INTERNATIONAL COMPUTERS AND TABULATORS

A GENERAL DESCRIPTION OF THE ORION 1 AND ORION 2 COMPUTERS

68 NEWMAN STREET,
LONDON, W.1.

LIST CS 391
SEPTEMBER 1963

CONTENTS

A GENERAL DESCRIPTION OF THE ORION 1 AND ORION 2 COMPUTERS

1. INTRODUCTION

This document contains a description of the I.C.T. Orion Computer System and is intended for those who are already familiar to some extent with digital computing systems in general.

The name 'Orion' in fact refers to two different designs of computer, Orion 1 and Orion 2. To the user, the principal difference between these two models lies in the speed of internal calculation, Orion 2 being the faster by a factor of about 5 on average. The two versions are identical as far as programming is concerned, but there are slight differences in the ranges of peripheral devices which can be attached to the computers.

Both Orion 1 and Orion 2 are high-performance systems of medium to large size; for both versions, systems of various sizes are offered. They find a wide range of application in business and commerce as well as in the scientific field. Applications currently implemented or planned include: -

stock control,

nuclear research,

life assurance work,

statistical calculation,

banking,

sales accounting and sales analysis,

payroll,

invoicing,

production control,

a range which indicates the versatility of the system*, a versatility which stems from three features of the system:

(a) in the centre is a fast general-purpose computer provided with stores of large capacity,

(b) round this can be grouped many peripheral devices to provide input, output and bulk storage,

(c) there is a comprehensive time-sharing system which allows several programs to be run concurrently; this virtually eliminates the need for special-purpose off-line equipment and greatly improves the efficiency of the whole system.

The system is expandable, the design being such that one can easily add more peripheral devices or more storage capacity at any time. This feature allows the capacity of an installation to be increased, without much disturbance, to meet additional requirements.

---

* Orion 1 and Orion 2 are so similar that, in what follows, they will often be regarded and discussed as a single design. When this is not possible they will be described separately.

The system is very reliable; there is a high level of checking in the storage and transfer of information.

A packaged modular construction method is used in the electronic equipment; this has many well-known advantages in design, manufacture and maintenance. The plug-in packages take the form of printed boards on which transistors, germanium diodes and ferrite-core transformers are the principal components.

The design is a result of very close collaboration between engineers, logical designers and programmers. The central computer has a comprehensive repertory of instructions and there are many advanced features. Fast floating-point operations are included. There are Built-in Programs to help the operators and programmers, including a comprehensive program input and assembly routine. There is a Library of basic subroutines and utility programs. Programming languages include two machine-oriented assembly schemes and three automatic programming languages, namely:-

Nebula, for commercial applications,

Fortran, for scientific applications,

Extended Mercury Autocode, also for scientific work.

The time-sharing features make for the maximum use of the whole system, with a complete embargo on interference between programs. A very important point to note is that the programmer does not have to allow for time-sharing when writing a program. From its beginning, Orion has been designed as a time-sharing computer.

In a time-sharing system of the speed and size of Orion, data must be presented to and taken from the computer at a high rate, and the problems of handling and identifying the data become significant. Delays and confusion in these actions could lead to unacceptably wasteful use of the system. Accordingly a comprehensive operating system has been devised for Orion, minimising the chances of delay and confusion.

2. THE CENTRAL COMPUTER

2.1 General

The Orion central computer is a parallel binary machine with packaged transistor logical circuits, a ferrite-core matrix as immediate-access working-store, and a backing-store composed of, usually, several magnetic drums.

Binary number representation is adopted in Orion, as in virtually all high-performance systems, because, in comparison with binary representation of decimal characters, it enables the logical equipment to be smaller, faster and cheaper, and affords savings of nearly 50% in the storage capacity required for a given amount of numerical data. A similar saving is realised on magnetic tape, which can lead in turn to a comparable saving in the time required for magnetic-tape-limited operations, i.e., those whose speed is effectively that of reading or writing magnetic tape. In terms of overall performance, these considerations heavily outweigh the disadvantages of radix conversion during input and output. Moreover, the user is not handicapped by lack of familiarity with binary notation since the latter is almost entirely concealed from him by modern programming and monitoring techniques.

The logical equipment of the central computer is supplemented by a set of built-in program routines, collectively known as the Monitor Program, for controlling time-sharing and carrying out a variety of other organisational operations which cannot conveniently or economically be provided by hardware. Monitor Program operations can be initiated by the operator, by the programmer, or by the computer itself.

In earlier computers, monitoring displays usually took the form of cathode-ray tubes, digital displays and the like. If a program failed and could not continue, the usual practice was to stop the computer while the programmer attempted to diagnose the error, using handswitches and the information given by the monitoring displays; interpretation of these displays was often difficult and nearly always time-consuming.

In Orion such methods are not used. Instead, in the event of a program failure, the Orion Monitor Program determines the nature of the failure and presents the details to the operator and programmer in a permanent, printed form which is immediately intelligible. Indeed, such actions are not confined to true program failures; if he wishes, the programmer may cause the Monitor Program to present information automatically whenever particular, normal events occur in the program - a powerful aid to program-development.

Moreover, if a program does fail and cannot be allowed to continue, only that program is stopped; the other programs with which it has been time-sharing continue. The time which the failed program would have used is made available to the other programs, so that no time is wasted.

This section is concerned only with the equipment of the central computer, and does not deal with peripheral equipment, which is described later in Section 4. However, a certain minimum of peripheral equipment is required for the central computer to be usable. This minimum consists of a paper-tape reader, a paper-tape punch, a Flexowriter for control and monitoring and a magnetic drum; of course, most installations have far more than this.

2.2 The Working-store

The working-store is a ferrite-core matrix and is available in various sizes: it is made up of a number of sections of 4096 registers each. Minimum and maximum numbers of registers are: -

	Orion 1	Orion 2
Minimum	8192	8192
Maximum	16384	32768

This division into sections is purely a matter of the construction of the store and does not affect its use; in particular, the division is not apparent to the programmer.

The working-store cycle times are:

Orion 1:	12 microsecs,
Orion 2:	2 microsecs;

that is, at least the stated cycle-time must elapse between successive references to the store to insert or extract a word. The time lapse is taken care of automatically by the logic design of the computer, and the programmer is not required to consider the point when writing his program.

Each register holds one computer-word, with 48 bits accessible to the programmer. A 49th bit is associated with each register for automatic parity checking; this check is applied whenever a word is transferred and, if the check is not satisfied, the computer is stopped.

Since Orion is a time-sharing computer, the working-store will usually be occupied simultaneously by several independent programs. For this purpose, each program is allocated its own part of the store. Three important features are:-

(i) the store is allocated according to the requirements of the various programs sharing it: the division of the store is in no way prescribed by the design of the computer except that the number of registers allocated to each program is a multiple of 64,

(ii) programs are not constrained to occupy particular registers; the programming languages used allow a program to occupy any suitably sized set of consecutive registers which happens to be free when the program is presented to the computer,

(iii) to the programmer, the store region occupied by his program consists of one continuous set of registers with consecutive addresses.

Associated with the working-store is a set of pseudo-registers which can be read from by all programs impartially. Each stores a prescribed 48-bit quantity; the quantities include Local Civil Time, representations of the state of the overflow indicator, and certain numerical and logical constants useful to the programmer.

2.3 The computer word

As stated above, the Orion word-length is 48 bits. A word can be regarded by the programmer in several ways, of which the most common are: -

(i) a signed integer, stored with complete precision and lying in the range -2^{47} to 2^{47} - 1(=140,737,488,355,327). This permits storage, as pence, of amounts up to about £586,000,000,000;

(ii) a signed fraction, stored to a precision of about 15 decimal places and lying in the range -1.0 to 1.0 -2^{-47};

(iii) a standard, packed, floating-point number, stored to a precision of about 12 significant decimal digits and lying in the range of about $\pm 10^{38}$;

(iv) a set of 8 alphanumeric characters, each stored as 6 bits according to some arbitrary code;

(v) a 48-bit logical quantity (e.g. an "and" mask);

(vi) an instruction.

Well established and convenient programming techniques permit several small items of data to be packed into one word, while other techniques allow one number to be stored in two or more words, increasing the range of magnitudes or the precision or both.

3. INSTRUCTIONS

Basically, the Orion instruction is of 3-address type; that is, an instruction specifies the addresses of three registers, two of which contain the operands while the third receives the result of the operation. This means that the typical operation of, say, adding two numbers together and storing their sum without overwriting either of the original numbers requires only one instruction. This compares with, say, a computer with a single-address instruction-code which must obey, usually, three instructions to achieve the same effect. This point should be borne in mind when considering the instruction-times given later.

Another commonly required effect is to perform some operation using two operands and to overwrite one of the operands by the result. This is possible on Orion using a single instruction, since most instructions are usable in both 2- and 3-address forms; in essence, for the 3-address form the programmer writes three addresses, while for the 2-address form he need write only two addresses.

The instruction repertory for Orion is unusually wide. It includes the normal arithmetical operations (addition, subtraction, multiplication, division, shifting); logical operations (and, or, not equivalent); instructions to facilitate manipulation of packed data; conversion of data between character-form and binary, the assembly of data in character form for output; a wide selection of jump instructions, including those for counting and for entering and leaving subroutines; the use of the input and output devices. Special facilities are included for dealing with tables and lists of data.

Fast, automatic arithmetic with floating-point numbers is included, together with conversion between fixed-point and floating-point forms.

There are very versatile facilities for the modification of addresses in instructions and for indirect addressing.

A brief survey of the instruction repertory is given in section 6 below while a full description is given in List CS374 "The Orion Programming Manual". The various programming languages available are discussed briefly in Section 7 below.

3.1 Instruction times

Orion instructions occupy only the minimum time necessary for the processes they involve and are not forced into a fixed rhythm as is necessary in serial computers; this makes it difficult for various reasons to summarise times for addition, multiplication, etc.

In the first place, the comprehensive instruction repertory of Orion exploits basic arithmetic operations in a variety of ways to give, for example, rounded or unrounded multiplication and division, division with single- or double-length dividend giving single-or double-length quotient, and logical or arithmetical shifts with single- or double-length operands; all such variants inherently have different timing.

Secondly, the time required to execute an instruction depends on the number of store references required and hence on whether modification of addresses is called for, on whether one operand is a constant stored within the instruction itself rather than a variable in a register, etc.

Also the times depend somewhat on the values of the operands themselves and on what other operations may be happening.

These factors are taken into account in the following tables, which give the approximate ranges of times, in microsecs, for various basic operations.

ORION 1

Operation	**Fixed-point**	**Floating-point**
Addition, subtraction	36 to 68	76 to 92
Elementary logical operation	36 to 68	
Multiplication	100 to 200	176 to 200
Accumulative multiplication	183 to 212	
Division	about 500 average	624 to 640
Conditional jump	20 to 62	
Convert 8 characters to binary	336 to 356	

ORION 2

Operation	Fixed-point	Floating-point
Addition, subtraction	8 to 10	17 to 70
Elementary logical operation	10	
Multiplication	35 to 39	45
Accumulative multiplication	43 to 45	
Division	118 to 233	
Conditional jump	7 to 12	
Convert 8 characters to binary	52 to 223	

4. THE PERIPHERAL EQUIPMENT

4.1 The Types of Peripheral Device

For commercial data-processing particularly, great importance attaches to the speed with which data can be read into and emitted by the computer. To this end, Orion has been designed to utilise a very wide range of input and output devices, including the fastest peripheral machines of each type currently available.

The range of peripheral devices currently offered includes: -

Ampex TM2 magnetic-tape units (90,000 characters/second)

I.C.T. 593 card-reader (600 cards/minute)

I.C.T. 582 card-punch (100 cards/minute)

I.C.T. 665 line-printer (600 lines/minute) (Orion 1 only)

Anelex Type 4-1000 line-printer (1000 lines/minute)

Rank Xeronic line-printer (3000 lines/minute) (Orton 1 only)

I.C.T. TR5 paper-tape reader (300 characters/second)

I.C.T. TR7 paper-tape reader (1000 characters/second)

Creed 25 paper-tape punch (33 characters/second)

Creed 3000 paper-tape punch (300 characters/second)

Teletype paper-tape punch (60 or 110 characters/second)

There are also a backing-store composed of magnetic-drums, with a capacity of 16,384 words each, and a Friden Flexowriter which is not an input/output device in the normal sense, but is used for controlling the operation of the system.

Within limits, any selection of peripheral devices can be supplied; one limitation is that each Orion must have at least one drum, one paper-tape reader, one paper-tape punch and a Flexowriter.

As and when other suitable devices become available they, too, can be fitted to Orion; this is simplified by the unified method of connecting devices to the computer.

Various special devices are, in fact, included in existing or planned Orion installations.

4.2 Peripheral Control Units

When a peripheral device is operating it can be thought of either as reading, from some medium such as punched tape or magnetic tape, data which it places directly in the working-store of the computer, or as writing on to some medium data which it has extracted from the working-store. These two operations are called reading and writing transfers, respectively.

In fact each device (or, in Orion 1, group of similar devices) is connected to a peripheral control unit, which communicates with the working-store and manages the details of the transfer.

4. 3 Peripheral Transfers, Hesitations

Peripheral transfers have, in general, the following three properties:

(a) they are of variable length,

(b) they are to or from any part of the working-store,

(c) they are autonomous.

These properties mean firstly that each transfer can, as a rule, be made up of any number of words or, if appropriate, characters; secondly, that the programmer can nominate any part of the working store to receive or emit the data without an intermediate transfer into some fixed and prescribed buffer; and thirdly, that a program continues to be obeyed after it has initiated a transfer, which can then proceed independently of the computer proper.

Prom time to time during a peripheral transfer the control unit concerned will require access to the working-store to extract or insert a single word or character. Should the computer happen to require access to the store at the same instant then the computer is made to hesitate, i.e, it gets held up, until the peripheral control unit has made its reference to the store. These hesitations may slow the computer slightly, but the effect is significant only when the fastest devices are in operation (for Orion 1 there is an average reduction in speed of at most about 15% during a magnetic tape transfer, and 8½% during a drum transfer, with appropriately smaller percentages for Orion 2).

4.4 Lockouts and Interruptions

Parts of the working-store currently involved in a peripheral transfer are locked out, i.e. the computer is prevented from referring to them, until the transfer is completed. This simplifies the task of the programmer and removes a potential source of error. For example, a program is not allowed to try to use data from a punched card before these data have been read in and checked. Every tine the computer requires access to the working store it checks automatically that the address concerned is not locked-out. If the carrying out of an instruction by the computer would involve access to a locked-out register then the instruction is not obeyed at that time; instead the program is interrupted and subsequent events occur which are described in the section on Time-sharing below (Section 4.7).

On Orion 1, the lockouts for all peripheral devices are effected by hardware (i.e. by the design of the computer's logic) whereas for Orion 2 this is true only of transfers involving magnetic-tape and the drum-store. In fact, on Orion 2, the lockouts, and indeed the peripheral transfers themselves, for the other devices are not effected directly by hardware but by special built-in programs, the ‘extracodes’, which use normal program-instructions to produce the desired effects.

The region of the store locked-out by a peripheral transfer is, with many peripheral devices, progressively reduced in size, one word at a time, as the transfer proceeds;

this allows the computer to have access to those parts of the store no longer needed by the peripheral device. Before any program is allowed to initiate a peripheral transfer the computer automatically tests whether the region of the working-store involved overlaps any locked-out regions associated with other devices; if so then the program is interrupted and the transfer is not started. A similar lockout also occurs if an attempt is made to use a peripheral device which is already concerned in a transfer, i.e. a busy device.

4.4.1 Strong and Weak Lockouts. The foregoing description of lockouts applies strictly only to what are known as strong lockouts, i.e. lockouts applying to reading transfers; no program is allowed to use or write over any information in a strongly locked-out register or to initiate any peripheral transfer involving such a register. This is the situation described above. Writing transfers are, however, treated differently. When a block of registers is involved in a writing transfer it is only weakly locked-out and a program is allowed to use information in the block provided no attempt is made to write over it; e.g. the same block of information may be written simultaneously on two peripheral devices.

4.5 Initiation of a Transfer

Peripheral transfers are initiated by program in much the same way whichever device is concerned. Two instructions, which must be consecutive, are needed: the first to select the device to be used and the mode of use (e.g. reading or writing), and the second to specify the part of the working store concerned by giving the starting address and the number of words (or, if appropriate, characters) to be transferred.

4.6 Descriptions of the Peripheral Devices

Given below are descriptions of the various devices available. Information regarding the maximum numbers which can be fitted, the numbers of simultaneous transfers, and related points are given in Appendix A.

4.6.1 Ampex TM2 Magnetic-tape Units. The Ampex magnetic-tape unit uses tape 1 inch wide and up to 3600 feet long. When the tape is being read from or written to, it moves at 120 inches/second; it is rewound at 240 inches/second. Thus, if the tape does not stop during reading or writing, a full reel can be processed in 6 minutes, while rewinding a full reel takes 3 minutes.

Across the width of the tape are 16 tracks; 12 carry the data, 2 are clock-tracks used for checking tape movement and one track carries only markers indicating the start and end of a block of data. The 16th track is not used.

The packing density is 375 bits per track per inch; thus, within a block, there are 4500 bits of data per inch. Between each pair of blocks is an interblock gap, in which no data are recorded; the interblock gap contains the leading and trailing block-markers, the block serial number and the data check-sum.

The transfer rate is quoted as 90,000 characters/second but, in isolation, this figure can be misleading since:-

(i) it assumes that the data are entirely in character form, whereas much will be in the more compact binary form, tending to increase the effective transfer rate;

(ii) it ignores the effect of the interblock gaps, which reduces the overall transfer rate.

To illustrate the implications of these two opposing effects, suppose that data have been written in 500-word blocks and that, as will usually be the case, some of the Interblock gaps are long and others short (see below). Then the average time to read one block and one interblock gap (i.e. the time between starting to read one block and starting to read the next) will be about 52 millisecs assuming that the tape does not stop between blocks (this is a fair assumption for many applications, since processes are often tape-limited). This leads to an effective transfer rate of about 74,000 characters/sec, or, if all the data are in binary, the equivalent of about 134,000 decimal digits/second.

The blocks can be of any desired length, and the position of each block along the tape is determined solely by the position of the end of the preceding block and the length of the interblock gap, although pre-addressed tapes with a fixed block-length can be used if desired.

As each new block of data is created and written on to the tape it is automatically preceded by the leading block marker and the block serial number; no action by the programmer is necessary for this. When the block has been completely written it is automatically followed by a check-sum and the trailing block marker, again without any action required from the programmer.

The check-sum is a 24-bit sum of the several pairs of 12-bit stripes comprising the data in the block, computed with end-around carry (i.e. carries from the more significant end of the check-sum are added in at the less-significant end). This check-sum is used in the following ways:-

(a) *When writing.* As the data are written on to the tape, the check-sum is computed and retained in the control-unit's writing circuits. When all the data have been recorded, the computed check-sum is written on the tape. A short distance behind the writing head is the reading head, and this reads back the data, from which another check-sum is computed. The reading head also reads back the check-sum recorded on the tape. Thus there are three check sums:-

 (i) that retained in the writing circuits,

 (ii) that computed from the data when read back,

 (iii) that read back from the tape.

 If all three are identical, the writing-transfer is accepted as correct. If not, the tape is automatically wound back to the start of the block, the block erased and a fresh attempt made to write the block a little further along the tape. If this repetition also fails, other attempts are made, at a new position each time, until the transfer is completed successfully, subject to a maximum of 4 attempts.* Moreover, during this process the performance and sensitivity of the reading circuits are deliberately reduced so that, if the quality of the recording is high enough to be read correctly under these conditions, there is little chance of failure during a subsequent reading transfer, when the reading circuits have been returned to full performance.

(b) *When reading.* A block of data is, of course, read with the circuits at full sensitivity. As it is read, a check-sum is computed. The check-sum recorded at the end of the block is also read and the two check-sums compared. If they are identical, the transfer is accepted as correct. If they disagree the tape is rewound to the start of the block and the operation repeated, this being done until the check-sums do agree, subject to a maximum of 4 attempts.*

* This number is merely a parameter in the Orion Monitor Program and so can be changed, easily, if need be.

If, after the maximum number of attempts, a reading or writing transfer has not been carried out successfully, it is abandoned and the program which initiated it is stopped, awaiting engineers' attention.

The interblock gap can have either of two lengths, largely at the programmer' s will. Consider a reading operation in which the tape stops between reading two successive blocks. Then the minimum length of the interblock gap must be such that, after reading the trailing marker of one block, the tape can be brought to rest with sufficient gap left for the tape to be accelerated to full speed before the reading head reaches the leading marker of the next block. This distance is about 0.73 inches, and is the shorter of the two possible lengths. If, when writing these blocks, the tape did not stop between them, this length of gap is adequate.

If, on the other hand, during a writing transfer the tape does stop between writing two blocks, a longer gap is necessary. This is because the gap must allow for the tape to come to rest after the reading head (check-reading) has passed the end of one block and for the tape to accelerate to full speed before the next block starts to be recorded (by the writing head). This requires the length of the interblock gap in such a case to be the 0.73 inches plus the distance between the reading and writing heads, which is 0.39 inches. Thus the longer gap is about 1.12 inches.

The programmer can easily specify which gap he wants, depending on the manner in which the tape will be used subsequently.

If it is intended that some blocks will be left unaltered while others will be overwritten, then the longer gap must be called for. (But such use of magnetic-tape on Orion is not normally recommended.)

