THE PEGASUS

PROGRAMMING

MANUAL

THE PEGASUS PROGRAMMING MANUAL

By

G. E. FELTON, M.A.

Ferranti

PUBLISHED 1962

By

FERRANTI LTD.
THE LONDON COMPUTER CENTRE
68 NEWMAN STREET
LONDON W.1

PRINTED BY SPECIALISED PRINTING SERVICES LTD.
30-34 LANGHAM STREET
LONDON W.1

THE PEGASUS PROGRAMMING MANUAL

PREFACE

Digital computers are being put to more and more diverse uses and their numbers are rapidly increasing. The programmer has a key role to play in this technological explosion. He and his fellows are needed in ever greater numbers since the machines are useless without their skill.

This handbook describes the programming techniques used with Pegasus, a medium-sized computer of which about forty are being used in business and commercial data-processing, in technical and scientific calculations and in educational work. It is hoped that many of these techniques will be of use also to those concerned with programming and applying other machines of similar scope.

This manual originated in a series of programming courses for Pegasus which started in October, 1955, for which an earlier manual (Ferranti document CS 50) was written by Mr. P.M. Hunt and the author. The need for a more comprehensive manual, including descriptions of the facilities provided in Pegasus 2, has led to the present volume. While attendance at one of these courses is desirable for anyone wishing to prepare programmes for Pegasus, an attempt has been made to keep this manual self-contained except for a set of programming exercises, which is available separately from the publishers (Ferranti document CS 204). At the same time it should be stressed that no one can expect to master a subject such as this one without writing programmes and running them on a computer.

This book was originally written for Pegasus 1. The author is grateful to Mrs. Mary Blyton, who adapted it to include Pegasus 2 and collected the material for most of the Appendices, as well as preparing the manuscript for printing. He is also indebted to Mr. M.J. Marcotty, who wrote the whole of Chapter 11 and advised on the additional material relating to Pegasus 2. The author would also like to thank many of his other colleagues, too numerous to name, with whom he has had many fruitful discussions. The editing and printing of a book of this kind raise many problems; the author is grateful to the following, who have contributed much to their solution: Mr. J.W. Moffatt, Mr. J.F. Nicholson, who has also compiled the Index, and Mr. H.G. Stanton (of Specialised Printing Services Ltd.), whose care and helpful advice have been invaluable. Special thanks are due to Mr. B.B. Swann, whose encouragement and practical support have made this book possible. Lastly the author wishes to thank his wife, who loyally endured much interference with their home life while he was writing the book.

G.E.F.

CONTENTS

CONTENTS

C O N T E N T S

▼ Note: Some portions of the text can be omitted at a first reading without loss of continuity. Such portions are marked in the left-hand margin by ▼ at the beginning and ▲ at the end.
▲

Chapter 1

Introduction — Digital Computers

In this Chapter we describe digital computers in general and introduce some of the basic ideas of programming and the way numbers are represented in a computer.

1.1 Computers

This book is concerned with automatic electronic digital computers. A digital computer is a machine which can perform arithmetical operations on numbers represented in digital form. This way of representing numbers is the one with which we are most familiar, since it is in everyday use. For example the number 53 might be represented (in a mechanical digital machine) by two gear wheels, each with 10 teeth, one turned through 5 teeth and the other 3 teeth, relative to some standard position. In an electronic machine this number might be represented by two trains of pulses containing 5 and 3 pulses respectively. The ordinary way of writing numbers is digital.

By contrast, in analogue computers numbers are represented by some physical quantity, such as length, angle or electrical potential. In an analogue machine arithmetical operations are performed by using some law of Physics, e.g. Ohm's law, and then making a measurement to find the answer. The most familiar analogue computing device is a slide rule, which uses lengths to represent numbers (the lengths are proportional to the logarithms of the numbers they represent); these lengths are added and subtracted mechanically to give lengths corresponding to products and quotients. In an electronic analogue computer numbers are usually represented by electrical potentials and the machine contains circuits for producing potentials proportional to sums, products and so on; these potentials can be measured to provide the results.

The precision of an analogue machine is limited by the precision with which the physical quantity used can be measured; it is seldom greater than two or three decimal figures. To increase this precision may be very difficult, and certainly expensive. In a digital machine the precision can be increased as much as desired simply by allowing enough digits in the numbers; this is usually quite easy at the time the design of the machine is being laid down. Most digital computers use numbers having from 8 to 12 decimal digits; this may seem over-generous since the raw data of a problem may be given to only three digits and this may be enough in the results. But it should be remembered that a computer may perform thousands of operations before arriving at these results, and rounding errors may therefore build up alarmingly. Further, the starting numbers and the intermediate quantities and results may vary over a very wide range.

This is not the place for a comparison of analogue and digital machines; it is enough to say that each has its uses. An analogue machine will perform certain restricted operations with great speed and efficiency but it cannot have the range and flexibility of a digital computer.

Probably the most familiar of digital machines is the ordinary desk calculator. This is a mechanical or electro-mechanical device; it is not fully automatic since it has to be individually set up for each arithmetical operation. Some of these machines can divide or evaluate a square root at a single setting but these are automatic only in a very limited sense.

1.2 Automatic digital computers

Automatic digital computers originated in the work of Charles Babbage in the early nineteenth century. Babbage proposed a digital machine capable of doing extended calculations without human intervention. This machine was, of course, to be purely mechanical. Its inventor's ideas seem to have been ahead of the technical possibilities of his day and it was never built. The first automatic digital computer to be made was the Automatic Sequence Controlled Calculator, an electro-mechanical machine which was not finished until 1944. Since this date the situation has been transformed by the speed, flexibility and reliability of electronics.

An automatic digital computer is a digital machine which can perform a large number of arithmetical operations when once set up. Generally such a machine will have a single *arithmetical unit* which is used repeatedly to do different operations on various numbers. This unit is sometimes called the *mill*, a term originated by Babbage. Since there is usually only one mill there must be arrangements for storing numbers, for selecting them and passing them to the mill, and for storing the results produced by the mill (since these may be required again at a later stage).

An automatic digital computer of this kind is performing only one operation at any given moment; after completing one operation it proceeds to do another, and in general it will perform a long sequence of operations with great rapidity. The machine must therefore have a *store* of some sort in which can be placed *orders* or *instructions* (the term *command* is sometimes used); the *control unit* of the machine extracts these orders one by one from its order-store and obeys them. As a rule a single order will cause the computer to carry out a single operation, e.g. adding two numbers together or moving a number

from one part of the number-store to another. In most digital computers the orders are expressed in a numerical code and *they can therefore be stored in the same store as the numbers;* in general a part of this store will hold numbers and another part orders. The store is sometimes referred to as the memory of the machine†.

The store of a digital computer must be capable of "remembering" numbers and orders until they are required. It must be able to give up any item of stored information and to record new information in place of the old. The use of electronics in the mill means that a pair of numbers can be added in considerably less than a millisecond (a thousandth of a second, often abbreviated to millisec). If this speed is not to be wasted the storage devices must also operate at high speed. These requirements have in the past been a major source of difficulty in the design of computers and developments in storage have as a rule lagged behind those in other parts of the computer.

A computer must be able to communicate with the rest of the world. We must be able to "tell" the machine what operations it is to perform, to supply it with the numbers on which to operate, and to extract from it the results of its work. The machine must therefore have *input* and *output* devices, and these must be fast and reliable in operation.

We see, therefore, that there are the following main parts of a digital computer:

(a) a *store* for holding numbers and orders,
(b) a *control unit* which can extract orders from the store, interpret them, and direct the operations of the rest of the machine,
(c) a *mill,* or *arithmetical unit,* which can operate on numbers from the store and send its results back to the store.
(d) *input* equipment (e.g. punched card or paper tape reader),
(e) *output* equipment (e.g. printer or card or tape punch).

These parts and their interconnections are shown in Fig. 1.1. It will be seen that the input and output devices are shown as connected to the mill; this is usually the most convenient arrangement. The connection labelled "discrimination" is described below.

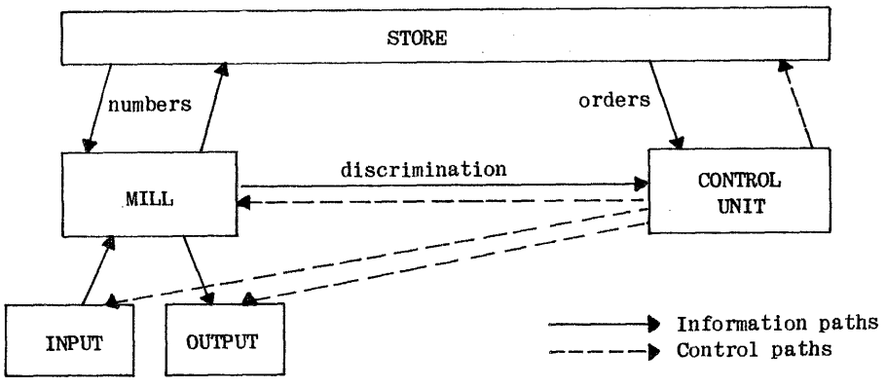


Fig. 1.1 *Block-diagram of a Digital Computer*

1.3 Programming

Let us consider how a digital computer is applied to the solution of a problem. The first essential step is to formulate the whole problem precisely and unambiguously; if the problem is a scientific one this formulation should preferably be in mathematical terms, and an equivalent degree of precision should be aimed at in other kinds of problem. This step is usually the hardest part of the whole process and the one where skill and experience are most important; it may well require considerable time and study. We shall not discuss this subject further here since it belongs properly to the field of study which gave rise to the problem.

The next step is the examination of this precise formulation of the problem in order to find the method of solution best adapted to the available computing tools. This step may, in scientific problems, require a knowledge of *Numerical Analysis;* and it may in any case be combined with the previous step. This subject also is outside the scope of this book, but there is one point which must be made in this connection. A digital computer can, at least in principle, undertake any task which can be expressed precisely as a finite sequence of finite arithmetical operations. It must be emphasised that it is *only* problems of this sort which can be handled by a computer; but this is not such a severe restriction as might appear at first sight and there are many non-numerical problems which can be reduced to arithmetic; for example, much work is now being done on the automatic translation of languages by computers. The practical limits on the range of problems which can be dealt with are largely imposed by the storage capacity and speed of the computer and its ancilliary equipment.

We shall assume, therefore, that we are presented with a definite numerical process, which we must express in a form assimilable by the computer. That is, we have to *programme*†† the problem, or prepare the programme of orders for the computer. In some circumstances it may be useful to subdivide this process into two stages. In the first stage, programming in a narrower sense of the word, the numerical process is converted into a *flow-diagram,* or *flow-chart,* showing what steps the machine is to take and how one operation leads to another. In the second stage, which is often called *coding,* the actual orders which the computer will have to obey are written down in some convenient code; this is relatively simple, once the flow-diagram has been prepared. The exact way in which programming and coding are distinguished (if at all) depends very much on individual preference and no inviolable rules can be laid down - except that great care is needed at all stages. The next step is to punch out the programme, order by order, on paper tape or cards and to feed the result into the computer, along

† Such anthropomorphisms are perhaps to be deprecated, but they sometimes provide the simplest way of describing the operations of the computer.

†† The spelling *program* is sometimes used.

with the numerical data (though these may be fed in as a separate operation). At this stage the orders and numbers appearing on the tape or cards are simply read by the computer, converted into its own internal code, and placed in the store. When this input process is complete the store of the machine will contain the whole of the programme and some or all of the numerical data; at this point the computer starts to select and obey the orders of the programme one by one. Some of these orders will ultimately cause the machine to print or punch the results of the calculation.

It will be clear that the preparation of the programme for a particular calculation may well be time-consuming. When it has been written the programme will have to be checked and got working on the computer; this stage is usually called the *development* of the programme. When the preparation and development are complete the programme may be used repeatedly with different numerical data. This means that it may be uneconomic to write a programme for a calculation which has to be done once only; but if a calculation has to be done many times, perhaps at regular intervals, the cost of preparing the programme can be distributed and it will usually be found that a digital computer offers by far the cheapest method of doing the work.

The orders which the computer obeys in the course of executing a programme are selected from the set of available types of order built into the machine. This set of orders is called the *order-code* of the computer; it is important for the programmer that a comprehensive set of orders should be available, and that the exact effects of each of them should be known to him. It is very helpful if the orders are systematically arranged and are free from objectionable exceptions and omissions. Many of the orders in the order-code of any computer are concerned with simple arithmetical operations; but there are others (for example, those for transferring blocks of numbers from one part of the store to another) which are needed only because the machine is automatic.

The orders in many computers are normally obeyed sequentially; that is to say, they are extracted one after another from adjacent places in the store and are obeyed. Certain orders, called *jump orders*, may break this regular sequence and cause the machine to start selecting its orders from some other specified place in the store; whether this jump occurs may be conditional on, for example, the sign of some numbers. In some computers the jump orders are called *control-transfer* orders since they can be said to *transfer control* to some other part of the programme. Other names are *test* or *discrimination* orders. All automatic digital computers have jump orders of some kind and they add enormously to the flexibility of the machine. When a conditional jump order is obeyed certain information is passed from the mill into the control unit; this information is used to determine whether or not the jump occurs (the information passes along the path labelled "discrimination" in Fig. 1.1).

Nearly all calculations are, at some stage or other, highly repetitive and the programmer can take advantage of this by writing groups of orders and arranging (with the aid of jump orders) that the machine obeys the orders in each group several times. This is of great importance, as will shortly become evident.

A book devoted to programming loses realism unless it is related to a specific machine. In this book, the operations are described in terms of the Ferranti Pegasus Computer. They are, however, readily adaptable to other digital computers and the reader who masters the techniques described in the pages which follow can approach any modern computer with confidence.

1.4 Comparison with desk calculators

An analogy with other methods of computing may be useful here. Let us suppose we have a desk calculating machine equipped with an unintelligent, though extremely reliable, operator and we have some particular numerical problem to be solved. A suitable numerical process for arriving at the solution must be found and this must be written down as a series of simple operations and given to the operator as a precise sequence of orders or instructions. Provided this work has been done correctly and the operator follows slavishly the instructions he has been given the solution of the problem will eventually be obtained. It should be noted that the operator need not understand the reasoning which led to his instruction list nor the significance of his operations. Furthermore the same set of instructions can be used again on another occasion to solve a similar problem involving different numerical data. It is usually advisable to include a few checks in the calculations, and this can be done by giving extra instructions to the operator.

The procedure for solving a problem with the aid of an automatic digital computer is similar; the operator is replaced by the control unit of the computer, his desk calculating machine becomes the mill, and his list of instructions becomes the programme of orders for the machine to obey. The work-sheet on which the operator writes his intermediate numbers can be likened to the store of the computer.

It will be seen that the whole process is in principle very much the same. The main practical points of difference arise from the fact that most desk calculator operators have a fair amount of intelligence (they will not, for example, attempt to divide a number by zero as some automatic machines may). A digital computer may well be 10,000 times as fast as the operator with his desk machine so that the difference of speed is of great importance. There are two main points of consequence resulting from this enormous speed ratio; first, problems can now be tackled efficiently which have hitherto been beyond the reach of computation or have been otherwise uneconomic; and second, a numerical process might well be chosen which would not be used on a desk calculator. In fact the digital computer is so much faster than a desk calculator as to make a qualitative, rather than quantitative, difference.

1.5 Flow-diagrams

We shall now illustrate some of the points mentioned above with the aid of flow-diagrams. In these diagrams each "box" represents a simple operation or a group of such operations.

(a) Let us first suppose we have in the store of the machine a list of whole numbers, each of which may have any value from 1 to 100, except that the last number in the list is known to be zero. Suppose that the computer is equipped with a printer as an output device and we wish to use this to print all those numbers in the list which are not less than 50. The flow-diagram of Fig. 1.2 shows a possible sequence of operations.

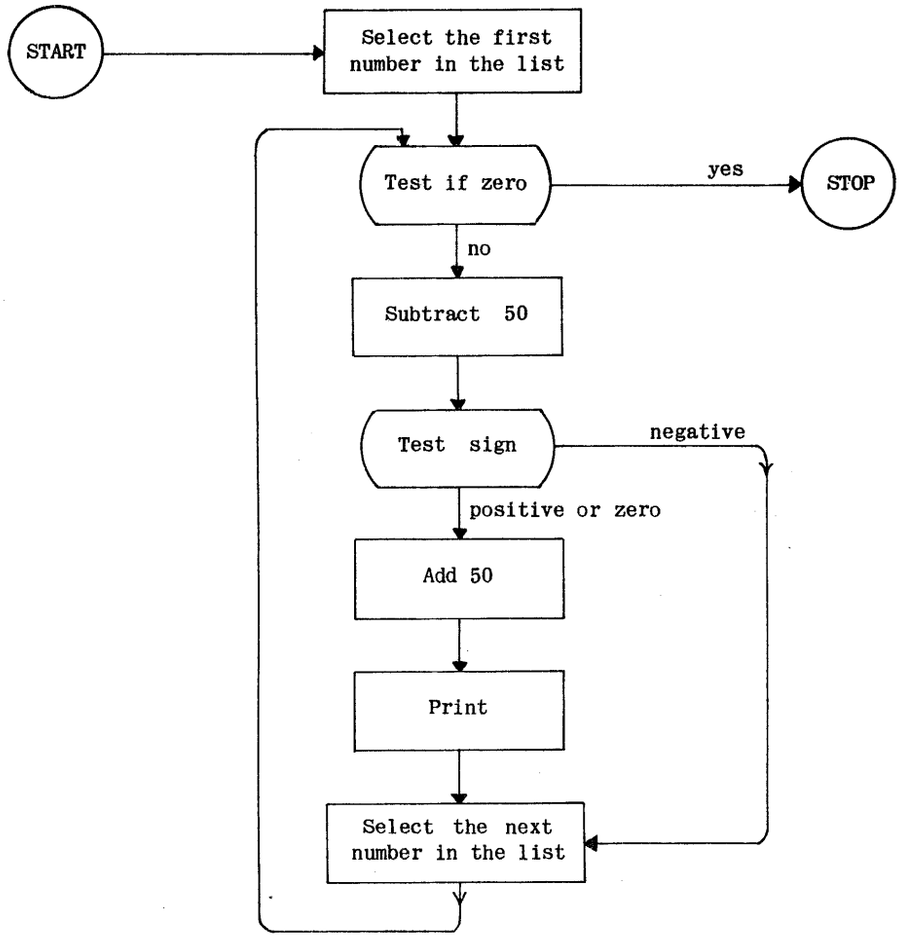


Fig. 1.2 *Flow-diagram of a programme to print those numbers in a list which are not less than 50.*

It will be noted that the process consists simply of exa ining each number in turn; if the number is zero we know we have reached the end of the list, and if it is not we arrange to avoid the "print" operation if the number selected is less than 50. The importance of the *test* or *conditional jump* operations is clear.

(b) In the next example, we shall assume that an input device is provided which can be used to read numbers into the computer one at a time. These numbers might, for example, be punched on paper tape or cards. Suppose that there are 1000 positive numbers waiting to be read and we wish to print the largest of them. It is convenient to use an algebraic notation; the letter x denotes one of the numbers read in from the input device and y stands for the largest number so far read. We read in each number in turn and compare it with y; if it is less than y or equal to it we simply pass on to the next number. If, however, the number is greater than y we must increase the value of y to the new number before reading in the next one. We also have to arrange to count the numbers as they are read in (a quantity c is used for this) so that we know when we have finished, at which point y is to be printed since it is now the largest of all the numbers. The flow-diagram is shown in Figure 1.3.

The technique shown in this flow-diagram for counting should be particularly noted as it occurs frequently. The main part of the above programme consists of a *loop* or *cycle* of orders which the computer is to obey exactly 1000 times; after each repetition a *counter* (in this case c) is reduced by unity until, after the orders of the cycle have been obeyed 1000 times, it is reduced to zero and the computer passes on to the next part of the programme. The details of this counting process depend very much on the facilities available in a particular computer; it may, for example, be more convenient on some machines to start with a negative counter and to *increase* it by unity at each repetition until it is no longer negative.

(c) As a further example let us find the smallest prime factor of a positive whole number N, i.e. the least number (other than 1) which divides exactly into N. We shall assume that N is greater than 1. A process which may be used is first to test if N is even, in which case the answer is 2; if N is not even we try dividing it by 3,5,7,9,..... (i.e. by consecutive odd numbers) until we find a divisor. Strictly speaking we need try only prime divisors but it is simpler to include composite numbers, such as 9, than to omit them, even though we know when we reach one that it cannot divide N exactly (for if N is not divisible by 3 then it is not divisible by 9). If N is a prime the first divisor we shall find will be N itself, but there is no need to go as far as this. As the trial divisor (d, say) steadily increases, the quotient (q, say) will steadily decrease; in the absence of a successful trial d will ultimately exceed q. Note that if d is a factor of N then so also is q, so that each trial with d is in effect also a trial with q. It is consequently pointless to go beyond the stage at which d becomes equal to or greater than q, for any quotient obtained after this must be one of the numbers already tried unsuccessfully as a divisor (or else the quotient is even, and we know N cannot have an even divisor). We need therefore a cycle of orders which tests whether our trial divisor d divides N; if it does not we increase d by 2 and re-enter the cycle provided d is less than the quotient q. The flow-diagram of Fig. 1.4 *(page 6)* shows the process. Note that it will in fact work even if $N = 1$.

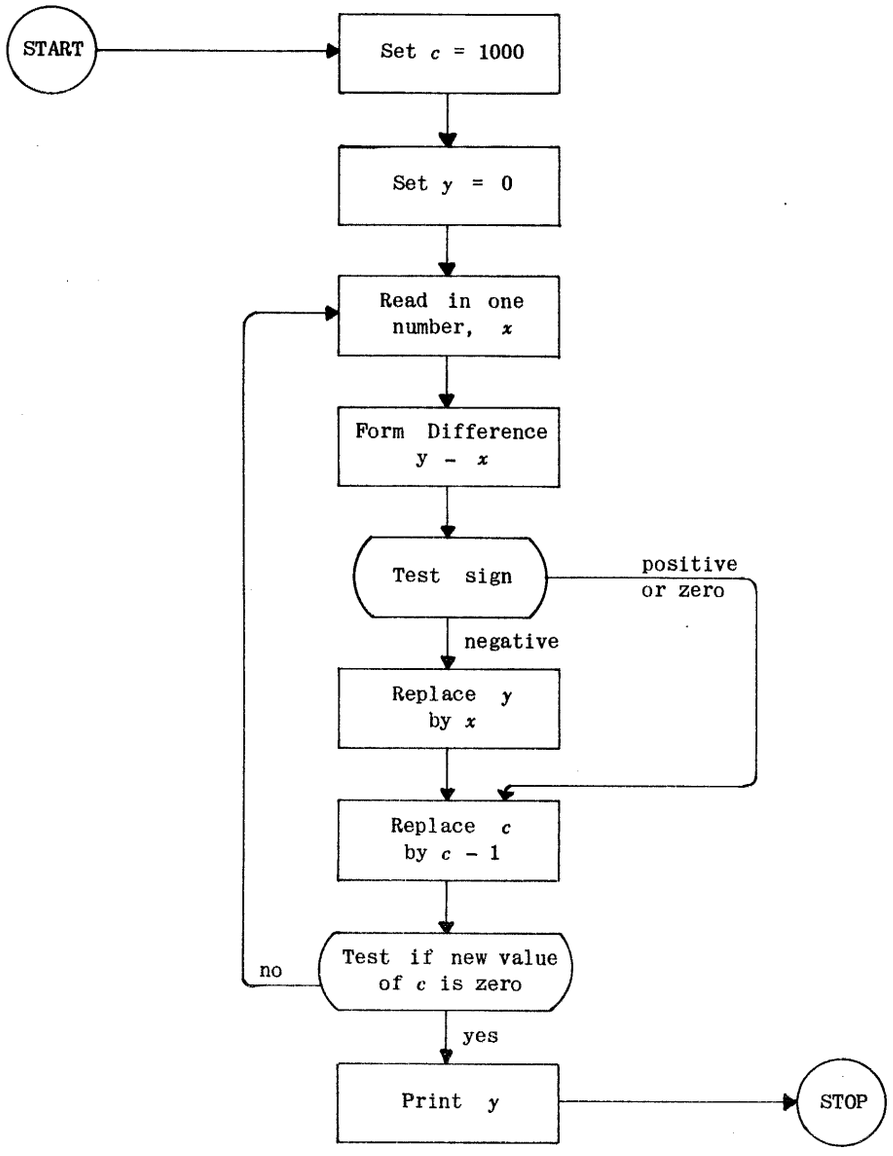


Fig. 1.3 *Flow-diagram of a programme to print the largest of 1000 positive numbers read in one by one.*

(d) Another illustration is provided by a programme to evaluate the square root of a positive number a. We shall use Newton's process; this is based on the fact that if x_1 is an approximation to $\sqrt{a}$, then

$$x_2 = \tfrac{1}{2}\left(x_1 + \frac{a}{x_1}\right)$$

is a better approximation. This calculation can be repeated with x_2 in place of x_1 to yield a further approximation x_3, and so on. The successive approximations are in general connected by the relation

$$x_{n+1} = \tfrac{1}{2}\left(x_n + \frac{a}{x_n}\right) .$$

If we are fortunate and if a is a perfect square we may ultimately arrive at the exact value $x_n = \sqrt{a}$ and the process may be said to have terminated. But as a general rule this will not happen and we shall have to be content with an approximation; for example we could stop whenever two successive approximations differ by not more than some small preassigned quantity h.

