MULTIPROGRAMMING FOR A SMALL-SCALE SCIENTIFIC COMPUTER SYSTEM.

J.P. Penny, M.Sc.

A Thesis submitted to the Department of Mathematics
in the University of Adelaide
for the Degree of Doctor of Philosophy.

May, 1965.

TABLE OF CONTENTS

Page

APPENDICES

SUMMARY

Use of the technique of multiprogramming has hitherto been confined entirely to the larger computer systems. In this thesis, the author puts forward the proposition that multiprogramming can be used to considerable advantage in a quite small computer system, particularly where the computer is used for scientific work.

The factors which have attracted designers of large computers to multiprogramming are considered, and it is shown that these factors apply also for smaller systems. To develop a case for multiprogramming, the author investigates a theoretical model of a small multiprogram computer, and discusses results from a computer simulation study of an equivalent model.

Much of the material in the thesis deals with a multiprogram system developed by the author for a low-cost computer. This computer, CIRRUS, has been designed and constructed within the University of Adelaide. The author's principal aim in developing the multiprogram system has been that the computer should be made to behave as "a set of separate and independent computers".

The CIRRUS system is covered in detail. The methods by which multiprogram control functions have been

provided and measures taken to ensure that these functions could be provided economically are described. The author discusses the conditions which it was felt should be fulfilled, and steps taken to satisfy these conditions. A number of features of the system which might be considered novel are also described.

The question of efficient processor utilisation in a multi-operator computer system is discussed. Results from the computer simulation study are also used in this context to justify arguments which the author puts forward.

PREFATORY STATEMENT

The author presented, in 1960, the results of a preliminary study of multiprogramming as a thesis for the degree of Master of Science in the University of Adelaide. Some of the work described in this thesis has followed on from the earlier work, and appropriate reference has been made in the text. However, the material in the present thesis has not been submitted for a degree in this or any other university.

The development of ideas in the field of computing science takes place at a more rapid rate than in most other fields of research. It is important, therefore, to point out that the proposals for the CIRRUS multiprogram system (which constitutes a substantial part of this thesis) were almost completely developed by the end of 1962. (See: Penny & Pearcey, 1962; Penny, 1963; Penny et al., 1963). The CIRRUS processor first operated on a restricted basis (with a 2000-word store) during 1962. Time-sharing between independent programs (in the manner described in Section 7 of the text) was in fact demonstrated with borrowed teleprinter equipment at this time. Subsequently, the CIRRUS project encountered financial difficulties, and it was not until mid-1964 that sufficient store and peripheral equipment were available to implement the full multiprogram system.

The thesis discusses the system in actual operation at September 1964. Appendix B gives a detailed specification for this system. Two procedures described in the text have, so far, not been implemented. Section 8.5. deals with a procedure for allocation of peripheral units beyond the basic sets. No such peripheral units are currently available. The "within-program" time-sharing procedure (Section 10) has also not been fully implemented, although allowance has been made for this mode of operation in much of the existing software.

The author acknowledges the cooperation of a number of persons associated with the CIRRUS project. (See Acknowledgements on the following page). However, the material in this thesis is, to the best of the author's knowledge, wholly the work of the author, except where reference has been made to another source.

J.P. Penny

May, 1965.

ACKNOWLEDGEMENTS

The author wishes to thank Professors E.A. Cornish, E.S. Barnes and E.O. Willoughby for permission to carry out the work described in this thesis.

The author also acknowledges the assistance of the following persons associated with the CIRRUS project:

Dr. M.W. Allen and Mr. R.J. Potter, who co-operated in integrating into the logical design of CIRRUS the hardware requirements for multiprogramming,

Mr. T. Pearcey, who read through the author's early proposals and made several helpful suggestions,

Mr. J.G. Sanderson, who wrote the CIRRUS compiler and made, in his own programs, several modifications to adapt to the needs of multiprogramming.

1. INTRODUCTION

Multiprogramming is a technique which allows concurrent operation of processes from a number of separate programs within a single-processor computer system. For a number of reasons, use of multiprogramming has been confined almost exclusively to the larger computer systems. The aim of this thesis will be to show that multiprogramming can also yield considerable advantages in a small-scale computer system, particularly where the system is used for scientific work.

The author will discuss multiprogramming generally in Section 2. In Section 3, he will show that the factors which have attracted the designers of large systems to multiprogramming apply also for small systems.

The case for multiprogramming with small systems will be continued in Section 4, where it will be shown that a very considerable improvement in processing capacity results when time is shared between as few as two or three programs. A theoretical model of a small multiprogram computer will be investigated and upper and lower limits found for the "improvement factor", a quantitative measure of the increase in processing capacity resulting from multiprogram time-sharing. Results from a computer

simulation study of an equivalent model will also be quoted. The case for multiprogramming will be concluded with some preliminary remarks on feasibility.

In Section 5, the author will propose design objectives for a small, low-cost multiprogram computer. Where the system is to be used for scientific computation, separate operating facilities for on-line working are considered essential. However, since the peripheral units used in a low-cost system are likely to be slow, considerable attention should also be given to achieving reasonably efficient time-sharing of input and output. The question of choosing a low-cost structure which will fulfil these objectives will also be discussed.

The author has developed and brought into operation a multiprogram system for a digital computer, CIRRUS, whose total system cost has been kept quite low. In his earliest statement of design objectives for the CIRRUS multiprogram system, the author (Penny, 1960) suggested that the computer should be made to behave as a set of "separate and independent computers". Each operating station of CIRRUS is equipped not only with an operating keyboard and typewriter but also with paper-tape input and output units. These stations are therefore more comprehensive than the "enquiry stations" to be found in other multi-operator computer systems. There are considerable advantages for

the on-line user in having an input and an output unit entirely to himself, rather than sharing them with other users.

Where there are to be many separate operating stations in a large system, designers invariably make use of a large backing store. Although he recognizes the value of a large backing store, the author considers that an efficient system using two levels of store cannot be implemented cheaply. If the total system cost must be low, a large core store is probably the better investment.

CIRRUS has a 32,768 word core store - a quite large store for a "small" system. The recent marked decrease in the cost of moderate performance (in this case 6 μs cycle-time) core store makes it economical to include stores of this size in a low-cost system. Efficient use of this storage is nevertheless essential. The CIRRUS system allows for "dynamic relocation" of partially executed programs. All vacant store-space can therefore be made available for any incoming program which needs it. A special technique allows sequences of stored instructions (for example, those instructions constituting the compiler or input-output routines) to be shared simultaneously by independent programs. This technique also contributes towards more efficient use of the available store-space.

Much of the discussion in this thesis will be

centred on the multiprogram system developed for CIRRUS. An introductory description of the system will be given in Section 6. For discussion, the additional functions needed to achieve multiprogram operation have been divided into three categories:

(1) Control of time-sharing,

(2) Control of space-sharing, where "space" can mean either storage or peripheral units,

(3) Provision of facilities for each of a number of operators to control his own program.

These topics will be covered in Sections 7, 8 and 9 respectively.

In any multiprogram computer, the needed control functions will be provided partly by hardware and partly by software. In a low-cost system, software must be preferred wherever possible. There is, however, the attendant problem that the quantity of systems program used must be kept reasonably low. It has been found that, apart from additional storage and peripheral units, the quantity of special hardware needed to achieve multiprogram operation is quite trivial. The amount of software used in CIRRUS has also been fairly small. The author will describe measures taken to ensure that the control functions could be provided simply, and therefore economically in both hardware and software.

The CIRRUS system also provides a facility for "within-program" time-sharing. It will be shown in Section 10 that the user may divide a single external program into a number of separate internal programs. These internal programs, though sharing time independently, may still communicate with one another.

During the development of the multiprogram system, certain requirements have been borne in mind. The most important requirement is that the user, either as programmer or operator, should not suffer from the additional complexity of internal operation.

The independence of operating stations in CIRRUS makes an important contribution towards the convenience of the user. The designers of some multiprogram computers have considered it essential that jobs be scheduled for execution in appropriate combinations. If the operating stations are independent, scheduling is impossible. However, the author feels that the variable nature of the work-load of a small scientific computer makes an effective scheduling technique difficult to implement in any case.

The question of efficient processor utilization in this type of multiprogram computer will be discussed in detail in Section 11. It will be shown that a great deal can be done to improve efficiency without prejudicing the

independence of the on-line users. Results from the simulation study will be quoted in this context.

The successful operation of the CIRRUS multiprogram system has shown that, without doubt, multiprogramming is economically feasible in a low-cost system. Since multiprogram operation offers considerable advantages, the author will conclude that provision for multiprogramming should be considered during the designing of any small-scale computer system.

2. TIME-SHARING AND MULTIPROGRAMMING.

2.1. Definitions.

We can regard any digital computer system as consisting of an interconnected set of separate facilities. If we examine the progress of any single program through all its phases, we can see that certain facilities are required virtually continuously, while some are required intermittently and others not at all. It has long been evident that efficiency can be improved by making intermittently used facilities share their time between processes which are to some degree independent.

The term "time-sharing" has itself been used with a variety of different meanings. In the Computer Handbook (Huskey and Korn, 1962) it is mentioned only with respect to multiplexing of analogue elements. "Time-sharing" is also sometimes used to describe the mode of operation of equipment which is actually multi-purpose. For example, the registers and logic making up the working-sections of CIRRUS are said to be "time-shared" (Allen and Rose, 1963; Allen et al. 1963) in that they are "assigned various roles as the control unit progresses through the steps necessary to execute a given machine function."

However, "time-sharing" is more commonly used to

refer to the particular type of operation described by Beckman *et al.* (1961): "When units which work together are widely disparate in speed, so that the faster must often be idle while waiting for the slower, efficiency can be increased by time-sharing the faster unit among several slower ones, which operate concurrently. Time-sharing and concurrency are thus the two sides of the same coin." Since the main objective is greater utilisation of the faster unit, time-sharing is most appropriate where the time-shared unit is appreciably more expensive, as well as being faster. If this is so to a marked degree, it may well be worthwhile replicating more or less identical slower and cheaper units to allow an even higher degree of concurrency and utilisation of the faster unit.

Concurrency of operation is a fundamental part of any definition of time-sharing in this context. Bucholz (ed., 1961) defines two forms of concurrency:

(1) "local concurrency", the "overlapping of the execution of an instruction with that of one or more of its neighbours", and,

(2) "non-local concurrency, the simultaneous execution of instructions which may belong to entirely separate and unrelated programs."

The concept of non-local concurrency leads directly

to the idea of "multiprogramming". Bright and Cheydleur (1962) have defined multiprogramming as "concurrency of the execution phase of two or more programs". Although on the whole an apt description, the choice of the word "execution" is unfortunate. Program "execution" usually implies only what takes place following program assembly, and the idea of multiprogramming certainly does not exclude the possibility of concurrent assembly of separate programs, or of the assembly of a program concurrently with the execution of others.

The technique of multiprogramming is used in the Honeywell 800 (Harper, 1960), but here it is described as "parallel programming". Gill (1959) uses the same term. Modes of operation covered by Bright and Cheydleur's definition are often described simply as "time-sharing", for example in the Ferranti computers: ATLAS (Kilburn et al., 1961); ORION (Ferranti, 1960); FP6000 (Marcotty et al., 1963); the SABRAC (Lehmann et al., 1963); and the LEO III (Lewis, 1963). "Time-sharing" is also used by Strachey (1959) and Corbato (1962).

The author has preferred "multiprogramming" (Penny, 1960, 1963; Penny and Pearcey, 1962; Allen et al., 1963). The designers of STRETCH (Codd et al., 1959; Bucholz, 1961) use the same term, as do Lourie et al. (1960) and Landes et al. (1962). The author justifies his preference by the arguments which follow.

In the terminology of computer design, the concept of "parallel" operation is well established. It is generally taken to imply simultaneous processing (in the most general sense) by more or less equivalent hardware units. Concurrency of operation is not the only fundamental factor in any definition of the type of operation under discussion; sharing of a unit is equally fundamental. Use of the word "parallel" is therefore inappropriate.

Apart from its indiscriminate use in other contexts, "time-sharing" was for some years understood to mean concurrent operation of processes which, though to some degree separate, were parts of a single program. What will hereafter be referred to as "single-program time-sharing" has been possible to some extent with virtually every computer built in recent years. The "concurrent processes" are the actual computation and one or more data transfers to or from the computer, all processes sharing the time of the single central processor.

A quite direct development from single- to multiprogram time-sharing may be seen in some systems. 'ATLAS' is a good example. For quite long periods in this computer, one program only shares time with simultaneous data transfers (made to or from a backing store) which are parts of several separate programs.

Classing operation in ATLAS as "multiprogram" may be disputed. There are certainly important differences between the mode of operation in ATLAS and "parallel" program operation in the Honeywell 800 or "multiprogram" operation in CIRRUS. In each of the latter two, several programs are at any time competing for processor time. However, the designers of all three systems have had a common aim: to take advantage of the fact that time-shared operation becomes many times more effective when concurrency of processes from multiple programs, rather than only a single program, is possible.

However, the definition of multiprogramming given by Bright and Cheydleur, as it stands, would also include the mode of operation of such computers as the RW-400 (Datamation, 1960) and the Burroughs D-285 (Anderson et al., 1962) where the computer may have more than one separate processor. The operation of such machines has been termed "multiprocessing" by several authors (e.g. Nekora, 1961; Amdahl, 1962; Critchlow, 1963).

Critchlow distinguishes between the two. He defines multiprogramming as the "time-sharing of a processor by many programs operating sequentially," and multiprocessing as "independent and simultaneous processing accomplished by the use of several duplicate hardware units." In the

former "sequentially" must refer only to actual processor use. To be classed as multiprogramming, some concurrency of separate programs is necessary.

According to Bucholz (1961), a STRETCH system might have one or more central processors, any of which might be multiprogrammed. The Control Data 6600 (Control Data, 1963) contains "eleven independent computers." However, although all have separate memories and can execute programs independently, 10 are used to control peripherals or to communicate with operators. The eleventh "computer", called the central processor, is a very high-speed arithmetic device. There is also a large central memory available to all 11 computers.

The author feels that some multiprocessor systems (and the Control Data 6600 is one of them) have much in common with a small multiprogram computer such as CIRRUS. CIRRUS has a single processor and a number of sets of peripheral equipment. As in the 6600, the primary reason for multiprogramming CIRRUS was a desire to utilize more efficiently the main body of hardware by time-sharing, and the chosen solution required replication of simpler hardware units. Greater work capacity for a given total system cost has been the aim in designing other multiprocessor systems. Gamma 60 (Dreyfus, 1958) which has a number of fairly specialised processors, is one example.

However, the objectives in multiprocessing or multiprogramming are often fundamentally different. A computer system may have several more or less equivalent processors for greater reliability in a special situation (e.g. Anderson et al., 1962) or for vast work output where use of independent computers is undesirable. Often, the most important objective in multiprogramming is to apply the computer to activities which, though individually too inefficient to contemplate, become feasible if time is shared between them.

In this thesis, we shall be specifically interested in small-scale systems. A system is unlikely to be "small" if there is more than one processor. Hence, multiprocessing will be ignored except for later mention of a case where a small multiprogram computer could profitably be used as part of a larger multiprocessor system.

2.2 A Summary of the Purposes of Multiprogramming.

The advantages to be gained by multiprogramming any computer system fall into the two broad categories of:

(1) Greater efficiency,

and (2) Greater flexibility.

From the description of time-sharing given by Beckman et al. (1961) quoted in Section 2.1, it is clear

that greater efficiency is brought about by more fully utilising the time of any unit which is time-shared. Greater flexibility results because, with many slower units able to time-share a faster unit, units can be used which would otherwise be too slow. The slower "units" mentioned are not necessarily peripheral units. The human operator, for example, is an essential unit of any computer system, and a "unit" whose "operation" is certainly slow. Nor need the units be only those which would at present be considered usual elements of a computer system.

In the conditions most commonly associated with multiprogramming, the fast, time-shared unit is the central processor and the slower units are standard units of peripheral equipment. Transfers of data to or from the computer require mechanical movement of the input or output medium through the reading or writing unit. The speed of data transfer is therefore limited in comparison with the speed of the electronic circuitry of the central processor.

Magnetic tape is far superior to other widely used media for speed of data transfer. Systems processing large quantities of data use magnetic tape almost exclusively for input and output. However, use of magnetic tape has definite disadvantages. Manual data preparation on cards or paper-tape requires only very simple equipment.

No equivalent equipment is available for magnetic tape data preparation. Occasionally, there may be sufficient data from a single source to justify the cost of special equipment recording directly on magnetic tape. Even so, equipment must always be available to transfer data off-line from cards or paper-tape to magnetic tape, or from magnetic tape to printed output. Frequently, a small computer with little computing power but considerable capacity for handling peripheral equipment is used as a data converter for a larger machine. A common example is the association of an IBM 1401 with an IBM 7090.

Even with all input and output being made from or to magnetic tape, the time of a very fast computer would be inefficiently used if no time-sharing of the central processor were possible. The fastest central processor could handle perhaps 50 simultaneous tape transfers if the necessary equipment were available. Nearly all recent computers allow peripheral units to operate simultaneously with computation. However, facilities to make this concurrent operation possible frequently involve substantial expense. If peripheral equipment in its simplest form is considered, it will be seen that the individual units of central processor time not required during data transfers, though adding up to a high proportion of total time, are each very small. To consolidate the many small periods of

unused time into more convenient units, computer designers have often included large-scale buffering and extra-computer control for peripheral equipment. In particular, elaborate and costly magnetic tape controllers have resulted.

It is doubtful whether the improvement in work done every really justified the additional cost. Time-sharing between processes from only a single program is limited in the advantages to be gained because the number of processes available to time-share must be very small. It is often difficult to organise a program to take advantage of the time-sharing capability. Theoretical calculations made by Boyell (1960) for a particular case of concurrent file searching and computation led him to conclude that an "improvement factor only in the order of 1.1 or 1.2" was likely.

The development of very complex magnetic tape systems preceded and gave impetus to the consideration of multiprogramming. Where there are several programs, it is obvious that there will be many more processes to time-share the central processor simultaneously. As the processes come from separate programs, there is no need to rearrange any program to make time-sharing possible. A multiprogram computer must have the ability to switch between programs. If this can be rapid enough, the need for extensive buffering and special control hardware is removed.

Programs may either control their own peripherals, or special-purpose programs (often at a lower program level) may be used specifically to handle data transfers. This has been well described by Strachey (1959). It is obvious, therefore, that multiprogramming offers a substantial improvement in the efficiency of input and output over single program time-sharing and that the cost of time-sharing hardware can in fact be reduced.

With a multiprogram machine, the designer can consider using slower units of any given type. Often, increased peripheral speed is achieved only at substantially higher cost and quite possibly with significantly slower reliability. Also, units of types necessarily very slow in operation (such as plotters) may be coupled on-line to the computer. Use of off-line converters is unnecessary.

Single program time-sharing has not been neglected while multiprogramming has been developed. Today, "interrupt subroutines" are widely used. These make the central processor and part of its own storage take over many of the functions of the peripheral unit controllers. Hardware cost is reduced, and the systems programmer at least has more direct control over time-sharing. However, the difficulty of making any significant number of processes function concurrently still remains.

It is logical to develop the use of interrupt

subroutines further by adding fast auxiliary storage, preferably a drum, as an input - output reservoir. Many peripheral units can then operate concurrently, feeding data for a number of problems to or from the drum. As far as computation by the central processor is concerned, the drum would be the only peripheral unit, and a high degree of central processor utilisation is likely. Development of the ATLAS system was based on this idea.

However, multiprogramming does not give only more efficient time-sharing of data transfers. The best run computer will, on occasions, be idle because of a delay by the human operator. Where there is multiprogram operation, most of what the operator must do for any given program can be done without stopping useful work by the computer on other programs. However, really significant benefits from multiprogramming are obtained when several separate operators are provided for.

According to Beckman (1961) the greatest promise of multiprogramming lies in "permitting closer collaboration of user and machine". Where there are a relatively large number of operators, or a few with at least one or two keeping the computer consistently supplied with work, it becomes reasonable for a user to work on-line with his program even if he makes only infrequent demands for central processor time. Several authors (e.g. Strachey, 1959; Corbato, 1962) have stressed the value of on-line working

in activities such as program debugging. But the on-line activities which a multi-operator system makes economically possible range far beyond the present area of computing. Licklider and Clark (1962) say that man-machine communication has been greatly impeded hitherto by the "economic factor" and they suggest that multiprogram time-sharing can do a great deal to overcome this barrier. They give many illustrations of what could be worth doing with a comprehensive system. Their point of view is shared by others. In a proposal for a research and development program for the Massachusetts Institute of Technology, Fano (1963) stresses the desirability of an "on-line mode of operation, where the individual scientist, problem solver or decision maker is tightly coupled with a computer system."

If the operating consoles are truly independent, they can be placed in many different locations, some of which may be quite remote from the computer itself. Consoles need not be identical. Some may even be tailor-made for special applications.

The activities made economically feasible by multiprogramming are not only those whose low-demand is caused by a human operator. The flexibility of the multiprogram computer makes it valuable in, for example, the field of automatic control. The use of high-speed computers in this field is still to be fully exploited. The economic

factor is again partly responsible. Equipment under computer control may require attention only rarely, but that attention may be necessary with a minimum of delay. However, a control program can be one of several programs sharing the time of a general-purpose multiprogram computer. Time need be given to it only when required.

Most authors quoted here have discussed multiprogramming with only large systems in mind. (Corbato (1962), for example, has suggested possibly several hundred consoles with large core memories of as many as a million words.) The present author believes that there are three main reasons for the apparent neglect of the smaller computer:

(i) The initial strong incentive to find a better approach to the problem of time-sharing input and output arose from the cost and limitations of the elaborate magnetic tape controllers used with large single-program machines.

(ii) The protagonists of closer man-machine contact were primarily attracted by the possibilities of having on-line the facilities and power of a very large computer.

(iii) The large machine of course seems more readily adaptable to multiprogram operation. It would more often have a considerable part of its storage not in use by a single program.

Multiprogram operation implies a need for additional hardware. The cost of such hardware would be relatively less in a large system.

However, in the following section, we shall discuss the extent to which the arguments for using multiprogramming apply to smaller systems. Later sections will deal with the feasibility of multiprogramming in small systems.

CONCLUSIONS TO SECTION 2:

1. For the purposes of this thesis, the author will define multiprogramming as "a technique to allow concurrent operation of processes from a number of independent programs which time-share a common processor."

2. To obtain the fullest value, both in efficiency and flexibility, from time-shared operation, it must be possible to share time between processes which are parts of separate programs. That is to say, multiprogram, rather than single-program, time-sharing is essential.

3. A CASE FOR MULTIPROGRAMMING SMALL COMPUTER SYSTEMS.

I: The Arguments for Multiprogramming Applied to Small Systems.

"Size" in computer systems involves many variables. When making comparisons, it is necessary to have some definition of what in the present context is meant by "small" computer system. The most generally acceptable criterion is cost. Any system costing less than about £200,000* can be regarded as small. Where multiprogramming is contemplated, there must obviously also be some lower limit. To operate in multiprogram mode, a system must be able to hold two or more programs, which means that the facilities available must be significantly greater than are required for an average problem. Let the lower limit, then, be £35,000. This gives a range, admittedly arbitrary, of £35,000 - £200,000.

A computer system costing £100,000 or so, though small, represents a substantial capital investment. Indeed, the figures quoted in Table 3.1. show that almost half the present total investment in computing is tied up in systems whose monthly rental is less than $10,000. ($10,000 per

---

*All figures in £A unless stated otherwise.
£A1 = £.8 stg. = $2.25

month is equivalent to an outright cost near the upper limit of $A200,000 defined previously.) We can hardly afford to neglect any technique which might materially improve the effectiveness of small computer systems.

The use of small computers is not likely to decline. The figures in Table 3.1. do show the smaller computers making up a lesser proportion of computers on order. However, there is a much longer delay between the placing of an order and delivery with larger systems and the proportion of larger systems on order is weighted as a result.

There is a case for considering multiprogramming if,

(1) during normal operation there would be substantial inefficiency in utilisation of the central processor, or

(2) there is scope for using the additional flexibility which multiprogramming allows.

For the moment, the question of feasibility is set aside until a more thorough examination has been made of the relevant characteristics of small systems. The author believes that utilisation of the central processor during operation of a small system can be at least as low as processor utilisation in a large system. However, the trends in computer design which have produced this inefficiency have been really evident only in the last year or two. The

TABLE 3.1: NUMBER AND COST OF COMPUTER SYSTEMS IN USE OR ON ORDER.*

	IN USE		ON ORDER	
	Number	Monthly Cost($)**	Number	Monthly Cost($)**
Small-scale systems (Monthly rental ≤$10,000)	18,950	6.1×10^7	9,118	3.6×10^7
Medium- & large-scale systems. (Monthly rental >$10,000)	2,398	7.1×10^7	1,645	5.4×10^7
Totals	21,348		10,763	

*Figures taken from the Monthly Computer Census, "Computers and Automation", October, 1964. Only computers for which a monthly rental is given have been included.

**Calculated as

$$\sum_{\text{all entries}} [(\text{average monthly rental}) \times (\text{no. of computers})]$$

flexibility given by multiprogramming can also be extremely valuable where a small system is used.

3.1. Central-processor Utilisation in Small Systems.

The first two entries of Table 3.2. (taken from Weik (1957)) give some idea of the computing systems of 1957.

TABLE 3.2. COMPARISON OF ARITHMETIC AND INPUT-OUTPUT PERFORMANCE

	Approximate Price	Storage	Arithmetic Speed	Input	Output
Small system (1957)[1]	$200,000	4K (drum) Possibly small delay line store	Add: 500μs (exc. access) 3ms (inc. access) Mult: 5ms (exc. access) 15ms (inc. access)	8-400 ch.p.s.* (paper tape) 50-200 c.p.m. (cards)	8-60 ch.p.s.(paper tape) 30-100 c.p.m. (cards) 150 l.p.m.* (printer)
Large system (1957)[2]	$1,500,000 to $2,000,000	8K-32K (core, 12μs cycle time)	Add: 24μs (inc. access) Mult: 240μs (inc. access)	150 or 250 c.p.m.(cards) 15 Kc/s (mag. tape)	100 c.p.m. (cards) 15Kc/s (mag. tape)
Small system (1964)[3]	$200,000 to $300,000	4K-32K (core, 1.25μs cycle time)	Add: 2μs (24-bit) Mult:10μs (24-bit)	1200 c.p.m. (cards)	300-1000 l.p.m.(printer)
Large system (1963)[4]	$1,500,000 to $2,500,000	16K-64K (core, 1.5μs cycle time)	Add: 2μs (48-bit) Mult:7μs (48-bit)	100-200 Kc/s (mag.tape)	1000 l.p.m. (printer) 100-200Kc/s(mag.tape)
CIRRUS	£20-25,000 (Cost to university)	24K(variable core, 6μs cycle time) 8K (semi-permanent core, 1μs cycle time)	Add: 30μs (36-bit) Mult:300μs (36-bit)	500 ch.p.s. (paper tape)	100ch.p.s.(paper tape) 11ch.p.s.(typewriter)

*Rates in cards/min. or lines/min. are equivalent to characters/sec., assuming that, on average, about 60 characters/card or /line represent useful information.

[1]Composite figures taken from Weik (A Second Survey ofDomestic Electronic Digital Computing Systems, June, 1957), for all systems within the cost range $70,000-$400,000 of which three or more had been sold. Systems examined: Ferranti NV1*, Elecom 125 125 FP, Ferranti Pegasus, IBM 650, Readix, Datatron, Elecom 120 & ICR CRC 102A.

[2]Figures based on the IBM 704, typical of the newer large systems in use in 1957.

[3]Figures based on the Control Data 3200, a good example of the fast, small systems now becoming available.

[4]Figures based on the Control Data 3600, a typical large, 1963-4 system.

The figures are interesting because they come from a time of particular importance for computer development in general and for multiprogramming in particular. Systems such as the IBM 704 which bridged the gap between the first- and second-generation machines were then in use, and true second-generation machines, such as the TRANSAC (later PHILCO) S2000 were imminent. The general use of systems with characteristics like the 704 had stimulated the first serious thinking about multiprogram time-sharing. This early thinking found expression in such papers as those of Gill (1958), Strachey (1959) and McDonough (1959).

The intervening years have brought little significant change in the relationship between central-processor speeds and input-output rates for large systems. Indeed, if we take as a very simple indicator:

$$\frac{\text{core cycle time}}{\text{character transfer time with magnetic tape}}$$

this was

$$\frac{12\mu s}{67\mu s} \quad \text{for the 704 in 1957,}$$

and is

$$\frac{1.5\mu s}{6.7\mu s} \quad \text{for the CDC 3600 in 1963-64.}$$

The need for multiprogramming large systems purely on the grounds of inefficiency of input-output has probably changed very little.

The picture is very different for small systems. It is only in the last couple of years that the developments which produced the second-generation of computers have had their impact on the smaller systems. Whereas the processors of small systems listed by Weik in 1957 were fairly equal in their performance, latest tabulations of computer characteristics (e.g. Adams Associates (1964)) show a wide disparity. One thing is clear, however. It is now possible to build for a modest price central processors whose basic speeds approach those of all but the fastest large systems.

The true speed of any processor of course depends on many things other than the nominal speed of simple operations. However, it may reasonably be said that a small system can now have a central processor whose real speed is less by a factor of as little as 10 than the speed of the central processor of a system costing 5 or 10 times as much. With such speeds obtainable at quite low cost, it is hardly worthwhile to build much slower central processors. A substantial change has therefore taken place. A few years ago, it was generally accepted that the larger processors could be a hundred or more times cheaper for a given unit of computation.

The differences in cost of large and of small systems are brought about in many ways. The large computer has a more powerful instruction set, substantially more storage and a greater range of software. But most important of all,

the larger system will have a much greater number and variety of peripheral units. Each peripheral unit may be faster and therefore more expensive than its counterpart in a small system.

Small computers are often associated with larger ones, most commonly for media conversions. This application will be disregarded for the present, and discussion restricted to the more important use of small computers as separate systems in their own right.

Almost all information reaches the small computer on cards or on paper tape. Output is either punched on the same media for off-line printing by tabulator or flexowriter, or printed directly by line- or occasionally by character-printer. The expense of an off-line converter for transferring data to or from magnetic tape, however desirable because of high central processor speed, is hardly justified because the cost of the computer itself is low. For the same reason, automatic data recording, if made at all, will be made on cards or paper tape.

A magnetic tape unit or two can certainly be useful in a small system for backing storage or in file work. The latter at least is infrequent if the system is used for scientific computing. However, in assessing the efficiency of input-output in small systems, we must consider the performance of those units most commonly used. Weik's

figures, summarised in Table 3.1., show that card units, paper tape units and printers in the small systems of 1957 were little slower than today's fastest. For a given cost, performance of any type of unit would have improved by a factor of 5 at most. There is a great difference between this improvement and the improvement in central processor speeds. The figures of Table 3.3. are intended to summarise the two trends for both small and large systems. These figures suggest that peripheral unit operation is many times less efficient in the smaller systems.

The extent of the need to time-share input and output is determined by the degree to which utilization of the central processor would be reduced by periods spent waiting for information from any peripheral unit. Two factors are involved:

(1) The efficiency of operation of each peripheral unit, and

(2) The proportions of total time for which each unit is used.

For a given processor, the "efficiency of operation" of any peripheral unit in transferring a particular type of information can be defined as the proportion of time for which the processor is actually in use during operation of the unit without time-sharing. Reasonably accurate estimates of the efficiency of peripheral units chosen for CIRRUS have been calculated (Table 3.4).

TABLE 3.3 : COMPARISON OF CENTRAL PROCESSOR AND INPUT-OUTPUT PERFORMANCE USING THE SMALL SYSTEM OF 1957 AS A BASE.

	SMALL SYSTEM		LARGE SYSTEM	
	1957	1964	1957	1964
CENTRAL PROCESSOR SPEED	1	100	100	1000
INPUT-OUTPUT SPEED (ch.p.s.)	1 (300)	4 (1200)	50 (15,000)	500 (150,000)
RATIO	1	25	2	2

Prediction of the proportion of total time for which any unit will be used is more difficult. While CIRRUS was under construction, the nature of its work-load was, of course, not known. However, some useful information could be obtained from a set of programs prepared for use in a simulation study of the time-sharing method. Ten problems were chosen at random from work being done in the University of Adelaide (see Appendix C1 and Table C1). Seven of these problems were being run on an IBM 1620 (in one, two or three parts) and three on an IBM 7090. A detailed analysis was made of the problems in their original form, and conversion factors were calculated to predict the behaviour of programs to handle equivalent problems on CIRRUS. This information is summarised in Table C2.

TABLE 3.4. UTILISATION OF CENTRAL PROCESSOR TIME DURING INPUT AND OUTPUT WITH CIRRUS.

	Transmission time per character	Non-trivial characters		Trivial characters 7		CPU utilization during operation
		Conversion	Spare	Conversion	Spare	
Reading: (at 500 ch.p.s.) Decimal Data	2.0ms	1.1ms	0.9ms	0.5ms	1.5ms	.45*
Object Programs	2.0ms	1.1ms	0.9ms	-	-	.72**
Source Programs	2.0ms	0.5-35ms	Usually none	-	-	.93***
Punching: (Decimal Data at 100 ch.p.s.)	10.0ms	1.0ms	9.0ms	0.5ms	9.5ms	.08*
Printing: (Decimal Data at 11 ch.p.s.)	90.0ms	1.0ms	89.0ms	0.5ms	89.5ms	.01*

7 Spaces and case changes.

* Assuming 2 non-trivial: 1 trivial character.

** Assuming 6 characters/word, 3ms CPU time at end of word.

*** See Appendix C1.

From the information obtained as described above, estimates were found for the utilization of the CIRRUS processor by each of the 10 programs if run without time-sharing. The total time to complete the j^{th} program, say, was divided into alternate periods T_{ij}, t_{ij} where:

T_{ij} was a period of time in which the processor was used continuously, and

t_{ij} was a period following T_{ij} in which the processor was idle while waiting for a peripheral unit.

The processor utilization, or "demand" for time, by the program, was then:

$$D_j = \sum_i \frac{\sum_i T_{ij}}{(T_{ij} + t_{ij})}$$

Two values of D_j (see Table C 2) were calculated for each program:

(1) D_j^c, if the program were compiled from source language,

(2) D_j^a, if the program were assembled from binary object code.

The values of D_j can be combined to find an estimate of the utilization of processor time by the set of programs as a whole. The most suitable estimate is the weighted value

$$D = \frac{p^a \sum_j D_j^c . T_j^c \; + \; p^a \sum_j D_j^a . T_j^a}{p^c \sum_j T_j^c \; + \; p^a \sum_j T_j^a}$$

where

$T_j^c = \sum_i^c (T_{ij} + t_{ij})$, the time to complete the program, assuming compilation,

$T_j^a = \sum_i^a (T_{ij} + t_{ij})$, the time to complete the program, assuming assembly,

p^c = probability of compilation, rather than assembly,

and p^a = probability of assembly.

By checking 64 successive runs on the 1620, an estimate of the probability that a program would be compiled was found:

$$p^c = .45$$

The value

$$D = .293$$

was then calculated.

Because of the small sample size, this figure for D must be regarded with reservations. The value may be thought too low for a complete work-load. However, it should be pointed out that processor utilization by a program to invert a 25 x 25 matrix, with paper tape input at 500 ch.p.s. and output at 100 ch.p.s., was calculated to be .35 . This problem is one in which the amount of computation is obviously substantial.

Although one cannot take the value of D stated above as a highly accurate estimate of processor utilization in practice, it can be said quite emphatically that the demand for processor time <u>during actual program operation</u> on CIRRUS

would be very much below saturation because of the imbalance between processor and peripheral unit speeds.

So far, the effect of delays by an operator has not been considered. The amount of idle time between jobs or during any job will fluctuate considerably. Without a monitor to run a continuous sequence of jobs, at least a few seconds will be lost before each job. During many jobs, several minutes will be lost. The reductions in the value of D, as calculated earlier, if there are mean idle periods per job of 10, 20, 30 and 60 seconds are shown in Table 3.5. Had CIRRUS been constructed as a single-program computer, it is apparent that the greater part of its processing capacity would have gone to waste.

Returning briefly to the problem of delays during input and output, it can be seen from Table 3.4. that, neglecting program compilation, CIRRUS carries out conversions between internal and external format in from 500-1100 μs per character. Fast peripheral units are now available which can supply or accept characters at rates approaching the speed at which CIRRUS performs character conversions. However, even if these units were used, a good deal of central processor time would still be wasted. Very often, only a small part of any card or line contains useful information.

TABLE 3.5. THE EFFECT OF OPERATOR DELAYS ON CPU UTILIZATION DURING OPERATION OF THE PROGRAMS IN THE SAMPLE.

Mean idle time per job. (secs.)	Utilisation of CPU	
	CIRRUS	1964 System (see text)
0	.293	.247
10	.274	.151
20	.258	.107
30	.243	.085
60	.207	.051

Further, the cost of the faster units is considerable and complicated peripheral controllers may be necessary. Substituting such units for the simple paper tape units used in the CIRRUS system would have almost doubled the total system cost. Furthermore, it must be remembered that the CIRRUS circuitry was fully developed by 1962, and that the peripheral units mentioned are among the fastest available in 1964.

The processors in today's small scientific systems are many times faster than that of CIRRUS. To examine the question of input-output efficiency for systems currently coming into use, let us consider a system with:

(1) A processor 10 times faster than the processor in the CIRRUS system,

(2) A 1200 c.p.m. card reader as the input unit,

and (3) A 1000 l.p.m. printer as the output unit.

The performance of this hypothetical system would be roughly equivalent to the performance of a Control Data 3200 equipped with the fast peripheral units mentioned earlier. The degree of central processor utilization in this system was calculated (Column 3 of Table 3.5) for the work-load considered earlier.

The figures of Table 3.5. suggest that the efficiency of input-output for the most recent systems is lower than

for CIRRUS, even though the faster peripheral units are used. Furthermore, in calculating utilization figures for the hypothetical system, it was assumed that all output would be made at a rate of 1000 l.p.m. In practice, some output, even if only of comments, would be made on a monitor typewriter. The figures of Table 3.5. also show how much more significant any period of completely idle time is for a faster processor.

Any marked improvement in the speed of card units, paper tape units or printers is unlikely in the near future. It is also difficult to see what can be done to reduce operating delays other than restricting operation to trained personnel. However, in an era when the scientific user is demanding more direct access to the computer, a "closed-shop" is most undesirable.

To make the fullest use of the considerable power of the inexpensive processors available today, time-sharing seems essential. However, conventional single-program time-sharing can fairly quickly be rejected as a possible solution. Apart from the fact that substantial improvements are unlikely, a serious disadvantage of single-program time-sharing is that thought must be given to its implementation during the writing of every program. In an environment of scientific computation, programs are prepared by a great number of users, most of whom are only part-time programmers.

Many programs have a very limited life, some being run only once. It is therefore unlikely that more than a few programs will be written in a way which makes good use of the time-sharing facilities.

On the other hand, time-shared operation in a multi-program computer is automatic. No thought need be given to time-sharing by the individual programmer. Multiprogramming is therefore an immediately attractive possibility.

3.2. The Value for a Small System of the Flexibility given by Multiprogramming.

Activities requiring only a small proportion of total computer time become more economic if their operation is time-shared. By time-sharing a system between a number of programs, the arguments against the use of very slow peripheral units, or against on-line computer operation, are greatly reduced. Many possible applications of the computer which would otherwise be rejected on grounds of cost become feasible. This increased flexibility resulting from multi-program operation, often considered the greatest benefit obtainable from multiprogramming large systems, can also be of great value with smaller systems.

Discussion on the efficiency of peripheral units has so far been confined to the operation of those basic units carrying the bulk of input and output. There are, however,

other commonly used units for which time-shared operation is even more important. The author is familiar with conditions in a network of four computers (one Control Data 3600 and three Control Data 3200's) set up by the Australian C.S.I.R.O. All four systems include incremental plotters, and it is already apparent that, since the computers are used for scientific work, use of the plotters will be particularly heavy. To counter the inefficient operation of the plotter in the large system, steps are being taken to force information for the plotter to be written on magnetic tape. The actual plotting will then time-share with work on later jobs.

Such a solution is not practicable for the smaller systems, since one, and possibly two, tape units are thus tied up for quite long periods.

The "solution" is, in any case, merely a make-shift, requiring as it does special attention to the needs of one peripheral. If another very slow unit is to be added to the system, a major reconstruction of the time-sharing procedure is necessary. On the other hand, an efficient multiprogram system would allow time-shared operation of the plotter on-line to the program using it. Later addition of units should require only the most minor modification. Since at most one program in three or four requires plotted output, it is unnecessary to have a plotter for each program of the multiprogram set. The CIRRUS system, for example, assumes that

there is a "pool" of peripheral units beyond the basic set for each program. A program may request allocation of any of these units not already allotted to another program.

A small computer is often included in large installations. Though here its primary purpose is magnetic tape preparation or output conversion for a larger machine, the small computer is also useful for card listing or reproducing and other odd jobs. The advantages of a small multiprogram, rather than a single-program, computer in this situation are obvious. A consistent advocate of multiprogramming would say that data conversions are better done by the large system if it is properly multiprogrammed. This argument certainly holds until a point is reached when the time of the large system is saturated. It would then be irritating to feel that a significant proportion of the time of a very powerful processor was being spent on quite trivial work.