If, however, the tape will only be read and not rewritten selectively, the shorter gap may be specified. Then, if the tape does not stop between writing two successive blocks, the shorter gap will be left but, if the tape does stop, the longer gap will automatically result. This is the more usual manner of using a magnetic tape and so normally the programmer can call for the short gap, confident that, if a long gap is necessary, it will be given automatically.

Using the short gap whenever possible results, of course, in increased effective capacity and rate of transfer of data: approximate performance figures for tapes written with all gaps long and all gaps short are given in Tables l, 2, 3 and 4.

The tape can be written to when it is moving forwards. It can be read from when moving either forwards or backwards. This facility is of great value, notably in such processes as sorting since the tape need not be rewound between writing and reading.

As described earlier, transfers are subjected to rigorous and comprehensive checks, to ensure that they are performed correctly. However it is, naturally, essential not only that data are transferred correctly but that they are the correct data; checks of this point are applied through the Orion Monitor Program and the operating system, described later.

The Ampex TM2 has a special feature which eliminates tape threading. The lower spool is fixed on its spindle and has permanently attached to it a metallised leader strip, so that when a spool of tape is fitted on the upper hub, its free outer end can be attached by a buckle to this leader, which is already threaded through the mechanism; after use, the spool is rewound, unbuckled from the leader and removed. Each spool of tape also has a similar metallised leader at its inner end, so that the end of tape travel in either direction can be sensed by suitable electric contacts; these are so arranged as to give also an early warning of end of travel.

TABLE 1 (gap of 1.12 inch)

B	U	Effective transfer rate in thousands of		
		w/sec	ch/sec	e.d.d./sec.
50	32.3	3.63	29.0	52.4
100	48.8	5.49	43.9	79.3
150	58.8	6.62	52.9	95.6
200	65.6	7.38	59.0	107
300	74.1	8.33	66.7	120
400	79.2	8.91	71.3	129
500	82.6	9.30	74.4	134
600	85.1	9.57	76.6	138
800	88.4	9.04	79.6	144
1000	90.5	10.2	81.4	147

TABLE 2 (gap of 0.73 inch)

B	U	Effective transfer rate in thousands of		
		w/sec	ch/sec	e.d.d./sec
50	42.2	4.75	38.0	68.6
100	59.4	6.68	53.4	96.5
150	68.7	7.73	61.8	112
200	74. 5	8.3S	67.1	121
300	81.4	9.1G	73.3	132
400	85.4	9.61	76.9	139
500	88.0	9.90	79.2	143
600	89.8	10. 1	80.8	146
300	92.1	10.4	82.9	150
1000	93.6	10.5	R4.2	152

B	=	number of words in each block
U	=	100 × (total length of blocks) ÷ (total length of tape)
	=	100 × (length of block) ÷ (length of block + length of interblock gap)
	=	percentage tape-utilisation
w	=	words
ch	=	characters
e.d.d.	=	equivalent decimal digits

TABLE 3 (gap of 1.12 inch)

B	Capacity in millions of w	ch	e.d.d.	t (time to read one block, millisecs)	Number of blocks (appx.)
50	1.31	10.5	18.9	13.7	26,200
100	1.98	15.8	28.5	18.2	19,800
150	2.38	19.1	34.4	22.6	15,800
200	2.66	21.3	38.4	27.1	13,300
300	3.00	24.0	43.3	36.0	10,000
400	3.21	25.7	46.4	44.8	8,000
500	3.35	26.8	43.4	53.7	6,700
600	3.45	27.6	49.8	62,6	5,800
800	3.58	28.6	51.7	80.4	4,500
1000	3.67	29.3	53.0	98.2	3,700

TABLE 4 (gap of 0.73 inch)

B	Capacity in millions of w	ch	e.d.d.	t (time to read one block, millisecs)	Number of blocks (appx.)
50	1.71	13.7	24.7	10.5	34,200
100	2.40	19.2	34.7	14.9	24,000
150	2.78	22.3	40.2	19.4	18,600
200	3.02	24.1	43.6	23.8	15,100
300	3.30	26.4	47.7	32.7	11,000
400	3.46	27.7	50.0	41.6	8,700
500	3.56	28.5	51.5	50.5	7,100
600	3.64	29. 1	52.5	59.4	6,100
800	3.73	29.9	53.9	77.1	4,700
1000	3.79	30.3	54.8	94.9	3,800

(N.B. The times given are for one block plus one interblock gap)

4.6.2 ICT 593 Card Reader. This machine reads standard sized cards photoelectrically at a rate of up to 600 a minute, the cards being read column by column. Each column is in fact read twice and the two readings compared; if they do not agree an error is signalled and the card is deflected into the reject rocket.

It is available in two versions:-

(i) to read 80-column cards only,

(ii) to read 80-column and 65-column cards; the two types of card cannot be mixed.

The first version reads cards punched with rectangular holes in the 'normal' positions. The second reads cards with holes of any of the standard shapes (viz. rectangular, circular and ovalised verified) in the normal and interstage positions.

The system is designed to accept cards punched in any manner, disciplined or undisciplined.

As is usual in punched-card work, the rows are considered to be divided into two sets, the upper-curtate (of 4 rows) and the lower-curtate (of 8 rows). If the card is punched in a disciplined manner, with the upper curtate composed of rows according with one of the usual IBM, ICT or Bull conventions, then the punching is regarded as 'coded' and each column can be used to give rise to one 6-bit character in the working store, in a code chosen arbitrarily by the programmer.

If the punching does not accord with one of these three conventions, or if it has interstage punching, then it is read in such a manner that a direct binary image of the card is produced in the working store, this image then being interpreted by program. If some columns of a card are punched in accordance with one of the IBM, ICT and Bull conventions and other columns are not, the card can be read in one of the 'coded plus binary' modes, in which all columns are read in both manners, facilitating interpretation of the mixed punching.

A card-reading transfer is initiated by a pair of instructions as mentioned in section 4.5 above. During each transfer, the data from any number* of columns can be stored. This includes:-

(i) only some of the columns of one card (in such a case, the columns are the first ones of the card),

(ii) all of one card or more than one card,

(iii) all columns of one card or more, together with some columns of the next card.

The whole of every card read passes the reading stations, even if the data from only some of its columns are stored.

If a card is alleged to be 'coded' but carries illegal punching (more than one hole in the lower curtate) then, when it has been read, an error is signalled and, on Orion 1, the card is deflected into the reject-pocket.

4.6.3 ICT 582 Card Punch. Operating at up to 100 cards/minute, the ICT 582 card-punch punches rectangular holes into 80-column cards of standard size. The cards are punched row-by-row and each row is check-read (by brushes) one card-cycle after punching. If the check-reading of any row does not agree with the original data, an error is signalled and the card is offset in the stacker.

---

* This number must actually be a multiple of 4, except when reading all columns of a 65-column card.

Cards may be punched in any desired code; a table included in the program gives the correspondence between the 6-bit internal characters and the punching in the card. In many installations, this code-table will be common to all or most programs and, for each installation, an appropriate standard code-table is included in the Orion Monitor Program and need not be included in the user's own programs.

If the desired punching does not conform to any code (e.g. with split column working), 'binary punching' can be used. For this, a binary image of the desired punching is built up in the working store and the card is punched according to that image.

Thus a card can be punched in any desired manner. There are no restrictions on the number of holes which can be punched in any row or column or in a card. A separate transfer is initiated for each card to be punched.

4.6.4 ICT 665 Line-printer. This machine, which is available with Orion 1 only, prints one line of data at a time. It consists essentially of a print-barrel rotating continuously at 800 r.p.m., with the printing characters cut in relief on the periphery of the barrel. The paper, which may be multi-ply to give up to 3 copies, is struck against the engraved characters by print hammers. The paper may be pre-printed (e.g. as forms) if desired.

The paper is moved by sprocket-toothed tractors, the distance between which is adjustable to allow for paper between 6 inches and 16 inches in overall width. The maximum printing width is 12 inches, occupied by 120 printing positions.

The character set comprises 50 characters. These are normally:-

the letters A to Z,

the numerals 0 to 11 and ½ ,

symbols, such as + - . * ( ) % £ / and 'comma'

but other character sets can be supplied.

The printing format is controlled entirely by program, including the paper movement between successive lines. Paper movement can be specified by program as either:-

(i) a number of line spaces, the number being variable and set by program, or

(ii) movement to a prescribed position (e.g. the top of the next form), known as 'paper throw'.

To print a line, the 6-bit characters in the arbitrary internal code are assembled in the working-store in printing-position sequence and the pair of instructions to initiate a transfer is obeyed. The conversion between the internal code and the printing characters is governed by a code-table. As with the Card-punch, this table will be common to most programs at any one installation (and, indeed, to many installations) and an appropriate table is included in the Orion Monitor Program. A separate transfer must be initiated for each line of print, regardless of the number of characters actually printed in that line.

The ICT 665 printer can be used in either of two ways, as the programmer chooses. If desired, both ways can be used in a program without programming complications.

As stated earlier, the print-barrel rotates continuously. In one mode of use, when a transfer is initiated printing starts with whichever character then happens to be opposite the print-hammers and continues while the barrel makes one complete revolution, giving access to the complete character set. On completion of the actual printing, the paper movement takes place. Assuming that the paper is moved one line-space (1/6 inches) between printing successive lines, printing is done at the rate of 600 lines/minute, with correspondingly lower rates for each additional line-space fed (8 millisecs per line-space). This mode of printing is often called 'asynchronous'.

In the other mode of use, printing starts only when one particular character (usually numeral 0) is opposite the print-hammers and continues while the barrel makes half a revolution, after which the paper is moved. Therefore, provided the paper movement is completed during the other half of the barrel revolution, one line is printed for each revolution, giving an actual printing rate of 800 lines/minute: for this condition to be satisfied, the paper-movement must not exceed 2 line-spaces (1/3 inches). In this mode, often termed 'synchronous', only half of the character set is accessible, those available including the numerals, symbols and the letters A and B. Thus, in practice, numerical data can be printed at 800 lines/minute.

4.6.5 Anelex Type 4-1000 line-printer. In principle, the Anelex printer is similar to the ICT 665; the differences are detailed below.

The character-set comprises 59 characters, the extra nine being various symbols.

The print-barrel rotates at, usually, 1000 rpm, with a switch to select 667 rpm instead.

In the asynchronous mode, with one line-space, the printing rate is 800 lines/minute (533 lines/minute at the reduced barrel speed) with all characters accessible: each additional line-space of paper movement takes about 6.67 millisec.

In the synchronous mode, with not more than 1 line-space between successive lines, the printing rate is 1000 lines/minute (667), 48 characters being accessible, including the numerals 0 to 11, the complete English alphabet and certain symbols.

The paper width is adjustable from 4 inches to 19 inches and an original plus up to 5 copies can be printed.

The paper-throw facility is more versatile than on the ICT 665 printer. There is a continuous loop of 8-channel paper-tape (7-channel on Orion 2) at the side of the printer, its length equal to the length of the pre-printed form, As the printing-paper moves, the loop of tape moves in step with it. When it is required to throw the paper, the number of one of the channels is specified in the program and the paper is moved until the next hole in the selected channel is sensed.

4.6.6 Rank Xeronic Printer. The Rank Xeronic Printer works on the Xerographic principle, which involves the following stages:-

(i) a positive electrostatic charge is applied to a selenium-coated cylinder in a darkened chamber.

(ii) an optical image of the characters to be printed is projected on to the selenium surface, removing the charge from the illuminated areas,

(iii) positively charged particles of powdered ink are sprinkled over the selenium surface, being attracted to the discharged (illuminated) areas but repelled by the other areas,

(iv) the ink particles are transferred, by electrostatic attraction, to a negatively-charged sheet of paper,

(v) the ink is fused into the paper by heating in an oven.

The paper is in a continuous roll from 12 inches to 26 inches wide and moves at a constant speed of 40 ft/min, corresponding to 2880 lines/minute. Normally a line is regarded as being 11½ inches wide, two identical lines being printed simultaneously side by side, each with 128 character positions. There are also facilities for suppressing printing in prescribed positions in each line independently. Alternatively, the paper-width can be divided into four parts, each 5s inches wide with 64 character positions, allowing forms to be printed in quadruplicate.

The paper is not pre-printed. If it is required to print a form outline, a master is used, printing the form simultaneously with the variable data. Registration marks on the master, which are not printed, are used to control positioning of the lines of print relative to the form outline. Other marks can be printed on the paper to control operation of a high-speed paper-cutting machine. The maximum form size is 24 inches wide by 16 inches long.

The character-set comprises 54 alpha-numeric characters, with an optional extension to 118.

A parity error causes a special 'error' mark to be printed down the rest of that form.

4.6.7 I.C.T. TR5 paper-tape reader. Already well known as an input device on earlier computers, the TR5 paper-tape reader has been developed and modified for use on Orion. In particular, it can be used to read 7-track and 5-track tape, a simple switching operation selecting either width. It operates at a maximum speed of 300 characters/second.

The 7-track paper-tape code includes a parity-track (odd parity) which is automatically checked. The 6-bit character which arises in the working-store is a direct binary image of the other six tracks. The code is similar to that adopted by the British Standards Institution.

At least one paper-tape reader must be fitted to each Orion, since paper-tape (in particular, 7-track) is regarded as the principal medium for the primary input of programs to the computer.

4.6.8 I.C.T. TR7 paper-tape reader. Like the TR5, the machine can read 7-track or 5-track paper-tape, but does so at speeds up to 1000 characters/second.

To match the higher reading speed, the paper-tape is spooled on to large reels with a capacity of up to 1200 feet and which can be easily mounted and removed.

Physically, it is much larger than the TR5 and is in the form of a free-standing cabinet.

4.6.9 Creed 25 paper-tape punch. These are two versions of this punch available, one punching 7-track tape and the other 5-track. Its maximum speed is 33 characters/second.

It uses parchment tape in reels of 1000 feet. It is a small machine which can be pedestal-mounted.

4.6.10 Creed 3000 paper-tape punch. The Creed 3000 is a high-speed synchronous punch with built-in tape-spools, photo-electric check-reading and checking electronics. For use with Orion it is available for 7-track and 5-track tape, punching at speeds up to 300 characters/second.

The parity-track is inserted automatically, and the punching is checked by a read-back station 3 character-positions after the punching station.

It uses parchment tape in 1000 feet reels.

Physically, it is in the form of a free-standing cabinet.

4.6.11 Teletype BRPE paper-tape punch. To punch 7-track or 5-track paper-tape on Orion, the Teletype BRPE punch operates at up to 110 characters/second. A modification kit is available to give a speed of 63.3 characters/second.

The tape used is parchment, in reels of 1000 feet. The machine is suitable for pedestal mounting.

4.6.12 Friden Flexowriter. There are several versions of the Friden Flexowriter. The one of current interest, which is used as the control and monitoring organ on Orion, is the model Z.

Essentially it is an electrical typewriter and is used for input to and output from the computer (via the Orion Monitor Program) though it is not used for the input and output of data. Messages typed by the operator are printed in red and are converted directly to electronic signals which are accepted, interpreted and acted upon by the Orion Monitor Program. The Model Z Flexowriter does not have a paper-tape reader.

Messages from the Orion Monitor Program are printed in black; some are also punched into a 7-track paper-tape as log-keeping and time-accounting data. Only the Orion Monitor Program can cause punching of the tape.

It has the standard Orion printing symbols and emits and accepts the standard Orion 6-bit code (see page 77). The character-set can include the extra letters Å, Ä and Ö of the Swedish alphabet.

The control Flexowriter is normally mounted on the control desk, for convenient use by the operator. If desired, additional control desks can be connected, each with its own Flexowriter; there must be at least one control Flexowriter on each Orion.

The carriage length is 12 inches, with 12 characters per inch. The line spacing can be selected manually at 6, 4 or 3 lines/inch.

There is an odd-parity check on the punch.

(Other versions of the Flexowriter are available for preparing data- and program-tapes and printing out results.)

4.6.13 Magnetic Drums. The magnetic drums form the computer's backing-store and each Orion must have at least one drum fitted. The Built-in programs are kept permanently in the drum-store, occupying about half of the first (or only) drum.

While a program is being run on the computer it is usual to keep a copy of it in the drum-store. The drum-store may also, of course, be used as a large-capacity, random access data store.

Normally, an Orion is fitted with more than one drum but this is not apparent to the programmer*; a transfer can start on one drum and finish on another without the programmer being aware of the fact, the switchover from one drum to the next being handled by the drum control unit.

The drum-store is regarded as a peripheral device, in that drum-transfers are autonomous, of variable length, between any parts of the drum-store and the working-store, and are initiated by a pair of instructions in the program, similar but not identical to those which initiate other peripheral transfers.

4.6.13.1 The Orion 1 drum-store. Orion 1 uses Ferranti MD5 magnetic drums, with a revolution time of about 24 millisecs, giving an average access time of about 12 millisecs.

Each drum has a capacity of 16,384 words, arranged as 128 words in each of 128 tracks, each word being 48 bits long with a 49th bit automatically attached for parity checking. Each word is recorded serially, i.e. its bits are in sequence in one track and, during a transfer, one word is read or written in about 188 microsecs.

* Except, of course, by virtue of the increased capacity.

4.6.13.2 The Orion 2 drum-store. Either Ferranti MD5 or Sperry 1C drums may be used on Orion 2. The revolution time is about 20.6 millisecs in either case, giving an average access time of about 10.3 millisecs. Each word is stored in a serial-parallel form as 18 bits in each of 3 tracks; of each group of 18 bits, 16 are data, one is a spacing-bit and one is a parity bit for that group; i.e. each group is checked for parity. The time to transfer each word is about 80 microsecs.

4.7 Time-sharing

4.7.1 General principles. As stated in Section 4.3, peripheral transfers have the three characteristics of autonomy, variable length and directness which allow the optimum speed and ease of use of the peripheral devices to be achieved. However, these features are not enough in themselves. For example, there are many cases in which the time needed to compute and assemble a set of characters for output is very much shorter than the time taken to emit them on a peripheral device, which would seemingly result in the central computer spending much time idle, waiting for one transfer to finish before the next can be initiated. One particular illustration is a program which accepts data from a magnetic tape, converts them into a suitable format and then prints them on a line-printer. On Orion 1, the computation time may be only some 5% to 10% of the printing time (or even less on Orion 2). Moreover such a program will normally occupy only a small part of the computer's stores. There is therefore a large amount of capacity (both in storage space and time) unused. Such waste of time and storage space can be minimised, indeed almost eliminated, by the use of time-sharing.

Suppose that there is a tape-to-line-printer program, as described above, in the store being obeyed, and let it be called Program A. Suppose that there is also another program, Program B, which is, for convenience of argument, mill-limited; that is, it is a program which, in the short time under discussion, is never held up waiting for a peripheral device.

There is also, always, a special program, the Time-sharer Program, in the working-store.

When the two programs A and B were read into the computer, the Orion Monitor Program drew up for each a 'directory-entry', a set of words containing useful information about the corresponding program. The directory-entries are arranged in a priority sequence to form the directory of programs. The two items of current interest in each directory-entry are:-

(i) the requirement; an indication of the reason (e.g. attempt to use a busy peripheral device, or a working-store lockout) why the program is not currently being obeyed, and

(ii) the control number; the address of the instruction at which the execution of the program must next be resumed.

Consider events from the moment when Program A initiates printing of a line, line P say. This transfer continues for about 100 millisecs (ICT 665 printer), during which time Program A continues to compute, by virtue of the autonomy of transfers, preparing line Q. A reasonable time for this is 5 millisecs*, at the end of which time Program A attempts to initiate printing of line Q. However, this is not yet possible, since the printer is still busy with line P. Therefore an 'interruption' is generated, causing Program A to be left and the Time-sharer Program entered.

The first action in time-sharing is to update the directory-entry of the program just left, viz. Program A in this case, by recording the requirement, the control number,

* The times used in this argument are appropriate to an Orion 1. The argument is equally valid qualitatively for an Orion 2.

and other details of no interest at present. (For Program .4, the requirement is, loosely speaking, ‘the ICT 665 printer’ and the control number is the address of the first instruction of the pair which attempted to initiate the transfer.)

This done, the Time-sharer Program scans the program-directory in priority sequence, starting always with the top-priority program, Program A in this case.

Naturally, Program A's requirement cannot be satisfied at present, and so Program B's directory-entry is examined. For reasons which will be given later, B has a ‘null’ requirement, i.e. it can be entered and obeyed at any time. It is therefore entered at the address given by its control number, and its instructions are obeyed simultaneously with the continuing transfer of line P initiated by Program .4. In due course, printing of line P is finished and, for this reason, another interruption is generated, causing Program B to be left and the Time-sharer Program entered once more*, first updating B's directory-entry. Since Program B has been left for some reason external to itself and not because, say, it attempted to use a busy peripheral device, it has a ‘null’ requirement.

The directory is scanned afresh: this time A's requirement can be satisfied and so A is entered to initiate the transfer of line Q. Autonomously, line R is computed, taking the assumed time of 5 millisecs, whereupon A tries to transfer it, is unable to do so and is therefore interrupted again, leading once more to B being entered from the Time-sharer.

All the time these two programs are in the store, with no others present, and so long as B does not become peripheral-limited, the pattern described above will be repeated.

The Time-sharer Program takes about 0.5 millisecs to carry out its actions and so, of each 100 millisecs or so of elapsed time;

about 5 millisecs are spent computing in Program A,
about 94 millisecs are spent computing in Program B,
about 1 millisec is spent in the Time-sharer.

Now Program A is running almost as fast as possible, its speed being limited to the speed of the line-printer (except that there is the half-millisec spent in the Time-sharer between completing one transfer and initiating the next).

Program B is being obeyed about 94% as fast as it would be if time-sharing were not applied, and so the computer is obeying two programs in little more time than is needed for either separately.

