, and its level is compared with the level currently specified. To simplify the explanation of the operation of this unit, a full cycle of events is described.

Initially, the current level of interrupts is sent to the ICU elements. This is controlled by a "write" signal which is derived for convenience on the input/output interface (See C.5). The "write" signal is applied to the ICU elements via the Enable Current Status (ECS) input line. The actual status is specified by the contents of the D-Bus lines coming from the central processor array.

Any subsequent incoming interrupt signal is encoded into binary via an 8-to-3-line encoder. If it has a higher status than the current status of the ICU, an Interrupt Acknowledge (IA) signal will be produced when an Interrupt Strobe Enable (ISE) indication is received. This IA signal is used, firstly, to disable the next address row outputs of the microinstruction sequencer, and secondly, to gate the level of the accepted interrupt on to the next address "row" lines. Once the interrupt has been effected, a new current status level should be sent to the ICU's.

In the design shown, 3 such ICU elements are used to provide 24 levels of interrupt. A fourth element is used to determine which of these three elements has been activated.

One major problem exists in the use of these elements: the clock signal applied from the master system clock is used to strobe the incom--ing interrupt signal into the ICU's. This implies that all interrupt signals must be of sufficient duration to ensure that they are detected. Also, if an interrupt cannot be serviced immediately, it must remain ac-

tive until it is, in fact, accepted. In other words, the ICU's are based on a "handshake" concept, whereby all interrupt signals must remain active until they are individually acknowledged. A combination of solutions was used in the microcontroller to achieve this. Firstly, all external interrupts are generated using the "handshake" concept. Secondly, interrupts from the minicomputer are held in latches preceding the inputs to the ICU, and the microprograms were constructed to ensure their acceptance. (The "handshake" concept cannot be used in the latter case as the method used to produce interrupts from the minicomputer does not permit detection of their acceptance.)

C.5 The Input/Output Interface: Drawing No. 5.

This section of the microcontroller is responsible for:

(i) Data transmission to and from the minicomputer.

(ii) The creation of interrupts to the minicomputer.

(iii) The writing of current status information into the interrupt control unit.

In order to follow the method of transmitting data to and from the minicomputer, Chapter III should be carefully reviewed.

The interfacing to the minicomputer, specifically the Varian 620I/L100 computer range, is fully covered in Ref. [43], and a review of the Varian input/output structure is given in Appendix B. The interfacing consists of the decoding of the I/O device address (EB00-EB05) and the strobing of this decoding at the FRYX (function-ready) time. If the instruction being executed by the Varian is an external control instruction, the line EB11 will be active and EB06-EB08 will indicate the sub-address. A three-to-eight-line encoder determines the correct sub-address, and this information is used to effect an interrupt to the appropriate interrupt control line. The other two operations which must be controlled by this interface are data input and data output. The decoded address is gated with the appropriate control signal (EB12 for data in and EB13 for data out), and results in the setting of either DTIX or DTOX. DTIX is present when (i) the correct address is generated, (ii) an input instruction is being executed and (iii) it is the correct time to strobe data into the minicomputer. DTOX is similar, and indicates that valid output data is available on the E-Bus lines. DTIX and DTOX are incorporated into the microcontroller control logic in the following way. If the microcontroller is executing a microinstruction specifying a data transfer to or from the minicomputer, the corresponding control signals will be generated by the input/output

control unit. As shown in drawing No. 5, these control signals are appropriately decoded. The reply signal, necessary for the microcontroller to complete the current microcycle, is Memory Acknowledge (MEM ACK). This signal is only issued once DTOX or DTIX occurs. The principle implies that the microcontroller will always wait until the minicomputer is ready for the transmission of data. This is an important feature in structuring the individual software routines to control such a transfer.

During the actual data transference, care must be taken to ensure that acceptance of data occurs only when the data is valid and in a stable condition. In the case of data being transferred to the minicomputer, the fact that the microcontroller is "held up" by the detection of microinstructions specifying the operation solves the problem (see Chapter III). When the microcontroller requests such a transfer, the data in question is held in the accumulator of the central processor array, and will remain there until the microcontroller execution recommences. When the minicomputer is ready to effect the actual transfer, the data will thus be in a stable state, and no additional buffer storage is required. With transmission from the minicomputer to the microcontroller, however, buffer storage is essential. The acceptance of data by the central processor array will only occur when the microcontroller recommences execution, after having been "held up" by the detection of an "input" microinstruction. So the data from the minicomputer must be read into a temporary buffer at the correct minicomputer data-strobe-time. The microcycle may then be continued, and the data held in the temporary buffer read into the central processor array.

There is one further aspect to be considered; the addressing structure of the microcontroller. The I/O control unit makes the distinction between transfers to the local memory, and those to the I/O interface. In the latter case the address held in the CPA address register must be decoded. Three sets of addresses are used. Firstly, any address in the range 0 - 7 is used for the selection of the interrupt level to be sent to the minicomputer's interrupt module. Address $17_{(8)}$ is the address related to all data transfers to and from the minicomputer, and address $16_{(8)}$ is assigned to the current status latch in the interrupt control unit. These addresses are directly derived from the CPA address bus by means of two three-to-eight-line decoders. If addresses 0 - 7 or 16 are detected, a suitably-timed Memory Acknowledge signal is immediately created to indicate to the I/O control unit that the microcontroller execution may continue. This is necessary since these two

"devices" will not require any extension of the microprocessor's cycle time.

C.6 The Input/Output Control Unit: Drawing No. 6.

The author wishes to credit much of the design of the structure of this unit to the authors of Reference {42}, and does not claim originality for the methods used in this section of the microcontroller.

This unit provides the basic control over all logical functions outside the central processor array. These functions include reading and writing to local memory and to the input/output interface, and the output of current status information to the interrupt control unit. The unit is also responsible for the control of the master system clock and of the central processor array clock.

The inputs to the I/O control unit are the data specified by microinstruction bits 19 - 21, and the master clock. The outputs from the unit are:

(i) The master system clock signal.

(ii) The central processor array clock signal.

(iii) Control lines to implement the input/output functions described above.

From the three microinstruction bits, eight different operations may be specified. These are:

000 - No I/O bus operation required.

001 - Inhibit CPA clock.

011 - No I/O bus operation; CPA may use the buses.

010 - Read - Modify - Write cycle.

110 - Memory read cycle.

111 - Memory write cycle.

100 - I/O device input operation.

101 - I/O device output operation.

In the last five operations specified, the unit produces the necessary control signals, and suspends the master system and CPA clock signals. These clock signals are re-enabled when a reply signal, denoted Memory Acknowledge (MEM ACK) is received from the devices specified.

The principal control signals produced by the unit are as follows:

(i) MRQ - Request memory or I/O cycle.

(ii) MEM - Determines either memory or I/O operation.

(iii) READ - Determines either read or write operation.

Other output lines are used to enable the data output or address output lines from the central processor array, according to the micro-

instructions being executed.

A further operation may be specified, which may be used to transfer data directly from the minicomputer to either the micro-instruction memory or the local memory. An active state on the External Request Input line will cause the inhibiting of the master clock signal, and the system buses may be used by the external device. Microprogram execution will recommence on receipt of a Memory Acknowledge signal.

C.7 The Console: Drawing No. 7.

As mentioned in the text, the console is intended to provide for initial testing of the system, and to aid subsequent microprogram development. Thus, in general the logic consists of suitable buffering of internal data buses so as to display the corresponding logic states. It also has two other important functions. It permits the "stepping-through" of microinstructions by the generation of a pseudo-master clock signal when the "step" switch is depressed. Full control over the microinstruction memory is also desirable. In order to examine or deposit a word in the micromemory, a 10-bit up-down counter is used to specify the address. The binary value of this counter is set by individual bit-switches and a "load address" switch. When the "console mode" is selected, the contents of the counter specify the corresponding micromemory address. The use of the up-down counter means that sequential locations may be examined. In order to write information into the micromemory, the desired bit-pattern is selected by the bit-switches. A memory write-cycle is effected by the "read-write" switch, which produces a suitable write signal to be sent to the memory.

It must be noted that the console logic was designed so that the removal of the unit would not have any effect on the overall operation of the microcontroller itself.

C.8 The 3000 Series Elements.

This section presents a brief discussion of the principal elements used in the design of the microcontroller. Full details are contained in Ref. {40}, while Ref. {32} contains a valuable introduction to the series of elements under consideration.

C.8.1 The 3001 Microprogram Control Unit.

Figure C - 1(a) shows the logical structure of this unit. In broad terms, it contains a microprogram address register, whose contents are made available to the micromemory via output buffers. The addresses which are held in this register are generated by the next-address logic. Provision is made for interaction between the unit and the rest of the system in which it is to be incorporated, by means of a flag input and a flag output. Typically, the flag output would be supplied as a carry-in signal to a central processing array, and the flag input would be derived from the carry-out signal of the array.

The next-address logic and the manipulation of the input and output flags are under the control of the address control function and the flag control function, which form part of the microinstruction. The next address is determined by the "instruction" supplied to the unit on the address control function lines. The resulting next address may be based on:

(i) The current address.

(ii) The status of one of the flag control latches.

(iii) The contents of the Primary (PR) latch.

Table 1 shows some of the possible address control functions, together with the format of the resulting addresses. (Note that this is an abbreviated list, as certain facilities were not required in the microcontroller).

The flag logic consists of three flip-flops, denoted the F-latch and the C- and Z-flags. The status of the flag inputs may be stored in one of these latches. The flag output may be set to the status of the C- or Z-flag, or may be set to a logic "0" or "1" condition. The actual operations to be carried out are specified by the status of the four flag logic control lines. See Table 1.

The primary and secondary instruction buses provide a direct means of setting the condition of the microprogram address register. The data contained on these input buses is sent directly to the microprogram address register when the "load" input line is activated and a clock signal is applied. This means of control may valuably be used to specify a micromemory address (which might be the start address of a series of microinstructions required to implement a macroinstruction). In the microcontroller developed in this study, the facility is used to initialise the system.

The secondary instruction bus contents may also be stored in the "PR" latch. These contents may be used for subsequent inclusion within the specification of a next address. The "PR" latch contents, under the control of a specified address control function, are also made avail--able to the system via an output buffer.

As discussed in C.1, the ISE (Interrupt Strobe Enable) output is active when a jump to row 0, column 15, function is executed. This is used to signify to the rest of the system that an interrupt sequence may be effected. The output buffers which supply the microprogram address to the micromemory may be disabled by means of the Output Enable (EN) input. The row address may similarly be disabled by the Enable Row Address (ERA) input.

Details of the precise timing relationships within the unit are described in Ref {40}. It is sufficient to note here that a cycle of operations is initiated by the rising edge of the incoming clock signal, and that the address determined during the last cycle is available (via the output buffer) within approximately 40 nanoseconds. Immediately after this, the address control outputs are read, and the next address is determined. The input clock cycle must have a minimum period of 85 nanoseconds.

C.8.2 The 3002 Central Processing Element.

Figure C - 1(b) shows the logical structure of this unit. It is essentially a complete, 2-bit-wide processing element, with the ability to input data from various sources; to store and process data; and to supply output data.

Thirteen registers are available for use. One of these is set aside as a memory address register, which holds the address for external memory or peripheral devices. Of the other twelve registers, one is demarcated as the accumulator, and is called the AC register. The contents of the accumulator are made available via an output buffer on to the Data Out Bus (D-bus). Both the memory address and the accumulator outputs may be disabled by activating specified input lines. The remaining registers, denoted T and R_0 to R_9, are available for general use. The T register is slightly different from the rest, as additional operations may be performed on it, and it may be regarded to a certain extent as forming a second accumulator register.

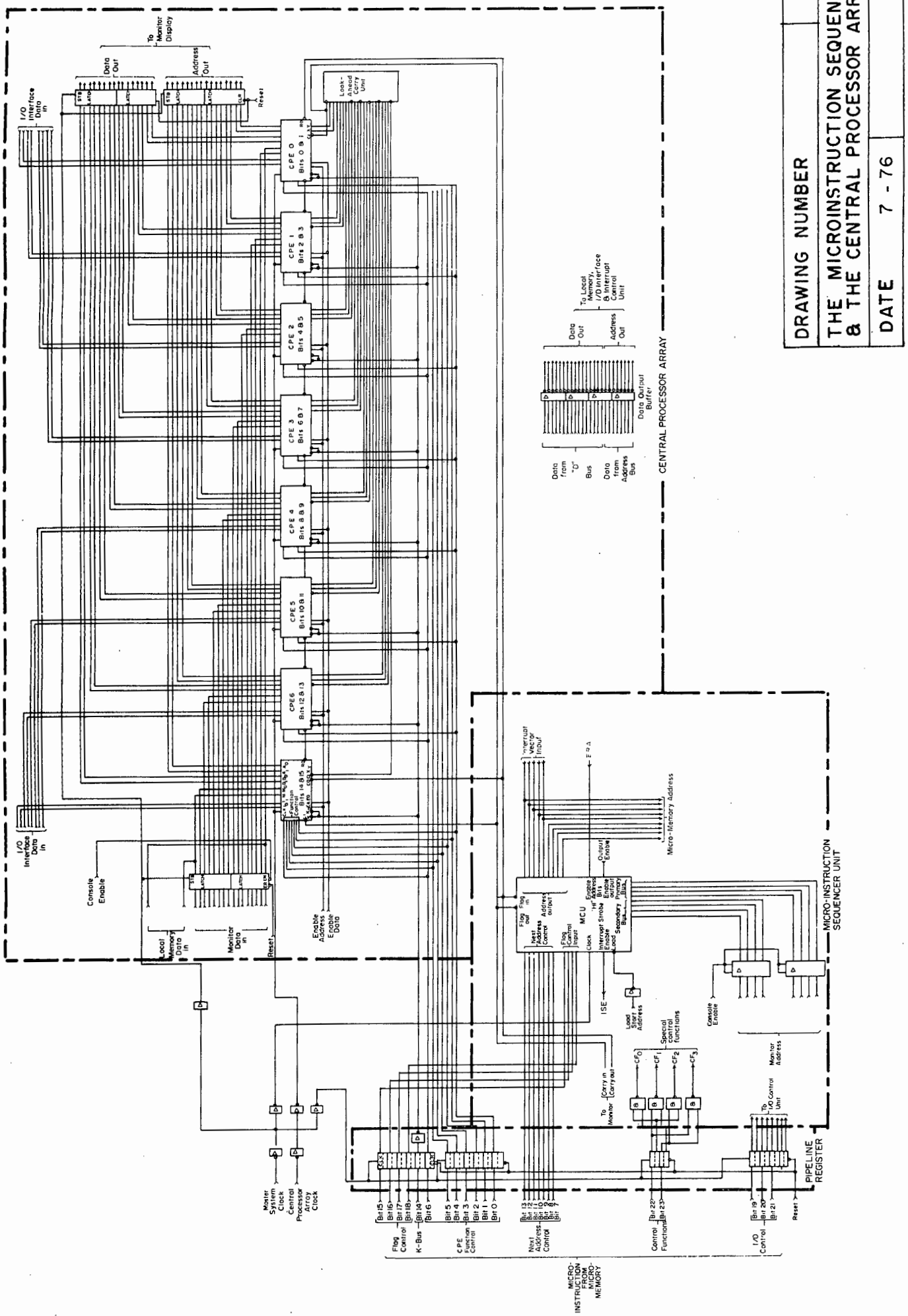
Data may be supplied to the unit via three separate input buses,