A small multiprogram processor could thus profitably be included in the system as shown in Fig. 3.1. For data conversions, a small processor designed with this purpose in mind can be as fast as the large processor at a fraction of the cost. If the time of the large processor on its own were divided fairly equally between computation and data conversion, the total work capacity of the system would be doubled merely by adding the small computer. In principle, this approach is similar to that used in the Control Data 6600, with the

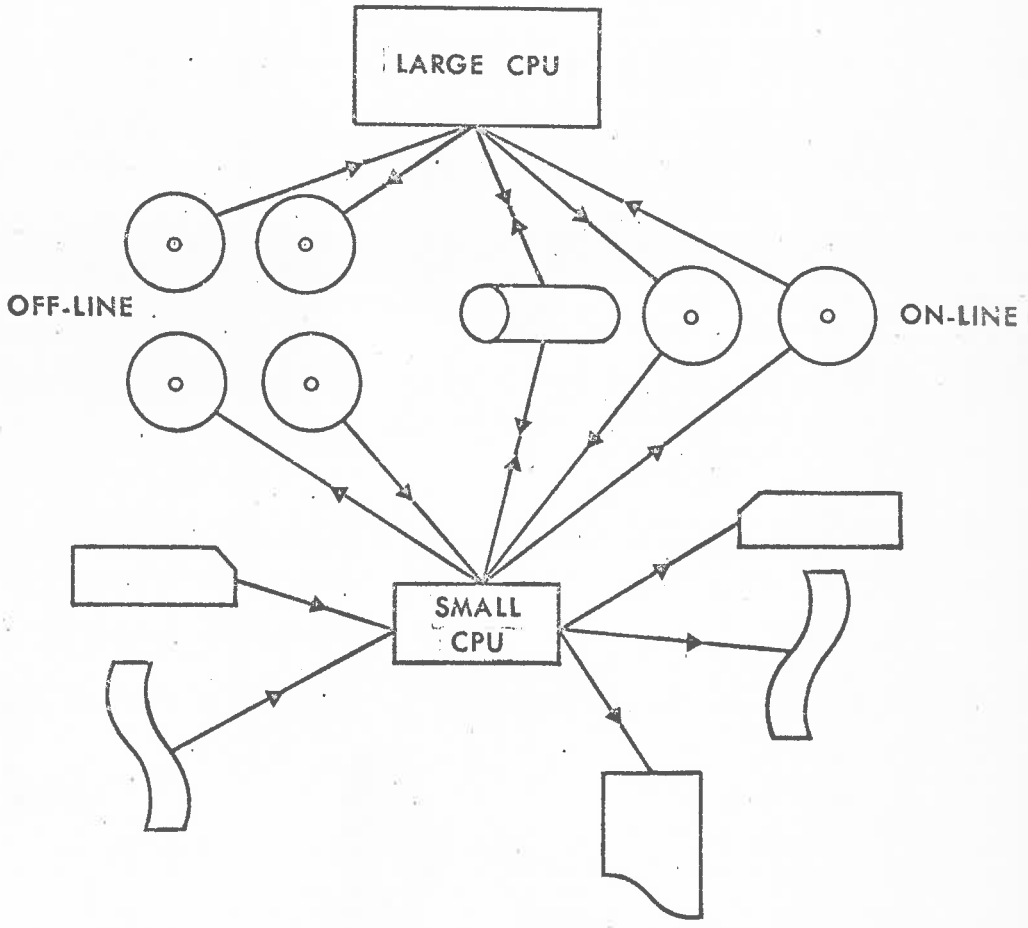


FIG.3.1.: USE OF A SMALL MULTIPROGRAM PROCESSOR
(ON-LINE OR OFF-LINE) WITH A LARGER PROCESSOR.

difference that using a single multiprogram peripheral processor rather than separate single-program processors is suggested.

The point of the example is that, although multi-programming has been regarded as chiefly suitable for large processors, situations do exist where large and small processors are used together and where multiprogramming the small rather than the large is preferable.

Many authors consider that the greatest benefits to be derived from multiprogramming are gained from allowing the computer to be shared by a number of independent operators. It is hardly economical for a small system to have, as the M.I.T. system does (Fano, 1963), a great number of independent enquiry stations. Nevertheless, a small system can have 3 or 4 separate operating stations of which 1 or 2 are reserved for users wishing to work on-line. This arrangement also has a profitable built-in bias towards the running together of low- and high-demand activities.

The cost of data transmission equipment makes it uneconomical to place operating stations at great distances from a small computer. However, the cost of low-speed data transmission links would in some cases be amply repaid. For example, in the author's experience of university computing centres, considerable inconvenience was suffered by some users whose departments were as much as a mile away from the

computer itself. Additional operating stations, strategically placed, would have been invaluable. The flexibility of the multiprogram computer also shows to advantage when the question of installing editing equipment at outlying stations is considered. For example, each operating station built for CIRRUS can reproduce or print from paper tape by using simple programs operating within the multiprogram system. No additional editing equipment need be provided.

The extremely powerful program testing systems which can be developed for large, multi-operator computers are, unfortunately, not adaptable for the smaller computer. While his program is under test, however, each user's purpose is fairly well served if he is able to have frequent access to the computer at short notice. This need should be adequately filled by the one or two consoles of a small system reserved for users.

Strachey (1959) has mentioned the value of on-line working with problems involving iterative calculations. His remarks apply equally to small and to large multiprogram computers. Other authors have suggested construction of special purpose consoles for multi-operator computers. Some applications are particularly appropriate for the smaller system. As one example, consoles may serve as elaborate desk calculators (as suggested by McCarthy et al., 1963). A further example may be seen in CIRRUS, where a set of

special-purpose micro- and machine-code programs will make the computer behave, to a user on one specific console, as a digital differential analyser.

A very small computer, the Elliott 803 (Cook, 1960), has been time shared between general computation and on-line process control. The computer was designed with this particular application in mind. A general-purpose multi-program computer storing control programs as high priority members of the multiprogram set would fulfil this function at least as well.

Much interest has recently been shown in the possible use of cathode-ray tube displays, particularly as teaching machines. One cannot justify a very large number of displays in a relatively small system, although a group associated with CIRRUS is having some success in developing low-cost displays. These, while operating independently, share a good deal of common hardware. An inexpensive equivalent of the "light-pen" facility is also being developed. A program used for teaching will probably be large. However, a technique which will be described in Section 6.2.2. makes possible the simultaneous and independent use of sequences of stored instructions by a number of separate sources. One result is that a single teaching program held in store can be shared by a number of independent users. Each user need be allotted only a quantity of private work-space. The

storage limitations of the small system therefore do not prevent the using of the small multiprogram computer for teaching.

Many other examples could be given, but those offered should be adequate to show the great advantages in flexibility which multiprogramming gives in a small system.

CONCLUSIONS TO SECTION 3.

Certain important conclusions may be drawn from the material of this section:

1. Trends in speeds and costs over the past several years have seriously increased the imbalance between central processor and peripheral equipment speeds in small scientific computer systems. The result is that a processor is likely to very much under-employed even during actual program operation. Central processor speed, up to a certain point, has become a quite cheap commodity. One should therefore attempt to take fullest advantage of it.

2. The greatly increased flexibility given by multi-programming permits more user-machine contact and allows use of the computer in areas where it might otherwise be uneconomic. While the gains may be greater for a large system, they are nevertheless of real significance for a small system.

3. Though single-program time-sharing can be rejected as a worthwhile means of more effectively using the high speed of the processor, multiprogram time-sharing should certainly be considered. Since there are other potential benefits, a thorough investigation should be made of what multiprogramming has to offer and whether multi-program operation is possible for a reasonable cost.

4. A CASE FOR MULTIPROGRAMMING SMALL COMPUTER SYSTEMS

II: Improvements Possible and Feasibility.

From the discussion in the preceding section, it was concluded that the factors giving designers an incentive to multiprogram large systems apply also where a small system is considered. Before a case for multiprogramming small systems is completely established, however, two obvious objections must be examined:

(1) The number of programs time-sharing a small computer must necessarily be quite small. It may therefore be argued that a worthwhile improvement in work output would be difficult to achieve.

(2) For even as few as two or three programs to time-share, certain parts of the system must be extended beyond what would suffice for single program operation. Although not the only requirement, additional storage is the most obvious. For a low-cost system, any extension of facilities is relatively more expensive. It may therefore be felt that the additional cost would outweigh the advantages to be gained.

It is therefore necessary to find, quantitatively if possible, the value of multiprogramming, and to assess the feasibility of achieving a reasonable degree of multiprogram

operation at a moderate cost.

Discussion in Section 2 has made it clear that at least two distinctly different methods exist to provide multiprogram time-sharing. The first method requires a backing store as an input-output reservoir, so that input-output processes from a number of programs may time-share the processor with computation on a single program. The second method, using only one-level store, provides time-shared operation between a number of programs which compete for processor time.

For the CIRRUS system, the second of these methods was chosen. CIRRUS is also a multi-operator system. For the purposes of a theoretical investigation, the CIRRUS processor can be regarded as being shared between work supplied to it through a number of separate operating stations.

The author's more detailed investigations of the effects and cost of multiprogramming have been made for the case where, as in CIRRUS, one-level store is used. To prove that multiprogramming can be worthwhile, it is necessary only to show its value for one particular method of implementation. The relative merits of an approach requiring a backing store will be discussed later.

4.1. The Improvement in Work Capacity through Multiprogram Operation.

The improvement in work done by time-sharing between a number of programs depends on many complex factors: for example, the average demand and pattern of demand for time by each program, the combinations in which programs happen to be run, and the procedure used for time-sharing. Though probably straightforward as conceived initially, the time-sharing procedure itself may become quite complex. Its detailed development and practical effects are subject to many limitations of the computer configuration and the software used. The number and complexity of such factors eliminate theoretical analysis as a method of finding an accurate estimate of the improvement possible by time-sharing. Short of an actual field test, the only satisfactory method is by simulation.

However, for the purposes of a preliminary investigation and as a check on the validity of later simulations, upper and lower limits will be found for an "improvement factor" I, a measure of the increase in the capacity of the processor to deal with the work which it must do. In discussion which follows, the term "work-load" will be used to mean a very large number of jobs to be processed.

Let us first consider the situation where the processor is not time-shared. The total time T required to

process a given work-load can be divided into alternate segments T_i, t_i, where

T_i is a period over which the processor is used continuously, and

t_i is a period, following T_i, in which the processor is not used.

Processor utilization is therefore

$$D = \frac{\sum_i T_i}{\sum_i (T_i + t_i)} = \frac{\sum_i T_i}{T}$$

We can take T, the time required for processing without time-sharing, as a quantitative measure of the "work" in the work-load. Delays by the operator contribute to Σt_i, and therefore to T. However, since the operator is part of the computer system, what he does is a necessary contribution to processing of the work-load. Operator delays unavoidable in single-program operation must therefore be taken into account in any comparison against multiprogram operation.

Suppose that, by time-sharing, the time required to complete the work-load is T'. The capacity of the system to carry out work has then been increased by a factor I, where

$$I = \frac{T}{T'}$$

For example, if the time required to process the work-load is halved, processing capacity is obviously doubled.

Now,

$$I = \frac{\sum_i (T_i + t_i)}{T'} = \frac{\sum_i (T_i + t_i)}{\sum_i T_i} \cdot \frac{\sum_i T_i}{T'} = \frac{D_T}{D}$$

where D_T = processor utilization when the processor is time-shared. Hence, the ratio:

$$\frac{\text{processor utilization with time-shared operation}}{\text{processor utilization without time shared operation}}$$

also measures the improvement in processing capacity.

Suppose that work is supplied to the processor through n stations, S_1 ..... S_n, and that each, if allowed exclusive access to the processor, would use a proportion d_j, $j = 1, \ldots n$, of total processor time. If there is no concurrent operation of processes from any single station*, an obvious upper limit for I is therefore

$$I_U = n$$

---

*The CIRRUS system does in fact allow "intra-program" time-sharing. For reasons made clear in Section 2.2., the contribution of this facility towards increasing work output may not be very great. However, there were other reasons for providing it. (See Section 10.)

Processor time available for the work-load is reduced in practice by "overhead", or time spent in implementing time-shared operation. The computer configuration, time-sharing method, composition of the work-load and the distribution of the work-load between stations will all affect overhead time. However, for a particular multiprogram computer and work-load, let us express overhead as ϕ_n, where ϕ_n is a proportion of total processor time.

Another upper limit for I is therefore

$$I_U = \frac{1 - \phi_n}{D}$$

In a system using only one-level store, the capacity of the store has an important bearing on the improvement obtained. For the present, it will be assumed that there is always sufficient storage for a program from every station.

To establish a lower limit for I, let us first consider a system where there are two constraints:

(1) Time is shared according to a fixed order of priorities between stations, work from station S_j taking precedence over work from station S_{j+1}, and

(2) There can be concurrency of useful work not requiring processor time with work requiring time only where the former takes place on a higher-priority station.

The improvement factor for this hypothetical system would be well below that for a more general case because:

(1) Varying the priority order depending on the nature of activities on each station can increase the amount of work done, and

(2) in practice, work not requiring processor time will often be done while the processor is busy with work from a higher priority station.

Let us suppose for the moment that the two constraints hold, and consider first a special case where there is no overhead time. Through station S_1 alone, a quantity of work will be processed equal to that done without time-sharing. Furthermore, a proportion $(1 - d_1)$ of total processor time will be shared between work from lower priority stations. Each station will have highest priority use of a proportion

$$\prod_{k=1}^{i-1} (1 - d_k)$$

of total time, and will leave a proportion

$$\prod_{k=1}^{i} (1 - d_k),$$

again of total time, free for lower priority stations.

The improvement factor for this hypothetical system in which ϕ_n is assumed to be zero, is

$$I_{H,\phi_n=0} = 1 + \sum_{j=2}^{n} \prod_{k=1}^{j-1} (1 - d_k)$$

Suppose that the work-load, requiring T sec. to complete without time-sharing, is instead processed in T" sec.,

i.e. $I_{H,\phi_n=0} = \frac{T}{T"}$

If, on the other hand, some overhead time must be spent, let the time required to process the work load be T' sec.

Let the <u>actual time</u> spent in overhead be θ sec.

Now,

$$T' \leqslant T" + \theta .$$

Inequality, rather than equality, will hold if any overhead time coincides with a period where no station wishes to use the processor.

Therefore,

$$I_{H,\phi_n \neq 0} \geqslant \frac{T' - \theta}{T'} \cdot \frac{T}{T"}$$

i.e. $$I_{H,\phi_n \neq 0} \geqslant (1 - \frac{\theta}{T'}) . I_{H,\phi_n = 0}$$

Now $T' \geqslant \frac{T}{n}$, so

$$I_{H,\phi_n \neq 0} \geqslant (1 - \frac{n\theta}{T}) \cdot I_{H,\phi_n = 0}$$

The right-hand side of the inequality is a lower limit of the improvement factor for the hypothetical case,

and, therefore, a lower limit for the more general case where the two constraints imposed do not hold. In the general case, the improvement factor therefore lies in the range:

$$\left(1 - \frac{n\theta}{T}\right)\cdot\left\{1 + \sum_{j=2}^{n}\prod_{k=1}^{j-1}(1 - d_k)\right\} \leqslant I \leqslant \text{Min}\left\{n, \frac{1 - \phi_n}{D}\right\}$$

Now,

$$\frac{1 - \phi_n}{D} = \frac{1 - \theta/T'}{D} \leqslant \frac{1 - \theta/T}{D}$$

The range of I may be expressed as:

$$\left(1 - \frac{n\theta}{T}\right)\cdot\left\{1 + \sum_{j=2}^{n}\prod_{k=1}^{j-1}(1 - d_k)\right\} \leqslant I \leqslant \text{Min}\left\{n, \frac{1 - \theta/T}{D}\right\} \qquad \ldots(4.1)$$

Overhead θ can be considered in two separate parts:

(1) Space-sharing overhead, processor time spent in assigning storage and peripheral units,

(2) Time-sharing overhead, processor time spent in switching from one activity to another.

It is probable that, for any multiprogram computer, the quantity of overhead time in processing a given work-load can be expressed in terms of the number of programs and quantity of input and output in the work-load. Suppose that, for each program to be processed, a period s such that

$$s_1 \leqslant s \leqslant s_2$$

is needed for space-sharing decisions. Suppose also that, for each unit* of information to be transferred, a period t such that

$$t_1 \leqslant t \leqslant t_2$$

is spent in time-sharing decisions.

If, in the work-load there are p programs and m units of information to be transferred, then Θ lies in the range

$$ps_1 + mt_1 \leqslant \Theta \leqslant ps_2 + mt_2$$

Hence,

$$\left\{1 - \frac{n(ps_2+mt_2)}{T}\right\}\left\{1 + \sum_{j=2}^{n}\prod_{k=1}^{j-1}(1 - d_k)\right\} \leqslant I \leqslant \text{Min}\left\{n, \frac{1 - \frac{ps_1+mt_1}{T}}{D}\right\} \quad \ldots (4.2)$$

Suppose that the work-load is very large and is divided randomly between stations. That is, d_i D for all i. If the work-load were not divided at random, it can be assumed that the "scheduling" carried out would be aimed at improving the rate at which the work-load was processed. D can therefore be substituted for all d_i in the expression for I_L.

i.e.,

$$\left\{1 - \frac{n(ps_2+mt_2)}{T}\right\}\left\{1+\sum_{j=1}^{n-1}(1-D)^j\right\} \leqslant I \leqslant \text{Min}\left\{n, \frac{1 - \frac{ps_1+mt_1}{T}}{D}\right\} \quad \ldots (4.3)$$

---

* Any convenient quantity.

For a particular computer and time-sharing procedure, t_1, t_2, s_1 and s_2 should be calculable with reasonable accuracy. For a given work-load run on this computer, T, D, m and p are constants and estimates for them should also be calculable. I_L and I_U are therefore expressed in terms of only one variable n, the number of separate stations.

The most important result to be found is the improvement in processing rate to be gained by time-sharing between a given number of programs. Processor utilization by the work-load without time-sharing (D) is the most important factor. The calculation is very much simplified if an estimate can be made for a range of θ/T. In a system using only one level store, the quantity of time used in overhead should be reasonably small, but will certainly be significant.

Let us take CIRRUS as an example. The time spent in space-sharing overhead now appears to be less than a second per program.

i.e.,

$$0 \leqslant s \leqslant 1 \text{ sec.}$$

Following a description of the CIRRUS time-sharing method in Section 7.3.3., it will be explained that a period t, such that

$$12\mu s \leqslant t \leqslant 184\mu s$$

is spent by the processor for each character of information transferred to or from the computer.

i.e.,

$$\frac{12.10^{-6}.m}{T} \leqslant \theta/T \leqslant \frac{p+184.10^{-6}m}{T}$$

where there are p programs in the work-load, m characters of information to be transferred and the work-load would require T seconds to complete without time-sharing.

For estimates of p, m and T, consider again the sample set of ten programs used in simulations and mentioned in Section 3.1. Suppose that each of the programs is run twice, being compiled once and assembled once from binary object code. Then, referring to Table C 2., it will be seen that:

$$p = 20$$

$$m = 3 \times 10^5 \text{ characters}$$

$$T = 2800 \text{ sec.}$$

Therefore,

$$.001 \leqslant \theta/T \leqslant .027$$

Suppose that, generally, overhead is such that

$$0 \leqslant \theta/T \leqslant 0.03$$

i.e.,

$$(1-.97n)\left(1+\sum_{j=1}^{n-1}(1-D)^j\right) \leqslant I \leqslant \text{Min}\left\{n, \frac{1}{D}\right\} \quad \ldots\ldots(4.4)$$

From (4.4) upper and lower limits of I can be calculated for particular values of D and n. I_U and I_L for $0<D<1$ are plotted in Figs. 4.1., 4.2., 4.3. for n = 2,3 and 4 respectively.

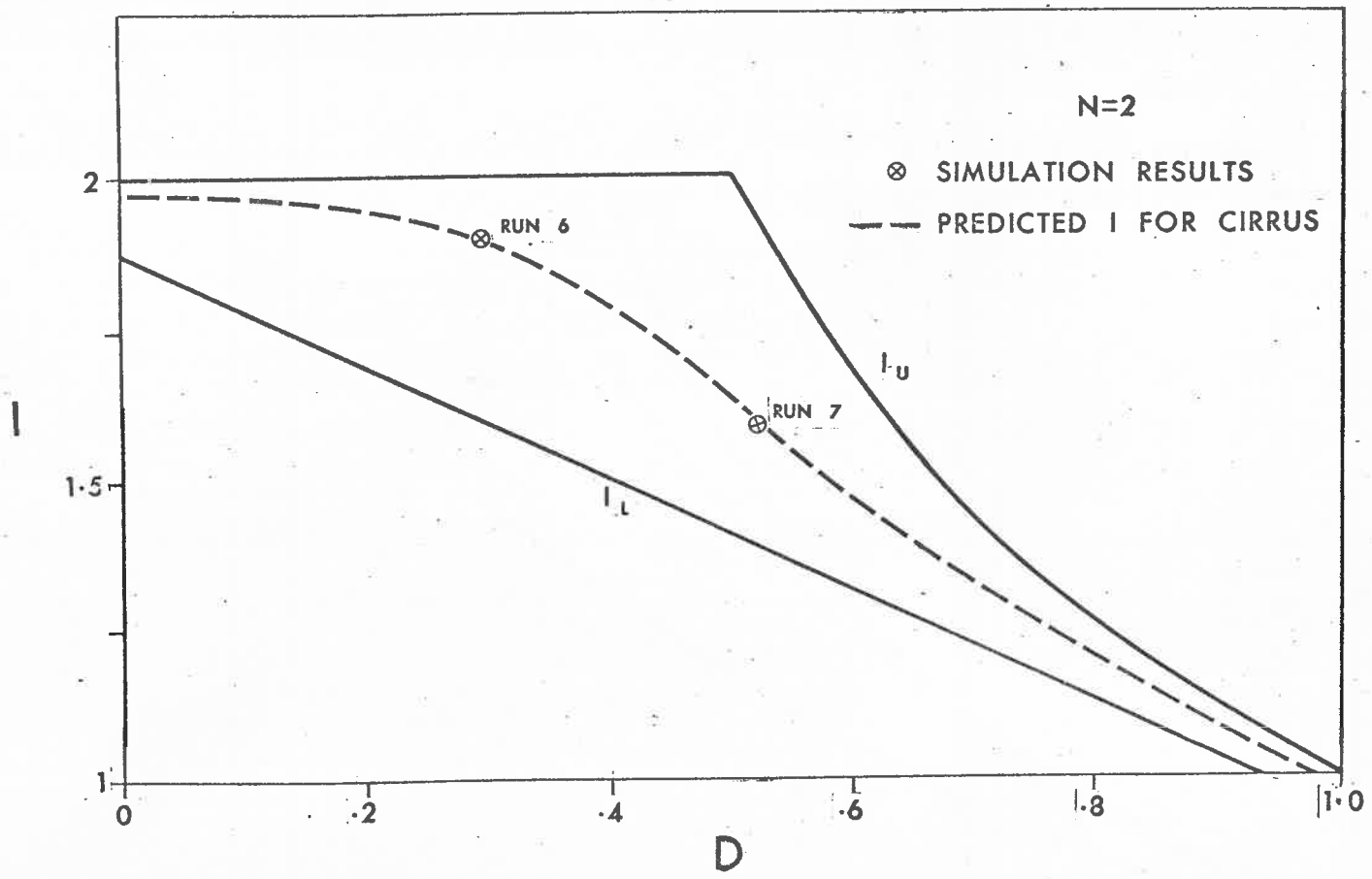


FIG. 4.1.: UPPER AND LOWER LIMITS I_U, I_L, FOR THE IMPROVEMENT FACTOR PLOTTED AGAINST PROCESSOR UTILIZATION D. 2-PROGRAM CASE.

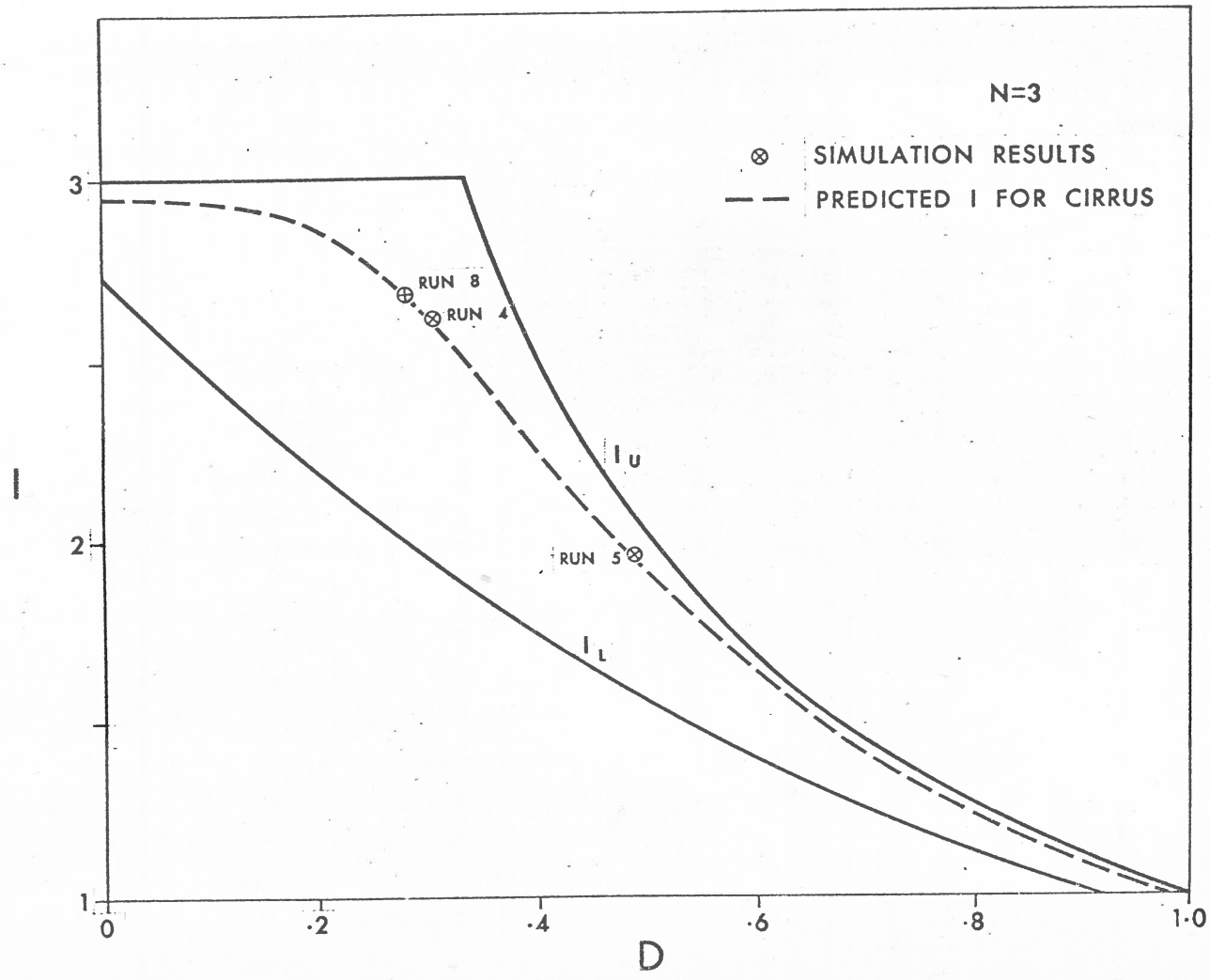


FIG. 4.2.: UPPER AND LOWER LIMITS I_U, I_L, FOR THE IMPROVEMENT FACTOR PLOTTED AGAINST PROCESSOR UTILIZATION D. 3-PROGRAM CASE.

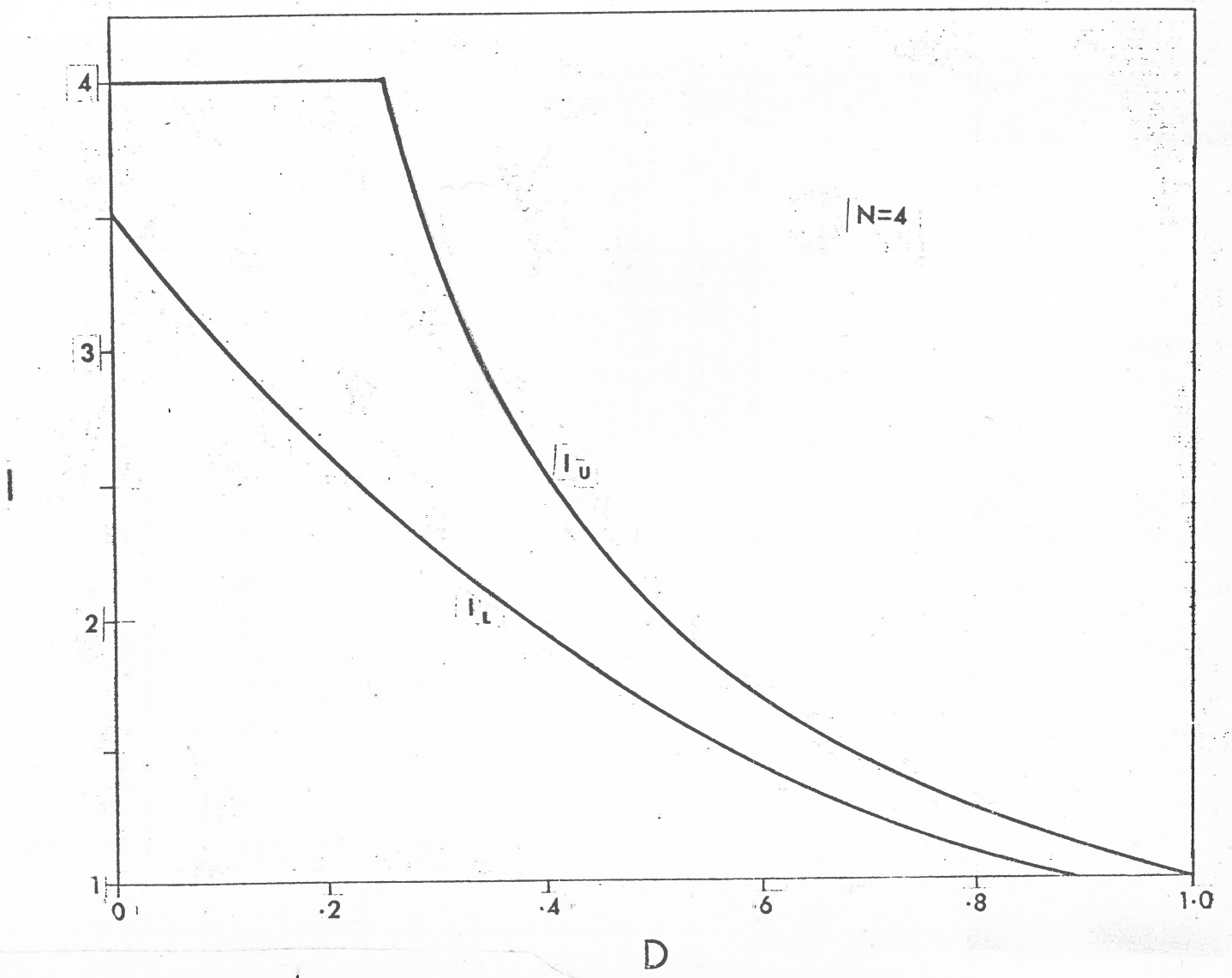


FIG. 4.3.: UPPER AND LOWER LIMITS I_U, I_L, FOR THE IMPROVEMENT FACTOR PLOTTED AGAINST PROCESSOR UTILIZATION D. 4-PROGRAM CASE.

This theoretical calculation is sufficient to show that the work-capacity of a system is very greatly increased by time-sharing between a quite small number of programs. If an estimate of processor utilization can be made for a particular computer, I_U and I_L can be calculated to give some idea of the improvement to be expected and the number of stations which could profitably be provided. However, over an appreciable range of D for each n, I_U and I_L are quite far apart. For a more accurate estimate of I, one must resort to simulation.

The author carried out extensive simulations on an IBM 7090 and, later, a Control Data 3600, testing the effect of different time-sharing procedures with CIRRUS (see Appendix C). The objective was not only to find more accurate estimates for the improvement factor, but also to assess the value of minor modifications of the basic time-sharing procedure. The simulation was made in considerable detail, and the results for the procedure chosen to be used should be an accurate indication of what can be expected in practice.

The "work-load" processed by the simulator was built up from programs in the sample set described in Section 3.1. For each simulation run, the work-load comprised about 20 programs. The ordering of programs in the work-load was random except for the restriction that no program could be included twice before all programs had been included once.

However, since one aim was to compare slightly different time-sharing procedures, the order of jobs was kept constant for all simulation runs.

The incidence and extent of delays by an operator was initially considered to be too difficult to predict. The early simulation runs were made with no delays assumed. Hence, processor utilization without time-shared operation would have been about .29 . The results of two simulation runs should be quoted here:

Run 6.

Initial conditions: $n = 2$; $D = .293$*

Improvement expected: $1.62 \leqslant I \leqslant 2$

Actual result: $I = 1.89$

Run 4.

Initial conditions: $n = 3$; $D = .294$*

Improvement expected: $2.09 \leqslant I \leqslant 3$

Actual result: $I = 2.61$

For one simulation run, a 10-second delay by the operator between jobs was assumed:

---

*D was calculated for work actually processed. The point at which simulation terminated varied slightly (see Appendix C). Hence, small variations in D were possible.

Run 8

Initial conditions: $n = 3$; $D = .277$

Improvement expected: $2.14 \leq I \leq 3$

Actual result: $I = 2.67$

No good purpose would have been served by repeating the simulation for lower values of D, unless a number of stations greater than 3 were to be considered. Examining the results of these simulations in conjunction with the theoretical values of I_U and I_L plotted in Figs. 4.1., 4.2., quite accurate estimates of the improvement factor can be made where D is small and n = 2 or 3. On the other hand, it is important to know whether substantial improvements are still obtainable if the work-load makes heavier demands on processor time. If there were a slower processor or faster peripherals for the same work-load, or if the work-load were to contain a higher proportion of computation, D for the work-load would be larger. For fairly high D, I_U and I_L are close enough together to allow a reasonable estimate of I to be made. However, where D = .5 and particularly for n = 2, I_U and I_L are far apart. Simulation is therefore desirable. A parameter in the simulator was altered to increase the length of each period of computation (see Appendix C 3), making D for the work-load about .5 . The results obtained where n = 2,3 were:

Run 7

Initial conditions: $n = 2;\ D = .518$

Improvement expected: $1.41 \leq I \leq 1.93$

Actual result: $I = 1.58$

Run 5

Initial conditions: $n = 3;\ D = .487$

Improvement expected: $1.69 \leq I \leq 2.05$

Actual result: $I = 1.97$

The results of these simulation runs have been plotted in Figs. 4.1., 4.2. Since each run required about 45 minutes of computer time, it was not possible to make more than a very few runs. However, taking the simulation results in conjunction with the graphs plotted for I_U and I_L, curves can be drawn to predict the improvement factor with reasonable accuracy for all D where n = 2 or 3 (Figs. 4.1., 4.2.).

4.2. Some Preliminary Remarks on Feasibility.

It is not intended at this stage to examine the question of feasibility in detail. However, before the case for multiprogramming is concluded, a few preliminary remarks on feasibility must be made. In the preceding part of the present section, improvements in work output were calculated for a situation where processor time was shared between work supplied through separate stations. It must be shown that the cost of providing time-shared operation is not excessive.

For the particular type of multiprogram operation under discussion, there are two obvious requirements. First, there must be, for each operating station, at least one input and one output unit. Second, there must be sufficient storage to hold, for a reasonably high proportion of the time, a program from every station. Additional cost may also be incurred in providing the time-sharing mechanism itself. More detailed discussion on feasibility will later show that this cost can be kept quite small.

The cost of additional operating stations can fairly readily be calculated, and will therefore be dealt with first. Each operating station used in the CIRRUS system comprises an Elliott paper tape reader, a Teletype paper tape punch, an IBM typewriter and an operator's keyboard. The first three of these units were purchased complete, while the last was built on site. Controlling circuits for all four were also constructed on site. The total cost to the University for each station was under £2,000. A market price for a complete station should not be more than twice this figure. Hence, the cost of additional stations would not in itself constitute a barrier to the implementation of multiprogram operation.

No attempt will be made here to suggest likely storage requirements by the programs in a typical work-load. However, much useful work has been done in recent years by small to very small scientific computers such as the IBM 1620 or the

Control Data 160. The stores in these computers held as few as 2,000 words. The most probable store size in one of the small computers now coming into use is 8,000 words. Furthermore, some comparisons of CIRRUS against the Adelaide University's 1620 suggest that 8,000 words of store in CIRRUS are worth not 4, but at least 10 times as much as the 20,000 characters of the 1620 store. The chief factors reducing the relative value of storage in the 1620 are: first, a need to use 8,000 characters of storage for arithmetic and input-ouput subroutines, and second, a much smaller instruction set. In only an 8,000-word store, CIRRUS could hold simultaneously 3 programs whose respective storage requirements were n, 2n, 7n, or 2n, 3n, 5n and so on, n being the equivalent total capacity of the 1620. There would no doubt frequently be large programs requiring most of an 8,000-word store. Nevertheless, the improvement in work output when the processor is actually being time-shared is so significant that the cost of one or two additional operating stations would probably be justified.

However, to take fullest advantage of what multiprogramming has to offer, a larger store must be provided. Let us therefore examine the economics of enlarging the core store. It is not, of course, necessary to increase storage capacity beyond that felt reasonable for a single-program

machine by a factor equal to the number of programs we hope usually to be able to hold in store. The magnitude of the increase depends on two things: the distribution of storage requirements by programs which would have been written for the small store, and the extent to which selection can be made of programs which are to be run together. Admittedly, when store size is increased, larger programs will be written, some of which will be so large that they will take up almost the whole of the store. However, one must assume that the problems for which such programs are written would have required two or three programs run sequentially in a machine with a smaller store. Any temporary inhibition of multi-program operation is therefore more than balanced out. If only those programs designed to fit the original store are considered, and provided also that some small degree of selection of programs is made, then a doubling of store size should almost always give storage for three programs.

The true cost of increasing core store is one of computing's mysteries. Certainly, if one has a system with, say, 8,000 words, and wishes to add a further store module, the price will be substantial. However, the cost of core store has decreased spectacularly in recent years. If experience with CIRRUS, the case for which the author can speak with most authority, is taken as a guide, core store

cost is one of the less serious impediments to multiprogramming. The CIRRUS store is addressed and buffered with general purpose 18-bit registers. Hence, the greater part of the store cost lies in the cost of the core planes themselves. The most recent prices quoted by suppliers for the core store itself have been £1,237 for 2048 36-bit words, and £2,783 for 16,384 words.

Suppose that a single-program CIRRUS system with an 8,000-word store is taken as a basis for comparison. Suppose also that processor utilization in this system is between .2 and .3 (see Section 3.1). A system with a 16,000-word* store and two additional operating stations, would have a work capacity at least 2½ times greater for a cost only 10 to 20% higher.

Unfortunately, CIRRUS is not in every respect a typical example. For many computer installations, punched card input and output is required. Card units are several times more costly than paper tape units. Nevertheless, unless processor utilization is quite high, the improvement obtainable by multiprogramming should be adequate to justify the cost of at least one further set of peripheral equipment.

---

*The original design for CIRRUS included a store of only 8,000 words. However, realization of the importance of multiprogramming, together with the drop in store cost, brought about a decision to increase store-size to 32,000 words.

CIRRUS has only moderate performance (6μs cycle-time) store. Faster stores whose cycle-time is as low as 1.5μs are now used in many small computers. Since such stores are, for the present, very much more expensive, the economics of increasing store size would be very different.

A fast store is undoubtedly a strong inducement to a prospective purchaser. However, unless processor utilization is fairly high, substituting a fast store for a slow one is of very limited value. Where processor utilization is initially D, an increase in processor speed by m times would increase work output by a factor

$$I' = \frac{1}{(1-D) + D/m} \quad \ldots\ldots (4.5)$$

A store four times faster in its operation would increase processor speed by, at most, a factor of 3. If utilization of the original processor were .25, the improvement gained by using the faster store would be only

$$I' = \frac{1}{.75 + \frac{.25}{3}} = 1.2$$

It was stated in Section 3 that processor speed has become a very cheap commodity, but only up to a certain degree of speed. Beyond a particular point, the cost of obtaining increased speed rises steeply. Store speeds and costs afford an illustration. A 1.5μs store at present costs more than twice as much as a 6μs store.

In the author's view, the first step in designing a low-cost processor should be to map out a basic structure which relies on the inexpensive yet fairly high-performance store and circuitry now available. An attempt should then be made to estimate the likely utilization of the processor under normal conditions of use. From this estimate, the true value of any measure to increase processing capacity can then be found and weighed against the expected cost.

In constructing the CIRRUS processor, the designers used only very low-cost components, yet produced a moderately high-performance processor. This processor was completed in 1962. Today, the "basic" low-cost processor would be significantly faster. In developing the case for multiprogramming CIRRUS in this thesis, the author has followed precisely those further steps suggested in the preceding paragraph. Unless there are definite reasons for believing that the time of a processor will be very heavily employed, similar conclusions on the value of multiprogramming should be made by designers of other computer systems.

So far, discussion has dealt only with the increased efficiency of processor utilization gained by multiprogramming. The neglect of the gains possible in flexibility has been deliberate, but should not be construed to mean that the advantages from increased flexibility are of less importance. The point is that the improvement in work

capacity and the cost of obtaining it are directly calculable; the value of the increased flexibility is not. The arguments given so far have shown that the improvement in processing capacity of a system by multiprogramming should, in itself, more than justify the cost of implementing multi-program operation. Increased flexibility may be regarded as an extra dividend, a dividend which a few examples will show to be considerable.

As a first example, let us suppose that a user wishes to work on-line. If the "normal" rate of working is taken to be that rate at which a trained operator can dispose of the work-load in a single program machine, then a user-operator interested only in his own problem could cause a reduction in this rate by a factor of 5 or 10 times. Though the user will often feel that the value of his own work justifies the reduced efficiency of operation, the computer management will rarely agree with him. On the other hand, if the user were given one operating station in a 3 station system, the reduction in efficiency is not nearly so serious. Rate of working is of course proportional to processor utilization. The simulation results show that:

(1) Where D for the work-load $\div$ .29

D_T = .76 for n = 3 (Simulation run 4),

and D_T = .55 for n = 2 (Run 6),

and (2) When D for the work-load $\doteq$ .50

D_T = .95 for n = 3 (Run 5),

and D_T = .81 for n = 2 (Run 7).

Hence, when D = .29, no matter how inefficiently the user chooses to work, the reduction in rate of working cannot be greater than 28%. When D = .5, the maximum reduction is 15%. The value of on-line working to the user certainly counterbalances reduced efficiency to only this degree. Even if two of the three stations were given over to the users, the rate of working would still be at least equal to that obtained with a single-program system.