As an illustration let us obtain an approximation to $\sqrt{3}$. We may take $x_1 = 1$ as a first guess; then

$$x_2 = \tfrac{1}{2}\left(x_1 + \frac{a}{x_1}\right) = \tfrac{1}{2}\left(1 + \tfrac{3}{1}\right) = 2,$$

and

$$x_3 = \tfrac{1}{2}\left(2 + \tfrac{3}{2}\right) = \tfrac{7}{4} = 1.75,$$

and

$$x_4 = \tfrac{1}{2}\left(\tfrac{7}{4} + \tfrac{3}{7/4}\right) = \tfrac{97}{56} = 1.7321428\ldots$$

The correct value to 6 decimal places is 1.732051. It will be seen that this process converges very rapidly - it is in fact an example of what is known as a "second-order" iterative process, in which the number of significant figures is approximately doubled at each step.

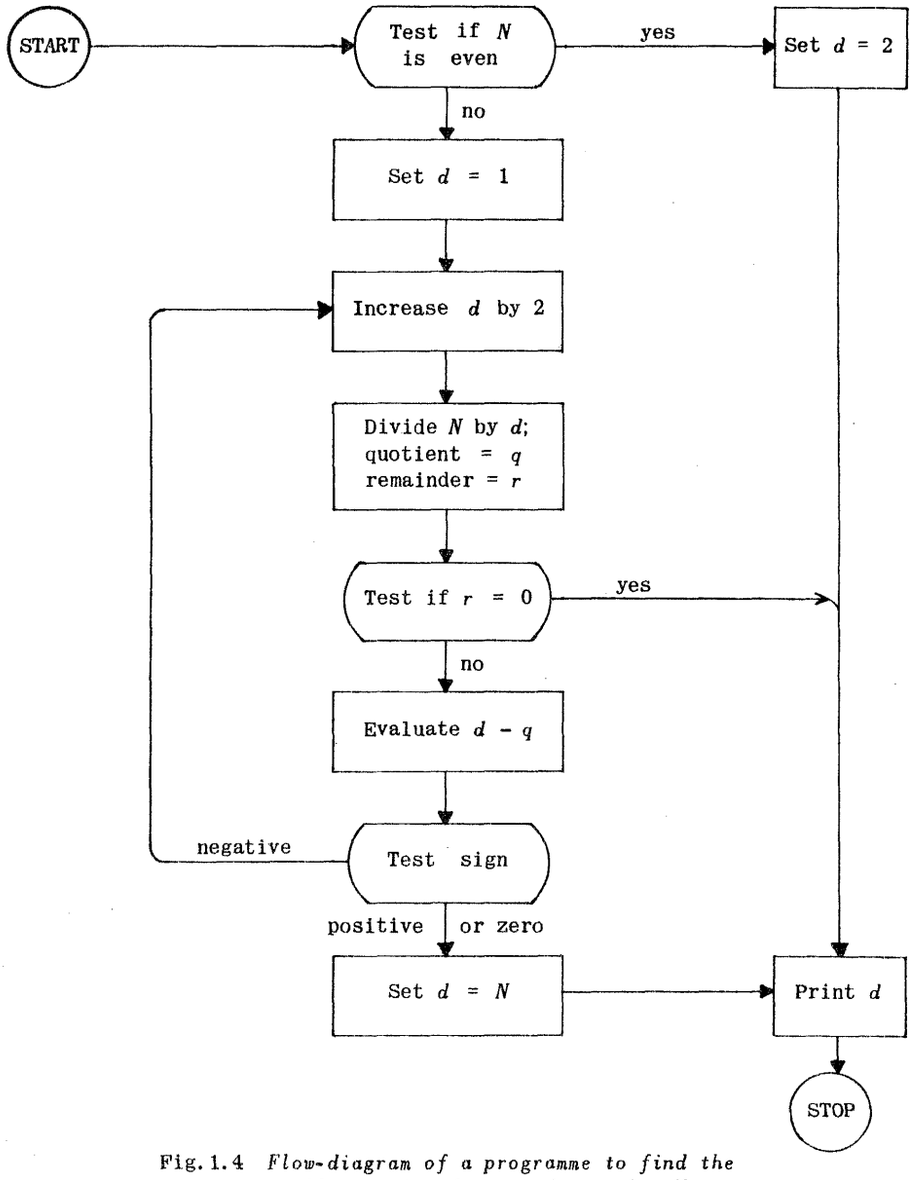


Fig. 1.4 *Flow-diagram of a programme to find the smallest prime factor of a number N*

Returning to the general process for $\sqrt{a}$, the difference between two successive approximations is

$$x_{n+1} - x_n = \tfrac{1}{2}\left(x_n + \frac{a}{x_n}\right) - x_n = \tfrac{1}{2}\left(\frac{a}{x_n} - x_n\right) = y_n \quad \text{say.}$$

We can therefore readily compute this quantity from the value of x_n and examine it. If it is greater than h in absolute value we simply add it to x_n to get x_{n+1} and then repeat the process; if it is less than h (or equal to it) we have finished.

Any of these flow-diagrams forms a suitable basis for a programme or a part of a programme. It will be remarked how each process has been broken down into very simple steps and how each step leads unambiguously to the next. Programming has been described as "explaining the problem to the computer in words of one micro-syllable".

The notation used for these flow-diagrams has been chosen to be self-explanatory. If the preparation of flow-diagrams is to be undertaken systematically as a preliminary to the programming or coding of a process it is probably advisable to use a more rigid system to exclude possible ambiguities.

1.6 Numbers

The ordinary way of writing numbers may be called a decimal (or denary) system, since it is based on the number 10, which is called the *radix* of the system. For example the number written 5428 is an integer (or whole number) whose value is

$$5428 = (5 \times 10^3) + (4 \times 10^2) + (2 \times 10^1) + 8 ,$$

which may alternatively be written

$$\big(\{[(5 \times 10) + 4] \times 10\} + 2\big) \times 10 + 8.$$

In this number 5 is the *most-significant* (or left-most) digit and 8 is the *least-significant* (or right-most) digit. The contribution made by each digit to the value of the number is just the value of the digit multiplied by a power of 10 determined by the position of the digit in the written form of the

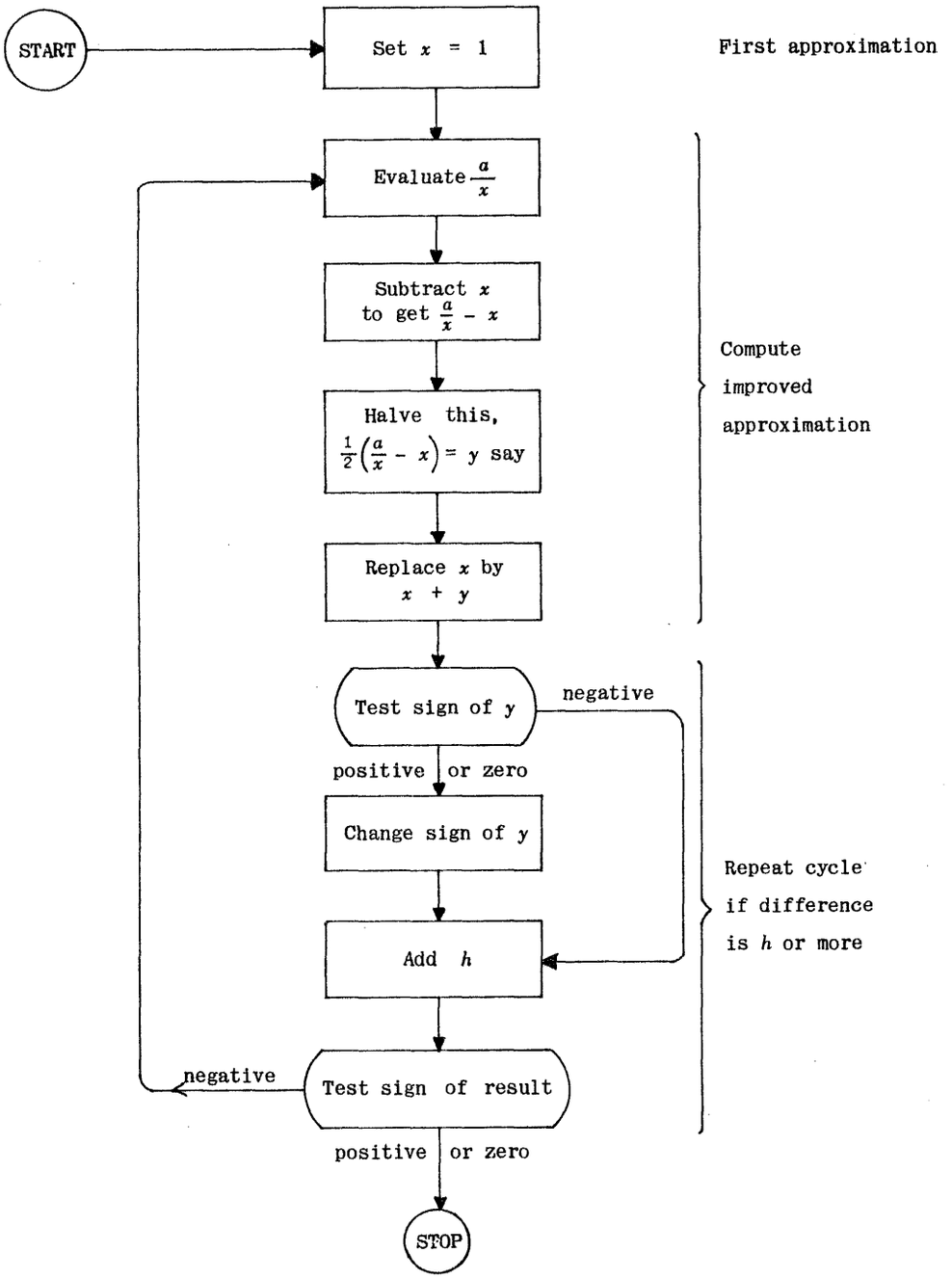


Fig. 1.5 *A process for evaluating $\sqrt{a}$; the result is x.*

number. Fractions and mixed numbers may be expressed by writing digits to the right of a *decimal point*, for example

$$27.93 = (2 \times 10) + 7 + (9 \times \tfrac{1}{10}) + (3 \times \tfrac{1}{10^2}),$$

which may be written†

$$(2 \times 10^1) + (7 \times 10^0) + (9 \times 10^{-1}) + (3 \times 10^{-2}).$$

The decimal point is not usually written in integers, it is understood to lie to the right (e.g. 5428 could be written 5428. or 5428.0).

This is not of course the only way of representing numbers. Sums of money in sterling and periods of time in hours, minutes and seconds are examples of *mixed-radix* systems; for example £461.16.8 represents

$$\{(4 \times 10^2 + 6 \times 10 + 1) \times 20 + (1 \times 10 + 6)\} \times 12 + 8 \text{ pence},$$

and 3 hours 24 minutes 52 seconds represents

$$\{(3 \times 60) + (2 \times 10 + 4)\} \times 60 + (5 \times 10 + 2) \text{ seconds}.$$

† The negative exponent in expressions such as 10^{-3} and 2^{-4} simply means that we have to take the reciprocal. For example 10^{-3} is simply another way of writing $1/10^3$, and $2^{-4} = 1/2^4 = 1/16$.

Various combinations of such radices as 5, 12, 14, 16, 20, 60 are in general use for weights and measures in many parts of the world; such systems are widely understood despite their complexity.

A simple system which finds some application in computers is the *octal* system. This is similar to the ordinary decimal system but is based on the radix 8 instead of 10; thus the number written 2736 in the octal system has the value

$$(2 \times 8^3) + (7 \times 8^2) + (3 \times 8) + 6,$$

$$= (2 \times 512) + (7 \times 64) + (3 \times 8) + 6,$$

$$= 1024 + 448 + 24 + 6,$$

$$= 1502$$

in the usual decimal system. In this system only 8 different digits are needed (0,1,2...,7) instead of the usual 10. Octal fractions may be written by introducing an octal point, for example 3.5 means $3\frac{5}{8}$. This is clearly much simpler than some of the mixed-radix systems with which we are familiar.

The simplest system of all, and one which is used in most digital computers (including Pegasus) is the *binary* system, which is based on the radix 2. This has the great advantage of needing only two different digits (0 and 1) for writing any number. The number written 1101 in binary has the value

$$(1 \times 2^3) + (1 \times 2^2) + (0 \times 2) + 1,$$

$$= 8 + 4 + 0 + 1,$$

$$= 13,$$

in the decimal system. We may write fractions and mixed numbers if we introduce a *binary point*, for example

$$10.101 = (1 \times 2) + 0 + (1 \times 2^{-1}) + (0 \times 2^{-2}) + (1 \times 2^{-3}),$$

$$= (1 \times 2) + 0 + (1 \times \tfrac{1}{2}) + (0 \times \tfrac{1}{4}) + (1 \times \tfrac{1}{8}),$$

$$= 2 + \tfrac{1}{2} + \tfrac{1}{8} = 2\tfrac{5}{8}.$$

It might appear at first sight that any departure from the usual decimal system would introduce many grave disadvantages to the programmer and user of the computer. This is not so however; in fact one need seldom be conscious that the machine is operating in binary. This is because all the numbers fed into the machine and all those which come out of it are in decimal (or in some other convenient system). All the necessary conversions to and from the binary system are merely arithmetical operations, which the computer itself does efficiently. When considering the internal operations of the computer it is usually enough to think of them as being carried out simply on *numbers;* one need not usually consider the way in which these numbers are represented inside the machine.

Table 1.1 shows several numbers in binary together with their decimal equivalents. It will be noticed that integers (whole numbers) have more digits when represented in binary than in decimal; as a rough rule 10 binary digits are about equivalent to 3 decimal digits†. The word *bit* is often used as an abbreviation of binary digit.

It is, of course, essential for the computer to be able to use negative numbers as well as positive ones. The ways in which negative numbers are usually represented will be described later.

It should be noted that the octal system described above is closely related to the binary system. In fact, since $8 = 2^3$, a single octal digit is exactly equivalent to three binary digits. An octal number can be converted to binary by simply writing down the binary equivalent of each of its digits, thus 473 in octal is 100 111 011 in binary. Conversely, the binary number

1011.101

is 13.5 in octal (or $11\frac{5}{8}$ in decimal). Note that one cannot use such a simple process for converting from binary to decimal, or vice versa. It is sometimes more convenient to write binary numbers in octal form since the conversion is easy and the result is more compact and more easily remembered.

While it is as well for the programmer of a binary computer to be familiar with the elements of binary arithmetic, fluency in handling binary numbers is not at all necessary. One point worth noting is that if we move the binary point one place to the right in a binary number we shall effectively double the value of the number, e.g.

$$10.101 \text{ is } 2\tfrac{5}{8}$$

and

$$101.01 \text{ is } 5\tfrac{1}{4}.$$

In the decimal system such an operation multiplies the number by 10 of course. In the same way if we move the binary point one place to the left we shall halve the value of the number.

1.7 Words

The numbers normally handled in a computing machine contain a certain fixed number of digits. It is convenient to use the term *word* for such a number. A word is a group of digits normally handled together by the computer. Words may be used to represent either *numbers* or *orders*, and this is the chief reason for introducing the term. In most computers the words representing numbers and those

† This is because $2^{10} = 1024 \simeq 10^3$; a more exact ratio is $\log_2 10$, which is about 3.3.

Binary	Decimal	Binary	Decimal
1 or 1.0	1	0.1	$0.5 = \frac{1}{2}$
10 or 10.00	2	0.01	$0.25 = \frac{1}{4}$
11	3	0.11	$0.75 = \frac{3}{4}$
100	4	0.001	$0.125 = \frac{1}{8}$
101	5	0.101	$0.625 = \frac{5}{8}$
110	6	0.0101	$0.3125 = \frac{5}{16}$
111	7	1.01	$1.25 = 1\frac{1}{4}$
1000	8	1.1	$1.5 = 1\frac{1}{2}$
1001	9	10.1	$2.5 = 2\frac{1}{2}$
1010	10	110.1	$6.5 = 6\frac{1}{2}$
1011	11	101.001	$5.125 = 5\frac{1}{8}$
10001	17		
11001	25		
100001	33		
1100100	100		

Table 1.1 *Some binary numbers with their decimal equivalents*

representing orders are indistinguishable in appearance, each is merely a string of digits. Words of these two kinds are usually stored in the same store, but the programmer often allocates certain parts of the store to numbers and other parts to orders.

In binary computers, words of 32 to 40 or more binary digits are generally used; the word-length is usually fixed by the construction of the machine. Pegasus has a word-length of 39 binary digits, which is equivalent to rather more than 11 decimal digits. In most such computers a word is represented by a train of pulses of electrical potential; the presence of a pulse indicates a "1" digit and its absence a "0" digit. The pulses representing a word follow one another at an interval called the *digit-time;* in Pegasus this is 3 microseconds†. Digit-times of 1 to 30 microsec are in general use. The digit-time is fundamental to the design of the computer; it is derived from a *clock* waveform, which has a frequency of about 333000 cycles per second (i.e. 333 kilocycles per second) in Pegasus. The operation of the whole computer is synchronised by the clock waveform. Computers of this kind are called *serial* machines, since the pulses representing a word may be sent serially along a single channel. This is in contrast to *parallel* machines in which, for example, 39 channels would be used to transmit simultaneously all the pulses of a 39-bit word. It is obvious that a parallel machine will as a rule be faster than a serial one and will contain very much more equipment. Some computers strike a compromise by operating in a mixed mode, known as *series-parallel,* in which, for example, one might use three channels, each carrying a train of 13 pulses. All small and medium-sized computers operate in the serial or series-parallel modes; some large machines work in the parallel mode. Pegasus is a medium-sized serial computer.

In a serial computer the time needed for the transfer of a word from one part of the machine to another is called a *word-time* (the term *beat* is also used). In Pegasus a word-time is equal to 42 digit-times (i.e. 126 microsec) since there are three unused *gap digits* between the end of one 39-bit word and the start of the next. The duration of any operation in the computer is always an integral number of word-times. When finding out how long some particular programme is going to take on the computer, we usually count the number of word-times and finally convert the total into seconds or minutes.

1.8 Words representing numbers

A word may be used to represent a number, in fact this is one of the major uses for words. If the digits of the word are written out side by side in the usual way (with the most-significant digit on the left) we shall get an integer (or whole number). This is one way of interpreting the word, or assigning a value to it, and we shall call it the *integer convention.* There is an implied binary point just to the right of the least-significant digit.

In Pegasus, as in most binary computers, the left-hand (or most-significant)†† digit in a number-word is used to indicate the sign of the number; it is called the *sign-digit,* (or sign-bit) and is 0 in a positive number (or zero) and 1 in a negative number. We shall explain shortly how negative numbers

† A microsecond is a millionth of a second; it is usually written microsec or, sometimes, μsec.

†† The abbreviations ms and ls will often be used with the meanings most (or more) significant and least (or less) significant, respectively. When we write words out it is always understood that the ms digit is on the left, as is customary when writing ordinary decimal numbers.

are represented by words. The digits of a word are numbered from left to right for reference purposes; the sign-digit is digit 0, the digit to the right of it is digit 1, and so on. The last or ls digit is digit 38. We shall often write *D* in front of the digit-number so that, for example, *D*12 means digit 12.

Instead of using the integer convention we could insert a point (a binary point in Pegasus of course) between two specified digits of the word. This point is *not* represented in the word and its position is largely a matter of convention. In Pegasus the binary point is normally placed between *D*0 and *D*1, i.e. immediately to the right of the sign-digit, so that a typical 39-bit word may be written

0.10001 10111 11001 00010 10000 01001 10001 001

which is a binary fraction. This way of assigning a value to the word will be called the *fractional convention.*

The word

0.00000 00000 00000 00000 00000 00000 00000 000

represents the number zero on either the fractional or the integer convention. The smallest positive number is represented by the word

0.00000 00000 00000 00000 00000 00000 00000 001

whose value is 1 on the integer convention, and $1/2^{38}$ (or 2^{-38}) on the fractional convention (this is about 0.00000 00000 036 in decimal). The largest positive number is represented by the word

0.11111 11111 11111 11111 11111 11111 11111 111

On the fractional convention this has a value just less than unity, in fact $1-2^{-38}$ (0.99999 99999 964 approximately in decimal). On the integer convention it can be shown that its value is $2^{38}-1$ = 27 48779 06943, i.e. rather more than 250 000 million. Thus, a word representing a non-negative number† can take values from 0 to 27 48779 06943 on the integer convention, or from 0 to $1-2^{-38}$ in steps of 2^{-38} on the fractional convention. These numbers are therefore expressed by the equivalent of rather more than 11 decimal digits.

†† ▼ The value of a non-negative word can be found by adding up contributions from each of its digits which are 1's. On the fractional convention digit *D*1 contributes ½ if it is a 1; *D*2 contributes ¼, and so on; in general digit *k* contributes 2^{-k} to the numerical value of the word if it is a 1. On the integer convention each digit contributes 2^{38} times as much. ▲

Let us now consider negative numbers. If we use a desk calculator which handles numbers of 10 decimal digits and we subtract 48 from 0 we shall get the result 99999 99952. This may be regarded as an alternative way of writing the negative number -48. If negative numbers are written in this way they may be added and subtracted correctly on the calculator, provided the results are consistently interpreted. We say that negative numbers written in this way are represented *in complementary form.* The complementary form of a negative number may be obtained by subtracting the absolute (unsigned) value (in this case 48) from 10^{10}, which is the number lying just outside the capacity of the calculator. The usual way of writing signed numbers is by means of a sign (+ or -) and the modulus# of the number.

Either of these ways of representing signed numbers may be used in a digital computer; and other systems are occasionally employed. If a computer uses the sign and modulus representation there may be two different ways of representing zero (i.e. as +0 and -0); it is important that these should be treated in the same way.

Pegasus uses the complementary system for negative numbers and we shall now describe this in more detail. The complements are taken with respect to 2^{39}, which plays a role corresponding to 10^{10} in the above example of a decimal desk calculator. Consequently a negative number which is small in absolute value is represented by a word having a string of 1's on the left (corresponding to the string of 9's above). For example, the integer 44 is represented by the word##

0.00000 00000 00000 00000 00000 00000 00101 100

We can find the word representing -44 by subtracting the above word from 0. The result is

1.11111 11111 11111 11111 11111 11111 11010 100

One way of determining the value of a negative number-word is to subtract it from 0 and evaluate the resulting positive word according to the usual rules. The process of subtracting a number from 0 is usually called negating the number, or changing its sign. It should be particularly noted that this is *not* done by simply changing the sign-digit (as it would be if the sign and modulus representation were used). An easy way of changing the sign of a given word on paper is to start at its ls digit (on the right) and, proceeding to the left, to copy all the 0's (if any) until we find a 1; this 1 is also copied. The remaining digits are then reversed, i.e. we write 1 for 0, and 0 for 1. Tables 1.2 and 1.3 illustrate how numbers are represented by words in the two important conventions.

---

† A non-negative number is a number which is either positive or zero. Zero is regarded as being neither positive nor negative.

†† ▼ ▲ The text between these symbols may be omitted at a first reading.

\# The *modulus* of a number is simply its magnitude or absolute value. The modulus of 48 is 48; the modulus of -48 is 48. Moduli are conventionally indicated by vertical lines on each side of the number, thus |48| means "the modulus of 48" and we can write

$$|3| = 3, \quad |1\tfrac{1}{2}| = 1\tfrac{1}{2}, \quad |-2| = 2, \quad |0| = 0.$$

\## We shall adhere to the convention of writing the point in number-words and grouping the digits in the way shown, regardless of the convention used to assign a value to the word.

It will be seen from these tables that the word whose value is 13 according to the integer convention has the value 13 x 2^{-38} according to the fractional convention. The small quantity 2^{-38} occurs frequently and we shall often denote it by ϵ (the Greek letter epsilon)†; thus 13 x 2^{-38} may be more conveniently written 13ϵ. We shall refer to number-words as *integers* or *fractions* according to the convention used to interpret them. Suppose that x_F is the value of some word interpreted as a fraction, and that x_I is the value of the same word as an integer; these are connected by the equations

$$x_I = 2^{38}\, x_F, \qquad x_F = x_I/2^{38} = x_I\epsilon.$$

It is important to realise that x_I and x_F have the same digits in binary, but that their decimal representations would be quite dissimilar.