Some very important features of time-sharing on Orion are:-

(i) the switching from one program to another is dictated solely by the circumstances then prevailing in the system as a whole, as necessary to keep the complete system working as near full capacity as possible; in particular, the switching is not prescribed in any way,

(ii) each program occupies and uses only the amount of the stores and the peripheral devices which it actually needs, making for the utmost flexibility as to which programs can share time together,

(iii) when writing the program, the programmer need not allow for the fact that it will be sharing time; each program is written as though it is to be the only program in a system of the required size,

(iv) since each current program has the exclusive use of the requisite amounts of working-store and drum-store, no time is taken up, when switching from one

* Note that the Time-sharer Program is always entered at its first instruction, unlike the object-programs whose execution it controls.

program to another, in copying data for the programs to and from special parts of the stores except, of course, for the very few, special items such as the requirement, working-store reservation limits, time used etc.

The illustration above dealt with two object programs. The same principles apply when more than two programs are running concurrently. The number of programs which can run together is in fact limited to 15 but it is expected that, in practice, there will seldom be more than four or five.

The priority-sequence is normally decided by the degree to which the speed of the various programs is limited by the speed of the peripheral devices, the most severely peripheral-limited program having top priority. Now: -

(i) it is very difficult for a programmer or operator to estimate the degree to which a program is peripheral-limited,

(ii) a program may change its degree of peripheral-limitation during execution.

To overcome these difficulties, the Orion Monitor Program itself usually decides the priority sequence, using the actual performances of the various programs for this purpose. Each minute the Orion Monitor Program examines, and if necessary re-orders, the priority sequence, using an item in each directory-entry which gives the time for which each program has used the central computer. (Within the central computer there is a special register, the Program-timer. When a program is entered from the Time-sharer the Program-timer is cleared. Then, while that program is being obeyed, one unit is added in to the Program-timer at regular intervals*. When the program is interrupted and left, the content of the timer is added into a word in the program's directory-entry, thus recording the computer-time used by the program.)

Nevertheless, it is desirable that the users should be able to exercise some control over the priority sequence, and so they can direct that a program shall:-

(i) be assigned its proper position in the priority-sequence automatically by the Monitor Program, or

(ii) be made the top-priority program, or

(iii) be made the bottom-priority program.

However, the option to interfere with the automatic assignment of priorities is exercised only in exceptional circumstances.

Before the Orion prototype was tested, the time-sharing system as originally planned was simulated on another computer and it was found that, in certain circumstances, a time-sharing scheme based solely on a priority-sequence conferred little or no advantage. Such circumstances can arise when a single peripheral control unit, of a type which can handle only one transfer at a time, is shared by two or more programs. It may then happen that one or more of these programs is seldom entered from the Time-sharer and, in some cases, the central computer may be idle for a substantial proportion of the time. This difficulty has been overcome by modifying the Time-sharer Program to introduce a ‘chronological peripheral queue’.

Whenever a program is interrupted for attempting to use a busy peripheral device, it is put at the bottom of the peripheral queue. Then, when the Time-sharer Program is seeking a program to enter, it first scans the peripheral queue; the priority-list is scanned only if no program in the peripheral queue can be entered.

The introduction of the peripheral queue emphasises another important aspect of the Orion time-sharing scheme, namely that it is carried out by a combination of hardware (the

* Of 16 microsecs on Orion 1 and 0.5 millisec on Orion 2.

computer's logic-design) and software (special-purpose program kept permanently in the computer's stores). This means that it has been possible to amend the program in order to accommodate refinements as they have been found to be necessary, while the program can use certain special instructions, provided by the logic, designed specifically for time-sharing. For example, the peripheral queue and the priority-list are scanned very quickly by means of an instruction designed for that one purpose.

Earlier in this section, two reasons for a program being interrupted were given, namely:-

(i) an attempt to use a busy peripheral device (a peripheral lock-out), and

(ii) the end of a peripheral transfer.

There are two other, additional reasons:-

(iii) an attempt to refer to a locked-out register in the working-store, i.e. to write to any locked-out register or to read from a strongly locked-out one, and

(iv) more than one second has elapsed since the last interruption: this reason is included chiefly to detect certain types of program error.

4.7.2 Reservations. Programs often contain mistakes. Even if they are carefully written and thoroughly tested it can never be safely assumed that they are entirely free of blunders. A program may have been successfully used for a long time and still contain a concealed error which may reveal itself only when an unusual set of circumstances arises, such as a rare combination of data, or when a seemingly minor modification to the program is made.

In a computer system using time-sharing such errors can be specially dangerous because the offending program may spoil the operation of others, perhaps in such a way that the error is not immediately detected. An unpleasant example could cause the overwriting of the results of a data-processing program just before they are written on to the new main file. This error would probably be detected only on the next up-dating of the file and the investigation of the reason for the error would be extremely difficult, since it would require study of all the programs sharing the system at the time.

This situation would be quite intolerable.

To prevent such undesirable interference between one program and another, Orion is provided with what are called ‘reservations’. While any particular program is being obeyed, the computer is allowed to use only those parts of the stores and those peripheral devices which are reserved for that program. Should there be an attempted violation of a reservation the program is at once abandoned and the Monitor Program is called in. When the computer switches from one program to another under the control of the Time-sharer, the reservation-settings are changed to those appropriate to the new program; the special instruction needed to do this is treated as an illegal instruction in all ordinary programs, its use causing immediate monitoring action.

By means of the reservations it is made impossible for one program to spoil the operation of another, and so it is quite safe, and, indeed, usual, for a new program to be tested by use of time-sharing while other programs are doing useful work.

The reservations for each program are suitably assigned on input of the program by the Orion Monitor Program, which ensures that they do not overlap those of any other program.

4.7.3 Time-Accounting. As stated earlier, the Orion computer includes an automatic program-timer which has unity added into it periodically while a program is using the central computer and which is used in establishing the optimum priority-sequence. It is also used to assist in log-keeping and time-accounting as described below.

When an interruption occurs the number in the timer is added into a total corresponding to the program just left; this total continuously records minus the computer-time available to that program, and is originally set when the program is read in. Should the program overrun its time this total will become positive, an event which causes monitoring action and which is designed to minimise the waste of computer-time which would occur should a program erroneously get into a loop.

During a peripheral transfer the speed of the computer is slightly reduced by hesitations. To reduce the consequent inaccuracy of time-accounting, the timer is made to hesitate whenever a peripheral control unit uses the working store. Whenever a transfer is initiated the number of hesitations that it will cause is added into the timer. Time spent in the Time-sharer program is charged to the program being entered.

The timer thus makes possible a reasonably accurate record of the amount of computer-time used by each program. Together with the Local Civil Time, which is also available (in one of the pseudo-registers - see Section 2.2 above), it provides the basis of a method of apportioning the costs of running an installation and automatically maintaining a log of the operations, since, whenever a program is started or finished, details of the store-space, peripheral devices, computer time and clock time used are punched into the paper-tape of the control Flexowriter by the Monitor Program; this tape can subsequently be analysed by a special program to produce a log of the computer' s operation and information for costing purposes.

5. THE BUILT-IN PROGRAMS

Five hundred and twelve of the working-store registers and about half of the first drum are used to store certain programs essential to the proper running of the computer system, and which are kept permanently in the stores. All programs can avail themselves of the facilities provided by the Built-in Programs.

There are two Built-in Programs:-

(i) the Basic Input Routine, and

(ii) the Orion Monitor Program, which includes the Time-sharer program.

In addition, in Orion 2, 1500 registers are used for the extracodes, that is, built-in programs to produce the effects of some of the more complex instructions which, on Orion 1, are performed directly by the logic design.

5.1 The Basic Input Routine

The Basic Input Routine is a routine designed to read programs in Basic Input Language and convert them to the stored, binary form ready for execution.

Basic Input Language is a character-form program-assembly language described briefly in Section 7.1. It is not normal for programs to be written initially in Basic Input Language, which is intended primarily as the object-language into which programs written in other languages (Symbolic Input, Nebula, Extended Mercury Autocode and Fortran) are compiled.

When reading a Basic Input Language program, the Routine uses as working-space the working-store and drum-store regions in which the object-program will work, subject to a minimum of 944 registers and 64 drum-locations; if a program requires fewer than 944 registers, the excess can be released when the Basic Input Routine has finished reading the program.

5.2 The Orion Monitor Program

Already there have been occasional references to the Orion Monitor Program (OMP) from which something of its purpose and effects will have been gathered.

Although it is strictly one program it can, for convenience, be considered in three parts: -

(i) an organisational part,

(ii) a monitoring part,

(iii) the Time-sharer Program.

5.2.1 The Organisational Part. This section of OMP is concerned with accepting programs into the system, allocating store space and peripheral devices, setting up the program directory-entries, adjusting the priority sequence, time accounting, and functions of this type.

It is called in

(a) whenever a program is presented to the computer,

(b) every minute on the minute.

Its effects are described in more detail in Section 8.

5.2.2 The Monitoring Part. In essence, the monitoring part of OMP is intended to detect, interpret and inform about failures; failures of the central computer, of a peripheral device, and of the program. In this connection, the term 'failure' is interpreted very broadly, as will become apparent later.

Suppose that a parity-failure occurs in some working-store register. Then this is detected by hardware when that register is next used and a signal is generated. The Orion Monitor Program receives the signal, stops execution of all programs and prints a message on the control Flexowriter informing the operator of the failure.

If a peripheral device fails in any way (the term 'failure' including such comparatively minor events as reaching the end of a magnetic tape or some other peripheral medium, e.g. the tape in a paper-tape punch), a signal is generated, leading to OMP interrogating the device and deciding into which of a number of categories the failure falls. It then disengages the device, so that it is temporarily unusable by the object-program, and prints a suitable message naming the device and the type of failure. There are alternative courses of subsequent action.

If the programmer has not made allowance in his program for that type of failure then;-

(i) the program is suspended, i.e. stopped in such a manner that it cannot be continued from the point it has reached, or

(ii) the failure is dealt with automatically by OMP, e.g. repeating an unsuccessful magnetic-tape transfer, or

(iii) OMP allows the program to continue, possibly after some action by the operator, e.g., if the paper runs out in a paper-tape punch, the program is allowed to continue normally except that no transfers can take place to that punch until the operator has reloaded and re-engaged it.

However, the programmer may include in his program special restart routines to be obeyed in the event of peripheral failures. If he has done so for the type of failure encountered then, after disengaging the device and printing the message, the Monitor Program returns control to the object program at the beginning of the restart routine. This routine may give any restart action, in conjunction with the operators, which the programmer wishes.

Concerning program failures, these are divided into two classes. The first class includes those failures so serious that the program cannot be allowed to continue. These are:-

(i) attempting to violate reservations in any way,

(ii) attempting to obey an illegal instruction, i.e. an instruction with an unassigned function-number or, say, of 2-address type when only the 3-address form is meaningful,

(iii) attempting to use an operand which is not in a permissible form for the operation, including attempted division by zero,

(iv) attempting to initiate a writing peripheral transfer when the overflow indicator is set. This last reason is included because setting of the overflow indicator suggests strongly that an incorrect result has been formed and so the data to be transferred are suspect. The onus is on the programmer to clear the overflow indicator (by program) before initiating the transfer, if he has considered and allowed for overflow occurring in his program.

In all four cases above, the Orion Monitor Program will suspend the offending program, printing a suitable message.

The other class comprises less serious failures, including certain types deliberately introduced into the program by the programmer to assist in detecting and diagnosing programming errors. For all events in this class, the programmer can choose which of several possible actions is taken by the Orion Monitor program. Such monitoring actions are termed 'optional monitoring' and the programmer has a choice of several 'styles' in each case.

(i) Monitoring on signal-instructions. The first (most significant) bit in the stored form of an instruction is termed the 'signal bit' . Normally it is a 1 but if the programmer writes S immediately after the instruction's function-number, the signal bit is made a 0 and the instruction becomes a signal-instruction. When a signal instruction comes to be obeyed, monitoring action may occur, depending on the style chosen by the programmer. The available styles are:-

Style 0: no monitoring occurs and the instruction is obeyed as though it were not a signal; the program is not slowed down and, in fact, the Monitor Program is not entered.

Style l: the program is stopped in such a manner that it can continue from the point then reached. A message to this effect, and including the signal instruction itself, is printed out. If RUN is typed by the operator on the Flexowriter the signal instruction is obeyed normally, the program continuing from there.

Style 2: the address of the signal instruction, its function number, its result, and the address to which the result is written are printed out. The signal instruction is then obeyed normally (without operator action) and the program continues.

Style 7 the Monitor Program returns control to the object program at an address specified by the programmer. This can be the start of a routine to carry out any action desired by the programmer, including return to obey the signal instruction normally.

The programmer can set, as signals, instructions at strategic points in the program, thereby obtaining information helpful during development of the program. Once the program has been developed, the signal-instructions can be changed to non-signals, or style 0 can be selected.

If the programmer does not specify a style, style 1 is set by default.

(ii) Monitoring on jump-instructions. If the programmer chooses, he can be provided with information about the instructions obeyed in his program which cause 'jumps' as follows.

Style 0: no monitoring; the program is not slowed down.

Style 1: every time a jump occurs, the address and function number of the jump-instruction are printed out, together with the address of the jump' s destination.

Style 2: every time a jump occurs, the address and function number of the instruction, together with the destination address are stored (not printed). Then, if a program failure occurs (including one of the optional events which is being monitored), the information relating to the last 16 jumps is printed.

Style 3: information similar to that appropriate to Style 1 is printed, but only for the special instructions used to enter routines.

Style 7: as in Style 7 monitoring on signals, a special program routine is entered.

(iii) Time monitoring. Sometimes a fault in a program causes it to take an unduly long time to be obeyed. This often happens because a loop of instructions which is to be obeyed repeatedly, perhaps for a prescribed number of times, has been programmed incorrectly with the result that the condition which decides when the loop should be left is never satisfied.

Since Orion includes the program-tinier mentioned previously, the programmer can easily avoid the waste of machine time which usually follows from such programming errors, by stating an amount of time in the central computer which he wishes his program, or a section of it, to be allowed. If that time is actually used up, monitoring action occurs.

One way in which the programmer can request some time is by means of a special instruction called a 'timing flag'. Suppose he is writing a loop of instructions. Then he can estimate the mill-time needed to obey the loop the requisite number of times. Then, before the first instruction of the loop, he includes a special instruction, a 'timing flag', requesting that amount of time (in units of 1 second) plus, say, 10%. After the end of the loop he puts another timing-flag, requesting sufficient time for the next section of the program. If the loop is programmed correctly it will be left before the time allowed expires, the second timing-flag will be obeyed and the program will continue normally. However, if the program is incorrect, so that the loop is not left in time, 'timer overflow' occurs and monitoring action is taken according to which of the following styles has been chosen.

Style 0: the program is stopped and a message is printed. If the operator types RUN on the Flexowriter, the program is given one more minute of mill-time and is allowed to continue.

Style 7: the program is given another minute and the programmer's own routine is entered. In this, the programmer can cause any desired action including, of course, requesting still more time.

In the foregoing, three of the optional types of program monitoring have been described in detail. There are several other program events on which monitoring action is optional, on the same general principles as described above. These other events include overflow (fixed- and floating-point), drum-transfers, and peripheral transfers. In addition, the Monitor Program allows the programmer to choose whether floating-point arithmetic is automatically rounded or unrounded.

There is another way in which the programmer can call in the Monitor Program, namely by means of the so-called 'special 150' instructions, which are instructions with function-number* 150. Now these instructions are illegal in ordinary object-programs and so, when one is encountered during program execution, it is regarded as a program failure, causing the object-program to be interrupted and the Monitor Program entered. However, provided the 150-instruction is of 3-address type and the Z-address** has one of certain prescribed values, the Monitor Program will carry out action appropriate to that value of Z and, usually, allow the object-program to continue thereafter.

The first of the special 150's is the timing-flag already mentioned. It is written, typically, as

```
        150     10      0       1
```

which causes the program to be given 10 seconds of mill-time.

Given below is a small selection of the permissible values of Z and the associated actions.

Z = 10:	the program is stopped in one of several possible modes according to what is written as the Y-address†.
Z = 11:	the program is abolished; that is, it is no longer considered by the Time-sharer Program, and appropriate terminating action is taken on all peripheral devices used by the program. The program's store and peripheral reservations are then released for use by any other program which needs them.
Z = 12:	this instruction causes the date and time to be placed into two of the program's registers.
Z = 13:	this allows the programmer to cause any desired message to be printed on the Flexowriter or, if he chooses, some other peripheral device.
Z = 14:	using such an instruction, the programmer can cause any desired question to be printed on the Flexowriter, and an answer typed by the operator to be received by his program.

* See Section 6

** In the instruction 150 10 0 1 , the 1 is the Z-address, see Section 6.

† The Y-address is the second address in a written instruction, e.g. in the instruction 150 0 2 10 the Y-address is 2.

Other instructions of this type allow the programmer to:-

> reserve and relinquish peripheral devices,
>
> obtain the address of the current block on magnetic tape,
>
> write a block with a non-sequential number, to mark the end of the data on a magnetic tape,
>
> change the size of the program' s drum-store reservation,

and achieve many other special effects.

6. THE INSTRUCTION REPERTORY

The object of this section is to give a very brief survey of the repertory of instructions and other facilities available to the Orion programmer: fuller discussions of the programming languages themselves are given in Section 7.

Basically, Orion instructions are of 3-address type, i.e. an instruction specifies the addresses of three working-store registers, two of which contain the operands while the third is destined to receive the result of the operation. Formally, a 3-address instruction is written as

$$F \qquad X \qquad Y \qquad Z$$

where F is the function-number, specifying the operation to be performed,

> X and Y are the main-addresses, i.e. (usually) the addresses from which the operands are taken,
>
> Z is the address to which the result is sent.

Each register has a unique address, called its absolute- or machine-address, and these could be written in instructions. However, since each program normally occupies only part of the store, sharing it with other programs, to write absolute-addresses in instructions would place severe restrictions on which programs could share the store. Accordingly, whenever a program is read into the computer it is automatically assigned a working-store datum-point, which can, of course, differ from one occasion to another, and all addresses are given relative to this datum-point. They are written as the letter A followed by an integer, i.e. as A0, A1, A2 ..... up to the largest required by the program.

Thus a typical instruction might be written as

```
00      A100      A200      A7
```

Since the function-number is 00, the effect of this instruction is to put into register A7 the sum of the 48-bit signed numbers from registers A100 and A200; the contents of these last two registers are unchanged by the instruction.

If general addresses are denoted by X, Y and Z, and their contents by x. y and z respectively, then the effect of a 00-instruction can be expressed as

$$z' = x + y$$

where a prime denotes the quantity after the instruction has been obeyed.

Of the addresses, X and Y can be those of any registers in the program's reserved region of the working-store, but the Z-address is restricted to lie in the range A0 to A63 inclusive; these first 64 registers available to the program are called its accumulators. Of the accumulators, A0 always contains zero and cannot be used as the destination of the result of an instruction.

Instructions can also be written in a 2-address form, typified by

```
00          A100          A200
```

which forms the sum of the numbers in registers A100 and A200, writing that sum into A100, replacing that operand. In the notation defined earlier, the effect is

$$x' = x + y$$

Other arithmetical and logical operations with 48-bit operands are performed by instructions with function-numbers 01 to 07.

In an instruction such as

```
10          A100          27          A7
```

one of the operands is the signed 48-bit number in register A100, while the other is the number 27, i.e. the Y-address itself, which is stored as the least significant 15 bits in the instruction and is treated as an unsigned quantity. Thus this particular instruction will add the 15-bit Y-address to the 48-bit number, writing the result into accumulator A7, i.e.

$$z' = x + Y$$

The 2-address form, e.g.

```
10          A100          27
```

has the effect

$$x' = x + Y$$

Other instructions in which the stored Y-address is used as a 15-bit unsigned operand are those with function -numbers 11 to 17.

In instructions with function-numbers 20 to 27, the Y-address is written as a number which in fact specifies one of the pseudo-registers (see Section 2.2). Thus

```
20          A100          12          A7
```

causes the number ½, stored in pseudo-register 12, to be added to the 48-bit signed number from register A100 and the result stored in accumulator A7, i.e. in general terms

$$z' = x + pY$$

where pY denotes the content of the pseudo-register numbered Y. Instructions in this group, too, can be of 2-address type.

Orion includes a wide range of facilities for automatic multiplication and division. In most multiplication instructions, the product of x and y is formed, but the various function - numbers produce different forms of product as below: -

function-number 32	full, signed, double-length product, stored in Z and Z + 1
function -number 30	signed, single-length product of integers, i.e. in effect the less-significant half of the double-length product

function-number 31	signed, rounded, single-length product of fractions, i.e. in effect, the more-significant half of the double-length product, but correctly rounded
function-number 33	accumulated product, for forming, e.g., $ab + cd + ef + \ldots$

Thus the instruction

```
32      A100      A200      A7
```

forms the full, double-length product of the 48-bit signed numbers in registers A100 and A200, writing the m.s. half into A7 and the l.s. half into A8.

In 2-address form, e.g.

```
32      A100      A200
```

the same product is formed, but it is stored in A100 and A101.

Function-number 34 allows for multiplication by a constant, since one of the multipliers is the stored Y-address itself, e.g., in

```
34      A100       27       A7
```

the content of A100 is multiplied by 27, producing a signed, single-length product.

The functions concerned with division cater for single- and double-length dividends and single-length divisors, giving various forms for the result, viz: -

quotient and remainder,

rounded, single-length quotient (fractional and integral),

double-length mid-point quotient.