Similar arguments can be used to show the degree to which on-line operation of a slow peripheral unit such as a plotter is more acceptable on a time-shared system.

A corollary from the above discussion is that multiprogramming can still be worthwhile even where processor utilization is very close to saturation. The worth to users of on-line working will in itself often justify the cost of an extra operating station or two and additional storage. Since the activities of the users on their private operating stations would make very small demands on processor time, their effect on the rate at which the main work-load is processed would be barely noticeable.

As a second example of the value of the flexibility given by multiprogramming, let us suppose that the computer

is to be used for some process control function. If, as is probable, very little processor time is needed overall, the total cost of permanently maintaining the control function is little more than the initial cost of storage for the control program and the necessary data links to and from the computer.

CONCLUSIONS TO SECTION 4

1. It was suggested in Section 3.1. that, if the processor of a small scientific computer were not time-shared, it would be in use for only about 20-30% of total time. If such a processor is time-shared between two programs, the processing power of the computer system is almost doubled. By time-sharing between three programs, processing power is increased to well over 2½ times the power of the original system.

2. The cost of additional storage and peripheral units, the most obvious requirements for multiprogram time-sharing, need not present an economic barrier to the implementation of multiprogram operation. For CIRRUS, the particular example on which much of the discussion has been centred, the increase in power and flexibility of the computer very definitely justifies the moderate increase in cost.

5. CHOICE OF THE BASIC STRUCTURE.

It has been shown in the preceding sections that multiprogramming is desirable and should be economically feasible in a small scientific computer. Discussion has so far been restricted to a particular type of multiprogram computer, that is, one in which programs would be supplied through separate operating stations and would share a quite large but low-cost core store. There are of course alternative structures.

In building a low-cost computer, one must attempt to make best use of the limited resources available. It may not be possible to take advantage of all that multiprogramming offers. The relative importance of the diverse benefits from multiprogramming should therefore be assessed.

Existing multiprogram computers differ substantially in their structures, apparently as the result of significantly different design objectives. Three systems will be examined and, with CIRRUS, will be used as illustrations of contrasting structures and objectives.

5.1. Design Objectives.

The author has taken the view that the primary design requirement for a multiprogram computer to be used in a scientific institution should be the provision of

separate operating stations. In fact, the guiding principle in designing CIRRUS has been that the computer should "behave as a set of separate and independent computers" (Penny, 1960).

It was suggested in Section 3.1. that the relatively slow operation of the peripheral units in a small computer system is a primary factor causing inefficient processor usage. Particularly slow peripheral units or slow-speed data links with other equipment will often be wanted in a scientific computer system. Hence, if multiprogramming is to be implemented, one must certainly ensure that a substantial contribution is made towards improved efficiency of input and output.

The other possible applications of a multiprogram computer should also not be neglected. In many installations, special-purpose consoles or computer control of other equipment will not be required. However, there may be installations in which one or more of these special applications are of fundamental importance. The multiprogram system must either provide for these applications or be flexible enough in structure to allow modification and expansion to allow for them.

5.2. Some illustrations.

Before attempting to specify a computer structure which should best satisfy the objectives outlined in the preceding part of this section, one should consider some existing computer systems. A most obvious difference is either the use of multi-level storage or reliance on core store only. ATLAS and the Honeywell 800 afford contrasting illustrations.

The Honeywell 800 (References: Harper, 1960; Honeywell 800 Executive System Manual, 1961) is a large computer in which time is shared between a number of programs held in core store. Generally, a single instruction is executed from each program in rotation. The programs awaiting completion of a data transfer are simply omitted from the time-sharing process until ready to proceed.

In many ways, the Honeywell 800 system is the result of quite direct development from the large, single program computer of a few years ago (Fig. 5.1.). The peripheral unit controller of the single program machine has its counterpart, the "Traffic Control" unit, in the Honeywell 800. When supplied with the necessary parameters by a data transfer instruction in any program, this unit controls the transfer of single words until the request is fulfilled. The "monitor" routine has been extended into an elaborate master

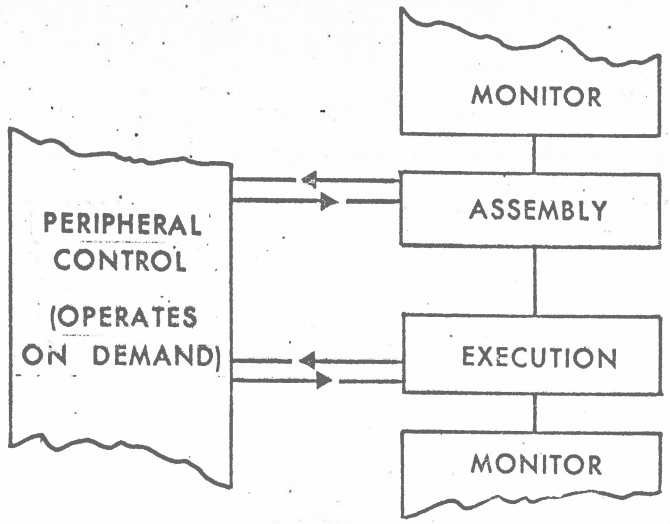


FIG. 5.1: PROGRAM OPERATION IN A LARGE SINGLE PROGRAM COMPUTER

TIME SHARING BETWEEN PERIPHERAL UNIT CONTROL AND THE MAIN PROGRAM IS ACCOMPLISHED WITH A MIXTURE OF HARDWARE AND SOFTWARE.

OPERATOR ACTION IS LIKELY TO INVOLVE A CONTROL TRANSFER TO THE MONITOR.

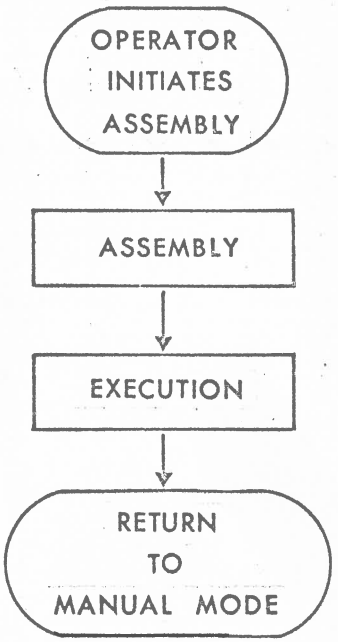


Fig. 5.2: PROGRAM OPERATION IN A SIMPLE SINGLE PROGRAM COMPUTER.

OPERATOR ACTION EITHER REQUIRES A COMPLETE HALT IN OPERATION, OR MUST BE ALLOWED FOR IN THE PROGRAM.

program, "Executive", which selects and schedules programs for a proposed production run. A "Multiprogram Control" unit has been added to give the time-shared execution of programs during each production run.

Backing storage, partly on drum and partly on tape, is fundamental in the time-sharing system of ATLAS (References: Kilburn et al., 1961; 1962; Howarth et al., 1963). Over fairly long periods, processor time is shared between a single "object program" in core store and a number of "interrupt subroutines" which transfer information between the peripheral units and input-output reservoirs or "wells" in the backing store.

At execution time, the object program takes its input information from an input well and stores output information in an output well. Direct control of peripheral units has therefore been removed from the program itself.

The interrupt subroutines form part of the master routine, the "Supervisor", which also provides monitor functions and program selection. Switches of control between the object program in core and the Supervisor can be made very rapidly. When paper-tape units are operating, for example, a switch is made from the object program to the Supervisor for each character transferred. Changes from one object program to another may require the interchange of a

large proportion of the contents of core store. Such changes are, however, necessary only after fairly long intervals.

In both ATLAS and the Honeywell 800, the designers' sole objective appears to have been to gain efficient time-sharing of input and output. There is, however, one noteworthy difference between the two. The references describing the Honeywell 800 emphasise the use of fast peripheral units. On the other hand, the references on ATLAS stress use of slower units, particularly paper tape units.

It is therefore not surprising that dissimilar structures were chosen for these two computers. The designers of ATLAS envisaged a work-load requiring a good deal of input and output through fairly slow peripheral units. Since the ATLAS processor is extremely fast, it was essential that many peripheral units be able to operate simultaneously. If a structure similar to that of the Honeywell 800 had been chosen, a very large number of programs would have had to be held in core store.

The idea of using separate consoles to give users on-line access to the computer appears to have been neglected by the designers of the two multiprogram computers so far described. Much of the important development work in the design of large-scale multi-operator computer systems has been carried out at the Massachusetts Institute of

Technology. For his third and final illustration of existing multiprogram systems, the author will consider a fairly small system, the PDP-1, which is very similar in principle (if not in scope) to the M.I.T. system.

The purpose of the designers of the PDP-1 time-sharing system (McCarthy et al., 1963) was to "increase the effectiveness of the computer for those applications involving man-machine interaction". Provision has been made for five on-line users, each to have his own typewriter. The system described in the reference has a core store of 8,192 words plus a drum store.

A program in "run status" may use the processor continuously for 140 msec. After this period, the contents of core store are transferred to a drum field designated as the "core image" for this particular program. Another program in run status then replaces the preceding program for the following 140 msec. The "memory swap" itself occupies 33 msec.

Of the 8192 words in core store, 4096 are reserved for the time-sharing system. Programs do not control the paper tape input and output units directly. As in the ATLAS system, part of the backing store is used as an input - output reservoir. The method of time-sharing input and output appears very similar to that in ATLAS, except that, as far as one can gather from the reference, only one input and one

output unit are used.

The principles of the CIRRUS structure can be understood from the theoretical model analysed in Section 4. It was suggested earlier that the Honeywell 800 system is the result of a quite direct development to multiprogram operation from the large, single-program computer of a few years ago. The CIRRUS system, on the other hand, may be regarded as having been developed from the smaller system whose operation is shown diagrammatically in Fig. 5.2.

Hardware associated with peripheral units is quite elementary and each program controls its units directly. The work-load is supplied to the computer through separate operating stations. Programs share the core store and "compete" both for store-space and for processor time. The systems software is chiefly concerned with ensuring that both store-space and time are shared in an efficient manner and with accepting and carrying out instructions received from the separate operators.

Two of the four systems described use backing storage in addition to core storage. The possible use of a drum in a low-cost multiprogram computer will be discussed in the following part of this section.

5.3. Choice of the Store Configuration.

Consider a simple computer having a small core store and one input and one output unit. Let us suppose that, instead of increasing core store size and replicating peripheral units as has been done in CIRRUS, a drum store is added. It can be assumed that a greater total store capacity would be obtained for a given cost. The additional storage available can be put to two uses. First, a greater number of programs can be stored. Second, part of the drum can be used as an input-output reservoir. That is to say, input information can be stored in advance of its actually being required and output information from the computing program can be stored when it is produced faster than the output unit can handle it.

By using part of the drum as an input-output reservoir, time-sharing of input and output can be achieved without any replication of peripheral units. A configuration including a backing store is, if only for this reason, worthy of investigation if the total system cost is to be low.

Replication of some unit or units may, however, still be desirable for reasons which will now be explained: Suppose that the total time of the system without any time-sharing were found to be divided in the following way:

(1) Idle time - a proportion p_1,

(2) Computing without input or output - a proportion p_2,

(3) Input - a proportion p_3,

(4) Output - a proportion p_4.

Suppose also that, during time-shared operation, the input and output units are kept in operation for proportions of total time $1 - \varepsilon_1$ and $1 - \varepsilon_2$ respectively. For this particular system, the values of

$$\frac{1 - \varepsilon_1}{p_3} \quad \text{and} \quad \frac{1 - \varepsilon_2}{p_4}$$

are therefore upper limits of the improvement factor as defined in Section 4.1. ε_1 & ε_2 are certainly non-zero. Their magnitude will depend partly on the ability of the operator to keep the units operating and partly on the likelihood of the input reservoir being completely filled or the output reservoir completely emptied.

Suppose, for example, that the output unit would be in use for half the total time if there were no time-sharing. If it is found that, when operation is time-shared, the unit can be kept working for 80% of total time, then

$$I_U = \frac{.8}{.5} = 1.6 ,$$

is an upper limit for the improvement factor. However, in these circumstances, it would soon be clear that output capability was the factor limiting the capacity of the system. If the output unit were duplicated, and if each

of the two units could be kept operating for proportions $1 - \varepsilon_2$ of total time, then the upper limit placed on the improvement factor by the output capability would become:

$$I_U = \frac{2(1 - \varepsilon_2)}{p_4}$$

If an accurate prediction can be made of the relative quantities of input and output in the work-load, a given degree of improvement should certainly be obtainable after replication of fewer peripheral units than are necessary in a system of the CIRRUS type. If interest lay solely in achieving efficient time-sharing of input and output, and particularly if the units to be used were of an expensive type, a system using backing storage might well be considered first.

However, it was stated in Section 5.1. that provision of facilities for on-line users should receive highest priority in the design. A personal keyboard and monitor typewriter are the most likely choices for each on-line user. If there is to be no replication, or limited and selective replication of input and output units, each unit must be shared between a number of users.

The particular type of multiprogram operation chosen for CIRRUS has made replication of input and output units obligatory. However, in having an input and an output unit entirely to himself, the user is very much better served

than if he had only the keyboard and typewriter. Moreover, the cost of replicating the input and output units need not be a great deterrent if, as in CIRRUS, paper-tape equipment is used. One must therefore conclude that, unless the input or output units to be used are of a very expensive type, choosing a configuration with a backing store is not justified solely on the ground that peripheral units need not be replicated.

Furthermore, where input and output units are shared between separate users in a multi-operator system, the backing store size must be considerable. In the PEP-1 system, for example, the user does not "except at the beginning and the end of his session, ordinarily use the paper-tape apparatus. Instead, he designates a position on the drum for the punch and a position for the reader". Space must therefore be available on the drum for <u>all</u> input to or output from each program which is to time-share the system.

The core store associated with a drum must itself be of a reasonable size. Areas of the core store must be reserved for the following:

(1) Buffers for information transfers to or from the drum. Transfers between the drum and peripheral units and between the drum and the computing program must be buffered if an efficient rate of transfer is to be maintained. One cannot afford to economise on the size of these buffer areas.

(2) Software to control the operation of peripheral units and the transfer of information to or from the drum.

(3) Software to handle the "memory swap" needed to change the computing program in core store.

(4) Software to read and interpret instructions from the on-line users. No user should be required to wait until his program is in core store. The system should be able to read an instruction from any operator at any time and either execute it or notify the operator that its execution will be delayed.

The PDP-1 system described by McCarthy *et al*. has a core store of 8192 words, half of these being reserved for the "time-sharing system". The author considers that a core store of this size is the absolute minimum for a reasonably efficient time-shared system. Each programmer of the PDP-1 "sees" a 4096-word core memory. One has only to consider the question of program compilation to realize that, with an apparent core store size of 4096 words, a great number of inter-store transfers will have to be performed. If the core store were any smaller, the position would be intolerable.

A number of techniques can assist the programmer in his use of two-level storage. The simulation of one-level

storage in ATLAS (Kilburn et al., 1962a) can be quoted as an example. Unfortunately, implementation of these techniques requires either special purpose hardware or core store space. They are therefore not really suitable for a low-cost system.

It is obvious that the minimum cost at which an effective system using two levels of storage can be implemented is moderately high. At present costs, a 32000-word core store of moderate performance is significantly cheaper than the two-level storage for this minimum configuration. It must be remembered that the cost per word of core storage decreases very substantially as the size of the store increases.

Use of only one level of storage gives the advantage of simplicity. Let us again consider the question of program compilation. In CIRRUS, the 3000-odd instructions constituting the compiler are stored permanently and are available for use independently by separate programs. Each program is compiled directly into the space which it will occupy during execution.

The biggest disadvantage in using a system which has only core store is that the number of programs held in store must be limited. Though the CIRRUS system allows for three separate programs, there are times when only one or two programs will fit into the store. If one wishes to permit

access to the computer by as many as, say, half-a-dozen on-line users, one would certainly choose a configuration with a backing store. However, a system having one or more sets of basic input-output units, half-a-dozen enquiry stations and core and drum storage of adequate capacity would be of quite considerable cost.

It was also stated in Section 5.1. that a minor objective should be to ensure that the system has the flexibility to allow for other possible applications of a multiprogram computer. Fulfilment of this objective depends less on the store configuration than on the software used to implement multiprogramming. It should be possible to fulfil the objective in either configuration, probably more simply if core store only is used but more economically if there is a backing store. If core store only has been used, additional storage may be needed. If one of these applications requires considerable program storage, the two-level store system would therefore be more suitable. On the other hand, if frequent short periods of processor time are needed, the one-level store system would be more efficient.

CONCLUSION TO SECTION 5.

The discussion on store configurations must lead to a conclusion which appears to be something of a paradox.

For a large multi-operator system, a configuration having a large backing store will certainly be used. However, for a really low-cost multiprogram computer and particularly if there are to be separate facilities for on-line users, one-level storage must be preferred. The requirement for multiple operating facilities reduces the weight of arguments in favour of the use of two-level storage on the ground that peripheral units need not be replicated. This requirement also increases the quantity of core and drum storage needed to produce an effective system. The minimum cost for which a two-level store system can be implemented is therefore moderately high.

6. THE CIRRUS SYSTEM - PRELIMINARY REMARKS

The objectives which guided development of the CIRRUS multiprogram system and the basic structure chosen have now been described. The methods by which multiprogram operation has been implemented will be covered in detail in Sections 7 to 9. The present section is intended to constitute an introduction to the later sections. In addition, some of the more important decisions which have been made and some features which might be considered unusual will be discussed.

6.1. External Form.

The CIRRUS multiprogram system at present allows for three separate operating stations. Further stations could be installed after some revision of the systems programs. Each station must have an operator's keyboard, a monitor typewriter and a paper-tape reader. Each of the present stations has also a paper-tape punch. The stations are therefore rather more elaborate than the "enquiry stations", comprising only a keyboard and typewriter, which might be found in a large, multi-operator computer system.

The cost of paper-tape readers is low enough to justify providing a reader on each station. Including these

units in each station is of course the first step towards fulfilling the aim that each operator should feel that he has a computer entirely to himself.

It is very convenient for each operator to be able to work completely independently when operating stations are close together. It becomes essential if any station is to be situated at some distance from the computer itself. Some preliminary development has in fact been done of low cost data transmission equipment intended to link the computer with an outlying station.

Only one program can be controlled from each station. It will however be shown later (Section 10) that a single external program may be divided into separate internal programs which can function concurrently. The system could certainly have been constructed to permit two or more programs to be controlled from a single keyboard. However, the cost of an operator's keyboard and the associated typewriter is small. Once a program has been initiated from a particular keyboard, all subsequent operating instructions must apply to that program. The need to identify the program to which an instruction refers is removed and the chance of an operator's error is greatly reduced.

It can now be seen that the theoretical model analysed in Section 4.1. does in fact cover the CIRRUS case. However, in the model it was assumed for simplicity that

the stations through which the work-load would be supplied were identical. Though all CIRRUS stations need not be identical, there is one definite advantage to be gained from keeping them as alike as possible. If stations have the same units, a user can take any station which happens to be free.

Output capability (at least for hard copy) of the system is at present low. A line-printer would be a very valuable addition. Unfortunately, the cost of a line-printer makes it feasible to provide only one at most, yet for almost every program the programmer would like to use it. To obtain maximum use of the printer, the author has suggested that it and a paper-tape reader constitute a separate, special-purpose station. No keyboard is needed. Loading the reader and making it "ready" would automatically initiate printing. For as long as paper-tape output were available to be printed, the printer would be kept operating continuously. However, it would certainly be possible to allow the printer to be used by some other station when circumstances warrant (Section 8.5).

There are likely to be other units which may or may not be needed by any individual program. These units must not be tied to any station but should rather constitute a pool of units which can be flexibly allotted to any station

(Section 8.5). Possible additions in this category are a magnetic tape unit or two, and a plotter.

6.2. Storage

CIRRUS has a core store of 32,678 words, each of 36 bits. Though a large store for a low-cost machine, the need to share its space between a number of separate programs has made its efficient use essential. The CIRRUS system permits "dynamic" relocation of partially executed programs (Section 8.4.), so that vacant sections of store space may be consolidated into a single section.

Instead of providing static registers to carry out the functions of accumulators and index registers, a second core store of 64 words, the "register" store, has been included. Additional expense has been incurred in providing this store. However, use of store rather than static registers has enabled each program to have several "registers" entirely to itself. The contents of registers therefore need not be stored when changing between programs (Section 7.3.1.) Efficiency is improved by using a second store since the main and register stores can be operated in parallel (Appendix A 2.).

Of the 32,678 main store words, 8,192 are reserved for semi-permanent "read-only" storage of a type developed

by Butcher (1964). In this store, validated system routines and other commonly used programs will be held. To be usable in a multiprogram computer, programs held in fixed store must be able to refer to different store addresses or peripheral units at different times. A technique called "parametric addressing" has been introduced which not only allows the necessary varying of addresses, but also permits sequences of instructions to be shared simultaneously by separate programs.*

Discussion of the sharing of store-space leads directly to the important question of possible interference by one program with another. The author's views on program protection will be explained in Section 6.2.3.

6.2.1. Efficient Use of Core-Store.

All vacant store space must be available to an incoming program which needs it. When any program ends, it is quite likely to leave vacant a section of store space between sections of space occupied by programs which are still operating. To the author, the most satisfactory procedure is to shift all programs to the head of the store,

---

*This technique was checked out in 1962 (see Penny, 1963). It gives a result similar to that of "re-entrant" procedures in NPL (see Radin & Rogaway, 1964, p.24). It must be stressed, however, that the instruction sequences are shared by separate programs.

thereby consolidating all vacant space at the end of the store.

Though relocation of partially executed programs has its difficulties, "dynamic" relocation is possible in at least two systems, the FP 6000 and CIRRUS. Each instruction in the FP 6000 referring to a store address holds this address not as an absolute address but as an address relative to the base or "datum" for the particular program. When needed, the datum is added automatically by hardware before the instruction is obeyed. When the program is shifted, only the datum must be modified.

CIRRUS, on the other hand, stores main store addresses as absolute addresses*. The base address for the program is added during program loading. During relocation of a program, these addresses are modified to become consistent with the new base address.

Simultaneous compilation of separate programs is possible in CIRRUS. The store space needed to compile a program is usually greater than would be required for execution. When compilation of a program is complete the exact storage requirement for execution is known. The upper limit shown for the program's store space is then adjusted.

---

*Addresses of register store location are, however, held as relative addresses - see Section 6.2.2. and Appendix A 2.

Should another program have taken store-space while this program was being compiled, there will probably be a section of vacant space between the two programs. When relocation is performed, this space also will be retrieved for use by other programs.

The CIRRUS procedure has required no hardware, and it should therefore be preferred to the FP 6000 method for a low-cost system. However, its implementation presented certain problems, which will be made clear when the procedure is discussed in greater detail in Section 8.4.

6.2.2. Use of Semi-Permanent Storage.

Semi-permanent or "fixed" storage devices have been available for some time. Their limited use is evidence enough of their drawbacks. Fixed store programs cannot be altered. The consequent restriction of one section of store space to a single purpose is, under normal circumstances, the most serious disadvantage of fixed storage. Yet it is precisely this attribute, that fixed store programs cannot be altered, that makes fixed storage attractive for the multiprogram computer.

The CIRRUS compiler is of the "load-and-go" type. In a small computer without a backing store, a compiler of this type would usually be read into store from cards or

paper tape for each program which is to be compiled. After compilation, the program will frequently use the compiler-space for its own data. In a multiprogram computer shared by programs originating from separate stations, it is sensible to store the compiler only once and to let the sequence of instructions constituting the compiler be shared by independent programs.

The chance that the compiler would be in use at any given time by at least one program is high. It is therefore worthwhile to store the compiler permanently, thereby obviating the need to read it in. Several other routines can also profitably be held permanently. Standard input-output routines, for example, are used by almost every program. If the "small" computer were to have a very small core store, permanent storage of routines would be uneconomical. However, CIRRUS has a quite large store and assigning as many as 8000 words of storage for permanent programs is not unreasonable.

Routines to be stored permanently might as well be held in fixed store. The type of fixed storage developed by Rutcher is both cheaper and faster than variable core store and is simple to pre-wire or modify. Programs are, in any case, checked out in variable store before being wired into fixed store. Since the fixed store functions as the high-address portion of main store, its use presents no

difficulties to the programmer. Most important of all, the instructions wired into it cannot be mutilated. Each of the independent programs using sequences of instructions in fixed store can rely on the routines being consistently available and correct.

For fixed store routines to be usable in a multi-program computer, fixed instructions must be able to refer to different store addresses or peripheral units when used by different programs. A single sequence of fixed instructions should also be available for use simultaneously yet independently by entirely separate programs. To meet this need, a method of addressing called "parametric addressing" has been introduced.

The key to the parametric addressing technique lies in storing register store addresses as relative addresses, relative that is to the address of the first register used by the program. Addresses are then converted to absolute form during the extraction from store of each instruction*. The following example illustrates this procedure. Suppose that there is a fixed instruction specifying register store address 2 as accumulator and address 3 as index register. When executed, the instruction will refer to

---

*The modification adds no time to the extraction process - see Appendix A 2.

the second and third registers of those registers belonging to the current program. Before entering the input or output routine, for example, the number of words to be transferred, the address of the first word and the peripheral unit address are pre-set in specific register store locations. Double index register modification is also possible to a limited degree. Sequences of store words may therefore be addressed.

The use of parametric addressing need not be restricted to sequences of instructions in fixed store. The ability to share a sequence of instructions in variable store can be particularly valuable where consoles are to be used as teaching machines. The basic teaching "program", which will probably be large, need be stored only once. Each console will have its own program which will make use of the basic instruction sequence but may also include sub-routines peculiar to the console. Each console would also have its own working space.

The decision on whether a given routine should be stored once and shared between separate programs must be determined by the probability that each program will require the routine. Store space has been saved if two or more programs use the routine. Space has been wasted if no program uses the routine. If, for example, it is hoped that three programs can usually be held in store, there will be

a net saving in space if those routines required by more than one program in three are held in fixed store.

Since holding a routine in fixed store saves the time which would be needed to read it in, it is probably reasonable to hold routines required by one program in six. The author examined a random sample of 64 programs and estimated from this sample the probabilities of a number of routines being required by individual programs (Table 6.1.). He suggests that the routines shown in the table be wired into fixed store.

TABLE 6.1.

ESTIMATES OF THE PROPORTION OF ALL PROGRAMS WHICH WILL REQUIRE A NUMBER OF COMMONLY USED ROUTINES

ROUTINE	EST. PROBABILITY OF USE*
Compiler	.44
Binary Assembler	.56
Input-output routine	1.0
Sine, cosine subroutine	.38
Square root "	.25
Exponential "	.22
Logarithm "	.19
Arc tangent "	.13

*These figures were found from a sample of jobs run on an IBM 1620. Where one job was run as a sequence of separate programs, the whole job was considered rather than its separate parts.

6.3.3. Inter-program Protection.

Some authors regard avoiding interference by one program with another as fundamental. Beckman (1961), for example, lists "program controlled memory or address protection" under "required hardware elements". Others are less dogmatic. Mills (1963) says: "Our experience is that hardware storage-violation protection is not worthwhile".

The problem needs careful investigation where the total system cost is to be low. If address checking is to be done with independently operating hardware, the cost will be substantial. The only alternative is to use the existing registers and arithmetic unit, in which case an appreciable amount of time will be lost.

All computers and computer operators are fallible. There is always some chance that any program will fail to reach a successful conclusion. The author feels that the protection provided within any multiprogram system is adequate if the chance of failure is not noticeably increased by multiprogram operation. Failures which could be directly attributed to multiprogram operation are those caused by:

(1) Operator action on another program,

(2) Program interference

(a) With another problem program,

or (b) With the control program.

The author has already stated (Section 6.1.) that only one program may be operated from each CIRRUS console. The control programs to read instructions from an operator are constructed in such a way that the operator can influence only that program which originated from his console.

Interference between two programs could occur through either:

(1) A program fault,

or (2) Hardware malfunction.

In CIRRUS, instructions are extracted from store by a sequence of microprogram (Appendix A 2). An optional branch in this sequence may be taken (Fig. A 2) to check the operand address of each instruction. An error exit occurs if the address is outside the permissible bounds. This check requires two further store cycles, but it can be suppressed on any program.

The address check is in fact suppressed at present on all but machine-code programs. In building the compiler, care was taken to ensure that the programs it compiled would refer to illegal addresses only in very rare circumstances. Experience so far in actual operation justifies the belief that these precautions are adequate. Furthermore, the address checking facility for machine-code programs and the checks built into the compiler have

proved extremely useful diagnostic aids.

The chance of interference through hardware malfunction can never be entirely eliminated. Some such malfunctions will affect only one program while others will cause all programs in the machine to fail. The most lavish program protection scheme can influence the effects of hardware failures only by reducing the chance that any given failure will disrupt more than one program.

Multiprogram operation can complicate the task of fault-finding, particularly when the fault is transient. However, on-line fault-finding in CIRRUS is complicated far less by multiprogram operation than it is by the fact that most of the processor hardware is general-purpose. (See Appendix A 1). A single fault may, for example, affect any of a hundred or more machine-code operations. Fortunately, this factor was well understood in the earliest stages of design and parity checking has been provided to a more comprehensive degree than would be normal for a low-cost machine. Transfers between all stores, registers and peripheral units are checked and the machine brought to halt on any failure other than where the transfer has involved a peripheral unit. It has been found in practice that nearly all hardware faults are detected by parity failures.

Van Horn (1964) quotes Professor Corbato of M.I.T. as saying that the type of program failure resulting from

interference appears exactly like transient hardware failure, and is therefore very difficult to diagnose. Van Horn concludes that, barring hardware failure, any program with a constant set of data should have the property of "repeatability". The author will now summarise his views on program protection for the small multiprogram computer and follow with some comments on Van Horn's conclusion.

Program protection by hardware is not economically feasible in a really low-cost system. A checking facility must certainly be available and will no doubt be implemented by program. As a result, time will be lost when checking is performed. It must be possible, therefore, to bypass the checking and one must assume that checking will most commonly be omitted.

If a program fails and a hardware fault is suspected, the normal procedure in a single program computer is to re-attempt execution. If the failure is not repeated, a hardware fault can be assumed. In the multiprogram computer, one should re-run the program with checking of addresses being made on <u>all</u> programs. Admittedly, there are two difficulties. First, conditions during the re-run cannot be identical, even if the same programs are involved. However, if an interference fault had occurred, the program responsible and the fault itself should be isolated. Second, the user who decides that address checking should be instituted

wants the check applied not to his own program but to the other programs. Where there are separate stations, possibly in different sites, a facility to introduce checking on all programs should be available on one station at least. The important point is that a checking facility must be available and it must be possible to introduce or remove the check at will.

The action to be taken in the event of mutilation of the multiprogram control program must also be considered. The problem is analogous to that of a large system where the monitor program or its working space may be interfered with. In such cases, the monitor is recalled from the library tape. In a small multiprogram computer, cards or paper tape must be used unless the control program is held in fixed store - a further argument in favour of using fixed storage. All that is necessary for CIRRUS is to reset the initial conditions in the working space at the head of store, a resetting of only some 128 words*.

The author realizes that his views on the degree of program protection which is necessary conflict to some extent with established opinions. However, it must be

---

*The initial setting-up of CIRRUS for multiprogram operation can be done only from Console 0.

stressed again that our concern is not with systems of the magnitude of STRETCH. Referring to STRETCH, Nash (1963) has said that, when an error occurs, "the operator has quite a headache to sort out which jobs had been run and which were still waiting on the input tapes when the error occurred . . . with a total time penalty of as much as 15 minutes of CPU time". From each CIRRUS station, programs are run sequentially. The restart procedure is therefore trivial.

6.3. Multiprogram Control.

The space and time-sharing procedures in CIRRUS allow for 15 separate programs*. "Program" here means simply an internal program. To execute a single external problem at least two internal programs are required.

The 15 internal programs are divided into two categories. Eight are referred to as "permanent" programs because their requirements in storage and peripheral units are known and constant. The remaining seven are called "variable"; storage and peripheral units can be allotted to them in a quite flexible way. In sharing time between programs, differentiation between the two categories is made only to the extent that an active permanent program will always be given priority over any variable program.

---

*Although many of these are in use in the present system, some are available for expansion.

For each operating station, there is a separate permanent program called the "operator control" program whose structure will be described in Section 9. This program reads, interprets and implements any of a standard set of operating instructions. When requested, it will set up a second, variable program, which is to carry out the work required in the external problem.

Though instructions from a particular operating station will always be read by the same control program, the variable program chosen by the control program may be any of the seven variable programs. In fact, two or more variable programs may be associated with a single control program. The purpose is not to allow several external programs to be operated from a single station but to allow the programmer, in suitable cases, to divide a single program into a master and subordinate routines which would function concurrently (see Section 10).

Consideration of those facilities which a variable program will use or is using is concentrated at two points: first, immediately prior to compilation or program loading and second, following execution. A "preliminary sequence" and an "end sequence" form part of the standard software held in fixed store (Section 8). These two sequences and the compiler and loader are fixed routines referenced by variable programs in much the same way that these programs

use "READ", "WRITE" or other subroutines. Though the instructions forming all these routines are stored only once, they can be used simultaneously yet independently by any number of variable programs. The "parametric addressing" technique (Section 6.2.2.) is the key to this simultaneous use.

Since these functions are carried out as parts of the variable program, the permanent program always remains free to implement on the variable program any instructions received from the operator. The relationship between the permanent and variable programs is shown in Fig. 6.1.

It may be said of CIRRUS that processor time is competed for by the programs rather than shared between them. A definite priority order exists, the control programs of course always having priority over the problem programs. A program will occupy the processor's time either until it is forced to halt on requesting a unit which is not available, or until a higher priority program which was halted earlier can again proceed. CIRRUS is microprogrammed, that is, each machine-code instruction has been built up with a sequence of micro-instructions (see Allen et al. (1964) and Appendix A). To control the sharing of processor time, certain of these microprograms have been extended (Section 7).

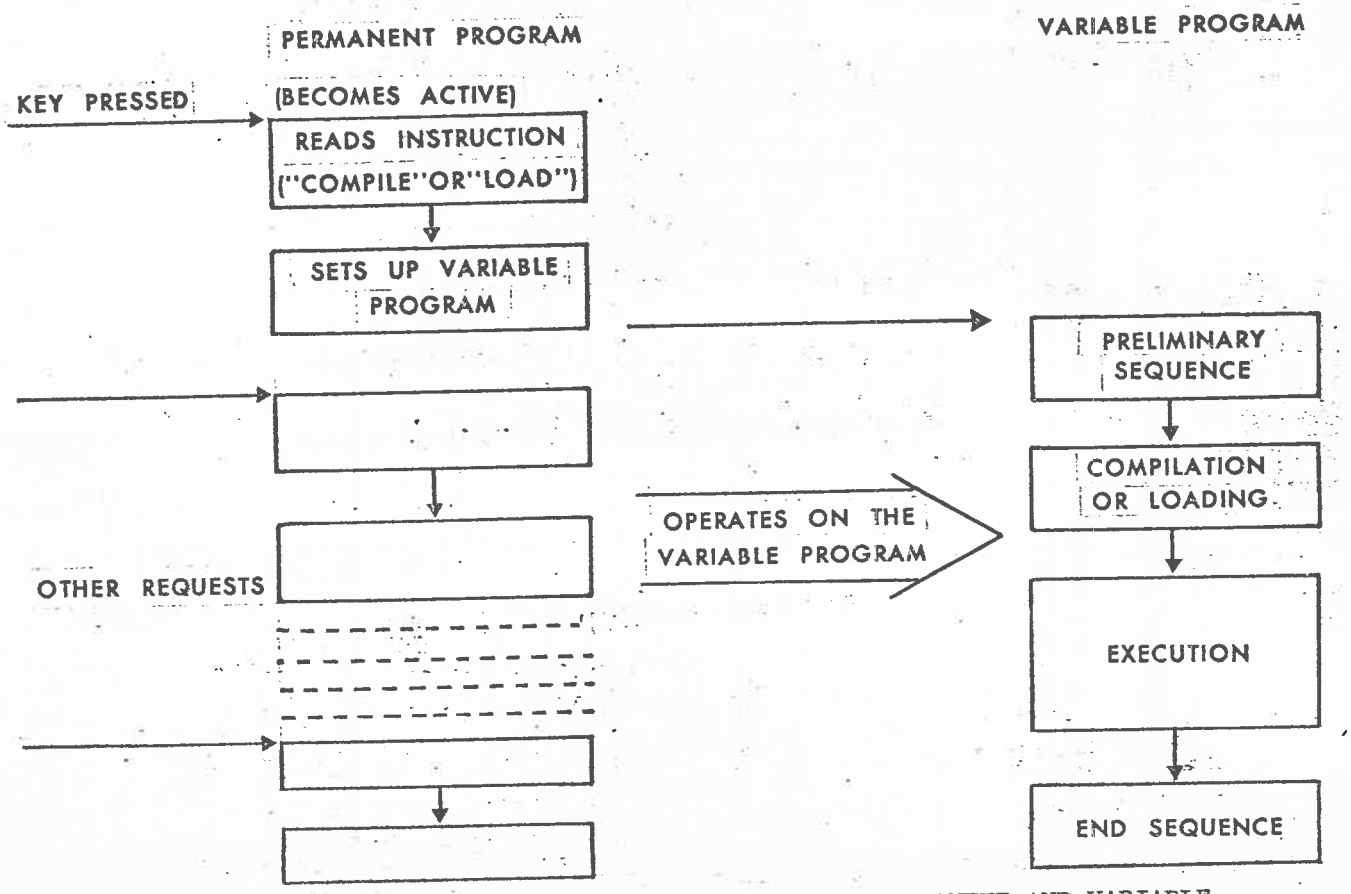


FIG. 6.1.: THE RELATIONSHIP BETWEEN PERMANENT AND VARIABLE PROGRAM IN CIRRUS

7. PERIPHERAL UNIT CONTROL AND TIME-SHARING.

In a small scientific computer, the bulk of input and output will, for the reasons set out in Section 3.1., be made through punched card or paper tape units. It has been stated that paper tape units have been chosen for use in CIRRUS. For an equivalent performance, a paper tape unit can cost as little as one-fifth as much as the corresponding card unit. Where the type of multiprogram operation desired involves replication of units, the choice of paper units is almost mandatory.

Using paper tape may have disadvantages, the most obvious being the difficulty of making isolated corrections. Since there will be, in a scientific installation, a continued flow of new programs to be checked out, a practical solution must be found to the problem of editing programs punched on paper tape.

Apart from lower initial cost, there are however other important reasons for preferring paper tape equipment. Paper tape units, since they can transfer single characters, are simpler than card units in their mode of operation. Furthermore, the multi-operator computer will almost certainly have monitor typewriters and input keyboards on its separate stations. These units, like paper tape units, receive or

transmit information character - by - character. If no other type of unit is used, a very simple procedure can be developed to control time-shared operation of all units in the system.

In this section, a simple multiprogram time-sharing procedure will be developed. This procedure is both efficient and inexpensive to implement. Initially, it will be assumed that only those units mentioned above need be catered for. However, a unit or units other than these will often be wanted in the system. It will be shown that this basic time-sharing procedure can readily be adapted to handle units of other types.

7.1. Editing of Programs on Paper Tape.

Before paper tape can be accepted as the primary medium for input and output, it must be shown that a practical solution exists for the problem of program editing. The facilities provided in the CIRRUS compiler by J.G. Sanderson (Sanderson, 1963 pts 2.1.2., 3; 1964 pt. 12.2.) have been found perfectly satisfactory, and will be briefly described.

Each source program must be preceded by a header tape in which corrections can be specified. For example, a correction tape with:

```
0 - 65
<.....................
.....................>
66 - 163
165 - 193
<.....................>
195 - END
```

directs the compiler to

(1) copy the source program following from line 0* to line 65,

(2) insert the line or lines bracketed,

(3) copy the source program from line 66 to line 193, omitting line 164,

(4) replace line 194 by the line or lines bracketed,

(5) copy the source program from line 195 to the end.

An option of the compiler may be used to print out line numbers of all labelled statements. This option would invariably be used on the first compilation run. Lines found to be in error are also indicated. The user will add to his correction tape to remove each program error. He may also, if the correction tape becomes unwieldy, use a further option of the compiler to punch out an up-dated source tape.

---

*Some statements, particularly declarations, may take many lines.

7.2. Control of Paper Tape Units.

It was stated in the introduction to this section that, quite apart from lower initial cost, there were other reasons for favouring paper tape units. These reasons will now be made clear.

Let us first define two quantities of information transfer:

(1) The "unit" of transfer, comprising a number of bits transferred in parallel,

and (2) the "block", that number of units of information on the input-output medium between successive points at which the peripheral equipment may be stopped without loss of information.

One of two measures must be adopted:

Either (1) Buffer storage independent of the central processor can be provided for a complete block of information,

or (2) The time-sharing mechanism can provide for interruptions from the peripheral, followed by transfer of a unit or number of units of information within some maximum permissible period.

In each case, there are obstacles to a low-cost solution. Buffering, whether provided independently for each

peripheral or multiplexed between a number of peripherals, will be expensive if required in any quantity. On the other hand much of the hardware of a low-cost computer will be general-purpose, and the particular hardware required for any information transfer could serve several functions. It might therefore be difficult to ensure that the hardware can always be made available at short notice.

With paper tape equipment, the unit and record of data transferred are identical, a single 5 to 8 bit character. If buffering only for single characters is provided, the obligation to allow interruptions within any given period is eliminated. With card equipment, however, the unit is either 12 or 80 bits* and the block is a complete card. Control of a card unit will either require more complex hardware or a more elaborate time-sharing procedure.

For a multiprogram computer to be operated efficiently, much more information must pass between the operator and the machine than would be necessary in a single-program machine. For this communication, an output typewriter with an input keyboard is an ideal choice. Since each of these units is essentially a single-character device, their mode of operation is identical to that of paper tape units. Thus, a single time-sharing procedure controlling paper tape units can also, without modification, cope with typewriters and

*Depending on whether cards are read "end-on" or "edge-on".

keyboards. The arguments in favour of using paper tape equipment are therefore augmented.