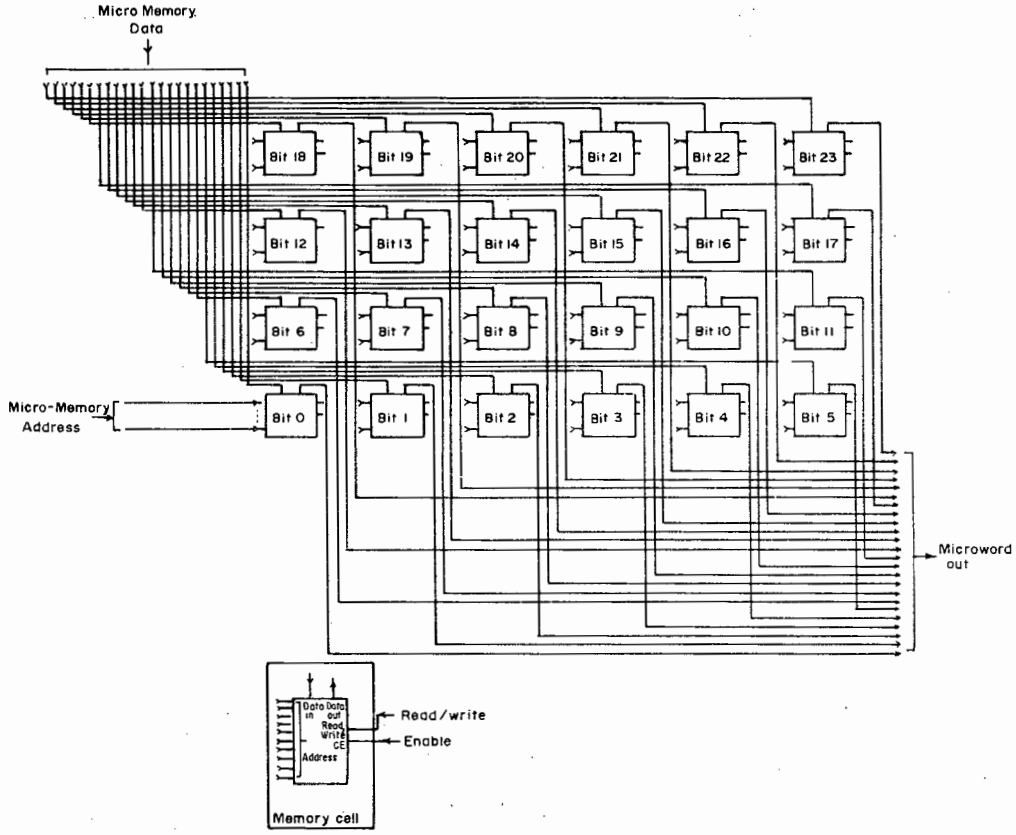
denoted the M-, I- and K-buses. Data on these buses is multiplexed (via two multiplexers) with the contents of the registers, or the output of the arithmetic/logic section. The multiplexing is, in fact, a logical gating between the specified sources. In practical terms, two buses are normally used for data sources, the M-bus for memory data and the I-bus for data from peripheral devices. The K-bus may be used to provide additional instructions, since the contents of this bus are logically-gated with the accumulator contents or the contents of the I-bus. The K-bus may also be used to supply the unit with constants, whose value may be held within microinstructions. Note that in the microcontroller, the two K-bus inputs were connected together, and could take the logic state either "0" or "1". The instruction set specified in Table 2 demonstrates this effect.

The input and output of data, as well as the multiplexing and specification of source and destination registers, are determined by the current status of the microfunction bus - i.e., the instruction to the unit. In addition, the instruction specified on the microfunction bus will determine the operation to be performed by the arithmetic/logic system (ALS). Essentially, the ALS is capable of performing logical shifting and simple arithmetic functions. In order to effect these operations, an additional set of input/output lines is provided. To make available the possibility of fast addition, standard look-ahead-carry outputs are provided, which may be supplied to a suitable look-ahead-carry-generator circuit. Arithmetic operations are implemented using the status of the Carry-In and Carry-Out lines. Shifting is effected by the provision of Left-In and Right-Out functions.

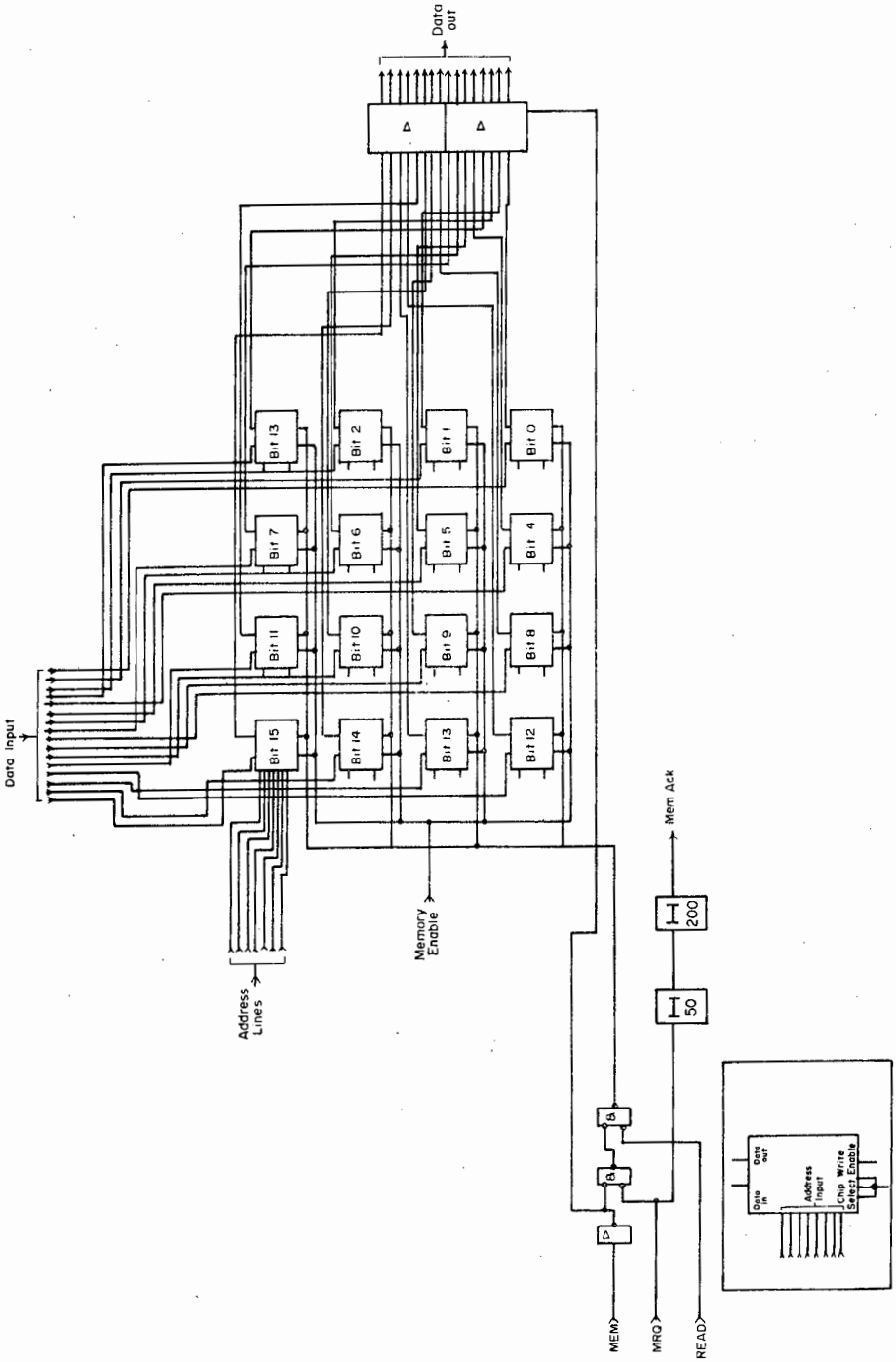
The "instructions" which are to be supplied to the unit via the microfunction bus are similar in format to the machine-code instructions in conventional computers. Of the seven bits in the instruction, the most significant three bits are used to specify the operation, and the least significant four bits indicate the source or destination register. In order to increase the possible number of functions that may be implemented, the register-specifying bits are split into three groups. The first group allows the selection of any of the registers, and the second and third groups permit only the selection of the T or AC registers. The selection of any one group, however, will select in turn a different effective operation in conjunction with the most significant three bits. Thus, with the possibility of control via the K-bus,a wide range of operations may be selected. An abbreviated instruction set is shown in Table 2.

The actual timing relationships within the unit are given in Ref {40}. Two aspects may be mentioned here. Firstly, during any operation which requires data to be stored in a register, the actual storing operation is carried out on the final negative-going edge of the applied clock signal. (The overall operation starts on the rising-edge of the clock signal). Thus, if this edge is witheld by means of external circuitry, the writing operation may be prevented. This gives rise to the idea of "conditional clocking" (See Section 3.3.4). Secondly, the overall minimum clock period is 120 nanoseconds, although it will be found in practice that a slightly shorter time is permissible.


DRAWING NUMBER 1
THE MICROINSTRUCTION SEQUENCER & THE CENTRAL PROCESSOR ARRAY
DATE 7 - 76
CENTRAL PROCESSOR ARRAY
MICRO-INSTRUCTION SEQUENCER UNIT
PIPELINE REGISTER
MICRO-INSTRUCTION MICRO-MEMORY
Data Out
Address Out
To Monitor Display
I/O Interface Data in
Reset
Look-Ahead Carry Unit
CPE 0 Bits 0 1
CPE 1 Bits 2 3
CPE 2 Bits 4 5
CPE 3 Bits 6 7
CPE 4 Bits 8 9
CPE 5 Bits 10 11
CPE 6 Bits 12 13
CPE 7 Bits 14 15
Function Control
Data Output Buffer
Data from "Q" Bus
Data from Address Bus
To Local Memory, I/O Interface & Interrupt Control Unit
Console Enable
Local Memory Data in
Monitor Data in
Enable Address
Enable Data
Interrupt vector input
Micro-Memory Address
Flag In
Flag Out
Next Address Control
Address output
Flag Control Input
MCU
Interrupt Strobe
Enable
Load Secondary Bus
Output Enable
Enable Write Address Bit
Enable Primary Bus
Secondary Bus
ISE
Load Start Address
Special control functions
CF0
CF1
CF2
CF3
Carry in
Carry out
To Monitor
To I/O Control Unit
Monitor Address
Master System Clock
Control Processor Array Clock
Bit 15
Bit 16
Bit 17
Bit 18
Flag Control
K-Bus
Bit 14
Bit 6
Bit 5
Bit 4
Bit 3
Bit 2
Bit 1
Bit 0
CPE Function Control
Bit 13
Bit 12
Bit 11
Bit 10
Bit 9
Bit 8
Bit 7
Next Address Control
Bit 22
Bit 23
Control Functions
Bit 19
Bit 20
Bit 21
I/O Control
Reset