Fraction	Word representing the fraction
$\frac{1}{2}$	0.10000 00000 00000 00000 00000 00000 00000 000
$\frac{1}{8} = 2^{-3}$	0.00100 00000 00000 00000 00000 00000 00000 000
$-\frac{1}{8}$	1.11100 00000 00000 00000 00000 00000 00000 000
$\epsilon = 2^{-38}$	0.00000 00000 00000 00000 00000 00000 00000 001
$13\epsilon = 13 \times 2^{-38}$	0.00000 00000 00000 00000 00000 00000 00001 101
-13ϵ	1.11111 11111 11111 11111 11111 11111 11110 011
$\frac{3}{4}$	0.11000 00000 00000 00000 00000 00000 00000 000
$-\frac{3}{4}$	1.01000 00000 00000 00000 00000 00000 00000 000
-1.0	1.00000 00000 00000 00000 00000 00000 00000 000

Table 1.2 *39-bit words representing numbers (fractional convention)*

Integer	Word representing the integer
1	0.00000 00000 00000 00000 00000 00000 00000 001
13	0.00000 00000 00000 00000 00000 00000 00001 101
-1	1.11111 11111 11111 11111 11111 11111 11111 111
-13	1.11111 11111 11111 11111 11111 11111 11110 011
$128 = 2^7$	0.00000 00000 00000 00000 00000 00000 10000 000
-128	1.11111 11111 11111 11111 11111 11111 10000 000
$2^{37} = 137438953472$	0.10000 00000 00000 00000 00000 00000 00000 000

Table 1.3 *39-bit words representing numbers (integer convention)*

▼ We can define the numerical value of a word according to the fractional convention by defining the contributions made by its various digits (if they are 1's) as follows:

(a) the sign-digit contributes -1,

(b) digit k contributes 2^{-k} (if k if not zero).

These rules may be expressed in a more compact way if we write d_k = 0 or 1 according as digit k is 0 or 1 ($0 \leqslant k \leqslant 38$). The value of the word on the fractional convention may then be defined as

$$-d_0 + \sum_{k=1}^{38} d_k . 2^{-k}$$

Its value on the integer convention is

$$-d_0 . 2^{38} + \sum_{k=1}^{38} d_k . 2^{38-k}.$$

▲

† For estimating the approximate sizes of numbers we can take $\epsilon \simeq 0.00000\ 00000\ 036$.

A word representing a negative fraction may take values from -1.0 to $-\epsilon$ in steps of ϵ. It follows that a number x can be represented on the fractional convention only if†

$$-1 \leqslant x \leqslant 1 - \epsilon,$$

i.e. x must be numerically less than unity, except that the value $x = -1$ is allowed. A word may represent an integer n provided

$$-2^{38} \leqslant n \leqslant 2^{38} - 1,$$

i.e.

$$-27\ 48779\ 06944 \leqslant n \leqslant 27\ 48779\ 06943.$$

A fraction x can be represented exactly by a word only if it is an integral multiple of $\epsilon = 2^{-38}$; in general we shall have to approximate. Any fraction can be represented with an error of at most $\pm \frac{1}{2}\epsilon$ by correct rounding of the last digit. Table 1.4 shows a few such approximations.

Fraction	Approximate representation by a 39-bit word
$\frac{2}{3}$	0.10101 01010 10101 01010 10101 01010 10101 011
$\frac{1}{3}$	0.01010 10101 01010 10101 01010 10101 01010 101
$\frac{1}{5}$	0.00110 01100 11001 10011 00110 01100 11001 101
$\frac{1}{7}$	0.00100 10010 01001 00100 10010 01001 00100 101
$\frac{1}{10}$	0.00011 00110 01100 11001 10011 00110 01100 110

Table 1.4 *Rounded approximations to fractions*

If the numbers occurring in a calculation are integers or fractions they would normally be represented according to the appropriate convention. Otherwise they must be *scaled* in some way. This scaling must be done in such a way that all the intermediate quantities formed in the machine during the course of the programme are within range; and, if fractions are used, the scaling must be such that accuracy is not lost. This is usually possible; but when it is not, or when it is very difficult to determine the scaling factors, then there are well-established programming techniques, such as floating-point working, which can be used. Occasionally we may want more precision than can be got by the normal representation of numbers, i.e. we may need more than 11 decimal digits: we can then use double-length (or double-precision) arithmetic, in which each number is represented by two words. These and other techniques will be described later.

If we say that a word has the value 0.75 then we obviously mean that the word is to be interpreted on the fractional convention; and if we say its value is 94 we intend the integer convention to be used. As a rule the fractional convention is regarded as the standard one.

The word whose value is -1 according to the fractional convention will usually be written -1.0 to prevent confusion with the integer -1; this word has a 1 in the sign digit position (*D*0) only, and has the value -2^{38} on the integer convention.

Nearly all of this Section applies, with only small changes, to other binary computers using the complementary representation (as most of them do). The fractional range $-1 \leqslant x < 1$ applies to most computers, though other ranges are occasionally used. The integer range depends, of course, on the word-length.

---

† The following useful mathematical symbols will be employed often; they are tabulated here for the convenience of those to whom they may be unfamiliar.

± 3 means $+3$ or -3 (read as "plus or minus 3"),

$x = y$ means x is equal to y,

$x \neq y$ means x is not equal to y,

$x < y$ means x is less than y,

$x \leqslant y$ means x is less than or equal to y (or x does not exceed y),

$x > y$ means x is greater than y (or x exceeds y),

$x \geqslant y$ means x is greater than or equal to y,

$x \simeq y$ or $x \fallingdotseq y$ means x is approximately equal to y.

It should be noted that $x < y$ is equivalent to saying that $x - y$ is negative. For example, the following are all true statements:

$$2 < 3,\ 2 \leqslant 3,\ -2 < -1,\ 6 > 0,\ 0 > -1,\ 5 \geqslant \tfrac{1}{2},\ -\tfrac{3}{4} < -\tfrac{1}{2}.$$

We shall write, for example, $-1 \leqslant x < 1$ to mean that both $-1 \leqslant x$ and $x < 1$. The symbol Σ means *the sum of*, for example,

$$\sum_{k=1}^{38} d_k.2^{-k} = d_1.2^{-1} + d_2.2^{-2} + d_3.2^{-3} + \ldots + d_{38}.2^{-38}.$$

Chapter 2

Pegasus

This Chapter contains a brief description of a Pegasus installation and its main components. This is followed by a description of the store of the computer and the way in which orders are obeyed and written. A discussion of the order-code then leads into an account of the simpler orders.

2.1 A Pegasus installation

In Section 1.2 we described briefly the main parts of a typical digital computer; let us now examine a Pegasus installation. The store in Pegasus is split into two parts; the *main store*, which is large, and the *computing store*, which is a small, fast, working store; these will be described in the next two sections. The control unit, which may be thought of as the "central nervous system" of the computer, obeys orders taken from the computing store; its mode of operation will be discussed further in Section 2.4. When it is obeying orders the control unit selects numbers from one of the stores (generally the computing store) and makes use of the mill to perform arithmetical operations. In Pegasus 2 the computing store, control unit and mill are housed in the main cabinet, on the front of which are the control panels and desk (see Plates 1,2).

There are three control panels; (a) the *monitor* (Plate 12), on which are two cathode ray tubes which may be switched so as to exhibit various words and waveforms; (b) the *programmers' switches* (Plate 11) which are the principal controls for the computer; and (c) the *engineers' switches* under a hinged flap on the desk which are not of great concern to the programmer. The control panels are described in Section 6.7. Above the monitor panel is a clock (timepiece).

On the desk stands the basic input and output equipment comprising two tape-readers (Plate 3) and the output punch (Plate 4). These devices handle punched paper tape, a useful medium for carrying information (see Plate 5). Either of the tape-readers can be operated by the computer itself so as to "read" the tape; this is the way in which programmes, numbers and other data are supplied to the computer. The input tape which is to be read by the computer may be prepared manually by using the *tape-editing equipment*, which includes a teleprinter with a keyboard and perforating device (Plates 7, 8, 9). The computer punches the results of the calculation into paper tape by using the output punch. The output tape so obtained is usually printed out almost immediately by an *interpreter* (Plate 4), which reads the holes in the tape and prints the corresponding characters on a roll of paper. The output tape may, if desired, be used as an input tape to the computer on another occasion.

Among the programmers' switches the most important controls are the *Start key* and the *Run key*. The Start key is used mainly to cause the computer to read a programme tape, i.e. to read in and store the orders making up a programme. The Run key can be used to stop the computer at any time or to allow it to carry on.

We shall be concerned for most of this book with the basic Pegasus 2 computer, which has just been briefly described, and certain *ancillary equipment*. We cannot provide here a full description of every item of ancillary equipment as the range of available equipment is continually being extended. A Pegasus installation may include some or all of the ancillary equipment according to the nature of the tasks it is required to perform. The following are items of ancillary equipment to which we shall refer; further details are given in Chapters 10 and 11.

(a) Magnetic tape auxiliary storage.
(b) Punched card input and output equipment.
(c) A converter for linking punched cards, magnetic tape and a high-speed printer.
(d) Multiple output punches.

Although we refer specifically to Pegasus 2, much of the text which follows is relevant to both the original Pegasus 1 with the small drum and the present Pegasus 1 with the larger drum. It will, however, be clear from the footnotes how these versions of Pegasus differ from the Pegasus 2.

2.2 The Main store

The first 128 words of the main store of Pegasus 2 are held on delay-lines, and the rest of the main store is a magnetic drum revolving at about 3720 revolutions per minute†. We shall first of all discuss the drum storage.

One revolution of the drum takes exactly 128 word-times, or roughly 16 milliseconds.†† The surface of the drum is coated with a magnetisable iron oxide, and just clear of it are a number of fixed *heads* containing coils. Parts of the drum surface may be magnetised by passing pulses through the coils in the heads: we can therefore record a word magnetically (or *write* the word) by passing the corresponding train of pulses through one of the heads. The resulting small areas of magnetisation will occupy a 128-th part of the *track* swept out by the head. By virtue of a property of the iron oxide, this magnetisation will remain until the next time this particular head is used to write on to the same part of its track. The heads used for writing may also be used for *reading* the recorded

† In Pegasus 1, the magnetic drum constitutes the whole of the main store.

†† A millisecond is a thousandth of a second; it is usually abbreviated to millisec or msec. Clearly 1 millisec = 1000 microsec.

words, since the small magnetised areas will induce pulses in the heads as they are carried past by the rotation of the drum. The reading process may be repeated as often as desired, since it does not disturb the magnetisation of the drum.

Information recorded on the drum is, for engineering reasons, split up into channels of odd and even digits which are recorded separately, thus, 2 heads are needed for each track of information. There are 71 head pairs and each sweeps out a track on which 128 words may be recorded; thus, with 128 words on delay-lines, the total capacity of the main store is 9216 words.† Each of the places where a word may be recorded is called a *storage location* or simply a *location* (the term *cell* is sometimes used). In order to write a word into some particular storage location on the drum the computer has to select the appropriate heads and then energise them at the correct time. The word may be read later by selecting the same heads and extracting the pulses from them at the appropriate moment. Any particular storage location is thus identified by specifying the heads or track to be selected and the time (or angular position of the drum). The selection and timing are performed quite automatically by electronic circuits associated with the drum; they are derived from the *address* of the storage location, which can be thought of as a label permanently attached to the location and used to identify it. The address of a storage location may be compared with the address of a house in a city, which can be specified by means of a street name (corresponding to the track) and the number of the house in the street; the analogy is even closer if we imagine the streets to be numbered (as in some American cities) and we suppose there are exactly 128 houses in each street. The word stored in a particular location is called the *content* of the location; it must not be confused with the address of the location, just as one must not confuse the inhabitants of a house with its address.

The storage locations are grouped into blocks, each consisting of 8 locations; there are consequently 16 blocks round each of the 71 tracks and 16 blocks of delay-line storage, making a total of 1152 blocks.

There are two addressing systems used in the main store. In the simpler system the locations are numbered straight through from 0 to 8191; these addresses will be called decimal addresses. In the other system, which is more generally useful, each location is specified by giving its *block-number* and its *position* within the block. The blocks are numbered 0 to 1023. Within each block the eight individual locations are given position-numbers between 0 and 7. Thus we may refer to a certain location as being in block 342 in position 5; this would be written 342.5, which is the block-and-position form of the address (this word would also have a decimal address 2741). The position-number may be thought of as an octal digit.

The block-numbers are sometimes referred to as *block-addresses;* this is an extension of the term "address" since it is here used to identify blocks rather than individual locations. The block-addresses are often prefixed by the letter *B,* thus we may refer to block 342 as *B*342.

It has been arranged that sixteen of the tracks are *isolated,* i.e. the corresponding heads may be used for reading but not writing. These 16 tracks form two isolated stores, each of 128 blocks (1024 locations) and both addressed from 896.0 to 1023.7 (or 7168 to 8191 on the decimal address system) either store may be selected by means of one of the engineers' switches on the control desk under the hinged flap. These isolated blocks are used to store permanently certain useful programmes; the Initial Orders, which are of great value to the programmer, are in one of the stores, and in the other are the engineers' test programmes, When the appropriate switch is in the normal position, the isolated store containing the Initial Orders is selected. The rest of the main store consisting of 896 blocks (or 7168 locations) addressed from 0.0 to 895.7 (or 0 to 7167 on the decimal address system), is available for other programmes.††

It should be noted that information stored on the drum, but not in the delay-lines, is not "volatile", i.e. it remains there even when the computer is switched off at the mains; in particular the programmes in the isolated part of the main store are always there and can be considered almost a part of the machine.

A magnetic drum combines economy with a reasonably large storage capacity; this is why it is used in many computers. It suffers however, from the disadvantage of a relatively long *access-time,* which is the time needed for information to become available. One will usually have to wait until some particular part of the drum is under the heads before one can actually read or write. This time will never exceed one revolution time (16 millisec in Pegasus) and this is therefore called the maximum access-time; the average waiting time will be about half a revolution (8 millisec in Pegasus) and is called the mean access-time, if we neglect the time occupied by the actual reading or writing. On some machines the head selection is done by relatively slow relays and time must be allowed for these to operate when changing from one head to another; on Pegasus this switching operation is done electronically and no time need be allowed for it.

If the speed of a magnetic drum computer is not to be severely limited by the access-time of the drum it is essential to provide as well some other kind of storage having a shorter access-time. This is sometimes called the "fast" store and various other terms are used, such as "working" store or "quick access" store or "high-speed" store; but it is referred to as the *computing store* in Pegasus, for reasons which will become apparent later.

As a consequence of investigations into the amount of waiting time spent in some of the standard programmes run on Pegasus 1 (i.e. the time spent in waiting for the drum to be in the correct position for a word to be read or written), the first 16 blocks of main store in Pegasus 2 are held on 8-word delay-lines.# This has the effect that the transfer of words between the computing store and this part of the main store is carried out almost immediately without having to wait for the drum to come

---

† Pegasus 1 has a drum storage capacity of 9216 words, but the earlier model of Pegasus 1 has a smaller drum which has a storage capacity of 5120 words. In what follows we refer specifically to Pegasus 2.

†† On the small drum version of Pegasus 1, there is one isolated store of 128 blocks: the non-isolated part of the store has a capacity of 512 blocks (4096 words) and is referred to as the *4096-word store.* (The main store on the present Pegasus 1 and Pegasus 2 is referred to as the *7168-word store).* A version of the Initial Orders, which lacks some of the facilities of the 7168-word store Initial Orders, is stored in the first part of the isolated store from *B*512, and the engineers' test programmes are stored beyond this.

A further 16 blocks of main store held on 8-word delay lines may be provided in Pegasus 2 as an optional extra.

into position. These 16 blocks (*B*0 to *B*15) have exactly the same addressing system as they would have if they were held on the drum, so that to the programmer the only difference is the increase of speed. (Further details of the time are given in section 3.10).

2.3 The computing store

The computing store of Pegasus is made up of *registers*, each of which can store one word (its *content*); the registers correspond to the storage locations in the main store.

Most of the registers are made up of circulating magnetostrictive delay-lines and amplifiers. Each delay-line consists of a length of nickel wire with a coil near each end. When a pulse is passed through one of these coils the nickel inside it shrinks momentarily (magnetostrictive effect) and compression waves (i.e. sound waves) travel along the nickel wire in both directions away from the coil. One of these waves is absorbed and the other one travels down the nickel wire to the second coil, in which it induces a small pulse. This small pulse can be amplified and, after having any distortions removed, can be fed back to the first coil; the whole process is then repeated. In this way a pulse can be kept circulating indefinitely, provided the power is switched on to the amplifier. The circulation-time is one word-time, i.e. 126 microsec, so that a whole word of pulses can be kept in circulation; any gaps in the train of pulses will of course persist. At any particular instant the 39 pulses (and "no-pulses") making up a word (together with the three gap pulses) will be strung out along the nickel wire, travelling down it with constant velocity.

The word stored in the delay-line is available at the output of the amplifier; it may be read as many times as desired without disturbing it. In order to replace it by another word we have only to break the circulation loop for exactly one word-time while the new word is being fed into the delay-line. This is shown diagrammatically in Fig. 2.1.

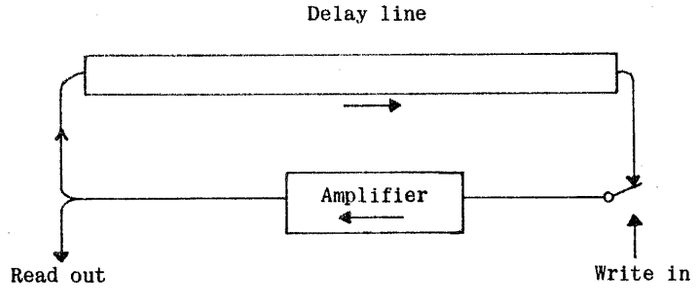


Fig. 2.1 *A delay-line storage unit*

Many computers incorporate delay-line stores of one form or another. In some of these each delay-line holds more than one word (as in the 8-word delay-lines comprising the first 16 blocks of main store in Pegasus 2); this increases the storage capacity at the expense of increasing the access-time. With single-word delay-lines, such as are used in the computing store in Pegasus, each stored word can be regarded as available at any time.

The registers making up the computing store are divided into three groups:

(a) the ordinary registers,
(b) the accumulators,
(c) the special registers.

The *ordinary registers* make up the bulk of the computing store; there are 48 of them, arranged in six blocks, each of eight registers. The addresses of the ordinary registers are written in the same way as the addresses of locations in the main store, e.g. the register in position 3 of block 4 has the address 4.3. If there is any danger of confusion we shall prefix computing store addresses by the letter *U* and main store addresses by the letter *B*; thus *U*5.2 is the address of an ordinary register, and *B*5.2 that of a storage location in the main store. The blocks in the computing store are numbered *U*0 to *U*5 so that the first of the ordinary registers is *U*0.0 and the last is *U*5.7.

The eight registers called *accumulators* bear the addresses 0 to 7; these addresses are often prefixed by the letter *X* (e.g. *X*2). The accumulators have special properties which single them out from the other registers; these will be described later. Accumulator 0 is sometimes called the *dummy accumulator*, its content is always zero and cannot be changed; this accumulator is not made up of a delay-line. Each of the seven other accumulators can be used to store any word required.

The *special registers* have the addresses 15, 16, 17, 24, 32, 33, 34, 35, 36 and 37.† They are not made up of delay-lines and are used for special purposes which will be described later (see Section 2.9).

We shall often need to refer to the content of a register, i.e. to the word held there. We shall write *C*(5.2) for the content of ordinary register 5.2, *C*(6) (or sometimes x_6) for the content of accumulator 6, and so on.

There are facilities for transferring words from the main store to the computing store and vice-versa. These words may be transferred either singly or in blocks of eight words.

Fig. 2.2 *(page 16)* shows the two stores of the basic Pegasus computer; some of the details will be explained later. In this figure the address of each of the registers of the computing store is shown; these addresses are always written in the way indicated; for example 3 means accumulator 3, and 3.0 means ordinary register 3.0. The reader should endeavour to keep this picture of the computing store in his mind while he reads the next few chapters; he should also note particularly that the word *register* applies not only to the ordinary registers but *also to the accumulators* (as well as the special registers).

† In the small drum Pegasus 1, registers 24 and 37 are not present, and register 36 only on those installations with magnetic tape. Both registers 36 and 37, but *not* 24, are included in the large drum Pegasus 1.

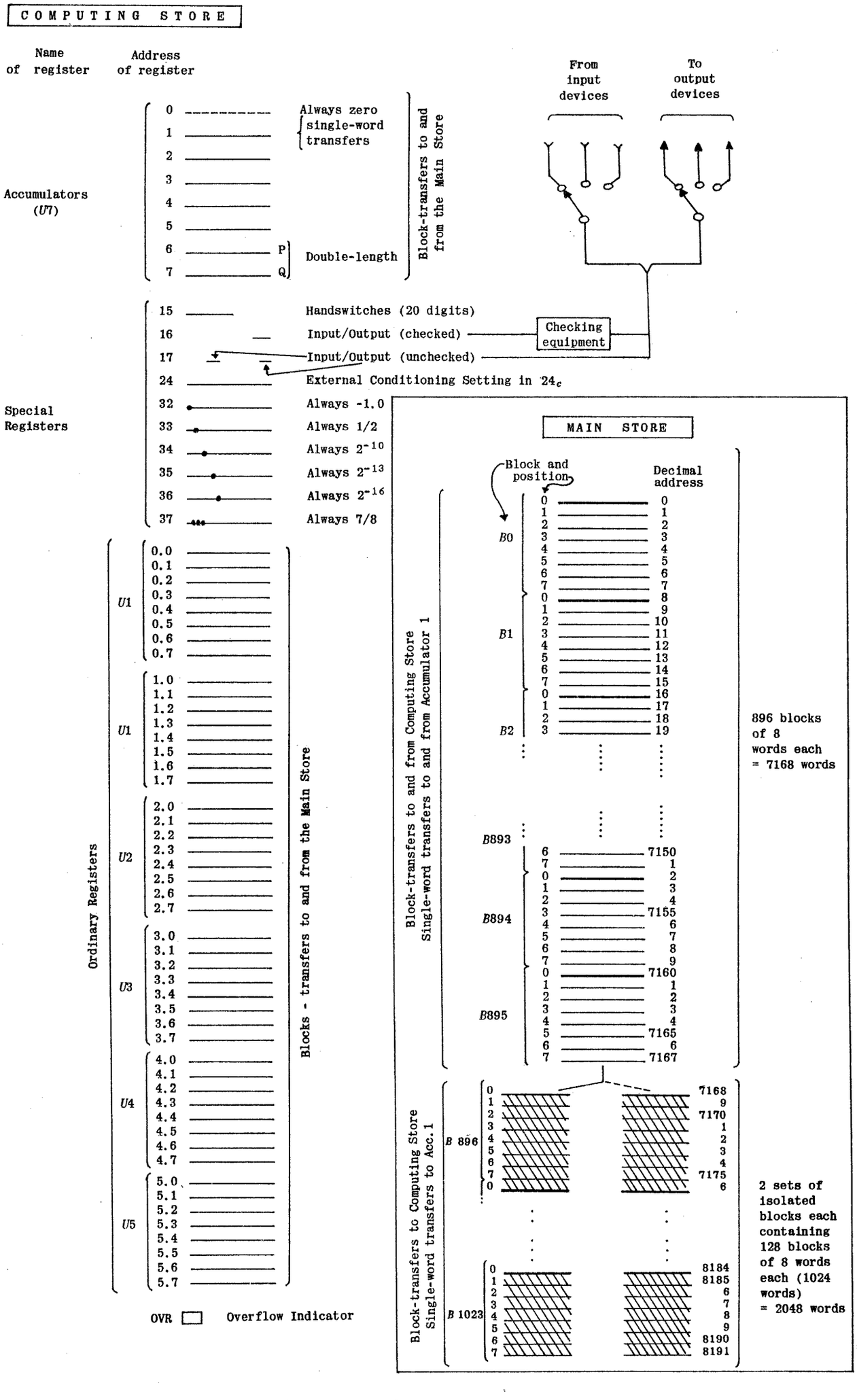


Fig. 2.2 *The Computing Store and Main Store of Pegasus*

2.4 Outline of operation

The orders or instructions which the computer obeys in the course of carrying out a programme are represented in coded form by words; such words are referred to as *order-pairs* in Pegasus, since they each represent two orders.

The control unit of the computer, which controls the whole machine, selects the orders to be obeyed exclusively from the ordinary registers in the computing store. The address of the ordinary register concerned is called the *order-number* and is held in a special *order-number register* (or O.N.R.) in the control unit. There is also an *order-register* (or O.R.) to hold the orders currently being obeyed.

The sequence of operations while the computer is obeying a programme is as follows:

(a) The control unit determines the order-number (from the O.N.R.) and connects the ordinary register specified to the order-register (O.R.). This causes a copy of the specified order-pair to flow into the order-register. This operation requires one word-time.

(b) The first order (*a-order*) of the order-pair is obeyed, i.e. the computer carries out an elementary operation determined by the digits of the order. This operation usually takes two word-times, but some orders (e.g. multiplication) require more than this.

(c) The second order (*b-order*) is obeyed in a similar way, which also takes two word-times as a rule. While this order is being obeyed the order-number in the O.N.R. is increased by unity, so that the next order-pair will be selected from the next ordinary register.

This cycle of operations is normally repeated a number of times. It will be seen that it usually occupies 5 word-times, during which two orders are obeyed; we can say therefore that the average time of obeying one (simple) order is 2½ word-times, or 0.315 millisec. Alternatively we can think of the *a*-order as occupying 3 word-times and the *b*-order 2 word-times; this is often a more useful approach, and it is the one we shall adopt. The orders obeyed are selected from consecutive registers and are therefore said to be obeyed sequentially.