With only single-character units, hardware control for input-output can be elementary. Figure 7.1. shows the essential features of the CIRRUS structure. A and M are general-purpose registers. For the micro-operation causing the actual data transfer, A must hold the absolute address of the unit, and the transfer is then made between M and the relevant buffer through the common bus. It should be noted that the punch and typewriter on the same console share a single buffer*. An extra bit is added to the transmitted character to indicate its destination. Buffer sharing is only worthwhile for the output units since they release the buffer after punching or printing the character. On the other hand, the reader refills its buffer as soon as a character has been transmitted to the computer.

It is of course preferable that paper tape units should not actually stop during transfers. The efficiency of a paper tape reader in particular would be much impaired if the unit were operated on a stop-start basis. In practice, priority can be given to programs through periods in which they actually use the peripheral units (Section 11.3). The units would then be brought to a halt during transfer only rarely. The important point is that, by using peripheral

---

*The CIRRUS keyboard also shares the punch/typewriter buffer. This sharing introduced some unforeseen complications and is not recommended.

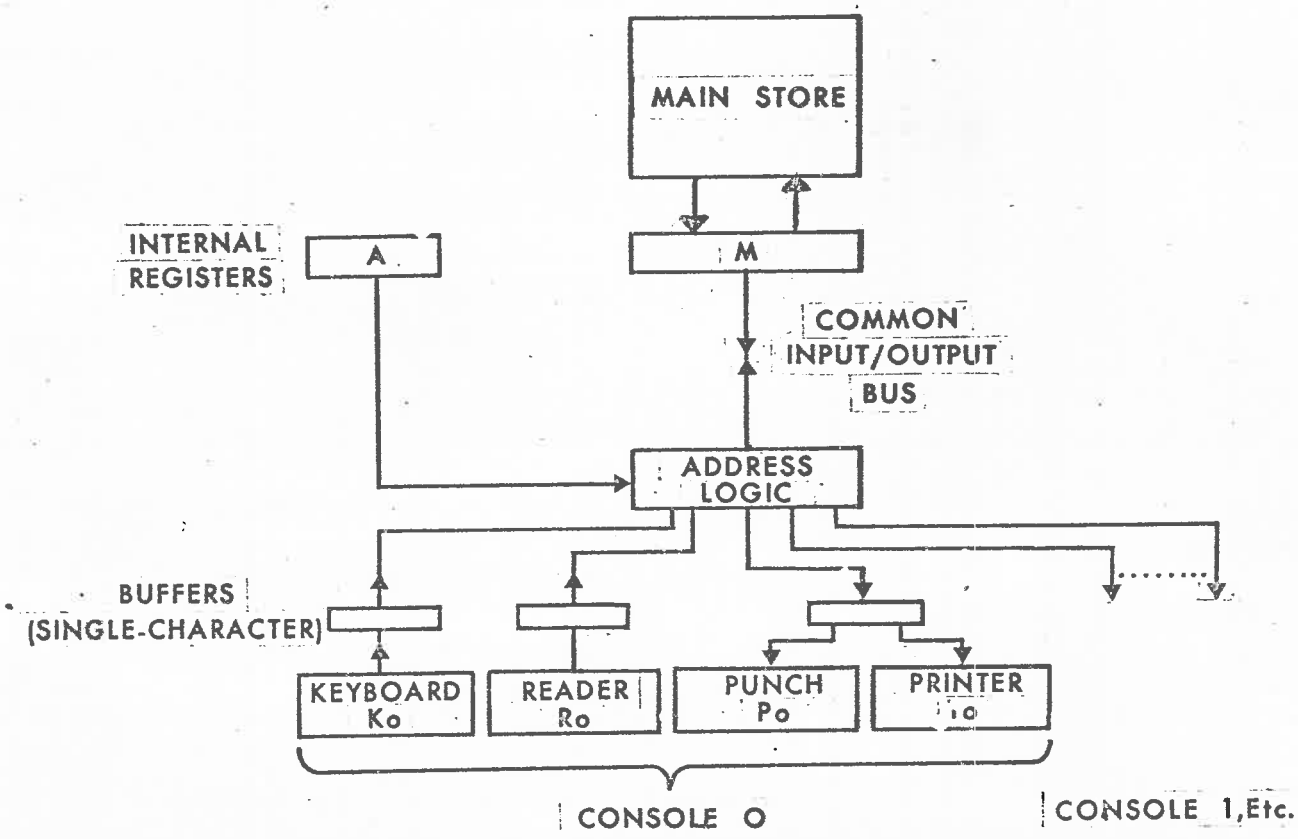


FIG.7.1.: HARDWARE FOR CONTROL OF SINGLE CHARACTER PERIPHERAL UNITS

equipment of this type, a constraint is removed which might on occasion be difficult to fulfil.

7.3. A Simple Multiprogram Time-sharing System.

In earlier sections, it has been suggested that implementation of multiprogram time-sharing is simplified if:

(1) only one-level storage is used,

and (2) all peripherals are single-character units. In this section, a very simple but adequate time-sharing procedure requiring a minimum of special-purpose hardware or program will be developed. It will be assumed that the two conditions stated above apply. The procedure to be described is largely that which has been implemented in CIRRUS. Minor details in which CIRRUS differs, if not clear from the text, will be covered in footnotes.

7.3.1. Time-sharing a Low-cost System.

For a number of reasons, it is necessary that programs requiring time be arranged in an order of priority. The importance of a priority order will be made clear in Section 9, which deals with use of programs to accept and implement instructions from computer operators; and in Section 11,

which will show the effect which the priority order can have on efficiency of the system.

Apart from the requirement for an order of priority, other factors influence the way in which processor time should be shared in a low-cost system. The decisions as to which programs are ready to use processor time, and of these, which has highest priority, will almost certainly be made by the processor itself. To save processor time, we should therefore limit both the number of decisions and the time spent in making each one.

The time-sharing procedure must therefore be very different from that used in the Honeywell 800, where single instructions are obeyed from each program in rotation. To keep down the number of time-sharing decisions the processor must work continuously on one program for a reasonably long period of time. The result should be that the program receiving processor time will continue to receive time until it can no longer itself use time, or until a higher-priority program, delayed for some reason, can again proceed. Though reducing the number of switches from one program to another will save processor time, there are other factors making fairly frequent inter-program switching desirable. The peripheral units mentioned so far will transfer information at rates varying from a character every second or so (keyboard) to 500 or more characters per second (paper tape reader).

While the highest priority program reads from paper tape, there will be as many as 500 short periods in a second during which the program cannot continue. If these periods of time are to be used by other programs, and if buffering beyond that needed for single characters is to be avoided, then it must be possible to switch between programs very rapidly indeed.

The time required to switch the processor from one program to another will depend mainly on the extent to which storage is time-shared between programs. Since, in the type of time-sharing being discussed, the switches are to be made between completely independent programs, one cannot predict what storage each might use. The contents of all time-shared storage must therefore be saved and subsequently replaced.

If program switches cannot be made rapidly enough, it would be advisable to introduce special-purpose programs to carry out the actual character-by-character transfers. These "programs" would in effect be interrupt subroutines. Since the storage needs of interrupt subroutines can be defined, the switches would require interchange of only a limited amount of storage. In a multiprogram computer using two levels of store, the whole of the core store may be time-shared between separate programs. Use of interrupt subroutines can therefore hardly be avoided. However, allowing programs to control peripheral units directly makes possible a very simple and

concise time-sharing procedure, thereby saving that most valuable commodity, store-space. One should certainly attempt to achieve very rapid inter-program switching in a system using only core store.

Most computers have a number of "registers" to which the machine-code programmer has access. If program switches are to be rapid, there must be either very few registers whose contents need be stored, or a sufficient number of registers to allow each program to have its own set of registers. Most small computers do in fact have very few registers. The writers of compilers in particular would invariably feel that there were far too few.

In at least two multiprogram computers, the FP 6000 and CIRRUS, core store locations perform the functions normally associated with accumulator, multiplier and index registers. A separate set of "registers" can therefore be allocated to each program. In the FP 6000, the first eight store locations of the program's work space are used. CIRRUS, on the other hand, includes a small second core store of 64 words,* of which 12 are usually allotted to each program. This store is called the "register store".

---

*A larger store would have been desirable, but insufficient address bits could be made available.

Using store locations rather than static registers will of course slow down some operations. Loss of efficiency in CIRRUS, where the register store can operate in parallel with the main store, should be small. The highly desirable result, that each program can have a very much greater than usual number of registers uniquely to itself, should justify the cost of providing the second store.

CIRRUS has a number of static registers (see Appendix A 1), all of which are in fact time-shared. These registers are only addressable from micro-code. They are multi-purpose, that is, their functions vary with different machine code instructions. Hence, in building each machine code instruction from micro-code, any value to be retained had to be placed in core store at the end of the instruction (see Appendix A 2). This requirement, though not dictated by the needs of multiprogramming, did give the result that program changes could be made at the end of any machine-code instruction without any clearing of these registers.

CIRRUS also has no sequence counter register. Instead, a number of store words carry out the functions of sequence counters for the various programs. Changing from one program to another therefore involves only the selection of the program to which the switch will be made, and substitution of the address of the new sequence counter. In any computer having a specific sequence counter, interchange of its

contents would be necessary.

Program changes in CIRRUS can be made rapidly enough to permit a change for each character read, printed or punched. Usage of processor time may be understood by examining Fig. 7.2., which shows the distribution of time during a punching sequence in the highest-priority program. Of the 10 ms needed to punch each character, from .5 to 1 ms only is used by the processor to prepare the next character for punching.

In the following part of this section, the facilities needed to implement sharing of processor time will be discussed.

7.3.2. The Basic Time-sharing Procedure.

To share processor time in the manner described in the preceding part of this section, the following facilities are required:

(1) an indication when the current program cannot continue,

(2) an interruption when a delayed program of higher priority can once again proceed,

and (3) a procedure in either case to determine which program should then be followed and to implement the actual switch.

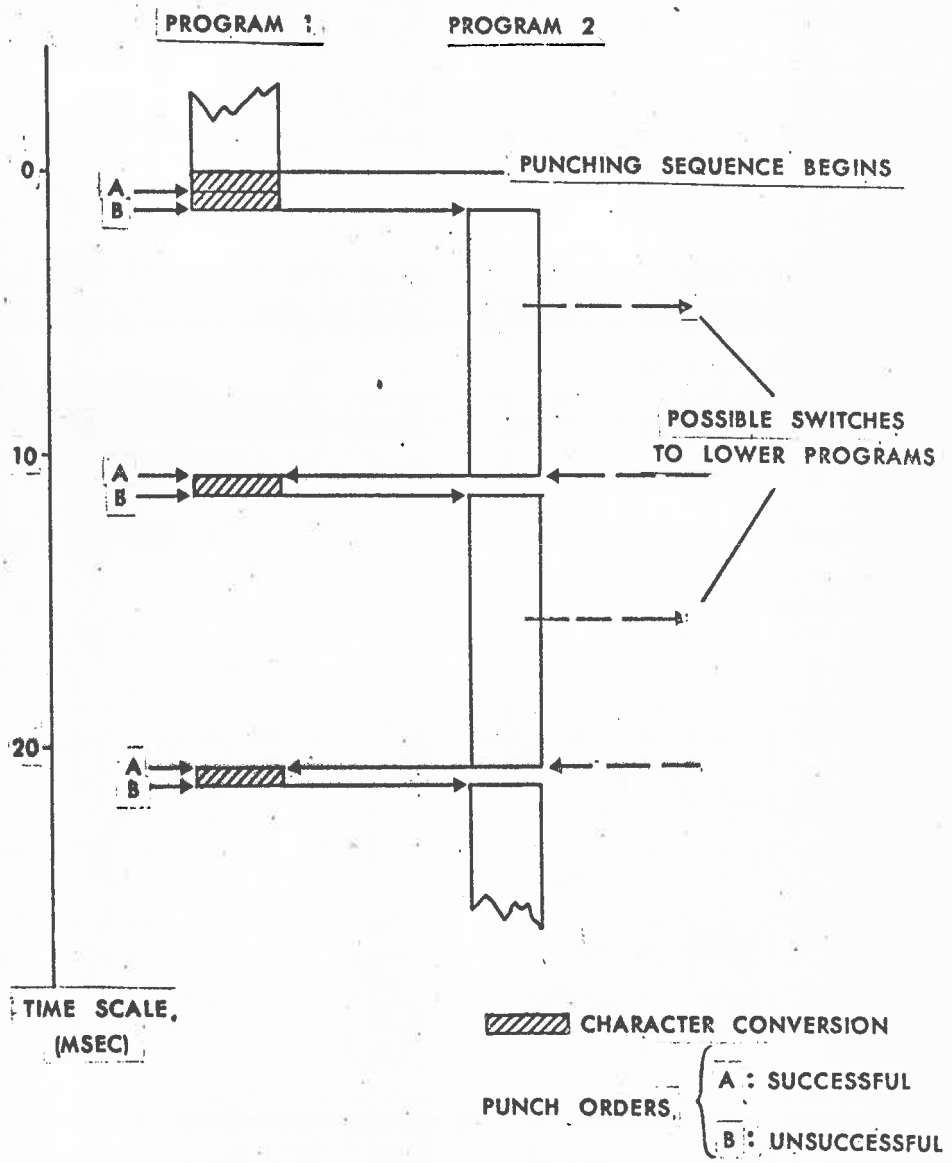


FIG. 7.2.: USE OF C.P.U. TIME DURING A PUNCHING SEQUENCE ON THE HIGHEST PRIORITY PROGRAM.

The availability or otherwise of peripheral units should be the cause of nearly all inter-program switches. The current program cannot continue further when it requires from an input unit a character before it has been set in the buffer, or when it attempts to transmit a character to an output unit before the buffer has been cleared.

In CIRRUS, there is a set of "peripheral indicators", each of which in general monitors the state of a single peripheral unit*. The state of each indicator is controlled solely by the buffer. The indicator shows the "on", or available, condition if a transfer can be performed, and "off" otherwise. For example, an input unit indicator is "on" while the unit's buffer is full and the parity condition is satisfied. The complete set of indicators can be examined through the arithmetic unit as an 18 bit (half) word. The "available" condition is shown as zero and the "unavailable" condition non-zero**.

---

*Each punch/printer pair, sharing a buffer, has a common indicator.

**The condition of each indicator is shown by a light on the appropriate console. Since the operator is working close to the units he uses, he can quickly check any unit which remains "unavailable". The simplicity of this method of indicator usage contrasts with the use of several "status bits" to monitor even simple units in, for example, the Control Data 3200. The time-sharing system, after all, needs to know only whether or not the character transfer can be made.

The possibility that a program cannot continue will only arise when the program requests transfer of a character. The indicator of the unit must be examined before the transfer itself is requested. If

$$I \wedge RU(n) = 0, \quad \ldots\ldots\ldots\ldots (7.1)$$

where I is a word showing the whole set of indicators, and

RU(n) (RU: Requested Unit) has one non-zero bit showing the position of the units' indicator,

then the program can proceed. Otherwise, the processor must switch to another program.

A record must be kept of those peripherals requested but found unavailable. This record UP, (Unavailable Peripherals), must be updated if

$$I \wedge RU(n) \neq 0,$$

i.e. the operation

$$UP' = UP \vee RU(n) \quad \ldots\ldots\ldots\ldots (7.2)$$

must be performed. Until a wanted peripheral once again becomes available, $(UP \neq I)^*$ will remain zero. The operation

$$UP \neq I = 0? \quad \ldots\ldots\ldots\ldots (7.3)$$

can be used to detect the change in the status of a wanted unit. Following the change of status, UP should again be

---

*$\neq$: Logical exclusive OR

updated by the operation

$$UP' = UP \wedge I \quad \ldots\ldots\ldots\ldots \quad (7.4)$$

At any time, therefore, UP will show only those units which have been requested and are still unavailable.

Since each CIRRUS machine-code instruction was built up from micro-instructions, it was possible to incorporate the indicator check (7.1) in each input or output instruction. (The appropriate value of RU(n) is found by referencing a table held in store). The programmer can therefore use the instruction, "Transfer a character between store location m and peripheral unit n", without considering whether the unit will be available. The check (7.3) for a change in any unit's status, also micro-programmed, is performed at the beginning of every machine-code instruction. UP is held in register store. Fortunately, during the first main store cycle required to execute each machine-code instruction, the register store is not in use. UP is extracted from register store, and the comparison against I made during the "write" phase of the main store cycle. As a result, no time is lost.

In a computer not microprammed, RU(n) would probably be generated by the peripheral unit itself. UP would probably be a static register, and the masking operations (7.1) and (7.3) would no doubt be performed by special hardware.

It can be seen that a new program must be sought if the indicator check prior to making a character transfer

finds the unit unavailable. A program switch may or may not be required when an indicator change is detected in the routine check. In the latter case, a switch should not take place if the changed indicator corresponds to a unit belonging to a program whose priority is lower than that of the program currently in operation.

However, for simplicity in a small system, a standard procedure should be followed in every case. In CIRRUS, this procedure is called "program selection". It involves the scanning of programs in descending order of priority, until one is found which can proceed. The criterion determining that the j^{th} program can proceed is that

$$RI(j) \wedge UP = 0,$$

where RI(j) has non-zero bits corresponding to the indicators of all those units allotted to that particular program. The question being asked for each program is, of course: "Of those peripherals which were requested, found to be unavailable, and are still unavailable, does one belong to this program?"

To hold the values of RI(j) in the current priority order, CIRRUS has a "priority ladder" of 16 store words. For each of the 15 programs allowed in the time-sharing system, there is a corresponding 36-bit "priority word" in the ladder. Half of each word holds a value for RI(j); the other half holds a "key-word", KW(j). Since, to change

programs in CIRRUS, only a change of sequence counter address is required, each key-word is merely the address of the programs sequence counter. In some other computer, where the contents of registers need be swapped, the key-word would indicate the store positions from which register contents may be retrieved.

The sixteenth word on the ladder is a "marker", dividing "active" from "inactive" programs. A program not in use or temporarily halted by the operator is regarded as "inactive". Its priority word is placed below the marker and not considered in the program selection procedure. For most of the time, only two or three programs would in fact be active. Searching down the priority ladder will therefore be done quite quickly. If the marker is reached during this search, all "active" programs must be waiting for peripheral units. Repeated checking of the indicators will then take place until a change is detected.

Fig. 7.3. shows the time-sharing procedure diagrammatically. Implementation of the procedure in CIRRUS* has

---

*The microprograms at present used in CIRRUS use a further quantity EI (Expected State of the Indicators) which has subsequently been found unnecessary. The regular indicator check is

$$EI \neq I = 0?,$$

i.e. "Are the peripheral units as we expect them to be?" rather than

$$UP \neq I = 0?$$

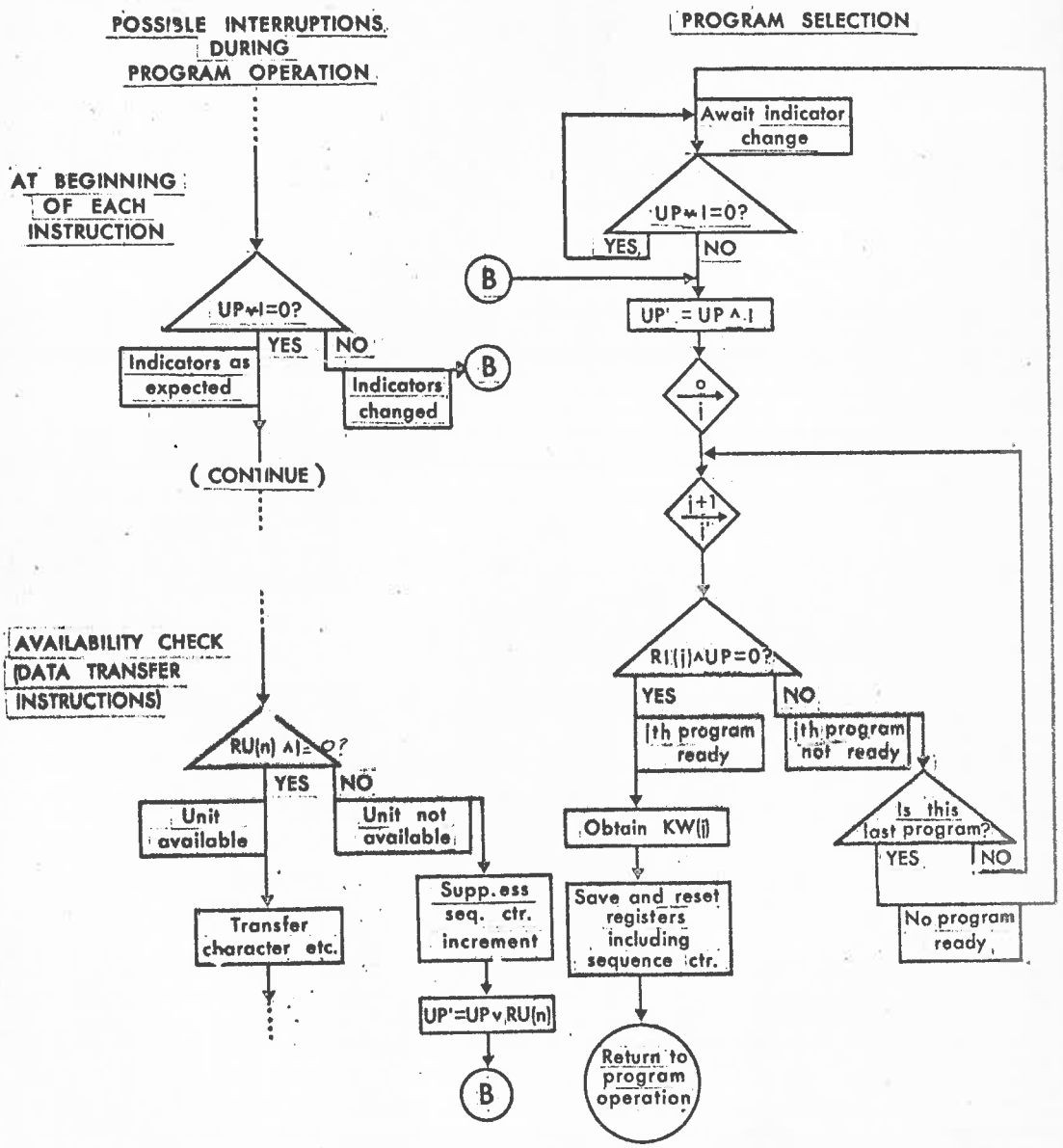


FIG. 7.3.: FLOW DIAGRAM FOR TIME-SHARING PROCEDURE
(For explanation of symbols, see text)

required no special-purpose hardware other than the peripheral indicators. About 40 micro-code instructions were used for the basic procedure, while a number of other micro-code sequences have been included to manipulate the words in the priority ladder. In a computer not micro-programmed rather more hardware would be required. Since the computer's machine-code would be much more powerful than CIRRUS' micro-code, much less program should be needed.

Simple though it is, the procedure contains all those functions necessary to share processor time between programs using only single-character peripheral units. However, in CIRRUS provision is also made for "external interrupts". The most significant bit of UP is always held as zero. Hence, when an external interruption is requested, UP ∨ I is found to be negative. Transfer is then made to a special set of micro-code subroutines which carry out the action required. At present, the external interrupt facility is used only to bring programs to read operating instructions from the inactive to the active part of the priority ladder. The facility could also be used to bring into operation other programs, for example, those which might be used in conjunction with equipment beyond the computer system itself.

Activating an instruction reading program through an external interruption is not essential. This program could in fact be kept permanently active in the highest priority

position, being selected only when a character was set in the keyboard buffer by the operator.

Although simple, the procedure is efficient. In CIRRUS, when a peripheral unit is found to be unavailable, processor time lost in switching to another program is (81 + 9n) μs, n being the priority of the new program. Time wasted in the unsuccessful input or output instruction has been included in this figure. To return to a higher-priority program following an indicator change, (40 + 9m) μs are required, where the program selected has priority m.

For any character which can be transferred immediately, only that time spent in the indicator check is lost. This time, 12μs, is the shortest period of time-sharing overhead per character (see Section 4.1.). If the character cannot be transferred immediately, then

$$121 + 9\,(m + n)\ \mu s$$

could be spent in overhead. If there were programs from four stations (the greatest number considered in Section 4.1.), up to 184μs could be spent in overhead for one character.

7.4. More Complex Peripheral Units.

A small multiprogram computer, though using paper tape units as the basic peripheral units, will quite possibly have at least one or two units of other types. The small

scientific computer system in particular could include almost any sort of peripheral unit used in a large system. Though any individual type of peripheral unit may be required, we can assume in this discussion that the total number will be very few. If there were to be many, the computer system would move out of the low-cost class, and we could afford to be more extravagant in our approach to providing time-sharing. Since there will be few, but the actual units could be any of a fairly wide variety, we should not allow the characteristics of any particular type to dictate the form of the time-sharing system. However, before any proposed time-sharing system can be accepted for implementation, it must be shown that the system could be adapted to control time-shared operation of other types of peripheral unit.

We have defined in Section 7.2. the "unit" and "block" of data transferred when using a given peripheral unit. At the machine-code level, the programmer will usually expect to be able to request transfer of a complete block, where the block would be a character from paper tape, a complete punched card, a magnetic tape record, and so on. Where buffering is practicable to the extent of a full block, the method of indicator usage described in Section 7.3.2. is directly applicable. A two-dimensional plotter, for example, could be included without modifying the system.

Although economisation on buffering is of course not the only consideration, we certainly wish to avoid the complex and costly peripheral controllers commonly used with large systems. In the smaller computer, the central processor should take over a great proportion of the peripheral unit control, but a reasonable balance must be found. For example, where the data transmission rate is high, the minimum quantity of data transferred should be one store word. For magnetic-tape units at least, consolidation of characters into complete store words would be essential. Beyond this, the extent to which buffering need be provided depends simply on the speed at which the time-sharing system allows the processor to divert from one activity to another.

As a first example of a case where buffering for less than a complete block could be provided, let us consider a fairly slow peripheral, a card reader which accepts cards "end-on". The unit of data transferred is a 12-bit character and the block a complete card of 80 such characters. If the maximum operating speed is, say, 250 c.p.m. (330 ch.p.s.), the reader can supply one character every 3 ms. If only a single character buffer is used, it can be assumed that, once a character has been set in the buffer, a delay of at least 1.5 ms can be tolerated before it must be transferred. The time-shared registers of CIRRUS are free after every machine-code instruction, that is, after 1 ms at most. We could in

fact keep in the highest priority position a special- purpose machine-code program which simply transfers a character when an indicator on the buffer shows it to be filled. In practice, a micro-program buffer transfer routine would almost certainly be preferred for greater efficiency.

This simple approach is not practicable for faster units. If faster units are to be used, construction of an elementary, general-purpose data channel should be undertaken. The data channel must provide a link between a particular buffer and a designated store location. Two registers are therefore required, D_1 to hold the buffer address, and D_2 to hold the store address. There must also be a general buffer transfer routine, almost certainly in micro-code, to which control can be given at the end of the micro-operation during which any buffer request arises. Control must then be given to a subroutine corresponding to the buffer from which the request is received. For each subroutine, a specific store address can be reserved to hold the address to or from which the present transfer is to be made. The subroutine, since it is concerned with a unique buffer, can set the buffer address directly in D_1, and take from store the current transfer address for setting in D_2. This current address can be incremented before being returned to store within the same store cycle. The actual data transfer can then be performed.

Suppose there are n such buffers with a definite priority order. The inequalities

$$\tau_j > t + T_j' \qquad (j = 1),$$

$$\tau_j > t + \sum_{i=1}^{j-1} T_i + T_j' \qquad (1 < j \leq n)$$

must hold, where

τ_j is the delay which may be tolerated before the j^{th} buffer is cleared (if input) or refilled (if output),

t is the delay before the buffer transfer routine gains control,

T_i is the time required to handle each higher priority request,

T_j' is the time spent on the j^{th} request before the buffer is cleared or refilled.

With CIRRUS, t is 6μs, the time to execute the longest micro-operation. The time to reach the correct buffer transfer subroutine would probably be about 4.5μs, and the time to handle the buffer request at least 2 store cycle-times. The ability in CIRRUS to overlap other micro-operations with those using the store should make this the maximum time needed. However, if we have magnetic tape transfers at 50 kc/s (i.e. one character per 20μs), it would be difficult to satisfy the inequality for one transfer, and

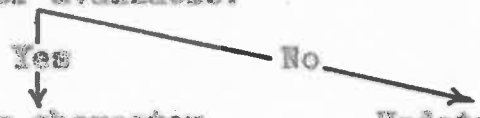
impossible for 2 simultaneous transfers. This difficulty can be overcome by providing for each tape unit two 18-bit (3 character) buffers which it uses alternately. τ_j is then that period (80μs) required by the unit to transfer four characters. For each of the two buffers there would be a separate transfer subroutine. However, each of these subroutines though setting different values in D_1 would reference the same store address for the value to be set in D_2.

The simple data channel could then handle 2 or 3 simultaneous magnetic tape transfers, and have time to spare for other units. Simultaneous operation of all units likely to be found in a small system would no doubt be possible. If desired, the capacity of the channel could be increased by providing static registers to hold the current store address for each buffer, thereby obviating the need for one of the core store references.

The time-sharing procedure at the machine-code level developed in Section 7.3. need not be altered. However, the method of setting indicators must be altered. For example, when reading from a single-character unit, each character, if available in the buffer, can be transferred immediately and the program can continue. The machine-code instruction, "read a character from paper tape" comprises:

Begin instruction ←——(Return when unit available)

Character available?



Transfer character — Update UP (See Section 7.3.2.)

Continue — Change program

When intra-block data transfers are necessary, the program requesting the transfer cannot continue until the transfer is completed. An instruction to read a record from magnetic tape must therefore comprise:

Initiate transfer (i.e. Set first and last store addresses, Put unit in motion.)

Set unit's peripheral indicator to "unavailable"

Update UP

Change program

←——(Return - when transfer completed)

The peripheral indicator for the magnetic tape unit must be set to show "unavailable" by the "READ" instruction itself. It must be set to "available", not by the buffer, but by the buffer transfer routine, following transfer of the last character of the block.

It is not necessary to develop in further detail methods of controlling these more complex input-output units. The solution put forward is oriented particularly towards CIRRUS. The approach in other cases must obviously be governed by the structure of the computer in question.

However, the time-sharing procedure developed in Section 7.3.2. and the method of using indicators should, with minor variations, still be applicable. The essential requirement is that machine-code instructions always refer to complete blocks. The method of transferring data within blocks, whether by higher priority machine-code program, by micro-program or completely by hardware, is not important.

CONCLUSIONS TO SECTION 7:

(1) For a multiprogram computer, paper tape rather than punch cards should be chosen for the main input-output medium. The grounds for this preference are as follows:

(i) Paper tape units are much lower in their initial cost, making replication of units more economical;

(ii) Being very simple in their mode of operation, paper tape units require little hardware for their control, and

(iii) The mode of operation of paper tape units is identical to that of the keyboard and typewriter likely to be used for machine-operator communication. Construction of the time-sharing mechanism is therefore simplified.

(2) If the computer system uses only paper tape units, keyboards and typewriters, a simple yet efficient time-sharing

system requiring little hardware or program can be developed. Extending the system to include a few more complex peripheral units should also be possible without incurring substantial additional expense.

8. SPACE-SHARING PROCEDURES.

In this section, the term "space" will be used in its most general sense to mean any set of facilities of which part might be required more or less continuously by a single problem program over the period during which it is in the machine. In developing space-sharing procedures, convenience for the user is of prime importance. Information demanded from the programmer beyond what he must supply to run his program on a single-program machine should be kept to a minimum. Furthermore, only information which can very easily be calculated should be required. When a program cannot be accommodated in the machine, the reasons must be indicated. The operator should then have the option of rejecting this program in favour of another.

Sharing of space will be implemented almost entirely by program. In writing space-sharing programs for a machine having only one level of store, conciseness is essential.

The nature of software necessary for space-sharing depends very much on such factors as store configuration, number and type of peripheral units and so on. The overall relevance of these factors has been discussed in earlier sections. We shall here confine ourselves more particularly to the space-sharing procedures in use with CIRRUS, together with any general remarks which are appropriate.

Three classes of "space" are to be shared in CIRRUS:

(1) Internal storage in the main and register stores,

(2) Peripheral units beyond the standard set on each console,

and (3) Internal variable programs, of which one or more* will be required for the execution of a single external problem.

8.1. An Outline of Space-sharing Procedures in CIRRUS.

It was pointed out in Section 6.3. that the operation of a single external problem program requires at least two programs within the CIRRUS time-sharing system. The first of these, a permanent "operator control" program, sets up the second variable program which begins with the "preliminary sequence".

Each program tape must be preceded by a "header tape**" which sets out the program's requirements in store space and, where necessary, lists any peripherals needed beyond the basic set or, again where necessary, requests an additional internal program or programs. This tape is read during the

---

*See Section 10.

**The header tape also includes the current set of corrections to be made to the source program (See Section 7.1.).

preliminary sequence. If the program's requirements can be met, the facilities will be allotted and the needs of any subsequent program can then be considered.

If the store is vacant, an incoming program will take up the storage it requires, beginning with the lowest address accessible to variable programs. If there is already a program or programs in store, the program will attempt to take the store space it needs immediately following the last program in store. If this space is insufficient, no attempt is made to fit the program into any other vacant store section. Instead, provided that the total of all vacant storage is sufficient for the program, the process of program relocation will be initiated, subject to certain conditions which will be stated later. During the process of relocation, those programs following the first vacant section of store will be shifted to consolidate all vacant space at the end of store. If, on the other hand, the total amount of vacant space is not adequate, the program will be delayed. In this case, the operator will be notified.

The book-keeping functions needed for space-sharing use a table called the "program catalogue". This table occupies 48 words near the head of main store. Quantities relating to the 7 variable programs are set out in an array for easy reference (Figs. 8.1. and Table B1). The "sequence counters" for each program are part of the first row of the

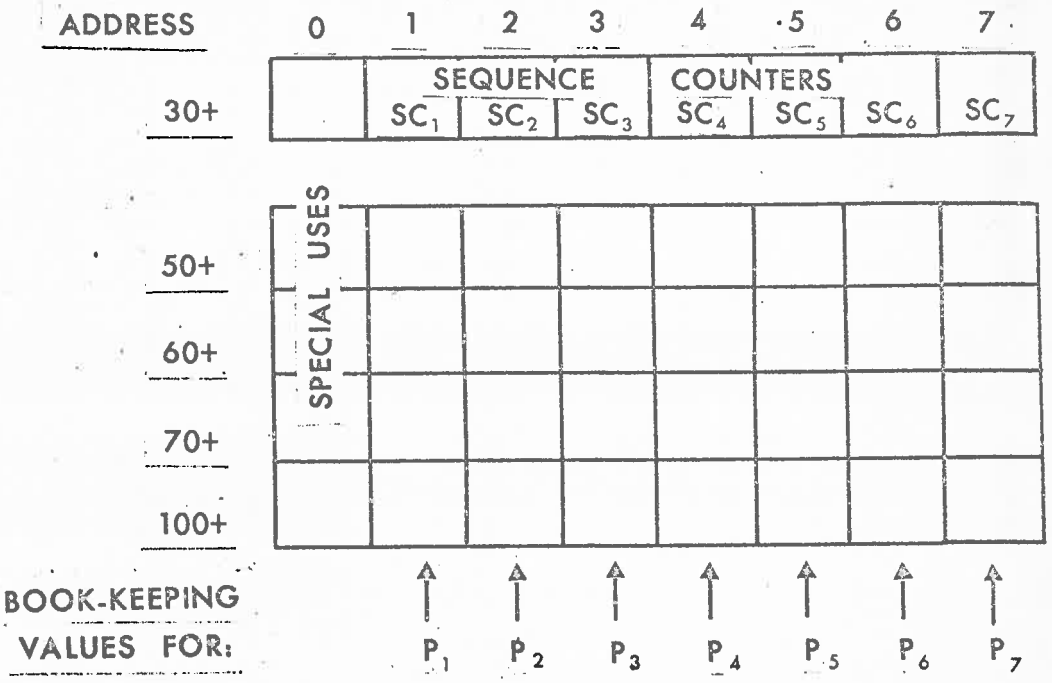


FIG. 8.1.: PROGRAM CATALOGUE

(See also Table B.1.)

array. Hence, a given quantity for any program may be found by adding the sequence counter address to the relative address of the first value in the row. Alternatively, to examine a given quantity over all programs, a particular row may be scanned.

The time-sharing mechanism treats all programs as if they were independent. (It was said earlier that they "compete for time".) However, possible interactions between programs must at times be strictly controlled. For example, while one program is referencing or modifying the catalogue, no other program can be permitted to attempt the same thing. It was decided that placing a general prohibition on interruptions over these periods would be undesirable. Two special instructions "lock" and "release", have been added to the instruction code. "Lock" enables one program to define specific conditions under which another program should be delayed. The constraints imposed by a "lock" instruction are removed by a later "release" instruction.

8.2. Tape Headings.

Program tapes used in CIRRUS fall into two categories:

(1) Source tapes prepared by a programmer,

(2) Binary object program tapes punched out by the compiler.

The form of the header which must precede the program tape is different in each case.

The standard source tape can have statements of two types: C-code which is a problem-oriented language; A-code, a hardware-oriented language. The compiler prepared by J.G. Sanderson (1963), (1964), accepts either or both (although in most programs C-code alone would be used). If both are used in a single program a declaration must be made at any point of change. The programmer must provide a header tape in a standard form which includes a job identification and any corrections to be made to the source program, as well as information on the space requirements of the program.

To show storage requirements, two numbers are punched, S and D. When a program contains only C-code, S is the number of statements and D the number of data words. These values need not be exact. Prior to compilation, the preliminary sequence allocates to the program the fairly generous estimate of

$$\text{Max}\,(7S + D + 50,\ 2000 + 18S)$$

words of storage.
The number of A-code statements (if any) is included in D. If, in some unusual case, the estimate proves inadequate, compilation will be terminated, and the programmer must prepare a new header tape.

The exact storage requirement is known when compilation is complete. If an object tape is requested,

a tape for use as a header on subsequent re-input of the object program is punched out after the main tape. This header tape shows the exact space requirements.

8.3. The Preliminary and End Sequences

A request from an operator for compilation or loading of his program causes the control program to establish a variable program which begins with the preliminary sequence. The program is "established" by:

(1) selecting a variable program sequence counter within the catalogue; registering in the catalogue that the sequence counter is tied to the console in question; and setting in the sequence counter the address of the first instruction of the preliminary sequence;

(2) allotting to the program 24 words of variable store immediately following the last program in store;

(3) raising the priority word corresponding to the selected sequence counter to the active portion of the priority ladder.

The program is then operational. The tape heading is read, and the program attempts to allocate to itself the facilities required. If these facilities are immediately available, the allocation will take a few hundred milliseconds.

During this period, any further requests from the operator will be ignored. Requests from another operator for compilation or loading will be delayed, but will take effect as soon as allocation for the first program is completed. If it is necessary, relocation of programs in store will be carried out automatically. However, if relocation is to be done, the operator will be notified. In some circumstances, a program already in the machine may have placed an inhibition on relocation. Since the incoming program would then be delayed, the operator must be given warning.

If the necessary facilities are not immediately available or cannot be made available through relocation, the operator will again be notified. He may, if he wishes, reject the program. If he does not do so, the program will remain suspended until space conditions within the machine change. Space conditions will change whenever another program completes compilation or execution, or is terminated by its operator. The incoming program will then automatically be resumed, and it can attempt once more to gain the space it requires.

The structure of the preliminary sequence is shown in outline in Fig. 8.2., and in more detail in Figs. B2. and B3. Allocations of the three types of facility needed - store, peripherals and additional sequence counters - are made separately. If any allocation is not possible, the

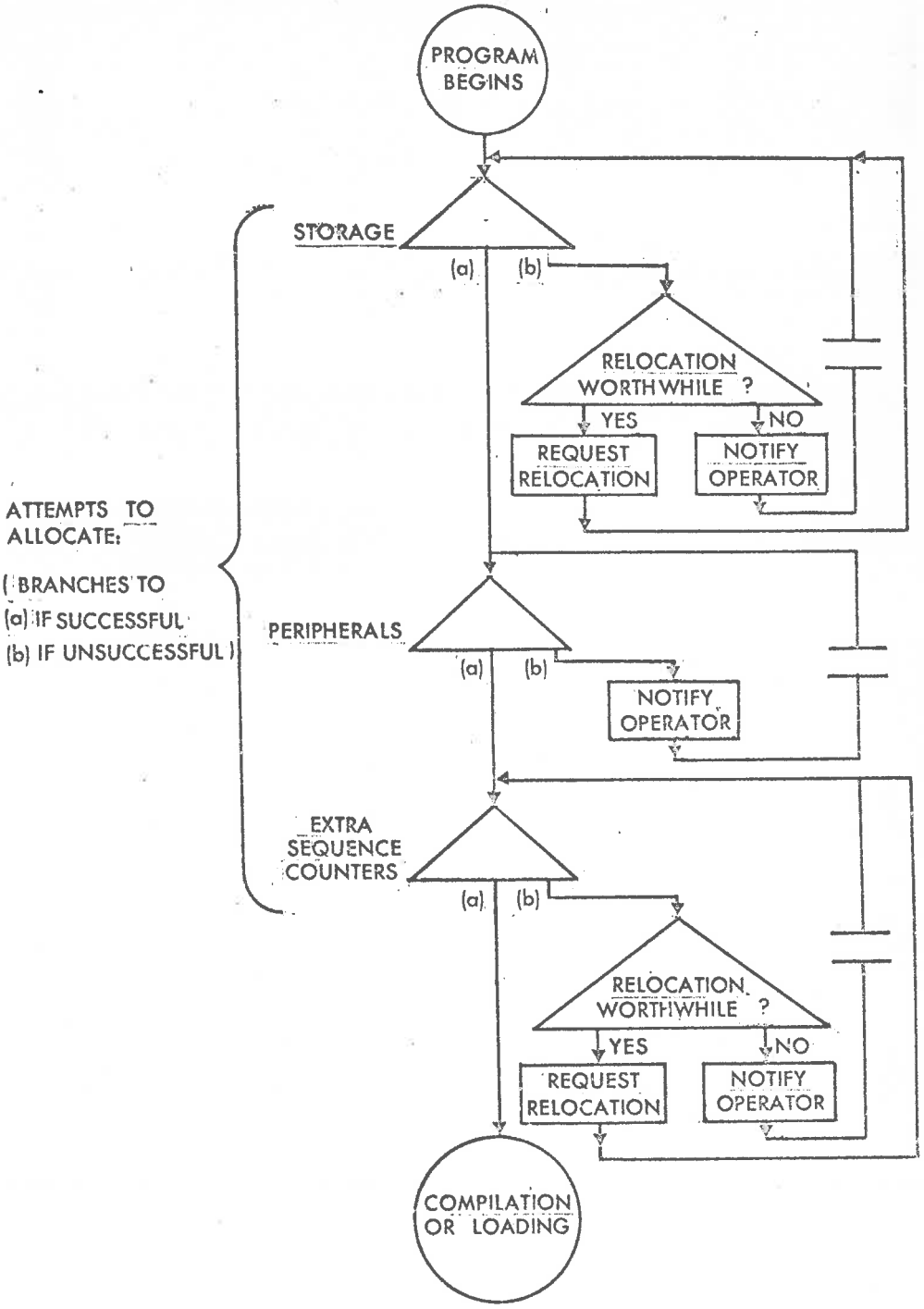


FIG. 8.2.: PRELIMINARY SEQUENCE (See also Fig. B3.)