The shift-instructions provided allow for: -

single- and double-length, arithmetical and logical, left and right (or up and down),

cyclic shifts, by a number of digit-positions or by a number of character-positions (6 bits per character),

a logical shift left or right which is automatically terminated when the end-most 1 bit is shifted off, the number of places of shift necessary then being written into an accumulator.

In most cases, the number of places to be shifted is given as the Y-address of the instruction, for example

```
52      A100       3        A7
```

is an instruction to shift the content of A100 logically left by 3 places, writing the result into A7. Two-address forms are provided.

There are, of course, jump-instructions. One group of these jumps or does not jump depending on the result of a comparison between two signed, 48-bit numbers, e.g. an instruction of the form

```
65      X       Y       Z
```

causes a jump to address X if y $\nless$ z. ($\nless$ denotes 'is not less than'.) In 2-address form, the comparison is between y and zero.

Another set of jump-instructions, those of function-group 7, make the comparison between the actual Y-address stored in the instruction (an unsigned 15-bit quantity), and the less-significant half (unsigned 24-bit quantity) in an accumulator. (The l.s. half of a word is usually termed the modifier half and is denoted by a suffix m.)

Thus the instruction

```
70      X       15      A7
```

causes a jump to X if the value of the l.s. 24 bits in A7 is equal to 15.

The common operation of counting the number of times a loop of instructions is obeyed is performed by instructions with function-numbers 80 to 83, summarised as:-

80:- add 1 to a counter and jump if l.s. half = Y

81:- add 1 to a counter and jump if l.s. half ≠ Y

82:- subtract 1 from a counter and jump if l.s. half = Y

83:- subtract 1 from a counter and jump if l.s. half ≠ Y

e.g.

```
81      X       15      A7
```

adds 1 to the number (signed, 48-bits) in A7, causing a jump to X if the l.s. half of the result does not have the value 15.

Since, in Orion, a quantity is often used both as a counter and as a modifier, this is an appropriate point at which to discuss address-modification and related topics. There are three facilities for the manipulation of addresses, namely:-

(i) internal modification,

(ii) address replacement,

(iii) external modification (or pre-modification).

If an instruction is written with three addresses and with X and/or Y immediately after the function-number, as in

```
00X     A100    A200    A7
```

then the instruction is of the internally modified, 2-address type.

When the particular instruction written above comes to be obeyed, then the modifier half of the content of accumulator A7 is added to the address A100 to produce the effective X-address from which one operand is taken: the X-address is so modified because X is written after the function-number. In this case, the Y-address is not modified and so is unchanged at A200. Note that the augmentation of the X-address is done in the control circuits while the instruction is obeyed - the X-address stored in the instruction remains unaltered at A100.

If a main-address is enclosed in parentheses, as in

```
00      (A100)      A200      A7
```

then that address is to be 'replaced'. In this particular example, one of the operands is the number in A200, as usual. The other operand is selected in the following way:- the modifier-half from register A100 is read and used as the address of another register, the word in which is used as the second operand. Either or both of the X- and Y-addresses can be replaced.

Addresses can be replaced and modified, as in

```
00XY    A100      (A200)     A7
```

in which the X-address is modified, while the Y-address is both modified and replaced. In such cases, the replacement is done first, the result of which is then modified.

The pre-modification facility is used by writing a pair of instructions such as

```
117      20         30
00      A100       A200      A7
```

When these are obeyed, the X- and Y-addresses in the first of the pair are added to the corresponding addresses in the second, so that the pair above is effectively equivalent to

```
00      A120       A230      A7
```

In fact, one would normally write this one instruction; the pair quoted above would not normally be written in a program, and were given above purely for the purposes of illustration. A more realistic example is

```
117     (A15)      (A16)
00      A100       A200      A7
```

in which the addresses A100 and A200 are augmented by the modifier halves from A15 and A16 respectively. Note that:-

(i) the effective instruction can be of 3-address type,

(ii) the X- and Y-addresses can be augmented by different, independent modifiers,

(iii) the modifiers need not be held in accumulators.

The main-addresses in the premodifying instruction can themselves be modified, while the substantive instruction can be of 2-address modified type, e.g.

```
117X    (A15)      (A16)      A17
00XY    A100       A200       A29
```

The main-addresses in the substantive instruction can be replaced; the effects depend on whether the function-number of the pre-modifying instruction is 116 or 117. If it is 116, the replacement is done after the pre-modification, while if it is 117, the replacement is done before the pre-modification.

Consider first the pair

```
117     (A15)       0
00      (A100)      A200      A7
```

and suppose that the modifier half in A15 has the value 20 while the modifier half in A100 is equivalent to the address A400. Then:-

(i) the A100 is, in the control circuits, replaced by A400,

(ii) the modifier from A15 is added to A400, giving A420 as the effective .X-address.

For the pair

```
116     (A15)       0
00      (A100)      A200      A7
```

(i) the modifier from A15 is added to the address A100. giving A120,

(ii) the modifier half from A120 is used as the effective X-address.

In fact, one is not restricted to a single pre-modifying instruction; there can be several, each operating on its successor until the substantive instruction is ultimately-reached.

These facilities of replacement and internal and external modification allow complex address-manipulations to be performed during program execution.

Peripheral transfers are initiated by pairs of instructions. A typical pair is

```
140.21    0        *MT2
142       A600      450
```

The element *MT2 is the name by which a particular magnetic-tape deck is referred to in the program. The number 21 after the function-number 140 is the 'mode' of the transfer; in this case it specifies that a block of data is to be written to the tape. The data will comprise the 450 words from registers A600 to A1049 inclusive.

Similar, but not identical, pairs of instructions are used to initiate:-

(i) drum-transfers,

(ii) internal transfers, viz. copying several words from one part of the working-store to another.

Floating-point arithmetical operations are performed by instructions with function-numbers 90 to 95, providing for addition, negation, subtraction, multiplication and division. Another function permits the closeness of agreement of two floating-point numbers to be determined, while others allow for conversion between fixed-point and floating-point forms.

Data enter and leave the system as characters (in paper-tape or punched cards or on a line-printer) while the internal arithmetical operations are on binary numbers. Special functions are provided for conversions between these forms, the various radices used in the conversions being specified as program-constants, eight radices to a word. These functions include facilities for checking the characters for validity and for ignoring characters during conversion after input, and for various special treatments of non-significant zeros when converting to character-form before output.

The many other functions available perform more or less complicated operations of great value in data-processing, but it is not possible to discuss them meaningfully here. Accordingly, the reader wishing details of them is referred to List CS 374, the Orion Programming Manual.

A summary of all the functions available is given as Appendix B, pages 66 to 76.

7. PROGRAMMING LANGUAGES FOR ORION

There are five main programming languages for Orion, namely: -

Basic Input Language,

Symbolic Input Language,

Nebula,

Extended Mercury Autocode, and

Fortran.

Of these, the first two are machine-oriented assembly languages while the three others are problem-oriented automatic programming schemes. Each is described in its own separate document and the descriptions which follow are intended to give only a general picture of the nature anti facilities of each language.

7.1 Basic Input Language

Basic Input Language is described in List CS 381 'Programming for Orion in Basic Input Language'. It is a character-form language, converted to the stored, internal form directly by the built-in Basic Input Routine.

In Basic Input Language, instructions can be written exactly as shown in Section 6,

e.g.

```
00      A100      A200      A7
```

Addresses of the form A100 are termed Basic Addresses; however, Basic Input Language employs addresses written not only as Basic Addresses but also those written as Identifiers, of which there are two types: -

(i) those whose first character is L,

(ii) those whose first character is V.

An Identifier whose first character is L can be written formally as L*n* or L*p.q* (e.g. L123 or L1.59) which are exactly equivalent if

$$n = 64p + q$$

and these two forms can be written interchangeably. In effect an Identifier of this form represents a special way of writing an address.

The other form of Identifier is written as V followed by an integer, e.g. V10. When such an Identifier is given a setting (see below) its value is set as the content of the correspondingly numbered register. V1 and V2 represent the current drum-store and core-store transfer addresses respectively.

An Identifier can be set either by an equation, e.g.

```
         LI.7    =   A1234
    or   L1234   =   557
    or   V10     =   94
    or   L2.4    =   L1.7-V10+15
```

or by using it as a label, as in

```
L1.19)) L1)    00    L2.6    A54    L1.1-4
```

Here L1.19, being followed by two right parentheses, is a drum-label and is set during program input to the drum-store address at which the labelled instruction is stored, while L1, being followed by one right parenthesis, is a working-store label and is set to the address of the core-store register from which the instruction will be obeyed.

In Basic Input language, instructions can be written in the format illustrated above, where each of the three address-fields can be written as the sum/difference combination of any number of Identifiers of both types, Basic addresses and integers.

Program constants can be written as: -

(i) integers, e.g. +1234 or -567,

(ii) fractions, e.g. +0.0714F or -.169F,

(iii) packed quantities, e.g. 0,1,17,21,,, where each comma causes the value formed thus far to be shifted up 6 places (multiplied by 64) and the element following that comma to be added in.

Each program constant is stored in one register. A program constant can be labelled in the same way as an instruction. This means that, at the time an instruction using a program constant is written, it is not necessary to know the address at which the constant is stored since its label, an Identifier, can be written in the instruction as its address. Moreover, the programmer need never know the constant's address (as a Basic address) since the Basic Input Routine automatically relates the two uses of the Identifier.

Since V1 and V2 represent the drum- and working-store transfer addresses respectively, the programmer has control over the storage of his program. Initially he sets these two variables to suitable values, by equations such as

```
V1  =  2
V2  =  A100
```

As each word of the program is read and stored, the values of VI and V2 are each increased by 1 automatically, but the programmer can reset either or both at any time. This provides the facility of dividing the program into chapters, a chapter being, essentially, that part of the program which is held in the working-store at any one time; the length of (number of registers occupied by) a chapter is decided by the amount of working-store which the programmer chooses to allocate to program, which can, of course, vary from one chapter to another, Thus if the programmer writes the program which he wishes to constitute one chapter and follows it by an equation resetting the working-store transfer address, e.g.

```
V2  =  A104
```

he is, in effect, starting a new chapter. The succeeding lines of program will be stored in consecutive drum locations, following directly after the preceding chapter. However, when, by program, the new chapter is brought into the working-store, it will be arranged to occupy registers A104 et seq., probably overwriting part of the chapter already in the working store.

Basic Input Language includes a number of directives, lines which are not stored as part of the program but which serve to control input, storage and execution of the program. The equations setting Identifiers represent one form of directive.

Another directive is ENTER, which controls the initial entry to the program when it starts being obeyed. At the end of the program is written ENTER followed by a small integer, e.g.

ENTER 0

When this is read during program-input it causes the Basic Input Routine to check that the program is in a fit state to be obeyed (e.g., all the peripheral devices needed have been reserved). Then the Basic Input Routine refers to the words at drum-addresses 0 and 1 (generally, 2n and 2n + 1, where n is the integer in the ENTER directive). There the programmer will have placed two words giving the address of the instruction at which the program is to be entered, and data about the chapter containing that instruction (the chapter's length, its position in the drum-store and the position it is to occupy in the working-store). That chapter is then automatically transferred into the working-store and entered at the desired instruction.

A program can have several entry-points. Normally one of these is the primary entry-point, the others being used for re-entering the program subsequent to some failure.

7.2 Symbolic Input Language

Another character-form, computer-oriented language, Symbolic Input, is described in "The Orion Programming Manual" List CS 374.

In many respects it can be regarded as an extension of Basic Input Language. In particular, the Basic Input facility of using Identifiers such as L2.7 to represent addresses and quantities is extended in Symbolic Input Language to allow for Symbolic Addresses.

A Symbolic Address is a set of characters, chosen arbitrarily by the programmer according to rules which are not very restrictive. These rules permit Symbolic Addresses such as GROSS, PENCE, CODENUMBER, X, Y, X1, X2, ALPHA24. Note that a Symbolic Address corresponds broadly to a Basic Input Identifier whose first character is L. There are no identifiers in Symbolic Input Language corresponding directly to the V Identifiers of Basic Input, except that a Symbolic Address of the form *V*n* means the Basic Identifier V*n*.

A Symbolic Address can be set by using it as a label (drum-label or core-store label) or by an equation, the forms being completely analogous to those of Basic Input Language.

Any address in any instruction can be written as a Basic Address (e.g. A100), a Symbolic Address or, in appropriate instructions, an integer. Indeed, a written address can be any sum-difference combination of Symbolic .Addresses, Basic Addresses and integers. Note that the function-numbers of instructions are written numerically.

Program constants can be written in many forms, of which six are regarded as 'normal' by the Symbolic Input Routine.

These are:-

(i) integers, e.g. +100 and -27,

(ii) signed fractions, e.g. +0.12726, including the special case -1.0,

(iii) character words, e.g.

7,6,19,30,24,3,15,0

in which each written integer is stored in the corresponding 6-bit field in a register, eight to a register,

(iv) quarter-words, e.g.

1234, 567, 4000, 3

in which each written integer is stored in the corresponding 12-bit field in a register, four to a register,

(v) half-words, e.g.

16000000, 1234567

in which each written quantity is stored in the corresponding 24-bit field in a register. The quantities written in this case can be Symbolic or Basic Addresses,

(vi) other 48-bit quantities, e.g. -SYMB and +A354 each of which is stored in one register: each of these particular examples will be stored as, say, the absolute address or number represented by the quantity written.

In addition to the six normal formats, other formats are recognised but must be introduced by a 'special format directive' .

Examples are: -

(i) Mask; if two successive lines on the program sheet are

```
MASK
1-16,   0-10,   1-6,   0-16
```

a word will be created and stored, containing sixteen 1-bits followed by ten 0-bits followed by six 1-bits followed by sixteen 0-bits. (The total number of bits set must, of course, be 48)

(ii) Octal; if the following lines are written

```
OCTAL
135726
.135726
```

two words will be created and stored. Each digit is regarded as an octal digit and is stored as a 3-bit quantity. Up to 16 octal digits may be written in each line. If fewer than 16 are written, those unset are stored as three 0-bits. Since the first actual number above does not contain an octal point, the six digits (18 bits) are stored right-justified. The point in the second number causes the digits to be stored left-justified.

(iii) Packed numbers; in this case the directive itself is followed on the same line by a set of integers, totalling 48, which specify the lengths of the several fields into which a word is to be divided. On the next line are written the quantities to be stored in those fields. A typical example is

```
PACKEDNUMBERS          5,   12,   19,   3,   9
27,   3192,   1000000,   6,   473
```

Each special format directive can be followed by any number of lines in that format.

Other special format directives permit the input of Sterling quantities, double-length numbers and floating-point numbers. Note that in all the illustrations above, the directives were written in full but in fact only the first three letters are significant, thus the PACKEDNUMBERS directive can be written as merely PAC.

A Symbolic Input program can be divided into chapters. To do this, the programmer introduces each chapter by the directive CHAPTER followed by any Symbolic Address, which is the chapter's name, e.g.

CHAPTER ONE

introduces the chapter named ONE.

The chapter directive may be followed by a TRANSFER directive, typically

TRANSFER A103

setting the working-store transfer address for the chapter; the transfer address can be written as a Basic Address or a Symbolic Address. If no TRANSFER directive is given, the transfer-address is automatically set to A64.

Entry to a program occurs when an ENTER directive is read; this is essentially similar to the ENTER directive in Basic Input Language, but the entry-point itself is indicated more simply by means of the START directive. This is written on a line of the program sheet as, typically,

START 0

and serves to mark the instruction on the next line as the correspondingly numbered entry-point of the program.

When an ENTER directive, e.g.

ENTER 0

is read, the chapter containing the similarly numbered entry-point is copied from the drum-store into the working-store and entered at the desired instruction.

It is often convenient to write a program as a number of logically distinct, communicating routines, linked together by a master-program. Symbolic Input language includes special facilities for this. To mark that a section of the program is to be regarded as a routine, the programmer precedes it by a line such as

ROUTINE SUB/

introducing a routine whose name is SUB/, and follows the routine by the END directive as in

END SUB/

Since the routines in a program may be written by different programmers, and for other reasons, it is possible that the same Symbolic Address may be used in different routines, with different meanings local to the routines. To avoid confusion in such cases the Symbolic Input Routine automatically prefaces the Symbolic Address with the name of the routine. Thus a Symbolic Address ADDR set (by label or equation) within routine SUB/ is effectively regarded as SUB/ADDR, and this ‘full name’ can be written in instructions. In fact, to refer to that address from outside the routine SUB/, the full name SUB/ADDR must be used. However, for simplicity, the same address can be referred to from within SUB/ as merely ADDR.

A special case of a routine is a Library routine (i.e. a routine of such wide applicability that it is included in the system's Library tape); to distinguish a Library-routine from a programmer' s own routine, the first character of its name is a colon, so that :SIN/ is the name of the Library routine to evaluate sines and cosines. Symbolic Language includes features to facilitate incorporation of Library Routines into Symbolic programs.

If some operation requiring several instructions is to be performed at several different points in one program, it is usually made into a routine, and only one copy of these instructions is included in each of the appropriate chapters. Cases do arise, however, in which the process requires only a few (say 3 or 4) instructions and does not merit being made into a routine. Nevertheless, it is inconvenient to write and punch this set of instructions each time it is to appear in the program. For such cases, the 'macro-instruction' facility is provided. Early in the program the macro-instruction is defined, being introduced by the directive MACRO followed on the same line by a fictitious function-number in the range 2000 to 2999, chosen arbitrarily by the programmer. The line is followed by the instructions etc. constituting the macro-instruction. The definition is terminated by the directive PROGRAM. The addresses in the constituent instructions can be written in the forms already described for Symbolic Input language, and also in a special form termed a 'variable address' and typified by =3. Thus a definition of a macro-instruction might be

```
MACRO    2010
           00      A100     ADDR      =1
           00       =1       =2
           32       =1      SYMB            A7
PROGRAM
```

At a point in the program where the process performed by this macro-instruction is required, the programmer might write

```
           2010    A15      QTY
```

Then, during program input, the three instructions

```
           00      A100     ADDR      A15
           00       A15      QTY
           32       A15     SYMB      AT
```

are stored according to the current values of the transfer addresses; the effect is exactly as though the programmer had written these three instructions in full. Note that the variable addresses =1 and =2 have been replaced respectively by the first and second addresses written after the macro's function-number. A macro-instruction can include as many variable addresses as desired, the only limit being set by the line-length of the Flexowriter on which the program-tape is prepared. Naturally, the variable addresses can be different at each appearance of the macro-instruction on the program sheet.

Two facilities in Symbolic Input language are connected with the allocation of the working-store between program, data and working-space. Thus suppose that a program has been given 2480 working-store registers, that it is required to keep the 64 accumulators A0 to A63 free, to store the program (strictly, one chapter of it) in A64 onwards, and then to arrange the input, output and working regions beyond the end of the chapter. To know the address of the first register beyond the program requires, of course, knowledge of the number of registers occupied by the chapter. In principle, this can be counted by the programmer but:-

(a) it is a slow, tedious job liable to mistakes,

(b) some lines of the program sheet occupy several registers (e.g. macro-instructions),

(c) some lines of the program sheet are not stored (directives).

These difficulties can be avoided, the counting being done by the Symbolic Input Routine itself, since &ONE, for example, is a Symbolic Address whose value is automatically set equal to the number of registers occupied by chapter ONE.

Therefore if the transfer address for chapter ONE is A64, then A64+&ONE is the address of the first register beyond the end of the chapter and can be used as such in instructions. Normally, for convenience, the program would include an equation such as

INP = A64 + &ONE

and INP used as the address.

The problem is more complicated when a program has several chapters of different lengths and, perhaps, different transfer addresses. Suppose that there are three chapters, AA, BB and CC, their transfer-addresses being, say, A64, A100 and A203 respectively. Then

>(A64 + &AA, A100 + &BB, A203 + &CC)

is a Symbolic Address whose value is equal to the largest of the values of the written quantities; it is therefore equal to the address of the first register (other than an accumulator) which will never be occupied by any of the three chapters. Again, it is usual to have an equation such as

INP = >(A64 + &AA, A10Q + &BB, A203 + &CC)

and to use INP as the address of that register.

An important feature of these special Symbolic Addresses, which they share with the ordinary Symbolic Addresses, is that they can be written in any instruction or quantity anywhere in the program, before or after they have been set.

7.3 Extended Mercury Autocode

The Mercury Autocode scheme is already well established as a convenient and efficient language in which to write programs concerned with scientific and technical calculations, and it has been extended in scope for use on Orion, the new version being called Extended Mercury Autocode (EMA). The same language is available for the Atlas computer, programs in this language being equally acceptable to both types of computer. The earlier Mercury Autocode scheme is compatible with the new, extended version.

Essentially, the language allows for two types of number;

(i) the indices, stored and manipulated as integers,

(ii) the variables, stored and manipulated as floating-point numbers.

Operations using these types of number are specified by instructions written in a conventional algebraic notation.

The set of indices is associated with the ‘index identifiers’

i j k l m n o p q r s t

i' j' k' 1' m' n' o' p' q' r' s' t'

Each index identifier can have one value associated with it at a time, these values being integers in the range -2^{47} to $2^{47}-1$.

There are three groups of variables. The 'auxiliary variables' are used chiefly for matrix manipulation and are described later. The other two groups are the 'special variables' and the 'main variables'.

The special variables are associated with the variable identifiers

$$a\ b\ c\ d\ e\ f\ g\ h\ u\ v\ w\ x\ y\ z\ \pi$$

$$a'\ b'\ c'\ d'\ e'\ f'\ g'\ h'\ u'\ v'\ w'\ x'\ y'\ z'\ \pi'$$

Each of these variable identifiers can have associated with it one value at a time, these being in the range -10^{38} to $+10^{38}$ approximately and held to a precision of about 12 significant decimal digits.