Certain orders, called *jump* orders, may interrupt the regular sequential selection of orders and cause the machine to start obeying orders from some specified register. They do this by changing the order-number in the O.N.R., and there are arrangements for jumping to or from either order of a pair.

It is important to realize that the computer selects its orders exclusively from the ordinary registers in the computing store; there are no arrangements for taking orders directly from the accumulators or from the main store. The usual process is to place the programme in the main store and then to transfer a few blocks of it into the computing store to be obeyed; when these orders have all been used up a further instalment is brought in from the main store, and so on.

Apart from a few orders effecting transfers to and from the main store, all the orders are concerned only with the computing store. All the arithmetical operations and organisational work are carried out in the computing store, and it is here that sections of the programme and numbers currently required are stored. Since there are no problems of access-time in the computing store it follows that the speed of operation of the machine is high. The computer is thus organised on the basis of a *two-level* store; all the work is done in the "working space" of the computing store, the main store being used largely to hold orders and numbers not immediately required.

In this way a high speed of operation is obtained despite the use of a magnetic drum and without resorting to *optimum coding* (this is an alternative system in which orders and numbers are placed in a long delay-line or round a drum in such a way as to be available when required). If the programmer has to use optimum coding then he must consider timing matters while he is writing the programme. Some computers are provided with a *single-level* store, i.e. there is only one store; this is much simpler from the programmer's point of view, but in most such computers the size of the store is limited because of its high cost (if it has immediate access) or there may be the necessity for optimum coding. In larger computers than Pegasus different considerations apply, of course.

2.5 The written form of an order

A single Pegasus order is made up of four parts; a typical one is written as follows:

3.1 2 01 4

In this order 3.1 is the *N-address* (or first address); it is here the address of one of the ordinary registers in the computing store. The second part of the order is called the *X-address* (or second address); in this order it is 2, indicating that the accumulator *X*2 is concerned. The *function* of this order is 01, which specifies a certain addition operation. The last part of the order is 4, which shows that the action of the order is to be modified by the content of *X*4 before being obeyed; this part of the order may be called the *modifier-address* (or *M*-part).

We shall confine ourselves at present to *unmodified* orders, i.e. those whose modifier-address is zero. In such orders the content of *X*0 is used for modification and no change occurs (it will be remembered that *X*0 is the dummy accumulator, whose content is always zero). If the modifier-address is zero there is no need to write it in; this part of the order may be left blank. For the moment therefore we shall write down only orders containing three parts.

Let us consider the effect of the order

3.1 2 01

when it is obeyed. This order causes the computer to take the word in the ordinary register 3.1 and to add it to the word in accumulator 2, leaving the result in *X*2. In other words, the content of *X*2 after the order has been obeyed is the sum of the previous content of *X*2 and the content of 3.1. This process does not disturb the content of 3.1 which will be the same before and after the operation of the order. After obeying the order in this way the computer will proceed to the next order. The order written

0.7 6 01

will similarly add *C*(0.7) and *C*(6) and leave the sum in *X*6, the expression *C*( ) denoting the content.

Both these orders have the function 01; in such orders the *X*-address (the second part of the order) is always the address of one of the accumulators. The *N*-address, on the other hand, may be the address of any register in the computing store (i.e. it may refer to an ordinary register, an accumulator, or a special register). For example, the order written

3 5 01

causes the computer to add $C(3)$ and $C(5)$ and leave the result in *X*5, and the order

7 7 01

will add $C(7)$ to itself, i.e. it will double $C(7)$.

We shall now introduce a notation which is useful for writing down the effects of various orders. We shall write *N* and *X* for the *N*-address and *X*-address respectively in an order. Thus in the order

3.1 2 01

N is 3.1 and *X* is 2, or we may write $N = 3.1$ and $X = 2$. In general *N* and *X* represent the addresses of whichever register and accumulator are written in the order. The content of *N* will be written as n, and the content of *X* as x, so that in the above order $n = C(3.1)$ and $x = C(2)$†. These represent of course the contents before the action of the order; we shall use n' and x' to represent the contents after the order has been obeyed. With the aid of this notation we can write the effect of an order with the function 01 as follows:

$$01 \qquad x' = x + n.$$

This equation simply states that the content of the accumulator specified by the *X*-address after the order has been obeyed is equal to the content of this accumulator before the operation of the order plus the content of the register specified by the *N*-address. Since n' is not referred to it is implied that the content of the register specified by the *N*-address is unaltered by the order; except of course for the special case in which $N = X$, as in the order

7 7 01

It is of course also understood that all the remaining registers are undisturbed.

The words on which an order operates are called the *operands*. In an order with function 01 the operands are the contents of the register and accumulator specified in the order. When describing the effects of various orders it is often convenient to refer to them by their function parts; thus we shall talk of an *01-order* when we mean an order whose function part is 01. The function part of an order is sometimes denoted by *F*.

It should be noted that the equation defining the effect of an 01-order is not affected by the conventional position of the binary point in the word, provided of course that we put it in the same position in all the words occurring. In particular, the words may be interpreted as numbers in either the fractional or the integer convention.

All the orders so far described have been 01-orders. In general the function of an order is written as a pair of octal digits, i.e. digits having values between 0 and 7. We may therefore get orders whose functions are 00, 01, 24, 53, 10, 65, 77, and so on. The operations caused by these different orders will be described in this and the next chapter, whose subject is the *order-code* of Pegasus, i.e. the catalogue or list of available functions and their effects.

The various operations which can be carried out by obeying an order fall naturally into *groups*. The group to which an order belongs is identified by its first function digit; thus the orders of group 6 are those whose function parts are written 60, 61, 62,...,67. The 01-order belongs to group 0. The effects of the orders in various groups may be roughly summarised as follows:

group 0	Simple copying or arithmetical operations, the result being left in an accumulator.
group 1	Similar to group 0 but the result is left in a register.
group 2	Multiplication and division.
group 3	Unassigned (i.e. a spare group) except for 37, see chapter 11.
group 4	Simple operations with integers.
group 5	Shifts (i.e. multiplication and division by powers of 2).
group 6	Jumps.
group 7	Transfers between the main and computing stores.

Some of the functions are not used, e.g. 07, 30, 75; orders with these functions are referred to as *unassigned orders*. The computer stops if it encounters such an order, and illuminates a special light (marked "unassigned order") on the control panel.

The whole order-code is summarized, for reference, in Appendix 1 and on a single sheet at the back of the book, and it is recommended that a beginner should refer to these frequently when starting to write programmes; it is not necessary to make the effort of learning the whole order-code by heart. The systematic arrangement of the orders make them easy to remember.

Before discussing any further how orders are written and what they do we shall describe briefly how orders are represented inside the computer.

2.6 The internal form of an order

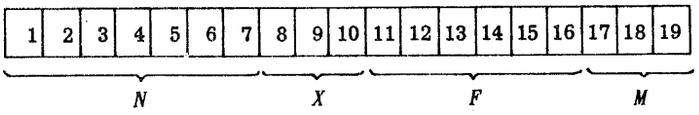
Inside the computer an order is represented by 19 binary digits; these we shall number 1 to 19, counting as usual from the left (i.e. from the most-significant or ms digit). The 7 bits numbered 1 to

† Occasionally it is useful to write x_2 instead of $C(2)$, etc.

7 are used to represent the *N*-address in the order; they may be called the *N*-digits of the order. The next 3 bits (digits 8,9,10) represent the *X*-address directly in binary; thus if *X* = 5 these three bits are 101. The function of the order is represented by the six *F*-digits (11 to 16) three of which are used for each octal digit in the written form of the order; this is illustrated by the following table:-

Written function	*F*-digits
01	000 001
24	010 100
53	101 011
67	110 111

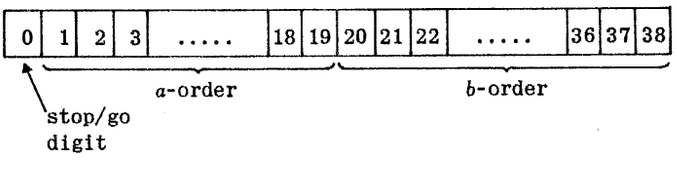
The last 3 bits of an order (digits 17,18.19) represent the modifier-address directly in binary; they are all zero in an unmodified order; these digits are sometimes called the *M*-digits in the order. The way in which the 19 bits of an order are allocated may be shown diagrammatically as follows.



We shall describe later, in detail, the way in which the *N*-digits are used to represent the *N*-address (see Section 3.12); at present we need only state that if the *N*-address is one of the accumulators then the *N*-digits give the address directly in binary. For example, the order written 2 5 01 is represented in the computer by the following 19 binary digits:

0000010	101	000001	000
N=2	*X*=5	*F*=01	*M*=0

Each of the two orders in a 39-bit order-pair occupies 19 bits. The left-hand or ms binary digit of the word does not belong to either of the orders; this digit (number 0) corresponds to the sign-digit of a number-word and is called the *stop/go digit*, its use will be described in Section 3.9. The *a*-order of the order-pair occupies digits 1 to 19 of the word, as described above; the *b*-order similarly occupies digits 20 to 38, these being used in the same way as corresponding digits of the *a*-order. The way in which the digits of an order-pair are allocated may be shown diagrammatically as follows.



It will be recalled that a simple order (such as an 01-order) is obeyed in two word-times. We shall now briefly describe the sequence of operations which occur when the order

3.1 2 01

is obeyed. The order will at this point be held in the order-register, and here its function digits (000 001) are decoded and used to set in motion a train of events. The first thing which occurs is that the mill is set up to do an addition; simultaneously the registers containing the two operands (i.e. 3.1 and *X*2) are connected to the input of the mill; these operands enter the mill and are added during the first word-time. During the second word-time the result of the operation comes out of the mill and is sent to its proper destination, viz. *X*2, where it replaces the previous content.

It is most important to distinguish the written form of a programme from its representation inside the computer. The written orders of a programme consist of ink or pencil marks on paper, and they are expressed in a way which has been chosen for the convenience of the programmer. Inside the machine the programme is represented in the form of trains of pulses or as magnetised areas on the drum. A conversion process has to be applied to the written orders before they can be stored or obeyed by the computer. The first step in this process is the "typing" out of the programme on a teleprinter (or a keyboard perforator), which provides a length of punched paper tape and a printed sheet (for checking purposes). The paper tape can be "read" by placing it in a tape-reader attached to the computer and calling in the Initial Orders. These are a permanently available programme stored in the isolated part of the main store; the computer can be caused to obey this programme by operating a special key on the control panel; this key is called the *Start key*, and the way in which it is used will be described in Section 4.3. The Initial Orders cause the computer to read the tape, one character at a time, and to build up the orders, which it places in the *main store* in the form required by the machine. The way in which the punched form of the orders is converted to the stored form depends *entirely* on the Initial Orders, and has been chosen so as to make the programme as convenient as possible to write and to punch. The main function of the Initial Orders is to read, convert and store programmes; it is used every time a programme is put into the computer. When the whole programme has been stored in the main store the Initial Orders can be caused to transfer a part of it to the computing store and to start the computer obeying it. This subject is discussed further in Section 4.3.

2.7 The orders of groups 0 and 1

The orders of groups 0 and 1 are concerned with simple operations such as copying, addition and subtraction. The 01-order has already been described; it will be recalled that its effects may be briefly summarized by the equation $x' = x + n$. The 00-order is even more simple; it is described by the equation

$$00 \qquad x' = n,$$

which means that the content of the specified accumulator is replaced by a copy of the content of the specified register. For example, the order

```
5.4  3 00
```

causes the word in ordinary register 5.4 to be copied into *X*3, the previous content of *X*3 being lost. The word in 5.4 is not changed by this order.

For example, if we have two numbers in ordinary registers 5.0 and 5.1, we can add them and leave the sum in *X*5 by the following two orders:

```
5.0  5 00     first number to X5
5.1  5 01     add second number
```

The result of any order of group 0 is always left in the specified accumulator. The first five orders of this group may be summarized as follows. Here the word "register" indicates the register (i.e. ordinary or special register, or accumulator) specified by the *N*-address written in the order.

F	Effect	Description
00	$x' = n$	Replace content of accumulator by copy of content of register.
01	$x' = x + n$	Add content of register into accumulator.
02	$x' = -n$	Replace content of accumulator by minus content of register.
03	$x' = x - n$	Subtract content of register from accumulator.
04	$x' = n - x$	Subtract content of accumulator from content of register, leaving the difference in the accumulator.

For example, suppose we have three numbers *a*, *b* and *c* in the ordinary registers 4.0, 4.1 and 4.2 respectively. We can form the quantity $a + b - c$ in *X*2 by the orders

```
4.0  2 00     a to X2
4.1  2 01     add b, result is a + b in X2
4.2  2 03     subtract c, final result in X2.
```

Alternatively the following orders could be used:

```
4.2  2 02     -c to X2
4.0  2 01     add a, result a - c
4.1  2 01     add b, result a + b - c
```

It is clear that there are many equally good ways of doing this operation.

The orders of group 1 are similar to those of group 0 but the result of the operation is always left in the specified register. The most important order of group 1 is the 10-order (read as "one-oh", not as ten):

$$10 \qquad n' = x,$$

which means that the word in the register is replaced by a copy of the word in the accumulator. For example, the order

```
2.4  7 10
```

replaces *C*(2.4) by a copy of *C*(7). The content of the accumulator is unchanged by this order.

As an example, suppose the numbers *a*, *b* and *c* are, as before, in 4.0, 4.1 and 4.2; we can place the quantity $a - b - c$ in 5.0 by the orders:

```
4.0  5 00     a to X5
4.1  5 03     subtract b, result a - b in X5
4.2  5 03     subtract c, result a - b - c in X5
5.0  5 10     a - b - c to 5.0
```

In this illustration we have used *X*5 to form the result; we have to use an accumulator for this, but any accumulator would have served equally well.

It will be seen that the 10-order is similar to the 00-order except that the roles of register and accumulator are interchanged. This analogy holds good for the remaining orders of group 1. The effect of any order of this group can be derived from the corresponding order of group 0 by interchanging the

words "accumulator" and "register" in the verbal description, or by interchanging the letters n and x in the defining equation. For example, the 01-order is defined by the equation

$$01 \qquad x' = x + n,$$

and the 11-order (read as "one-one") by the equation

$$11 \qquad n' = n + x.$$

In this order the number in the accumulator is added into the register. Thus, the order

```
0.6  3 11
```

causes the sum $C(0.6) + C(3)$ to appear in 0.6; the content of $X3$ is *unchanged*. It will be seen that the ordinary registers may be used for addition and subtaction. The first five orders of group 1 may be summarized as follows.

F	Effect	Description
10	$n' = x$	Replace content of register by copy of content of accumulator.
11	$n' = n + x$	Add content of accumulator into register.
12	$n' = -x$	Replace content of register by minus content of accumulator.
13	$n' = n - x$	Subtract content of accumulator from register.
14	$n' = x - n$	Subtract content of register from content of accumulator, leaving the difference in the register.

As an illustration, suppose we have two numbers in 4.0 and 4.1, and we have to replace the second number by their sum. We must use an accumulator for intermediate storage, let us use $X4$, and the following orders will do:

```
4.0  4 00     first number to X4
4.1  4 11     sum to 4.1
```

As a further example, suppose a, b and c are the numbers in 4.0, 4.1 and 4.2; the following orders will replace b by $a + b$ in 4.1, and c by $c - a - b$ in 4.2:

```
4.0  3 00     a to X3
4.1  3 01     a + b to X3
4.1  3 10     a + b to 4.1
4.2  3 13     c - (a + b) = c - a - b to 4.2
```

It sometimes happens that we wish to clear a register, i.e. to replace its content by zero. We use the dummy accumulator $X0$ for this. To clear an accumulator, say $X4$, we can use the order

```
0  4 00
```

and to clear an ordinary register, say 2.0, we use the order

```
2.0  0 10
```

There are, in nearly all these examples, many other solutions which are equally "good", i.e. which require as few orders. If possible we always prefer the most direct and simple method. Thus to form in $X4$ the difference $a - b$ of a (in 4.0) and b (in 4.1) we would prefer the orders

```
4.0  4 00     a to X4
4.1  4 03     a - b to X4
```

to the orders

```
4.1  4 00     b to X4
4.0  4 04     a - b to X4
```

If the numbers we are dealing with are in accumulators we prefer orders of group 0 to those of group 1; thus the orders

```
3  2 10
6  5 13
```

are equivalent to the orders

```
2  3 00
5  6 03
```

but the latter would be preferred. The choice between such alternative orders is largely an aesthetic matter, but if unconventional methods are used the checking of programmes is made much more difficult.

In the equations, such as

$$x' = x + n,$$

which define the effects of the orders of groups 0 and 1, the letters x and n represent the contents of the registers used; these may be interpreted according to either the fractional or the integer convention, provided of course that only one convention is used throughout any particular equation. In the orders 00 and 10 the operands need not have any particular significance - they may be fractions or integers or words to which no numerical value is attached (e.g. order-pairs).

Note that it takes two orders to add together two numbers in ordinary registers but only one order to add two numbers in accumulators. This is just one of the differences between the accumulators and the other registers.

The remaining orders of groups 0 and 1 are described in Section 3.11.

2.8 Writing the programme

The orders of a programme are written on printed programme sheets in the columns provided; an order-pair occupies two lines on the sheet. When the programme is eventually obeyed the order-pairs occupy ordinary registers in the computing store, and we usually write to the left of each order-pair the address of the register which will hold it at that time. For example, two order-pairs obeyed from 0.0 and 0.1 in the computing store, might be written as follows:-

0.0	5.0 1 00
	5.1 1 01
0.1	5.2 1 03
	3.6 1 10

Since the computing store is seldom large enough to hold the entire programme and the numbers on which it is to operate, these are all placed in the main store during the input process. When the whole programme (and, probably, the numbers) have been read in and placed in the main store, it is arranged that the first four blocks of the programme are copied into the computing store. This is all done by the Initial Orders programme, which also arranges for the computer to start obeying the orders at a specified point (this is called *entering* the programme). Consequently the programme has to be divided up into blocks, each of 8 order-pairs (i.e. 16 orders), and the programmer has to insert block-transfer orders in the programme at suitable places so as to read fresh blocks of orders from the main store into the computing store when those already in the computing store are no longer needed. Two or three of the six blocks of ordinary registers in the computing store are usually reserved for those parts of the programme which are currently being obeyed; the remaining blocks are used for constants, data, intermediate results, etc.

When starting to write a programme one cannot usually forsee exactly how best to divide it up into blocks. Parts of the programme are therefore sketched out in pencil on squared paper before being laid out in blocks on one of the printed programme sheets.

Figure 4.3 on page 72 shows a four-block programme sheet with the orders written in. It will be noticed that the addresses of the registers holding the order-pairs are abbreviated; the address is written in full against the first order-pair only in each block. The "box" above each block is used to indicate the main store block-number where the block of orders is held. Many of the details of this programme will be explained in later chapters. Programme sheets in general use are laid out with 2, 4 or 6 blocks.

Sometimes the orders needed will not completely fill a block. In this case unused words may be filled with zeros, or extra "dummy" orders may be inserted. A dummy order is one which has no effect, e.g. the order

0 0 00

which simply copies $C(0)$ into $X0$. This order is the one conventionally used as a dummy order; it may be written on a programme sheet simply as a zero in the N-address column.

Numbers may be written on the programme sheets as well as orders. They may be either fractions or integers and each number will of course occupy the whole of a register in the computing store (i.e. *two* lines on the programme sheet). A number must be preceded by its sign (+ or -) since this is used to distinguish it from orders. If it is a fraction the decimal point must be written. Numbers and order-pairs may be intermingled on the programme sheets, but the programmer must take care that the computer does not try to "obey" a number; unconditional jumps may be used to avoid this. As an illustration of how numbers may be written, let us suppose that the fractions 1/4 and -1/8, and the integer 1000 are to be placed in $X2$, 3 and 4 respectively at some stage in a calculation. The relevant part of the programme sheet might appear as shown on the facing page, where we assume that the computer starts to obey the orders in 0.3 (as indicated by the arrow).

Numbers written as a part of a programme in this way are often called *constants*, since their values are not altered. Usually only occasional constants that are required at some stage in a calculation are written on programme sheets. If the problem uses many numbers they would not be written on the programme sheets but all together on other sheets of paper.

The numbers and order-pairs are punched more or less as written on the programme sheets, and the resulting tape is read by the Initial Orders programme, which converts the items on the tape into the appropriate binary form and places them in the main store. We shall describe later the details of the punching; at present all we need say is that every number must be preceded by its sign and must be either a fraction (with a decimal point) or an integer. A fraction may have up to 11 decimal digits after the point, and will be converted to the nearest binary equivalent (with an error of at most $\frac{1}{2}\epsilon$, see

Section 1.8). An integer is converted into binary without error, as also is a fraction which can be precisely represented by a 39-bit word (e.g. 0.25 or −0.625).

			B 2		
0.0	+0.25				= 1/4 ⎫ fractions
1	−0.125				= −1/8 ⎭
2	+1000				integer
→ 3	0.0	2	00		Place 1/4 in *X*2
	0.1	3	00		Place −1/8 in *X*3
4	0.2	4	00		Place 1000 in *X*4

We often rule a vertical line to the left of a number written on a programme sheet: this is done to emphasise that it is not an order.

2.9 The special registers

The special registers are a group of registers in the computing store which are used for special purposes, they are not made up of delay-lines and cannot be used for storing words in the same way as the accumulators or the ordinary registers. They have the addresses 15, 16, 17, 24, 32, 33, 34, 35, 36, 37 (some other addresses are used when certain extra equipment is fitted to the basic computer). These registers are shown in the diagram of the store of the computer (Fig. 2.2).

Registers 32 to 37 hold permanently certain useful constants. For example, register 33 holds ½ so that the order

33 4 01

has the effect of adding ½ to the number in *X*4. The order

33 4 10

has no effect. The contents of these *constant registers* are as follows:

32	−1.0
33	1/2
34	2^{-10}
35	2^{-13}
36	2^{-16}
37	7/8

It would of course be easy to place one of these constants in a register by writing it on the programme sheet along with the orders which need it. These particular numbers are needed fairly often however, as will appear later; it should be noted that each of them (except that in 37) is represented by a word only one of whose 39 digits is a one.

The content of register 15 is determined by the settings of a row of switches on the control panel; these are referred to as the *handswitches*. Twenty of these switches are referred to as *H*0, *H*1, *H*2, ..., *H*19 and each of them may be either up or down. The up-position of each switch corresponds to 0 and the down-position to 1, so that we may set up any twenty-digit binary number by pressing down the switches where ones appear in the number. When this has been done an order such as

15 4 00

will place in *X*4 a word whose first 20 digits are determined by the handswitches; the remaining 19 bits of this word are all zero. For example, if *H*19 only is down the above order will place 2^{-19} in *X*4. We write *H*19 = 1 to indicate that the switch *H*19 is down. This kind of order is said to *read* the handswitches. An order such as

15 4 10

has no effect. The handswitches can be very useful to the programmer for manually directing the operations of the programme; how they are used will be explained later.

Registers 16 and 17 are used for input and output and will be described later. Register 24 stores the current setting of the external conditioning relays (described at the end of Section 3.10). Registers 20 and 21 are used only on those computers equipped with magnetic tape auxiliary storage. The content of any other register, such as 31, is always zero; such registers have the unallocated addresses between 8 and 63. The addresses 64 onwards are actually those of the ordinary registers; it is much more convenient to use the customary notation for these, however.

2.10 The orders of group 4

By using the orders of groups 0 and 1 together with numbers placed in ordinary registers along with the programme we can carry out many useful operations. In fact most of the orders in the majority of programmes are concerned with simple operations such as addition, subtraction and copying. The special constant registers relieve us of the need to punch and store certain useful numbers.

Most programmes require small integers for a variety of purposes. Such integers could of course be punched on the programme tape and read into the computer along with the programme; but this would be wasteful of storage, especially in the computing store, since many small integers are required during the course of a programme. This procedure would also be inconvenient. The orders of group 4 can be used to produce any small integers as they are needed. These orders closely resemble the orders of group 0; but the *N*-address (or first address) in the order is not any longer an address, but is actually the integer required. For example, the order

(27) 3 40

causes the integer 27 to be placed in *X*3. The number written on the left in the order is usually encircled so as to emphasize the fact that it is not the address of any register. This number can have any value from 0 to 127 (since it is represented by 7 binary digits inside the computer).

The effect of the 40-order may be written

$$x' = N,$$

where N stands for the encircled number written first in the order. Note that N is to be carefully distinguished from n, which is the content of a register. In this equation the words are, of course, to be interpreted on the *integer convention*.

The order

4 6 00

copies *C*(4) into *X*6, and the order

(4) 6 40

places the integer 4 in *X*6. The circle round the number is merely an aid when reading programme-sheets, no special punching is used to indicate that the number is encircled. In fact the above two orders are differentiated only by their function digits when they are punched on tape or stored inside the computer.