: Point at which rejection by the operator is permitted.

operator will be told, and he may reject the program if he wishes.

The quantity of storage shown to be available for allocation is always 24 words fewer, and the number of sequence counters always one fewer, than are in fact available. Because of this, it is always possible for a program to be set up and to begin its preliminary sequence, provided only that the previous program has successfully completed allocating to itself storage, peripherals and sequence counters.

The logically final instruction in any program must be a control transfer to the end sequence (Fig. B4). There is, in fact, a special instruction, "End Program", in the machine-code to carry out this transfer. During the end sequence, facilities used by the program are returned to the pool from which another program may draw. A statement on available store-space is then made. If the operator requests that his program be rejected (not only during the preliminary sequence but also at any other time) the control program resets the sequence counter of the variable program so that the end sequence will be executed.

Since the programmer will often use the typewriter for personal output, it is undesirable to have this output cluttered with unnecessary system print-outs. Fortunately, messages to the operator are generally printed only during

either the preliminary or end sequence. Fig. 8.3. shows these messages. In the case illustrated in the figure, the operator requested loading of a program for which there was insufficient store-space. He did not reject his program, but waited until another program terminated. Relocation (possibly only of his program, occupying at that stage 24 words of store) was then performed. Loading and execution of the program then proceeded normally. In this case, the operator was playing noughts and crosses against the machine. Tiring after one game, the operator rejected the program. During the end sequence, a statement on currently available store space was printed out.

8.4. Program Relocation.

Relocation of programs is called for in the preliminary sequence if:

(1) the total amount of vacant store space is adequate for the incoming program,

but (2) there is not sufficient vacant space following the last program in store.

If a program is to be relocated at a fairly random time during its execution, then it must satisfy certain conditions. Since the structure of an object program compiled from C-code can be determined by the compiler, all

```
* INSUFFICIENT SPACE
   1444 REQUIRED
    128 AVAILABLE

*RELOCATION NEEDED


    THE GAME IS NOUGHTS AND CROSSES.
                SSW1 ON TO PLAY WITH X
                SSW2 ON FOR FIRST MOVE IN THE FIRST GAME
                SSW3 ON TO ALTERNATE FIRST MOVES
                SSW4 ON TO PRINT CURRENT SCORES
 YOU MAY CONCEDE DEFEAT BY PUSHING THE FULL STOP AT ANY OF YOUR TURNS.


 SET YOUR SWITCHES
 YOUR MOVE
 THE POSITION IS
                ...
                .O.
                X..
 YOUR MOVE
 THE POSITION IS
                .OX
                .O.
                X..
 YOUR MOVE
 THAT IS AN ILLEGAL MOVE
 THE POSITION IS
                .OX
                .O.
                XXO
 YOUR MOVE
 THE POSITION IS
                XOX
                OO.
                XXO
 YOUR MOVE
 THE GAME IS DRAWN
 PUSH 1 TO RESET SCORES FOR A NEW PLAYER, ELSE 0
 SET YOUR SWITCHES

** PROGRAM ENDS       3928 LOCATIONS AVAILABLE
```

FIG. 8.3.: SAMPLE OF TYPEWRITER OUTPUT.

* Printed during Preliminary Sequence.
** Printed during End Sequence.

object programs are designed to fulfil these conditions. On the other hand, relocation of programs written in hardware-oriented A-code cannot be permitted. Relocation of a partially compiled program is also not possible.

The compiler (which as pointed out, handles both A- and C-code) inserts instructions to prohibit, and, later, to permit relocation before and after any sequence during which relocation must not occur. The relocating program waits until all other programs will permit it to proceed; there is no question that, after relocation is requested, a program in "permit" mode will wait for others in "prohibit" mode.

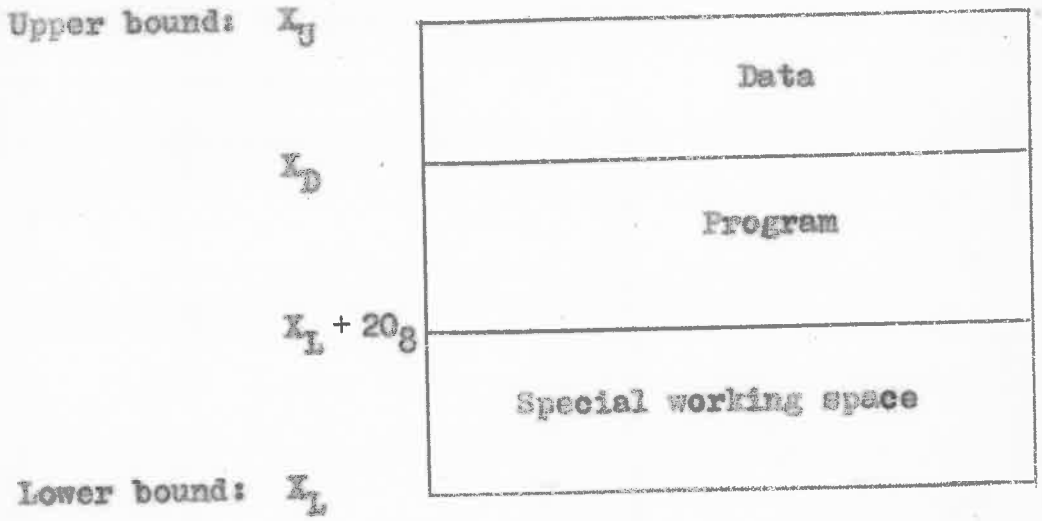
Although A-code programs are rare, one which is long-running could delay relocation for quite some time. It is suggested that such a program be preferably read into an empty store. Since the program at the head of store will never need to be shifted, "prohibit relocation" requests from this program are always ignored.

The instructions to prohibit or permit relocation are in fact the "Lock-out" and "Release" instructions mentioned in Section 8.1. These two instructions are to be discussed in detail in Section 8.6.

Relocation of variable programs is performed by a special permanent program (program 10_8) made active by any variable program requesting relocation. Once operational,

that is, when no other variable program inhibits it, the relocating program, being of higher priority and having no input or output, will inhibit all variable programs until relocation is completed. (The procedure takes only a second or so in the worst possible case.) Operation of control programs is also inhibited over this period.

All compiled programs have the structure shown below:



The values of X_L, X_D, X_U are held in the program catalogue. Suppose that this program is relocated, and after being shifted occupies addresses X'_L to X'_U. Those instructions held in the program space which refer to main store addresses* must be modified to refer to the new addresses. A quantity,

*Except those addresses in fixed store.

$(X_L - X'_L)$, must therefore be substracted. To distinguish those instructions which refer to store addresses from those referring to a peripheral unit or specifying an invariant as operand, one bit of the instruction has been reserved for a "relocation tag". This bit is set during compilation.

A program would not be relocated correctly if, at the time of relocation, an instruction referring to a main store address were held in the register store. The relocating program could not decide whether quantities held in register store were instructions or data. However, the results of all machine-code arithmetic operations can be placed in either store. Instructions are therefore most conveniently altered "in situ", there being no need to transfer them to register store. In programs compiled from C-code, this requirement is always fulfilled. On the other hand, the programmer writing in hardware-oriented A-code may have definite reasons for placing instructions in register store. It is for this reason that relocation of A-code programs is prohibited.

The program catalogue plays an important part in the overall process. The relative positions of programs in the catalogue correspond with their positions in the store. A search is made through the catalogue to find the first store section not in use, and then to find the program following this vacant section. It is probable that the vacant store section will correspond to a vacant position in the catalogue,

in which case the position within the catalogue is changed. The store limits shown for the program in the catalogue are adjusted to correspond with the store area which the program will use after relocation. The program itself is then shifted in the store. The process is repeated until the last program in the catalogue is reached. This program is the elementary program which, in its preliminary sequence, requested relocation. The relocating program moves the 24 words so far in use by this program to the addresses immediately following the preceding program in store. Relocation of programs now being completed, the relocating program terminates and becomes inactive.

8.5. Allocation of Peripherals.

It was stated in Section 6.1. that peripheral units not required by every program should form a "pool" of units from which allocations to any station may be made. Furthermore it should be possible for a programmer to use units on other consoles. Although this practice is not generally encouraged, a program should be able to use two paper tape readers (say) in a special case.

The procedure developed for CIRRUS has the flexibility necessary to meet the following requirements:

(1) The allocation must be practicable - since stations may be in separate rooms, individual units may be available only to some stations.

(2) If there is more than one unit to choose from, the one in the most suitable position should be allotted.

(3) The order of allocation should be consistent, so that the operator can predict what unit he will use and can, if he wishes, preload it. The actual allocation should always be printed as a check.

(4) The procedure must be readily amendable. Some changes will be only temporary when, for example, a unit is removed for servicing. Other changes will be more or less permanent. A new unit may be added or constraints on the allocation procedure varied.

An example will illustrate the procedure. Let us suppose that there are 3 consoles, each having one paper tape reader (unit nos. 2,5,10), 3 magnetic tape units (M_0, M_1, M_2 - unit nos. 16, 17, 20) and a plotter (PL - unit no. 21) with the room layout as shown in Fig. 8.4. A program may use any or all tape units, the plotter, or a further tape reader provided it is on an adjacent console. The programmer must state his additional requirement in the tape header. If he wants two tape units and the extra reader, he will punch

M0 M1 R1,

followed by a carriage return.

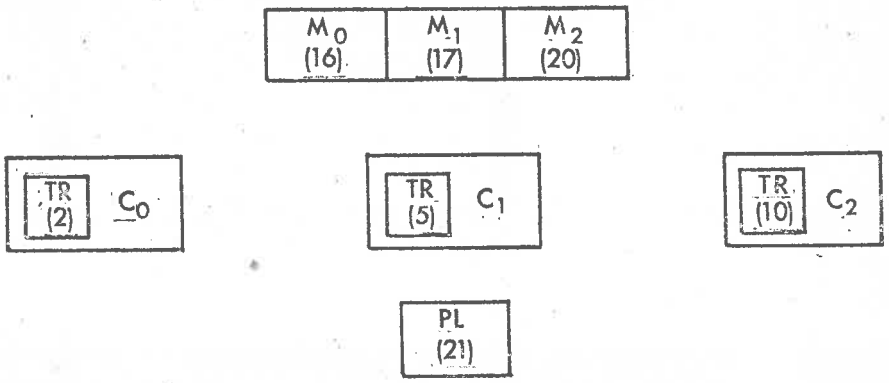


FIG. 8.4.: ROOM LAYOUT FOR THE EXAMPLE OF PERIPHERAL UNIT ALLOCATION.

UPPER HALF	LOWER HALF		
TR	77*	2	77*
TR	05	10	05
PL	21	21	21
M2	20	20	16
M1	20	2	16
M1	17	16	17
M0	20	20	16
M0	17	16	17
M0	16	17	20
MNEMONIC	C_0	C_1	C_2

PERMISSIBLE ALLOCATIONS

FIG. 8.5.: OAT
(Order of allocation table)
*No allocation permitted.

	UPPER HALF	LOWER HALF
21	PL	
20	M2	
17	M1	
16	M0	
10	TR2	
5	TR1	
2	TR0	

MNEMONIC — CURRENT ALLOCATIONS (CONSOLE NOS.)

FIG. 8.6.: CAT
(Current allocation table)

Two tables in variable store are required (Figs. 8.5. and 8.6.). The "order of allocation" table (OAT) shows, in order of preference, the allocations permissible to any console for a particular unit requested in the header. The "current allocation table" (CAT), in order on unit number, shows at any time those units available for allocation. The procedure is then as shown in Fig. 8.7. Note that, in the example, a user on Console 1 would get the nearer tape reader on Console 2 as first preference.

The initial condition of the tables is contained in a "primer" tape which sets the computer up for multiprogram operation. The addition of a unit to the system would require an alteration to this tape. If a unit is withdrawn temporarily from service, an instruction from any console would be used to set a flag in the correct position in CAT.

In writing his program, the user refers to each unit by the same mnemonic which he punches in the header. When an object program is punched, this mnemonic must be preserved, since the units used may vary on different runs. The substitution of the actual unit numbers is made immediately prior to execution.

8.6. The "Lock" and "Release" Instructions.

The time-sharing procedure considers all programs to be independent. However, no program is entirely

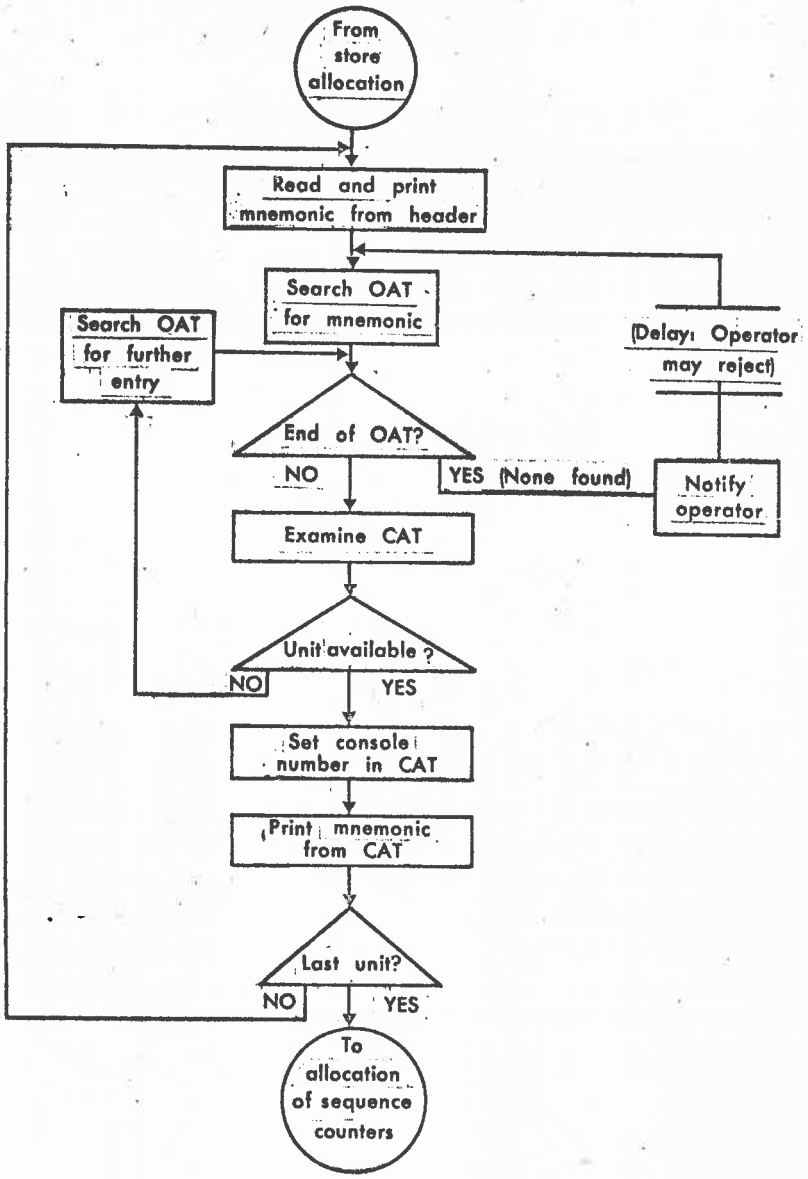


FIG. 8.7.: PERIPHERAL UNIT ALLOCATION PROCEDURE
(Part of the Preliminary Sequence - See Fig. 8.1.)

independent of others. The program catalogue, for example, is common to all programs. In some cases two programs may be directly related. Examples are: the permanent control program and the variable program which it establishes; the program requesting relocation and the relocating program itself. The splitting of a single external problem into main and subordinate programs (to be described in detail in Section 10) results in a set of internal programs which are very much dependent on one another.

A method of controlling interaction between programs must therefore be found. Often, the order of priority suffices. Any program, unless delayed by a data transfer, will inhibit the operation of lower priority programs. Equally important in CIRRUS is the use of a set of "lock-out" flags which enable one program to specify conditions under which any other program is to be delayed.

If a program is to be temporarily delayed by the action of another program, this delay must be indicated in a way which distinguishes it from delays due to other causes. Allowance has therefore been made for registering "internal delays". (Refer back to Section 7.3.2.: One bit of UP is always held non-zero. To register an internal delay for the j^{th} program, the corresponding bit of MI(j) is set non-zero. Selection of the program will be inhibited until this bit is reset to zero. Since the priority word can still be moved

on the ladder, a request by the operator to suspend the program does not conflict with the internal delay.)

Though the lock-out flags can be interrogated by conventional machine code instructions, their setting or resetting and the insertion or removal of internal delays are always carried out by two special instructions added to the machine-code specifically for this purpose. These two instructions, "Lock" and "Release", are not available to users and cannot be used in A-code programs.

The "Lock" instruction is used at the beginning of any sequence of operations which might conflict with another program. The instruction specifies those flags which must be zero before the program may proceed. If all these flags are zero, they are set non-zero and the program continues. At the conclusion of the critical sequence of operations, the flags which were set are cleared with a "Release" instruction.

If, on the other hand, the "Lock" instruction finds that any of the specified flags are already set, it registers an internal delay for the current program, and nullifies the usual sequence counter increment. When the internal delay is removed, the "Lock" instruction will be reattempted.

Any "Release" instruction not only resets the flags which it specifies but also removes all internal delays.

If it has reset the particular flags delaying a given program, that program will continue. If more than one program has been delayed by the same flag then the program of highest priority will take precedence.

Lock-out flags so far allocated are:

(1) C : Catalogue. This flag is used to prevent simultaneous reference to or modification of the catalogue by different programs.

(2) P : Preliminary. Use of this flag ensures that only one program at a time will attempt to allocate facilities to itself in the preliminary sequence.

(3) E : End. Use of this flag will be illustrated later by an example. It is set during the preliminary sequence by any program which must wait until another program ends. It is released during the end sequence in another program.

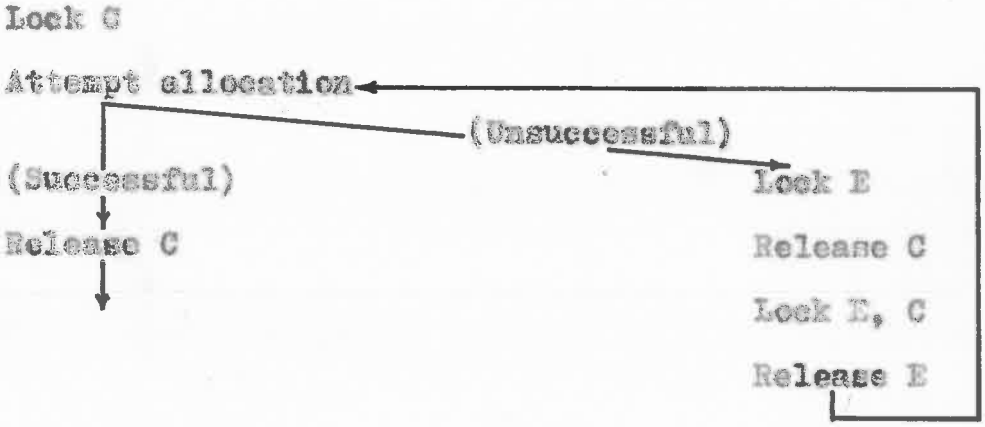
(4) R : Relocation. This flag is set when relocation is requested and reset when relocation is complete.

(5) R_i(i=1..3) : Relocation inhibits. For each problem program there is a separate relocation inhibit flag which the program sets for any period during which it must not be relocated.

(6) S_j(j=1..5) : Subordinate program flags. The uses of these flags will be explained in Section 10.

The usage of the "Lock" and "Release" instructions is best understood by considering examples. In earlier

discussion, several situations were mentioned where a program will be delayed and later resumed automatically. Let us examine one such case, in the preliminary sequence, where a needed facility is not immediately available:



During an attempt to allocate any facility, C (Catalogue) must be locked as only one program may refer to or adjust the catalogue at one time. If the allocation is unsuccessful, the program must wait. The sequence of instructions:

Lock E
Release C
Lock E,C

will then halt the program following "Release C". "Lock E" appears nowhere but in the preliminary sequence and only this one program can be in its preliminary sequence. The first two instructions are therefore obeyed, and access to the catalogue is open to any other program. However, since E has been locked by the first "Lock E", "Lock E,C" cannot

be obeyed. Although remaining in the active portion of the ladder, the program will not proceed while E is still set.

The program must wait until

(1) The operator rejects the program,

or (2) Conditions within the machine change, making it worthwhile to reattempt the allocation which was originally unsuccessful.

If the control program receives a "REJECT" request, it resets the sequence counter of the variable program so that the program will begin the end sequence. The control program includes a number of "Release" instructions which remove all internal delays, (but reset only those lock-out flags which each instruction specifies). Although the E flag is not in fact reset, the next instruction which the program must obey is no longer "Lock E,C" because its sequence counter has been altered. It is therefore free to proceed through the End Sequence, during which E is released.

It is worthwhile to reattempt the allocation for the delayed program if :

either (1) Another program completes compilation and alters its store limits shown in the catalogue,

or (2) Another program ends.

In each of these cases, E is released.

i.e.

After Compilation:

Lock C

Adjust store limits

Release C, E

In the End secuence:

Lock C

Adjust catalogue

Release C, E

It follows that if either sequence of instructions is executed in any other program, the first program mentioned can proceed through the "Lock E, C" instruction.

As a second example, use of "Lock" and "Release" instructions before and after relocation will be described. The program requesting relocation must not proceed until relocation is complete; the relocating program cannot begin until all other programs will permit it to do so. The instructions are used as follows:

Program requesting relocation	Relocating program
Make relocating program active	
Lock R	Lock R
Release R	Lock R_i for all i.
	Carry out relocation.
	Release R, R_i for all i.

Once active, the relocating program will take precedence over the program requesting relocation and will lock R. The program requesting relocation will be delayed until the R flag is released after relocation.

9. COMPUTER OPERATION

Instructions from the operator of a multiprogram computer will almost certainly be transmitted through a keyboard, the keyed instructions being read and interpreted by program. Though CIRRUS is no exception to this general rule, it is perhaps unique in at least one important respect. Rather than having one program to handle all keyboards, separate programs for each keyboard have been used*.

The structure of the CIRRUS control programs and the place of these programs within the time-sharing system will now be discussed.

9.1. Conditions to be Fulfilled.

The procedures chosen to accept and implement instructions from operators should satisfy the following conditions :

(1) It must be impossible for an instruction from an operator to have any effect on a program other than the one to which he refers.

Allowing only one program to be operated from each keyboard (Section 6.1.) helps toward satisfying this condition.

---

*Again, only one sequence of instructions is stored and shared by the separate control programs.

(2) It is important that the multiprogram computer be no more difficult to operate than a comparable single program machine.

Indeed, because of the likelihood of greater operator-machine communication, operation must be as simple as possible. Where possible, the machine should actually assist the operator.

(3) Provision must be made for two classes of message from the operator.

These may be defined as "public" or "private" messages. The former constitute the standard repertoire of operating instructions available to all operators. The latter would be provided for by programmers to be used in their own programs. Since it is undesirable to have two separate sets of keys, the character corresponding to a particular key could be read either by a control program, or by the problem program. For example, a decimal digit might be part of an address in an operating instruction, (e.g. "dump store from . . ."), or part of a value being read as data by the problem program.

The keyboard must therefore be usable for either class of message without confusion or extra work on the part of the operator.

(4) The operator should be able to transmit characters without fear that the control program might not function rapidly enough to receive them.

(5) The operator should be told if an instruction cannot be carried out.

In practice, it should be possible to execute without delay all instructions other than requests for a program to be compiled or loaded. The question of fulfilling this condition has therefore been covered in Section 8. On the other hand, notification to the operator should preferably not be given when an instruction is executed unless there could be some doubt in the operator's mind. Since the operator will probably use the typewriter for some of his own output, it would be foolish to clutter this output with unnecessary print-outs.

(6) Reasonable provision should be made against errors.

The accidental pressing of a key when there is no corresponding program in the machine, the pressing of a "start" key when the program is running or of a "stop" key when the program is stopped, and so on, should have no worse effect in the multiprogram than in the single program machine.

(7) It is important to be able to add further instructions to the repertoire without difficulty.

Many useful applications of a multiprogram computer will not be foreseen at the time of its design.

(8) To conserve valuable store space, control programs must be concise.

The use of separate control programs rather than a single program has, in the author's view, contributed significantly towards satisfying conditions (1), (3) and (8) above, for reasons which will be explained in Section 9.2. The fulfilment of Conditions (2), (5), (6) and (7) depends largely on the structure of the control programs themselves. They will be described in Section 9.3. The question of satisfying Condition (4) will be discussed in Section 9.4.

9.2. Status of Control Programs in the Time-sharing System.

Incorporating the time-shared operation of control programs within the general time-sharing scheme is simple enough. Given the procedure described in Section 7.3.2., a single control program could be used, retaining the

position of highest priority and being permanently ready to accept a character when any key is pressed. However, separate control programs for each keyboard are used in CIRRUS.

Each of these control programs is normally inactive (i.e. its priority word is below the marker in the ladder) and becomes active only when an instruction is to be read. The first key pressed in any instruction must be what is termed an "operative" key. As well as setting the corresponding character in the keyboard buffer, pressing an operative key also sets an "external interrupt" (Section 7.3.2.). When the interruption is detected, the priority word of the control program is raised to a position in the active portion of the priority ladder which gives it a priority higher than all variable programs. It will therefore read the operative character immediately.

In several cases, the operative character gives the whole instruction. After the character has been read the instruction is executed and the control program terminates. Its priority word is automatically returned to the bottom of the ladder. For a longer instruction, the control program will remain active until all characters have been read.

Since the character produced by pressing an operative key is always read at the beginning of the control program

the same character may be used with another but non-operative key. The entire character set could therefore still be available for use in private messages.

Use of separate control programs rather than a single program has been found to give several advantages. Each program must consider only one keyboard and the problem which originated from that keyboard. It was thus easier to fulfil conditions (1), that there be no interference with other programs, and (8), that storage for control programs be kept small.

The requirement that private messages be distinguished from public messages (Condition (3)) is also satisfied. If the control program is inactive, any keyed characters are read by the problem program - provided only that this program is in a position to read them. The programmer must include (as he would do in any other computer) a print-out to show that input from the keyboard has been accepted. The operator may, if he wishes, interpose operating instructions during input of private messages. Since each public message must begin with an operative character, the control program will be brought into operation. Following complete transmission of the instruction, the control program becomes inactive. Subsequent characters will again be read by the problem program.

9.3. Structure of the Control Programs.

The author has divided operating instructions into two categories. In the first category are simple instructions needed in "normal" operation. In the second are instructions which may be more complex. The operator should be able to begin his program, stop it temporarily, resume it or reject it by pressing only a single key in each case. Provision for "Reject" as well as "Stop" is essential in a multiprogram machine. If the operator decides that his program can be terminated, he must reject it, thereby releasing the facilities which the program has been using. For each of these instructions, only the operative key is needed.

With the exception of the "Reject" instruction, none of these instructions require a print-out to be made. After "Reject", the total quantity of available storage is stated (Section 8.3.).

For an instruction requiring several characters, each character is printed as it is keyed. The operator can therefore check its correctness.

The structure of a CIRRUS control program is shown in Fig. 9.1.* Because there is no fixed correspondence between control programs and variable programs, the control

---

*(For more detail, see Fig. B 1)

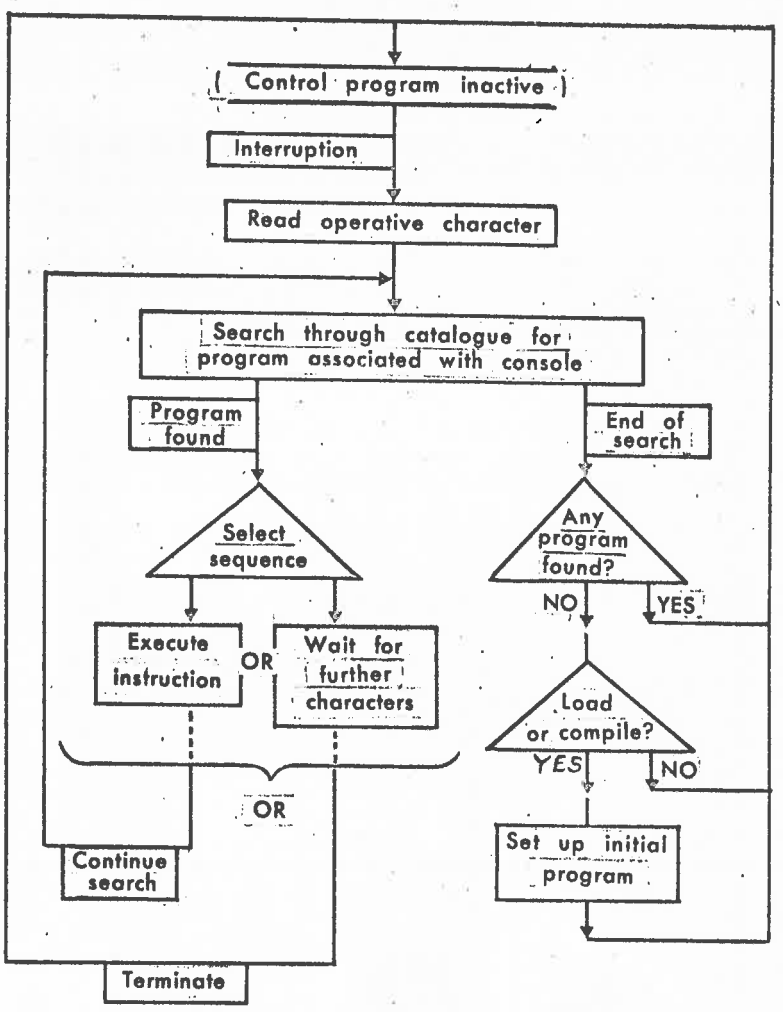


FIG. 9.1.: FLOW DIAGRAM FOR A CIRRUS CONTROL PROGRAM

program must search through the program catalogue for programs associated with it. On finding a variable program, the control program branches to a sequence chosen on the value of the operative character which has been read. If the operative character contains the whole instruction, the chosen sequence executes the instruction. If the operative character is one which begins an instruction consisting of several characters, the sequence waits to read the characters before implementing the instruction.

It was pointed out in Section 6.3. that, in allowing variable programsto be allocated to control programs in a quite flexible way, the intention was that the programmer should be able to divide his problem into a master and subordinate programs.

When the control program searches through the catalogue, the master program is always found first. Some instructions (for example, "Reject") need be implemented only on the master program. Other instructions ("Stop" and "Resume", for example,) must be applied also to the subordinate programs. After implementing an instruction of this second group on the master program, the search is continued until the end of the catalogue is reached.

In all cases, the control program becomes inactive after completing execution of the instruction. Note in Fig. 9.1. that the whole program constitutes a closed loop.

Whenever it becomes active after an operative key is pressed, the program begins at the same point.

If the control program finds no variable program associated with it, the operative character is checked. If compilation or loading has been requested, a variable program is "established" (see Section 8.3.). If any other request has been made, the control program terminates, that is, the request is ignored.

The present directory allows for 24 operative characters, of which 13 are now used (see Appendix B 5). This directory could be expanded to cover the whole set of 64 characters. Any operative character could also be followed by secondary characters to indicate sub-operations. Expansion of the instruction repertoire therefore presents no problems. The only limitation is the quantity of storage which can be spared for holding programs to execute further instructions.

9.4. Time-sharing Requirement for the Control Programs.

To fulfil Condition (4) of Section 9.1., each control program must be able to accept successive characters more rapidly than any operator can transmit them. If, when all control programs are in use simultaneously the program of lowest priority can fulfil the condition, all other

programs can do so.

Suppose that there are n keyboards and that the minimum possible time between successive characters from one keyboard is T sec. After any character has been set in the buffer, it must be removed within this period T. If more than one control program is active the program of lowest priority receives only the residue of time not required by higher priority programs.

The inequality

$$(n-1)t + t' + \varepsilon < T \quad \ldots\ldots\ldots (9.1)$$

must therefore be satisfied, where

t is the maximum amount of time any control program would use in any period T,

t' is the maximum period needed before a character is removed from the buffer,

and ε represents time which could be lost in setting up control programs, or in inter-program switching.

We can take .2 sec. as a reasonable value for T. ε can therefore be neglected since the time to set up a control program is only a few hundred microseconds and program switching times are even smaller.

For the present there is no problem. The only instructions requiring more than 20 or 30 msec. to complete use the typewriter. While typing, a program uses only a few milliseconds in any period of 200 msec. However, the

position must be reconsidered for each instruction added.

If microprograms are included to control "within-block" data transfers as suggested in Section 7.4., their effect must also be taken into account. Since these microprograms would use frequent but small quantities of time (of the order of microseconds rather than milliseconds), their effect is most easily considered by assuming that the values of t and t' could be increased by a certain factor. Suppose that, if all possible data transfers at the microprogram level were to occur simultaneously, a fraction p of total time would be needed. The inequality (9.1) then becomes

$$\frac{(n-1)\ t + t'}{(1-p)} < T \quad \ldots\ldots\ldots \quad (9.2.)$$

For example, if $n = 3$, $p = .3$, $T = .2$ sec, then

$$2t + t' < .14$$

must be satisfied.

Though public messages can therefore be transmitted at a maximum keying speed, the same may not always be true for private messages. Problem programs will certainly receive priority over the variable programs while reading from a keyboard (see Section 11.3). Nevertheless, the programmer is advised to print each character in a private message as it is received.

10. "WITHIN PROGRAM" TIME-SHARING.

The CIRRUS system provides for time-sharing of processes within a single external problem program by allowing the problem to be divided into a master and as many as five subordinate programs, all of which the time-sharing procedure treats as independent programs. These programs will hold separate successive positions in the program catalogues, the master program occupying the first position. However, they will be shown in the catalogue as having the same store space. Information may therefore be transferred between them.

The original intention was to provide a facility equivalent to that given by the BUFFER IN.... and IF UNIT.... statements available in some versions of FORTRAN (e.g. Control Data 3600 FORTRAN (Control Data, 1964)). However, the facility in CIRRUS is more flexible, and should be valuable for a very much wider range of applications.

A subordinate program may consist of any operations thought desirable by the programmer. For example, the user could transmit private messages through the keyboard in parallel with other operation. Alternatively he might use a subordinate program to read data from the keyboard, to edit the data and to print on the typewriter while the main program computes and punches.

A number of operators, each independently using separate keyboards, might participate in a single problem. For each keyboard there would be a separate subordinate program, each being supervised by the single master program.

A subordinate program should be used only for an activity in which there was input or output. Indeed, there is no point in using one except for an activity during which there can be delays. A single subordinate program can contain several branches, the choice of branch being made according to a value set usually by the master program or perhaps by itself or by another subordinate program. However, if activities are to occur simultaneously, they must be in separate programs.

In earlier sections it was shown how the programmer can request additional sequence counters for subordinate programs in the heading to his program tape. Their allocation in the preliminary sequence and their release in the end sequence have also been described. The allowance made for subordinate programs by the control programs has been covered in Section 9.

In this section, an example will be given to show how the programmer can control interaction between the separate programs. Since the programs receive processor time independently of each other, the programmer must be able to specify what might be called "points of

reconciliation". At these points, one program would check that another program (or other programs) had reached a stage enabling it to continue further. If the first program were forced to delay, it must receive no more time until the required point had been reached in the other program. It must then be resumed automatically. The "Lock" and "Release" instructions described in Section 8.6. are used to fulfil this purpose.

Given a problem with the normal sequence of operations:

Input $\bar{x}_1$
Compute with $\bar{x}_1$ to get $\bar{y}_1$
Output $\bar{y}_1$
Input $\bar{x}_2$
.....
Input $\bar{x}_n$
Compute with $\bar{x}_n$ to get $\bar{y}_n$
Output $\bar{y}_n$
.....

(where $\bar{x}_n$, $\bar{y}_n$ are input, output vectors), the aim is to make:

Output of $\bar{y}_{n-1}$,
Computation with $\bar{x}_n$ to get $\bar{y}_n$,
and Input of $\bar{x}_{n+1}$,

take place simultaneously. Buffer areas in core must be

provided for the input and output vectors. These buffers will be referred to hereafter as buffer A and buffer B respectively. Two subordinate programs, one to handle input and one output, and two corresponding lock-out flags, S_1 and S_2, are required. Overall control is exerted through the master program which functions as shown below. (Initial and final conditions are omitted for simplicity.)

Master program:

(A): (Provided output buffer is emptied)

Transfer $\bar{y}_{n-1}$ to output buffer

Start output program

(Provided input buffer is filled)

Transfer $\bar{x}_n$ from input buffer

Start input program

Compute with $\bar{x}_n$ to get $\bar{y}_n$

Return to (A)

The master and subordinate programs would be compiled as below:

Main Program	Subordinate Programs	
P_1	P_2 (Input)	P_3 (Output)
→Lock S_2	→Lock S_1	→Lock S_2
Transfer to buffer B	Input to buffer A	Output from buffer B
Resume P_3	Release S_1	Release S_2
Release S_2	Suspend	Suspend
Lock S_1 (b)		
Transfer from buffer A		
Resume P_2		
Release S_1 (a)		
COMPUTATION		

Notes:

(1) The operations Lock, Release, Suspend and Resume are performed by single instructions already in the instruction-code. (See Appendix B 4.)

(2) Programs P_2 and P_3 will have priorities above P_1 (see Section 11.3). Hence, after S_1 is released by P_1 at point (a), the Lock S_1 instruction in P_2 will always be executed before the Lock S_1 instruction at point (b) in P_1.

The actual form of the source language statements to be used has not yet been decided. Because of the restricted storage available for holding the compiler, the programmer

may have to delineate his buffer areas by specifying separate variables, and make the buffer transfers with standard arithmetic statements. However, compilation of those instructions needed to control the separate programs is quite straightforward. The subordinate programs will be set out as "procedures" (the CIRRUS equivalent of FORTRAN subroutines), with some indication to show that they will function internally as separate programs. Beyond this indication, a "WAIT ....." statement for the master program and a special "RETURN" statement for the subordinate programs are all that is necessary.

The source program for the earlier example is given below. Since CIRRUS C-code is not described in this thesis, an equivalent in FORTRAN statements is given.

SOURCE PROGRAM	COMPILES TO
n WAIT WRITE*	→ Lock S_2
(Transfer to Buffer B)	
CALL WRITE	→ Resume P_3 Release S_2
WAIT READ*	→ Lock S_1
(Transfer from Buffer A)	
CALL READ	→ Resume P_2 Release S_1
(Computation)	
GO TO n	
.....	
END	
SUBROUTINE (SP_1) READ	
.....	
RETURN	→ Release S_1 Suspend Jump to head of subroutine
END	
SUBROUTINE (SP_2) WRITE	
.....	
RETURN	→ Release S_2 Suspend Jump to head of subroutine
END	
.....	

Notes:

(1) The subroutines to constitute subordinate programs are compiled as normal subroutines, except that:

---

*It must be remembered that the initial conditions were disregarded. In making the first CALL to each subroutine, the WAIT... statement must be bypassed.

(a) The sequence counters chosen are set with the address of the first instruction,

(b) The first instruction compiled is the appropriate "Lock" instruction.

(c) Neither the "CALL" nor "RETURN" statements result in a control transfer as would be the case for normal subroutines.

(2) Instead of using "WAIT..." statements, the programmer could use an "IF...." statement to check the lock-out flags.

11. MAXIMISING WORK-OUTPUT.

The structure of the CIRRUS multiprogram system and the time- and space-sharing techniques used therein have now been fully described. The primary intention throughout development was to produce for a reasonable cost a computer most suited to the environment in which it would operate. Whether the chosen structure allows the most efficient utilization of the processor has, so far, largely been neglected.

Perhaps the most significant feature of the system is the provision for separate and independent operating stations. In some systems, selection or "scheduling" of jobs in appropriate "mixes" for execution is considered fundamental for efficient processor utilization. At first glance, the independence of the operating stations in CIRRUS would appear to prevent any useful scheduling.

To simplify the time-sharing procedure in CIRRUS, programs are allowed to control their peripheral units directly. It is generally accepted that data transfers should take priority over computation. Furthermore, some peripheral units function extremely inefficiently if they come to a complete stop and have to be restarted. Obviously, some method ensuring priority to a program

during input or output must be found.

These matters and others affecting the efficiency of the system will be discussed in this section.

11.1. Principles.

To increase the rate at which work is done by the processor, there are three generally accepted principles which may be followed:

(1) The number of separate activities available for the processor to work on should be kept as high as possible.

(2) The activities available for the processor at any given time should be as far as possible a balanced mixture, some activities utilizing the processor heavily and others only occasionally.

(3) Of the activities available to it, the processor should give priority to those whose demand for time is lowest.

The term "activities", rather than "programs", has been used in the above statements deliberately. From an examination, later in this section, of practical methods of improving efficiency by following one or other of these principles, it will be seen that by "activity" could be meant a program, a part of a program or a sequence of programs run in a particular way.