The family of main variables is split into one or more sets, each set being identified by one of the 'set identifiers'

$$a\ b\ c\ d\ e\ f\ g\ h\ u\ v\ w\ x\ y\ z\ \pi$$

the number of variables in each set being specified by the programmer. The individual members of a set are referred to by following the variable identifier by a suffix, which may be an integer, an index identifier or a combination of these in an arithmetical expression. The programmer declares the desired sizes of the sets by directives such as

$$a \rightarrow 134$$

$$b \rightarrow 10$$

$$x \rightarrow 257$$

directing that there be three sets of variables, namely

$$a_0, a_1, a_2, \ldots\ a_{134} \quad \text{(135 in all)}$$

$$b_0, b_1, b_2, \ldots\ b_{10} \quad \text{(11 in all)}$$

$$x_0, x_1, x_2, \ldots\ x_{257}, \quad \text{(258 in all)}$$

The total number of variables available is determined largely by the amount of working-store the programmer reserves for his program. They may be divided into sets in any way. The values of the main variables are held in the same range and to the same precision as for the special variables.

The indices can be used to denote particular members of sets of variables. Thus if, at some moment, the indices s and t have the values 50 and 70 respectively then a_{70}, $a_{(t-s+50)}$ and a_t all denote the 71st member of the set of a's (provided that the set has at least 71 members).

The instructions of a program include arithmetical expressions and assign a new value to some variable or index. Thus a possible instruction is

$$y = a_s + b_t + c_{20}$$

causing the sum of the current values of a_s, b_t and c_{20} to be computed and set as the new value of the special variable y. The variable on the left-hand side can be a main variable, as in

$$x_k = 0.05a_{(s+10)} + a_{(s-t)}/(2\pi)$$

This last instruction in fact, includes numerical values, directly written as such*. These can be included freely in instructions; possible formats include

15

15.0

015

-1.41421

0.666667

the first three of which are equivalent. Numerical values can also be written as decimal, floating-point numbers, e.g.

103.725&+2

which is treated as 10372.5, i.e. the signed integer after the ampersand (&) represents a power of 10 by which the fixed-point part is to be multiplied.

Similar expressions can be written assigning a new value to an index, for example,

$$t = (s+r-123)(s+q)$$

and $$p = p+1$$

the latter of which causes the value of p to be increased by 1.

A particular form of expression is typified by

$$a_i = -10.73$$

and $$k = 52$$

assigning directly a definite numerical value to a variable or index.

The autocode scheme includes several standard functions, the simplest uses of which are typified by

$y = \phi\text{sqrt}(x)$

$y = \phi\text{sin}(x)$

$y = \phi\text{log}(x)$ (natural logarithm)

$y = \phi\text{intpt}(x)$ (integral part)

$y = \phi\text{frpt}(x)$ (fractional part)

These functions can be used with general arithmetical expressions as arguments, e.g.

$$z = \phi\text{sqrt}(xx+yy)$$

and $$x_n = \phi\text{cos}(nd)$$

although the first of these is provided directly by a function with two arguments, viz.

$z = \phi\text{radius}(x,y)$ $(\sqrt{}(x^2+y^2))$

* The special variable π has the value 3.14159. . . . unless and until altered by the program.

Most of these functions may themselves he included in arithmetical expressions, e.g.

$$x' = [-b+\phi sqrt(bb-4ac)] / (2a)$$

$$y3 = \phi sqrt(\phi log(x+3i)) + 9x'$$

Another function evaluates a polynomial. Written typically as

$$y = \phi poly(x)\ a_m,n$$

it evaluates the polynomial

$$a_m + a_{(m+1)}x + a_{(m+2)}x^2 + \ .\ .\ .\ + a_{(m+n)}x^n$$

setting the result as the new value of y.

Input and output of data are easily programmed. The basic input instruction is, typically

```
read (x)
or   read (k)
```

each of which causes the next number to be read from the source of data, and the specified variable or index set to that value. Numbers to be read can be punched in the formats described earlier, or in a floating-point format as a signed argument and a signed integral exponent (power of 10), separated by a comma or an ampersand (&). Facilities are also available for reading several numbers into the computer by a single instruction.

To output numerical data, the instruction

```
print  (x)  m,  n  (typically)
```

is used, causing the current value of the variable x to be printed, with up to m digits (at least one digit) before the decimal point, and n digits after the point. (If n = 0, the decimal point is not printed.) The numbers of digits can be set explicitly, as in

print (x_p) 2, 5

or they can be written as arithmetical expressions as in

```
print (a')  p+1, q - p
```

Moreover the quantity printed need not be simply a variable, it can also be an arithmetical expression, e.g.

print $(zz + w_i y_i)$ 3, n

To allow data to be output in tabular form, the instructions

```
        space
and     newline
```

are provided. If more than one space or newline is required, instructions such as

```
           space 4
           newline 6
or even    new line (i + 6m)
```

are used.

Intelligibility of output data can often be improved by incorporating headings. To this end, the instruction

```
           print  ('TEMPERATURES  -  CASE 5')
```

causes

```
           TEMPERATURES - CASE 5
```

to be printed.

Every programming language must include control instructions (e.g. jump instructions) to control the path taken through the program. In EMA any instruction can be labelled, a label comprising an integer followed by a right parenthesis, and written to the left of the instruction, as in

```
           1)      n = n+1
```

Then an instruction such as

```
            jump 1
```

causes a jump to the correspondingly labelled instruction.

Most jumps are actually conditional, i.e. the jump does or does not occur according as sane condition is or is not satisfied. Thus the instruction

```
           jump 4,   x ≥ (y + z) / a
```

causes a jump to label 4) if $x \geq (y + z) / a$, otherwise the instruction on the next line on the program sheet is obeyed.

The available conditions are $=, \neq, >, \geq, <, \leq$.

The instruction

```
           jumpdown 3
```

causes a transfer of control to the instruction labelled 3. Instructions are then obeyed normally until an instruction

```
           return
```

is reached. This causes a jump back to the instruction immediately after the jumpdown instruction. These two instructions provide a convenient method of entering and leaving subroutines.

There is no objection to several jumpdowns taking place before the first return instruction is reached. Each return in fact relates to the last jumpdown not yet cancelled by a return instruction.

Another facility is the automatic loop designed for those cases in which it is desired to repeat a loop of instructions while an independent variable takes successively each of several different values. If the loop of instructions is preceded by a line such as

```
        i = 2(1)5
```

and followed by

```
        repeat
```

then the loop is obeyed four times, with i having the successive values 2, 3, 4 and 5. The index i can be used as an operand within the loop. It may be reset within the loop but, obviously, the programmer must take care in so doing. Loops may be nested, i.e. loops within loops are allowed, to a total depth of 24.

A program in this scheme can be divided into chapters. This is done by writing

```
        CHAPTER 3
```

(where 3 is the number of the chapter) at the start of the chapter and

```
         Close
```

at its end. All programs have a chapter 0 and initial entry is to the first instruction of this chapter. Extra chapters 0 may be included. The instruction

```
        rmp
```

meaning 'read more program' causes the current chapter 0 to be replaced by the next one in sequence; this is then entered at its first instruction.

Special facilities are included for dealing with double-length (double-precision) numbers. A double-length number is written on the program sheet as, for example,

```
        ((x,y))
```

where variable x is the more-significant half and variable y the less-significant half. Double-length numbers can be included in arithmetical, read, and print instructions in much the same way as single-length numbers, e.g.

$$\text{read } ((a_0,\ a_1))$$

$$((x,y)) = ((x,y)) + ((a_0, a_1)) / ((c, c'))$$

$$\text{print } ((x,y))\ 5,\ 10$$

Complex numbers, too, can be manipulated directly in arithmetical instructions. A complex number is written as

```
        (u, v)
```

where u is the real part and v the imaginary part, i.e.

$$(u, v) \text{ denotes } u + iv = u+v\sqrt{(-1)}$$

Complex numbers may be used as the arguments of certain functions. A typical instruction involving complex numbers is

$$(a, b) = (f_0 + c_i, g_0 - d_i) - (3, 2)\phi\exp(x,y)$$

Matrices and vectors figure prominently in much scientific and technical calculation, and this has been taken into account in the Extended Mercury Autocode. The set of auxiliary variables is provided chiefly to hold the matrices.

Since the auxiliary variables are never referred to except in special matrix instructions, they are identified simply by integers or by arithmetic expressions with integral values. Normally, auxiliary variables numbered 0 to 10751 are available but the programmer may specify a larger or smaller set if desired.

A matrix is stored row by row. If the first element of a matrix A with m rows and n columns is at auxiliary variable 100, then the element $a_{i,j}$ is at 100 + im + j.

The matrix operations are performed by special instructions; for example, if a, b and c have the values 101, 801 and 1661, the instruction

$$a = \phi 11(b, c, u)$$

forms the sum of two m x n matrices (where m × n = u) whose starting elements are at 801 and 1661 and places the resulting matrix in the auxiliary variables with its first element at 101. Each matrix can consist of any number of elements (subject to the limits imposed by store size), including a single element.

The functions permit addition and subtraction, with or without multiplication of one matrix by a scalar, multiplication, division, transposition, inversion, evaluation of the determinant etc. In some cases the second operand matrix can be the unit matrix or the diagonal matrix stored as a vector. Matrices can also be read as data, and printed.

There are also special functions and Library programs for many other processes (generation of random numbers, quadrature, solution of differential equations, harmonic analysis, correlation, etc.).

A full description is given in 'Extended Mercury Autocode for Atlas and Orion', Ferranti List CS 350A.

7.4 Fortran

Fortran - a name derived from FORmula TRANslation - is an automatic programming system for Orion of considerable power and flexibility, providing a simpler means of writing many programs, both large and small, than using the basic machine language. It is particularly suited for problems of a mathematical or scientific nature. Actual programming is carried out by writing a series of statements in the FORTRAN source language, having a general resemblance to algebraic formulae and commands in English; these statements have to be translated by a special program - the compiler - into sets of machine instructions to carry out the required computation.

7.4.1 The Fortran Source Language (Forsol). Simple variables are created in FORTRAN merely by choosing, according to a few simple rules, suitable names termed 'identifiers'. All references to variables are by their identifiers, the compiler assigning, and compiling machine instructions to refer to, the actual storage addresses. Arrays of variables having any number of dimensions may be similarly named, individual elements being referred to by appending subscripts to the array names as in conventional mathematical notation. The compiler assembles the instructions required to compute storage addresses from the subscripts. Arithmetic operations may be carried out upon variables by writing down formulae of arbitrary complexity: in particular the symbols + - * / are used to denote addition, subtraction, multiplication and division respectively, and an extensive range of mathematical functions having standard names is available.

As an example, a program may refer to variables named A, B and C whose values are the lengths of the sides of a plane triangle. In order to compute at some point in the program the area of the triangle and to assign this value to a variable named, say, DELTA it is sufficient to write the statements

```
S   =    (A+B+C)  / 2

DELTA  =  SQRTF(S *(S-A) * (S-B) * (S-C))
```

where SQRTF is the name for the square root function and the working variable S has been created for convenience. The compiler will construct machine instructions to carry out the necessary sequence of arithmetical operations.

The reader must refer to the FORTRAN manual (List CS 390) for full details of the language: it is possible here to mention only a few of the facilities provided.

Array sizes and the types (modes) of variables may be declared. Variable names may represent, for instance, integers, floating-point numbers, logical variables (i.e. 48-bit digit patterns to be manipulated by Boolean operations) or text variables for the storage of strings of characters. A typical declaration is

```
INTEGER Q (5, 10), W
```

stating to the compiler that Q. is the name of a two-dimensional array of integral elements, of size 5 by 10, and that W is the name of a single integral variable.

The flow of computation is directed by control statements; integer labels are attached to statements so that control statements may refer to them. Thus the statement

```
GO TO 7
```

calls for control to be transferred to the statement labelled 7. Conditional statements allowing branching dependent on the computed value of some expression, and a statement form controlling the repetition of a loop are also available.

Statements exist to handle the transfer of blocks of numbers to and from the magnetic drums, and for the input and output of data via any peripheral device with automatic conversion from decimal to binary and vice versa. Input/output statements are powerful and flexible. The principle adopted is that of an input/output list of variables to be transmitted and an associated list of format specifications specifying the forms of conversion required, the two lists being scanned in conjunction with one another. The specifications may call for conversion of numbers represented as integers, floating-point numbers or fixed-point numbers on the external medium, of logical quantities represented as octal integers or of text quantities as character strings. Other information may be included in the format specifications; for printed results, for instance, this may specify the spacing of values across the page, vertical line spacing and incorporation of titles and headings, thus allowing full control over the appearance of the output.

7.4.2 An Example of a FORTRAN Program. A simple example of the tabulation of an elementary 'cost curve' will serve to illustrate the general appearance of a FORTRAN program; a representation of this is punched on paper tape or cards for input to the compiler.

It is required to tabulate values of n and $C_n = 100n^{p'}$ as two parallel columns, with suitable headings for integral values of n running from 1 to 50. The value of p is to be determined from a value C_2 (< 100.0) read in at the start which satisfies

$$C_2 = 100 \times 2^p$$

After completing the tabulation a new value of C_2 is to be read in and the process repeated, continuing until a zero value is encountered indicating the end of the data.

A possible program is as follows:-

```
10 READ 1, C2
 1 FORMAT (F5.2)
   IF (C2) 2, 3, 2
 2 P = LOGF(C2/100)/LOGF(2)
   PRINT 5, C2
 5 FORMAT   (5H1CASE,   F8.2//4X 1HN5X1HC/)
   DO 9  N    =     1,   50
   CN    =     100  *  N  *   *   P
   PRINT 9, N, CN
 9 FORMAT (15, F8.2)
   GO TO 10
3  CALL  EXIT
   END
```

Considering the statements briefly, the first (labelled 10) calls for a value of C_2 (named here C2 to satisfy FORTRAN rules) to be read according to the specification in format statement labelled 1. (This format indicates that the value is to be punched on the external medium as a five-character number, with two digits after the decimal point.) The IF-statement tests the value of C2 and illustrates the general form of this statement; control is transferred to the statement labelled 2, 3 or 2 according as C is negative, zero or positive. A zero value therefore transfers control to the statement CALL EXIT, which is used as the dynamic end of the program. (The statement END marks the physical end of the program-tape to the compiler.) Otherwise statement 2 is obeyed to compute the value of p, and then a print statement executed, in conjunction with format statement 5, to print the value of C for reference together with some text in the form of headings. The DO-statement illustrates a simple form of loop control; it calls for execution of the group of statements up to the statement labelled 9 with N taking successively the values 1, 2, 3, - - , 50 . This loop computes C_n (named here CN) and prints one value of n and the corresponding value of C_n on each line. Finally control is returned to statement 10 to begin a new case.

Without considering the details of the FORMAT statements, the first few lines of a typical output would be as follows

```
CASE     80.00


         N        C
         1    100.00
         2     80.00
         3     70.21
         4     64.00
```

7.4.3 Routine Structure. Large programs may need to be written in chapters, each of a size capable of being held in the core-store. Smaller programs will consist of a single chapter only. A very powerful feature of FORTRAN is the facility for writing an individual chapter as a set of routines: one routine, the main routine, is unnamed and other routines are given distinct names. Routines are called by the main routine, and may call each other, by name; values of variables are communicated between routines by the substitution of actual arguments for dummy arguments at each call or by placing selected variables in an area of storage designated COMMON where they may be referred to by more than one routine. The names of variables are private and cannot clash with similar names in other routines; each routine is a self-contained set of statements and is compiled as such without reference to other routines.

This 'disjoint routine structure' has a number of important advantages. Routines may be written by different programmers, agreement being required only on the means of communication, and routines written in connection with one program may be simply incorporated into another. They may be compiled and tested independently, reducing to a minimum effects of errors in programming and the amount of recompilation necessary to correct errors - programmers are encouraged to make corrections in the source language and not to tamper with compiled programs. Routines may be quite small, making it easier to reflect the logical structure of the overall flow chart: indeed a set of short routines frequently requires less total compiling time than one large routine.

7.4.4 The FORTRAN Master Program (FORMAP). The FORTRAN system on Orion consists of more than just a compiler, which is in fact a subroutine of the FORTRAN master program. An individual routine written in the source language is compiled into an intermediate form of machine language - the assembler language FALAN - which must contain a certain amount of symbolic information, such as the names of other routines called by this routine. In order to assemble a set of routines into a complete chapter, a FORTRAN assembler (FAS) is required; this also is a subroutine of FORMAP. The assembler is responsible for removing the symbolic content from individual routines, including filling in cross-references, assigning storage and collecting required standard functions from the Fortran library: the output from the assembler is a complete chapter in Basic Input language.

The FORTRAN system is thus very flexible. Whenever a FORTRAN job is put on to Orion, the Monitor Program enters FORMAP, which controls subsequent operations by recognising directives defining the job. The functions of FORMAP include calling in the compiler and the assembler as required to deal with individual routines - which may be in FORTRAN, or previously compiled and hence in FALAN - and calling in the Basic Input Routine to load and enter the object program if execution is desired. If execution is not desired, then FORMAP will abolish itself after the required compilation and/or assembly has been carried out.

7.5 Nebula

Nebula is a very advanced automatic programming scheme designed for commercial applications of computers; its name is abbreviated from Natural Electronic Business Language.

A Nebula program consists of three main parts:-

(i) the Machine Description

(ii) the Data Description

and (iii) the Procedure Description.

The Data Description can in turn be divided broadly into:-

(a) the Physical Description, and

(b) the Logical Description.

A Nebula program cannot be obeyed directly; it must first be compiled into Basic Input Language. This compilation is normally done once only for each Nebula program. Thus a single run of a Nebula program may involve two computers; the machine on which the program is compiled, and that on which the compiled (object-language) program is subsequently obeyed*. The purpose of the Machine Description is to give details of these two machines, in such respects as store sizes, peripheral devices etc., so that: -

(i) the compiling run can be done as efficiently as possible,

(ii) the object program will be the most efficient practicable.

The object program, when being obeyed, will require and emit data, these being defined in the Data Description.

In the Procedure Description, the various operations to be performed on the data are specified; it is this part which most closely corresponds to a 'program' in the more traditional sense, being composed of explicit instructions.

The various items of data are referred to in the Procedure Description by names chosen arbitrarily by the programmer; in the Data Description these names are introduced, their inter-relationships defined, and other information about them is given, so that the compiler can correlate the several uses of the names of the items.

The Data Description is prepared on two forms, one for the Logical and one for the Physical descriptions. These forms are reproduced, to a reduced size, as the next page. It will be seen that each is divided into two parts by a vertical double line. To the left of that double line, the two forms are identical in format and content, and it is here that the data names and relationships are defined.

There are four 'levels' of data name, as follows: -

(i) the file name,

(ii) the record name,

(iii) the group name,

(iv) the detail name,

A 'file' is the stream of information which enters or leaves by one input or output channel. For example, the set of punched cards carrying information about Insurance proposals, claims, premiums paid etc. can constitute an Amendments File.

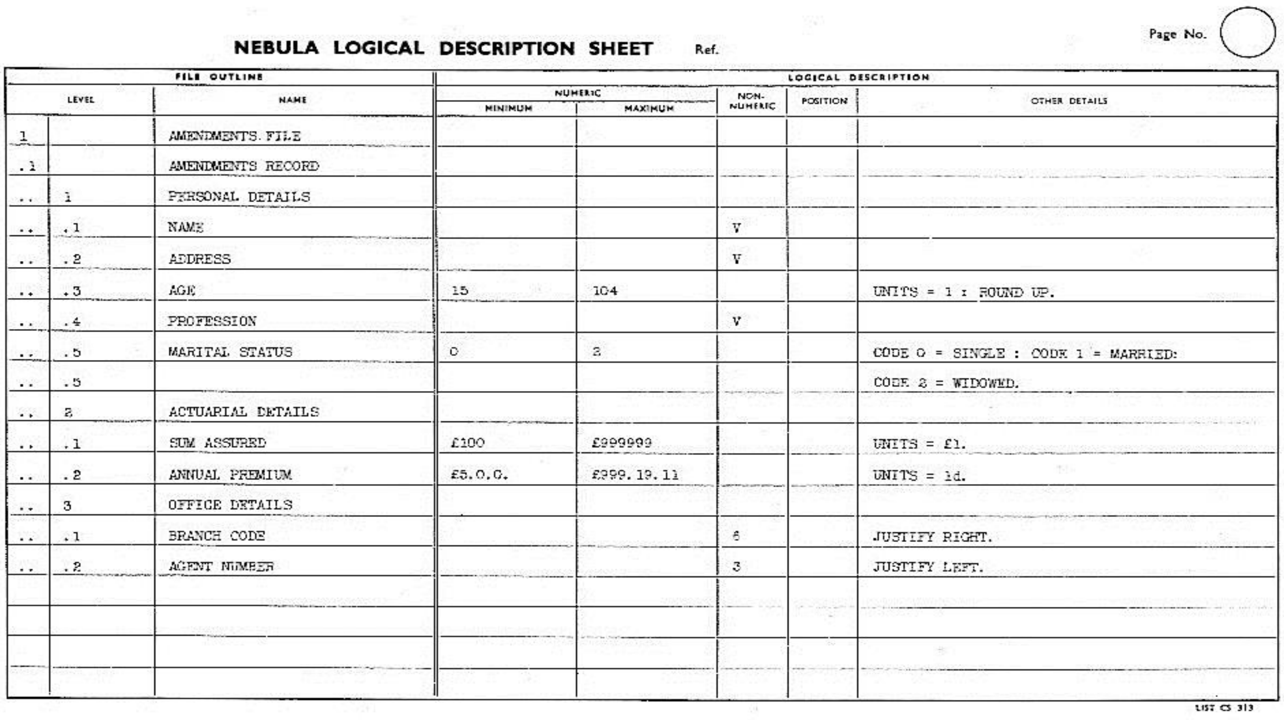
A record is the broadest logical subdivision of a file. The amendments concerning one particular insurance policy could be called an Amendments Record.