It will be seen that the 40-order is similar to the 00-order in that the result is left in the specified accumulator. The first operand is, however, explicitly written in the order and is not the content of any specified register. This analogy also holds for the remaining orders of group 4. The effect of any order of this group can be derived from the corresponding order of group 0 by replacing the words "content of register" by "number written in the order" in the verbal description. This is equivalent to replacing n by N in the defining equation. For example, the 01-order is defined by the equation

$$01 \qquad x' = x + n,$$

and the 41-order by the equation

$$41 \qquad x' = x + N,$$

where it is understood that the numbers concerned are integers.

Where necessary we shall put a suffix $_I$ or $_F$ on letters such as n or x to denote that the corresponding words are to be interpreted as integers or fractions respectively. For example x_F means the fraction x, and x_I means the integer x. The above equation could be written:

$$x_I' = x_I + N.$$

The first five orders of group 4 may be summarised as follows.

F	Effect	Description
40	$x_I' = N$	Replace content of accumulator by integer written in the order.
41	$x_I' = x_I + N$	Add integer written in the order to integer in accumulator.
42	$x_I' = -N$	Replace content of accumulator by minus the integer written in the order.
43	$x_I' = x_I - N$	Subtract integer written in the order from integer in accumulator.
44	$x_I' = N - x_I$	Subtract integer in accumulator from integer written in order; result in accumulator.

In this summary the equations are all written on the integer convention; if we use the fractional convention they must be written differently, for example the equation of the 41-order becomes

$$41 \qquad x_F' = x_F + N.2^{-38},$$

or

$$x_F' = x_F + N\epsilon.$$

As an example suppose we wish to use the integer 49 for counting purposes, it can be set initially into accumulator 2 by the order

(49) 2 40

To subtract 1 from the integer in *X*2 we would write

(1) 2 43

The remaining orders of group 4 are described in Section 3.11.

2.11 Overflow

According to the standard fractional convention any number x represented by a word must be between −1 and +1, in fact such a number must satisfy the inequalities

$$-1 \leqslant x < 1.$$

It may happen during arithmetical operations that numbers are produced which exceed capacity, i.e. lie outside the permitted range of values. This is called *overflow*. For example, overflow will occur if we add 0.71 to 0.95. In such an event the computer will produce a wrong result and a warning is given by a special *overflow-indicator*. This overflow-indicator is usually referred to as OVR; it is a two-state device which is normally *clear* but will be *set* when overflow occurs. Once OVR has been set it will remain set, regardless of other operations, until certain special orders are obeyed (see Section 3.8). One of the lights on the control panel shows the state of the overflow-indicator.

A "Stop on Overflow" key is provided which, when depressed, will cause the computer to stop on the completion of an order during which OVR has been set. On switching to STOP and RUN the computer will continue working as though the stop had not occurred, and OVR must be cleared and set again before the stop can occur again†.

When OVR is set it is impossible to transfer words to the main store; the computer will stop if it comes to an order calling for such an operation. This means that the numbers obtained from a sequence of calculations cannot be written away into the main store if overflow has occurred at any stage. Since, as a rule, the programmer will not want numbers to overflow this stop prevents the wastage of machine time which would occur if much further calculation were done with the wrong numbers. A special light on the control panel indicates this kind of stop; it is marked "writing with overflow".

If overflow occurs during any order except a division order, then the result actually obtained will differ from the correct result by 2 or a multiple of 2 (on the fractional convention). For example, if we attempt to add 0.71 to 0.95 we shall get −0.34, which is 2 less than the correct sum 1.66, and OVR will be set. Overflow can, of course, also occur with integers; in this case the result actually obtained will differ from the correct result by a multiple of 2^{39}.

Overflow can be thought of as a loss of significant digits at the left (or ms) end of the word; this is usually a serious matter unless the programmer has foreseen it, which may be difficult in a complicated programme. Overflow is usually caused by insufficient scaling-down of the numbers occurring in the calculation. One cannot lay down any definite rules about the action to be taken if overflow occurs, but the overflow-indicator can be sensed by the programme and it may be possible to arrange that the offending numbers are all automatically scaled down and the appropriate part of the calculation repeated.

Apart from orders causing writing into the main store (functions 71 and 73), the only orders affected by the setting of OVR are those with functions 23 (justify) (see Section 3.4), 64 and 65 (see Section 3.8). The results obtained from any other order will be the same whether OVR is set or clear. Note that the orders 00, 10, 40 and 42 cannot set OVR; the orders 02 and 12 will set OVR only if the number concerned is −1.0 (on the fractional convention).

▼ When the computer obeys an order such as

33 5 11

the sum, i.e. $C(33) + C(5) = \frac{1}{2} + C(5)$, is actually formed in the mill, although $C(33)$ is, of course, unaffected. An order of this kind may, however, set OVR. Similar remarks apply to the other constant registers and to register 15 (handswitches). This can sometimes be turned to good account; the above order will, for example, set OVR only if $C(5) \geqslant \frac{1}{2}$ and can be used to indicate this fact without disturbing the contents of any registers. The order

32 0 02

▲ is occasionally useful; it sets OVR but has no other effect.

---

† In the case where overflow is caused by a 20- or a 21-order (i.e. -1.0 x -1.0) which is obeyed in an *a*-order, the stop will occur after the *b*-order, i.e. the computer will be waiting to obey the next *a*-order. In all other cases the stop will occur immediately on completion of the order causing the overflow.

The Stop on Overflow key is not present on Pegasus 1.

Chapter 3

The Order—Code

This chapter is concerned with all the orders in the Pegasus order-code with the exception of those in groups 0, 1 and 4, which are described in Chapter 2, and a few less commonly used orders, which are described in later chapters.

3.1 Multiplication

In Pegasus most of the orders of group 2 are concerned with multiplication and division; the structure of this group is different from that of groups 0, 1 and 4.

If we multiply together two decimal numbers having each the same number of digits we get a product which contains up to twice as many digits. For example,

$$381 \times 615 = 234315$$

$$.381 \times .615 = .234315$$

The numbers on the left are the *factors* in the multiplication. In the same way if we multiply together two binary fractions, each having 38 bits after the point, we shall get a product with 76 bits after the point. If we add a sign-bit we get a 77-bit product. This requires *two* computer words for its representation and a number of this kind is therefore called a *double-length* or *double-precision* number. It is important that the computer should be able to form the full product of two words and generally to manipulate double-length numbers. In Pegasus, accumulators 6 and 7 can be used in the usual way like any of the other accumulators, but they have some special properties as well. They can be coupled together to form a *double-length accumulator*, in which double-length numbers can be conveniently handled. The full double-length product is placed there when the computer performs a multiplication.

There are three multiplication orders, of which the simplest is the 20-order (read as "two-oh", not twenty). Let us consider the action of the order

```
5.3  4 20
```

This order takes the number in ordinary register 5.3 and multiplies it by the number in *X*4, the full product being left in *X*6 and 7 (in place of the previous contents): the contents of 5.3 and *X*4 are undisturbed. In carrying out this multiplication the machine of course takes note of the signs of the two factors (i.e. the operands) and ensures that the product is correctly signed. The 20-order *always* places the product in *X*6 and 7, whatever the *N*-address and *X*-address written in the order; in fact the computer is so built that the multiplication is actually carried out in accumulators 6 and 7 and not in the mill at all. Let us now consider the way in which the product occupies the space available (78 bits) in *X*6 and 7.

The product has 77 binary digits, of which the left-most is the sign-bit. For convenience we shall number these digits from the left: 0, 1, 2, ..., 76; so that digit 0 is the sign-bit and digit 76 is the least-significant bit of the product (this is an extension of the usual way of numbering the digits of a single word - see Section 1.8). When the product is placed in *X*6 and 7, digits 0 to 38 of the product occupy *X*6 and the remaining digits (39 to 76) occupy *X*7, except that the sign-digit position in *X*7 is not used and the digit there is *always made zero*. We can think of *X*6 as receiving the left (or ms) half of the product, including the sign-digit, whereas *X*7 receives the right (or ls) half. This is illustrated in Fig. 3.1, in which the digit-numbers are shown.

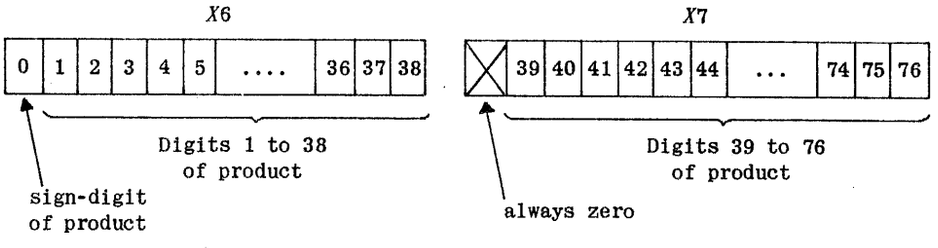


Fig. 3.1. *A double-length product in accumulators 6 and 7.*

Since accumulators 6 and 7 are often used for special purposes it is convenient to introduce a special notation for their contents. We shall write p for $C(6)$ and q for $C(7)$. The double-length number formed by placing the digits of p and q side by side will be denoted by (pq), but when we write this we

usually imply that the sign-digit in *X*7 does not belong to the number and is zero (i.e. $a \geqslant 0$). With the aid of this notation we can indicate the effect of the 20-order by means of the equation†

$$20 \qquad (pq)' = n \times x,$$

where $(pq)'$ means the double-length number after the order has been obeyed. The order can be described as:

20 *Multiply the content of the specified register by the content of the specified accumulator and place the product in accumulators 6 and 7.*

Just as with the orders of groups 0 and 1, the *N*-address can refer to any register (e.g. to an accumulator).

If there are two numbers in 5.0 and 5.1 and we wish to find their full double-length product, we could use the orders

```
5.0  3 00    First number to X3
5.1  3 20    multiply by second number
```

One of the two factors must be in an accumulator, and we have here used *X*3. The product will be placed automatically in *X*6 and 7 and we could, for example, store it in 5.2 and 5.3 by adding the following extra orders

```
5.2  6 10    ms half of product to 5.2
5.3  7 10    ls half of product to 5.3.
```

A sequence of orders like this may be used whenever we wish to form the full product of two words; it is unaffected by whether we interpret the operands as fractions or as integers.

Before going any further we must consider carefully how we are to interpret the double-length number (pq). To illustrate these interpretations we shall use products of 3-digit decimal numbers; and to avoid complications we assume at first that all the numbers concerned are positive.

If we multiply two integers the product will be an integer,†† for example,

$$381. \times 615. = 234315.$$

and if the two factors are fractions the product is a fraction:

$$.381 \times .615 = .234315$$

and if one factor is an integer and the other a fraction then the product will be a mixed number with its point in the middle:

$$381. \times .615 = .381 \times 615. = 234.315$$

In other words, if the point is to the left (or right) in both factors then it lies on the left (or right) in the product; if the point is to the left in one factor and to the right in the other then it lies in the middle in the product. All this applies to binary numbers just as well as to decimal ones; it is illustrated diagrammatically in Fig.3.2, *(page 28)* where the boxes represent the digits and the position of the binary point is marked.

The equation defining the effect of the 20-order can consequently be written

$$20 \qquad (pq)_I' = n_I . x_I,$$

where the $_I$ suffixes indicate that the quantities are integers, or

$$20 \qquad (pq)_F' = n_F . x_F,$$

where the $_F$ suffixes indicate that the quantities are fractions. In other words, this means that the double-length product is a fraction if both factors are fractions; it is an integer if both factors are integers.

Frequently both factors are "small" positive integers, in which case the left half (i.e. p) of the full product is zero. We can then take the right half (i.e. q) as the product (single-length). Examination of the top line of Fig.3.2 should help to clarify this. For example, suppose we have two "small" positive integers in 5.0 and 5.1; we can place their product in the single register 3.2 by the following orders

```
5.0  6 00    first integer to X6
5.1  6 20    multiply by second integer
3.2  7 10    product to 3.2 (X6 will be clear).
```

In this example *X*6 is used for two purposes; it first holds one of the numbers to be multiplied, but this gets *overwritten* (i.e. replaced) by the left half of the product (which is in fact zero). A

---

† It is common practice to leave out multiplication signs or to replace them by points, so we shall often write nx or $n.x$ instead of $n \times x$. But the special symbol (pq) does *not*, of course, mean the product $p \times q$.

†† We have written the decimal point in these integers merely to emphasize that it really lies to the right of the ls digit.

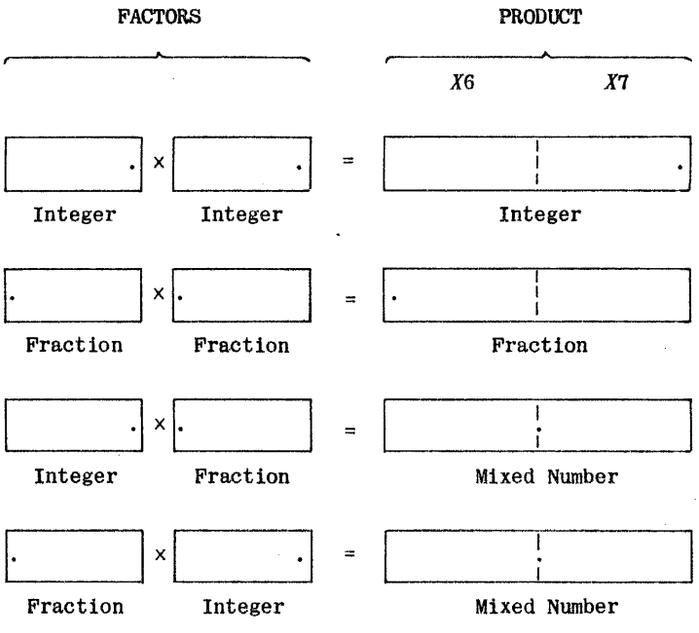


Fig.3.2 *How the position of the binary point in a double-length product is determined by its position in each of the two factors.*

sequence of orders like this can be used whenever the product is non-negative and less than 2^{38} (which is about 250 000 million); for example, one factor could be as large as 250 000 and the other a million. If the product is greater than or equal to 2^{38} the above sequence of orders will not place the correct product in 3.2, but only its 38 ls bits (such products cannot in fact be placed in a single register). Alternatively, if the product is single-length but may be negative then steps must be taken to correct the sign-bit in *X*7 before the product is used; this will be explained in Section 3.8.

When we multiply two integers it is usually the right half only of the product which is of interest. When we are dealing with fractions it is usually the left half that we wish to retain; this should be clear after looking at the second line in Fig.3.2. For example, if the two factors are 3/4 and 5/8 the product is 15/32, and all but the first few bits of the double-length product will be zero. As a rule the digits appearing in *X*7 will not all be zero but they can often be neglected. We shall discuss the multiplication of fractions in the next Section.

Let us now consider the multiplication of an integer by a fraction. It should be clear from Fig.3.2 that we must now regard the binary point in the product as lying between p and q. The integer in *X*6 (i.e. p_I) after the multiplication is the integral part† of the product, and the fractional part is the fraction in *X*7 (i.e. q_F). For example, suppose an integer a (= 9 say) is stored in 4.0 and a fraction y (= ¾ say) is in 4.1; the following orders will place in 4.2 the integral part (= 6) of the product $a.y$ (= 6¾).

```
4.0  6 00     a to X6
4.1  6 20     multiply by y
4.2  6 10     integral part [a.y] to 4.2
```

The fractional part (= ¾) of the product will be left in *X*7.

If we write $(pq)_M$ for the number in the double-length accumulator interpreted as a mid-point number, we can get another form of the defining equation of the 20-order

$$20 \qquad (pq)_M = n_I.x_F = n_F.x_I.$$

† The *integral part* of a number is the greatest integer which does not exceed the number, e.g. the integral part of 3.27 is 3 and the integral part of 2 is 2. Integral parts are sometimes denoted by square brackets; thus we can write $[x]$ for the integral part of x, and

$$[15\tfrac{1}{2}] = 15, \quad [6] = 6, \quad [0] = 0, \quad [\tfrac{3}{4}] = 0.$$

We must note carefully how the above definition applies to negative numbers; in fact

$$[-6] = -6, \quad [-2\tfrac{1}{4}] = -3, \text{ and } [-15\tfrac{1}{2}] = -16.$$

The *fractional part* is the number minus its integral part; it is never negative. For example, the integral part of 3.456 is 3 and the fractional part is 0.456.

Before describing the multiplication of fractions we must examine further the double-length number (pq). Suppose we know the values of p and q, what is the value of (pq)? We shall for the moment adhere to the fractional convention and write p_F, q_F and $(pq)_F$ for the fractions concerned. It is easy to find the value of $(pq)_F$ by a simple extension of the rules given in Section 1.8 for evaluating a single-length fraction. The sign-digit (if it is 1) contributes -1 to the value, and if digit k of the product is 1 it contributes 2^{-k} (we must now consider k as running from 1 to 76). Now the total contribution from all the digits in $X6$ is simply p_F. When we consider q_F as the right half of a double-length fraction we see that each of its digits is 38 places further to the right than usual; and consequently the contribution made by these digits to the double-length fraction is $q_F \times 2^{-38}$, i.e. ϵq_F. The value to be assigned to the double-length fraction is therefore

$$(pq)_F = p_F + \epsilon q_F,$$

with $q_F \geqslant 0$. If q_F is negative we can still assign the value $p_F + \epsilon q_F$ to the double-length number. On the integer convention the two halves of the number, p_I and q_I, are integers and the point in $(pq)_I$ is 76 places further to the right than in $(pq)_F$. The value of $(pq)_I$ is therefore

$$(pq)_I = 2^{38} p_I + q_I$$

on the integer convention. When the product is a mid-point number we can write it $(pq)_M$; clearly

$$(pq)_M = p_I + q_F.$$

3.2 Rounded multiplication

We have explained in the preceding section that when we multiply two fractions together it is usually only the left half (i.e. p) of the product that is of interest. The error committed in disregarding the right half entirely is always less than ϵ, which is about 0.00000 00000 036. If this is small compared with the value of the product we can often legitimately take the single-length fraction p_F (in $X6$) to be the product. It is important to realise that as a rule p_F is only an approximation to the product; it is true that it is normally a very good approximation (the error is in fact only ϵq_F), but it is nevertheless subject to bias. This is because q is non-negative, which means that p_F may (exceptionally) be equal to the product but will normally be too small by any amount up to ϵ. This bias may lead to considerable error in the later stages of an extended calculation, and to avoid it we must *round* the product.

This rounding process is automatically carried out by the computer if the 21-order is used for multiplication instead of the 20-order. Thus if we have two fractions in registers 5.3 and 5.4 and we wish to place their rounded product in 5.5, we can use the following orders

```
5.3  2 00    first fraction to X2
5.4  2 21    multiply by second fraction and round
5.5  6 10    rounded product to 5.5.
```

The 21-order may be described as follows:

21 *Multiply the fraction in the specified register by the fraction in the specified accumulator and place the rounded product in X6 (X7 receives the rest of the product).*

The content of $X7$ after this order is seldom wanted, we shall shortly explain just what it is. The rounding can be thought of as putting into $X6$ the best single-length approximation to the true double-length product. This approximation may be larger or smaller than the correct product but the error never exceeds $\pm \frac{1}{2}\epsilon$ and is unbiased.

As an example, suppose we have three fractions x, y and z in registers 5.0, 5.1 and 5.2 respectively, and we wish to place in 5.3 the quantity $x + yz$. Since the result is to be a single-length fraction it must be rounded to get the closest approximation. The following orders will do what is required

```
5.1  6 00    y to X6
5.2  6 21    yz (rounded) to X6
5.0  6 01    x + yz (rounded)
5.3  6 10    result to 5.3.
```

The addition of the single-length fraction x to the product cannot affect the rounding.

The 21-order is intended mainly for multiplying fractions; but it is sometimes useful when one factor is an integer and the other a fraction, when the product is a mid-point double-length number (see Fig.3.2). That part of the product which appears in $X6$ will now be the integer *closest* to the product; if a 20-order had been used it would have been the integral part of the product. For example, suppose an integer a (= 9 say) is stored in 4.0 and a fraction y (= ¾ say) is in 4.1; the following orders will place in 4.2 the nearest integer (7) to the product $a.y$ (= 6¾).

```
4.0  6 00    a to X6
4.1  6 21    multiply by y and round
4.2  6 10    nearest integer to a.y to 4.2
```

The reader should compare this example with a similar one in the previous Section.

To explain how the rounding is done in a 21-order it is simplest to consider the product as a mid-point number. Suppose we have used a 20-order to produce the product $(pq)_M$, i.e. a double-length number with integral part p_I and fractional part q_F. This number is now to be adjusted so that $C(6)$ will be the integer nearest in value to $(pq)_M$. Clearly we need not alter p_I if $q_F < ½$, and we should increase p_I by 1 if $q_F \geqslant ½$. This can be done by adding ½ to q_F and adding any carry which then occurs to the least-significant digit of p_I, i.e. we must add ½ to $(pq)_M$. This is shown in Fig.3.3 (adding 1 to digit 39 of the product is equivalent to adding ½ to $(pq)_M$). This rounding is done automatically by the

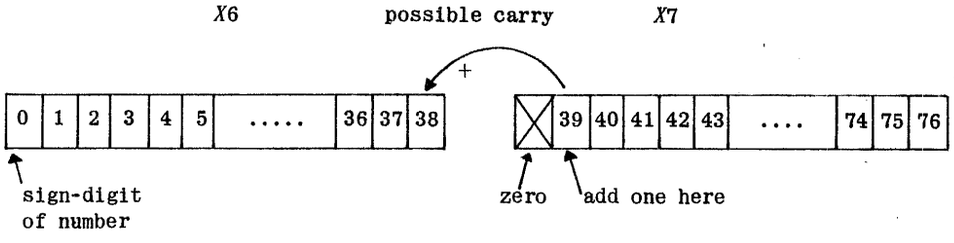


Fig.3.3 *Rounding a double-length fraction to produce a single-length fraction in X6.*

21-order. The order may thus be defined by means of the equation

$$21 \qquad (pq)'_M = n_I.x_F + ½ = n_F.x_I + ½ \qquad (q' \geqslant 0),$$

if the product of an integer and a fraction is being formed; or by means of the equation

$$21 \qquad (pq)'_F = n_F.x_F + ½\epsilon \qquad (q' \geqslant 0),$$

if we are multiplying two fractions.†

One of the reasons for providing special register 33, whose content is ½ (see Section 2.9), is to facilitate the rounding of double-length fractions which are not obtained by a simple multiplication. The rounding of a double-length number should be done only when the closest single-length approximation to its left half is needed.

3.3 Cumulative multiplication

The 22-order is the third of the multiplication orders in the Pegasus order-code; it facilitates the accumulation of products, which is a frequently needed operation. This order may be defined as††

$$22 \qquad (pq)' = (pq) + n.x,$$

or verbally as follows:

22 *Multiply the content of the specified register by the content of the specified accumulator and add the resulting double-length product into the double-length accumulator (X6 and 7).*

In this order the factors may be integers or fractions provided (pq) and $(pq)'$ are properly interpreted.

For example, suppose a, b, c, d are four positive integers stored in 3.0 to 3.3, and we have to place the integer $ab + cd$ in 3.4, we could use the following orders

3.0	5	00	a to $X5$
3.1	5	20	ab to $X7$
3.2	5	00	c to $X5$
3.3	5	22	add cd to ab in $X7$
3.4	7	10	$ab + cd$ to 3.4.

As another illustration, suppose u, v, w, x, y, z are six fractions held in 5.0 to 5.5, and we wish to evaluate the fraction $uv + wx + yz$ and place it in 5.6. In this case we want a rounded single-length result and we must therefore arrange to add ½ϵ to the sum of products before storing the ms half; this is most easily done by using a 21-order to form the *first* product, since it does not matter at which stage the addition of ½ϵ is done.

---

† It is instructive to compare this method of rounding with that applicable to decimal numbers. Suppose we wish to discard the last 3 digits of the number 0.142857. The result we should retain is 0.143, and this can be got by adding 0.0005 to the number before discarding the unwanted digits. The amount to be added is ½ in the last place to be retained.

†† Strictly speaking this equation should read

$$22 \qquad (pq)' = p + \epsilon q + n.x,$$

on the fractional convention, since there is no restriction on the sign of the original $C(7)$.

```
5.0  1 00    u to X1
5.1  1 21    (pq)' = uv + ½∈
5.2  1 00    w to X1
5.3  1 22    add wx to (pq)
5.4  1 00    y to X1
5.5  1 22    add yz
5.6  6 10    rounded result to 5.6.
```

Sequences of orders like this are useful in the evaluation of the scalar product of two vectors.

It should be noted that the three multiplication orders are all correctly signed; they will give arithmetically correct results regardless of the signs of the operands. This of course assumes that we require either the full double-length product or the single-length rounded product in *X*6. If the right half of the product (in *X*7) is all that is wanted, as when multiplying "small" integers, special arrangements (described in Sections 3.8 and 5.10) will have to be made if the result is likely to be negative; this does not often occur.