The expressions calculated in Section 4.1 for limits of the improvement factor I and plotted in Figs. 4.1, 4.2, 4.3, show the importance of principle (1) above. Unless the value of D for the work-load is close to 1, I must increase very significantly for each increase in n. For example, where D = .3 and $d_i = D$ for all i:

$I_L = 1.6, \quad I_U = 2 \quad$ if n = 2,

$I_L = 2.1, \quad I_U = 3 \quad$ if n = 3,

$I_L = 2.4, \quad I_U = 3.3 \quad$ if n = 4.

The number of activities available to time-share a system with only core store depends largely on the store size and the efficiency of the method by which storage is shared. The question of selecting programs to make the most effective use of available storage will be discussed in Section 11.2 .

In calculating the lower limits for I in the preceding paragraph, it was assumed that the values of d_i were equal for all i. That is, the very large work load was assumed to be divided at random between stations. If steps are taken to follow the second and third principles stated above, this assumption can no longer be made.

Let us suppose that, rather than considering the work-load to be supplied through n stations, we regard it as being supplied in n separate streams, $s_1 \ldots s_n$, where the work making up one stream may originate from different

stations. However, suppose that work in stream s_i always receives priority over work in stream s_{i+1}. Stream s_i is then, by definition, composed of all work having priority i. Let processor utilization by the work making up stream s_i again be d_i.

The expressions found for I_U and I_L in Section 4.1 are still valid. The value for I_U is unchanged by varying the d_i, but I_L increases if d_1, for example, decreases. The relationship between the d_i is a complex one. Considering only the case where n = 2, one could certainly divide a very large work-load into two equal parts, one intended to constitute stream s_1, and the other stream s_2, so that

$$\frac{1}{2}(d_1 + d_2) = D .$$

However, the work in stream s_1 would be processed more rapidly than the work in stream s_2. To continue simultaneous processing until the whole work-load had been processed, work from the second part would have to be transferred to the first. One can assume only that, if d_1 is decreased, another d_j must increase.

Following the third principle obviously tends to reduce d_1 relative to d_2 ....d_n, d_2 relative to d_3....d_n, and so on. Predicting the effect on the improvement factor itself is difficult, partly because of the complexity of the relationship between the various d_i but more importantly because the expression in terms of d_i represents only a

lower limit and not the improvement factor iself. One can assume only that any measure which contributes to an increase in I_L will probably increase I. Hence, any measure which tends to concentrate lower-demand activities in higher priority streams should be worthwhile. However, one must resort to simulation to obtain reliable estimates of the effect on the improvement factor itself. Results from a simulation study of methods for priority adjustment will be discussed in Section 11.3 .

Following principle (2) in conjunction with principle (3) would also increase the value of I_L. Consider, for example, the case where n = 2. Suppose that, instead of supplying the work-load in a completely random way, attempts are made to couple low- with high-demand activities. Processor utilization by the work in the higher-priority stream would decrease, that is, d_1 would decrease and d_2 increase. Steps which might be taken to balance low- and high-demand activities will be discussed in Section 11.2 .

11.2 Standard Operating Practices.

The first and second principles states previously suggest that some selection should be made of those problems which are to occupy the machine together. Selection or "scheduling" procedures can be quite elaborate. In STRETCH (Codd, 1960) and the Honeywell-800 (Honeywell,

1961), for example, space and time requirements of jobs waiting to be run are examined and the jobs combined into appropriate "mixes" for execution.

The independence of operating stations in CIRRUS eliminates the possibility of scheduling in this way. However, it is doubtful whether what might be termed "static scheduling" would be of much value for a small scientific computer. If scheduling is to be effective, the storage requirement, execution time and processor utilization of each program must be known fairly accurately. Hence, scheduling is of most value where the work-load contains a high proportion of recurring jobs.

The work-load of a small scientific computer varies considerably from day to day. Many programs are run only once or twice before being discarded. Though the storage requirement of each program can be estimated fairly accurately in advance or found exactly when the program is compiled, the extent to which the processor would be utilized would be difficult to predict accurately and the execution time even more difficult. If the forecast of execution time for a program were in error by a factor of 2 (as could quite often happen), a prepared schedule would become meaningless.

If a scheduling procedure is to be effective for this sort of work-load, it must be "dynamic". In other

words, selection of the appropriate job should be made only when another job is needed, and made according to the current conditions. The author feels that, if certain standard operating practices are laid down, most of the benefit to be gained from scheduling can in fact be gained without jeopardising the independence of the operating stations.

The Adelaide University's IBM 1620 has been operated during normal working hours by a trained operator, and outside those hours by the user-programmers themselves. The author has suggested earlier that one (and occasionally two*) of the three CIRRUS stations be manned by the full-time operator. His task would be to maintain a continuous flow of work to the machine. The remaining console or consoles would be available to user-programmers.

The main operator should keep programs active from his station or stations for as much as 90% of total time. The user-programmer would have his program active for perhaps as little as 10-20% of the time. Hence, there would be a natural bias towards the coupling of low- and high-demand activities.

---

*Since one program only can be run from each station, a single operator should be able to look after two stations simultaneously without confusion.

In a number of articles on multiprogram time-sharing, mention has been made of a need to schedule "I.O.-limited" and "compute-limited" jobs for execution together. Using these terms to describe whole jobs is, however, unjustified, for any program (unless intended purely for data conversion) is comprised of successive periods over which it is alternately I.O.- and compute-limited. In the author's view, processor utilization by any single program is more likely to be near the median utilization figure for all programs than to be very high or very low. Furthermore, predicting the degree of utilization accurately would be difficult. Hence, the assigning of contrasting functions to different operating stations as suggested should be very much more effective than attempting to schedule jobs of low- and high-demand.

The task of a full-time operator is hardly onerous. He is, nevertheless, obliged to be in attendance almost continuously and can therefore contribute to efficient processor utilization by selecting on each occasion the job most suited for the current conditions. His selection need be based solely on the amount of vacant store and the storage requirements of those programs which he himself has waiting to be run.

To make this selection, the operator needs:

(1) A statement of the storage required by each job;

(2) when each job ends, a statement of vacant space and number of programs still operating;

(3) the ability to reject a job which, when loading is attempted, proves too large.*

If, in the three-station system, two programs were already in the store, the operator would select from his stack the program whose storage requirement was largest but which would still fit into the machine. If one program or no program were already in the store, he might select a program whose requirement was small. His selection should be influenced by the composition of his job stack at the time.

---

*It has been shown earlier that:

(1) The header tape required by the CIRRUS system includes a statement of storage requirement for the program. This information can also be written on the Job Request supplied to the operator.

(2) The quantity of vacant storage is printed out during each program's end sequence. The number of programs still in operation is not stated at present.

(3) Rejection of jobs is possible.

User-programmers having two or more jobs to run can also select the most suitable of those jobs. The result should be extremely efficient use of the storage available.

The procedures suggested in this section, that different stations be assigned for different types of activity and that operators base their selection of programs solely on storage requirements, may seem extremely primitive compared to the elaborate scheduling procedures used in other systems. Nevertheless, the measures suggested should be almost as effective as one would expect from any very elaborate procedure - given the nature of the work-load to be processed. Most important of all, the independence of separate operators is not affected.

11.3 Priority Adjustment at the Machine-code Level.

Section 11.2 showed the extent to which the first and second principles stated in Section 11.1 can be adopted in a small scientific computer having independent operating stations. In this section, we shall examine ways in which the third principle - that higher priorities should be given to lower demand activities - can be satisfied. It must first be decided what is to constitute an "activity" for the purpose of priority assessment. It might be suggested, for example, that, since the demand for

processor time from some stations will be consistently less than from others, work from these stations should receive priority.

Suppose that a single station has exclusive use of the processor. Total time may then be divided into segments, each of which fall in one of the following categories:

(1) An operator delay, during which the demand for time is zero,

(2) A sequence of computation without input or output, during which the demand for time is continuous, and

(3) A sequence of input or output with one peripheral unit, during which the demand for time is intermittent but fairly regular.

Processor utilization during any of these "activities" is both consistent and predictable. The activities should therefore constitute the basis for priority assessment.

The suggestion that priority might be given to a station from which the overall demand is low can therefore be disposed of. To give priority in a way which minimises utilization in the highest priority stream, priority should, in theory, be given to any station during an operator delay. However, during such delays, no time is required and no priority adjustment need be made.

The aim has been to develop for CIRRUS a procedure which would give each program higher priority during input or output, yet would require no special information either from the programmer or the operator. Advantage has been taken of the fact that almost all programs would be compiled from problem-oriented language*. For practical purposes, an "input-output sequence" has been defined as those operations resulting from a single READ or WRITE statement in source language, and a "computation sequence" as those operations between two successive input-output sequences.

A variation in demand for time by one program (and a probable requirement for priority readjustment) arises at the moment of transition from one type of sequence to the other. The logically first and logically final instructions compiled for each input or output statement are therefore control transfers to a priority adjustment subroutine. This subroutine places in the program catalogue a "demand figure" for the current program (.01 if the statement were to type, .1 on beginning to punch, .5 on beginning to read, or 1 on ending input or output,) and then adjusts priorities according to the current demand figures for all programs. Special priorities may also be introduced. For example, during the Preliminary Sequence described in Section 8.3,

---

*Since efficiency only is involved, the rare program in hardware-oriented language can be neglected.

a demand figure of zero would ensure priority over other working programs.

To establish a standard for comparison a simulation run (Run 1 - Appendix C) was made where priorities between programs were determined simply by the order of their entry into the computer. It was assumed that 3 programs were always available to time-share, that is, that there were no delays due to an operator, and no storage limitations. When simulation was discontinued the average demand over all programs (or part programs) run was 0.296, total processor utilization was .708 (of a possible .888) and the improvement factor 2.40.

This simulation model was equivalent to the theoretical model of Section 4.1 except that priority was not given to particular stations, but to programs in the order in which they began. Since priorities were still determined without considering the demand for time by the activity, this variation was probably of little significance. In the theoretical model,

$$I_U = 3$$

$$I_L = 2.09$$

for $D = .296$, $n = 3$. In developing the expression for I_L in Section 4.1, concurrency of useful work not requiring time with work requiring time was disregarded unless the former were of higher priority. The extent to which the

value of I found by simulation exceeded I_L was no doubt almost wholly due to this concurrency.

A further simulation (Run 2) was made, during which priorities were adjusted as indicated earlier. Rather surprisingly (at first glance), the improvement was disappointing: utilization was .714 (from an average demand of .290 indicating a maximum possible utilization of .870), and the improvement factor was 2.465. However, the reason was not hard to find. Each program was given lowest priority at the end of each sequence of input or output. Hence, during every particularly long computation sequence, the program soon obtained highest priority. Thenceforward, the program monopolised central processor time, thoroughly defeating the purpose of the priority adjustment. In the preceding run, there had been only a 1-in-3 chance that a program would have highest priority during a long period of computation.

This second result raised the question of what general strategy should be adopted to determine priorities between two or more programs, each engaged in computation sequences. An examination of the program sample showed that logically successive input or output statements were, in most cases, separated by only a very few other statements. In two or three programs, virtually all computation was done in a single computation sequence. The distribution of the lengths of all computation sequences in the sample is shown in Fig. 11.1 . It is likely that the extreme positive

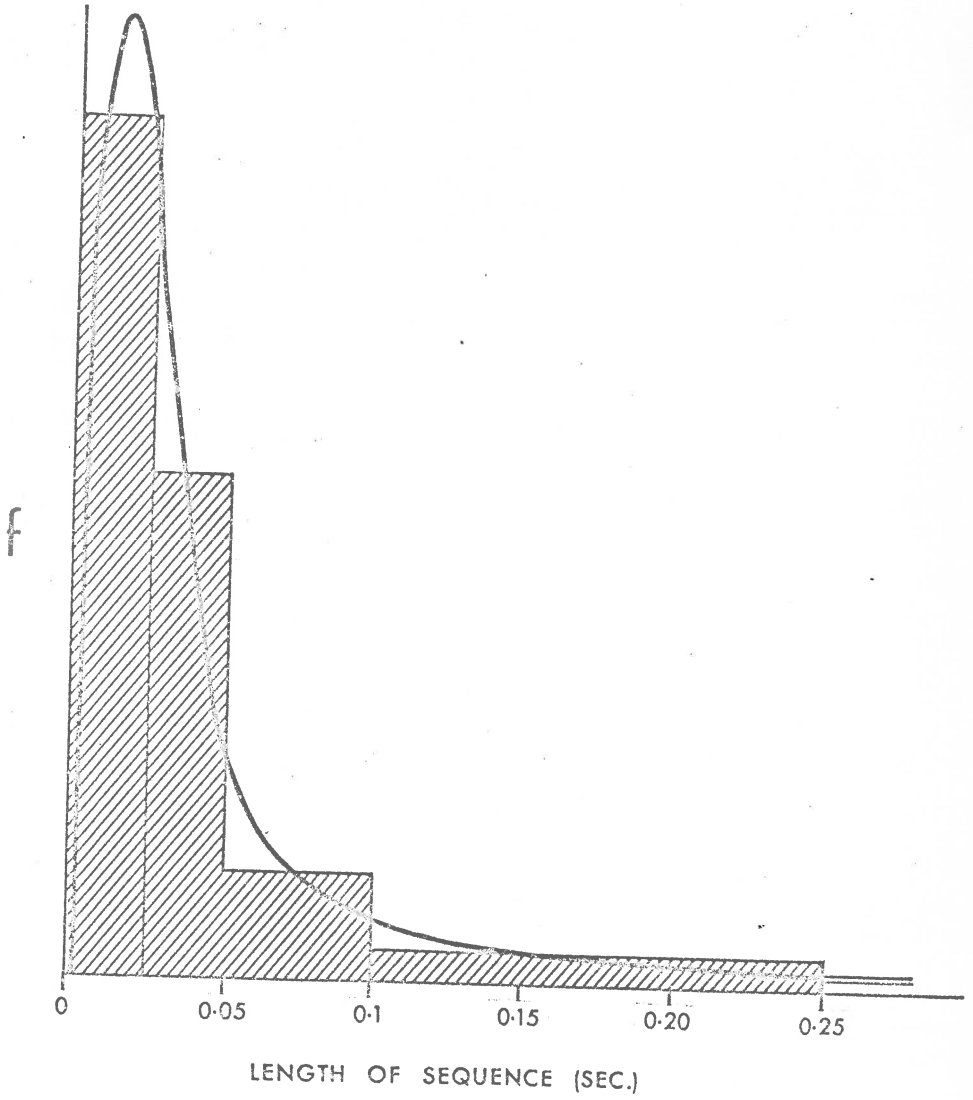


FIG. 11.1.: DISTRIBUTION OF LENGTHS OF COMPUTATION SEQUENCES IN THE SAMPLE SET OF PROGRAMS.

skewness there exhibited would hold if all programs in any work-load could be examined.

It is obviously preferable that, if there are two or more computation sequences, it is best to choose the shorter or shortest. Without any knowledge of the programs, such a choice cannot be made. Attempting to extrapolate from the past history of the program is also not practicable, because a particularly long period of computation is quite possibly an isolated one. It was therefore decided that a profitable strategy would be any one which reduced the expected time for at least one of the two or more programs to reach the end of its computation sequence.

Given the case where there are two programs A and B:- Program B is in a computation sequence at some unknown point, and Program A is about to begin a computation sequence. Nothing is known of the relative lengths of these sequences. It must be assumed that their lengths are random, chosen from the distribution of all computation sequence lengths. Therefore, the expected period of the computation sequence in program A is

$$E(T_0^A) = \bar{T},$$

where $\bar{T}$ is the mean length of all computation sequences. For the 4000 odd computation sequences of the program sample used in simulation,

$$\bar{T} = .12 \text{ sec.}$$

One might expect that, since some work has presumably been done on the computation sequence from program B, the expected length of the residue would be less than $\bar{T}$. Such is not the case. Although the lengths of the sequences in both A and B come from the same distribution, the chance of any particular event occurring (such as the beginning of a computation sequence on the other program) during any sequence in B increases with the length of the sequence.

An estimate of the expected length of the residue period of computation from B was found in the following manner. On the IBM 7090, 20 periods of computation $T_1 \ldots . T_{20}$, were selected at random from the program set, and combined to represent a single time period 0 to $\sum_{i=1}^{20} T_i$ of program B (Fig. 11.2). (The alternate periods of input-output could be ignored because we were interested only in an event occurring during computation.) A random value x from the rectangular distribution $0 < x < 1$ was also chosen, and with this, a random point $x \cdot \sum_{i=1}^{20} T_i$ chosen in the time period under consideration. The position of this point was then found relative to the time increments T_i. Then, if

$$\sum_{i=1}^{j} T_i < x \cdot \sum_{i=1}^{20} T_i < \sum_{i=1}^{j+1} T_i ,$$

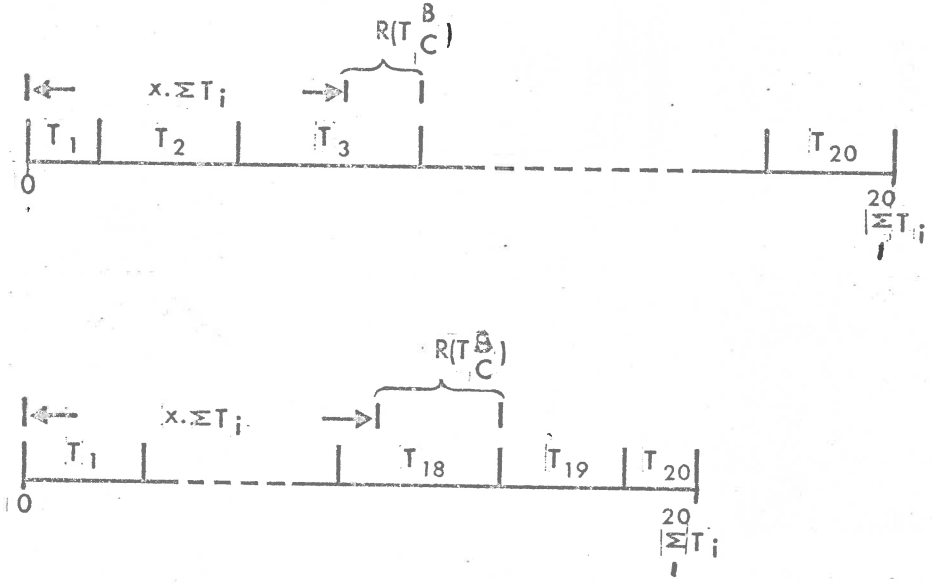


FIG. 11.2: EXAMPLE OF TWO SUCCESSIVE EVALUATIONS OF $R(T_C^B)$

the residue length of computation on program B was taken to be

$$R(T_C^B) = \sum_{i=1}^{j+1} T_i - x.\sum_{i=1}^{20} T_i \ .$$

The process was repeated for different sets of T_i. After 10^5 times, the mean of $R(T_C^B)$ had stabilised at:

$$R(T_C^B) = .60 \text{ sec.}$$

Giving priority to program B (parallelling what was done in simulation Run 2) would have meant waiting, on the average, about five times longer for the end of the computation sequence than if priority had been given to program A.

The simulation study dealt with the case where there were three programs rather than two. However, the conclusion to be drawn from the preceding calculations is that a program about to begin a computation sequence should have priority over another program at some unknown point in a computation sequence. The simulation was repeated (Run 3), but with the difference that, on completing a sequence of input and output, a program was given priority over other programs, if any, in computation sequences. The result showed a substantial improvement: utilization was 0.768 (from an average per program of .300) and the improvement factor 2.57.

One requirement remained unfulfilled. A program could still on occasions gain highest priority and monopolise the processor during very long periods of computation. An independent external interruption, presumably from a real-time clock, is needed to remove a program from the highest priority position when that program is using all the processor's time.

The frequency with which interruptions should be made must now be considered. The interval between successive interruptions must certainly be very long relative to the time needed to adjust priorities. The priority adjustment would take about 500 μs in CIRRUS. However, interruptions must be frequent enough to prevent any program being held out of operation for a period long enough to be inconvenient for an on-line operator. An interval of from .25 to 2 seconds would be satisfactory on both counts.

One further factor must be considered. It was suggested that, when two or more programs were engaged in computation sequences, the strategy should be aimed at decreasing the expected time for one program, at least, to reach the end of the sequence. One might imagine that alternation between programs would increase this expected time. In fact, the converse is true, as will now be shown.

Suppose that an external interruption is to be made at intervals of ΔT sec. and that, if at the time of

interruption two or more programs are found to be in computation sequences, the one of highest priority is to be given lowest priority. Consider again the case where there are two programs, A and B. Program B is in a sequence of computation, A is about to begin a computation sequence and is given priority over B. The effect of alternating priorities on the expected time, $E(T_{IO})$, before one program completed a computation sequence was calculated in the following way:

Random values T_C^A and $R(T_C^B)$ were found as described earlier for the length of a computation sequence in A and the residue length of a computation sequence in B. Since the interruptions come from an independent source, the first was taken to occur $x.\Delta T$ sec., after the computation sequence began on program A. (x a random variate from the rectangular distribution $0 < x < 1$). Subsequent interruptions were then assumed at intervals of ΔT sec.

Time for one program to complete its computation sequence was therefore:

$$T_{IO} = T_C^A \quad \text{if} \quad T_C^A < x.\Delta T$$

$$\text{or } x.\Delta T + R(T_C^B) \text{ if } T_C^A > x.\Delta T \text{ and } R(T_C^B) < \Delta T,$$

or, generally:

$$T_{IO} = T_C^A + n\Delta T \text{ if } x.\Delta T + (n-1)\Delta T < T_C^A < x.\Delta T + n\Delta T$$

$$\text{and } R(T_C^B) > n\Delta T,$$

$$\text{or } x\Delta T + n\Delta T + R(T_C^B) \text{ if } T_C^A > x\Delta T + n\Delta T$$

$$\text{and } n\Delta T < R(T_C^B) < (n+1)\Delta T.$$

The process was carried out with about 3.10^5 random values of T_C^A, $R(T_C^B)$ and x for a number of values of ΔT. The mean values found for T_{IO} are shown in Table 11.1 . The results show that, by alternating between two programs, each in computation sequences, the expected time for one at least to complete its computation sequence decreases.

The underlying reason for this decrease is that the chance of an incomplete computation sequence being completed within some small period ΔT actually decreases as the time spent in the computation sequences increases. This fact can be understood by considering Fig. 11.3 .

The above calculations were made only for the case of two programs. Meaningful calculations for a three-program case would be difficult to make. There would sometimes be three programs in computation sequences but more often two. Furthermore, when two programs were in computation sequences, the proportion of total time available for them would be reduced to different extents depending on what was being done on the third, higher priority program. Nevertheless, one can assume that alternation of priority between those programs in computation sequences would have a small, beneficial effect.

The simulation was repeated (Run 4) with an interruption and, where necessary, alternation of priority after each whole second. Processor utilization was found to be

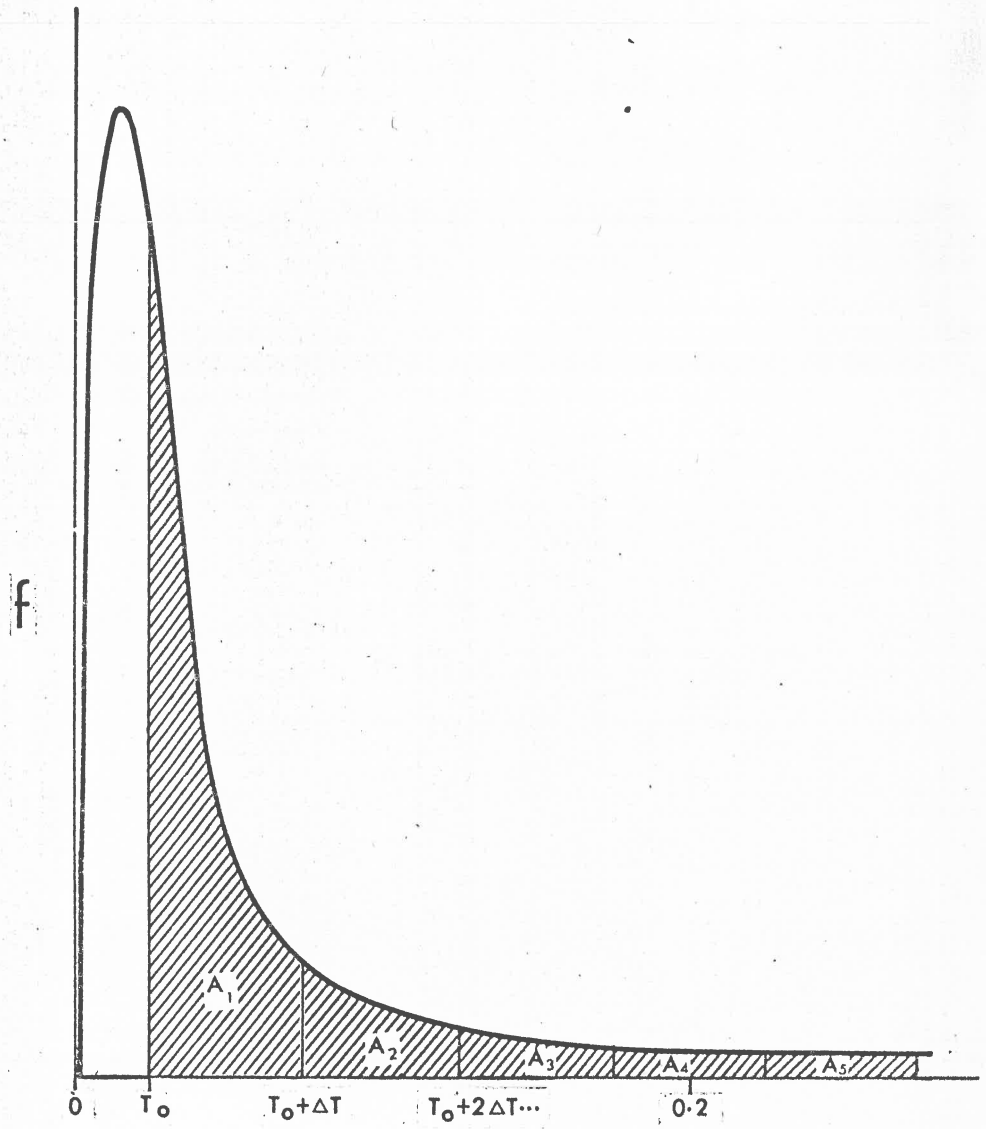


FIG. 11.3: ILLUSTRATION SHOWING DECREASE IN PROBABILITY OF COMPLETING A COMPUTATION SEQUENCE IN SUCCESSIVE PERIODS ΔT.

$$\Pr(\text{completion between } T_0+(n-1).\Delta T \text{ and } T_0+n.\Delta T) = \frac{A_n}{\sum_{i=n}^{\infty} A_i}$$

TABLE 11.1 : ESTIMATES OF EXPECTED TIMES FOR ONE PROGRAM TO COMPLETE A COMPUTATION SEQUENCE, WITH INTERRUPTIONS AND SUBSEQUENT ALTERNATION OF PRIORITY AT INTERVALS OF ΔT.

Interruption period ΔT (sec.)	$\overline{T_{IO}}$ (sec.)
.010	.053
.050	.053
.100	.053
.500	.056
1.000	.058
2.000	.064
10.000	.067
50.000	.092
75.000	.118

Note: $\overline{T_C^A}$ was calculated to be .12 sec.

$\overline{R(T_C^E)}$ " " " " .60 sec.

.764 (of a possible .879) and the improvement factor 2.61, showing a small but probably significant increase.

The results from the simulation runs discussed in this section are summarised in Table 11.2. The chief conclusion to be drawn from these results is that it is not sufficient merely to give priority to programs during input or output*. The order of priority between programs not engaged in input or output must also be considered. This fact is relevant for other systems in which programs do not control their peripheral units directly and in which data transfers would automatically have priority.

---

*If one compares the results from Run 2 and Run 1, and then the results of Run 3 and Run 1, it would appear that simply giving priority during input and output is hardly worthwhile. However, for Run 1, the reduction in efficiency due to a peripheral unit stopping completely during transfer was neglected.

TABLE 11.2 : SUMMARY OF RESULTS FROM SIMULATION RUNS.

Run	Expected time to complete computation sequence (estimate)	I
1	-	2.40
2	$\overline{R(T_C^B)}$ = .60	2.46
3	$\overline{T_C^A}$ = .12	2.57
4	$\overline{T_{IO}}$ = .06	2.61

12. CONCLUSIONS.

The conclusions to be drawn from arguments put forward in this thesis have been explicitly stated during the preceding sections. They will therefore not be restated here in detail.

The aim of this thesis has been to show that multiprogramming is both desirable and economically feasible for the smaller scientific computer. The author offers successful multiprogram operation in CIRRUS as proof that multiprogramming is feasible in a low-cost system. Apart from additional storage and peripheral units, the hardware needed to implement multiprogram operation has been quite trivial. The software required - some 200 micro-instructions and 380 machine-code instructions - has also been quite small. A complex supervisory program is certainly not needed to produce a multiprogram system which is efficient, reasonably comprehensive, and which imposes no burdens on the user.

The desirability of multiprogramming can hardly be in doubt. For the cost of little more than the additional storage and peripheral units to provide for three programs, a computer system has been built with the processing capacity and operating convenience of three separate computers each having very nearly the capacity of a single-program CIRRUS computer.

The author concludes that the use of multiprogramming should be considered during the designing of any small-scale scientific computer system.

APPENDIX A. CIRRUS: A GENERAL DESCRIPTION.

A brief description of CIRRUS is given here for reference purposes. More detailed descriptions may be found in articles by Allen and Rose (1963), Allen *et al.* (1963).

A1. System Structure

The complete system design is shown in Fig. A 1. Two separate core stores are used, the main and register stores. The basic word length is 36-bits. The instruction-code format allows for addressing 32,768 words in the main and 64 words in the register store. Of the 32,768 words in the former, 24,576 are provided with standard 6μs cycle-time coincident current cores, while the remaining 8,192 words are reserved for a special form of semi-permanent wired storage (Butcher, 1964) intended to hold system programs and subroutines.

The main working section of the central processor is made up of general purpose flip-flop registers: M, R, N, and Z of half-word (18 bit) length and two smaller registers A (7 bits) and E (10 bits). A general-purpose "arithmetic" unit is used to perform arithmetic, logical or shift operations on 18 bit half-words.

A variety of interconnections between stores, registers and arithmetic unit can be made for any single operation.

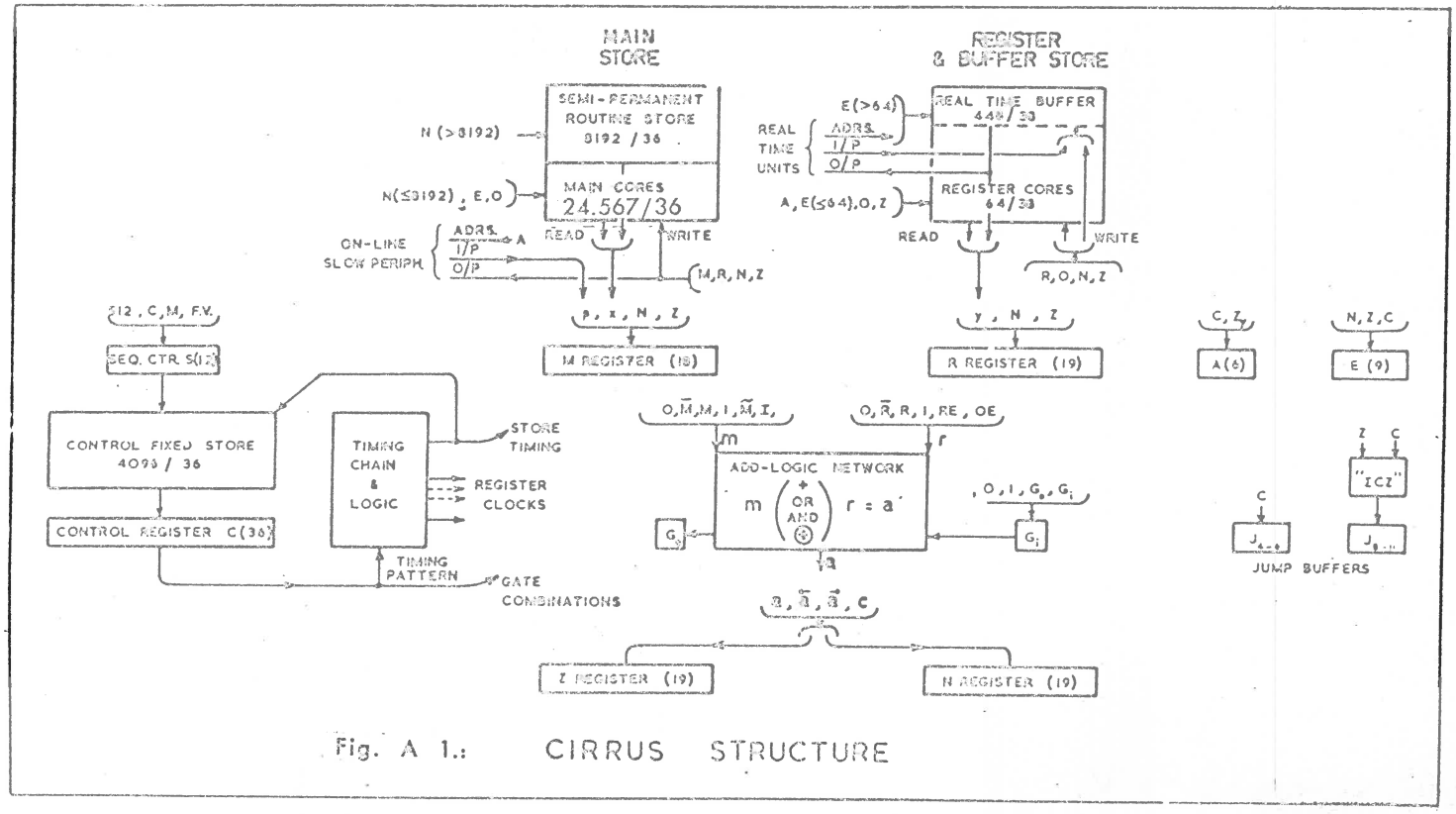


Fig. A 1.: CIRRUS STRUCTURE

In a typical operation, a half-word might be taken from the main store, another half-word from the register store or from a static register, and the result of an operation on the two quantities returned to the store within a single 6μs store cycle. If no reference to a store is needed, each operation takes only 1.5μs.

The interconnections for a given operation are determined by the value held in the 36-bit control register C. Patterns are read into C from a control or microprogram fixed store whose physical construction is identical to that of the semi-permanent routine store. The control store is addressed from the 12-bit microprogram sequence counter S. The patterns wired into this store determine the behaviour of the machine.

Each machine-code operation consists of a sequence of micro-operations. For each machine-code instruction, a number of patterns have been wired into microprogram store. The machine-code programmer need not know of the internal register structure as none of these registers are referenced in machine-code. The functions normally associated with accummulators and index registers are carried out by any of the 64 words of the register store. In any machine-code operation, the results always appear in one or both core stores and no information needed by the program is retained in the static registers.

The microprogram store has a total capacity of 4,096 words, each of 36-bits. Its cycle-time is 1μs. When fully used, it will contain eight separate plates, each holding 512 words. So far, only four plates have been installed. The micro-code sequences making up machine-code operations were tested by simulation on an IBM 7090 before the plates were wired. Minor changes or additions to existing micro-programs can be made in very few minutes.

The type of fixed store used to hold microprograms is inexpensive yet has a large capacity. As a result, the computer has a comprehensive machine-code instruction set containing about 200 instructions. The "built-in" arithmetic functions, for example, include operations on 18-bit integers, 36-bit fractions and floating-point numbers. Several instructions have been provided for use solely by the multiprogram control programs (Section 8, 9, 10 and Appendix B). The cost to the University has been between £20,000-£25,000*.

A2. Machine-code Instructions

The sequence of micro-operations constituting a machine-code operation can be considered as two separate "phases". The first or "routine" phase is common to all operations.

---

*This figure includes the cost of 24,576 words (36-bit) of variable store, of peripheral equipment and of some construction work done by private contractors.

Its purpose is to select the microprogram sequence which will carry out the operation in the second or "execute" phase. The two phases are shown diagrammatically in Fig. A 2. The routine phase is closely analogous to a standard interpretive procedure in machine-code. It is, however, more efficient, since the hardware was designed for the purpose. Three store cycles are required and the routine phase normally takes 21μs to complete. In the first cycle, the contents of the current sequence counter* are extracted from store, incremented and returned to store. The 36-bit instruction itself is then extracted in two store cycles. Finally, control is transferred to the appropriate microprogram sequence for carrying out the requisite operation in the execute phase.

The speed of the routine phase is assisted in two ways. First, the two stores can be driven in parallel; second, micro-operations not referring to store can overlap the write phase of a store cycle. For example, during the first store cycle of the routine phase, the sequence counter contents and the expected state of the peripheral indicators are extracted simultaneously. The comparison of the expected and actual states of the indicators is made while the incremented value is being returned to the sequence counter.

---

*There are 15 separate sequence counters, each a half-word in the main store. The address of the sequence counter for the current program is held in the B register.

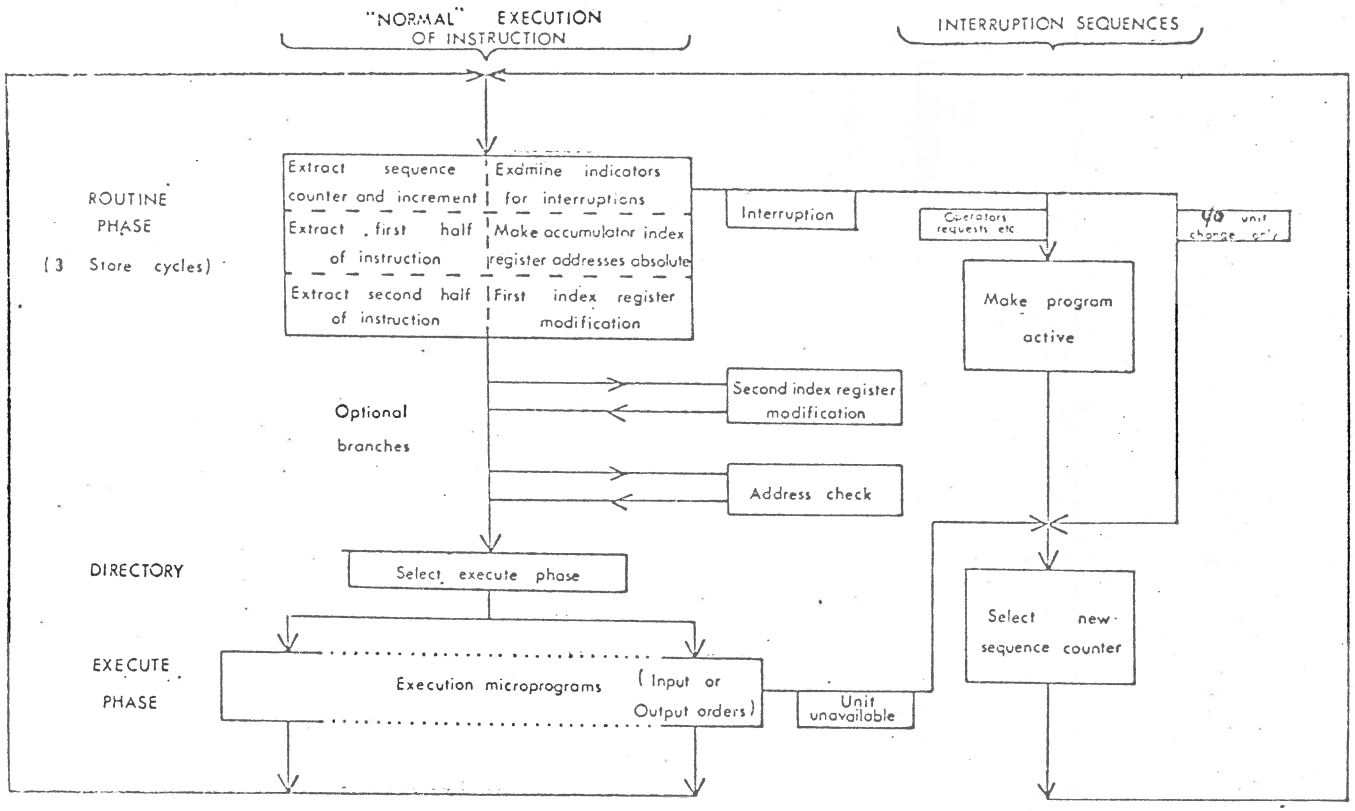


FIG. A2.: FLOW DIAGRAM FOR MICROPROGRAMS IN THE CONTROL STORE

Accumulator and index register addresses within each instruction are stored as relative addresses (relative, this is, to the first register store address used by the program). They are converted to absolute addresses without loss of time during the second store cycle of the routine phase. The routine phase also includes an optional branch for a possible double index register modification. Holding register store addresses in relative form and providing double index register modification has proved valuable in several ways. The parametric addressing technique (Section 6.2 of the text) in particular depends on this mode of addressing.

The standard instruction format allows for 9 bits specifying the operation, thereby providing for up to 512 separate operations. These 9 bits are transformed directly via a special gating link to the microprogram sequence counter S at the end of the routine phase. The first 512 words of the control store constitute a "directory". The n^{th} position of the directory usually holds a control transfer instruction addressing the microprogram sequence for the operation whose function code is n. In some cases, however, one micro-instruction only is needed to execute the operation. The directory entry itself is then this instruction.

To simplify programming, the 512 possible operations have been broken up into 64 basic functions, each having

eight possible "variants". For example, the function MP (multiply) has five variants: normal (giving the rounded product of two 36-bit fractions); special (giving the double-length product); floating point; upper half-word (giving a 36-bit product from two 18-bit integers); lower half-word. In symbolic code, these are written N MP; S MP; F MP; U MP; L MP.

The comprehensive set of 18-bit operations has been valuable in improving both the speed and space requirements of the systems programs. Several special-purpose operations have also proved particularly useful. A special-purpose 72-bit instruction, for example, gives the complete increment and test in a "DO" statement without making use of index registers.

The total execution time for 18-bit arithmetic and logical operations is 27μs, that is, one store cycle only is needed after the routine phase. For similar 36-bit operations, a further store cycle lifts this time to 33μs. However, the speed of more complex operations suffers because intermediate quantities must be returned to core store. Floating point addition for example takes 210μs. Nevertheless, the execution times for floating point operations are much better than could be obtained with a machine-code interpretive process.