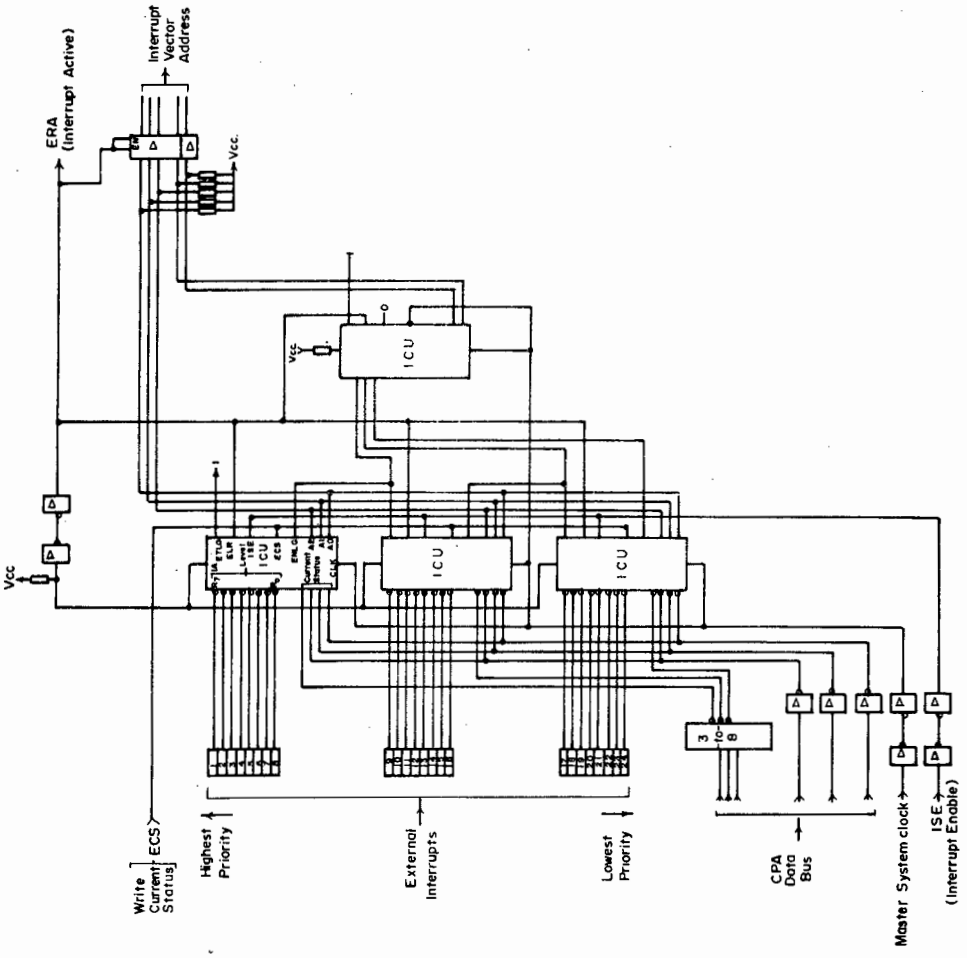


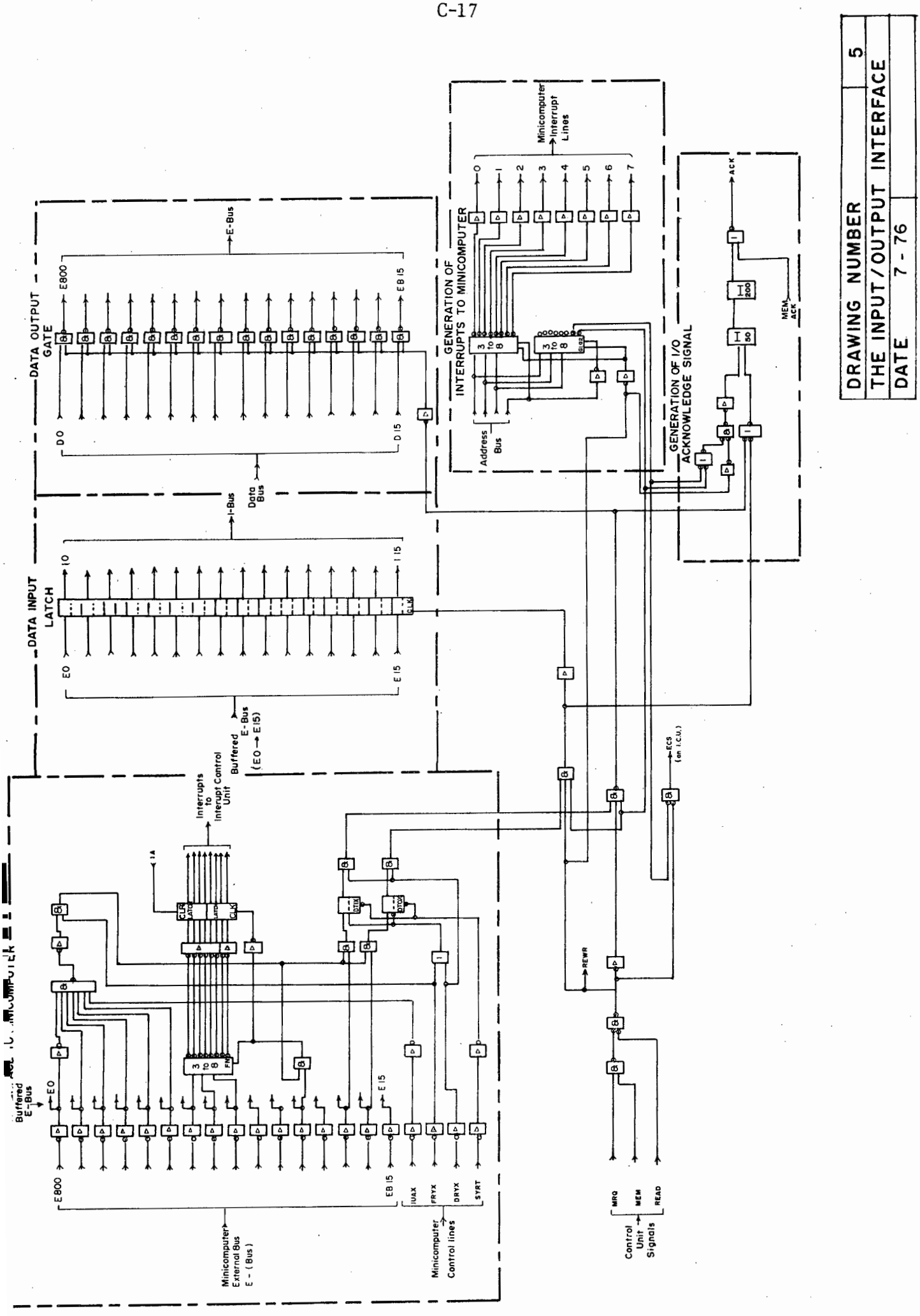
DRAWING NUMBER	2
THE MICROINSTRUCTION MEMORY	
DATE 7 - 76	

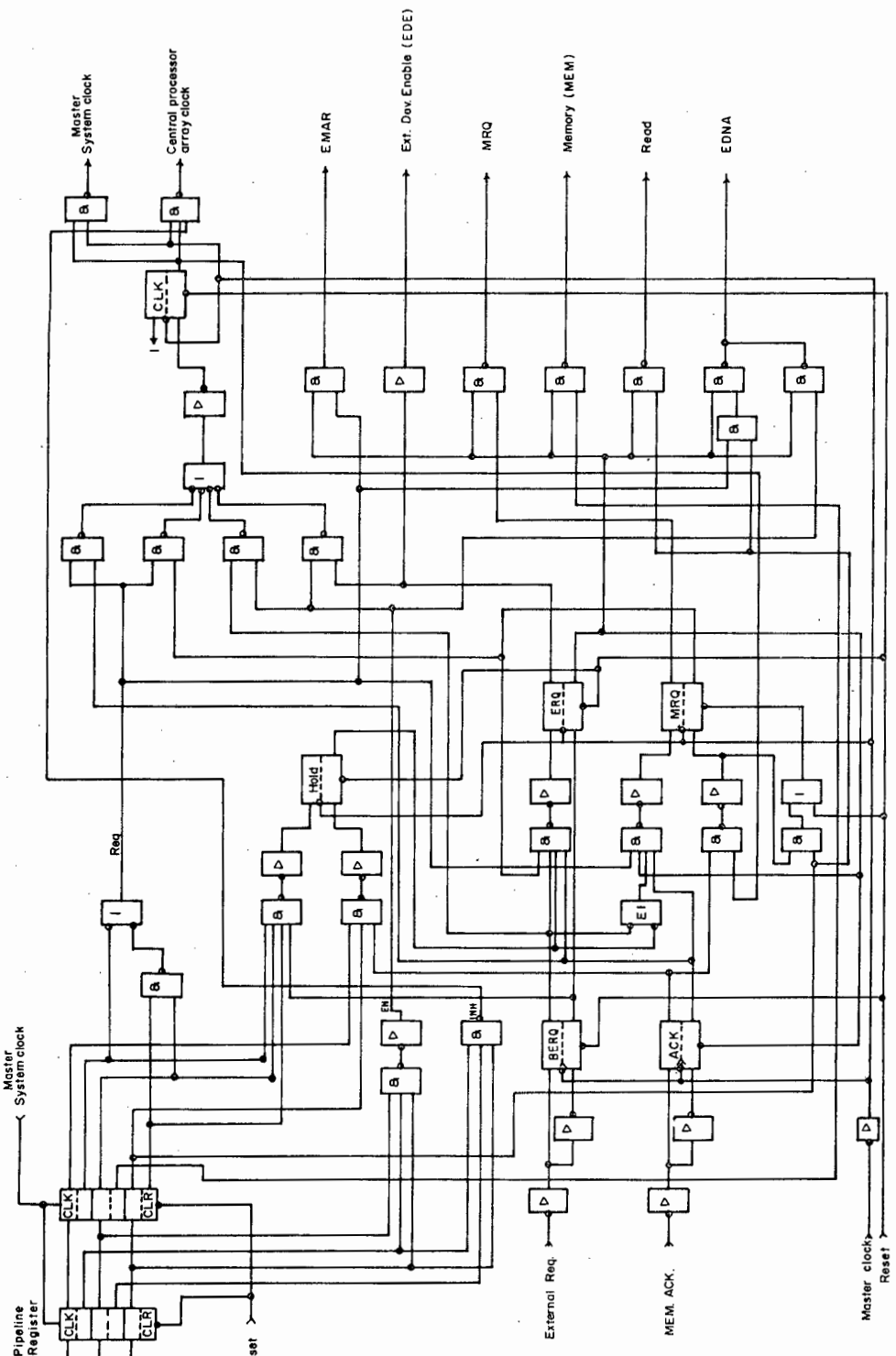


DRAWING NUMBER	3
THE LOCAL MEMORY	
DATE 7-76	

DRAWING NUMBER	4
THE INTERRUPT CONTROL UNIT	
DATE 7-76	

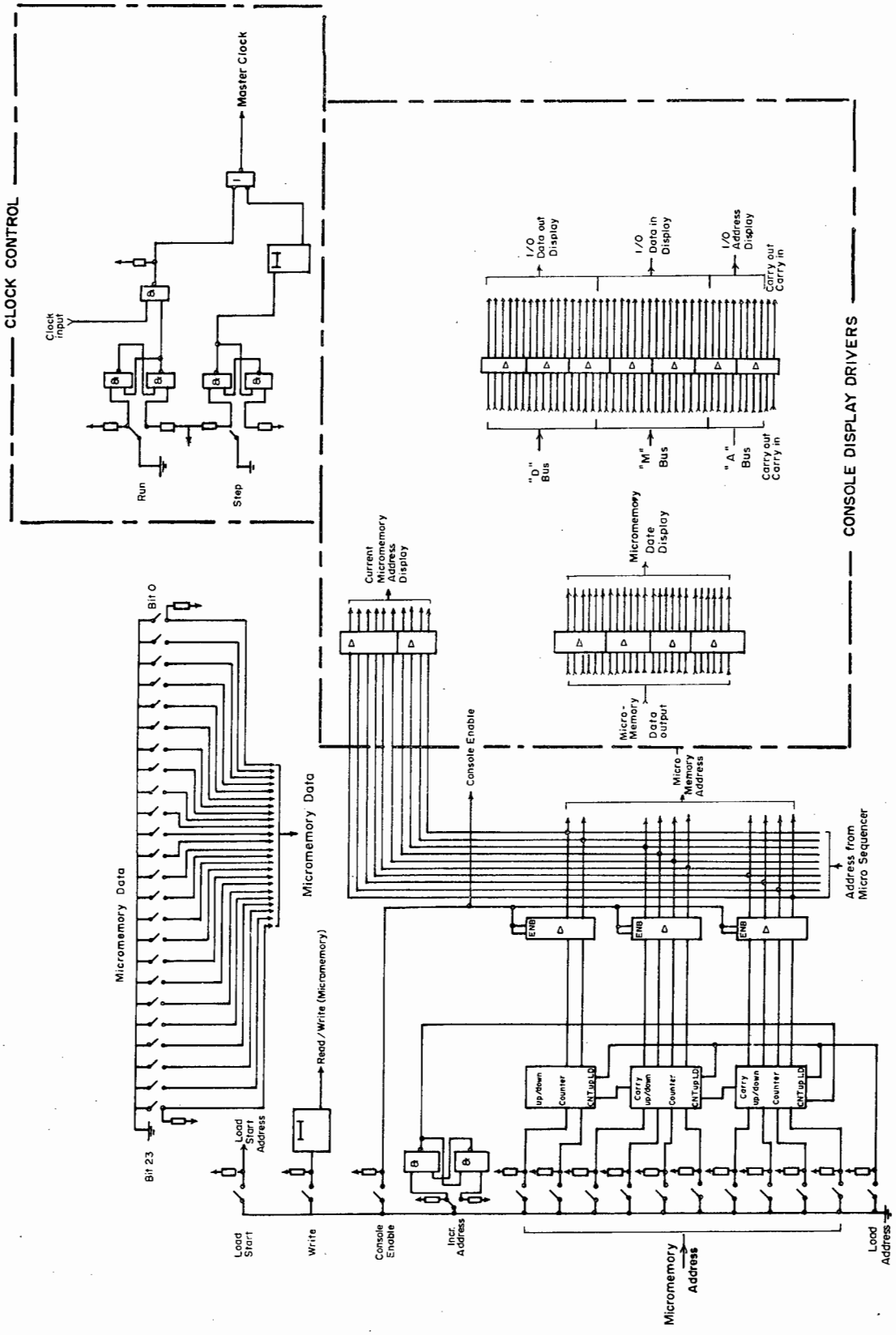



DRAWING NUMBER 5
THE INPUT/OUTPUT INTERFACE
DATE 7 - 76
DATA OUTPUT GATE
E-Bus
E800
EB 15
D0
D15
Data Bus
GENERATION OF INTERRUPTS TO MINICOMPUTER
Minicomputer Interrupt Lines
Address Bus
GENERATION OF I/O ACKNOWLEDGE SIGNAL
ACK
MEM ACK
DATA INPUT LATCH
I-Bus
I0
I15
E0
E15
Buffered E-Bus (E0 → E15)
Interrupts to Interupt Control Unit
ECS (on I.C.U.)
REWR
Buffered E-Bus
E0
E15
E800
EB 15
IUAX
FRYX
DRYX
SYRT
Minicomputer External Bus E - (Bus)
Minicomputer Control lines
MRQ
MEM
READ
Control Unit → Signals



FUNCTION	Code 21	20	19	EMAR	EDE	MRQ	MEM	READ	EDNA
No I/O Operation	0	0	0	1	1	1	0	1	1
Inhibit I/O Bus	0	0	1	1	1	1	0	0	1
CPA uses Bus	0	1	1	1	1	1	0	0	0
Read-Modify-Write	0	1	0	1	1	0	1	1	1
Read from Memory	1	1	0	0	1	0	1	1	1
Write to Memory	1	1	1	0	1	0	1	0	0
Read from I/O	1	0	0	0	1	0	0	1	1
Write to I/O	1	0	1	0	1	0	0	0	0