Overflow (see Section 2.11) may occur as a result of a multiplication. The 20 and 21 orders can cause overflow only if the fraction −1.0 is squared, the true result should be +1.0 (or +1.0 + ½∈ for the 21-order) which is outside the permitted range; the result actually obtained in this case will be −1.0 (or −1.0 + ½∈ for the 21-order) and OVR will be set. With the 22-order there are of course more possibilities for overflow; but it should be noted that it is only the final result which determines whether OVR gets set, i.e. overflow will *not* occur if −1.0 is squared and added to a negative number standing in the double-length accumulator.

Multiplication is, not unnaturally, a slower operation than addition or subtraction. A single order of groups 0, 1 or 4 is obeyed in 3 word-times if it is an *a*-order, or in 2 word-times if it is a *b*-order (see Section 2.4). A 20- or 21-order takes 13 extra word-times, and a 22-order takes 14 extra word-times. Thus a 20-order takes altogether 16 word-times if it is an *a*-order, or 15 if it is a *b*-order. As a rough figure one can say that multiplication takes about 2 milliseconds. In timing a section of programme containing multiplication orders it is usually simplest to find the time on the assumption that multiplication is as fast as addition, and one can then add in 13 word-times for each 20- and 21-order and 14 word-times for each 22-order.

3.4 Double-length addition and subtraction

The numbers normally handled by Pegasus are represented by single words and have 39 binary digits. We have seen how double-length numbers can arise as a result of multiplications. It is sometimes desirable to operate entirely with double-length numbers, which contain the equivalent of nearly 23 decimal digits. When adding or subtracting double-length numbers we must arrange for carries to take place between the two words. The 23-order is provided to facilitate this; it is called the *justify* order.

Operations on double-length numbers are usually carried out in the double-length accumulator formed by *X*6 and *X*7. A fraction in this accumulator is denoted by $p + \epsilon q$ in general, but the notation (pq) is preferred when q is non-negative. It is usually desirable to adjust a double-length number so that its right (or ls) half is non-negative (this is essential when shifting, as will be explained in Section 3.7), such a number is said to be *in standard form*.

Let us consider the addition of a double-length number in registers 5.0 and 5.1 to another such number in *X*6 and 7; we need *not* assume that these numbers are in standard form. We must first add the right halves:

```
5.1  7 01
```

This is the operation which may give rise to a carry which is to be added to the ls end of the left half of the sum. A carry will be necessary if the above order sets OVR, or if *C*(7) is negative. The justify order therefore examines OVR and the sign bit in *X*7, determines from them, and effects, the required carry, and then clears OVR and the sign bit in *X*7. There are the following four possibilities.

OVR	Sign of *C*(7)	Carry required
clear	+	0
clear	−	−∈
set	+	−2∈
set	−	+∈

This table may easily be checked by noting that if OVR is set then the sign of *C*(7) is incorrect. The 23-order adds the carry into the ls end of the register specified by its *N*-address (the *X*-address is not used and we usually write zero here). The complete sequence for adding the above two double-length numbers is therefore as follows.

```
5.1  7 01    add right halves
  6  0 23    justify into X6
5.0  6 01    add left halves.
```

Subtraction may be done by replacing the two 01-orders in this sequence by 03-orders.

A verbal description of the 23-order reads as follows:

23 *Put into standard form the double-length number in the specified register and X7, on the assumption that a preceding addition or subtraction in X7 has determined C(7) and the state of OVR. Leave C(7) non-negative, and leave OVR clear unless the left half of the number (in the register) overflows.*†

The 23-order takes the same time as an 00-order.

The operation of the order may be defined algebraically as follows:

$$23 \qquad (nq)' = n + \epsilon q + \text{contribution from OVR}, \qquad (q' \geq 0),$$

where $(nq)'$ is the double-length fraction in N and $X7$ with $q' \geq 0$, and the "contribution from OVR" is zero if OVR is clear but is $\pm 2\epsilon$ if OVR is set (the sign being opposite to that of q); OVR is left clear unless n' overflows.

The sequence given above for the addition of two double-length numbers will as a rule leave OVR set only if the double-length sum exceeds capacity. There are exceptional cases in which the carry causes overflow in the intermediate partial sum while the complete sum (after the last addition) is within capacity††.

The 23-order may also be used when operating with numbers more than two words long (multi-length numbers).

3.5 Division

Division and multiplication are normally thought of as inverse processes and it is desirable that the correspondence between them should be reflected in the way the computer operates. We should therefore expect that if we divide the product of two numbers by one of them, then the resulting quotient should be the other number; or, in symbols, we expect that

$$\frac{a.b}{a} = b.$$

One consequence of this apparently trivial statement is that, since the product $a.b$ is a double-length number, we must be prepared to divide a double-length number by a single-length number. It is also useful on many occasions to get not only the quotient but also the remainder after a division (in the above example the remainder is zero). In general, therefore, we require to take a double-length *dividend* and a single-length *divisor* and from them to find a *quotient* and *remainder*, which will both be single-length numbers, and we must provide for negative numbers as well as positive ones.

To take a simple example, consider the division of 43 by 5. Here 43 is the dividend and 5 is the divisor; and we can see that 5 goes 8 times into 43 with 3 left over, so that the quotient is 8 and the remainder is 3. We should note that the remainder is less than the divisor and is non-negative (i.e. $0 \leqslant 3 < 5$), and there is a simple relationship between these numbers which can be written as

$$\frac{43}{5} = 8 + \frac{3}{5} \qquad \left(0 \leqslant \frac{3}{5} < 1\right)$$

or, what amounts to the same thing,

$$43 = 5 \times 8 + 3 \quad (0 \leqslant 3 < 5)$$

The quotient can usefully be thought of as the integral part of the fraction 43/5. In general, if we write u and v for the dividend and divisor, and q and r for the quotient and remainder, we can write

$$\frac{u}{v} = q + \frac{r}{v} \qquad \left(0 \leqslant \frac{r}{v} < 1\right), \tag{1}$$

or

$$u = v.q + r \qquad (0 \leqslant r < v). \tag{2}$$

These two statements are equivalent provided the divisor v is positive. Assuming that the divisor is positive for the moment, either statement completely determines q and r when u and v are specified (it is assumed that all these quantities are integers and that v is not zero).

The 24-order may be used to carry out this kind of division process. Let us consider the order

5.2 3 24

In this division order the double-length dividend is formed by taking $C(3)$ for the left half and $C(7)$

---

† It is assumed throughout that $N \neq 7$. The sign-bit in $X7$ will not be cleared by the order

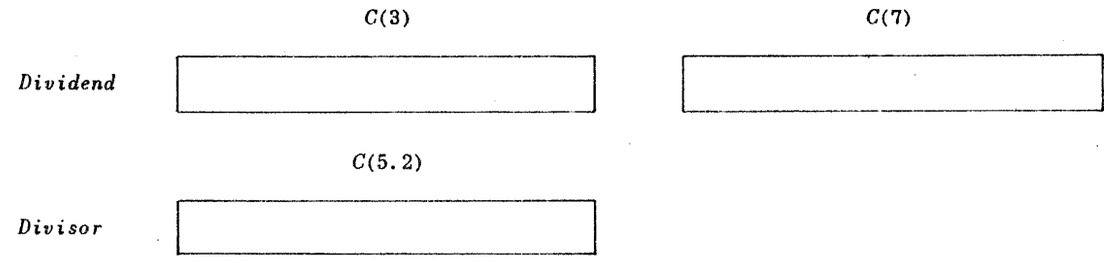
7 0 23

†† For example, with the numbers

$$p = 1 - \epsilon, \quad q = \tfrac{1}{2}; \quad C(5.0) = -\tfrac{1}{2}, \quad C(5.1) = \tfrac{1}{2}.$$

More elaborate sequences of orders can be written down for handling such numbers.

for the right half. The divisor is $C(5.2)$. Thus



The left half of the dividend is taken from the accumulator specified in the order; it is made into a double-length number by adjoining the content of *X*7 as shown above. If the dividend were in the double-length accumulator (*X*6 and 7) we would simply specify *X*6 in the division order. The divisor is the content of the specified register. The 24-order always places the quotient in *X*7 (the mnemonic "*q* for quotient" may help in remembering this) and the remainder in *X*6.

For example, to divide a double-length integer u in *X*6 and 7 by a single-length integer v in 3.0, placing the quotient q in 4.0 and the remainder r in 4.1, we could use the following sequence of orders.

```
3.0  6 24     divide u by v
4.0  7 10     quotient to 4.0
4.1  6 10     remainder to 4.1.
```

The quantities u, v, q, r satisfy the relationships given above.

Frequently the dividend is a single-length integer, when we put it into *X*7 and specify *X*0 in the division order; the left half of the double-length dividend is then zero. For example, to find the quotient and remainder when 43 is divided by 5 we could use the orders

```
(43) 7 40     43 to X7
 (5) 6 40     5 to X6
 6  0 24
```

Here the dividend is formed from $C(0)$ and $C(7)$, and the divisor is $C(6)$; as always the quotient (in this case 8) appears in *X*7 and the remainder (3 here) in *X*6. Note that the quotient and remainder always replace the previous contents of *X*7 and *X*6.

The double-length dividend in a 24-order may be denoted by (xq), since it is formed from x, the content of the specified accumulator, and q, the content of *X*7. This notation strictly implies, however, that q is not negative, and in fact there is *no restriction* on the sign of q; so that to be precise we should write the dividend $x.2^{38} + q$ on the integer convention, or $x + \epsilon q$ on the fractional convention. Since this notation is clumsy we shall continue to write (xq) for the dividend, with the understanding that in this case q may be negative.†

The operation of the 24-order may be described, on the integer convention, by either of the relationships

$$\frac{(xq)}{n} = q' + \frac{p'}{n} \qquad (0 \leqslant \frac{p'}{n} < 1), \tag{3}$$

or

$$(xq) = n.q' + p' \qquad (0 \leqslant p' < n), \tag{4}$$

which the reader should compare with those numbered (1) and (2) at the beginning of this Section. Since we must allow the divisor n to be negative we prefer the form (3) above (the inequality in (4) is impossible when n is negative). The definition of the 24-order on the integer convention is therefore as follows: ††

$$24 \qquad q' + \frac{p'}{n} = \frac{(xq)}{n} \qquad (0 \leqslant \frac{p'}{n} < 1).$$

It may be described verbally as follows:

24 *Divide the double-length number in the specified accumulator and X7 by the number in the register; place the quotient in X7 and the remainder in X6.*

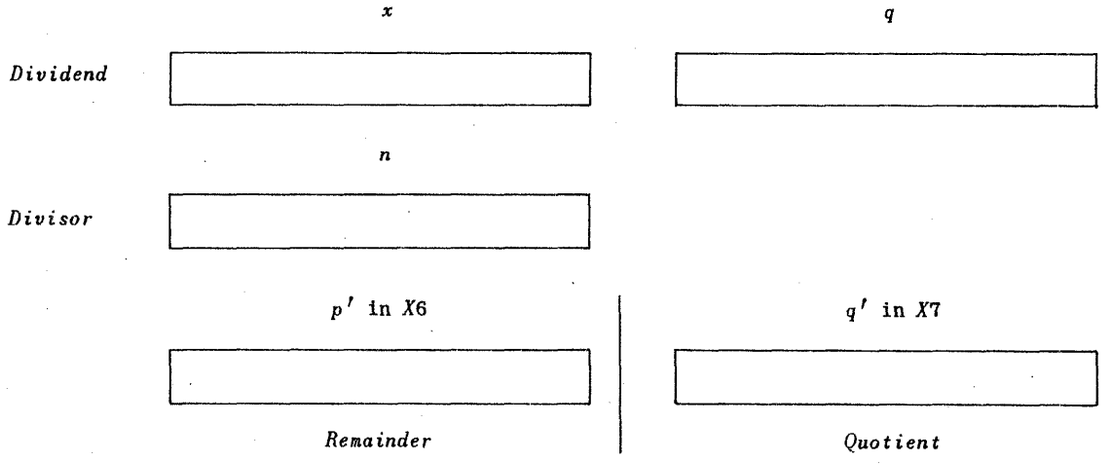
These descriptions are valid even for negative divisors or dividends; note that the remainder (unless it be zero) always has the same sign as the divisor. It is not often that negative remainders are of interest. The quotient produced by the 24-order is always the integral part of the number $(xq)/n$, i.e. the greatest integer not exceeding this number (see Section 3.1).

---

† At the start of the division process the divisor is placed in a special multiplicand/divisor register; the left half (x) of the dividend is simultaneously copied into *X*6 and a *partial justification* occurs in which q is made non-negative and $C(6)$ correspondingly corrected (if necessary). During the division the divisor gets added and subtracted into *X*6, the altered dividend is shifted up (double-length), and the quotient is built up, digit by digit, in *X*7.

†† We adhere to the convention that the results of an order (p' and q' in this case) appear on the left in the defining equation.

The diagram shows the five words concerned in the order.



The overflow-indicator will be set on attempting to divide by zero or if the quotient exceeds capacity. The quotient can never exceed capacity if the dividend is a single-length integer (unless $q = -2^{38}$ and $n = -1$).

As an illustration of the use of the 24-order, let us suppose that a positive integer stored in 5.0 represents a sum of money expressed in units of a penny, and that we require to convert it to £.s.d., placing the number of £'s in 4.0, the number of shillings in 4.1 and the number of pence in 4.2. If we denote these last three quantities by l, s and d respectively, and p is the sum involved then

$$p = 240l + 12s + d,$$

$$= 12(20l + s) + d,$$

so that d is the remainder when p is divided by 12 and $20l + s$ is the quotient; if this quotient is in turn divided by 20 we can find l and s. Thus the following sequence of orders can be used.

```
(12)  6 40     12 to X6
 5.0  7 00     p to X7
   6  0 24     divide p by 12
 4.2  6 10     d to 4.2
(20)  6 40
   6  0 24     divide (20l + s) by 20
 4.0  7 10     l to 4.0
 4.1  6 10     s to 4.1.
```

3.6 Rounded division

We have so far considered the division of integers, where the 24-order takes a dividend $(xq)_I$ and a divisor n_I and evaluates a quotient q_I' and remainder p_I' connected by the relationships

$$24 \qquad q_I' + \frac{p_I'}{n_I} = \frac{(xq)_I}{n_I} \qquad \left(0 \leqslant \frac{p_I'}{n_I} < 1\right) \qquad (1)$$

The $_I$ suffixes indicate that the quantities are all integers. These relationships must be supplemented by the information that all the numbers concerned are integers. If we wish to divide fractions we must interpret all the words concerned as fractions and the above relationships must be written

$$24 \qquad q_F' + \epsilon\frac{p_F'}{n_F} = \frac{(xq)_F}{n_F} \qquad \left(0 \leqslant \frac{p_F'}{n_F} < 1\right). \qquad (2)$$

We must also supplement these relationships by the information that all the numbers concerned are "fractions", in the sense that they are expressible by words according to the usual conventions; this implies that they are all integer multiples of ϵ.

In general if we express in binary the ratio of two numbers (xq) and n we obtain an infinitely long number, which can be called their "true quotient". For example, if the dividend (xq) is 1/8 and the divisor n is 7/8, the true quotient is 0.001001001001001... (this corresponds in decimal to the fact that 0.1/0.7 = 0.142857142857....). In the 24-order the division process is stopped when the first word-full of this true quotient has been evaluated; and it is at this stage that we get a remainder, which is the number we could use as a new dividend (with the original divisor) to continue the division process. Apart from the exceptional case when the remainder p' is zero, the quotient q' obtained in this way is an approximation which is always less than the true (infinitely long)

quotient.† In fact the value of the true quotient always lies between q' and $q' + \epsilon$ on the fractional convention, and sometimes $q' + \epsilon$ is a better approximation than q'. When we are dividing fractions it is this best single-word approximation to the true quotient that is usually wanted; it never differs from the true quotient by more than $\frac{1}{2}\epsilon$ and is unbiased. We here have a situation similar to that arising in the 21-order; in both cases we wish to get the best single-word approximation to a long number, and this is done by rounding the long number.

The 25-order is provided to give rounded quotients. It is generally similar to the 24-order but it yields a quotient q' which never differs from the true quotient by more than $\frac{1}{2}\epsilon$. The 25-order may be defined by the relationships

$$25 \qquad q' + \epsilon\frac{p'}{n} = \frac{(xq)}{n} \qquad (-\tfrac{1}{2} \leqslant \frac{p'}{n} < \tfrac{1}{2}), \tag{4}$$

on the fractional convention. The only difference from the relationships (2) is in the inequality satisfied by p'/n. As with the 24-order, there is no restriction on the sign of q in the dividend. A verbal description is as follows:

25 *Divide the double-length number in the specified accumulator and X7 by the number in the register; place the rounded quotient in X7 (and the corresponding remainder in X6).*

When dealing with fractions the remainder is not often needed but p' is still a "true" remainder, in the sense that it may be used as a new dividend to continue the division process; if this is done the result will be a fraction numerically not exceeding $\frac{1}{2}$ which may have either sign.

As an example, suppose u, v and w are three fractions stored in 4.0, 4.1 and 4.2, and we wish to place in 5.0 the fraction uv/w. Since we are dealing with fractions it is understood that we require the rounded value. We can use a 20-order to evaluate the full double-length product uv since the 25-order can deal with this as dividend.

```
4.0  6 00    u to X6
4.1  6 20    uv to X6 and 7
4.2  6 25    divide uv by w (rounded)
5.0  7 10    quotient to 5.0.
```

In this example the double-length dividend is in X6 and 7. Note that the rounding is deferred as long as possible.

Frequently the dividend, as well as the divisor, in a single-length fraction. Suppose, for example, we have to divide a fraction in X3 by another fraction in 5.0 and place the rounded quotient in 5.1. We could use the following orders:

```
 0   7 00    clear right half of dividend
5.0  3 25    divide
5.1  7 10    quotient to 5.1.
```

This kind of division occurs so often in practical calculations that a special order, function 26, has been provided for it. In the 26-order the dividend is simply the single-length fraction in the specified accumulator, in other respects the order resembles the 25-order. Thus in the above example the following orders provide a solution:

```
5.0  3 26    divide C(3) by C(5.0)
5.1  7 10    quotient to 5.1.
```

The 26-order may be defined by the relationships

$$26 \qquad q' + \epsilon\frac{p'}{n} = \frac{x}{n} \qquad (-\tfrac{1}{2} \leqslant \frac{p'}{n} < \tfrac{1}{2}), \tag{5}$$

on the fractional convention (it is of little use with integers). A verbal description is:

26 *Divide the fraction in the specified accumulator by the fraction in the register; place the rounded quotient in X7 (and the corresponding remainder in X6).*

The overflow-indicator (OVR) will be set by any of the three division orders if the divisor is zero, or if the quotient exceeds capacity.

The 25-order is sometimes useful with integers, when its defining relations become

$$25 \qquad q' + \frac{p'}{n} = \frac{(xq)}{n} \qquad (-\tfrac{1}{2} \leqslant \frac{p'}{n} < \tfrac{1}{2}).$$

The quotient q' is then the integer nearest in value to $(xq)/n$.

---

† In fact

$$q' \leqslant q' + \epsilon\frac{p'}{n} = \frac{(xq)}{n} < q' + \epsilon, \tag{3}$$

since $0 \leqslant \frac{p'}{n} < 1$.

Suppose a and b are the dividend and divisor respectively in a division; the 24-order produces a rounded-down approximation to a/b, i.e. on the integer convention the quotient given by the order is the greatest integer not exceeding a/b. The 25-order gives an approximation which is rounded to the nearest integer (in case of ambiguity it rounds up). Occasionally we want a quotient which is rounded *up*, i.e. it is the smallest integer not less than a/b. We can get this by dividing $-a$ by b (or a by $-b$) and changing the sign of the quotient. For example, if a and b are integers in 5.0 and 5.1 we can place a rounded-up quotient in 4.0 by means of the orders:

```
5.0  7 02     -a to X7
5.1  0 24     divide by b
4.0  7 12     minus quotient to 4.0.
```

A rounded-down quotient would be obtained by replacing the 02 and 12 orders by 00 and 10 orders respectively.

▼ It is sometimes useful to use a double-length dividend in which the binary point is between the two halves. If such a number is divided by an integer then the quotient must be interpreted as a fraction, and vice versa. This fact can be derived either from the appropriate defining equations or by consideration of the inverse process of multiplication. For example, suppose we have a fraction u in 5.0 and we wish to replace it by $7u/13$ we could use the following sequence of orders

```
(7)  6 40     7 to X6
5.0  6 20     7u to X6 and 7
(13) 5 40     13 to X5
  5  6 25     divide 7u by 13
5.0  7 10     rounded quotient to 5.0.
```

To divide a fraction u in 5.0 by 18 we could use the following orders

```
5.0  7 00
(18) 6 40
  6  0 25
5.0  7 10
```

To place in 4.0 the integral part of u/v, where u and v are the fractions in 5.0 and 5.1 we could use this sequence.

```
5.0  7 00
5.1  0 24
4.0  7 10
```

Any division order takes a time of about 5½ milliseconds, or precisely 41 word-times longer than ▲ an 00-order.

The 27-order is described in chapter 11.

3.7 Shifts, the orders of group 5

The orders of group 5 are concerned with shifting the digits of words, i.e. taking the binary digits of a word and moving them to the left or right. Let us consider the equivalent operation on a decimal number, for example,

0.00123

If we move the decimal point two places to the right in this number we get 0.123 (discarding extra zeros), i.e. a number $10^2 = 100$ times as big as the original number. We prefer to think of this operation as moving the digits of the number two places to the left (past the fixed decimal point). In general, if we move the digits of a decimal number N places to the left we shall have multiplied the number by 10^N; movement in the opposite direction corresponds to division by 10^N (or multiplication by 10^{-N}). In binary the effects are similar but we must use powers of 2 rather than powers of 10. For example, if we take the binary fraction 0.00101, which has the value 5/32, and shift its digits two places to the left we get 0.101, which has the value $5/8 = 4 \times 5/32$ (since $2^2 = 4$). If we had shifted the digits one place to the right we would have got 0.000101, whose value is $5/64 = ½ \times 5/32$. To prevent confusion we shall often talk of shifting a number *up* or *down* instead of shifting its digits left or right. If we shift a number up it becomes numerically larger.

The 50-order is the order normally used for shifting a number up. For example, the order

```
(3) 5 50
```

causes the number in $X5$ to be shifted up 3 places, i.e. multiplied by $2^3 = 8$. The number of places through which it is to be shifted is called the shift-number and is written in the N-address position in the order; as with the orders of group 4, this number is encircled since it is not an address.

The 51-order is written in a similar way and is used to shift numbers down. For example the order

(6) 2 51

will shift the number in *X*2 down six places, or divide it by $2^6 = 64$.

The 50– and 51–orders may be defined by the equations

$$50 \qquad x' = 2^N x,$$

$$51 \qquad x' = 2^{-N} x = x/2^N;$$

or, verbally, as follows.

50 *Multiply the content of the specified accumulator by 2^N, where N is the number written first in the order.*

51 *Divide the content of the specified accumulator by 2^N, where N is the number written first in the order.*

These descriptions and equations apply to both integers and fractions. The shift-number (N) may be anything from 0 to 127, inclusive; a shift of zero places does not, of course, affect the number being shifted.

The 50– and 51–orders are collectively called the *single-length arithmetical shifts* since they can be used for shifting single-length numbers. Apart from the movement of digits which these orders cause there are a few other effects which have been provided in order to facilitate their use.

If the 50-order is used to shift up a non-zero number then overflow will occur if the shift-number is sufficiently large; thus OVR will be set if $2^N x$ exceeds capacity.† For example, overflow will occur if $x = \frac{1}{2}$ and $N \geqslant 1$, or if $x = -\frac{1}{2}$ and $N \geqslant 2$. The computer maintains the word-length of 39 bits by (a) discarding the N digits shifted up beyond the sign-digit position, and (b) by supplying N extra zeros at the right-hand (least-significant) end of the word. The result of the shift is exactly $2^N x$ provided overflow does not occur; if overflow does take place the result obtained differs from the correct one by a multiple of 2 (on the fractional convention).

The 51–order cannot set OVR but it has two special effects. In order to keep the result arithmetically correct the sign-digit is repeated during the shift, i.e. N copies of the sign-digit are supplied at the left-hand end of the word. For example, if the binary numbers

0.101000... and 1.011000... ,

whose values are 5/8 and –5/8 respectively, are shifted down two places by a 51–order we get

0.00101000... and 1.11011000...

respectively, having the values 5/32 and –5/32.

In general the result of a 51–order will not be exactly $x/2^N$ since significant digits may be shifted out of the word at the right-hand (ls) end. In order to minimize the error the result is rounded.†† The method adopted is to add $\frac{1}{2}\epsilon$ (on the fractional convention) to the number before discarding the unwanted digits; this is equivalent to adding ϵ to the result if the first discarded digit is a one. In this way the error does not exceed $\pm \frac{1}{2}\epsilon$, in fact

$$-\tfrac{1}{2}\epsilon \leqslant x/2^N - x' < \tfrac{1}{2}\epsilon,$$

on the fractional convention. Consequently the result is identical with that obtained from a rounded multiplication by the fraction 2^{-N}, or from a rounded division by the integer 2^N.

For example, suppose we have two fractions u and v in 5.0 and 5.1, and we have to put the fractions $u/16$, $4v$ and $5v$ in 4.0 to 4.2. We must do the shifting in one of the accumulators, say *X*6.