The use of microprogram to control time-sharing has

been mentioned in the text. Fig. A 2 shows the inter-relationship between this microprogram and that used for the routine and execute phases. For example, the regular indicator check is made during the first store cycle of each routine phase. The indicator check before a character transfer is part of the execute phase of the input or output operation. The regular indicator check may show that a control program is to be made active (Section 9). Either indicator check may show that a new program should be selected (Section 7).

An optional branch from the routine phase may be taken to check each operand address against the program's bounds (Section 6.2.3). If this branch is taken, each operation is slowed by 20μs. A bit in the E register beyond those bits used to hold the sequence counter address indicates whether or not this branch is to be taken. Since changing from one program to another requires E to be reset, this bit is also reset. Address checking can therefore be applied or not applied with any given program as the situation demands.

APPENDIX B: THE CIRRUS MULTIPROGRAM SYSTEM - A SPECIFICATION.

The author has, in the text, been mainly concerned with showing the principles of the CIRRUS multiprogram system and with the reasons behind various decisions which have been made. A detailed specification is given here for the software in use at 15/9/64. Three operating stations are allowed for.

The appendix should be read in conjunction with the Prefatory Statement, Sections 7, 8 and 9 and Appendix A. The subject matter of the specification is dealt with in the following order:

(1) Program allocation.

(2) Indicators.

(3) Working space.

(4) Micro-code sequences.

(5) Machine-code sequences.

Note that numbering and addressing are given in octal notation, unless obviously otherwise.

B.1. Program Allocation

The 15 internal programs are used for the following purposes:

Variable programs:

1	- Sequence counter in	31	Allotted as needed for current problems
2	" "	32	
3	" "	33	
4	" "	34	
5	" "	35	
6	" "	36	
7	" "	37	Used for elementary initial program (see Section 8.3) if all lower numbered programs are in use.

Permanent programs:

10	- Sequence counter in	40	Relocating program	
11	" "	41	Console 0	Control Programs
12	" "	42	" 1	
13	" "	43	" 2	
14	" "	44	Console 0	Secondary Programs*
15	" "	45	" 1	
16	" "	46	" 2	
17	" "	47	Spare	

*The "secondary" programs are used by the control programs to carry out such operations as store dumps, thereby leaving the control programs free for further instructions. However, if expansion were desired, their functions could be incorporated into the control programs themselves.

B.2 Indicators

(1) Peripheral indicators, I.

Of the 18 peripheral indicators (I of Section 7.3.2), the following have been allocated:

b_0 (most significant)	Common for I' (see below)	
b_1	K_0-Keyboard	
b_2	R_0-Reader	Console 0
b_3	P_0, T_0-Punch, Typewriter	
b_4	K_1	
b_5	R_1	Console 1
b_6	P_1, T_1	
b_7	K_2	
b_{10}	R_2	Console 2
b_{11}	P_2, T_2	

(2) External interrupt indicators, I'

Interrupt indicators are considered as being in 2 classes:

Class (a) (up to 18 indicators): The setting of any one of these indicators will cause execution of a corresponding microprogram sequence. Their purpose is primarily to initiate buffer transfer routines as described in Section 7.4. However, as there are at present

no peripheral units beyond those specified above, none of these indicators has so far been allotted.

Class (b) (up to 8 indicators): The setting of any one of these indicators will cause a microprogram sequence to make active one of the 8 permanent programs (10-17). So far, three indicators of Class (b) are in use:

b_0	(Most significant)	Not used
b_1		Interruptions from Console 0
b_2		" " " 1
b_3		" " " 2

Apart from console interruptions, an indicator of Class (b) would be used to show a request to make active any other permanent program (such as one to perform some on-line control function).

Note: b_0 of I is the result of an 'OR' circuit over all of I'. If any indicator of I' is set (=1), b_0 of I becomes non-zero. The change in any I' is therefore detected on checking I at the beginning of the next routine phase.

B.3 Working space

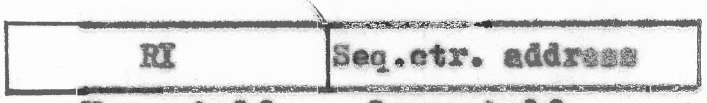
Addresses 4-137 (total 92_{10}) in main store are used as working space in the current three-station system.

(1) Priority ladder - addresses 4-23.

15 of the 16 words correspond to sequence counters (words 31-47) while the 16th is a marker dividing active programs

from inactive programs.

e.g.



RI: Relevant indicators - indicators of those peripheral units in use with this program.

Example:

	RI	Seq.ctr.	
4	200000	000041	Program 11
			←——————*
5	340000	000032	Program 2
6	034000	000031	Program 1
7	000000	400000	Marker
	.....		
22	020000	000042	Program 12
23	010000	000045	Program 15

In the example, programs 11, 2 and 1 only are active and have that order of priority. The order of the words below the marker is not significant. In all cases where an inactive program becomes active (by Interrupt Indicator sequence or instruction X DR), its priority word is placed at a point immediately after the lowest priority program whose number is $\geqslant 10$ (shown by * above). Hence, priority of programs 10-17 is always preserved over programs 1-7.

Where an active program is suspended (instructions N SP, F DI, F DR), the priority word is placed at the bottom of the ladder.

(2) Program Catalogue - addresses 30-107 (48_{10} words) (See Table B 1.)

(i) Addresses 31-37, 51-57, 61-67 (upper halves), 71-77 (upper halves), 101-107 hold information relating to variable programs 1-7.

(ii) Addresses 40-47 hold information relating to permanent programs 10-17.

(iii) Addresses 61-63, 71-73, 74-76 (lower halves in each case) are used by the control programs for Consoles 0-2.

(iv) Addresses 30, 50, 60, 70, 100 are used for general information.

The nature of quantities stored for programs 1-17 is probably clear from the annotations to Table B 1. However, the following are explained in more detail:

PC_j - A flag to show when the program from console C_j is in the Preliminary Sequence. This flag is checked during the End Sequence to decide whether lock-out flag P should be reset. Resetting of the P flag during the End Sequence is necessary only when there is a "reject" request

TABLE B.1. PROGRAM CATALOGUE

30	31	32	33	34	35	36	37	
-	YL_1	YL_2	YL_3	YL_4	YL_5	YL_6	YL_7	Upper
F	SC_1	SC_2	SC_3	SC_4	SC_5	SC_6	SC_7	Lower
40	41	42	43	44	45	46	47	
YL_{10}	YL_{11}	YL_{12}	YL_{13}	YL_{14}	YL_{15}	YL_{16}	YL_{17}	Upper
SC_{10}	SC_{11}	SC_{12}	SC_{13}	SC_{14}	SC_{15}	SC_{16}	SC_{17}	Lower
50	51	52	53	54	55	56	57	
X_{base}	XU_1	XU_2	XU_3	XU_4	XU_5	XU_6	XU_7	Upper
X_{avail}	XL_1	XL_2	XL_3	XL_4	XL_5	XL_6	XL_7	Lower
60	61	62	63	64	65	66	67	
X_{last}	XD_1	XD_2	XD_3	XD_4	XD_5	XD_6	XD_7	Upper
-	PC_o	PC_1	PC_2	-	-	-	-	Lower
70	71	72	73	74	75	76	77	
-	C_1	C_2	C_3	C_4	C_5	C_6	C_7	Upper
W.L.	$YL(C_0)$	$YL(C_1)$	$YL(C_2)$	CL_0	CL_1	CL_2	-	Lower
100	101	102	103	104	105	106	107	
W.L.	P_1	P_2	P_3	P_4	P_5	P_6	P_7	Upper
W.L.	C_1	C_2	C_3	C_4	C_5	C_6	C_7	Lower

F: Lock-out flags

YL_i: Base Y address (Y: Register store)
SC_i: Sequence Counter

($1 \leqslant i \leqslant 7$: Variable program

$10 \leqslant i \leqslant 17$: Permanent program)

XU_i: Upper X Address (X: Main store)
XL_i: Base X address

XD_i: Data base address

PC_j: =1 while program is in preliminary sequence
(j = console no., $0 \leqslant j \leqslant 2$)

C_i = Console of origin

$YL(C_j)$: Base Y for console j

CL_j: Compile/load indicator

P_i: Peripheral modifier

C_i: Console modifier (=C_i above)

W.L. General working location.

from the operator during the Preliminary Sequence.

$YL(C_j)$ - The first address to be used in register store by a program originating from console C_j.

CL_j - An indicator to show whether the program from C_j is being compiled or loaded.

F - The lock-out flags described in Section 8.6.

X_{base}, X_{last} - The first and last addresses available for variable program use. These remain unchanged during operation, but would be altered if the store were extended or the multiprogram working space increased.

X_{avail} - The current record of storage not on use by variable programs.

(3) Register Store

The first 20_{10} words of register store are also used as working space:

	Upper half	Lower Half
0	E I (See Section 7.3.2)	Y_L
1	X_U	X_L
2-6	Working space - control program for Console 0	
7-11	" " - " " " " "	1
12-20	" " - " " " " "	2
21-23	" " - relocating program	

Y_L : base register address of current program.

X_U, X_L: main store bounds of current program.

B 4. Micro-code sequences

In CIRRUS, micro-code is used extensively for purely multiprogram functions, partly as adjuncts to the order extraction procedure, and partly in special machine-code instructions. For most of these functions, machine-code sequences could have been used, and would have been used in preference to hardware if CIRRUS had not been microprogrammed. In machine-code, fewer instructions would have been necessary. However, micro-program gives a considerable advantage in speed which is important for inter-program switching in particular. Construction of special machine-code instructions from micro-code is also valuable on this and two other counts. First, it provides the simplest possible means of referring to a sequence of micro-operations which is used in several places. Second, many of the sequences are such that a program change cannot be permitted during their execution. Since an interruption can only occur between successive machine-code operations, providing each sequence as a single micro-coded machine-code operation has eliminated the difficulty.

Those micro-code sequences directly connected with multiprogramming are stated in Table B 2. The "adjuncts to the order extraction procedure" shown under I of Table B 2 and the special instructions under IIb are concerned solely with multiprogram control functions. The special instructions

TABLE B.2. SUMMARY OF MICRO-CODE USED FOR MULTIPROGRAMMING

				Number of micro-instructions	
I	Adjuncts to the Order Extraction Procedure:				
	Program Selection			29	
	Interrupt Indicator Inspection			51	
	In Routine Phase: Peripheral Indicator Check			1	
	Address Check			10	
II	Special Machine-Code Instructions:				
IIa	N	SP	Suspend current program	16	(15)
	S	SP	Terminate current program	3	(3)
	U	IP	Input to upper half	24 + some in L OP	(7)
	L	IP	Input to lower half	16 + some in L OP	(7)
	U	OP	Output from upperhalf	22 + some in L OP	(7)
	L	OP	Output from lower half	18	(11)
IIb	F	DI	Suspend current program and reset sequence counter	4 + use of N SP	
	I	DI	Obtain current sequence counter address	1	
	N	DR	Begin address check	12	
	S	DR	Cease address check	1 + use of N SP	
	F	DR	Suspend program	Entry to "Interrupt Indicator Inspection"	
	X	DR	Resume program	25	
	U	DR	Alter relevant indicators	12	
	Y	DR	Lock	10	
	L	DR	Release		

IIa: Instructions available to the user.

IIb: Instructions reserved for system programs.

under IIa would have been required in the machine-code even if the computer had not been multiprogrammed. However, the instructions under IIa have become more elaborate because of multiprogram operation. The number of micro-instructions needed for each has therefore been increased, probably by the number of instructions indicated in brackets.

Hence, implementation of multiprogram operation in CIRRUS has required about 206_{10} micro-instructions.

The microprogram sequences are described briefly below:

I Adjuncts to the Order Extraction Procedure

The relationship to the "normal" order extraction procedure of those special-purpose sequences concerned with multiprogramming can be understood from Fig. A.2. If, in the first store cycle of the routine phase a changed peripheral indicator is detected, a branch is taken out of the routine phase. If one or more of the External Interrupt Indicators I' have been set, these are examined. Indicators of Class (a) are considered first. Provision has been made for branches to microprogram sequences to service these interruptions*.

Indicators in Class (b) are then examined. If the nth, say, has been set, the priority word for program (10 + n) is moved to the active portion of the priority ladder.

---

* See note on External Interrupt Indicators in Appendix B.2.

Once all external interruptions have been cleared, control is transferred to the program selection sequence. This sequence has been described in Section 7.3.2. and illustrated in Fig. 7.3. However, it should be noted that two further operations not mentioned in Section 7 are carried out after program selection. First, the base register address for the selected program is set in the lower half of register store address zero (absolute). This address is used during the second store cycle of each subsequent routine phase to convert the accumulator and index register addresses in each machine-code instruction from relative to absolute form. (See Appendix A.2 and Fig. A.2). Second, the upper and lower limits of the selected program's main store space are placed in register store address 1 (absolute). This information is used if the address check branch is taken after each routine phase (Fig. A.2).

The time spent in the program selection sequence is $(31+9n)\mu s$, where the program selected has priority n.

II. Machine-code Instructions

IIa. Instructions available to the user:

N SP Suspend current program

This instruction is used to indicate a pause. It has the result of removing the program's priority word from the active portion to the bottom of the priority ladder.

S SP Terminate current program

This instruction is used as the logically final instruction of any program. It is simply a control transfer to the End Program sequence described in Section 8.3.

U IP, L IP, U OP, L OP Input and output

These instructions are used to transfer a single character between the buffer of a peripheral unit and either the most or least significant position of a designated register store address*. All include a check of the unit's indicator prior to initiating the transfer itself. If the unit is found to be unavailable, transfer is made automatically to the Program Selection sequence (see Section 7.3.2. and Figs. 7.3. and A 2.).

Times for these instructions are:

If successful	U IP	78.0 μs
	L IP	67.5 μs
	U OP	73.5 μs
	L OP	61.5 μs
and if unsuccessful	U IP	50 μs
	L IP	50 μs
	U OP	52.5 μs
	L OP	52.5 μs

All these times include the time spent in extracting the instruction.

---

*Although these instructions are available to the machine-code programmer, he would almost invariably use the standard "read" and "write" subroutines.

Processor time lost in an inter-program switch can now be specified. If a peripheral unit is found to be unavailable, total time lost is that spent in the unsuccessful input or output operation plus the time needed to select another program.

i.e. $(50 + (31 + 9n))$ μs,

where the new program selected has priority n. When a return is made to a higher priority program following a peripheral indicator charge, the program selection sequence is reached after one store cycle. Time lost is therefore

$(9 + (31 + 9m))$ μs,

where the program selected has priority m.

Most frequently, m will be 1 or 2 and n will be 2 or 3. The mean switching time is therefore well below 100 μs.

IIb Instructions reserved for systems programs

F DI Suspend and reset

Use of this instruction has the same effect as use of N SP except that the program's sequence counter is reset to a designated value.

I DI Obtain sequence counter address

This instruction enables the program to find its sequence counter address and thence its current position in the catalogue.

N DR Begin address check F DR Cease address check

These instructions are used to delineate sequences of operations during which operand addresses, if in main store, must be checked to ensure that they lie within the program's bounds. Use of N DR causes one bit of the lower half of the priority word to be set non-zero. Use of F DR removes that bit (See notes on address checking in Appendix A 2.).

F DR Suspend program

This instruction is similar to N SP except that it must have an operand and is therefore used by one program to suspend another. One example of its use is by the operator control program following a "STOP" request from the operator.

X DR Resume program

This instruction, like F DR, must have an operand. It is used to place the priority word of the designated program in the active portion of the priority ladder. The operator control program, for example, uses this instruction after receiving a "RESUME" request. The priority word is always placed in a position which gives the program priority over all variable programs (see "priority ladder", Appendix B 3). The instruction can therefore also be used for shuffling priorities.

U DR Alter relevant indicators

This instruction can be used to insert in or remove from the upper half of the current program's priority word the indicator bit corresponding to the designated peripheral unit. The instruction is used only during the Preliminary and End Sequences.

Y DR Lock L DR Release

The functions of these instructions have been dealt with in Section 8.6.

B 5. Machine-code Sequences

The machine-code software needed for multiprogram operation can be regarded as four separate sequences:

(1) Operator Control	- 125 words	
(2) Preliminary Sequence	- 87 words	
(3) End Sequence	- 43 words	Total: 387 words
(4) Relocation	- 132 words	

The sequence of instructions making up the Operator Control "program" is used by programs 11-13. (See Section 9.3 and Figs. 6.1, 9.1). The Preliminary Sequence and End Sequence are used by the variable programs, programs 1-7 (See Sections 6.3, 8.3 and Fig 6.1). The Relocation Sequence is used only by program 10 (See Section 8.4).

All sequences of instructions are held permanently near the end of the store. All except the Relocation Sequence refer to the standard READ and WRITE routines,

also held near the end of store.

The operating keyboards are separate from the typewriter and have keys labelled as shown in Table B 3. The typewriter keyboard could have been used but it was found a little cheaper and more convenient to build a separate keyboard.

The Operator Control program is illustrated in Fig. B 1. The 125-word sequence includes instructions to execute all requests shown in Table B 3 except those under ACCEPT. The "establishment" of a variable program, also part of the operator control program, is shown in Fig. B 2.

The Preliminary and End Sequences are illustrated in Figs. B 3 and B 4 respectively. The Relocation Sequence is shown in Fig. B 5.

TABLE B 3

KEYS ON THE CIRRUS CONTROL KEYBOARD AND THEIR FUNCTIONS

Label on key	Function
START*	(i) (no program in store from this console) Initiate compilation or assembly of source program. (ii) (program in store from this console) Resume the program.
LOAD*	Initiate loading of object program.
STOP*	Stop the program.
RESET*	Transfer control to head of program.
REJECT*	Terminate the program.
SENSE SWITCH 1 SENSE SWITCH 2 SENSE SWITCH 3 SENSE SWITCH 4 READ SENSE SWITCHES*	Sense switches may be referenced singly or in combinations (16 combinations in all). Each variable program refers to a reserved address in the program's work space. Settings (ON or OFF) are shown by lights within the keys. A new setting is not actually registered until the READ SENSE SWITCH key is pressed.
ACCEPT*	This is a general-purpose key which can precede one of a number of separate messages. For example, in the present system: ACCEPT, 0, space, d_1,d_2,d_3,d_4,d_5, space will result in a store dump beginning from the address d_1.....d_5 relative to the head of the program. A secondary program rather than the control program carries out the actual dump. Dumping is terminated when any other operative key (usually "STOP" is pressed.
digits 0 - 9	

* Operative key.

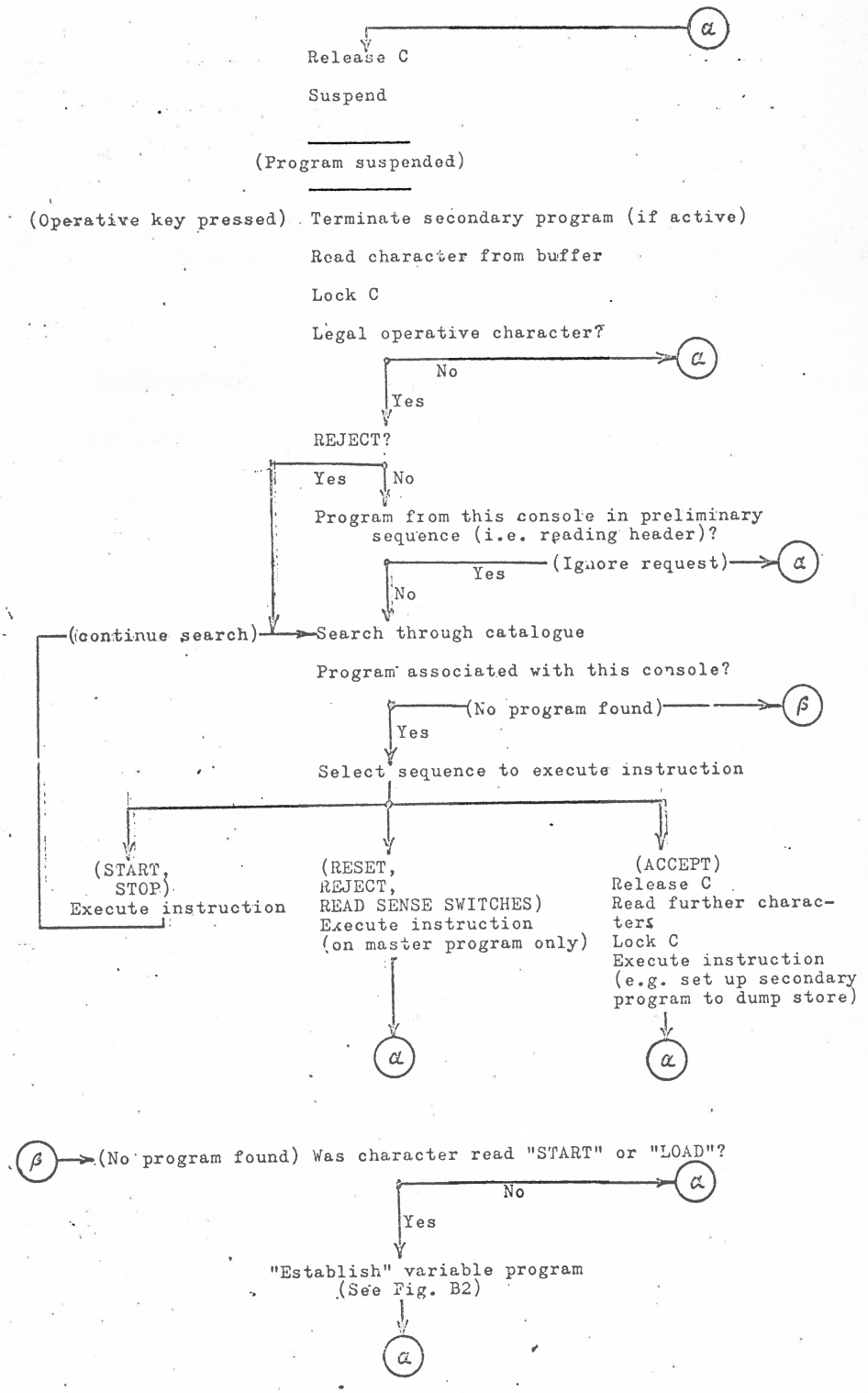


Fig. B 1: Operator Control Program

Set CL_j (compile/load indicator) in program catalogue

Release C

*(1) Lock C,P

Set PC_j

Find first vacant position (i^{th}) in catalogue following last position occupied (always possible)

Set C_j, P_j in catalogue

Set indicators for K_j, R_j, P_j/T_j in RI_i

Set $YL_i = YL_j$

Set $XL_i = XL_{i-1}+1$, or $X_{base}+1$ if $i = 1$

Set SC_i = address of head of preliminary sequence

*(2) Raise i^{th} priority word to active portion of ladder

Go to a of Fig. B 1.

Fig. B 2. "Establishment" of a variable program by the control program.

In this example, the control program from console j is setting up variable program i.

* Notes:

(1) P is released at the end of the preliminary sequence in the variable program. If the point marked *(1) above is reached in any control program before P has been released, the control program will wait until the preliminary sequence has been completed.

(2) The variable program will, however, have lower priority than the control program. It cannot therefore begin before the control program terminates.

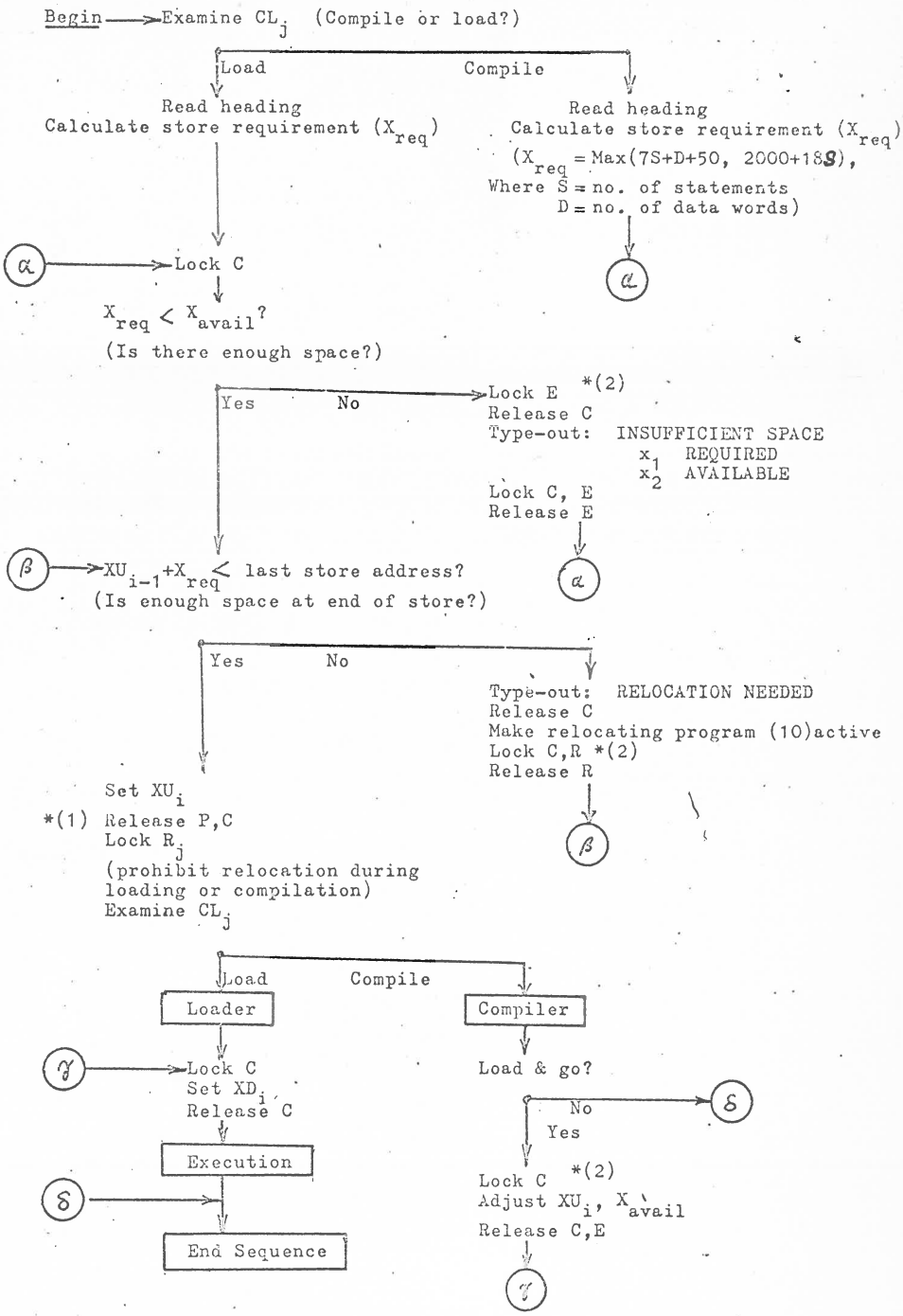


Fig. B 3: Preliminary Sequence. (Console j, variable program i). The position of the loader and compiler and of program execution and the end sequence are also shown.

*Notes: (1) At this point, another program may enter the Preliminary Sequence.

(2) For an explanation of the use of Lock and Release instructions at these points, see the examples given in Section 8.6.

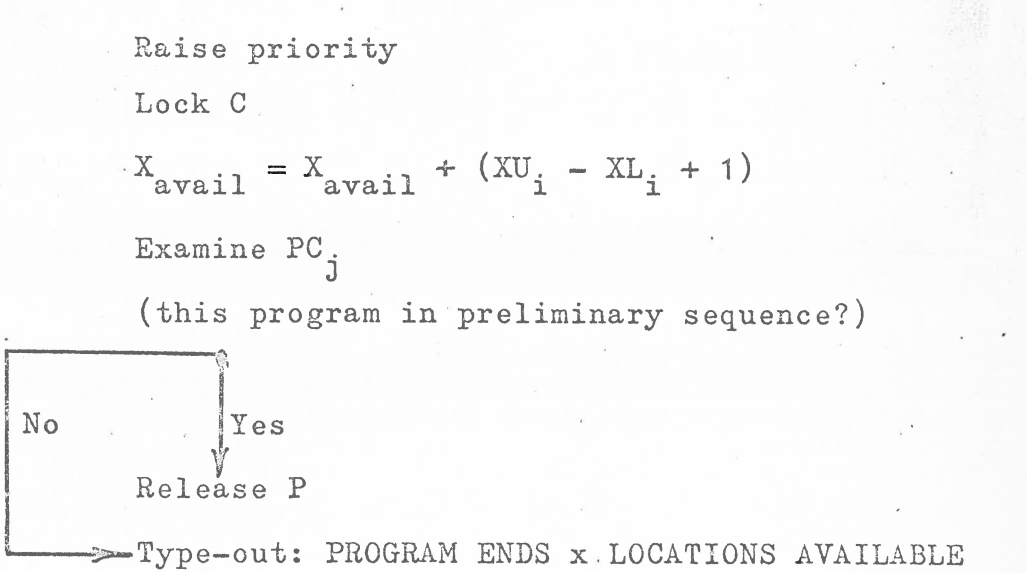


Fig. B 4: End sequence
(Variable program i, from Console j)

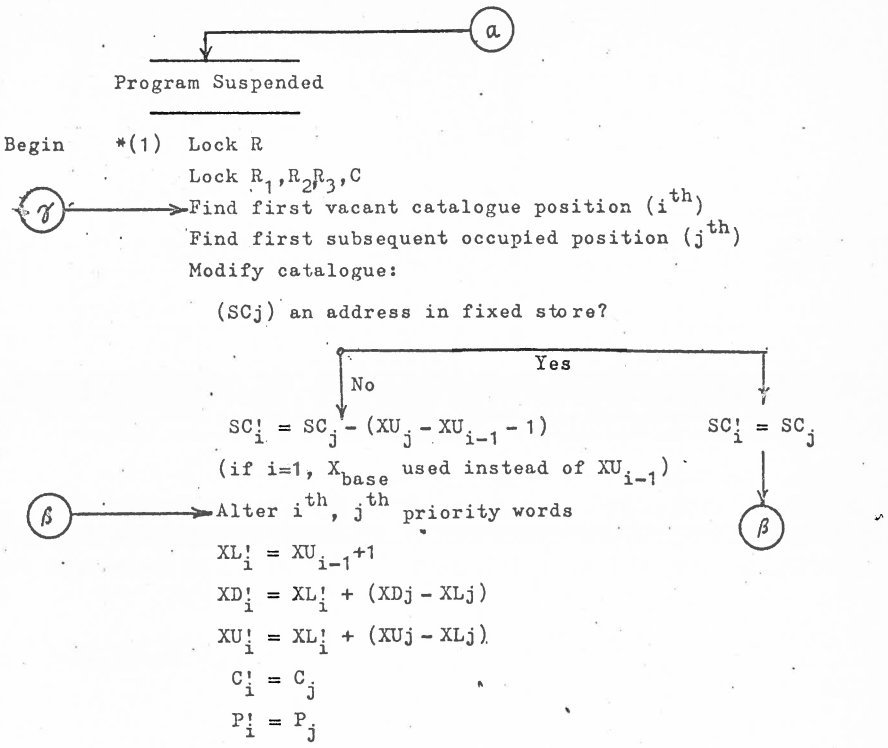


Fig. B 5: Relocation Sequence

* Note: (1) For an explanation of the use of Lock and Release instructions at these points, see Section 8.6.

Summary to Appendix B

To implement multiprogram operation

(a) 116 words of working space in main store,

(b) 20 words of working space in register store,

(c) 206 instructions in micro-code, and

(d) 387 instructions in machine-code,

have been used. Further machine-code program has been used and more will be used in future to improve the facilities for the on-line operator. However, the above statement covers all that has been necessary.

C. THE SIMULATION STUDY.

The results obtained from computer simulations of time-shared operation in CIRRUS have been discussed in Sections 4.1 and 11.3 of the text. For use by the simulator, a set of programs was constructed from information found by examining a number of problems run on other computers. The sample set of programs has been mentioned in Section 3.1. The method by which the program set was constructed and the simulator itself will now be described.

It is important to find with reasonable accuracy the improvement to be gained through time-sharing. Unfortunately, the demand for time and the pattern of this demand very greatly from program to program. The time-sharing method itself, though it may be conceived initially as a fairly straightforward basic procedure, is likely to become complicated by the use of measures intended either to meet certain constraints or (hopefully) to increase efficiency. Finding an accurate estimate of the improvement by analytic methods would be difficult if not impossible.

Since an analytic solution was not possible, simulation on another computer was necessary. To make the desired simulations a FORTRAN program was used, initially on an IBM 7090 and later on a CDC 3600.

C.1. The Program Set.

The author's original intention was to prepare a set of hypothetical programs for use in the simulation study. However, after a preliminary inspection of a number of programs being run on other computers, it was felt that it would be more satisfactory to take a random sample of existing programs, and to attempt to predict the behaviour of similar programs if written for CIRRUS. At this stage of the investigation, users in the Adelaide University had access to two computers, an IBM 1620 on the premises, and an IBM 7090 owned by the Weapons Research Establishment. Ten problems were chosen. Three of these were being run on the 7090, and seven on the 1620 (Table C1). Of the 1620 problems, a number were being run in 2 or 3 program parts, but these were regarded as single programs for CIRRUS, and the intermediate output-input phases were disregarded.

Extraction of the relevant characteristics was in all cases done on the 1620. Because of storage limitations, most programs were divided into logically separable parts. In several cases, sizes of arrays were reduced. The recombinations or transformations needed were made later. For example, to acquire the information on the 25 x 25

TABLE C.1: A SUMMARY OF THE PROBLEMS MAKING UP THE PROGRAM SAMPLE USED IN THE SIMULATION STUDY.

1. Fitting a harmonic curve (θ to 6θ) to 121 values.
2. Inversion of a matrix of order 25 x 25.
3. Solution of the differential equation

$$\frac{\partial}{\partial x}\left(h^{5/3}\left(1-\frac{\partial h}{\partial t}\right)^{\frac{1}{2}}\right)+\partial h/\partial t = A$$

with various sets of boundary conditions.

4. Fitting of a curve of the form :

$$y = a_1(R_1-\bar{R}_1)+a_2(R_2-\bar{R}_2)+a_3(R_3-\bar{R}_3)+ t/(a_4+ a_5 t)$$

for 66 values of R_1, R_2, R_3, t.

5. Factor analysis with 500 observations for each of 30 variables.
6. Processing of doppler data from meteor streams.
7. A calculation of simple correlations between 2 different sets of temperature recordings.
8. An iterative solution of equations describing the geometry of radio reflection from a meteor trail.
9. Calculation of the co-ordinates of concrete blocks making up a double curvature arch dam.
10. Calculation of simple tables:

$$D = 3xy-9(1-x-y)$$

$$A = 8(1-x)-y(17+3x)/D$$

$$B = 8y/D$$

matrix inversion, the investigation was made with 3 x 3 and 5 x 5 matrices. The information for matrices of order 25 was then found by extrapolation.

Each program was regarded as a succession of alternate periods of computation and input or output. (At this stage, a "period" of input or output was considered as the transmission of the information in a single card read or punched, or a single line printed on the typewriter. During simulation, these periods were combined to form input or output "sequences". An input or an output "sequence" has been defined in Section 11.3 as the complete operation resulting from a single statement in source language). A tracing program was used under which the program sections were executed. A count n_T, was produced of the 1620 instructions obeyed between successive branches to the input-output subroutine, followed by a statement of what was involved in the particular input-output record. The latter included, first, the type of data transfer, and, second, the number of characters transferred expressed as: $S_1, C_1, S_2, C_2 \ldots$, where S_i was a number of trivial characters (blank card columns, printed spaces, Hollerith characters), and C_i a number of non-trivial characters (decimal digits, decimal points, signs).

The relevant extrapolations were made and program sections combined. The information was then punched on cards. To assess behaviour during compilation, several CIRRUS C-code statements were examined. Information from these statements was punched on cards in a similar format. Cards were added to each program deck to make up the relevant number of statements. A number of cards specifying behaviour during object program loading was also added, the choice of compilation or loading being made at the time of simulation. When all cards (four to five thousand) had been punched, the information was loaded on magnetic tape for use by the simulator.

The next step was to find suitable conversion factors for the CIRRUS case. Each value n_T specified a number of 1620 operations within a period of continuous computation. To find the appropriate factor for converting this number of operations to a period of time in CIRRUS, three short programs (10 x 10 matrix multiplication, sine and square root) were run on the 1620 under the tracing program and counts of n_1, n_2, n_3 were found. Equivalent programs were written in CIRRUS code, and the expected actual execution times calculated (T_1, T_2, T_3). The value

$$t = 1/3 \sum_{i=1}^{3} \frac{T_i}{n_i} = .000048$$

then gave a factor by which each n_T should be multiplied to give the appropriate period of time in CIRRUS.

READ and WRITE routines prepared for CIRRUS were examined and the average times for handling trivial and non-trivial characters estimated. These, together with waiting times, if any, were tabulated for each peripheral unit (as in Table 3.3). From this table, the simulator was able to convert information on characters transferred, to alternate periods of processor use and waiting time. Where the card reader and punch had been used in the 1620, it was assumed that the paper tape reader and punch would be used in CIRRUS. However, if the number of blank columns in any card field exceeded ten, the excess was disregarded as it was realized that spacing to this extent would be unlikely with paper tape. Where the 1620 typewriter had been used, typed output was assumed in CIRRUS with programs 1, 7 and 10 (See Table C 2) in which the quantity of output was small. Where typed output was substantial (programs 3, 4, 8, 9), it was assumed that the punch would be used in CIRRUS.

Prior to simulation, processor utilization and execution time of each program if run on CIRRUS were calculated (Table C 2). The figures obtained give a fairly reliable indication of the behaviour of programs coded to handle the same problems in CIRRUS.

TABLE C 2.

SUMMARY OF CHARACTERISTICS OF THE PROGRAM SAMPLE EXTRACTED FOR USE IN SIMULATION RUNS

PROGRAM NUMBER	ORIGINAL COMPUTER	SOURCE STATEMENTS	COMPILED INSTRUCTIONS*	STORE REQUIRED*	I-O (CHARACTERS) READ	TYPED	PUNCHED	CPU DEMAND* COMP	CPU DEMAND* ASS	TIME REQUIRED* (SEC) COMP	TIME REQUIRED* (SEC) ASS
1	1620	46	479	558	542	2212	0	.09	.08	217.30	215.25
2	7090	77	703	2053	6296	65	9114	.36	.34	149.41	146.99
3	1620	60	529	680	25	0	861	.64	.54	24.01	21.58
4	1620**	162	1152	1672	3914	0	1818	.49	.39	54.90	50.30
5	7090	208	1703	2193	14502	0	4875	.72	.69	222.51	222.21
6	7090	150	1969	2669	26389	0	7566	.37	.35	163.59	170.92
7	1620**	63	565	1351	12100	208	0	.43	.38	58.32	56.12
8	1620**	105	995	1161	3435	0	6900	.33	.28	101.29	99.02
9	1620	57	501	854	143	0	21572	.14	.12	234.23	231.90
10	1620	28	369	552	50	1926	0	.02	.02	176.22	176.13
	TOTALS:	956	8965		67396	4411	52706			1401.78	1390.42

*ESTIMATES FOR CIRRUS
**ORIGINALLY RUN IN TWO OR MORE PARTS

Unfortunately, the program sample is too small for its statistics to be taken as more than a rough guide to what might be expected overall. Though chosen at random, and covering a fair range of scientific problems, the set was by no means fully representative. For example, it contained none of those occasional problems which involve extensive computation but almost no input or output. On the other hand, there were also none of those very trivial programs which occur fairly frequently in the Adelaide University computing centre where large groups of users are each year taught the rudiments of computer programming. For use in the simulation study the sample was no doubt satisfactory.

C.2. The Simulator.

The main input for the simulator was the magnetic tape file of information on the 10 programs of the sample. The order in which these programs were taken by the simulator was determined by a "run schedule" prepared in advance. The order of the schedule, usually about 20 programs long, was set up initially by random selection, except that no program was included twice before all programs had appeared once. The choice whether a given program would be compiled or assembled was also made at

random. The same schedule was purposely used for all runs to avoid variations which could have invalidated comparisons between the slightly different time-sharing procedures simulated.

The schedule was supplied to the simulator on cards, together with other parameters. Card input was as follows:

Card 1: Period at which interruptions were to be made (Runs 4 - 9 only).

Cards 2-11: Table of conversion and waiting times for each type of peripheral. For the reader, three separate classes of input were included: data, C-code and object programs. For each unit, separate times for trivial and non-trivial characters (as defined earlier) were specified.

Card 12: Number of programs to share time.

Cards 13 on: Run schedule (program numbers).

The structure of that portion of the FORTRAN program which simulated time-sharing itself is shown in Fig. C.1. The rest of the simulator was fairly straightforward. When three programs, for example, were to share time, the first three cards of the schedule were read and the corresponding files (or parts thereof if the files were

Notes on Fig. C1:

ELT Elapsed time from origin.

PRT_i Proceed time - time at which the i^{th} program is ready to use CPU time. PRT_i is reset for each input-output delay.

IU_{ij} Unit involved in the particular input-output sequence (j^{th} sequence with i^{th} program).

$(n_T)_{ij}$ Number of instructions in compute sequence (preceding j^{th} input-output sequence, i^{th} program).

IS_{ijk} Number of characters to be transferred in the k^{th} word (or set of trivial characters), j^{th} sequence of input-output, i^{th} program.

BT_i Length of next period for which CPU time is required continuously (i^{th} program).

ST_i Period following BT_i for which the CPU will not be required.

Either $BT_i = (n_T)_{ij} \times 0.000048$, $ST_i = 0$

or $BT_i = T_1$, $ST_i = t_1$

where T_1, t_1 are selected from the table of input-output times (cf. Table 3.3 of text), according to the value of IU_{ij} and whether IS_{ijk} is a number of trivial or non-trivial characters).

TNINT Time at which next regular interruption is due.

PINT Interval between successive interruptions.