DRAWING NUMBER	6
THE INPUT/OUTPUT CONTROL UNIT	
DATE 7-76	



DRAWING NUMBER	7
THE CONSOLE	
DATE 7 - 76	

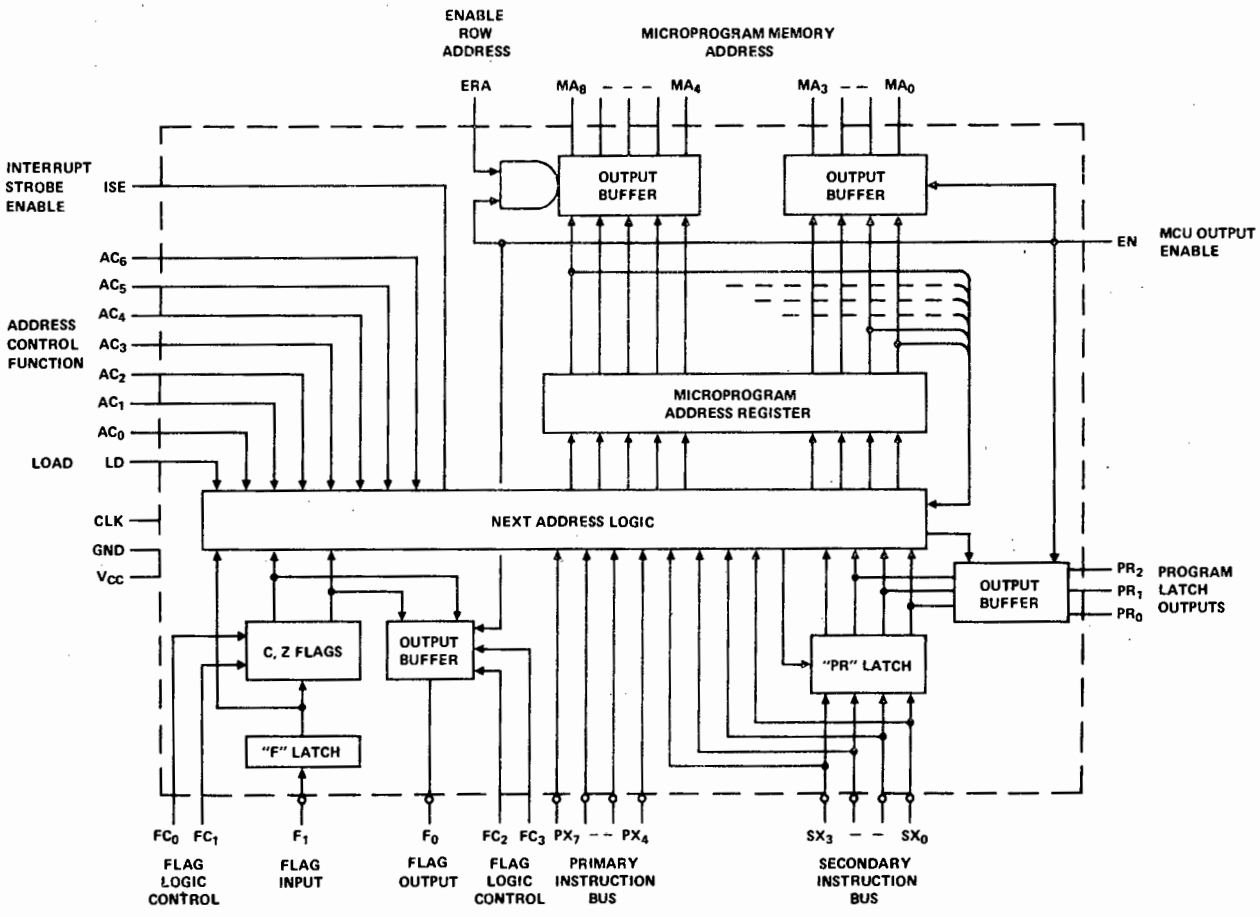


FIGURE C-1(a): THE 3001 MICROPROGRAM CONTROL UNIT

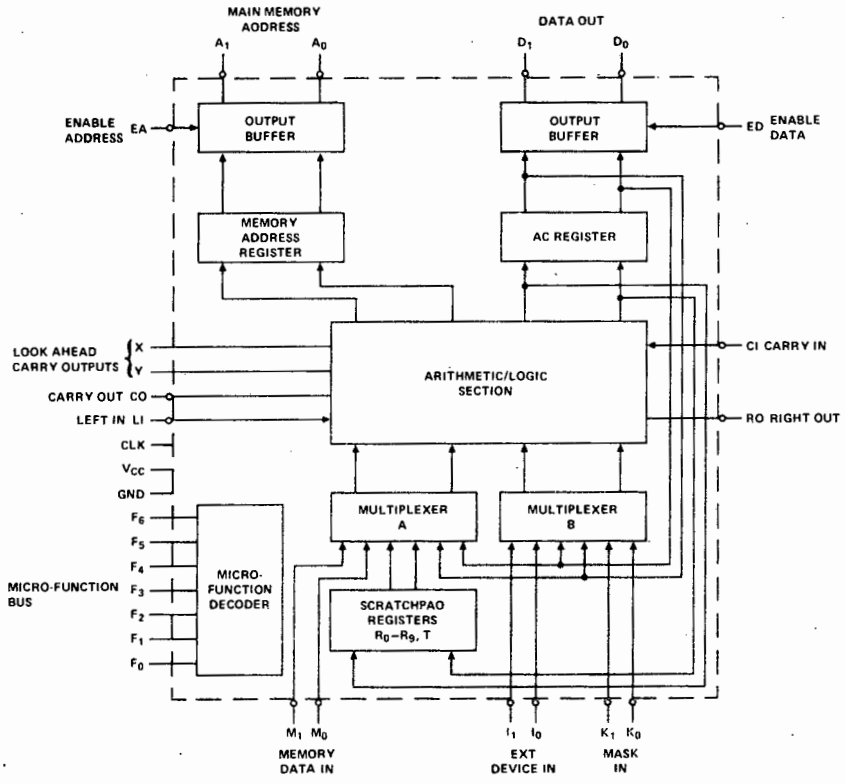


FIGURE C-1(b): THE 3002 CENTRAL PROCESSOR ARRAY

APPENDIX D: A CRITICAL REVIEW OF THE BASIC COMPONENTS USED TO IMPLEMENT THE MICROCONTROLLER.

The concept proposed in this thesis has become a really feasible prospect in terms of cost and simplicity only as the result of recent technological advances made in the field of large-scale integration, and using high-speed logic. At present, two technologies exist which may be considered of use in the microcontroller, as they provide viable, user-oriented elements. These are, firstly, those based on Schottky-clamped, bipolar TTL techniques and, secondly, those using emitter-coupled techniques. Reference [80] discusses the relative attributes of each technology, and carries out a valuable comparative study. A third technology which might be considered of use in bit-slice microprocessor systems, is that of integrated-injection-logic (I^2L). As will be discussed later, however, this admirable technology has at present severe speed limitations.

D.1 The Current Status of Bit-Slice Components.

At the present time, a growing number of manufacturers is producing bipolar bit-slice microprocessor elements. Four different families of elements are currently being supported. These are:

(i) The 2900 series, produced by Advanced Micro Devices [36], Motorola and Raytheon.

(ii) The 9400 series, produced by Fairchild [31].

(iii) The 3000 series, produced by Intel and Signetics [40].

(iv) The 6700 series, produced by Monolithic Memories [25].

Probably the most valid comparison which may be made between these series is that based on the capabilities of the central processor elements. The 2901 and 6701 elements are 4-bit-wide slices, and have very similar processing capabilities. Each has 17 working registers, and approximately 8 different possible arithmetic/logic functions. The 2901, however, has twice the speed capabilities, with a cycle-time of 100 nanoseconds. Of relevance in these two elements is the possibility of using double-address operations, thus allowing for data to be entered into the elements from two sources simultaneously. The 2901 is implemented using low-power Schottky technology, and its power dissipation is therefore significantly lower than that of any of the other similar bipolar elements. Currently,the 2900 series elements appear to be the most widely-used in industry - the 2901 itself forming the arithmetic unit of one of the newest and least expensive

minicomputers, the Interdata 6/16 {81}.

The 3000 series is a 2-bit-wide system, necessitating a greater number of actual elements for a particular application than would be the case if 4-bit-wide slices were used. It also has the disadvantages of a smaller number of internal registers and the lack of double-addressing facilities. Power consumption, too, is significantly higher than for the other series mentioned. It does, on the credit side, offer a rich instruction set, and a wide range of possible input ports. These are offset by a single address and a single output port. The strength of this series is its family completeness, as well as strong manufacturer support and relevant software (for example, a microinstruction Assembler program {71}).

The Fairchild 9400 series appears to be similar to the 2900 series. Little detailed information is, however, as yet available.

In the area of emitter-coupled technology, the proposed Motorola MC 10800 series (based on their ECL IOK technique) promises to be of great interest. The central processor element is a 4-bit slice, but is rather different in structure from the equivalent bipolar elements, containing a mask-programmable latch network, an arithmetic/logic unit, an accumulator and a shift network. Details of the system are still not freely available, but it would appear to offer a cycle-time of approximately 50 nanoseconds. The major drawback to its use is the problem of interfacing the elements to a system which would be predominantly implemented using bipolar techniques. Its application appears to be limited to large production areas. Ref { 73 } discusses the proposed specifications of the series.

The Texas Instruments development, integrated-injection-logic (I^2L), is applied in their SBP0400 4-bit-wide microprocessor element {82}. The I^2L technology's strength lies in the low power dissipation, and in its relative compatibility with bipolar elements. A major disadvantage is its speed, the SBP0400 having a cycle-time of 1 microsecond. Otherwise, this element is very versatile, having a rich instruction set and 9 working registers. It is significant, however, that the originators of the SBP0400 are intending to produce a similar, 4-bit element which, it is claimed, will be equivalent in performance to its bipolar rivals {81}.

Ref {83} discusses considerations to be taken into account when developing the internal architecture of a bit-slice system, and provides an excellent insight into the trade- offs which have to be made. Of particular significance is the concluding remark, "The availability of a low-cost means of microprogram development is liable to lead to the emergence of microprocessors with unique instruction sets, providing special advantages for

particular applications. This may allow the user to have more influence on product development, in that those microprogrammed innovations that have wide application could well impact the design of future microprocessors." It is important to stress that the available bit-slice elements are still in an early stage of development, and each will therefore have its own advantages and disadvantages. As happens with any emerging technology, there will undoubtedly be many changes in design philosophy as a clear pattern evolves.

D.2 Choice of Components for the Microcontroller.

In any engineering application, the choice of components required for a specific project will be based on a number of factors. The choice will, typically, be a compromise between ideal specifications and practical considerations. Thus, a designer might have to modify his "ideal" in order to ensure local availability and support for the components he requires. Also, he must consider carefully the impact on the rest of his system of the components selected. These principles have, therefore, to be remembered when determining the components which form the backbone of the microcontroller.

The first decision which has to be made, is that of technology. As discussed above, the choice lies between emitter-coupled and Schottky bipolar logic. In terms of ease of use, Schottky bipolar elements have the upper hand, providing, as they do, logic levels directly compatible with normal minicomputer interfacing requirements, and demanding conventional power supplies. In addition, in a system such as the proposed microcontroller (which comprises sections such as memory), direct compatibility in terms of logic levels and edge-speeds will ease design and testing. Emitter-coupled logic provides many headaches for the designer, which tend to counter the use of an otherwise valuable logic line. This is fully discussed in Ref {80}. The range of readily-available ECL elements is limited, and delivery times are usually prohibitively long for the consumer with very limited requirements.

For these reasons, it was decided to use Schottky bipolar elements in the prototype system. A major problem, however, results from this choice, namely power requirements. Such elements will have high power supply demands (for example, the microinstruction sequencer unit has a power supply current requirement of approximately 250 milliamps). Critical analysis of actual speed requirements will reduce this problem wherever possible. Certain areas of the microcontroller will not be required to operate at the same speed as the central processor array or the microinstruction sequencer;

for instance, a large percentage of the interfacing to the minicomputer may be implemented by using slower components. In these non-critical regions, low-power Schottky elements may usefully be employed to reduce power supply current.

The actual choice of microprocessor elements to be used within the microcontroller was extensively investigated. As pointed out above, there are many manufacturers producing suitable elements, but availability and family completeness tend to rule the final selection. In this case, the 3000 range of elements proved to be adequate, providing, firstly, the basic elements to implement (in a relatively simple fashion) the central processor array, the microinstruction sequencer, and the interrupt control unit. In the second place, the prime manufacturer, Intel Corporation, produces a compatible range of memory devices, latches and bus-driving elements. Finally, the Intel range is locally-available and well-supported.

D.3 Use of the 3000 Series Elements.

As may be seen throughout Chapter III and Appendix C, there are many difficulties to be encountered when using the 3000 series elements, but these may generally be overcome by careful hardware design and software construction. (An example here is the idea of "conditional clocking" as discussed in Section C.8).

There are, however, two serious limitations which must be faced. Firstly, a "bottle-neck" problem arises. This may be seen by observing the logical structure of the central processor elements (See Fig. C-1(b)). In most internal logical operations, the data concerned is routed through the accumulator (as is demonstrated by the instruction set shown in Table 2). Also, direct register-to-register transfers are not possible, and must be effected by means of a two-stage operation. The register contents are moved to the accumulator, and then the contents of the accumulator are moved to the required destination register. Some relief may be obtained by the use of the T-register, but this, too, has limitations. Data transferred to the external circuitry must also be routed via the accumulator. In all, this results in lengthy microsequences to effect primitive operations.

The second problem lies in the inability of the microsequencer to handle, internally, microinstruction subroutine calls. No internal provision is made for the storing of return addresses when a microinstruction sub-routine is entered. If such a facility is required, an external register must be provided. In the design of the microcontroller, this

problem was considered, and was resolved by being ignored! In other words, microinstruction sub-routines, in their classical form, were not permitted. This is not such a disadvantage as might be thought, since the system is strictly event-driven, and all microsequences will terminate in a common "idle-routine", awaiting the next event (i.e. the occurence of an interrupt).

Within the limitations discussed above, and accepting the "bottle-neck" problem, the overall microinstruction repertoire is satisfactory. As with any computing system, all the instructions that the user would desire will never be available!

From the point of view of purely practical implementation, the use of the 3000 series of elements presents very few problems. The only major factor to be considered is the question of heat dissipation. At an ambient room temperature of 23^{o} Celsius, the operating temperature of the microinstruction sequencer and the central processor elements is approximately 55^{o} Celsius! The actual elements are, however, specified to operate up to a temperature of 125^{o} Celsius, and so component failure is not of particular concern. Nevertheless, if (as has been proposed here) the system is to be contained within the mainframe of a conventional computer, the heat dissipation is important to consider, and techniques using forced cooling become relevant. In terms of the physical interconnection of the elements, no special techniques have to be introduced, as would be the case if emitter-coupled logic was used. Conventional techniques employed in high-speed digital circuitry, such as adequate decoupling of power supplies, short inter-element connections, correct termination of any unavoidably long lines, etc., prove to be sufficient, Reference {84} provides guidelines to the necessary practical considerations.

APPENDIX E : CONSOLE FACILITIES.

The role of the console in the microcontroller system introduced in this text, is twofold. In the first place, during the development phase, it is an obvious requirement that the system should be capable of being controlled as a stand-alone unit. In the second place, when taken into practical use (and remembering that the intention of the system as proposed is that it should be user-modifiable if necessary), it is an advantage to be able to monitor the performance of the microcontroller when introducing new microprocedures.

The console developed had, therefore, to meet both of these requirements. The functions which it was designed to permit were:

(i) The entry of data into, and the observation of data from, the microinstruction memory.

(ii) The entry of data directly into the central processor array.

(iii) The observation of the status of:

  (a) The data and address output buses of the central processor array.

  (b) The data on the two input buses of the central processor array.

  (c) The carry-in and carry-out flags.

  (d) The current microinstruction being executed.

(iv) The specification of initial microinstruction memory addresses.

(v) The initialization of the microcontroller and the starting of a procedure.

(vi) The single-stepping of microinstructions.

Appendix C.7 describes the hardware details of the console.

APPENDIX F: THE METHOD OF PLACEMENT OF MICROINSTRUCTIONS IN THE MICROMEMORY.

The constraints resulting from the addressing method of the microprogram sequencer unit employed in the microcontroller, present interesting problems to the user. Contemporary machine-code programmers are accustomed to great flexibility when allocating memory locations to their programs. This picture changes rapidly when allocating memory for the storage of a microprogram in the unit under discussion. Appendix C.8 describes the Intel 3001 microinstruction sequencer unit in detail. After much experimentation with various methods of placement, particularly those mentioned in {39} , the following approach is proposed. (It must be stressed that this strategy is intended primarily for a system into which multi-interrupt automatic micro-vectoring is incorporated)

(i) The microinstruction sequences are coded into relatively short-length blocks. The most convenient sequence-length is 16, which means that a sequence of microinstructions may be located within a common address column. Wherever possible, the expected column destination addresses of conditional jumps should be specified. It is also important at this point to separate long and short sequences.

(ii) Those instructions which, by virtue of their nature, are committed to certain locations are placed. A particular example of this is the interrupt-enabling target location (jump to row 0, column 15).

(iii) Row 0 should be kept for general pointer usage. This is particularly important if a multi-interrupt, automatic vectoring system is to be adopted. In general, the vector addresses will always be to column 15,the row address being determined by the level of interrupt. At these interrupt vector addresses, the most convenient response is a jump to row 0, in a column specified within the jump code being used.

(iv) Interrupts are dealt with first, as mentioned in (iii), that is, a jump to row 0, at a specified column. At row 0, a jump in current column to a specified row will determine the start address of the required microroutine. Note that this system can only be adopted for a maximum of 16 interrupt levels, and should be used primarily for those interrupts which have particularly awkward destination locations. All other interrupts should be handled by a jump within the specified row.

(v) Once interrupts have been dealt with, the long instruction sequences, coded as suggested in (i), may be placed. It is vital at this point to stress that the programmer must be willing to abandon the gener-

al principle of using sequential allocation. It is certainly simplest to stay within a current row, but when the end of that row is reached, two options are open. Firstly, and preferably, a jump should be made to a different row, but still in that column. If this is at the end of a column, the jump will be to the end of another row. The next jump should be back to the start of that row. In view of the ease of moving within a current row, the structuring of microprograms within short blocks can now be seen to be an advantage.