5.0	6 00	u to *X*6
(4)	6 51	divide by $2^4 = 16$
4.0	6 10	$u/16$ to 4.0
5.1	6 00	v to *X*6
(2)	6 50	multiply by $2^2 = 4$
4.1	6 10	$4v$ to 4.1
5.1	6 01	add v
4.2	6 10	$5v$ to 4.2

Note that the order

(1) 3 51

---

† To shift N places the computer shifts one place N times. If at any stage the sign-digit is changed then OVR is set. The shifts take place in the mill, and the computer counts the shifts in the order-register.

†† The situation is analogous to that obtaining in multiplication and division; in all these cases we have a number which is too long to be fitted into one word.

has the effect of halving the content of *X*3, and the order

(1) 3 50

is equivalent to the order

3 3 01,

which doubles *C*(3). The order

(0) 3 50

has no effect.

The orders with functions 52 and 53 are called the *logical shifts;* they closely resemble the orders 50 and 51 respectively and are written in a similar way. These orders simply shift the digits of the word without any of the special effects of the 50– and 51–orders. They are not primarily intended for use on words representing numbers but on words used for special purposes which will be described later. The 52-order is similar to the 50-order, the main difference being that OVR is not affected. The 53-order is generally similar to the 51-order but there is no rounding and no repetition of the sign-digit (*N* extra zeros are supplied at the left-hand end of the word). Thus the orders

(10) 4 52

(10) 4 53

have the effect of replacing the 10 left-hand digits of *C*(4) by zeros. The 53-order may be used to effect an unrounded arithmetical shift down provided the operand is non-negative.

The 52- and 53-orders may be described as follows.

52 *Shift the binary digits of the word in the accumulator to the left (up) N places (N being written in the first position in the order). Discard the N digits which are shifted beyond the ms position; make the last N digits of the word all zeros. Do not affect OVR.*

53 *Shift the binary digits of the word in the accumulator to the right (down) N places (N being written in the first position in the order). Discard the N digits which are shifted beyond the ls position; make the first N digits of the word all zeros.*

The orders with functions 54 and 55 are the *double-length arithmetical shifts.* The shifting always occurs in the double-length accumulator formed from *X*6 and 7; the *X*-address in the order is not used by the computer and we usually write zero here. Thus the order

(6) 0 54

multiplies the double-length number (pq) by $2^6 = 64$, and the order

(2) 0 55

divides it by $2^2 = 4$. Since these shift orders are intended primarily for use with numbers, the overflow-indicator may be set by a 54-order, and the sign-digit is repeated with the 55-order. There is, [illegible] when shifting down with a 55-order.† The double-length shift orders may be

defined by the equations

$$54 \qquad (pq)' = 2^N(pq),$$

$$55 \qquad (pq)' = 2^{-N}(pq) = (pq)/2^N \qquad \text{(unrounded)}.$$

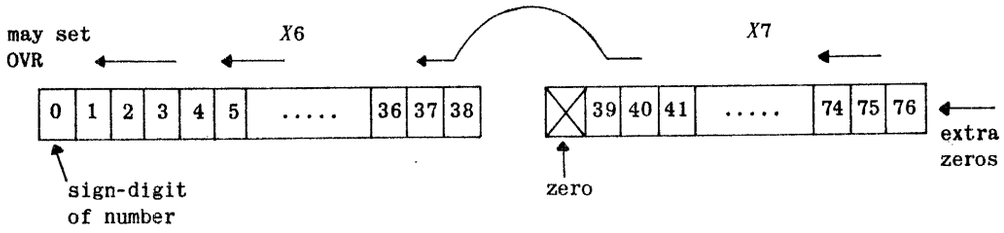
A verbal description of the 54-order is as follows.

careful about the sign of $C(7)$; if there is any doubt then the order

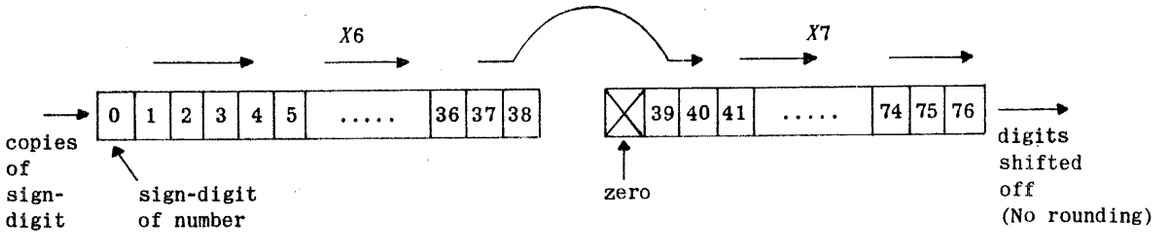
6 0 23 (see Section 3.4)

should be inserted immediately before the shift. It is only rarely that this need concern the programmer since double-length numbers commonly occur only as a result of multiplication; they then always have a non-negative right half.

In the two double-length shift orders the sign-digit in $X7$ does not take part in the shift and is made zero before any shifting occurs.† This digit is by-passed during the shift, so that digit 38 of the double-length number (digit 38 in $X6$) can be thought of as lying immediately to the left of digit 39 (i.e. digit 1 in $X7$). This is shown in Fig.3.4.



The action of the 54-order. Digits shifted up from $X7$ hop over the sign-digit in $X7$ (which is always zero), and appear at the right end of $X6$.



The action of the 55-order. Digits shifted down from $X6$ hop over the sign-digit in $X7$ (which is always zero) and appear in the next position in $X7$.

Fig.3.4 *The double-length arithmetical shifts.*

Since in all the above shift orders the computer operates by repeatedly shifting one place at a time, it follows that a long shift (of many places) is slower than a short shift. In fact the time required by almost any order 50 to 55 to shift N places is just N word-times longer than a simple order such as an 00-order. For example the order

(10) 3 51

requires 10 extra word-times. There is an exception in the order.

(25) X 52

which takes no longer than a simple 00-order. This fast 25-place logical shift up (or Counter-to-Modifier Shift) has a particular application when the technique of modification is used, and is described further in Section 5.5.†† In a 52-order, where the shift is of N places, $N \geqslant 25$, there will be a slow shift of $N-25$ places followed by a fast shift of the remaining 25 places. In this case the order will take $N-25$ extra word-times, for $N < 25$, the order takes N extra word-times.

Most computers are provided with shift orders because they require little extra equipment and are very useful. Since Pegasus is a binary machine the arithmetical shifts correspond to multiplication and division by powers of 2; and it is perhaps when using a shift order that one is most conscious of the binary nature of the machine's operations. Apart from this numbers often have to be doubled or halved. Longer shifts are very useful if the scaling of numbers is done in powers of 2. In addition to their purely arithmetical applications, shifts are indispensable for the so-called logical operations, in which words are treated as strings of binary digits which may have no numerical significance but which are used to represent all kinds of information.

As an aid to remembering the shift orders we may note that the even orders (50,52,54) shift up and the odd orders (51,53,55) shift down. The 56-order is called the normalize order and is described in Section 9.1. The 57-order is described in chapter 11.

3.8 Jumps, the orders of group 6

The vital role of *jump orders* (sometimes called test, discimination, control transfer or branch orders) in even very simple programmes should be clear from the illustrations of Section 1.5. In a

---

† The sign-digit of $C(7)$ is *not* cleared by a null double-length shift, i.e. one in which the shift-number is zero, such as

(0) 0 54

or (0) 0 55.

After such an order $C(6)$ and $C(7)$ are unaltered.

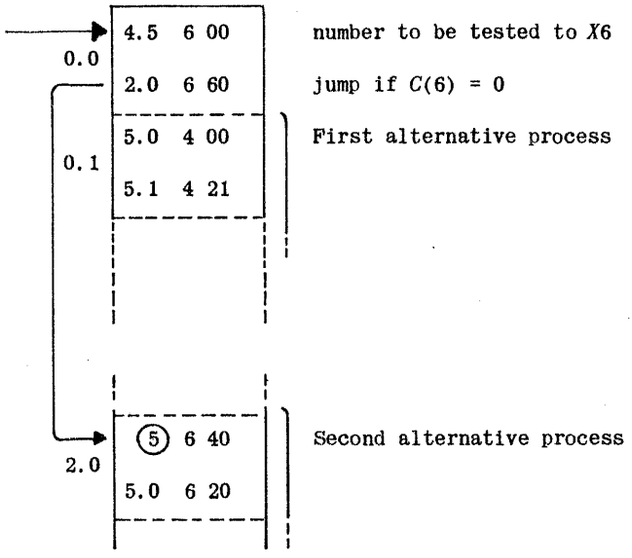
†† This shift order is not fast in Pegasus 1.

typical calculation there are many points at which a choice has to be made between a number of alternative procedures; which one of these is selected depends on results obtained earlier. A multiple choice can always be broken down into a number of simple two-way alternatives and the selection at each of these can be made to depend on a very simple yes-no attribute of some number, such as its sign (negative or not) or its zero-ness (i.e. whether it is zero or not). In Pegasus the orders are normally obeyed sequentially and the jump orders provide the possibility of interrupting the regular sequence and causing the computer to start obeying orders elsewhere. This jump is usually conditional; the content of one of the accumulators may be tested in a certain way and the jump occurs if the test is satisfied, if it is not the computer carries on with the next order as though the jump order had not been there.

The first four orders of group 6 may be used to test the content of any accumulator. Consider, for example, the order

0.7 5 60

which tests $C(5)$. This order will cause a jump if $C(5)$ is zero, in which case the computer will start obeying orders from the a-order in 0.7. As an illustration, let us suppose we are in the middle of a programme and we have two alternative processes to be followed; the second of which is to be chosen only if a previously calculated number stored in 4.5 is zero. The following sequence shows how this might be written.



The number to be tested must first be put into one of the accumulators, here $X6$ is used. The b-order in 0.0 will cause a jump to the a-order in 2.0 if $C(6)$ is zero, and in this case the computer obeys orders sequentially from 2.0 onwards. If $C(6)$ is not zero no jump occurs and the computer carries on with the a-order in 0.1 as usual. It is customary to draw an arrow, as in the above illustration, to show the path of a jump; these arrows are very helpful when one is studying a programme.

We may wish to jump to a b-order instead of an a-order; in this case we simply write a + sign after the address. For example, the order

2.0+ 4 60

will cause a jump to the b-order in 2.0 if $C(4) = 0$. This way of writing addresses is used with all the orders of group 6, and is not normally used with any other orders.

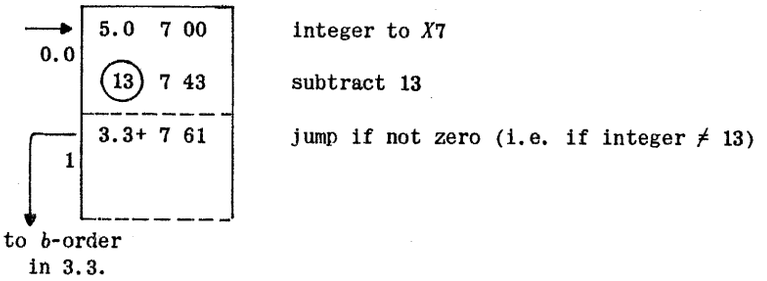
The first four orders of group 6 may be summarized as follows.

60	Jump to N if $x = 0$	(jump if number in accumulator is zero)
61	Jump to N if $x \neq 0$	(jump if number is not zero)
62	Jump to N if $x \geqslant 0$	(jump if positive or zero)
63	Jump to N if $x < 0$	(jump if negative).

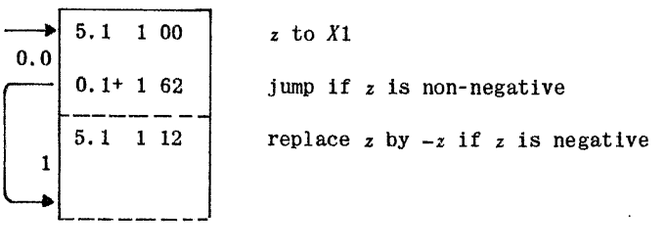
In this description N stands for the a-order or b-order specified in the way described above. The jump may be to any order held in one of the *ordinary registers* in the computing store; there is no provision for obeying orders anywhere else (e.g. in the accumulators or the main store).

It will be noted that these orders can be grouped into pairs with contrary sense (60 and 61; 62 and 63). If, for example, the order 0.2 7 60 causes a jump under certain circumstances then the order 0.2 7 61 will, in the *same* circumstances, not cause a jump, and vice versa. This pairing of the jump orders is a great convenience to the programmer. The 62- and 63-orders test the sign-bit only.

As an example, suppose we have an integer in 5.0 which should be 13; if it is 13 we take no special action and carry on with the programme; if, on the other hand, the integer is not 13 we wish to obey a special sequence of orders starting at the b-order in 3.3.



As another example, suppose we have a number z in 5.1 which may be negative, and if so we have to replace it by its absolute value ($-z$, which will of course be positive).†

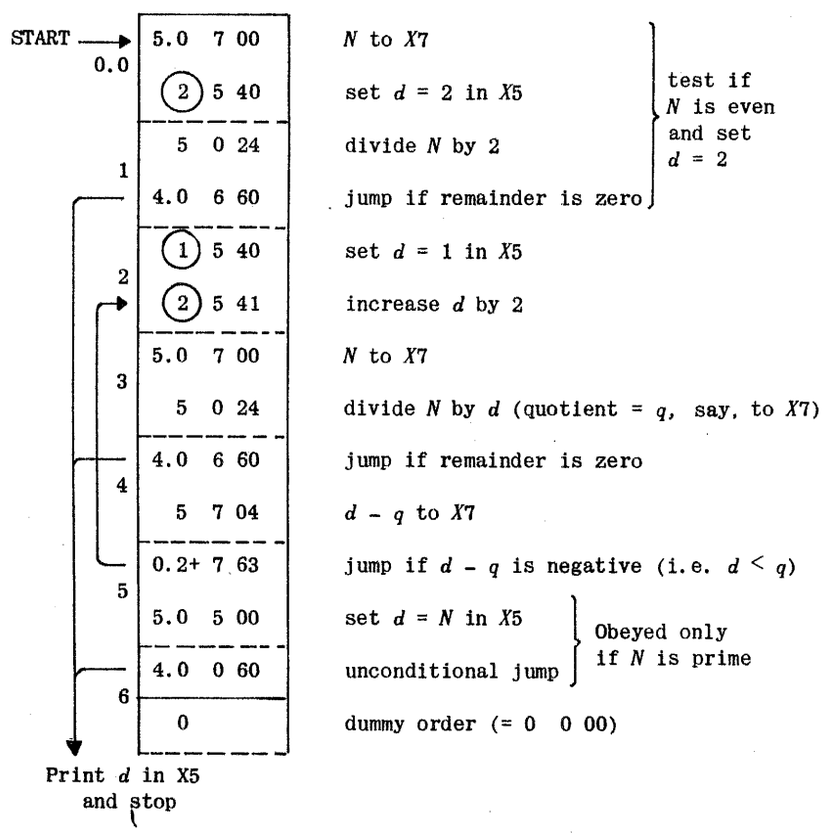


Sometimes we want to jump unconditionally; there are several ways of doing this but the one conventionally used is to test $X0$ to see if its content is zero, which it always is, of course. Thus to jump unconditionally to the a-order in 3.0 we would write

3.0 0 60

and this order would be underlined on the programme sheet to emphasize the break in the sequence of orders. Such unconditional jumps may be used, for example, to prevent the computer from "obeying" numbers or other constants.

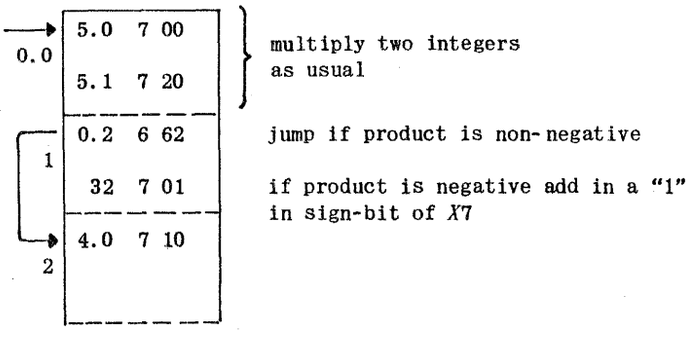
Consider now the problem of finding the smallest prime factor of a positive whole number N. This problem was discussed in Section 1.5 (Example (C)) and a flow-diagram of a possible process is given in Fig. 1.4. In order to write a programme corresponding to this flow-diagram let us assume that N is stored in 5.0; we shall use $X5$ to hold our trial divisor d. We ultimately require to print this and stop; and, since we have not yet described how this can be done, we shall simply assume that the necessary group of orders starts in 4.0 in the computing store.



† The absolute value of z is called the *modulus* of z and is written $|z|$. For example, the absolute values of ½ and −½ are both ½ so we can write

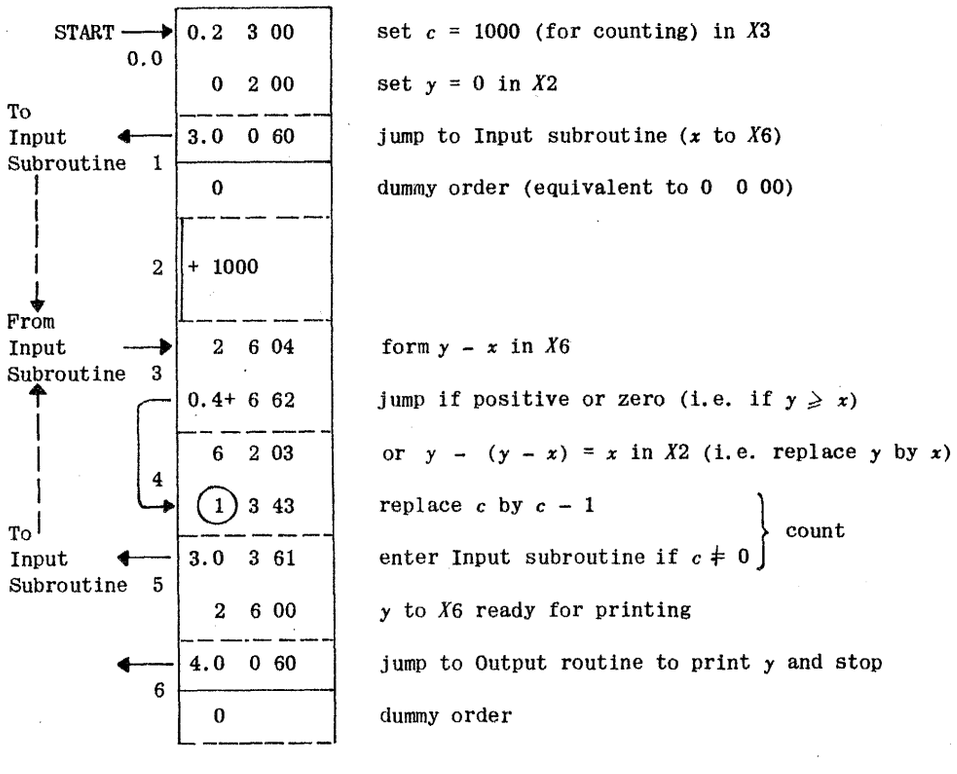
$|½| = |-½| = ½$. Clearly $|7| = 7$, $|-12| = 12$.

It was pointed out in Section 3.1 that the product of two integers can be put into *X*7 by using a 20-order. This product will be correct provided it is not negative and does not exceed about 250 000 million. We can now show how negative products of "small" integers can be correctly obtained. The full double-length product in *X*6 and 7 is always correct; but if the product is "small" and we take it from *X*7 we must insert the missing 1-digit in the sign-position in *X*7 if the product is negative (i.e. if *C*(6) is negative), but not if *C*(6) is positive or zero. Suppose we have two integers in 5.0 and 5.1, and we have to place in 4.0 their (possible negative) single-length product.



Special register 32 is used to provide the missing 1-digit. It will be seen that we have to insert two extra orders if the product can be negative: by use of a trick which we shall describe later (Section 5.10) we can do this with the aid of only *one* extra order.

As a further example of the use of jumps let us consider the flow-diagram of Fig. 1.3 (Example (B) of Section 1.5); this programme is designed to read in 1000 positive numbers via the input equipment, select the largest of them, print it and stop. Since we have not yet described how input and output are done, we shall assume that the group of orders needed to read in a single number is in the computing store, and that all we have to do to read in a number is to jump to 3.0: we shall suppose further that this group of orders, which may be called the *input subroutine* (this subject will be discussed in Chapter 6), places the number read in (x, say) into *X*6 and then jumps to 0.3. In the same way we assume that in order to print a number we need only put it in *X*6 and jump to 4.0, where the necessary group of orders (the *output routine*) starts; these orders cause the computer to stop when it has completed the printing. The quantity y, which at each stage is the largest number read in thus far, will be stored in *X*2. The integer c, which is used to count the numbers as they are read in, will be stored in *X*3. We assume that these are not disturbed by the input subroutine.



The *a*-orders in 0.1 and 0.6 are unconditional jumps. The *b*-orders in 0.1 and 0.6 are dummy orders inserted to make up the order-pairs; any orders would have done here but we have used the conventional 0 0 00 order (which may be written simply as 0). The two orders labelled "count" should be noted particularly; the content of *X*3 is initially 1000 and these orders reduce it by 1 each time a number is read in, the result being tested; eventually *C*(3) will be reduced to zero and the jump will not occur. We shall see later that these two orders can be replaced by one order (with function 67).

When one is sketching out a sequence of orders which includes a number of jumps it is often impossible to fill in immediately the *N*-addresses of some of the jump-orders; these can be left blank and written in afterwards. The use of arrows to show the paths of the jumps is of especial value in helping one to insert the correct addresses.

The orders with functions 64 and 65 test the *overflow-indicator* (OVR). Either of these orders clears OVR, even if no jump occurs; they may be defined as follows.

64	Jump to *N* if OVR is clear;	clear OVR.
65	Jump to *N* if OVR is set;	clear OVR.

The *X*-digits of the order are not used and we usually write zero here.

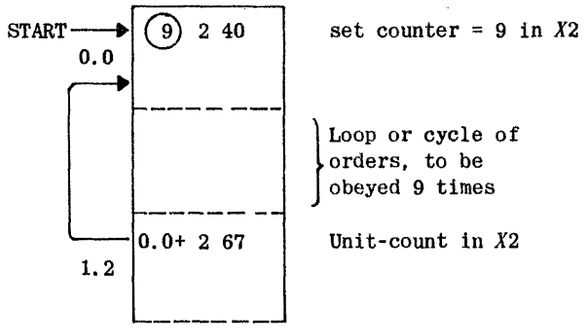
For example, the order

3.6+ 0 65

will cause a jump to the *b*-order in 3.6 if the overflow-indicator is set; after the order has been obeyed OVR will be clear, whether the jump took place or not. If we except writing into the main store (when the computer stops if OVR is set), the only orders affected by the setting of OVR are the two jump orders just described and the justify order (function 23, see Section 3.4); any of these orders will clear OVR. The overflow-indicator can be thought of as a very "sensitive" device which clears itself whenever it is looked at. There is no other order which clears OVR, though it can of course be set in a number of ways.

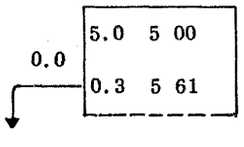
It is impossible to lay down any universally applicable course of action should overflow occur in the course of a programme. Sometimes overflow will not matter, in which case OVR may be cleared by, for example, a 64- or 65-order jumping to the next order. Sometimes it may be possible to adjust the sizes of certain numbers and repeat a part of the programme. Sometimes the only sensible action is to print some information and stop. The choice of a suitable course of action is bound to be considerably affected by the details of the particular problem.

The orders with functions 66 and 67 are used mainly in connection with the facility of modifying orders and will be described in detail later. The 67-order has, however, other uses and we shall therefore give now a brief description of it which is not quite precise. The 67-order is called the *unit-count order;* roughly, its effect is to subtract one from a count-number, or *counter,* in the specified accumulator and then test the result; if this is *not* zero a jump occurs to the order specified in the *N*-address. For example, suppose we have a small group of orders which have to be obeyed a definite number of times, say 9 times. The relevant part of the programme could be written as follows.

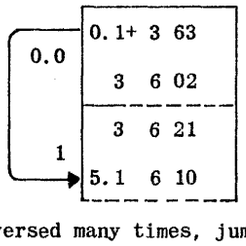


Here the counter in *X*2 is initially set equal to 9, just before entering the cycle of orders which have to be obeyed 9 times. At the end of the cycle is a 67-order which reduces the counter by one at each repetition, and causes a jump back to the beginning of the cycle until the counter has been reduced to zero, when no jump takes place and the computer carries on with the next stage of the programme. On examination it will be found that the cycle will be traversed exactly nine times. Of course, any accumulator (other than *X*0) may be used for counting in this way provided its content is not disturbed during the cycle (or, of course, the counter can be temporarily stored elsewhere at the beginning of the cycle and reset in *X*2 just before the end).