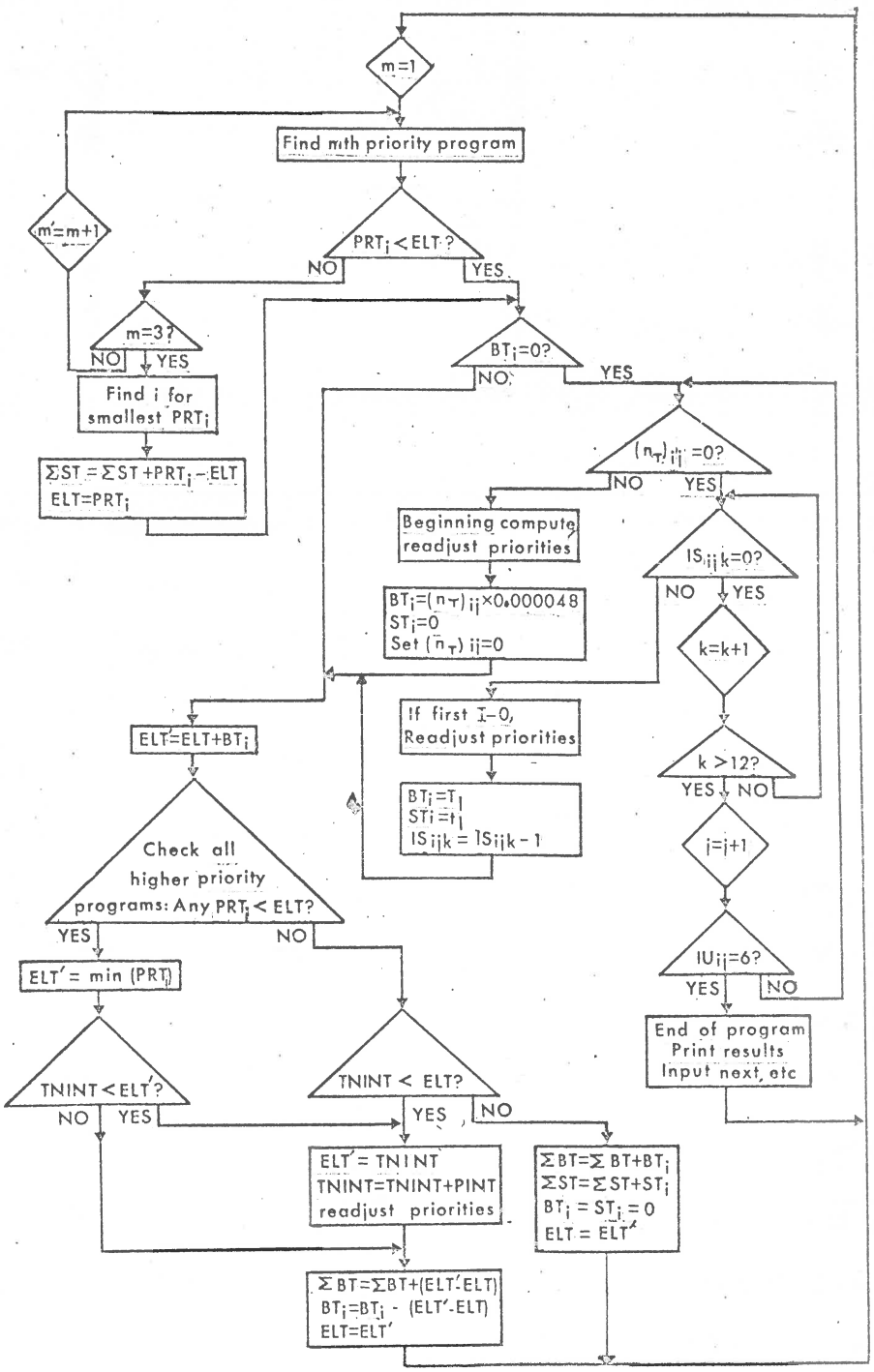


FIG. C1: FLOW DIAGRAM FOR THE TIME-SHARING SECTION OF THE SIMULATOR.
(Priority adjustments as in Run 4)

very long) extracted from the tape. As each program terminated, a further card was read and the relevant file taken from tape. The process was continued until the schedule was exhausted. In all cases except Runs 8 and 9, where an operator delay was introduced before beginning each program, a new program was assumed to start as soon as the preceding one concluded.

Consider each program as comprising successive periods T_i, t_i where

T_i is a period during which the processor is required continuously, and

t_i is a period during which the processor is not required.

Then

$$\sum_{\substack{\text{all}\\\text{programs}}} \sum_{\substack{\text{each}\\\text{program}}} (T_i + t_i)$$

at any stage shows the total amount of work done, which, without time-sharing, would have taken exactly

$$\Sigma\Sigma(T_i + t_i) \text{ sec.}$$

The most important result is the ratio:

$$I = \frac{\Sigma\Sigma(T_i + t_i)}{ET} ,$$

where ET is the current elapsed time. This ratio shows the estimate for the "improvement factor" (defined in Section 4.1) gained by time-sharing.

This result must be considered in relation to the mean utilization

$$\frac{\sum\sum T_i}{\sum\sum (T_i + t_i)}$$

of the work-load, since the demand for time to a large extent determines the degree of improvement possible. The overall utilisation of the processor

$$\frac{\sum\sum T_i}{ET}$$

is also of interest, as it gives some idea of the economics of loading the processor with more programs.

It must be pointed out that the simulation terminated when no cards remained in the run schedule. In the three program case, for example, parts of two programs remained. The composition of what was left varied with different time-sharing procedures. For this reason, the work which had been done was not quite the same in each run even though the run schedule was kept constant.

C.3. Simulation Runs.

Altogether, nine runs were made with the simulator. Each required 30-45 minutes depending on the computer used and the nature of the run. The basic time-sharing procedure was in each case that described in Section 7.3.

The information used in all runs came from the program sample. When this information was used directly, at the point of termination the mean processor utilization $\frac{\sum\sum T_i}{\sum\sum(T_i + t_i)}$ by individual programs was of the order of .3 . However, a number of runs were made to assess the effect of time-sharing when programs made heavier demands on processor time. For these runs, periods of computation were considered to be increased by 4 times, giving a mean demand for time of the order of .5 . (The factor t, by which n_T was multiplied, was increased from .000048 to .000192). The higher demand set of programs is referred to hereafter as "Program Set 2".

Runs 1-4, made for three programs in each case, were aimed at testing variations of the basic time-sharing procedure. These variations are discussed in Section 11.3 where an interpretation of the results is made. The procedure simulated in Run 4 was chosen for use in CIRRUS.

In all subsequent runs, the procedure of Run 4 was used. Run 5 also simulated the three-program case, but used the higher demand Program Set 2. Runs 6 and 7 were made with Program Sets 1 and 2 respectively for the case of two programs. To gauge the effect of operator delays, two further runs, Runs 8 and 9 were made. Each of these

was for 3 programs, using Program Sets 1 (Run 8) and 2 (Run 9). A ten second delay was introduced before beginning operation on each program, i.e.

Processor utilization/program was $\frac{\sum\sum T_i}{10 + \sum\sum (T_i + t_i)}$

Results from Runs 4-9 were discussed in Section 4.1.

A summary of the conditions for each run and of their results is given in Table C.3. The output results from each simulation are shown on the pages following Table C.3.

TABLE C3: SUMMARY OF CONDITIONS AND RESULTS FOR EACH SIMULATION RUN.

NATURE OF RUN				SUMMARY OF RESULTS		
RUN	NO. OF PROGRAMS	PROGRAM SET	TIME-SHARING PROCEDURE	MEAN DEMAND BY WORK-LOAD	PROCESSOR UTILIZATION	IMPROVEMENT FACTOR
1	3	1	Priorities of programs in order of entry	.296	.708	2.40
2	3	1	Priorities adjusted inversely to demands. Lowest priority among programs in computation sequences to the last one to begin computation.	.291	.714	2.46
3	3	1	As Run 2, but with highest priority among programs in computation sequences to the last one to begin computation.	.300	.768	2.57
4	3	1	As Run 3, but with interruptions at 1 second intervals followed by alternation of priorities of programs in computation sequences.	.294	.764	2.61

TABLE C3 (Continued)

NATURE OF RUN				SUMMARY OF RESULTS		
RUN	NO. OF PROGRAMS	PROGRAM SET	TIME-SHARING PROCEDURE	MEAN DEMAND BY WORK-LOAD	PROCESSOR UTILIZATION	IMPROVEMENT FACTOR
5	3	2	As Run 4	.487	.957	1.97
6	2	1	As Run 4	.293	.551	1.89
7	2	2	As Run 4	.518	.814	1.58
8	3	1	As Run 4, but with 10 second operator delay before starting each program.	.277*	.738	2.67
9	3	2	As Run 8	.462*	.938	2.04

* Includes the 10 second operator delay.

SIMULATION OF TIME-SHARING PROCEDURES

RUN 1 PRIORITIES IN ORDER OF ENTRY

PROGRAM SET 1
3 PROGRAM CASE

INCOMING PROGRAM		TERMINATING PROGRAM				DEMAND FIGURES		WORK DONE	
ELAPSED TIME	SERIAL	SERIAL	DEMAND	TIME TO COMPLETE NOMINAL	TIME TO COMPLETE ACTUAL	AVERAGE/ PROGRAM	UTIL OF CPU	(SEC)	RATIO*
0,00	10 A		-0,00	-0,00	0,00	0,000	0,000	0,00	0,00
0,00	3 A		-0,00	-0,00	0,00	0,000	0,000	0,00	0,00
0,00	4 C		-0,00	-0,00	0,00	0,000	0,000	0,00	0,00
23,12	7 A	3 A	0,54	21,58	23,12	0,416	0,976	54,30	2,35
68,62	8 A	4 C	0,49	54,90	68,62	0,285	0,728	170,93	2,58
93,10	6 A	7 A	0,36	56,12	69,96	0,290	0,748	240,16	2,58
176,14	9 A	10 A	0,02	176,13	176,14	0,270	0,720	470,12	2,67
177,90	1 A	8 A	0,28	99,02	109,28	0,271	0,722	473,95	2,67
283,31	2 C	6 A	0,35	170,92	190,20	0,229	0,614	761,91	2,69
414,25	5 A	1 A	0,08	215,25	236,35	0,217	0,594	1135,95	2,75
419,45	6 C	9 A	0,12	231,90	243,31	0,217	0,595	1150,33	2,75
448,45	10 C	2 C	0,36	149,41	163,15	0,224	0,600	1205,50	2,69
640,16	1 A	5 A	0,69	222,21	225,94	0,298	0,694	1494,94	2,34
743,81	4 C	8 C	0,37	163,59	293,35	0,267	0,642	1792,20	2,41
780,88	9 A	10 C	0,02	176,22	276,55	0,264	0,643	1901,38	2,44
801,51	5 A	4 C	0,49	54,90	57,70	0,262	0,638	1957,94	2,45
866,61	7 C	1 A	0,08	215,25	226,43	0,262	0,646	2141,63	2,48
1019,62	2 A	9 A	0,12	231,90	238,74	0,283	0,683	2463,09	2,42
1048,18	8 C	5 A	0,69	222,21	240,67	0,283	0,687	2527,97	2,42
1052,81	3 C	7 C	0,43	58,32	186,20	0,286	0,688	2536,54	2,41
1133,32	2 C	3 C	0,64	24,01	33,75	0,297	0,704	2685,35	2,37
1183,12	10 C	2 A	0,34	146,99	163,20	0,297	0,707	2819,77	2,39
1190,37	3 A	8 C	0,33	101,29	142,19	0,298	0,709	2831,51	2,38
1234,06	6 A	3 A	0,54	21,58	43,69	0,301	0,712	2922,48	2,37
1299,42	7 A	2 C	0,36	149,41	166,10	0,296	0,708	3112,14	2,40

*IMPROVEMENT FACTOR

IMPORTANT NOTE: When the simulator was constructed, it was thought (wrongly) that simultaneous compilation of independent programs would not be possible. Since a program uses in excess of 90% of processor time during compilation, the effect on the results should be negligible. However, a program to be compiled might therefore not be started immediately. "ACTUAL TIME TO COMPLETE" shown above is, in these cases, less than the difference between the elapsed times at which the program is shown as "INCOMING" and "TERMINATING".

SIMULATION OF TIME-SHARING PROCEDURES

RUN 2 PRIORITIES IN INVERSE ORDER TO DEMAND
LOWEST PRIORITY ON COMPLETION OF I/O

PROGRAM SET 1
3 PROGRAM CASE

INCOMING PROGRAM		TERMINATING PROGRAM				DEMAND FIGURES		WORK DONE	
ELAPSED TIME	SERIAL	SERIAL	DEMAND	TIME TO COMPLETE NOMINAL	TIME TO COMPLETE ACTUAL	AVERAGE/ PROGRAM	UTIL OF CPU	(SEC)	RATIO
0,00	10 A		-0,00	-0,00	0,00	0,000	0,000	0,00	0,00
0,00	3 A		-0,00	-0,00	0,00	0,000	0,000	0,00	0,00
0,00	4 C		-0,00	-0,00	0,00	0,000	0,000	0,00	0,00
39,85	7 A	3 A	0,54	21,58	39,85	0,430	0,686	82,12	2,07
63,19	8 A	4 C	0,49	54,90	63,19	0,309	0,728	149,03	2,36
105,62	6 A	7 A	0,38	56,12	65,77	0,267	0,721	260,06	2,52
171,85	9 A	8 A	0,28	99,02	108,67	0,277	0,724	440,98	2,62
189,86	1 A	10 A	0,02	176,13	189,86	0,274	0,721	499,86	2,64
290,79	2 C	6 A	0,35	170,92	185,16	0,225	0,607	786,56	2,71
406,56	5 A	9 A	0,12	231,90	234,70	0,227	0,591	1060,75	2,61
441,09	6 C	2 C	0,36	149,41	150,30	0,229	0,600	1150,59	2,63
462,31	10 C	1 A	0,08	215,25	272,46	0,240	0,618	1196,72	2,58
639,00	1 A	10 C	0,02	176,22	176,68	0,262	0,703	1593,13	2,50
659,43	4 C	5 A	0,69	222,21	252,87	0,262	0,704	1640,92	2,51
716,58	9 A	4 C	0,49	54,90	57,19	0,275	0,695	1612,61	2,53
730,60	5 A	6 C	0,37	163,59	239,51	0,272	0,690	1855,70	2,54
946,96	7 C	1 A	0,08	215,25	307,96	0,292	0,714	2310,65	2,45
957,66	2 A	5 A	0,69	222,21	227,06	0,292	0,714	2344,47	2,45
1045,13	8 C	7 C	0,43	58,32	96,18	0,297	0,720	2533,69	2,43
1091,95	3 C	9 A	0,12	231,90	325,37	0,294	0,715	2658,46	2,44
1105,87	2 C	2 A	0,34	146,99	148,21	0,295	0,716	2690,42	2,44
1125,55	10 C	3 C	0,64	24,01	35,70	0,298	0,720	2724,95	2,43
1204,96	3 A	8 C	0,33	101,29	159,83	0,296	0,717	2923,65	2,43
1228,09	6 A	3 A	0,54	21,58	23,36	0,294	0,715	2991,47	2,44
1262,96	7 A	2 C	0,36	149,41	157,09	0,291	0,713	3092,68	2,45
1302,15	5 C	10 C	0,02	176,22	176,60	0,291	0,714	3200,49	2,46

SIMULATION OF TIME-SHARING PROCEDURES

RUN 3 PRIORITIES IN INVERSE ORDER TO DEMAND
PRIORITY ON COMPLETING I/O ABOVE OTHERS IN COMPUTATION

PROGRAM SET 1
3 PROGRAM CASE

INCOMING PROGRAM		TERMINATING PROGRAM				DEMAND FIGURES		WORK DONE	
ELAPSED TIME	SERIAL	SERIAL	DEMAND	TIME TO COMPLETE NOMINAL	TIME TO COMPLETE ACTUAL	AVERAGE/ PROGRAM	UTIL OF CPU	(SEC)	RATIO
-0,00	10 A		-0,00	-0,00	0,00	0,000	0,000	0,00	0,00
-0,00	3 A		-0,00	-0,00	0,00	0,000	0,000	0,00	0,00
-0,00	4 C		-0,00	-0,00	0,00	0,000	0,000	0,00	0,00
43,12	7 A	3 A	0,54	21,58	43,12	0,382	0,893	100,86	2,34
60,42	8 A	4 C	0,49	54,90	60,42	0,293	0,733	151,32	2,51
106,55	6 A	7 A	0,38	56,12	63,43	0,275	0,720	274,63	2,63
174,60	9 A	8 A	0,28	99,02	114,16	0,272	0,726	467,31	2,68
178,53	1 A	10 A	0,02	176,13	176,53	0,273	0,730	477,31	2,68
289,92	2 C	6 A	0,35	170,92	183,38	0,223	0,610	792,56	2,79
406,14	5 A	1 A	0,08	215,25	227,61	0,218	0,603	1125,37	2,78
418,42	6 C	9 A	0,12	231,90	243,83	0,219	0,607	1156,94	2,77
445,83	10 C	2 C	0,36	149,41	155,90	0,231	0,630	1217,61	2,74
615,94	1 A	6 C	0,37	163,59	197,52	0,268	0,722	1665,12	2,71
624,10	4 C	10 C	0,02	176,22	176,26	0,269	0,718	1688,61	2,71
680,18	9 A	4 C	0,49	54,90	56,08	0,275	0,733	1815,35	2,68
701,84	5 A	5 A	0,69	222,21	295,71	0,271	0,723	1877,27	2,68
859,29	7 C	1 A	0,08	215,25	243,35	0,276	0,750	2316,58	2,70
942,37	2 A	7 C	0,43	58,32	83,08	0,287	0,758	2494,45	2,65
954,26	8 C	9 A	0,12	231,90	274,10	0,285	0,754	2526,60	2,65
961,44	3 C	5 A	0,69	222,21	259,60	0,267	0,756	2539,53	2,65
1025,98	2 C	3 C	0,64	24,01	59,06	0,300	0,768	2629,99	2,57

SIMULATION OF TIME-SHARING PROCEDURES

RUN 4 PRIORITIES IN INVERSE ORDER TO DEMAND
PRIORITY ON COMPLETING I/O ABOVE OTHERS IN COMPUTATION
INTERRUPTIONS AT INTERVALS OF 1 SEC

PROGRAM SET 1
3 PROGRAM CASE

ELAPSED TIME	INCOMING PROGRAM SERIAL		TERMINATING PROGRAM SERIAL		DEMAND	TIME TO COMPLETE NOMINAL	TIME TO COMPLETE ACTUAL	DEMAND FIGURES AVERAGE/ PROGRAM	DEMAND FIGURES UTIL OF CPU	WORK DONE (SEC)	WORK DONE RATIO
-0,00	10	A			-0,00	-0,00	0,00	0,000	0,000	0,00	0,00
-0,00	3	A			-0,00	-0,00	0,00	0,000	0,000	0,00	0,00
-0,00	4	C			-0,00	-0,00	0,00	0,000	0,000	0,00	0,00
43,12	7	A	3	A	0,54	21,56	43,12	0,382	0,893	100,86	2,34
60,42	8	A	4	C	0,49	54,90	60,42	0,295	0,733	151,62	2,51
105,44	6	A	7	A	0,38	56,12	62,02	0,275	0,720	270,17	2,62
177,21	9	A	8	A	0,28	99,02	115,80	0,273	0,729	474,21	2,68
178,53	1	A	10	A	0,02	176,13	176,53	0,273	0,730	477,53	2,68
287,77	2	C	6	A	0,35	170,92	182,34	0,224	0,611	785,82	2,73
408,27	5	A	1	A	0,08	215,25	229,74	0,218	0,600	1122,39	4,76
425,92	6	C	9	A	0,12	231,90	246,71	0,220	0,605	1173,55	2,76
443,57	10	C	2	C	0,36	149,41	155,80	0,226	0,620	1210,82	2,73
616,52	1	A	6	C	0,37	163,59	190,60	0,266	0,717	1661,82	2,70
625,21	4	C	10	C	0,02	176,22	176,69	0,266	0,716	1682,83	2,70
682,61	9	A	4	C	0,49	54,90	57,39	0,275	0,732	1815,25	2,67
704,46	5	A	5	A	0,69	222,21	296,19	0,270	0,721	1881,55	2,68
857,39	7	C	1	A	0,08	215,25	240,87	0,277	0,747	2309,87	2,70
942,39	2	A	7	C	0,43	58,32	85,00	0,287	0,757	2487,04	2,64
956,72	8	C	9	A	0,12	231,90	274,12	0,285	0,753	2526,18	2,65
967,59	3	C	5	A	0,69	222,21	263,13	0,288	0,756	2545,09	2,64
1003,59	2	C	3	C	0,64	24,01	34,06	0,294	0,764	2612,04	2,61

SIMULATION OF TIME-SHARING PROCEDURES

RUN 5 PRIORITIES IN INVERSE ORDER TO DEMAND
PRIORITY ON COMPLETING I/O ABOVE OTHERS IN COMPUTATION
INTERRUPTIONS AT INTERVALS OF 4 SEC

PROGRAM SET 2
3 PROGRAM CASE

ELAPSED TIME	INCOMING PROGRAM SERIAL		TERMINATING PROGRAM SERIAL		DEMAND	TIME TO COMPLETE NOMINAL	TIME TO COMPLETE ACTUAL	DEMAND FIGURES AVERAGE/ PROGRAM	DEMAND FIGURES UTIL OF CPU	WORK DONE (SEC)	WORK DONE RATIO
-0,00	13	A			-0,00	-0,00	0,00	0,000	0,000	0,00	0,00
-0,00	3	A			-0,00	-0,00	0,00	0,000	0,000	0,00	0,00
-0,00	4	C			-0,00	-0,00	0,00	0,000	0,000	0,00	0,00
73,76	7	A	4	C	0,59	68,10	73,76	0,430	0,964	165,63	2,25
87,05	8	A	3	A	0,79	46,97	87,05	0,409	0,954	203,19	2,34
162,93	6	A	7	A	0,57	80,62	89,17	0,382	0,919	392,42	2,41
188,80	9	A	10	A	0,82	176,71	188,80	0,376	0,914	456,89	2,44
351,97	1	A	8	A	0,49	138,25	264,92	0,418	0,936	788,05	2,24
433,02	2	C	6	A	0,51	226,04	270,09	0,383	0,855	1000,38	2,32
522,47	5	A	9	A	0,24	267,54	333,67	0,382	0,891	1219,63	2,34
638,81	6	C	1	A	0,22	253,12	286,54	0,404	0,911	1442,60	2,26
901,71	10	C	6	C	0,53	218,70	262,90	0,465	0,937	1818,39	2,02
1078,79	1	A	10	C	0,33	176,79	177,08	0,460	0,948	2223,02	2,07
1109,14	4	C	2	C	0,61	243,73	676,12	0,460	0,949	2287,78	2,07
1269,94	9	A	4	C	0,59	68,10	180,80	0,462	0,956	2559,50	1,99
1390,95	5	A	5	A	0,88	585,85	668,48	0,468	0,949	2824,25	2,04
1478,39	7	C	1	A	0,22	253,12	399,50	0,462	0,948	3035,85	2,06
1582,97	2	A	7	C	0,60	82,82	104,58	0,465	0,950	3234,83	2,05
1618,77	8	C	9	A	0,24	267,54	328,83	0,466	0,951	3307,29	2,05
1784,46	3	C	6	C	0,52	140,51	165,69	0,481	0,956	3548,79	1,99
1834,40	2	C	3	C	0,83	49,40	49,94	0,487	0,957	3607,26	1,97

SIMULATION OF TIME-SHARING PROCEDURES

RUN 6 PRIORITIES IN INVERSE ORDER TO DEMAND
PRIORITY ON COMPLETING I/O ABOVE OTHERS IN COMPUTATION
INTERRUPTIONS AT INTERVALS OF 1 SEC.

PROGRAM SET 1
2 PROGRAM CASE ...

INCOMING PROGRAM		TERMINATING PROGRAM			TIME TO COMPLETE		DEMAND FIGURES		WORK DONE	
ELAPSED TIME	SERIAL	SERIAL	DEMAND		NOMINAL	ACTUAL	AVERAGE/ PROGRAM	UTIL OF CPU	(SEC)	RATIO
-0,00	10 A				-0,00	0,00	0,000	0,000	0,00	0,00
-0,00	3 A				-0,00	0,00	0,000	0,000	0,00	0,00
22,96	4 C	3	A	0,54	21,56	22,96	0,306	0,593	44,62	1,95
78,15	7 A	4	C	0,49	54,90	55,19	0,264	0,523	154,70	1,98
134,53	8 A	7	A	0,38	56,12	56,38	0,235	0,466	267,17	1,99
176,16	6 A	10	A	0,02	176,13	176,16	0,220	0,437	350,25	1,99
244,79	9 A	8	A	0,26	99,02	110,26	0,270	0,519	471,26	1,93
354,81	1 A	6	A	0,35	170,92	178,65	0,240	0,465	686,55	1,94
460,40	2 C	9	A	0,12	231,90	235,60	0,205	0,399	935,17	1,95
574,87	5 A	1	A	0,08	215,25	220,06	0,219	0,425	1116,68	1,95
631,47	6 C	2	C	0,36	149,41	151,07	0,232	0,451	1220,69	1,95
806,46	10 C	6	C	0,37	165,59	172,00	0,293	0,569	1516,09	1,88
854,77	1 A	5	A	0,69	222,21	279,89	0,290	0,545	1609,21	1,89
962,93	4 C	10	C	0,02	176,22	176,47	0,257	0,488	1865,59	1,90
1035,28	9 A	4	C	0,49	54,90	55,37	0,259	0,490	1968,77	1,90
1077,34	5 A	1	A	0,08	215,25	222,57	0,253	0,480	2049,40	1,90
1277,18	7 C	9	A	0,12	231,90	236,90	0,280	0,531	2424,56	1,90
1322,14	2 A	5	A	0,69	222,21	244,80	0,283	0,536	2504,96	1,90
1341,83	8 C	7	C	0,43	58,32	64,65	0,285	0,540	2538,77	1,90
1444,96	3 C	8	C	0,33	101,29	105,13	0,293	0,551	2721,90	1,89

SIMULATION OF TIME-SHARING PROCEDURES

RUN 7 PRIORITIES IN INVERSE ORDER TO DEMAND
PRIORITY ON COMPLETING I/O ABOVE OTHERS IN COMPUTATION
INTERRUPTIONS AT INTERVALS OF 4 SEC.

PROGRAM SET 2
2 PROGRAM CASE

	INCOMING PROGRAM		TERMINATING PROGRAM				DEMAND FIGURES		WORK DONE	
ELAPSED TIME	SERIAL		SERIAL	DEMAND	TIME TO COMPLETE NOMINAL	TIME TO COMPLETE ACTUAL	AVERAGE/ PROGRAM	UTIL OF CPU	(SEC)	RATIO
=0,00	10	A		=0,00	=0,00	0,00	0,000	0,000	0,00	0,00
=0,00	3	A		=0,00	=0,00	0,00	0,000	0,000	0,00	0,00
49,27	4	C	3	A 0,79	46,97	49,27	0,414	0,808	96,28	1,96
117,77	7	A	4	C 0,59	68,10	68,50	0,345	0,682	232,90	1,98
176,71	8	A	10	A 0,02	176,71	176,71	0,317	0,627	350,38	1,99
285,67	6	A	7	A 0,57	80,62	87,90	0,351	0,669	392,23	1,91
394,85	9	A	8	A 0,49	138,25	218,14	0,437	0,759	686,00	1,74
446,56	1	A	6	A 0,51	226,04	240,89	0,409	0,720	786,57	1,77
675,15	2	C	9	A 0,24	267,54	280,30	0,346	0,626	1221,84	1,81
717,36	5	A	1	A 0,22	253,12	270,79	0,350	0,632	1296,39	1,81
1196,21	6	C	2	C 0,61	243,73	521,06	0,495	0,780	2885,60	1,58
1416,21	10	C	6	C 0,53	218,70	220,00	0,515	0,804	2212,10	1,57
1513,54	1	A	5	A 0,68	585,85	796,18	0,505	0,801	2402,19	1,59
1593,79	4	C	10	C 0,03	176,79	177,58	0,485	0,779	2562,40	1,61
1664,58	9	A	4	C 0,59	68,10	70,79	0,480	0,775	2687,37	1,62
1790,73	5	A	1	A 0,22	253,12	277,20	0,457	0,747	2924,39	1,64
1900,54	7	C	9	A 0,24	267,54	295,76	0,464	0,761	3213,08	1,64
2050,04	2	A	7	C 0,60	82,82	89,49	0,473	0,770	3339,34	1,63
2394,31	8	C	2	A 0,60	241,31	344,27	0,509	0,803	3780,71	1,58
2535,51	3	C	8	C 0,52	140,51	141,21	0,518	0,814	3989,83	1,58

SIMULATION OF TIME-SHARING PROCEDURES

RUN 8 PRIORITIES IN INVERSE ORDER TO DEMAND
PRIORITY ON COMPLETING I/O ABOVE OTHERS IN COMPUTATION
INTERRUPTIONS AT INTERVALS OF 1 SEC.
10 SECOND OPERATOR DELAY BEFORE EACH RUN

PROGRAM SET 1
3 PROGRAM CASE

	INCOMING PROGRAM		TERMINATING PROGRAM				DEMAND FIGURES		WORK DONE	
ELAPSED TIME	SERIAL		SERIAL	DEMAND	TIME TO COMPLETE NOMINAL	TIME TO COMPLETE ACTUAL	AVERAGE/ PROGRAM	UTIL OF CPU	(SEC)	RATIO
-0,00	10	A		-0,00	-0,00	0,00	0,000	0,000	0,00	0,00
-0,00	3	A		-0,00	-0,00	0,00	0,000	0,000	0,00	0,00
-0,00	4	C		-0,00	-0,00	0,00	0,000	0,000	0,00	0,00
53,12	7	A	3	A	0,54	21,58	53,12	0,294	0,725	130,86	2,47
69,94	8	A	4	C	0,49	54,90	59,94	0,242	0,625	180,94	2,59
124,41	6	A	7	A	0,38	56,12	71,29	0,228	0,610	333,76	2,69
184,53	9	A	10	A	0,02	176,13	188,53	0,232	0,629	512,42	2,72
193,60	1	A	8	A	0,28	99,02	129,66	0,231	0,631	535,16	2,74
316,51	2	C	6	A	0,35	170,92	192,10	0,202	0,558	875,40	2,77
432,53	5	A	1	A	0,08	215,25	236,93	0,203	0,563	1203,94	2,79
444,36	6	C	9	A	0,12	231,90	255,03	0,200	0,556	1238,88	2,79
462,41	10	C	2	C	0,36	149,41	165,90	0,212	0,583	1320,17	2,78
646,66	1	A	6	C	0,37	163,59	202,29	0,250	0,679	1761,13	2,73
673,34	4	C	10	C	0,02	176,22	190,93	0,248	0,677	1840,11	2,74
736,97	9	A	4	C	0,49	54,90	65,63	0,251	0,682	2007,95	2,72
739,72	5	A	5	A	0,69	222,21	307,19	0,250	0,681	2014,51	2,74
884,59	7	C	1	A	0,08	215,25	237,90	0,257	0,700	2416,39	2,74
994,42	2	A	7	C	0,43	58,32	69,87	0,267	0,722	2585,03	2,71
995,96	8	C	9	A	0,12	231,90	257,99	0,267	0,721	2695,53	2,71
1042,16	3	C	5	A	0,69	222,21	302,44	0,274	0,733	2789,50	2,68
1061,60	2	C	3	C	0,64	24,01	39,44	0,277	0,738	2883,37	2,67

SIMULATION OF TIME-SHARING PROCEDURES

RUN 9 PRIORITIES IN INVERSE ORDER TO DEMAND
PRIORITY ON COMPLETING I/O ABOVE OTHERS IN COMPUTATION
INTERRUPTIONS AT INTERVALS OF 4 SEC,
10 SECOND OPERATOR DELAY BEFORE EACH RUN

PROGRAM SET 2
3 PROGRAM CASE

ELAPSED TIME	INCOMING PROGRAM SERIAL	TERMINATING PROGRAM SERIAL	DEMAND	TIME TO COMPLETE NOMINAL	TIME TO COMPLETE ACTUAL	DEMAND FIGURES AVERAGE/ PROGRAM	DEMAND FIGURES UTIL OF CPU	WORK DONE (SEC)	WORK DONE RATIO
=0,00	10 A		=0,00	=0,00	0,00	0,000	0,000	0,00	0,00
=0,00	3 A		=0,00	=0,00	0,00	0,000	0,000	0,00	0,00
=0,00	4 C		=0,00	=0,00	0,00	0,000	0,000	0,00	0,00
85,13	7 A	4 C	0,59	68,10	85,13	0,364	0,847	198,38	2,34
96,46	8 A	3 A	0,79	46,97	96,46	0,353	0,848	231,90	2,41
184,97	6 A	7 A	0,57	80,62	99,84	0,334	0,827	458,02	2,48
198,52	9 A	10 A	0,02	175,71	198,82	0,324	0,810	497,89	2,51
367,76	1 A	8 A	0,49	138,25	271,30	0,387	0,880	837,97	2,28
465,10	2 C	6 A	0,51	226,04	280,13	0,357	0,843	1100,09	2,37
543,39	5 A	9 A	0,24	267,54	344,27	0,354	0,842	1292,30	2,38
665,55	6 C	1 A	0,22	253,12	2.7,89	0,377	0,871	1540,07	2,32
939,92	10 C	6 C	0,53	218,70	274,27	0,441	0,909	1937,92	2,07
1126,73	1 A	10 C	0,03	176,79	136,81	0,441	0,924	2364 48	2,10
1154,40	4 C	2 C	0,61	243,73	689,30	0,439	0,926	2433,94	2,11
1330,29	9 A	4 C	0,59	68,10	175,89	0,461	0,936	2699,46	2,03
1440,60	5 A	5 A	0,88	585,85	897,51	0,449	0,931	2987,86	2,08
1540,41	7 C	1 A	0,22	253,12	413,68	0,442	0,927	3232,11	2,10
1651,82	2 A	7 C	0,60	82,82	111,41	0,446	0,931	3453,17	2,10
1560,95	8 C	9 A	0,24	267,54	330,65	0,445	0,931	3478,67	2,10
1856,08	3 C	8 C	0,52	140,51	195,13	0,462	0,938	3774,17	2,04

BIBLIOGRAPHY

Adams Associates, Inc.(1964):- Computer Characteristics Quarterly, March 1964.

Allen M.W., Pearsey T., Penny J.P., Rose G., Sanderson J.G. (1963):- CIRRUS, an economical multiprogram computer with microprogram control. IEEE TEC Special Issue on Computer Systems, Dec. 1963.

Allen M.W., Rose G.A. (1963):- A flexible and economic approach to digital system design. Proc. of Conf. of Aust. Nat. Committee for Computation and Automatic Control, March 1963.

Amdahl G.M. (1962):- New concepts in computing system design. Proc. IRE, May 1962.

Anderson J.P., Hoffman S.A., Shifman J., Williams R.J., (1962):- D-825 - A multiple computer system for command and control. Proc. AFIPS Fall JCC, 1962.

Bauer W.F. (1959):- Computer design from the programmer's viewpoint. Proc. EJCC, July 1959.

Beckman F.S., Brooks F.P., Lawless W.J. (1961):- Developments in the logical organisation of computer arithmetic and control units. Proc. IRE, Jan. 1961.

Boyell R., (1960):- Analysis of time-sharing in digital computers. J. Soc. Indust. and Applied Maths., March 1960.

Bright H.S., Cheydleur B.F. (1962):- On the reduction of turnaround time. AFIPS Fall JCC, 1962.

Bucholz W., (editor, 1962):- Planning a Computer System - Project Stretch. McGraw Hill Book Co. Inc.

Butcher I.R. (1964):- A pre-wired storage unit. IEEE TEC, April 1964.

Codd E.F., Lowry E.S., McDonough E., Scalzi C.A. (1959):- Multiprogramming STRETCH: Feasibility Considerations. Comm. ACM, Nov. 1959.

Codd E.F. (1961):- Multiprogram scheduling. Parts 1 and 2, (Introduction and theory), Comm. ACM, June 1960. Parts 3 and 4, (Scheduling algorithm and external constraints), Comm. ACM, July 1960.

Control Data Corporation (1963):- Control Data 3200 Computer System - Preliminary Reference Manual, Oct. 1963.

Control Data Corporation (1963):- Control Data 6600 Computer System - Reference Manual (1st Edition), Aug. 1963.

Control Data Corporation (1964):- 3600 FORTRAN Reference Manual, Feb. 1964.

Cook R.L. (1961):- Time-sharing on the National Elliott 802. Comp. J., Jan. 1960.

Corbato F.J., Merwin-Daggett M., Daley C. (1962):- An Experimental Time-sharing System. AFIPS Spring JCC, May 1962.

Critchlow A.J. (1963):- Generalized multiprocessing and multiprogramming systems. Proc. AFIPS Fall JCC, 1963.

Dace D.J. (1963):- Experience with data transmission. Comp. J., April, 1963.

Dreyfus P. (1958):- Programming design features of the GAMMA 60 computer. Proc. EJCC, Dec. 1958.

Fano R.M. (1963):- A Proposal for a research and development program on computer systems, to the Advanced Research Projects Agency from the Massachusetts Institute of Technology.

Ferranti Ltd. (1960):- Provisional description of the Ferranti Orion Computer System (Second Edition), April 1960.

Gill S. (1958):- Parallel Programming. Comp. J., 1958.

Haley A.C.D. (1962):- The KDF-9 Computer System. Proc. AFIPS Fall JCC, 1962.

Harper S.D. (1960):- The Honeywell approach to low-cost data processing. Published by Minneapolis-Honeywell Regulator Co.

Harper S.D. (1960):- Automatic parallel processing. Delivered to Second Conference, Computing and Data Processing Society of Canada, June 1960.

Howarth D.J., Jones P.D., Wyld M.T. (1963):- The ATLAS scheduling system. Proc. AFIPS Spring JCC, 1963.

Huskey H., Korn S. (1962):- The Computer Handbook, McGraw-Hill Book Co.

Kilburn T., Howarth D.J., Payne R.B., Sumner F.H. (1961a):- The Manchester University Atlas Operating System. Part I: Internal Organisation. Part II: User's Description. Comp. J., Oct 1961.

Kilburn T., Payne R.B., Howarth D.J. (1961b):- The ATLAS Supervisor. Proc. EJCC, Dec. 1961.

Kilburn T., Edwards D.B.C., Lanigan, M.J., Sumner F.H. (1962a):- One level storage system. IRE TEC, April 1962.

Kilburn T., Howarth D.J., Payne R.B., Sumner F.H. (1962b):- The Manchester University Atlas Operating System. Comp. J., July 1962.

Landos N., Maros A., Turner L.R. (1962):- Initial experience with an operating multiprogramming system. Comm. ACM, May 1962.

Lehmann M., Netter Z., Eshed R. (1963):- SABRAC, a time-sharing low-cost computer. Comm. ACM, Aug. 1963.

Lewis J.W. (1963):- Time-sharing on LEO III. Comp. J., April 1963.

Licklider J.C.R. (1960):- Man-Computer Symbiosis. IRE Transactions on Human Factors in Electronics, March 1960.

Licklider J.C.R., Clark W.E. (1962):- On-line man-computer communication. Proc. AFIPS Spring JCC, 1962.

Lourie N., Schrimpf H., Reach R., Kahn W. (1960):- Arithmetic and control techniques in a multi-program computer. Minneapolis-Honeywell Regulator Co.

Lucking J.R., O'Neil J.P. (1963):- The time-sharing facilities of the KDF-9 Computer. Presented to Aust. Nat. Conf. on Comp. and Auto. Control, March 1963.

McCarthy J. et al, (1961):- Report of the Long-Range Computation Study Group. Massachusetts Inst. of Technology, April 1961.

McCarthy J., Boilen S., Fredkin E., Licklider J.C.R. (1963):- A time-sharing debugging system for a small computer. Proc. AFIPS Spring JCC, 1963.

McDonough E.S. (1959):- Multiprogramming STRETCH - Feasibility considerations. Comm. ACM, Nov. 1959.

McDonough E.S. (1962):- STRETCH experiment in multi-programming. ACM Nat. Conf., Sept. 1962.

Marcotty M.J., Longstaff F.M., Williams A.P.M. (1963):- Time-sharing on the Ferranti-Packard FP6000 computer system. Proc. AFIPS Spring JCC, 1963.

Mills M.R. (1963):- Operational experience of time-sharing and parallel processing. Comp. J., April 1963.

Minneapolis-Honeywell Regulator Company (1961):- Executive System manual for the Honeywell 800.

Nash J.A. (1962):- Time-sharing Aspects of the STRETCH computer. In Wegner Peter (Ed): Introduction to System Programming.

Nekora M.R. (1961):- Comment on a paper on parallel processing. Comm. ACM, Feb. 1961.

Penny J.P. (1960):- Use of multiprogramming in the design of a digital computer. Thesis submitted to the University of Adelaide for the degree of Master of Science, Nov. 1960.

Penny J.P., Pearcey,T. (1962):- Use of multiprogramming in the design of a low-cost digital computer. Comm. ACM, Sept. 1962.

Penny J.P. (1963):- The CIRRUS multiprogram system. Proc. of Conf. ofAust. Nat. Committee for Computation and Automatic Control, March 1963.

Penny J.P., *et al.* (1963):- See Allen *et al.* (1963).

Radin G., Rogoway H.P. (1964):- NPL: Highlights of a New Programming Language. IBM Technical Report TR00.1211, Nov. 1964.

Ryle B.L. (1961):- Multiple programming data processing. Comm. ACM, Feb. 1961.

Sanderson J.G. (1963):- CIRRUS A-Code Manual, July 1963.

Sanderson J.G. (1964):- CIRRUS C-Code Manual, May 1964.

Strachey C. (1959):- Time-sharing in large, fast computers. Paper to International Conf. on Information Processing, Paris, France, Jan. 1959.

Van Horn E.C. (1964):- Processes, spheres of protection and independent computations. Memorandum MAC-M-200, Massachusetts Institute of Technology.

Weik M.H. (1957):- A second survey of domestic electronic computing systems. Ballistic Research Labs. Report No. 1010, June 1957.

Weil J.W. (1962):- A heuristic for page-turning in a multiprogram computer. Comm. ACM, Sept. 1962.