(vi) Catering for conditional jumps presents a major problem. When, say, testing for zero, the resulting jump will be to locations with column addresses, 02, 03, $12_{(8)}$ and $13_{(8)}$. This implies a further restriction. These jump targets should therefore be placed in unused locations in different rows.

(vii) This has, hopefully, eliminated the interrupt requirements and any long sequences. The next task is to deal with the short sequences. Spaces left unallocated when specifying conditional jump targets in (vi) can now be utilized. It will be found that a certain re-allocation of locations used in step (vi) will be necessary.

(viii) Finally, the remaining space is used to fit in the short, non-critical sequences.

A very valuable fact which emerges is that if a jump is required to a completely different row and column address from that currently being used, a two-stage operation is required (that is, a jump to the required column, and then a jump within that column to the specified row, or vice versa). Generally, this cannot be avoided, but what must be resisted is the temptation to fill the locations used at these intermediate stages with redundant instructions! In most applications, the microinstruction storage is very limited, and valuable space must not be consumed by the famous "no-operation" code!

Finally, this procedure might appear to the uninitiated to be time-consuming and full of hidden traps. In practice, it was found to be a relatively straightforward procedure if the above guidelines are adhered to. It was felt that the time taken to organise a microprogram did not justify investigating a possible method of automating the process. Even after a considerable period of investigation and research into methods of automation, Wakely of Stanford University {39} reported that an interactive method was still required!

APPENDIX G : THE EVALUATION OF A COMPUTER SYSTEM.

In computer system evaluation, there are many techniques which may be employed to ascertain the value of the particular system under scrutiny { 47 }. The commonest are described in this appendix. Ref { 46 } contains an excellent overview of the subject.

(i) Timing. This is the oldest, and possibly the most naïve, method. The sheer speed of the hardware is measured, usually in terms of cycle-times and arithmetical processing speeds (say, the time taken to implement an "add" instruction). The method ignores both the overall organisation of the machine and the software structure in general. The results obtained contain relatively little useful information except possibly an indication of the hardware configuration!

The method may be improved if a technique of instruction-mixing is introduced. In this approach, an execution-time is measured for certain categories of instructions, and an overall figure of merit is obtained by using "suitable" weightings for each instruction. The "mix" chosen would depend heavily on the application under consideration. The same defects that exist for evaluation of cycle-times and add-times hold good here, as the total system is not considered. There is also the problem of defining the system of weightings to be used.

(ii) Kernel Programs. A kernel program is a coded and timed "typical" program. It may be simple, or may include elaborate procedures involving a certain amount of input/output programming. These kernel programs are normally freely available, but there is unfortunately no universal standard. Auerbach have published extensive lists in their "EDP Standards" { 85 }. At the outset, such kernel programs may be seen as providing a very useful insight into computer system performance. In most cases, however, they tend to be restrictive to user program oper--ations, and overall system performance cannot readily be assessed, particularly in a multiprocessing structure. The problem in these cases is that input/output operations, are, in practice, asynchronous, as well as strongly dependent on the current state of the system.

(iii) Modelling Techniques. The technique of mathematical modelling of systems may valuably be applied in the evaluation of computer systems. Reference {4}gives a very useful description of the process as applied to system design. Numerous models have been developed, and have proved to be of great value, particularly in analysing a specific attribute of a system. The models may then be used to evaluate

the performance under changed conditions. Of particular interest in this project is the work by Stimler and Brons on the mathematical analysis used to calculate and optimise the performance of real-time systems {86}.

Inherent in the mathematical modelling technique is the development of suitable, accurate models. To ensure fidelity, they must be carefully tuned to the system under consideration, and be capable of accommodating any situation which may arise. In practice, many simplifications will be introduced, and this does somewhat limit a model's validity. The technique has great merit, however, given a comprehensive knowledge of the system being studied, as well as sufficient time to produce the model.

(iv) Benchmarks. A benchmark is an existing program, coded in a specific language, which may be executed on the computer system being evaluated. The benchmark approach is of particular value in determining the efficiency of compilation procedures. A suite of suitable benchmark programs can also usefully evaluate the performance of a system under different configurations or with various input/output facilities. Benchmark tests must be viewed with a certain amount of circumspection, however, as their use has in many cases led to erroneous and unfair comparisons. Consider the compilation and execution of a simple arithmetic routine. One system might make use of a compiler which generates highly efficient coding, at the expense of compilation time. A second system might produce very inefficient coding, but in a very short time.

An additional problem that arises with benchmark programs is the question of whether they are truly representative of the final user applications.

(v) Synthetic Programs. The synthetic program approach tends to combine the virtues of the benchmark and the kernel program methods. As with a benchmark program, a synthetic program is coded and executed, but is not necessarily a program already in existence. Like a kernel program, it does not represent a real user-type program but is coded normally, will include input/output operations, and will work within the operating system environment{87}To be of value, a synthetic program must include sufficient features to make use of all the system's facilities. As with benchmark programs, it does lead to the problem of selecting a suitable mixture of these features to fit in with the intended user applications.

(vi) Simulation. This powerful tool of the scientist has great value in the evaluation of computer systems. As with simulation in its broad context, two forms are normally adopted here {45}. The first is to derive empirically a sufficient amount of information about the system, and then to use this data to produce a simulated model of the system. The crux here lies in the accuracy of the information. The second method is based on event-orientated simulation, in which the actual operations of the computer system are modelled. Probability theory is introduced in order to describe the subsequent performance of the system. Either method leads, hopefully, to an accurate model of the system, and once this has been validated, system performance may be evaluated. With a suitable simulation strategy, changes in the system configuration may be introduced, and their resulting effects observed.

The major restraint which is inherent in the simulation approach is that of cost. In order to produce an accurate simulation, a great deal of information about the actual system must be obtained, and the resulting model must be carefully validated. The cost and time involved will be high, and generally the resulting model will be of value only in the simulation of that specific system, and of little use in the simulation of others.

(vii) Performance monitoring {38}. To the engineer, this approach is the most obvious one, but it is of relatively little use in the theoretical design of systems. Two methods of monitoring are widely used. Firstly, hardware-monitoring devices may be developed to observe and record parameters such as cycle-times, input/output response times and CPU idle times. The processing of the data obtained can be carried out fairly simply, if a degree of intelligence is built into the monitor. The acquisition facilities can usefully be connected directly to a small computing system to perform such a task {89}.

A second method of system monitoring may be introduced by the inclusion of suitable software within the computer system. These programs can collect data about the system's performance, and can carry out the required processing of this information. In most large computing systems, this form of monitoring is carried out as part of the general housekeeping and accounting systems.

It must be pointed out, however, that monitoring techniques are generally intended to evaluate the performance of an existing system {90}. The information produced can be of great value in determining the possible enhancement of the system, and then in assessing the subsequent effect

of enhancement. By definition, it does require the existence of a fully-operational system in the first place.

APPENDIX H: SOME THOUGHTS ON MULTIPROGRAMMING SCHEDULING ALGORITHMS.

"As in almost any rationing situation, there is considerable controversy over what are equitable, efficient or economical (scheduling) policies - a considerable amount of computer folklore has developed in this area" {53}.

Multiprogramming is a term given to a computing system which may have several tasks in a state of execution at the same time {53}. The simultaneous demand by "users" for the services offered by a computer system leads to the formation of queues- a situation which exists in any walk of life where there are limited facilities to be shared by many users. The queues that may develop are controlled by a suitable scheduling method, which chooses the order in which various users receive attention. The goal of scheduling is to provide the best possible service for all users {91} . The approach revolves around the philosophy adopted in a particular application, and is highly dependent on the choice of priorities. A widely-held opinion is that priorities may be "bought" (possibly by bribing), "earned" (by having characteristics favourable to the system), "deserved", or any combination of these three. This simple idea is of great value, and its implications may be seen in most contemporary scheduling arrangements {91}.

It is not intended, however, that this appendix should delve deeply into this highly complex field, as applied in large computer systems. Much of the information available on multiprogramming is devoted to commercial time-sharing systems and the related statistical problems. Reference {23} contains a comprehensive literature survey on this topic.

The interest in this present work lies in multiprogramming in on-line process-control applications - aptly called by Liu and Layland {92} multiprogramming in a "hard" real-time environment. Of the limited amount of literature available in this field, it is Liu and Layland's paper {92} which provides a really valuable survey of current attitudes. Of relevance also is the work by Martin {93} , which is, unfortunately, somewhat lacking in depth although it does cover a wide range of possible approaches to a variety of situations.

In general terms, computers applied to the so-called "hard" real-time situations have to cater for a number of time-critical tasks as well as, possibly, a number of non-time-critical tasks. The mixture of these two categories is, of course, application-dependent. A "scheduling algorithm" is a set of rules that determines the task to be executed at a particular

moment {92}. Two broad types of algorithm may be distinguished - static and dynamic. In the static case, all task priorities are fixed when the system is configured, and remain so throughout its subsequent use. "Dynamic" implies that the priority of any task may be altered during system operation. In most situations a mixture of the two, i.e. a mixed scheduling algorithm, will be adopted. In such situations also, the scheduling algorithms will be pre-emptive, and will be driven by the priority levels assigned. ("Pre-emptive" implies that tasks may be forced to release temporarily the use of the computer resources in favour of other tasks). This implies that if a task is being executed, and a higher-priority interrupt occurs, the current task will be suspended and that indicated by the interrupt immediately executed. The problem which then arises is that of developing suitable algorithms to cater for these requirements, and to lead to optimal supervision.

Following the work by Liu and Layland, certain terminologies may be introduced {92}. The "deadline" of a request for a task to be executed is defined as the time of the next request for that same task. An "overflow" will occur at time "t", say, if "t" is the deadline for an unfulfilled request, i.e., a request has not been honoured within the maximum time. For a set of tasks specified, the scheduling algorithm is termed "feasible" if task scheduling can be effected so that no overflow will ever occur. The "response time" for a request for a specified task is the interval between the request and the end of execution of that task. A "critical instant" for a task is the instant at which a request for that task has the largest possible response-time.

Of practical interest, and of importance in the evaluation of a real-time computer system, is the question of processor utilization. This may be defined in a variety of ways, but the following definition seems the most relevant:

$$UF = \sum_{i=1}^{m} \left(\frac{C_i}{T_i} \right)$$

where τ_1, τ_2, τ_3 ... τ_m denote m tasks, with request periods T_1, T_2 .....T_m, and having execution times of C_1, C_2 .....C_m.

This means that the utilization factor is the fraction of processor time spent on the execution of the specified tasks. Alternatively, it is one minus the fraction of idle processor-time. Thus, a suite of tasks may be said to utilize a processor fully if the priorities assigned yield a "feasible" scheduling algorithm, while an increase in the execution time of any of the tasks leads to an unfeasible algorithm.

H.1 Defining the Scheduling Algorithms.

In order to define suitable scheduling algorithms, a statement of the system objectives is required. This (from a theoretical viewpoint) is a relatively simple procedure. Following such a statement, algorithms may be developed for situations involving both static and dynamic scheduling. Liu and Layland carry out an excellent treatment along these lines and arrive at some most valuable conclusions. But (and this is where the problems start) all their work is based on system requirements which are totally unrealistic when applied to on-line process control, in particular the following premises:

(i) "The requests for all tasks for which hard deadlines exist are periodic, with constant intervals between requests"

(ii) "The tasks are independent"

(iii) "Run-time for each task is constant".

Such assumptions are, naturally, in a process-control environment, totally unacceptable. These authors do hasten to add that they recognise the limitations, but do not offer any valid means of minimising them.

Martin {93} does, however, accept a realistic situation. His approach, as adopted in this work, is that it is not really possible to produce hard and fast guidelines to meet the variety of practical situations which will arise. Nevertheless, the work by Liu and Layland is important, as many of the ideas produced prove to be valuable starting-points for subsequent studies.

Realistic system requirements may be stated as follows:

(i) The requests for tasks for which hard deadlines exist may occur in a random fashion.

(ii) Only certain tasks will have to be completed before being superseded. Others will be permitted "soft" deadlines.

(iii) Tasks may be interdependent.

(iv) Execution-times of tasks may be variable.

The scheduling algorithm adopted in the experimental "process-control" environment described in the text (the third experiment) is based on the following principles:

(i) All tasks are assigned a unique, fixed priority. This means that each task will have a different priority level from any other task. This method greatly aids the selection of "next tasks", as even if a task is suspended, it will retain its priority assignment, regardless of the time it may have been in execution before suspension. An exception is introduced in (ii) below.

(ii) All tasks are defined as either interruptable or non-interruptable. (As discussed in the text, such a sit--uation does arise in process-control applications. For example, a low-priority task might not have a deadline, but once its execution has commenced, it must not be stopped. A case-in-point might be the control of a conveyor-belt system. The time when the overall operation starts might not be critical, but once the conveyor belt has been started, all subsequent routing must immediately be established).

(iii) Queues are permitted to form in such a way that whenever a request for a task arrives (i.e., an interrupt occurs), if the priority of that task is lower than that of the current task, the requested task is placed in a priority queue. When the next task is to be selected, the highest-priority task waiting for execution is allowed to run.

(iv) Each task must be completed before it may be requested again. (This will be ensured if the scheduling algorithm is feasible).

(v) The object of the scheduling is to achieve the best possible utilization factor.

(It should be remembered at this point that the operating system is event-driven. This implies that tasks are requested by interrupts, the sources of which are categorised in Section 4.4.)

In order to achieve these goals, a large amount of information must be collected, relevant to the execution of the required tasks. Note also that (i) indicates that the scheduling algorithm is based on fixed priori-

ties. The possibility of dynamic scheduling is, however, introduced by (ii). What this amounts to, is that a certain amount of dynamic re-scheduling is permitted, whereby low-priority tasks are temporarily allocated maximum priority while they are in execution, if they are non-interruptable.

A logical step at this point is to set about analysing and evaluating the scheduling system proposed. Two standard methods which might be applied would be (i) simulation, using available, general-purpose simulation programs, such as GPSS {94}, or (ii) analytical techniques. Of interest and relevance to the latter approach is the survey by McKinney {23}. In this survey, various applications of analytic techniques are discussed, with particular reference to "Round Robin" and "Multiple-Level Queue" methods.

The object of such an analysis is to determine the effectiveness of the proposed strategy, and to analyse the overall system impact which occurs if modifications (such as a change in priority assignment) are introduced.

Such research is undoubtedly of great importance, and essential in large-scale applications. It must be remembered, however, that the techniques discussed in this dissertation are aimed at relatively small systems (in large-computer operating-system terms), with clearly-defined objectives. So, while a theoretical diversion is attractive, it was decided that a more practical analysis would be of greater value. The technique used was, therefore, to study the feasibility of the scheduling, i.e., whether it was possible, in practice, to implement. Then, in the practical application of the scheduling, its success (or otherwise) would be observed. The analysis is, therefore, based on objective, practical reasoning, rather than on theoretical (and slightly hypothetical) computerised studies.

H.2 Analysing the Proposed Algorithm.