Each record can normally be regarded as being composed of a number of groups, logically distinct from each other, each group containing several individual items. Each item, i.e. some data which cannot be logically subdivided, is termed a detail, and each logical collection of details is termed a group. Thus a group name might be Personal Details, some of its component details being Name. Address, Sex, Age and Marital Status.

The right-hand portion of each form gives more information regarding the data represented by the data-names. On the Logical Description form, this information concerns the more abstract characteristics of the data. Thus the detail 'Name' is described as being non-numerical, with a variable number of characters. The compiler would use that information to put into the object-program instructions to store the name as a set of characters (i.e. not to convert the name to a binary number), and to pack the variable number of characters in some economical way.

---

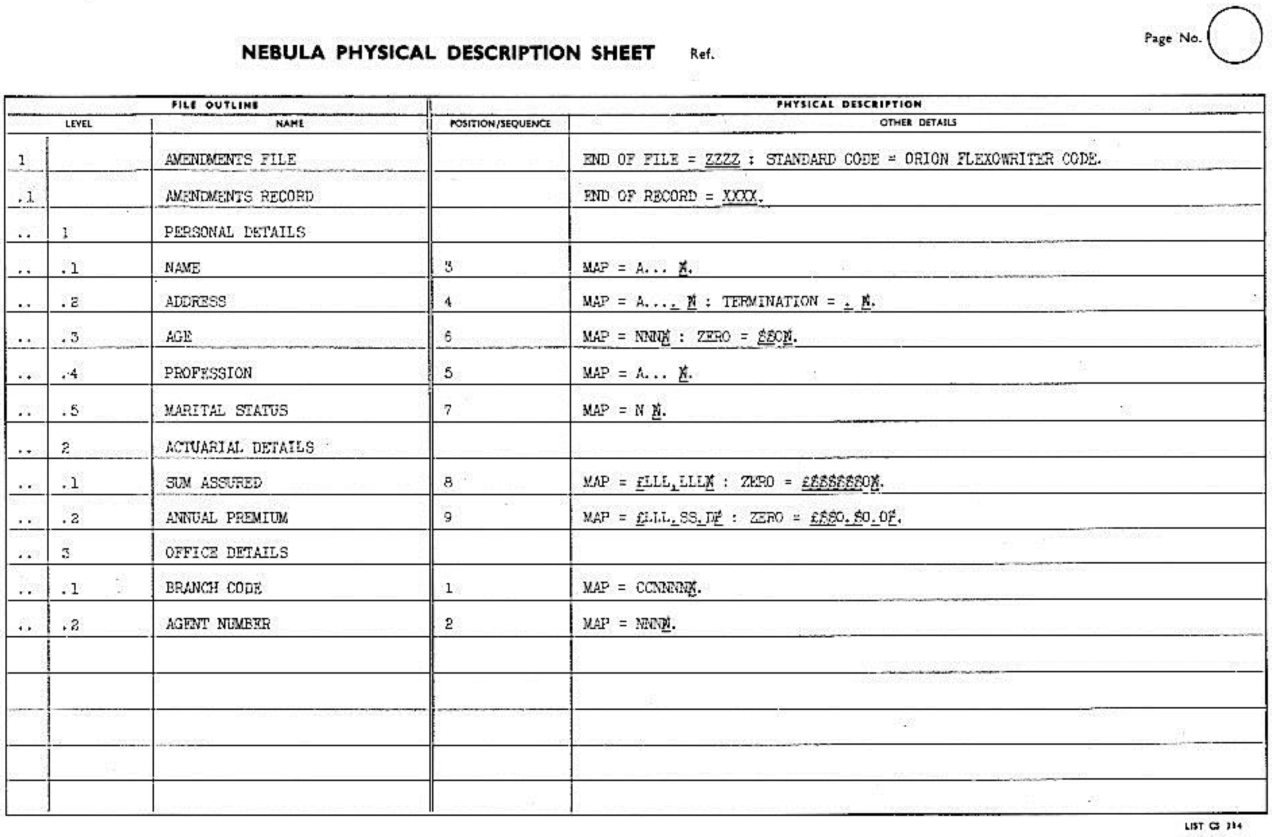
* They may, of course, be the same computer.



NEBULA LOGICAL DESCRIPTION SHEET Ref. Page No.

FILE OUTLINE			LOGICAL DESCRIPTION				
LEVEL		NAME	NUMERIC MINIMUM	NUMERIC MAXIMUM	NON-NUMERIC	POSITION	OTHER DETAILS
1		AMENDMENTS FILE					
.1		AMENDMENTS RECORD					
..	1	PERSONAL DETAILS					
..	.1	NAME			V		
..	.2	ADDRESS			V		
..	.3	AGE	15	104			UNITS = 1 : ROUND UP.
..	.4	PROFESSION			V		
..	.5	MARITAL STATUS	0	2			CODE 0 = SINGLE : CODE 1 = MARRIED:
..	.5						CODE 2 = WIDOWED.
..	2	ACTUARIAL DETAILS					
..	.1	SUM ASSURED	£100	£999999			UNITS = £1.
..	.2	ANNUAL PREMIUM	£5.0.0.	£999.19.11			UNITS = 1d.
..	3	OFFICE DETAILS					
..	.1	BRANCH CODE			6		JUSTIFY RIGHT.
..	.2	AGENT NUMBER			3		JUSTIFY LEFT.

LIST CS 313



NEBULA PHYSICAL DESCRIPTION SHEET Ref. Page No.

FILE OUTLINE			PHYSICAL DESCRIPTION	
LEVEL		NAME	POSITION/SEQUENCE	OTHER DETAILS
1		AMENDMENTS FILE		END OF FILE = ZZZZ : STANDARD CODE = ORION FLEXOWRITER CODE.
.1		AMENDMENTS RECORD		END OF RECORD = XXXX.
..	1	PERSONAL DETAILS		
..	.1	NAME	3	MAP = A... N.
..	.2	ADDRESS	4	MAP = A...._ N : TERMINATION = . N.
..	.3	AGE	6	MAP = NNN : ZERO = ££0N.
..	.4	PROFESSION	5	MAP = A... N.
..	.5	MARITAL STATUS	7	MAP = N N.
..	2	ACTUARIAL DETAILS		
..	.1	SUM ASSURED	8	MAP = £LLL,LLLN : ZERO = ££££££££0N.
..	.2	ANNUAL PREMIUM	9	MAP = £LLL.SS.DF : ZERO = £££0.£0.0F.
..	3	OFFICE DETAILS		
..	.1	BRANCH CODE	1	MAP = CCNNNNN.
..	.2	AGENT NUMBER	2	MAP = NNNN.

LIST CS 314

Similarly the detail 'Age' is described as numerical, with a minimum value of 15 and a maximum value of 104 in units of 1. The compiler would produce instructions to convert the age to a 7-bit binary integer and to pack it into some convenient 7-bit field.

The Physical Description form carries descriptions of the actual physical form in which the data will be presented to (or output by) the object-program.

Thus, in the present illustration, the Amendments Pile is described as being punched in the standard Orion Flexowriter code (on paper-tape by implication). The numbers in the Position/sequence column show the sequence in which the various details actually appear on the tape for each record. For each detail a MAP statement is given, showing the formats in which they will be punched. Thus the statement

MAP - C. . . <u>NL</u>

against the detail-name PROFESSION informs the compiler that a profession will be punched as a series of letters (of unknown number) terminated by a 'newline' character. Similarly the detail ANNUAL PREMIUM will be punched as £ followed by 3 digits giving the pounds, a full stop, two digits giving the shillings, another full stop and one digit (0 to 11) giving the pence. In this case there is also a ZERO statement, giving information of what will be punched if the premium is small; e.g. if the premium is less than £1, then 'spaces' will be punched for the hundreds and tens of pounds but 0 will be punched for the units of pounds, while the tens-of-shillings will be punched as either 'space' or 1. Note that the presence of a ZERO statement does not necessarily imply that a zero premium will ever actually be encountered.

At the end of each record, four X's will be punched as a marker, while four Z's will mark the end of the whole Amendments File.

A sample record according with the given data description is:-

```
NE0179
007
R. P. O'Neill.
'Delcote' ,
The Ridgeway,
Warlhan Magna,
Bedfordshire.
Teacher
24
1
£ 3,500
£114.  7.  10
XXXX
```

From the combination of the Logical and Physical Descriptions, the compiler inserts into the object-program instructions to acquire, convert and store the details.

Having thus defined and described all the data referred to in the program, the Procedure Description expresses, in a quasi-English language, the operations to be performed upon them. A possible Nebula sentence is:-

If age in amendments record > 45 and profession in amendments record = "Steeplejack" then add 50% * premium into premium.

Some of the words and symbols in this sentence, namely **If, in, >, and, then, add, *, %, into** and 'full stop' have fixed meanings in the language. Others, namely, **age, amendments record, profession** and **premium** are amongst those defined in the Data Description.

The remaining elements in the sentence, namely **45, "Steeplejack"** and **50** are 'literals' a term intended to suggest that they are stored literally as program constants. In fact the literal **"steeplejack"** will be stored as the eleven characters which represent the letters s,t,e,e,p,l,e,j,a,c,k and a detail profession read as part of an amendment will be compared with these characters; if identify is registered, then the condition

profession = "steeplejack"

is satisfied. Any element (word, number or symbol) enclosed within quotation marks or underlined is a literal. A number is always a literal; it is stored as the equivalent binary number. Thus 45 is stored as the binary integer 101101, while 50% is stored as the binary fraction equal to 0.5.

Since the compiler has been supplied with full descriptions of the data, and produces the instructions needed to read or emit them, the programmer is not required to write detailed instructions for input and output. Thus the simple statement

read Amendments Record

which may be all or part of a Nebula sentence, has the effect of making available to the program all the details in the next amendment record and, to the programmer, the problems of data input are reduced to this degree of simplicity. Output of results is correspondingly simple. For output, the programmer gives information regarding, say, printing format in the Physical Description. Thus if, against a detail name, the entry

23/→25

is written, the detail will be printed on line 23 of a form, with its right-most character in printing-position 25. Apart from the position, the actual format of a printed detail and the printed representation of non-significant digits can be specified, using the MAP and ZERO statements respectively. Thus if a detail has associated with it the statement

MAP = £LLLLL.SS.D

then it will be printed in a format such as

£12345.15.6

With the same detail would be associated a statement such as

ZERO = £ZZZZ0.$0.0

These particular MAP and ZERO statements will cause various sterling amounts to be printed in formats typified by

£95481.18.9

£209. 9.4

£0. 11.7

£0. 0.0

since the letter Z in the ZERO statement means 'no character'.

Other statements allow data to be printed in a format which bears no direct relationship to its internal form, by means typified by

```
PRINT 1 AS "MALE"
PRINT 0 AS "FEMALE"
```

causing printing of either MALE or FEMALE according as a particular detail (called, say, SEX) has the value 1 or 0.

The facilities available in the Procedure Description provide for all the usual arithmetical operations, directions for input and output of data, the use of routines, table searching, etc.

A full description of Nebula is given in Ferranti List LD12.

8. THE OPERATING SYSTEM AND THE MONITOR PROGRAM

The Orion central computer and peripheral equipments put powerful computing resources at the disposal of the user, and efficient use of these resources is facilitated by the advanced design of the Orion operating system.

This design is influenced by three major factors. The first of these is the introduction of time sharing, with the sharing also of other physical resources (such as peripheral equipment and storage space) that this implies. The second is the sheer volume of input and output that the system can dispose of in quite a short time; this demands simple and foolproof operating procedures that can be carried out by largely semi-skilled staff. The third is the complexity of the large programs that may well be run on Orion; this demands comprehensive facilities for monitoring and program development.

The Orion operating system enables a single installation to be shared by several simultaneous users in such a way that each of these users in effect has at his disposal his own computer, tailored to fit his own particular program, operating independently of all the other programs running at the time and providing him with all the facilities that he would have if he were the sole user of the installation, To a large extent this is achieved by special built-in programs operating ‘behind the scenes’ in a way that provides exceptional flexibility at relatively low cost.

An important advantage of the Orion operating system is that the resources of an installation are allocated entirely according to the requirements of the jobs that are in hand at any given time. The operating system is not based on any fixed division of computer time, storage space or peripheral equipment, or on any fixed number of jobs. On one occasion there may be a number of small jobs running, and on another all available storage space and peripheral equipment may be devoted to a single very large job.

8.1 The Job Directory

This is an important set of data which is created and used by the Monitor Program. It gives all the basic information required by the Monitor Program about the jobs currently in the system. Each job has a directory-entry, mentioned briefly in Section 4.7 above, which includes the following items:

Name of the Job

Working-store reservation setting

Drum-store reservation settings

Peripheral Devices reserved

Total time allotted

Monitor Settings

Time unused so far

Link address (after an interruption of the program)

Type of Lockout (after a time-sharing interruption of the program)

Index or Pointer Words, giving the position of the directory-entry of the next jobs in the priority-list and the peripheral queue.

Much of this information originally enters the system in the form of directives on the job tape, as described in Section 8.3.

The job directory can conveniently be thought of as a single compact ordered set of data, and it is scanned by the Monitor Program as if it were so; in fact, however, its entries are widely dispersed throughout the working store, because each one occupies 16 words immediately following the working-store space reserved for the program to which it refers. This arrangement ensures that the space occupied by the job directory is only the minimum necessary for the number of jobs currently in the system; none is wasted to provide for some maximum number of jobs which may never be encountered.

The entries in the directory are ordered by the index words, because each of these stores the address of, and thus 'points to', the index word of the next entry in order. Similar items of information have the same position relative to the index word in every entry; the Monitor Program can thus scan the entries in a predetermined order for any particular kind of information simply by repeated replacement and modification, taking from its own working space the index word address of the first ordinary program entry in the directory. The way in which the entries are ordered by the index words is of fundamental importance to the Time Sharer, for reasons that were described in Section 4.7.

When, for one or another of a number of reasons, a program cannot be allowed to continue, the Monitor Program sets a special mark in the program' s directory-entry, and as long as this mark is present the Time-sharer will not select that program. Programs are said to be inactive or active, according to whether or not they are marked in this way.

8.2 Jobs and Job Tapes

8.2.1 General. It does not often happen that a program is written, run once, and then thrown away. Most programs are run a number of times, at first just to get their details right and later on to solve the problems for which they have been written. This fact leads to a distinction which is important on Orion, namely that which exists between jobs and programs. As understood in connection with Orion, a program is an assembly of information which defines a procedure, or a set of procedures, for operating on suitable input data; a job, on the other hand, is the actual execution of a program, on a particular occasion, with particular values of parameters, particular sets of input data and so on.

The Orion operating system must recognise this distinction because Orion is a timesharing machine and, for example, may be required to carry out concurrently two jobs which both use the same program. The Monitor Program therefore keeps a directory of all the jobs on the computer, and allocates storage space, peripheral devices etc., to jobs rather than to programs.

The way this is done allows the specification of a job to be physically quite separate from its program, and also allows one program to be used for many different jobs without having to be re-edited. Furthermore, it enables the specification of even a complex job to be reduced to a comparatively simple task that can readily be carried out by semi-skilled staff.

This section outlines the way in which jobs are specified and introduced to the system.

8.2.2 Job Tapes and the Job Directive. The Specification of a job takes the form of a short program conventionally known as a job tape, although in practice it may be on either punched paper-tape or punched cards. It consists almost entirely of directives, and its general form, although not its precise content, is prescribed by the programmer responsible for the main program of the job. This ensures that all essential information which is not in the main program will be given in the job tape, and conversely, that there will be nothing in the job tape that conflicts with what is already in the main program.

The job tape can be loaded in various ways. It may be physically separate from other inputs for the job; it may be at the beginning of some other input particular to the job, such as input data; it may simply be the beginning of the main program itself. However it is loaded, there must always be a job tape and it must always start with a Primary Input JOB directive which names the job, e.g:

```
JOB    TABLES
```

because this is the only way in which a new job can be introduced to the computer. The job name can be any 7-letter combination that is convenient for the purpose of identification.

The programs to be read in for most jobs will be in one or another of the various programming languages available, but some jobs will involve restarting an already assembled program which has earlier been dumped in non-relocatable machine language (i.e. absolute binary) on magnetic tape. Such programs must operate with the datum point and working store reservations originally used and these items can therefore optionally be specified with the JOB directive.

When the Select button of the input device loaded with the job tape is pressed, an interruption occurs and the Monitor Program is entered. This reads the JOB directive and reads and stores the job-name, but not the rest of the job tape, and immediately takes various actions is preparation for the job: it sets up a new entry in the job directory, it assigns the datum point and reserves the working-store space requested in the job directive or, in the absence of such requests, chooses a suitable datum point and provisionally reserves 944 working-store registers. It reserves for the job 64 words in the drum-store. It reserves the device on which the job tape is loaded. It confirms acceptance of the job by printing and punching a Flexowriter message which gives the job name, the time of acceptance, the working-store space reserved and the datum point, and statements that drum-store space and the current input device have been reserved, e.g:

```
TABLES      ACCEPTED        13.41.57
            CORE            1712
            DATUM           704
            DRUM            64
            RESERVED *SR20  -  TRA
```

It then copies the first chapter of the Basic Input Routine from the drum into the working-store space provisionally reserved and enters this chapter, which continues reading in the job tape. If any of these preliminary actions cannot be carried out, e.g. because of shortage of storage space, the Monitor Program rejects the job and prints a message on the Flexowriter, thus:

```
TABLES      REJECTED
```

Every job is initially given, in its own storage space, its own copy of the first chapter of the Basic Input Routine, so that several job tapes can be read in simultaneously by time-sharing.

8.2.3 Layout of Basic Job Tapes. Routine production jobs will generally use Basic Language programs; so may development jobs when the program is almost fully developed. In these cases, the job tape will be in Basic language.

The following example shows, with explanatory notes, the layout of a typical job tape for a Basic program in the final stages of development:

Job Tape Directives	*Items Requested, etc.*
JOB TABLES	*Acceptance of new job TABLES*
RESERVE *CORE 1000	*Working Store Space - words*
RESERVE *DRUM 2000	*Drum Space - words*
RESERVE *SR1 TABLES/INPUT/DATA	*7-track tape reader*
RESERVE *SPI TABLES/OUTPUT/DATA	*7-track tape punch*
REPORT 2 *LP1 TABLES/MONITOR	*Line printer for monitoring information*
	Preset Monitor Action:
MONITOR *OVR 1	*Style 1 on overflow*
MONITOR *JUM 2	*Style 2 on jumps*
TIME 25	*Computer Time - minutes*
READ TABLES/PROGRAM/MK3	*Read Main Program*

Expressions such as TABLES/INPUT/DATA are 'document names' and the way in which they are used in RESERVE and other directives is described in Section 8.3. The MONITOR directive is used as shown above to set the desired styles for the various program-events. All RESERVE directives are acknowledged by the Monitor Program, which prints on the Flexowriter and punches on the log-tape a record of the storage space and peripheral devices reserved for the job. If any of the items requested cannot be provided, the job is rejected with a suitable notification on the Flexowriter.

The production version of the job tape shown above would probably omit the REPORT and MONITOR directives, which are primarily development facilities, and might also omit the request for working store space, if this could be the same for all runs of that program and therefore go in the main program. The job-tape would then typically be as follows:

```
JOB TABLES
RESERVE *DRUM 2000
RESERVE *SRI  TABLES/INPUT/DATA/2
RESERVE *SP1  TABLES/OUTPUT/DATA/2
READ  TABLES/PROGRAM/MK4
```

The READ directive at the end of the job tape causes the Basic Input Routine to start reading in the main program from some other input device; it is not required if the job tape immediately precedes the main program on a single length of paper tape or a single pack of cards. When it is encountered, the device on which the job tape is being read is relinquished.

The last main program item to be read in is usually an ENTER directive which specifies an entry point of the program by means of a numerical identifier, thus:

ENTER 2

This directive marks the end of program input and assembly. On reading it. the Basic Input Routine makes a number of final checks, and then calls in the Monitor Program to copy the required chapter of the main program into the working store and enter it at the specified point. When this is done, the job's copy of the Basic Input Routine is overwritten and lost.

8.2.4 Layout of Symbolic Job Tapes. A program may be written originally in Symbolic language and used in this form until it is well enough developed for the Basic language version to be used instead. While it is still in Symbolic form, the job of running it can be introduced by a Symbolic job tape.

This can make use of various features of Symbolic language such as, for example, Symbolic identifier equations to set parameters: for ordinary program runs, however, it will differ little in other respects from a Basic job tape, because both languages have virtually the same set of organisational directives for requesting storage space etc.

On a Symbolic job tape, the JOB directive is followed by a Basic COMPILER directive, which calls in the Symbolic Input Routine. If this is already in the drum-store, its first chapter is copied into the job's working-store space and entered at once; if not, the Basic Input Routine reads it in and stores it on the drum first, provided it is loaded on an input device. The Symbolic Input Routine then reads in the rest of the job tape and the main program, compiles the latter into Basic language, stores the Basic version of the program on the drum and recalls the Basic Input Routine to assemble it. The Basic Input Routine then operates in much the same way as it would on a program originally written in Basic language, except that it reads the program from the drum instead of from an input device.