The following analysis, based on the above principles, relates the actual situation to that outlined in the third experimental situation, as described in Chapters IV and V. Although the number of tasks has deliberately been limited, the method is applicable to a wide range of situations.

Consider a situation with 5 tasks, τ_1, τ_2, ... τ_5, such that:

(i) τ_1 and τ_2 are periodic, their request periods being T_1

and T_2 and their run-times C_1 and C_2.

(ii) τ_2 is requested by τ_1 every n.th time that τ_1 is completed, i.e. τ_2 has a request period of $T_2 = nT_1$, and a run-time of C_2.

(iii) τ_3 is random, such that its request period is T_3 where $t<T_3<\infty$ $(t>0)$, but its run-time, C_3, is fixed.

(iv) τ_4 is a background task, and will be brought into execution when the processor is in an idle state. τ_4 may have a totally-variable run-time, determined by the processor-time available.

(v) τ_5 is a task occurring randomly and infrequently. It is vital that it be executed immediately it is requested. Its run-time, C_5, is unimportant.

τ_1, τ_3 and τ_5 are defined as non-interruptable tasks. τ_2 and τ_4 are interruptable.

The next problem is to assign priorities to the tasks. It is inherent that the priority of task τ_4 will be the lowest and that of τ_5 the highest. Beyond this, the most critical time for any task will occur when that task is requested simultaneously with higher-priority tasks. Consider the case when a low-priority task, τ_m, say, has been requested and the subsequent execution commenced. If a higher interrupt occurs, say task τ_i, where $i<m$, then τ_m must be suspended for a time (i.e. C_i) to permit τ_i to be executed. Extrapolating this, it is apparent that the most critical situation arises when requests for all tasks occur simultaneously. (A formal proof is given in {92}.) From this result it may be concluded that a feasible scheduling algorithm can only be possible where all requests can be met before their respective deadlines, even if requests for all tasks occur at the critical time.

This conclusion leads to a means of priority assignment. Consider two tasks, τ_n and τ_m where $T_n<T_m$. If τ_n has a higher priority, then it may be seen that:

$$\left\lceil \frac{T_m}{T_n} \right\rceil \times C_n + C_m \leq T_m \qquad \text{(i)}$$

Since $\left\lceil \frac{T_m}{T_n} \right\rceil$ is the integer ratio of occurrences between T_m and T_n, and remembering that $T_n < T_m$, so that within time T_m there must be time to execute τ_m plus $\left\lceil \frac{T_m}{T_n} \right\rceil$ times the number of requests that occur for τ_n.

If τ_m has a higher priority, then:

$$C_n + C_m \leq T_n \qquad \text{(ii)}$$

$$\text{Since} \left[\frac{T_m}{T_n}\right] C_n + \left[\frac{T_m}{T_n}\right] C_m \leq \left[\frac{T_m}{T_n}\right] T_n \leq T_m$$

This means that if $T_n < T_m$, and C_n and C_m are such that the scheduling algorithm is feasible with τ_m at a higher priority than τ_n, it will also be feasible with τ_n at a higher priority then τ_m. The opposite, i.e., $T_n > T_m$ will not be true. This means that the priority assignment should be based according to request rates, i.e., the higher the request rate, the higher the priority (Proof due to {92}).

This procedure may therefore be adopted in the allocation of priorities for tasks τ_1 and τ_2. The random task τ_3 presents certain difficulties which appear to require a degree of statistical knowledge of the nature of this task. The practical characteristics of τ_3 will determine whether this is necessary. In the case under consideration, it was felt that this was not warranted, and the approach adopted was to scrutinise the characteristics of the task, and to use a "worst-case" analysis. In other words, the shortest possible request period T_3 (i.e. t) was established, and was used as a fixed value for this task. A statistical mean value might be more appropriate, but this would result in the possibility of certain instances when the scheduling algorithm becomes unfeasible. The approach caters for the worst possible case, and at any other time, any "slack" in the system which results from τ_3 having a longer request time will be to the benefit of the system as a whole.

From the point-of-view of the utilization factor of the processor, two approaches should be made. Inherently, the system will be utilized to a maximum, in view of the fourth task which will be executed whenever possible. If such a task is not present, the UF would be:

$$UF_4 = \frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{C_3}{T_3} + \frac{C_5}{T_5}$$

Again, the question of assigning a value to T_3 must be considered. In this case, because an overall figure is valuable, the statistical properties of τ_3 could be evaluated and a suitable value of T_3 arrived at. It was decided that worst-case analyses are of greater interest, however, as they reveal the maximum efficiencies of which the system is capable.

When τ_4 is present, the UF will be brought to a maximum, the value of which is dependent on the "time-wasted" in effecting task-swopping.

A final consideration must be the question of the effect of

the "rescheduling" caused by the activating of non-interruptable tasks. It is clear from what has been described above that if the scheduling algorithm is feasible in the first place, then the rescheduling will not have any effect on the total algorithm. This is because the algorithm has its fundamental origins in an absolutely worst possible situation, i.e., at the critical time. Since task run-times will not be altered, no extension of the resulting deadlines can occur.

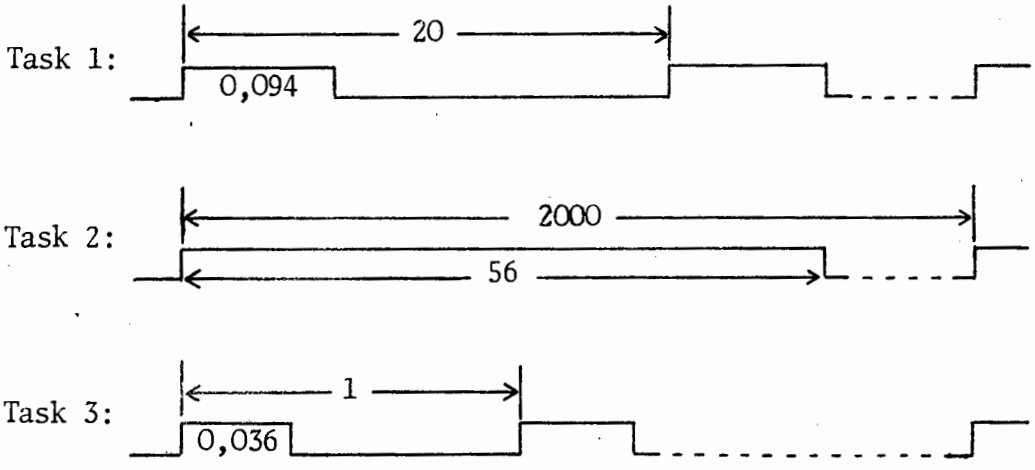
H.3 Applying the Algorithm in Practice.

In the third experimental situation, the five tasks specified had the following parameters, expressed in milliseconds:

	Request Period T_n	Run-Time C_n
Task 1, τ_1	20	0,094
Task 2, τ_2	2000	56
Task 3, τ_3	$1<T_3<\infty$	0,036
Task 4, τ_4	Whenever possible	Max Allowable
Task 5, τ_5	*	*

* Task 5 is the power failure/restart task, and thus must have the highest priority, with a run-time which is not critical.

The problem is to resolve the priorities of τ_1, τ_2 and τ_3, within the theory developed in the last section. On the assumption that the higher the request rate, the higher the priority should be, τ_3 must have the highest priority of the three, and τ_2 the lowest. The next point is to check the feasibility of this arrangement. The following sketch aids the expression of the relationships previously discussed.



This assumes the most critical situation, that is, when requests for all three tasks occur simultaneously. The critical deadline, then, is that of task 2 - in other words, it must be possible to complete all tasks within this deadline. Therefore:

$$\left[\frac{2000}{20}\right] \times 0{,}094 + \left[\frac{2000}{1}\right] \times 0{,}036 + 56 \leqslant 2000$$

for the algorithm to be feasible,

i.e. $\left[\frac{T_2}{T_1}\right] C_1 + \left[\frac{T_2}{T_3}\right] C_3 + C_2 \leqslant T_2$ (from equation (i))

$\therefore$ $9{,}4 + 72 + 56 = 137{,}4 \leqslant 2000$

$\therefore$ the algorithm is feasible with the priorities as assigned.

Analysing the Utilization Factor (UF), where

$$UF = \sum_{i=1}^{m} \left(\frac{C_i}{T_i}\right)$$

This gives

$$UF = \frac{0{,}094}{20} + \frac{56}{2000} + \frac{0{,}036}{1} = 0{,}0687$$

In order to evaluate the utilization factor for the overall system in which the background task, τ_4, is included, the following method may be used:

The assumption is made that τ_4 may be executed between <u>each</u> execution of tasks 1 to 3. This is obviously only an approximation, and will not be true in the case where simultaneous requests occur. In the average over a long period, however, it will yield sufficiently accurate results, and still be a worst-case evaluation.

Considering a 2-second interval, the total time used to execute tasks 1 to 3 will be given by:

$$\left[\frac{2000}{20}\right] \times 0{,}094 + \left[\frac{2000}{1}\right] \times 0{,}036 + \left[\frac{2000}{2000}\right] \times 56 \text{ milliseconds.}$$

$= 137{,}4$ mSec.

Now, total time taken to effect the swopping-in and -out of these tasks is given by:

$\left(\left[\frac{2000}{20}\right] + \left[\frac{2000}{1}\right] + \left[\frac{2000}{2000}\right]\right) \times$ Swop-Time, and swop time was 0,1656 mSec. (See Appendix I)

$= 2101 \times 0{,}1656 = 347{,}93$ mSec.

$\therefore$ Total time to implement tasks 1 to 3 is (137,4 + 347,93) mSec

$= 485{,}33$ mSec.

$\therefore$ Time available to execute task 4 is
(2000 - 485,33) mSec (Since the time interval under consideration is 2 seconds) $\approx$ 1514,7 mSec.

On the assumption that task 4 will go into execution between each of tasks 1 to 3, there will be 2101 task-swops required.
$\therefore$ Request period for task 4, $T_4 = \frac{2000}{2101} = 0,952$ mSec, and the time taken to effect the swops is 2101 x 0,1656 = 347,93 mSec
$\therefore$ Total run-time available to task 4 is given by
1514,7 - 347,93 $\approx$ 1166,8 mSec.
$\therefore$ Average run-time for task 4, $C_4 = \frac{1166,8}{2101} = 0,555$ mSec
$\therefore$ UF $(\tau_4) = \frac{0,555}{0,952} = 0,582$
$\therefore$ Total Processor Utilization Time, $UF_{Total} = 0,0687 + 0,582 \approx 0,65$.

This is a most significant figure, as it indicates that the processor itself is being used (in the worst possible case) for the actual execution of tasks for 65% of the total available time. It should be reiterated that the power fail/restart task, τ_5, has not entered into the calculations. This is acceptable, as it is a task which only occurs in a crisis situation, and results in the suspension of all other tasks and the closing-down of the system.

APPENDIX I: QUANTITATIVE ANALYSIS OF THE EXPERIMENTAL CONFIGURATIONS.

This appendix details the measurements which were made on the three experimental configurations described in Chapter IV. The third configuration is the most relevant in view of the objectives of this project. The first two situations, however, provide a useful insight into the problems which must be overcome by the hardware-based real-time operating system, and constitute convenient mechanisms for the taking of relevant measurements.

I.1 Experimental Configuration 1.

The periodic nature of the tasks in this configuration allowed for accurate determination of the time taken to effect a task-swop. In view of the fact that the swopping-algorithm and its hardware implementation were identical in the subsequent experiments, this measurement is of universal value. The measurement was made by means of external instrumentation which detected the issuing (by the microcontroller) of the interrupt to the minicomputer, suspending execution of its current task. Secondly, the instrumentation detected the execution of a special single-word instruction, placed at the end of the minicomputer routine used to control the swopping-in of a new task. In practice, the average interval between these signals (which represents the total swop-time) was measured at 165,6 microseconds. The theoretical determination of this time may be based on an estimation of the total number of minicomputer cycles which must occur to send and fetch the respective process control blocks. This was estimated to be approximately 80 full computer cycles, which (at a cycle-time of 1,8 microseconds) implies a theoretical swop-time of 144 microseconds. The difference is accounted for by the time required to synchronize the two systems. It is interesting to note that the microcontroller requires 128 microcycles to effect the operation and, at a microcycle-time of 0,35 microseconds, this amounts to a possible swop-time of 45 microseconds. The degradation of performance lies in the minicomputer, and optimum overall system performance may be achieved only by re-designing the method of communication adopted. This could be carried out by ensuring close synchronization between the two systems, so that the minicomputer could input and output the data in a continuous sequence of instructions. In terms of overall system efficiency, however, the swop-time achieved is acceptable.

The second point of interest in this configuration lies in

the overall processor utilization. This will, naturally, be a direct function of the rate at which tasks are swopped. The higher the swop-rate, the more time is spent in swopping, and the less available for the execution of tasks. In the experimental situation, the quantum of time allocated to each task could be varied, so that different strategies could be observed. The criteria used to determine the most desirable swop-rate are discussed in Ref. {52}. In the actual experiment, it was decided that the scheduling objective was to operate the fast paper-tape reader at its maximum possible speed. This, in turn, yielded an acceptable response-time for the user of the teletype terminal, as well as an efficient use of the paper-tape punch. The reader was capable of operation at a speed of $\pm$ 140 characters per second, so the swop-rate was set at $(\frac{1}{140})$ seconds (7,14 milliseconds). In terms of processor utilization (UF), this implies that the number of task-swops which must be effected per second was (4 x 140) = 560.

At a swop-rate of 165,6 microseconds, the total time used in effecting swopping of tasks was $(560 \times 165,6) = 92,74 \times 10^3$ μ sec.

$\therefore$ Total available time per quantum was given by

$(\frac{1000 - 92,74}{560}) = 1,62$ milliseconds.

This time-quantum had to be checked to ensure that each task could, in fact, be viably executed at that rate.

The resulting processor utilization is, therefore:

$UF = \sum_{i=1}^{m} \left(\frac{C_i}{T_i}\right)$, Where C_i are the execution times and T_i the request times of the tasks.

$\therefore$ $UF = \frac{4 \times 1,62}{7,14} = ,91$ (See Appendix H)

This implies that at the chosen value of the quantum, the utilization of the processor is 91% of the total available time.

In order to effect the communication between the two systems, the minicomputer required programs which occupied 76 locations. The microcontroller required the use of 123 microinstructions.

It was also possible to measure the response-times of the microcontroller. The time taken by the microcontroller to accept an interrupt and to access the first microinstruction in the corresponding microroutine was 0,72 microseconds. The total time taken for the microcontroller to accept an interrupt and request the suspension of the task currently in execution in the minicomputer was 5,4 microseconds (15 microcycles). In general, the response-times and the swopping-of-task times will remain constant for all applications.

I.2 Experimental Configuration 2.

This exercise was rather more of a feasibility study than a measurable experiment, as it was aimed at applying the microcontroller to a real-time disc-operating system. The practical limitations which resulted from the equipment in use and the non-re-entrant interpreter program structures, implied that the background tasks could only be swopped at an interval greater than the time required to write and read a task from the disc. This resulted in the adoption of a swopping-rate of 570 milliseconds (allowing, on average, time-slices of approximately 70 milliseconds to be allocated to each user each 1,14 seconds), which is not very satisfactory. The possibility of the foreground task's being requested during the disc transfer operation was very high, the execution of the foreground task being requested every 1 second. This further degraded the system. To improve user response, the scheduling algorithm allowed for a background task-swop to be missed, if a previous disc-transfer had had to be aborted because of a request for a foreground task. Note, also, that if the foreground task is requested more often than every 1 second, the system can become "deadlocked", as it could be possible for no disc-transfers to be completed! This is an extreme case of the so-called "deadly embrace", as discussed by {95}.

This configuration did, however, serve to demonstrate the versatility of the basic concepts on which this work is based.

I.3 Experimental Configuration 3

The result of experiments in the use of such a configuration determines the overriding acceptability of the proposed means of effecting a real-time operating system. The situation is representative of a typical on-line process-control application; as is described in Chapter V, the essential difference between the experimental configuration and a real industrial application lies in the number of tasks included within the system.

It is of importance also in this configuration that a series of comparisons may be made between the performance of the microcontroller-based system and that which would result from the control of the situation, using a conventionally-structured system. Ideally, it would be desirable to execute the identical tasks with the identical minicomputer, but using a conventional real-time operating system. This was not immediately possible with the minicomputer available for the experimental work relating to this

operating system, and represent (in basic terms) the average time required to write data onto, or to read data from, the disc system. They are, naturally, directly related to the choice of hardware. This implies that meaningful comparisons cannot be made.

(v) Longest Uninterruptable Time. Within the operating system, it is important to have an estimate of the longest possible time during which no interrupt will be accepted, or responded to. This aspect is difficult to quantify, since it will be a direct function of the user's tasks and of the particular configuration used. What may fruitfully be examined, however, is the maximum non-interruptable time which exists within the executive component of the operating system.

(vi) Core Utilization. At this stage, the sheer overheads in terms of core required to implement an operating system (in other words, the amount of computer memory which must be allocated to contain the operating system) will be analysed. A fuller discussion is presented in Chapter V.

1.3.1 Results and a Comparative Analysis

In the table which follows, the actual parameters measured are shown, together with (wherever possible) the comparative figures obtained from the two conventional systems and the experimental RTEX-based configuration. The two operating systems chosen were:

(i) The Interdata OS/32 MT Real-Time, Disc-Based Operating System {59}.

(ii) The Data General RTOS Real-Time, Core-Based Operating System {58}.

PARAMETER	MICROCONTROLLER SYSTEM	RTEX	RTOS Note (v)	OS/32 MT Note (iv)
Time to recognise the presence of an interrupt.	1,8	7,2	10,6	6
Time to accept an interrupt and initiate the task requested. Note (i)	171	408,6	261,5	2106
Time to effect a task-swop. Note (i)	165,6	408,6	200	2100
Time to initiate a task called by another task. Note (i)	171	471,6	200	2300
Time needed to update the Real-Time Clock.	5,4	3,6	9,5 + nt Note (ii)	1500
Intervals at which the Real-time Clock may be updated.	5,4	1000	Note (vi)	Depends on RTC used
Time to react to an interrupt generated by a peripheral. Note (i)	1,8	7,2	10,6	6
Time to initiate an input/output operation. Note (i)	171	408,6	± 300 Note (vii)	1800
Time to terminate an input/output operation.	171	408,6	± 300 Note (vii)	1700
Longest non-interruptable time. Note (iii)	7,7	495	Not available	97
Core required to implement a basic operating system (in actual locations).	76	508	2805	± 10K

TABLE 3: Comparison of Key Real-time Operating System Parameters

I.3.2 Notes to Table 3.

A. All Times are in microseconds.

B. (i) Assumes the requested task is in memory, and has a higher priority than the current task.

(ii) 'nt' is time taken by particular update routine.

(iii) Only considers tasks which form part of the operating system, and excludes any user tasks.

(iv) 32-bit words.

(v) 16-bit words.

(vi) Function of hardware setting of RTC.

(vii) Dependent on peripherals concerned.

I.4 A Review of the Results.

Table 3 sets out parameters of importance when evaluating the effectiveness of real-time operating systems. On a quantitative basis, the hardware-implemented system claims improvement in the following areas, related directly to the performance of real-time computer systems.

(i) Interrupt handling.

(ii) Task-swopping.

(iii) Real-time clock control.

(iv) Memory requirements.

I.4.1 Interrupt Handling.

Two aspects are at issue here: the time taken to recognise the presence of an interrupt, and the overhead imposed on the processor in dealing with such an event. The high speed at which the microcontroller operates leads inherently to a rapid recognition of the occurrence of an interrupt, as is demonstrated by the figures quoted in Table 3. As to the time taken to process that interrupt, the delegation of this task to a separate unit, external to the processor, implies that this function is not imposed on the processor. In the case of RTEX (See Appendix J.3), the handling of an interrupt, even if no immediate action must be effected, can imply "waste" of processor-time of up to 472 microseconds. Thus, each time an interrupt occurs, the processor is not available during this interrupt-handling period. In a practical situation, the problem may be reduced by extensive use of interrupt masking, whereby interrupts may occur, but will be "masked out" so that they do not immediately interfere with the task currently in execution. Only when the operating sys-

tem sees fit to change this "mask" will any action be taken. This system must be used with great care; in a typical process-control situation, certain interrupts will require immediate attention, and should never be "masked out". The solution adopted by many systems (for example, the Data General RDOS {28}) is to permit certain high-priority interrupts to be recognised immediately, while "masking out" others.

The hardware-implemented system does not present these problems, since all interrupts are dealt with virtually as soon as they occur, and the appropriate action taken. The comprehensive priority-structuring and the full vectoring of the unit also mean that the source of the interrupt is immediately determined. In most conventional systems, vectored interrupts are available, but normally to a limited extent only. Software interrogation is necessary to determine the precise source.

I.4.2 Task-Swopping.

The time required to cause suspension of a current task and to start execution of a new task is of major importance when analysing the overall efficiency of a system. The time involved is largely system-dependent: an incore operating system, with all tasks immediately available, will be inherently faster than a disc-based system in which tasks are not necessarily core-resident. Thus, it can be seen that while the Data General RTOS requires a swop-time of the order of 200 microseconds, a large, disc-based operating system (such as the Interdata OS/32) will require 2100 microseconds! It must be realised that this swop-time, although vital, is strictly non-productive. In the case of the hardware-implemented system, the time is somewhat reduced, and (as indicated in J.1) reconfiguring the hardware can lead to a swop-time of the order of 45 microseconds.

The effect of the swop-time is dramatically demonstrated by considering the processor Utilization Factor (UF) obtained for the third experimental configuration. As shown in Appendix H.3, the UF for the configuration (under the control of the hardware-based operating system) is 0,65, that is, 65% of the available computing time is used in the actual processing of tasks (i.e., in useful work). When controlled by RTEX, the incore real-time operating system, however (See Appendix J.2), the UF is degraded to 0,076, that is, only 7,6% of the total computing time is devoted to task execution.

I.4.3 Real-Time Clock Control.

The value of this factor is very much a function of the actual

computer configuration. In a conventional structure, one real-time clock is normally included, and suitable software is used to create any other "system" clocks or time-of-day clocks which might be required. The updating rate of the real-time clock, being hardware-controlled, is normally predetermined. The advantage of the hardware-implemented system is that a multiple real-time clock facility may be provided at no extra cost in terms of hardware and software, each clock being independently controlled by an external (or interrupt) signal source.

Of value in the hardware system is the fact that time-of-day clocks and interval timers may constantly be monitored without any overhead being imposed on the central processor. This may be achieved in a conventional configuration, at the cost of additional hardware. An interval timer may be designed, the required timing interval sent to the unit, and an interrupt to the processor created when that time has elapsed. This does, however, require a suitable hardware structure.

The flexibility which the hardware-implemented real-time operating system offers is the major factor in this area of consideration.

I.4.4 Memory Requirements.

The utilization of available memory space must be considered here, as well the the protection and security of the operating system. Table 3 shows the memory requirements of various systems, and the advantages of the hardware-based system. In view of the decreasing cost of providing memory, however, the question is "how" the memory is being used, rather than "how much". In a conventional system, protective hardware and software must be incorporated so that the memory containing the operating system can never be accidently corrupted. This is not a simple facility to provide, for, while part of the operating system may be contained in read-only memory (or protected by other hardware/software strategy), certain areas (notably for the storage of process control blocks) must be read/write memory, and are therefore difficult to protect.

In the hardware approach, the storage containing the operating system, including alterable sectors, is totally contained outside the processor's own memory. The few locations which are required to contain the computer's program to effect task-swopping may be held in read-only memory. Operating-system security is therefore inherent.

Finally, the low memory overhead imposed by the hardware implementation implies a greater flexibility in terms of placement of user's tasks, within the available memory space.

I.5 Effect of Swop-Time on Average Task Delay.

A prime reason for adopting the Hardware-based Real-time Operating System strategy is to reduce the effective swop-time. If the times quoted in Table. 3 are included in a calculation of average task delay time (obtained from equation 13 in Appendix L), the significance is demonstrated. It will be seen from the task characteristics which are used in the third experimental configuration, that the average rate of requests is 1/2101 (per second) and that the average task duration is 0,036 milliseconds. To this average task duration must be added the appropriate task-swop time. The following estimates then arise:

(i) Under the control of the microcontroller, the average delay time is ≈0,19 milliseconds

(ii) Under the control of RTEX, the average delay time is ≈0,44 milliseconds

(iii) Under the control of RTOS, the average delay time is ≈0,23 milliseconds

(iv) Under the control of a disc-based system, OS/32, the average delay time is ≈ 2,1 milliseconds.

I.6 Conclusion.

To summarise these analyses, it is worth considering the solutions to the problems involved in task-swopping, as offered by a leading manufacturer of minicomputers designed specifically for process-control applications. Siemens of West Germany have currently produced two parallel systems, each representing a different philosophy. The Siemens 320 Process Computer {100} adopts the method of having all processor registers located in the core memory. A block of 64 registers is made available for each task. When a task-swop is implemented, the only necessary action is the altering of a "pointer" so as to allow the selected task to access its particular register block. Thus, register loading and storing are not required. A task-swop may, therefore, be effected within a very short time, typically 4 microseconds. The main drawback to this approach is that all register operations will require a memory-read or -write cycle. The average processor cycle-time is degraded to 4 microseconds, resulting in inherently slow program execution {101}.

In the Siemens 330 Process Computer, a conventional approach to the swopping of tasks is made, that is, the active registers are saved, where necessary, in the memory. Thus, a task-swop will take a minimum of 204,5 microseconds {101}. To provide greater system efficiency, the 320 has a dual-processor configuration, a second processor being devoted to the control of input/output operations and interrupt handling {102}. The idea is that the main processor will only be interrupted when a task-swop is required. The dilemma of the hardware designer is clearly indicated!

The designer of minicomputer systems for use in real-time applications must provide an architecture allowing highly-efficient task-handling, yet simple enough to be a viable industrial component. It has been shown above that the proposed hardware-based real-time operating system meets these requirements.

APPENDIX J: - A REAL-TIME EXECUTIVE PROGRAM FOR THE VARIAN COMPUTERS.

In order to demonstrate the effectiveness of the hardware-orientated strategy, it is essential to compare the performance of the microcontroller system with that of a conventionally-configured arrangement. The most logical approach would be to study the performance of two identical computer-system configurations with identical tasks, etc., operating the first under the control of the microcontroller and the second under the supervision of a commercially-available, real-time operating system.

The experimental situations to which the microcontroller-based, operating system was applied involved the use of Varian 620 Series computers. This range of minicomputers presents a major problem, since it does not have available a suitable, real-time operating system. Only in the latest series, the V70 range, have Varian Data Machines produced such a facility, known as VORTEX {96}. Unfortunately, due to the introduction in this new series of an extended instruction set, VORTEX cannot be executed in the 620 series. The only software packages which may be implemented in the 620 series, and which approach the requirements of a real-time operating system, are those called "BEST" {97}, and "MP3" {98}. "BEST" is designed simply for the handling of time-shared "BASIC"-language programs. The "MP3", which has many relevant specifications, is available only at a high cost, being produced by an independent software house. The operating system normally used on the 620 series, MOS (Master Operating System) {99}, is a non-real-time, single-user system.

To overcome these difficulties, it was necessary to develop a suitable, real-time operating system, which could be used in the comparative studies. This, naturally, is a formidable task, so a minimal system was created, with relatively restricted facilities. The system is referred to as RTEX, a Real-Time Executive.

J.1 An Overview of RTEX.

The philosophy adopted in the definition of RTEX was based on the proposals made in Chapter II. This has the advantage of providing for meaningful comparisons in subsequent performance evaluation tests. RTEX therefore has the following structure:

(i) It is an in-core, pre-emptive, event-driven system. No facility is included to provide for file-management, disc-handling or memory management. As in the hard-

ware-based system, all peripheral-handling is effected by predetermined tasks with a nature similar to that of the user tasks.

(ii) Being pre-emptive, queues are formed, and used to determine the next task to be executed. The queues are always maintained in a strictly priority-ordered form.

(iii) All tasks are assigned a unique priority, and may be defined as being either interruptable or non-interruptable. All information relevant to each task is held in a unique process control block (PCB).

(iv) Foreground-Background modes of operation are permitted.

(v) Multiple-level interrupts are permitted. Both periodic and asynchronous interrupt signals may be used to drive the system. Intertask communication and task-completion are indicated by software "traps" back into the executive. These are termed pseudo-interrupts. Intertask communication is aided by means of buffer passing.

(vi) Time-dependent tasks may be specified. These are tasks which must be executed after suitable intervals, or at certain times. A real-time clock-handling routine is available.

RTEX may, therefore, be seen to form the kernel of a conventional real-time operating system, and may be compared to the Data General RTOS [58], which is available for the Nova range of minicomputers, and for which performance figures are quoted in Appendix I. Having been specifically developed for this dissertation, RTEX has the advantage of having a fully-defined and well-understood structure, making possible the insertion of "software probes" (See Appendix I), which may then be used to control external monitoring equipment.

J.2 Analysing the Application of RTEX.

To provide realistic comparative figures, an environment was created, identical to the third experimental configuration. The scheduling algorithm adopted was identical in all respects to that discussed in Appendix H, as was the assignment of task priorities.

J.2.1 Measurements.

(i) Core required to contain RTEX - 508 locations.

(ii) Minimum response time to an interrupt - 408,6 microseconds.

(This is measured as the time between the arrival of an external interrupt requesting execution of the highest-priority task not currently in execution, and the time at which execution of this task commences.)

(iii) Time taken to acknowledge the occurrence of an interrupt - 7,2 microseconds.

(iv) Time taken to acknowledge an interrupt requesting a task with a lower priority than that currently being executed, to place the request in a wait-queue, and to return execution to the current task - 275 microseconds.

(v) Time taken to effect a task-swop - 408,6 microseconds.

(vi) Time taken to effect a task-swop which is requested by means of a pseudo-interrupt - 471,6 microseconds. (This occurs, typically, when one task requests another).

(vii) Maximum non-interruptable time - 495 microseconds. (This refers only to internal RTEX operations, and not to user tasks).

(Note: The figures quoted above refer to RTEX being executed in a Varian 620I computer, with a cycle-time of 1,8 microseconds).