A Symbolic job tape may also be written for a job which compiles or corrects, but does not actually run, a Symbolic problem program. The main program of such a job is the Symbolic Input Routine itself, and the problem program is, in effect, input data. The following annotated illustrations show typical job tape layouts for these cases:-

(1) For a Program Compiling Run

JOB PROGCOM	*Arbitrary Job Name*
COMPILER SYMBOLIC	
BASIC *SP1 TABLES/BASIC	*Compile and output Basic version*
READ TABLES/SYMB	*Read in Symbolic version*;

(2) For a Program Correction Run

JOB PROGMOD	*Arbitrary Job Name*
COMPILER SYMBOLIC	
CORRECT TABLES/SYMB/MK1	*Read in original version*
WITH TABLES/MODS/1	*Read in corrections; correct original version*
GIVING *SP1 TABLES/SYMB/MK2	*Output corrected symbolic version*

(3) For a Correction and Recompiling Run

```
JOB      CORCOMP                          Arbitrary Job Name
COMPILER SYMBOLIC
CORRECT  TABLES/SYMB/MK1                  Read in original version
WITH     TABLSS/MODS/1                    Read in corrections;  correct
                                               original version
BASIC    *SPI    TABLES/BASIC/MK2         Compile  and output corrected
                                               Basic version
```

The document names in these illustrations have been chosen so as to indicate what each document actually is, but the terms used are not parts of the directives; in practice, the document names are quite arbitrary.

In the cases (2) and (3) above, the correction data would consist of a series of Symbolic ALTER directives each introducing an alteration, deletion, or insertion to the original program; these would all be punched on a single length of tape or deck of cards and loaded on a suitable input device in the same way as the program itself. This method of organising and executing program corrections is a very valuable feature of Symbolic language.

8.2.5 Job Tapes for Autocode Programs. These are written in Basic language, but it is not possible to generalise further about their layout, because this will depend on the compiler program concerned. The manual for each compiler available with Orion will, however, contain full details of job tape layout, for both compiling runs and problem program runs, if these are separate jobs.

8.2.6 Ending and Rerunning Jobs. A job generally ends by being 'abolished'. This means that its storage and peripheral reservations are cancelled, appropriate terminating action is taken on all peripherals and its directory-entry is deleted. These actions are taken by the Monitor Program in response to a Primary Input ABOLISH directive or a special instruction which has the same effect, at the end of the program. When abolishing a job the Monitor Program also prints out a Flexowriter message giving the job name, the time, the amount of central computer time in hours, minutes and seconds used by the job and statements regarding the relinquishing of peripheral devices, thus:

```
TABLES      RELINQUISHED  *SR1    TRB FREE
            RELINQUISHED  *SP1    SPC FREE
            ABOLISHED             00.15.34
            14.13.23
```

where the central-computer time used was 15 minutes 34 seconds, and the job was abolished at 14 h 13 m 23 secs, local civil time. This message is also punched on the log tape by the Flexowriter punch. When a job is abolished, all the storage space and peripheral devices formerly reserved for it immediately become available for allocation to new jobs, or to existing jobs whose requirements have increased.

There is one case in which a job is not abolished, although it has been completed. This is when the program ends with a particular instruction which informs the Monitor Program that the job is to be 'halted awaiting rerun'.

The rerun facility is used when the programmer requires several runs of the same program one after the other, each with a different set of input data. If all these runs are treated as separate jobs, the program will have to be read in and assembled for every one, even though this is not really necessary. Instead, therefore, the assembled program and its storage reservations can be preserved by a special instruction at the end of each run, and the second and subsequent runs are initiated by rerun tapes, special forms of job tapes. A typical rerun tape would be:-

```
RERUN      TABLES
RESERVE    *SR1        TABLES/INPUT/DATA/3
RESERVE    *LP1        TABLES/OUTPUT/DATA/3
ENTER      2
```

The rerun tape starts with a RERUN directive which names the original job. When the Monitor Program reads this, it associates the rerun with the original job and calls in the Basic Input Routine to read the rest of the rerun tape, which is always in Basic Language. In this case, the only new information required for the rerun are the identifications of the input and output data. The rerun tape ends with an ENTER directive because the program is already assembled on the drum and can be entered immediately.

A rerun tape can also alter other settings, such as monitoring conditions or the initial contents of certain registers. However, since it relates to a program already assembled in machine language, it cannot reset any quantity whose meaning is defined only in the original Basic or Symbolic version. Any recompilation or reassembly of the program will require a new job.

When the last rerun of a series has been completed, the whole job is abolished by entering on the Flexowriter keyboard an ABOLISH directive which overrides the instruction in the program which allowed for the rerun.

8.2.7 Supervisory Facilities. In addition to JOB, RERUN, and ABOLISH, there are certain other Primary Input directives which assist the operator to review the current commitments of the system-E and decide what further jobs can be loaded. These include the following:

DIRENT - preceded by a job name, this causes output of certain information from that job' s directory-entry. It can be entered on the monitor Flexowriter or on a tape or card reader, and it produces the required information on the monitor output device of the job.

NAMES - causes output of the names of all the jobs currently in the store.

SPACE - causes printing of details of those parts of the system (viz. working-store, drum-store and peripheral devices) which are currently not allocated.

8.3 Documents and Peripheral Devices

8.3.1 General. On Orion, a document is any identifiable set of information, other than a job tape or rerun tape, which is recorded on a peripheral medium and used or produced by the computer.

Even a small job may involve quite a variety of documents. For example, there may be a program on punched paper-tape, library routines on magnetic tape, input data on punched cards and output to a line printer and punched cards. Development jobs with diagnostic tapes and monitor print-outs, or data-processing jobs with new and old main files, amendment files etc., may involve still more documents. Add to this the fact that several jobs can run concurrently, owing to time sharing, and it will be seen that the number of documents to be handled by the operator may on occasion be considerable.

Valuable time may be wasted if the operator confuses documents or loads them on the wrong input devices, and the Orion operating system is therefore so designed that, from the operator' s point of view, all document handling is as simple and foolproof as possible.

For input documents, the operator has merely to load all those required for the job in any order on any available input devices of appropriate types, pressing the Select buttons of these devices. For output documents, in general, she has only to ensure that all output devices are adequately loaded with blank tape, blank cards etc., and that they have been made available to the computer by pressing their Engage buttons. Her actions are extensively checked by the Monitor Program which prints confirmations, queries and requests on the control Flexowriter whenever necessary.

Below are outlined the methods adopted for dealing with various kinds of documents, but in the interest of simplicity certain details of procedure and programming technique are omitted.

8.3.2 Identification of Documents. Every Orion document must have a name, so that the operator and the computer will know what document it is, and so that the programmer can refer to it in his program. A document name consists of two to eight components, separated by solidi, and each component can have up to eight characters which must be letters, numerals, or points; within these rules, the name can be anything the programmer chooses. Two simple document names which have already appeared in this description are:

TABLES/INPUT/DATA

TABLES/PROGRAM/MK3

but a more typical one is

GEN/WAGES/UPDATE/HOURROLL/PROG/JLB/11.12.63

For documents on punched paper-tape or punched cards, the document name is written on the tape leader or leading card, and a Primary Input DOCUMENT directive giving the name is punched at the beginning of the document itself, thus:

```
DOCUMENT        TABLES/INPUT/DATA
```

For documents on magnetic tape, the document name is written on a label attached to the reel and also recorded in the first block of the tape, known as Block 0; absence of this information denotes a new tape. On printed output documents the name is printed at the top of the paper before the data themselves.

8.3.3 Identification of Peripheral Devices. Every peripheral device on Orion must have a name so that programs can refer to it, but this is complicated by time sharing. A programmer who writes a program referring to a tape punch, for example, must do so without knowing which punch it will actually use when run, because this will depend on what other jobs are running, and which punches they leave free, at the time. Every device in use therefore has two names, a permanent geographical name by which it is known to the operator and the Monitor Program, and a temporary 'programmer' s name' by which it is known to the program currently using it.

The programmer's name of a device occurs in Basic and Symbolic programs. It consists of an asterisk, to distinguish it from an identifier, followed by two letters which specify the type of device, and a serial number, e.g.

*LP1	Line Printer
*SR2	7-track paper-tape reader
*FP3	5-track paper-tape punch
*MT1	Magnetic Tape Deck

The geographical name of a device is displayed on the device itself and also occurs in input or output message about it on the control Flexowriter. It is somewhat similar to a programmer's name, but there is no asterisk and a serial letter is used instead of a number, e.g.

LPA	line printer
TRB	paper-tape reader

When the operator loads an input document on an input device and presses its Select button, an interruption occurs and the Monitor Program is entered. This reads the DOCUMENT directive or Block 0 of magnetic tape, but not the document itself, and stores the document name together with the machine address of the device.

The operator is not concerned at all with linking together geographical names, programmer's peripheral names and document names; this is done quite automatically by the Basic Input Routine and the Monitor Program during input of the problem program.

8.3.4 Data Input Documents. Data input documents are read in by the problem program, and the system must assign the peripheral names specified by the programmer to the input devices on which the documents are loaded. The way that this is done can best be explained by an example. Suppose that the operator loads a job tape for a job called TABLES; that this uses a data document on paper tape called TABLES/INPUT/DATA; that the program will read the document on a tape reader which it calls *SR1; and that the operator loads the document on a particular reader called TRA.

On reading in the job tape, the Basic Input Program will encounter the following RESERVE directive:

```
RESERVE      *SR1    TABLES/INPUT/DATA
```

The system keeps count of all the SR devices which are free, i.e. available for future reservation, and the count is tested when this directive is read. If it is zero, there is no SR available, the job cannot be done and is therefore rejected. The operator is then notified by the following Flexowriter message, which is also punched on the log tape:

```
TABLES         NO      SR
```

and the job is stopped until a 7-track paper-tape reader becomes free. If the count is not zero, it is reduced by one, and further action depends on whether or not the document has already been loaded. If it has, the Monitor Program has recorded that it is on TRA and so associates the name *SR1 with TRA. The reservation is then printed on the Flexowriter and punched on the log tape, thus:

```
TABLES         RESERVED  *SR1 - TRA
```

However, it should particularly be noted that the operator does not in fact have to know either of these names in order to set up and run the job correctly.

If the document has not yet been loaded, the reservation remains floating, i.e. the count of free SR' s is reduced by one, but no particular device is reserved. The operator is then requested to load the document by the following Flexowriter message:

```
LOAD       TABLES/INPUT/DATA
```

The Basic Input Routine continues to read the job-tape and the program, and will not be held up provided the document is loaded before the ENTER directive is read, i.e. by the time the problem program is ready to run. Otherwise it will be held up at that point and the request to load will be repeated, but as soon as this is done it will continue automatically.

The main advantages of these arrangements can be summarised as follows:

(i) an operator can load data documents for a job in any order on any available input devices, either before or after loading the job tape,

(ii) in the latter case, floating reservations ensure that the job will not be accepted unless sufficient input devices are available for all the documents required,

(iii) the programmer does not need to know the geographical names, and the operator need not know the programmer' s names, of the input devices used.

8.3.5 Data Output Documents. Data output documents are produced by the problem program, and the system must assign the peripheral names specified by the programmer to the output devices used. Suppose that the job TABLES is to produce an output document called TABLES/OUTPUT/DATA on a tape punch with the programmer's name *SP1.

While reading the job tape, the Basic Input Routine encountered the following RESERVE directive:

```
RESERVE    *SP1    TABLES/OUTPUT/DATA
```

whereupon the Monitor Program selected an available device of the stated type (7-track paper-tape punch in this case) and allocated it to the job TABLES, printing on the control Flexowriter the message

```
TABLES    RESERVED    *SP1 - SPB    (say)
```

If at least one such device was engaged at that time, one of them was allocated. If, however, no such devices were engaged, a disengaged one was allocated and a request that it should be engaged was printed. If no such device was available, then the job was stopped and notification printed on the Flexowriter. As soon as a device is both allocated and engaged, the document-name in the RESERVE directive is output on it.

8.3.6 Magnetic-tape Documents. Since a magnetic-tape deck can be used for input or output or both by one job, the operating system has to allow for this and, in particular, ensure that valuable data on a tape are not destroyed by inadvertently allowing the tape to be written on.

Consider the creation of a new magnetic-tape file. Then this requires a tape which can be written on freely; if the tape already carries data, they must be of no future value. Such a tape is described as 'scratch'. Once the new data have been written on the tape, they must be protected from inadvertent overwriting for a time, perhaps for the next two cycles of the file-updating run (e.g. for two weeks if the file is updated weekly).

To allow for this, Block 0 on each reel of tape includes a 'void-date'. If the void-date has arrived or passed, the tape is considered scratch and can be written on freely. To reserve a scratch reel of tape (to create the new file) a job-tape directive typified by:-

```
NEW    *MT3      29.11.63     MY/MAIN/FILE/12.11.63
```

is used. Then any available magnetic-tape deck carrying a scratch tape is reserved as *MT3 for the job, and the new void-date (29.11.63) and document-name (MY/MAIN/FILE/12. 11.63) are automatically written into Block 0.

When this version of the file is to be read at the next updating run, it is called for by a job-tape directive such as

```
RESERVE      *MT1        MY/MAIN/F1LE/12.11.63
```

and cannot be used by any job except the one which calls for it by its document-name.

However, on 29.11.63 the tape again becomes scratch and so can then be written to by any program which calls for a scratch tape. Note, however, that before even a scratch tape can be written on, the computer operators must:-

(i) fit a 'write-permit ring' to the spool before mounting it on the hub of the mechanism,

(ii) put a switch on the tape-deck into the 'write-permit' position.

Suppose, though, that having created such a magnetic-tape file, it is required deliberately to write upon it, e.g., to add new blocks of data. Then this can be done, provided that the programmer whose program created the file took deliberate steps in his program to allow it.

In Block 0 are two particular bits which can be set by program, called the 'write-permit bit' and the 'date-control bit' respectively. The date-control bit determines whether the date in Block 0 is a void-date or a write-permit date, according as that bit is 0 or 1 respectively. There are four possible combinations:-

(i) Write-permit bit = 0, date-control bit = 0. Before the void-date the tape cannot be written to, while on the void-date the tape becomes scratch exactly as described above.

(ii) Write-permit bit = 0, date-control bit = 1. On and after the write-permit date, but not before, the tape can be written to by the job which calls for it by its document name.

(iii) Write-permit bit = 1, date-control bit = 0. Before the void-date, the tape can be written to by the job which calls for it by its document-name. On and after the void-date, the tape is scratch and can be written to by any job.

(iv) Write-permit bit = 1, date-control bit = 1. The tape can be written to at any time by the job which calls for it by its document-name.

One effect of the write-permit date is to allow a user (e.g. a department within an organisation) to have exclusive use of particular reels of tape, since such tapes must be called for by their document-names.

In these ways, the operating system provides complete protection for valuable data on magnetic-tape while allowing data to be overwritten when no longer of value.

APPENDIX A

This Appendix sets out the maximum numbers of the various types of peripheral devices which can be attached to an Orion computer, the maximum number of transfers which can take place simultaneously, and related information.

1. ORION 1

On Orion 1, each device or group of similar devices is attached to and controlled by an appropriate type of Peripheral Control Unit. Overall control is exercised by the Central Peripheral Control Unit.

1.1 Ampex TM2 Magnetic-tape Units

Up to nine TM2 units can be fitted, associated with 1 or 2 Magnetic-tape Control Units. Each control-unit can handle one transfer (reading or writing) at a time.

If a single control-unit is fitted, all the decks are connected to it, and any one deck can be engaged in a transfer at a given time.

If two control-units are fitted, all the decks are connected to both and, when a transfer is called for on a particular deck, that deck is connected to the working-store through either of the control-units, provided one happens to be free at that time; if both control-units are busy the new transfer cannot be started and the program which attempted to initiate it is interrupted. Thus, with two control-units, up to 2 decks can be operative simultaneously but these can be any two decks.

1.2 ICT 593 Card-reader

One card-reader can be fitted, associated with a Card-reader Control Unit. It can operate independently of all other types of peripheral device.

1.3 ICT 665 Line-printer, Anelex Type 4-1000 Line-printer, ICT 582 Card-punch

If one or more devices of these types is attached, then one Printer and Card-punch Control Unit (also known as a Parallel Output Control Unit) must be fitted. This Control Unit can have several devices attached to it: counting each ICT665 as 1 device, each ICT582 as 2 devices and each Anelex 4-1000 as 3 devices, then each Control Unit can control up to the equivalent of 8 devices. Note, though, that the maximum number of ICT665 printers is 3, and the maximum number of Anelex 4-1000 printers is 2; these maxima are independent of the overall limit for the Control Unit.

All devices attached to the Control Unit can operate simultaneously.

1.4 Paper-tape Devices, including the Control Flexowriter

Attached to every Orion 1 must be either 2 or 3 Character Control Units, to control the paper-tape readers and punches and the Flexowriter.

Each Character Control Unit can control up to 4 devices of any type; in this context the Flexowriter counts as 2 devices (one input and one output).

Each Control Unit can be engaged in one transfer (reading or writing) at a time. Note that each device in this category is associated with one particular Control Unit; there is no 'exchange' as with magnetic-tape decks.

There must be at least one paper-tape reader, one paper-tape punch and a Flexowriter.

1.5 Magnetic Drums

One Drum Control Unit is fitted to every Orion 1 and can control up to 4 drums, carrying out one transfer (reading or writing) at a time. A transfer can start on one drum and finish on another, the changeover from one drum to the next being handled automatically by the Control Unit; to the programmer, there is a single drum-store composed of consecutively addressed locations.

1.6 Rank Xeronic Printer

One printer of this type can be attached; it requires a Xeronic Printer Control Unit. The Xeronic Printer can operate independently of all other devices.

2. ORION 2

Except for the magnetic-tape decks and the drums, each peripheral device attached to an Orion 2 requires its own control-unit. Any selection of up to 20 devices (excluding magnetic-tape decks and drums) can be attached, each operating simultaneously with and independently of any others.

Up to 60 Ampex TM2 magnetic-tape decks can be attached. Up to 20 can be controlled by one pair of control units, and up to 3 pairs of control units can be fitted to increase the possible number of simultaneous transfers, since each control-unit can handle one transfer at a time.

Any one deck is connected to both members of one pair of control-units. Thus a deck can become engaged in a transfer if either (or both) of the control-units to which it is connected is free. However, a deck cannot be controlled by any other control unit.

Either 1 or 2 Drum Control Units are fitted. Each control unit can handle up to 4 drums, performing one transfer at a time. Although, with two control units, a given drum can be controlled only by the control-unit to which it is connected, this does not mean that the drum-store must be regarded as being in two parts; the drum-store can be thought of as a single set of consecutively addressed locations, the changeover from one drum to another, or from one control-unit to the other being handled automatically.

APPENDIX B

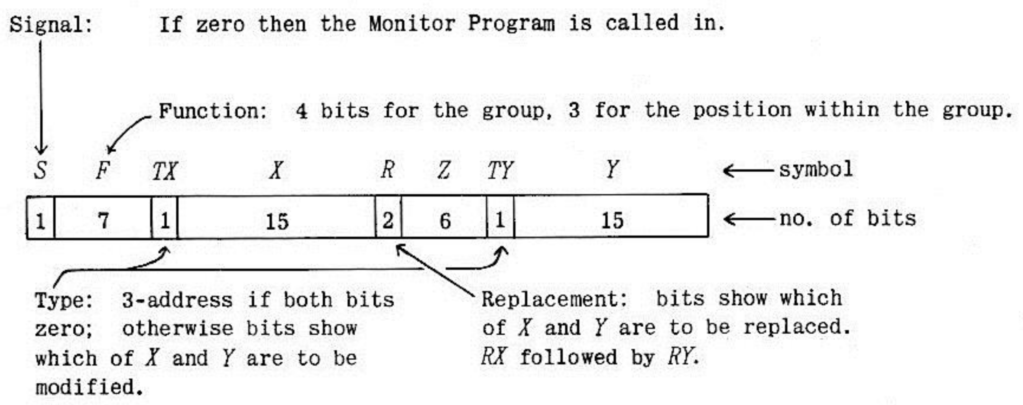
INSTRUCTION - REPERTORY

Notation

X, Y	The two main addresses in the instruction being described, after possible modification or replacement (effectively 24 bits each).
Z	The six-bit address in the instruction.
x, y, z x', y', z'	The contents of X, Y and Z; primed symbols refer to the contents after the instruction has been obeyed. x , y,
PY	The pseudo-register numbered Y.
pY	The content of the pseudo-register PY.
&, v, $\not\equiv$	Logical operations: "and", "or", "not-equivalent".
$\bar{a}$	The bar indicates the logical "not" operation.
$(\ )_r$	The result of rounding the quantity in parentheses.
d.l.	Double-length or double-precision.
s.l.	Single-length or single-precision.
m.s.	More or most significant.
l.s	Less or least significant,
ε	2^{-47}
x*	The content of X + 1.
x:	The d. 1. quantity formed from x and x*
x_F	x regarded as a fraction (= εx_I)
x_I	x regarded as an integer.
$x:_F$	x: regarded as the d.l. fraction $x_F + \varepsilon x^*_F$
$x:_I$	x: regarded as the d.l. integer $x_I.2^{47} + x^*_I$
$x:_M$	x: regarded as the d.l. mid-point number $x_I + x^*_F$
$x:_L$	x: regarded as a 96-bit logical quantity.
x_G	x regarded as a floating-point number.
x_m	The modifier in *X* (l.s. 24 bits of x)
x_u	The upper half of x (m.s. 24 bits)
x_s	The sign-bit (m. s. bit) of x.

x_c	The unsigned integer represented by the three m.s. bits of x.
c	The control-number (address of current Instruction).
OVR	The overflow-indicator (the contents of pseudo-registers *P4, P5, P6* and *P7* depend on the state of overflow).