J.2.2 Analysis of Measurements.

It is of interest to attempt to analyse overall processor efficiency as measured in this configuration. The calculation of the Utilization Factor (UF), as discussed in Appendix H, will be affected by two considerations: firstly, the longer task-swop-times and, secondly, the actual time wasted in dealing with interrupts which do not result in any change of task. These are two areas in which the hardware-based operating system is seen to be of great advantage, particularly as in the latter case (the handling of interrupts which do not immediately require task-swopping), the microcontroller does not interfere with the task currently being executed.

Analysing the third experimental situation in a manner similar to that described in Appendix H.3, the following pattern emerges:

The tasks' Run-Times (C_n) and Request Periods (T_n) will not alter. Therefore, the processor utilization (UF) for a situation in which Task 4 (τ_4) is omitted will be 0,0687.

Now, to evaluate the overall system utilization, Task 4 must be considered.

As assumed in H.3, τ_4 will be executed between each execution of Tasks 1 to 3.

So, considering a 2-second interval, the total time to execute Tasks 1 to 3 will be given by:

137,4 mSec (Time to run Tasks) + Time to Effect Swops.

The time to effect the swops will be given by:

$$\left(\frac{2000}{20}\right) \times \left(\text{Swop-time for } \tau_1\right) + \left(\frac{2000}{1}\right) \times \left(\text{Swop-time for } \tau_3\right) + \left(\frac{2000}{2000}\right) \times \left(\text{Swop-time for } \tau_2\right)$$

$$= \left(\frac{2000}{20} + \frac{2000}{1}\right) \times 408{,}6 + \left(\frac{2000}{2000}\right) \times 471{,}6 \ \mu\text{Sec.}$$

= 858 mSec.

(Swop-times are given in J.2.1)

∴ Total time, within the 2 second interval, which is consumed by τ_1 to τ_3,

= (858 + 137,4) mSec

= 995 mSec.

Assuming τ_4 goes into execution 2101 times, (i.e. between the execution of the other tasks),

Time taken to swop-in τ_4 = (2101 x 471,6) μSec.

= 991 mSec.

Thus, the time available to execute τ_4 is given by

(2000 - 995 - 991) mSec.

= 14 mSec.

∴ Run-time $C_4 = \frac{14}{2101} = 6{,}6$ μSec.

The request period, $T_4 = \frac{2000}{2101} = 0{,}952$ mSec.

∴ Utilization Factor for $\tau_4 = \frac{0{,}0066}{0{,}952} = 0{,}007$.

∴ Total system Utilization Factor, $UF_{Total} = 0{,}0687 + 0{,}007$

= 0,0757.

This implies that the processor is only being used for the actual execution of tasks for 7,6% of available time.

This figure will be further degraded by the time wasted by RTEX in having to deal with incoming interrupts which cannot immediately be handled.

J.3 Conclusion.

This appendix has described the application of a totally-software-implemented Real-Time Executive Operating System in a situation identical to that in which the hardware-based system was applied. The efficiency obtained in each case, in terms of the Utilization Factor, demonstrates vividly the advantages which may be gained by relegating the operating system to a hardware unit. In quantitative terms, a higher swopping-rate may be achieved, but of greater significance is the processor-time gained when the microcontroller makes the decision as to whether the incoming interrupt must result in a rescheduling of user tasks, or not. This is illustrated by the fact that every interrupt occurrence requires at least 275 microseconds of processor time to respond to it, regardless of whether a task-swop must be effected, or not.

APPENDIX K: THE OVERALL SPECIFICATIONS OF THE MICROCONTROLLER.

It should be noted that the following specifications are applicable only to the prototype microcontroller system. The figures quoted for microcycle time, system clock frequency, input/output delays, etc. may be improved by the use of faster elements in the microinstruction memory. The maximum cycle time which may be achieved is approximately 150 nanoseconds.

All times stated are accurate to ±5%.

Overall microcycle time	- 360 nanoseconds.
Microinstruction word length	- 24 bits.
Data word length	- 16 bits.
Access-time of local memory	- 50 nanoseconds (read) - 45 nanoseconds (write)
Local memory structure	- 256 x 16-bit (Random Access Memory).
Maximum allowable local memory-capacity	- 32768 x 16-bit.
Microinstruction memory access-time	- 280 nanoseconds (read).
Microinstruction memory structure	- 1024 x 24 bits (Random Access Memory).
Maximum microinstruction memory size.	- 4 blocks of 512 x 24 bits.
Number of distinct central processor array functions	- 32.
Number of microsequencer control functions	- 6 and 8 flag control functions.
Number of interrupt lines	- 24 (fully-vectored).
Maximum number of interrupt lines	- Unlimited (Increments of 8).
System clock frequency	- 5,55 MHz.
Maximum delay occurring on write operations to I/O device or local memory	- 0,5 x system clock period (assuming device is ready).
Maximum delay occurring on read operation from I/O device or local memory	- 0,5 x system clock period (assuming device is ready).

Power supply requirements	- 7,1 Amperes at 5 Volts.
Cooling required	- Forced ventilation.
Physical construction	- 2 x (19,5 x 31) cms. double-sided printed-circuit boards, wire-wrapped.
Total number of Integrated Circuits	- 173
Type of Logic used	- Schottky-clamped transistor-transistor-logic.

APPENDIX L: A STATISTICAL ANALYSIS OF THE TRAFFIC IN A MULTIPROGRAMMED COMPUTER.

The statistical analysis set out in this appendix is based on standard communication traffic switching theory, and on classical probability theory {103}. Of particular relevance are the theories developed by the authors of References {104} and {105}.

The important characteristics of a multiprogrammed computer traffic study may be stated as follows:

(i) The inputs to the system. These are, in effect, the requests which are directed at the computer from various sources, as described in Chapter I. Telephone switching theory would term this the "pattern of arrival of calls".

(ii) The queue lengths and queue delays which will result because only one request may be dealt with at a time. This situation is akin to a telephone switching network that can "store and forward" calls, having a single server (processor).

L.1 The Input Pattern.

The underlying premise is that individual requests arrive in a random manner at an average arrival rate which is constant.

Consider a time interval T. Let this interval be divided into (n + 1) sub-intervals, each of duration t_o, assuming that there is a finite probability of one request occurring during each sub-interval t_o, and that this probability is the same for each sub-interval. (Provided that t_o is sufficiently small, there will be virtually no probability of more than one request occurring during t_o.)

If the average arrival rate is K, then Kt_o will be the probability, p, of the request occurring during t_o.

So the probability of m requests occurring during T is $(Kt_o)^m$.

The probability of the remaining (n - m) sub-intervals remaining empty during this period is $(1 - Kt_o)^{n-m}$.

Assuming that the requests are independent of each other, then for a situation where m sub-intervals have a request occurring within them, the probability will be $(Kt_o)^m(1 - Kt_o)^{n-m}$.

There will be $\frac{n!}{(n-m)!m!}$ possible combinations of m sub-intervals which have requests occurring within them, so that the probability distribution is

$$P(m) = \frac{n!}{(n-m)!m!} (Kt_o)^m(1-Kt_o)^{n-m} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (1)$$

The cumulative probability distribution is

$$(P(m \geq c) = \sum_{m=c}^{m=n} \frac{n!}{(n-m)!m!} (Kt_o)^m (1-Kt_o)^{n-m} \quad \ldots\ldots\ldots\ldots (2)$$

where c is the lower bound on the summation.

Setting $Kt_o = p$ (the probability of the request occurring during the period T), and letting $a = pn$, equations (1) and (2) may be simplified if it is assumed that p is small and n is large. This yields

$$P(m) = \frac{a^m e^{-a}}{m!} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (3)$$

$$P(c,a) = \sum_{m=c}^{\infty} \frac{a^m e^{-a}}{m!} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (4)$$

It may be observed that these are, in fact, the Poisson distributions for probability density and cumulative probability.

Equations (3) and (4) may be used to estimate the intervals between requests.

$$a = pn = (Kt_o)n = KT$$

$$\therefore P(m) = \frac{(KT)^m e^{-KT}}{m!}$$

The probability of no requests occurring within the interval from t = 0 to t = T will be

$$P(0) = e^{-KT}$$

This is also the probability P ($t \geq T$) of the time interval between requests equal to or greater than T, or in the general case for time t,

$$P(t) = e^{-KT} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (5)$$

The probability distribution of time intervals between requests may be determined by evaluating the probability that the instantaneous request time lies between t and $(t + \Delta t)$, as $\Delta t \to 0$., i.e.

$$\lim_{\Delta t \to 0} \frac{P(t)-P(t+\Delta t)}{\Delta t} = \frac{dP(t)}{dt} = \frac{d(e^{-KT})}{dt}$$

$$= Ke^{-KT} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (6)$$

The probability of one or more requests arriving in the intervals, for t = 0 to t = T, is 1-P(0), i.e. $1-e^{-KT}$.

If T is assumed to be small, Δt, then the probability of one or more requests occurring, approaches the probability of exactly one request

occurring, i.e. $1-e^{-K\Delta t}$.

Expanding this expression, and neglecting terms in $(\Delta t)^n$ where $n \geq 2$, it becomes $1-(1-K\Delta t) = K\Delta t$.....(7)

This, then, indicates that the overall demands made on the system by a large number of independent requests may be expressed by a Poisson distribution, and that the probability distribution of intervals between requests has a negative exponential form. The probability of the arrival of one request is $K\Delta T$, which is a function of the average rate of requests, and is independent of all previous requests.

L.2 Characterising the Tasks Requested.

Each request received by a system will require, ultimately, the execution of a task. In order to evaluate the total time taken to execute the tasks within a specified interval of time, it is necessary to characterise task lengths, in terms of time.

Using statistical data accumulated, relevant to switching systems {104}, a distribution of task lengths may be proposed as being $f(t) = k_1 e^{-k_2 t}$.

In such situations, it has been shown that this may be approximated to $\frac{1}{h} e^{-\frac{1}{h}t}$, where h is the mean call length. Such a distribution is acceptable in the case of task lengths, in which a similar situation arises.

Using this as a starting-point, the probability of a task lying between t_o and $(t_o + \Delta t)$ (in time units) will be given by

$$\frac{1}{h} e^{-t_o/h} \Delta t \quad \text{.....(8)}$$

The cumulative probability of a task being longer is, then ,

$$P(t>t_o) = e^{-t_o/h} \quad \text{.....(9)}$$

Note that the probability distribution of task length is similar to that derived for the inter-arrival intervals of requests. It also indicates that the time to terminate a task from a point of initiation is independent of elapsed time, t, to that point.

Thus, an estimation of the probability of exactly one task terminating in a time interval, t_o, is given by

1-(the probability of a task being longer than t_o),

and if t_o is made small, i.e. Δt, then this is $1-e^{-\Delta t/h}$ (from (9)).

Expanding this, and neglecting terms in $(\Delta t)^n$ where $n \geq 2$, this yields

$$\frac{1}{h}\,\Delta t \qquad \text{.....(10)}$$

(note, independent of all previous occurrences.)

L.3 Queue Lengths and Queue Delays.

Using the distributions obtained above for task lengths and task requests, estimates of the characteristics of the resulting queues may be made.

The assumption is that there are random requests made for tasks which have random execution times. The requests arrive with a mean arrival rate of K. The tasks have a mean duration of h, and a negative exponential distribution. It is assumed that the tasks in queue and in execution are handled by a single processor. Thus, their termination rate is $(\frac{1}{h})\,\Delta t$ (from (10)), and the arrival rate is $K\Delta t$ (from (7)). At any time t, the system has probability $P_n(t)$, of n simultaneous tasks in queue and in process. The system can get to this present state, $P_n(t)$ from the state just preceeding it, $P_n(t-\Delta t)$ in one of the following ways (Note that in the following, P_{n-1} and P_{n+1} signify system states with one less or one more task than the number in queue at state P_n):

(i) A new task is requested. The task in execution continues in execution, in the state P_n. The probability of this transition from $P_{n-1}(t-\Delta t)$ to $P_n(t)$ is
$P_{n-1}(t-\Delta t)K.\Delta t.(1-\frac{1}{h}\,\Delta t)$.

(ii) The task currently in execution terminates, and no new request occurs.
The probability of transition from $P_{n+1}(t-\Delta t)$ to $P_n(t)$ is
$P_{n+1}(t-\Delta t)\,\frac{1}{h}\,\Delta t\,(1-K\Delta t)$.

(iii) There is neither a task termination nor a new request. The probability that this occurs is
$P_n(t-\Delta t)(1-\frac{1}{h}\Delta t)(1-K\Delta t)$

(vi) A request occurs and the task in execution terminates. The probability that
$P_n(t-\Delta t)$ goes to $P_n(t)$ is
$P_n(t-\Delta t)\frac{1}{h}\Delta t K\Delta t$

Thus, the probability of the actual existence of the state $P_n(t)$ will be the product of the individual probabilities established in (i) to (iv) above, i.e.

$$P_n(t) = P_{n-1}\ (t-\Delta t)K\Delta t + P_{n+1}\ (t-\Delta t)\frac{1}{h}\Delta t + P_n\ (t-\Delta t)\{1-(K + \frac{1}{h})\Delta t\}$$

Rearranging, and omitting terms in $(\Delta t)^n$ when $n \geq 2$,

$$\frac{P_n(t) - P_n\ (t-\Delta t)}{\Delta t} = P_{n-1}\ (t-\Delta t)\ K + P_{n+1}(t-\Delta t)\frac{1}{h} - P_n\ (t-\Delta t)(K + \frac{1}{h})$$

Now let $\Delta t \to 0$.

$$\therefore \frac{d\ P_n}{dt} = K\ P_{n-1} + \frac{1}{h}\ P_{n+1} - (K + \frac{1}{h})\ P_n.$$

If a state of statistical equilibrium is assumed, then $\frac{dP_n}{dt} = 0$ (i.e., the system is statistically stable at this point).

$$\therefore (K + \frac{1}{h})P_n = KP_{n-1} + \frac{1}{h}\ P_{n+1} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (11)$$

Equation (11) represents a set of equations in n, for $n \geq 0$. These may be solved by successive substitution using the condition $\sum P_n = 1$.

It has been shown {111} that the solution for constant parameters K and h provides an expression for the probability distribution of queue length. (Again, this has been achieved using telephone traffic switching theory.) This is

$$P_n = (1-Kh)(Kh)^n \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (12)$$

The mean of (12) will be

$$L = \frac{Kh}{1-Kh}\text{, which is the mean queue length} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (13)$$

The distribution of the delay experienced by tasks in the queue may also be derived, since this is the probability of (n tasks in queue) times (the sum of n exponential lengths which precede the (n+1) requests).

The sum of n independent exponents is found by convolution. The resulting distribution is given by

$$P_D = (\frac{1}{h} - K)\ e^{-(\frac{1}{h} - K)t}$$

The mean of this, which is the mean delay experienced in queue is

$$D = \frac{h}{1-Kh} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (14)$$

Note that this may also be verified directly from (13), since the requests which arrive during the mean delay of KD are equal to the mean queue length L, i.e.

$$L = KD$$

$$\therefore D = \frac{L}{K} = \frac{Kh}{K(1-Kh)} = \frac{h}{1-Kh}.$$