Instruction-format


Signal: If zero then the Monitor Program is called in.
Function: 4 bits for the group, 3 for the position within the group.
S F TX X R Z TY Y
symbol
1 7 1 15 2 6 1 15
no. of bits
Type: 3-address if both bits zero; otherwise bits show which of X and Y are to be modified.
Replacement: bits show which of X and Y are to be replaced. RX followed by RY.

Note: Blank entries in this instruction-code denote illegal functions (e.g. 37), attempted use of which will result in suspension.

FIXED-POINT ADDITION/SUBTRACTION INSTRUCTIONS

	3-address form	2-address form
00	$z' = x + y$	$x' = x + y$
01	$z' = x - y$	$x' = x - y$
02	$z' = y - x$	$x' = y - x$
03	$z' = -y$	$x' = -y$
04	$z' = y$	$x' = y$
05	$z' = x \,\&\, y$	$x' = x \,\&\, y$
06	$z' = x \vee y$	$x' = x \vee y$
07	$z' = x \not\equiv y$	$x' = x \not\equiv y$
10	$z' = x + Y$	$x' = x + Y$
11	$z' = - x - Y$	$x' = x - Y$
12	$z' = Y - x$	$x' = Y - x$
13	$z' = -Y$	$x' = -Y$
14	$z' = Y$	$x' = Y$
15	$z' = x \,\&\, Y$	$x' = x \,\&\, Y$
16	$z' = x \vee Y$	$x' = x \vee Y$
17	$z' = x \not\equiv Y$	$x' = x \not\equiv Y$
20	$z' = x + pY$	$x' = x + pY$
21	$z' = x - pY$	$x' = x - pY$
22	$z' = pY - x$	$x' = pY - x$
23	$z' = -pY$	$x' = -pY$
24	$z' = pY$	$x' = pY$
25	$z' = x \,\&\, pY$	$x' = x \,\&\, pY$
26	$z' = x \vee pY$	$x' = x \vee pY$
27	$z' = x \not\equiv pY$	$x' = x \not\equiv pY$

FIXED-POINT MULTIPLICATION INSTRUCTIONS

30	$z'_I = x_I\, y_I$	$x'_I = x_I\, y_I$
31	$z'_F = (x_F\, y_F)_r$	$x'_F = (x_F\, y_F)_r$
32	$z{:}' = xy$	$x{:}' = xy$
33	$z{:}' = z{:} + xy$	$x{:}' = z{:} + xy$
34	$z' = xY$	$x' = xY$
35		
36		
37		

FIXED-POINT DIVISION INSTRUCTIONS

Three-address forms are given; for the two-address forms write x' for z' and x*' for z*'.

40 Unrounded division of integers; z'_I = quotient, $z^*_I{}'$ = remainder when x_I is divided by y_I.

$$z_I' + (z^*_I{}'/y_I) = x_I/y_I;\quad 0 \le z^*_I{}'/y_I < 1.$$

(Note: z_I' = quotient, $z^*_F{}'$ = remainder when x_F is divided by y_F
z_F' = unrounded quotient when x_F is divided by y_I).

41 Rounded division of integers: z_I' = rounded quotient when x_I is divided by y_I
(or x_F by y_F).

$$z_I' = (x_I/y_I)_r;\quad -\tfrac{1}{2} \le (x_I/y_I) - z_I' < \tfrac{1}{2}.$$

(Note: z_F' = rounded quotient when x_F is divided by y_I.)

42 Division with double-length quotient: z_I' = signed integral part
and $z^*_F{}'$ = non-negative fractional part of rounded quotient when x_I is divided
by y_I (or x_F by y_F).

$$z{:}_M' = (x_I/y_I)_r \text{ or } (x_F/y_F)_r;\quad -\tfrac{1}{2}\varepsilon \le (x/y) - z{:}_M' < \tfrac{1}{2}\varepsilon$$

(Note: $z{:}_F'$. = rounded d. l. quotient when x_F is divided by y_I).

43 Rounded division of fractions: z_F' = rounded quotient of x_F divided by y_F.

$$z_F' = (x_F/y_F)_r;\quad -\tfrac{1}{2}\varepsilon \le (x_F/y_F) - z_F' < \tfrac{1}{2}\varepsilon$$

(Note: $z_F' = (x_I/y_I)_r$ and $z_I' = (x_I/y_F)_r$.)

44 Unrounded division with d.l. dividend: z_I' = quotient
and $z^*_I{}'$ = remainder when $x{:}_I$ is divided by y_I

$$z_I' + (z^*_I{}'/y_I) = x{:}_I/y_I;\quad 0 \le z^*_I{}'/y_I < 1.$$

45 Rounded division with d.l. dividend: z_F' = rounded quotient of $x{:}_F$ divided by y_F

$$z_F' = (x{:}_F/y_F)_r;\quad -\tfrac{1}{2}\varepsilon \le (x{:}_F/y_F) - z_F' < \tfrac{1}{2}\varepsilon$$

46, 47

SHIFT INSTRUCTIONS

50	$z' = x.2^{Y}$	s. l. arithmetical	$x' = x.2^{Y}$
51	$z' = x.2^{-Y}$	s. l. arithmetical	$x' = x.2^{-Y}$
52	z' = x up *Y* bits	s. l. logical	x' = x up *Y* bits
53	z' = x down Y bits	s. l. logical	x' = x down Y bits
54	$z:' = x:.2^{Y}$	d. l. arithmetical	$x:' = x:.2^{Y}$
55	$z:' = x:.2^{-Y}$	d. l. arithmetical	$x:' = x:.2^{-Y}$
56	$z:_L' = x:_L$ up *Y* bits	d. l. logical	$x:_L' = x:_L$ up *Y* bits
57	$z:_L' = x:_L$ down Y bits	d. l. logical	$x:_L' = x:_L$ down Y bits

Note on shifts: In all shift-instructions Y is treated as signed. Arithmetical shifts may set OVR when shifting up, and are rounded when shifting down (when the sign-bit is also propagated). In double-length arithmetical shifts the shifting is preceded by a "partial-justify" operation, i.e. the operand is adjusted so that the l.s. half is non-negative, and shifting then skips the sign-bit of the l.s. half. In double-length logical shifts this sign-bit is included in the shifting.

JUMP INSTRUCTIONS

	3-address	**2-address**
60	Jump to *X* if $y = z$	Jump to *X* if $y = 0$
61	Jump to *X* if $y \neq z$	Jump to *X* if $y \neq 0$
62	Jump to *X* if $y > z$ (i.e. $z < y$)	Jump to *X* if $y > 0$
63	Jump to *X* if $y \ngtr z$ (i.e. $z \geq y$)	Jump to *X* if $y \ngtr 0$
64	Jump to *X* if $y < z$ (i.e. $z > y$)	Jump to *X* if $y < 0$
65	Jump to *X* if $y \nless z$ (i.e. $z \leq y$)	Jump to *X* if $y \nless 0$
66	Jump to *X* if $pY = z$	Jump to *X* if $pY = 0$
67	Jump to *X* if $pY \neq z$	Jump to *X* if $pY \neq 0$
70	Jump to *X* if $Y = z_m$	Jump to *X* if $Y = 0$
71	Jump to *X* if $Y \neq z_m$	Jump to *X* if $Y \neq 0$
72	Jump to *X* if $Y > z_m$ (i.e. $z_m < Y$)	Jump to *X* if $Y > 0$
73	Jump to *X* if $Y \ngtr z_m$ (i.e. $z_m \geq Y$)	Jump to *X* if $Y \ngtr 0$
74	Jump to *X* if $Y < z_m$ (i.e. $z_m > Y$)	Jump to *X* if $Y < 0$
75	Jump to *X* if $Y \nless z_m$ (i.e. $z_m \leq Y$)	Jump to *X* if $Y \nless 0$
76	Jump to *X* if $\neg y \,\&\, z = 0$	Jump to *X* if $\boldsymbol{Y} = 0$
77	Jump to *X* if $\neg y \,\&\, z \neq 0$	Jump to *X* if $\boldsymbol{Y} \neq 0$

80	$z' = z + 1$; jump to X if $z_m' = Y$	$y' = y + 1$; jump to X if $y_m' = 0$
81	$z' = z + 1$; jump to X if $z_m' \neq Y$	$y' = y + 1$; jump to X if $y_m' \neq 0$
82	$z' = z - l$; jump to X if $z_m' = Y$	$y' = y - 1$; jump to X if $y_m' = 0$
83	$z' = z - 1$; jump to Y if $z_m' \neq Y$	$y' = y - 1$; jump to / if $y_m' \neq 0$
84	Jump to X if character Z. is in y.	Jump to X if character "space" is in y
85	Jump to X if character Z is not in y.	Jump to X if character "space" is not in y
86	(2) Enter subroutine: $y_m' = c + 1$; $y' = OVR$; $OVR' = 0$; jump to X.	
	(3) As 2-address, but for y' read z'	
87	(2) Leave subroutine: $OVR' = OVR \vee y_s$; jump to $X + y_m$.	
	(3) As 2-address, but for y' read z'.	

FLOATING-POINT ARITHMETIC INSTRUCTIONS

90	$z_G' = x_G + y_G$	$x_G' = x_G + y_G$
91	$z_G' = x_G - y_G$	$x_G' = x_G - y_G$
92	$z_G' = y_G - x_G$	$x_G' = y_G - x_G$
93	$z_G' = - y_G$	$x_G' = - y_G$
94	$z_G' = x_G.y_G$	$x_G' = x_G.y_G$
95	$z_G' = x_G/y_G$	$x_G' = x_G/y_G$
96		
97	z_I' = no. of places needed to standardize $x_G - y_G$.	x_I' = no. of places needed to standardize $x_G - y_G$.

Note: The suffix G indicates a floating-point number ("gliding-point", since the suffix F is used for fractions): $x_G = x_a . 2^{x_e}$, where x_a is a signed fraction (the "argument" or "mantissa") represented by the m.s. 40 bits of the word, and x_e is an integer exponent ($-128 \leq x_e \leq 127$); the "characteristic", $x_e + 128$, is represented by the l.s. 8 bits of the word. The floating-point results of the above instructions are correctly standardized so that:- $\frac{1}{2} \leq x_a < 1$, or $-1 \leq x_a < -\frac{1}{2}$, or $x_a = 0$ and $x_e = -128$. They may be rounded or unrounded, as the programmer chooses.

CONVERSION INSTRUCTIONS

Note: (3) means three address (2) means two-address.

100 Convert mixed-radix number in character form to binary integer.

(3) $z' = y_7 (y_6 (\ldots(y_1 (y_0 z + x_0) + x_1) + \ldots) + x_7$, where x_i and y_i denote the characters in x and y; the two m. s. bits in each character are used for checking.

(2) $x' = y_7 (y_6 (\ldots (y_1 x_0 + x_1) + x_2) \ldots) + x_7$.

101		Convert signed binary integer to mixed-radix form.
	(3)	x' = characters resulting from conversion of z according to information in y and y*.
		Here y* contains the radices and zero-suppression information, and y contains the product of the radices. OVR is set if the converted form contains more than 8 characters, or if a minus-sign cannot be correctly inserted. z is unaltered.
	(2)	x' = characters resulting from conversion of x according to information in y and y*. Otherwise as 3-address form.
102		Convert scaled fraction to standard floating-point form.
	(3)	$z_G' = (x_F.2^Y)_r$ *Y* is treated as signed.
	(2)	$x_G' = (x_F.2^Y)_r$ *Y* is treated as signed.
103		Convert standard floating-point number to scaled fraction.
	(3)	$z_F' = (x_G.2^Y)_r$ *Y* is treated as signed.
	(2)	$x_F' = (x_G.2^Y)_r$ *Y* is treated as signed.
104		Convert card-input data from 4 columns to character-form.
105		
106		
107		

MISCELLANEOUS LOGICAL INSTRUCTIONS

	3-address	2-address
110	$x' = (x \& \neg y) \vee (z \& y)$	$x' = x \& \neg y$
111	$x' = (x \& \neg Y) \vee (z \& Y)$	$x' = x \& \neg Y$
112	$x' = (x \& \neg pY) \vee (z \& pY)$	$x' = x \& \neg pY$
113		
114	z' = y, y' = x	x' = y, y' = x (exchange)

115 Add l.s. 6 bits of *Y* to m.s. 6 bits of x, and rest of *Y* to l.s. bits of x, with end-around carry (cannot set OVR).

(3) z' = result, (2) x' = result.

116 Add *X*, *Y* to the corresponding addresses in the next instruction **before** any replacements therein. modify

117 Add *X, Y* to the corresponding addresses in the next instruction **after** any replacements therein. next instruction

120 Count 1-bits. If $Y > 0$ then read Y for N in the following description and count bits from the m.s. end of x. If $Y < 0$, read -Y for N and count bits from the l.s. end of x. In either case $N > 0$.

(3) z_m' = number of 1-bits in the first N bits of x, and z_s' = inverse of the $(N + 1)$th bit of x; rest of z' consists of 0-bits.

(2) Similar, but with x' in place of z'.

121 Cyclic shift.

(3) z' = x shifted cyclically right Y bits   Y signed

(2) x' = x shifted cyclically right Y bits   Y signed

122 Table look-up.

(3) x' = y shifted cyclically left N characters, where N is the unsigned integer given by the l.s. 3 bits of the written Z-address.

(2) x' = y shifted cyclically left z_c characters.

123 Insert field.

(3) Leave unchanged the first N characters of $x:_L$; replace the next 8 characters of $x:_L$ by the whole of y, and clear the rest of x:. N is the unsigned integer given by the l.s. 3 bits of the written Z-address.

(2) Leave unchanged the first z_c characters of $x:_L$; replace the next 8 characters of $x:_L$ by the whole of y, and clear the rest of x:.

124 Find end-most 1-bit. If $Y \geq 0$, then read Y for N in the following, with x shifted logically left. If $Y < 0$, read -Y for N, with x shifted logically right. In either case $N \geq 0$.

(3) x' = x shifted logically m bits; $z'_I = m$, where m is chosen so that the end-most 1-bit is just shifted off, provided that $0 \leq m \leq N$. If a shift of N places removes no 1-bits, then $z'_m = N$ and $z'_s = 1$.

(2) $x'_I = m$ (or $x'_m = N$; $x'_s = 1$), where m is defined above.

125 Standardize (normalize) unpacked floating-point number.

(3) $x:'_F = (x:_F + \upsilon).2^m$; $z'_I = z_I - m$; OVR' = 0;
where (a) υ is zero if OVR is clear but is $\pm$ 2.0 if set, the sign being opposite to that of x.

(b) $x^{*\prime} \geq 0$.

(c) m is an integer chosen so that

(i) $\frac{1}{2} \leq x:'_F < 1$ and $-1 \leq m \leq Y$,

or (ii) $-1 \leq x:'_F < -\frac{1}{2}$ and $-2 \leq m \leq Y$,

or (iii) $-\frac{1}{2} \leq x:_F' < \frac{1}{2}$ and $m = Y$.

(d) If $m < 0$, the shifting down which occurs is unrounded; this can happen only if $\upsilon \neq 0$ (i.e. if OVR set on entry).

(2) illegal.

126 Justify double-length number.

(2) $x' + \varepsilon y' = x + \varepsilon y + \varepsilon\upsilon$; $y' \geq 0$; $OVR' = 0$;
where υ is zero if OVR is clear but is ± 2 if set, the sign being opposite to that of y. OVR is left clear unless x' overflows.

(3) As above but the result is $z' + \varepsilon y'$ = justified form of $x + \varepsilon y$.

127

130 to 137

PERIPHERAL DEVICES; BLOCKS OF REGISTERS

Note: The instructions 140 to 143 are essentially 2-address instructions; the instructions 144 to 146 have only 3-address forms, the 2-address forms being illegal.

140.*M* Select device *Y*; prepare to carry out peripheral transfer of mode *M*.
Must be followed by a 142-instruction.

141.*M* Select drum-address *Y*; prepare to read if *M* = 1 or to write if *M* = 21.
Must be followed by a 142-instruction.

Note: In the above two instructions the integer *M* is represented by the five m.s. bits of X. The address *X* is conventionally written as zero; in fact its ten l.s. bits are used in stored instructions to extend the *Y*-address to 24 bits.

142 (a) A 142 after a 140 or 141: Initiate the peripheral transfer, copying *Y* words (or characters) starting at register *X*.

(b) A pair of 142's thus:

```
142    X1    Y1

142    X2    Y2
```

Copy Y2 words, starting at register X2, into consecutive registers starting at X. (N.B. Y1 is immaterial, conventionally zero.)

143 (3) Copy the content of the single accumulator *Z* into *Y* consecutive registers starting at *X*.

(2) Clear (or zeroise) *Y* consecutive registers starting at *X*.

144 Search a table starting at *X* until the first word w is found for which $w \geq y$;
set z' = address of w.

145 Search a table starting at *X* until the first word w is found for which $w \leq y$;
set z' = address of w.

146 Search a table starting at *X* until the first word w is found such that
either $w_u = y_u$ or $w_m = 0$; set z' = address of w.

147

TABLES OF MODES FOR ORION PERIPHERAL DEVICES

In this table, *Y* refers to the *Y*-address in the following 142-instruction (which causes *Y* characters or words to be transferred, starting at word *X* in the store).

Paper Tape

Mode 1	Read 7-track tape up to and including next "New Line" character, or at most Y characters
Mode 2	Read Y characters (7-track)
Mode 8	Read Y characters (5-track)
Mode 21	Punch 7-track tape up to and including next "New Line" character, or at most Y characters
Mode 22	Punch *Y* characters (7-track)
Mode 28	Punch *Y* characters (5-track)

Punched Cards

Mode 1	Read *Y* words in I.B.M. Code
Mode 2	Read *Y* words in I.C.T. Code
Mode 4	Read *Y* words in BULL Code
Mode 7	Read *Y* words in I.B.M. Code and binary
Mode 8	Read *Y* words in I.C.T. Code and binary
Mode 11	Read *Y* words in BULL Code and binary
Mode 14	Read *Y* words in binary
Mode 19	Read *Y* words, normal and interstage punching

In modes 1, 2, 4 and 14, data from the first 4*Y* columns are stored. In modes 7, 8, 11 and 19, data from the first 2*Y* columns are stored.

Mode 21	Fill the Data buffer with *Y* words and punch a card in accordance with the contents of the Code buffer.
Mode 22	Fill the Data and Code buffers with *Y* words and punch a card in binary
Mode 26	Fill the Code buffer with *Y* words

Line Printers

Mode 21	Fill the Data buffer with Y words and print a line in accordance with the contents of the Code buffer, using full character set (asynchronous mode).
Mode 22	Fill the Data buffer with Y words and print a line in accordance with the contents of the Code buffer, using partial character set (synchronous mode).
Mode 26	Fill the Code buffer with Y words

Magnetic Tape

Mode 1	Read forward 1 block, or its first *Y* words if the block contains more than *Y* words
Mode 2	Read backward 1 block, or its last *Y* words if the block contains more than *Y* words
Mode 14	Rewind
Mode 21	Write, with long inter-block gap, *Y* words as 1 block
Mode 22	Write, with short inter-block gap, *Y* words as 1 block
Mode 28	Erase

Xeronic Printer

Mode 21	Print up to and including next "New Line" character, or at most Y characters
Mode 22	Print Y characters

Drum Store

Mode 1	Read *Y* words from drum-store
Mode 21	Write *Y* words to drum-store

All Devices (except Drum-store and magnetic tape)

Mode 16	Disengage device

APPENDIX C

FERRANTI FLEXOWRITER CODE

Character in Computer	Character on tape	Upper Case printing	Lower Case printing	Character in Computer	Character on tape	Upper Case printing	Lower Case printing
0	0010.000	Space		32	1000.000		
1	0000. 001			33	1010.001	A	a
2	0000. 010	New line		34	1010.010	B	b
3	0010.011	Paper Throw-		35	1000.011	C	c
4	0000. 100	Tabulate		36	1010100	D	d
5	0010. 101	Back space		37	1000.101	E	e
6	0010. 110	Lower case		38	1000.110	F	f
7	0000. 111	Upper case		39	1010.111	G	g
8	0001.000			40	1011.000	H	h
9	0011.001			41	1001.001	I	i
10	0011.010			42	1001.010	J	j
11	0001.011			43	1011.011	K	k
12	0011.100	Stop		44	1001.100	L	l
13	0001.101	Punch on		45	1011.101	M	m
14	0001.110	Punch off		46	1011.110	N	n
15	0011.111	/	:	47	1001.111	O	o
16	0100.000	0	'	48	1110.000	P	p
17	0110.001	1	[	49	1100.001	Q	q
18	0110.010	2	]	50	1100.010	R	r
19	0100.011	3	<	51	1110.011	S	s
20	0110.100	4	>	52	1100.100	T	t
21	0100.101	5	=	53	1110.101	U	u
22	0100.110	6	_	54	1110.110	V	v
23	0110.111	7	|	55	1100.111	W	w
24	0111.000	8	(	56	1101.000	X	x
25	0101.001	9	)	57	1111.001	Y	y
26	0101.010	10	%	58	1111.010	Z	z
27	0111.011	11	£	59	1101.011		
28	0101.100	½	?	60	1111.100		
29	0111.101	+	&	61	1101.101		
30	0111.110	-	*	62	1101.110		
31	0101.111	.	,	63	1111.111	Erase	

Note: In the character on tape, 1 denotes a hole, 0 denotes no hole. On lower case the character valued 16 is apostrophe, and the character valued 22 is underline. All characters have an odd number of holes in the tape. The third track from the left in the representation above is the parity track.