▼ The time taken by any jump order is usually the same as the time for a simple 00-order, i.e. 3 word-times if it is an *a*-order or 2 if it is a *b*-order, whether the order causes a jump or not. If however there is a jump to a *b*-order (i.e. the address in the order ends with a + sign and the jump actually occurs) then the computer obeys a dummy *a*-order just before obeying the *b*-order, such a jump therefore takes 3 extra word-times. For example, consider the sequence:



Here the two orders shown take a total of 5 word-times, whether or not the jump occurs. In the following example there is a jump to a *b*-order; if the jump takes place the total time is 8 word-times, if there is no jump the time is 23 word-times (N.B. a 21-order needs 13 extra word-times).



▲ In a loop which is traversed many times, jumps to *a*-orders are to be preferred.

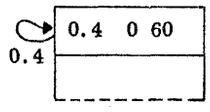
3.9 Stopping the computer

We may wish to stop the computer for any one of a number of reasons. For example, the machine may have reached the end of the programme, or we may wish to change the tape in one of the tape-readers or the setting of the handswitches, or perhaps an error has been detected (e.g. unexpected overflow, or a mis-punched tape).

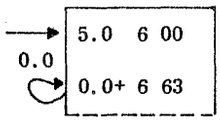
The computer can be stopped manually at any moment by means of one of the keys (or switches) on the control panel. This key is called the *Run key*, or sometimes, the *Stop/Run* key (it is visible near the centre of Plate 11); it has three positions. When the key is up, in the position labelled RUN, the computer obeys orders successively in the usual way. When the key is placed in the middle position, which is labelled STOP, the computer finishes the order it is currently obeying and then stops; if the key is returned to the RUN position the machine will continue with the next order as usual. If the key is pushed *down* from the STOP position into the position labelled SINGLE SHOT and then released, it will spring back into the STOP position and the computer will have obeyed a single order; by doing this repeatedly the computer will obey the orders of a programme one at a time.†

There are a number of orders which can stop the machine. We have already mentioned the fact that the computer stops if trying to obey an order causing writing into the main store when OVR is set (this is called a *writing-with-overflow* stop). An *unassigned order* also causes a stop. There are special lights on the control panel (Plate 11) to indicate these two kinds of stop (and some others), and there is a hooter which can be turned on as an additional warning (the key for this is labelled HOOT ON STOP and is mounted on the left of the control panel).

Another order which stops the computer is a jump order which jumps to itself, for example:



When this order is encountered the computer will take its next order from 0.4, i.e. the order will be repeatedly obeyed. It is reasonable to say that the machine has been stopped by this order since there is no change in any stored word. A stop of this kind is called a *loop stop* (or *dynamic stop*); the only way of getting the computer out of the loop is by operating the Start key (this will be described later). Conditional loop-stops are useful; for example, the following pair of orders



will stop the computer only if $C(5.0)$ is negative. Such loop stops are often used to detect errors in the input data (e.g. punching errors or numbers that have the wrong sign). Any of the orders 60 to 64 (but *not* 65) can be used to produce a loop stop. An unconditional loop stop may be used to terminate a complete programme.

Although there is no warning light associated with a loop stop, the fact that the computer has stopped is usually immediately evident. Input and output cease and the monitors show a static display in all registers. In fact the machine seems to be "dead".

Frequently we wish the computer to stop temporarily at a certain point in the programme, for example to allow us to change an input tape, after which we want to allow the machine to continue. One way of doing this is to use the *77-order*. In this order only the function digits matter, so it is is usually written:

0 0 77

When obeyed it stops the computer and lights one of the special warning lamps on the control panel (and sounds the hooter if it is on). The machine can be caused to continue with the next order by moving the *Run* key to STOP and then back to RUN; this process is called *operating the Run key*. The stop caused by this order is essentially a temporary one; it is usually called a *77-stop*.

Another kind of stop is the *optional stop*. In Section 2.6 we described how the 39 bits of an order-pair are allocated; 19 bits are used for each order. The bit corresponding to the sign-digit in a number-word belongs to neither of the orders; it is called the *stop/go digit*, and can be used to stop the computer. It is usual for this digit to be a one, in which case the order-pair is called a *go order-pair*; the computer obeys *go* order-pairs in the usual way as has been described. If the stop/go digit is a zero then the order-pair is called a *stop order-pair* and the computer will normally stop just before obeying either of the orders. When it stops the order-pair will just have entered the order-register. There is a key on the control panel labelled INHIBIT OPTIONAL STOP; when this key is down the computer will disregard the stop/go digit and will treat all order-pairs as go order-pairs. This is why the stop is described as optional. When the computer has stopped optionally in this way it can be made to continue by operating the Run key as described above for the 77-order. There is also a special light (and the hooter) to indicate the stop.

It is usual, when a programme is being developed, i.e. when it is being tested and made to work for most of the order-pairs to be go order-pairs; here and there will be a few stop order-pairs at strategic points. When the computer stops the monitors can be used to check the contents of various registers, or a specially-prepared checking programme can be called in to print out some information.

---

† This facility is intended primarily for use by the maintenance engineers; but it is *sometimes* useful when testing a new programme.

When the programme is known to be correct the optional stops can be inhibited or, alternatively, all the stop order-pairs can be changed into go order-pairs (this is easy to do by using certain special features of the Initial Orders).

On the programme sheet go order-pairs are not specially indicated. A stop order-pair is indicated by writing a full-stop after either the *a*-order or the *b*-order (but *not* both). This full stop must be written after the *M*-address (modifier address) of the order; if one of the orders is modified the full-stop would be written after it; if neither of the orders is modified the full-stop is usually written after the *a*-order (it is customary, and advisable, to write a zero in the *M*-address of the order in such a case). For example, here are two stop order-pairs:

```
0.0 | 5.0  6 00  |
    | 4.0  6 21 4.|
    |-------------|
    | 3.0  6 01 0.|
  1 |             |
    | 6  6 01     |
```

If the computer jumps to a stop order-pair then there will be an optional stop as usual, *even if the jump is to the b-order.*

There are therefore the following three main ways of programming the computer to stop.

(a) Loop Stop. This is used when it is not desired to carry on with the next order; for example at the end of the programme, or because an error has been detected which cannot be put right automatically (e.g. tape punching error).

(b) 77-Stop. This is used when some manual operation may be needed before the computer is to carry on with the next order; for example the handswitch-setting or the input tape may have to be changed. It should not be used unless it is reasonable to continue the programme.

(c) Optional Stop. This is most useful when developing a new programme but has other uses.

The writing-with-overflow stop and the unassigned-order stop normally occur because of a programming error or incorrect data. Like the loop stop they can be cleared only by operating the Start key, which is normally used to call in the Initial Orders to read in further tape.

The computer will also stop if an attempt is made to read tape when there is none in the tape-reader or when the tape is incorrectly positioned (e.g. upside down); this is called an *input busy* stop and has its own special warning light (but the hooter does not sound). The input busy light usually flickers on and off during normal input. Apart from a manual stop (the Run key at STOP), all the other kinds of stop are due to machine faults. For convenience the stops are tabulated in Table 3.1; some of the details will be described in later chapters. The hooter will sound only if it is switched on and if the Run key is in the RUN position.

Kind of Stop	Reason	Warning Light	Hooter	Action to continue
Loop	Programmed stop	No	No	Start Key
77-order	Programmed stop	Yes	Yes	Run key (or Start)
Optional	Programmed stop	Yes	Yes	Run key (or Start)
Writing-with-OVR	Error in programme or data	Yes	Yes	Start key
Unassigned order	Error in programme or data	Yes	Yes	Start key
Stop on Overflow	Programmed indicator or error	Yes	Yes	Run key (or Start)
Run key at STOP	Operator	No	No	Run key (or Start)
Paper tape busy	Operator or computer fault	Yes	No	Ensure that input tape is in reader. If computer fault, hand over to maintenance engineer.
Parity failure	Computer fault	Yes(2)	Yes	Hand over to maintenance engineer.
Magnetic tape failure	Programme, operator or equipment fault	Yes(2)	Yes	Clear or repeat order or hand over to maintenance engineer
Magnetic tape busy	Programme, operator or equipment fault	Yes	No	Clear or repeat order or hand over to maintenance engineer
Card Reader		Yes		Start key (or hand over to maintenance engineer)
Card Reader	Magazine empty	Yes		More cards
Card Reader	Stacker full	Yes		Clear stacker
Card Reader	Wreck	Yes		Hand over to maintenance engineer

Table 3.1 *Summary of stops*

▼ It should be noted that if, due to a programming error, the computer starts to "obey" numbers instead of orders then it will probably stop fairly quickly for one reason or another. In the majority of programmes nearly all the numbers used are positive, so that they appear as stop order-pairs if interpreted as orders; this is why the computer was so designed that a stop order-pair is shown by a zero sign-bit. If the numbers are small and negative then the *a*-order will represent a 77-stop; also many numbers will represent unassigned orders, so the computer is likely to stop when obeying numbers even if optional stops are inhibited.

The "non-existent" ordinary registers 6.0 to 6.7 and 7.0 to 7.7 can all be regarded as storing zero permanently (like the dummy accumulator *X*0 and the unused special registers, see Section 2.9). The number zero, if interpreted as an order-pair, is two dummy orders (0 0 00) and is a stop order-pair. If the computer is obeying orders from U5 and "runs off the end" after obeying the *b*-order in 5.7 it will therefore immediately encounter an optional stop. If optional stops are inhibited the computer will quickly run through all the dummy orders in the non-existent ordinary registers and will then ▲ return to start obeying orders at the *a*-order in 0.0.

3.10 Main-store transfers, the orders of group 7

There are four orders concerned with the transfer of words between the computing store and the main store; these are the orders with functions 70 to 73 and are the only orders concerned with the main store. The transfers may be done either in blocks of 8 words or one word at a time, and they may take place in either direction. Since these transfers involve both stores the numbers written in the *N*- and *X*-positions of the orders are interpreted in a special way, they are not simply the addresses of registers and accumulators. In connection with computers the word *transfer* is used in the sense of *copying* (or *posting*); the place from which the information is taken is *not* cleared and its content after the transfer will be the same as before. The original content of the place into which information is transferred is simply lost and replaced by the new words.

Before describing the transfer orders a summary of the main store addressing system might be helpful. The block-and-position form of the address of a storage location is the one usually used; this is similar to the notation used for the ordinary registers in the computing store. A letter *B* is often written in front of addresses in the main store to prevent confusion with those in the computing store. The blocks in the main store are numbered *B*0 to *B*1023, of which *B*0 to *B*895 are in the non-isolated part of the store. The blocks in the computing store are referred to as *U*0 to *U*5.

The 70-order is the *single-word read order;* it causes the computer to read one word from a specified location in the main store and to place a copy of it in accumulator 1. For example, the order

34 6 70

places in *X*1 a copy of the word in *B*34.6. The block-number in the main store is written in the *N*-part of the order and the position-number is written in the *X*-part. Because the *N*- and *X*-parts of the order are now both parts of a single address it is usual to write a "box" round them; the above order would therefore usually be written

34 6	70

The box is merely a visual aid on the programme sheet; it is not punched on the tape. The word read is always placed in *X*1; this restriction is required because all the binary digits in the *N*- and *X*-parts of the order are needed to specify the main store address. In fact there are only 10 bits in these two parts of the order so that the address written can be from *B*0.0 to *B*127.7 only. These locations will be referred to as the *first part* of the main store (they are the first 1024 locations of the 7168 in the non-isolated part). Access is obtained to the rest of the main store by means of modified orders; these will be described later.

A location in the main store may, if desired, be specified by its decimal address instead of its block-and-position address. Thus the location whose block-and-position address is 34.6 has a decimal address of 34 x 8 + 6 = 278. This kind of address may be written in a single-word transfer order in the *N*-position; a *minus sign* is then written in the *X*-position. For example, the order

278 -	70

is equivalent to the order 34 6 70 and in fact these are merely two different ways of writing the *same* order; they are both converted to the same internal form when read in by the Initial Orders. The decimal address written in the order may not exceed 1023.

The 71-order is the *single-word write order;* the address is written in the same way as in a 70-order. The transfer occurs in the reverse direction; for example, the order

12 4	71

causes a copy of the word in *X*1 to be placed in *B*12.4 in the main store. The previous content of *B*12.4 is lost. As before, decimal addresses may be used, and locations in the first part only of the store may be directly specified. This order causes the computer to stop, light a warning lamp (and sound the hooter) if it is encountered with the overflow-indicator set; this stop occurs instead of the writing operation.

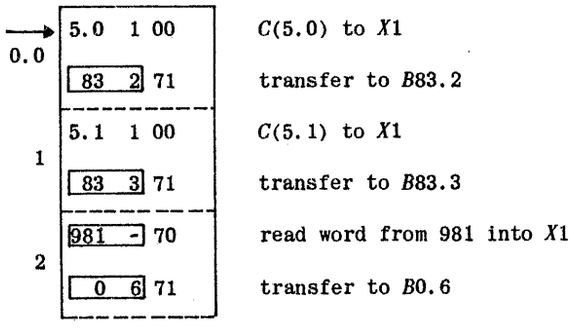
The 70- and 71-orders may be described as follows.

70 *(single-word read) Place in X1 a copy of the word in the main store whose block-number is in the N-part and position number in the X-part of the order.*

† The blocks are numbered *B*0 to *B*639 on the 4096-word store, of which *B*0 to *B*511 are non-isolated.

71 *(single-word write) Place a copy of the word in X1 in the main store location whose block-number is in the N-part and position-number in the X-part of the order; but stop if OVR is set.*

As an illustration suppose we wish to copy into *B*83.2 and *B*83.3 the words in *U*5.0 and *U*5.1 respectively, and copy into *B*0.6 the word in the location whose decimal address is 981:



C(5.0) to *X*1
transfer to *B*83.2
C(5.1) to *X*1
transfer to *B*83.3
read word from 981 into *X*1
transfer to *B*0.6

The two single-word transfer orders are useful for the occasional transfer of odd words. For example, it may take a considerable amount of calculation to arrive at a single number, which could then be transferred to the main-store by a 71-order until it is needed later. Before any transfer can take place the appropriate location must be available; the computer will usually have to wait until the drum has turned so that the location is under the read/write heads. This waiting time may be as long 16 milliseconds but is on the average about half this. It is usually more efficient to use the block-transfer orders, and this is the normal practice.

For blocks 0-15 of the main store, a single-word transfer may take up to 1.25 milliseconds, but an average of .75 milliseconds should be allowed.

The 72-order is the *block-read order;* it may be used to read any block in the main store and place a copy of it in any block in the computing store. For example, the order

51 3 72

causes the eight words in *B*51 to be copied into the corresponding registers of *U*3. After obeying the order the word in *U*3.0 will be a copy of the word in *B*51.0, that in *U*3.1 will be a copy of the word in *B*51.1, etc. The main store block-number is written in the *N*-part of the order (as with the single-word transfer orders); the *X*-part of the order specifies the block in the computing store. In order to emphasize that the *X*-part of the order is not the address of an accumulator we usually write a "box" round it on the programme sheet. The above order would therefore usually be written

51 [3] 72

These boxes are very helpful when one is scanning a programme sheet either to pick out block-transfer orders or to find out when a certain accumulator is used.

The 73-order is the *block-write order;* it is generally similar to the 72-order but writes the content of a computing store block into a main store block. Thus the order

24 [0] 73

stores the eight words of *U*0 in the corresponding locations in *B*24; the original words in *B*24 are lost, but the contents of the registers 0.0 to 0.7 are *unchanged.* If OVR is set when a 73-order is encountered the computer will stop instead of writing.

The 72- and 73-orders may be described as follows.

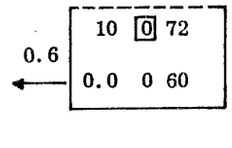
72 *(block-read) Read the block in the main store whose address (block-number) is in the N-part of the order and place a copy of its contents in the computing store block specified by the X-part of the order.*

73 *(block-write) Take the computing store block specified by the X-part of the order and write a copy of its contents into the main store block specified by the N-part of the order.*

With the 72- and 73-orders, as with the single-word transfers, access can be obtained directly only to the first part of the main store (*B*0 to *B*127); we shall describe later how the remaining blocks are transferred by using modified orders.

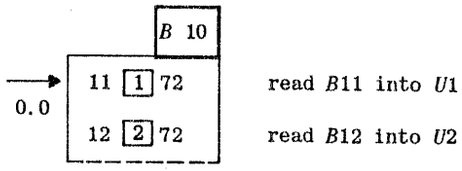
The 73-order is used mainly for writing newly-computed numbers or other information into the main store for later use. The 72-order has two principal uses; it is used to read in new data from the main store which are required in a computation, and it is also used to read in fresh blocks of programme as required. It is the latter, important, application that we shall now discuss a little further.

Suppose the computer is obeying orders in *U*0 and that the next section of programme which we wish the computer to obey is in the main store, in *B*10 say. This kind of situation arises frequently since it is quite exceptional for a whole programme to fit into the computing store. The following orders can be used to read *B*10 into *U*0 and start obeying the new orders at 0.0



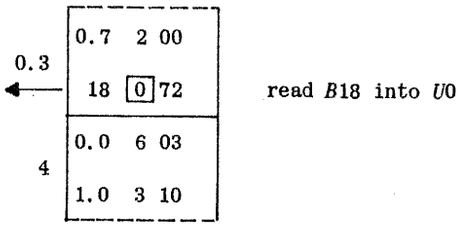
Read *B*10 into *U*0
jump to 0.0

It should be noted that these two orders must form an order-pair, i.e. they must be held in the same register when obeyed (0.6 is used here for illustration). The reason for this is that the computer normally operates by placing an order-pair in its order-register, and then obeying the two orders in it successively before returning to extract another order-pair from an ordinary register (see Section 2.4). When the *a*-order of the above order-pair is obeyed it causes all the words in *U*0 to be over-written by the new words from *B*10. In particular *C*(0.6) is overwritten, but the *b*-order is still in the order-register and is correctly obeyed as written, causing a jump to 0.0. Should we wish to read in more than one block of programme the extra block-read orders can be written at the end of the old block or at the beginning of the first new block; for example, in the above illustration the first two orders of *B*10 could read



The main store block-number (*B*10 here) may be written in the small "box" printed on the programme sheets at the top right corner of each block. It will be seen that the complete programme has to be divided into blocks, at most six of which can be in the computing store at a time (as a general rule there will be only three or four since some space will be needed for numbers). The main aim is to ensure that, so far as possible, speed is not lost by including programme-transfers in the "inner loops" of programmes, i.e. in those loops which are traversed many times and which would be fast if there were no programme-transfers. If possible the division of a programme into blocks should be made logically so that the break between two fairly distinct stages of a programme does not occur in the middle of a block.

Order-pairs consisting of a block-read and a jump as illustrated above occur frequently in most programmes. Occasionally the jump can be dispensed with and the computer can *run-on* into the new block. Consider, for example, the following sequence.



The *b*-order in 0.3 causes the whole of *U*0 to be overwritten, including *C*(0.4), so that when the computer places the next order-pair (from 0.4) in its order-register this will *not* be the order-pair written above but the new order-pair in 0.4 (i.e. that from *B*18.4). The above block read-order resembles an unconditional jump, inasmuch as the next orders on the programme sheet are not obeyed after it, and it is therefore underlined on the programme sheet and marked like an unconditional jump.

There are no ordinary registers with addresses 6.0 to 6.7 or 7.0 to 7.7, although such addresses may be written in orders since there are enough binary digits to specify them. Like the unused special registers (see Section 2.9) and the dummy accumulator *X*0, the non-existent ordinary registers always hold zero. This fact can be put to good use if we require to clear a block in the main store (i.e. replace its content by zeros). Thus the block-write order

62 6 73

will transfer *U*6 to *B*62, i.e. it will clear *B*62. The corresponding 72-order has no effect.

If *U*7 is specified in a block-transfer order then the computer will transfer to or from *the block of accumulators*. This is a very useful facility. At a certain stage in a programme it may be convenient to transfer the contents of the accumulators to a block in the main store, do some auxiliary calculation needing a number of accumulators, and finally to restore the original contents of the accumulators by a block read-order. It is conventional to use *B*0 to store the accumulators in this way. Thus the order

0 7 73

transfers *C*(0) to *B*0.0, *C*(1) to *B*0.1, ..., *C*(7) to *B*0.7; and the order

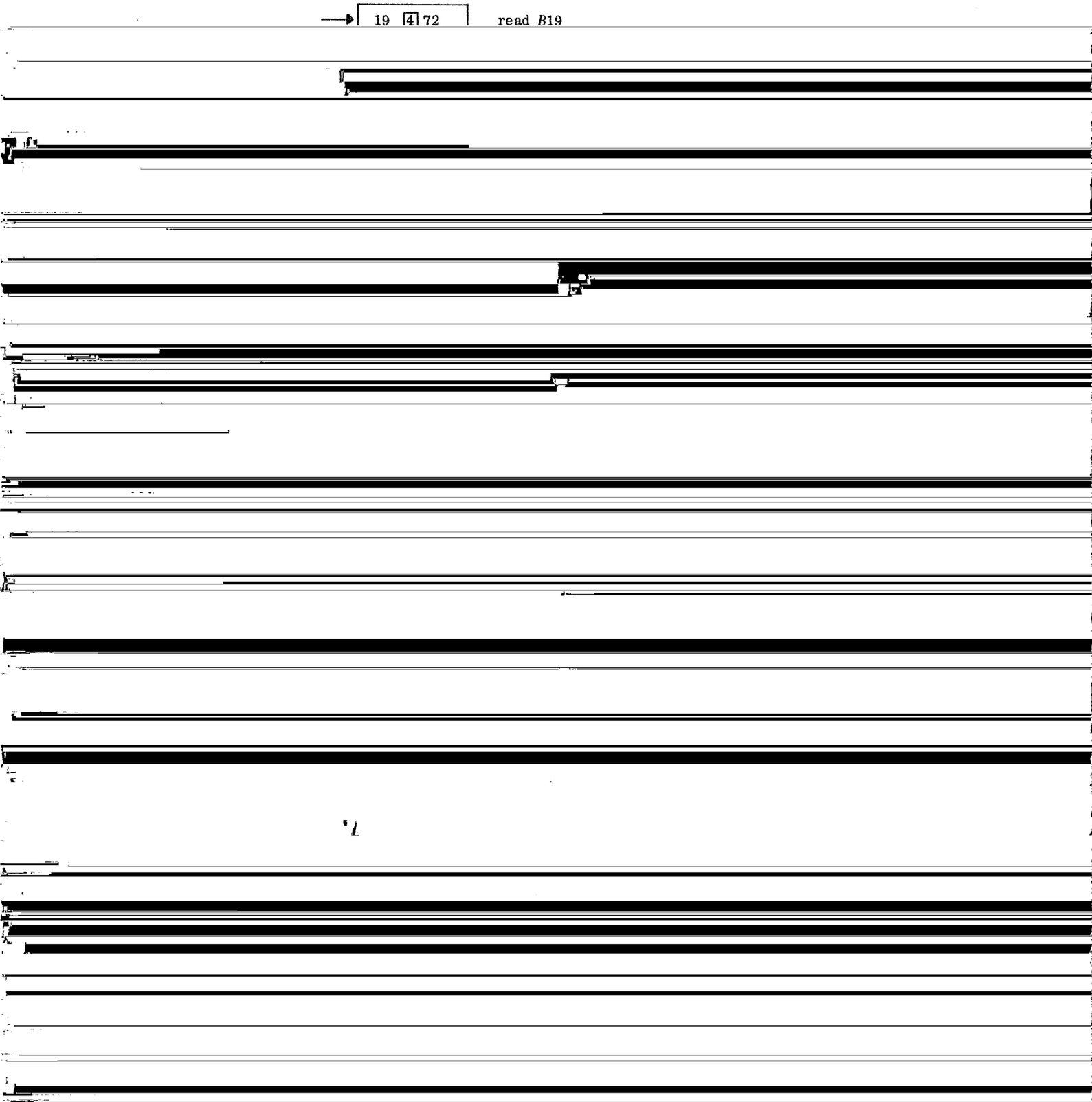
0 7 72

sets all the accumulators from the contents of *B*0 (e.g. *X*4 will contain the word from *B*0.4; any word sent to *X*0 is, of course, lost).

Most programmes can be divided up into fairly well marked stages, and during one of these certain blocks of numbers will probably be read into the computing store, used and replaced by fresh blocks of numbers; there will also, as a rule, be a continual flow of numbers into the main store. Programme (i.e. orders) and numbers are as a rule kept in the computing store only as long as they are wanted; when no longer needed for some stage of a calculation, programme blocks are overwritten by new matter and numbers are transferred to the main store (if they are wanted later). If overflow occurs this is usually a symptom of a programming error (e.g. the scaling of numbers may be wrong or a certain combination of circumstances may not have been anticipated), or it may be due to incorrect data or a

mis-punched data-tape. Since block-write orders are usually frequent the computer will normally stop fairly soon after overflow has occurred. This stop prevents the machine from carrying on with a, perhaps extensive, calculation using wrong numbers; it may therefore prevent the waste of much valuable computer time.

The block-transfer orders, 72 and 73, like the single-word transfers, can be obeyed only when the drum is in the right position, and the computer may therefore be held up before the transfer can occur. As a rule block-transfer orders are to be preferred to single-word transfers since eight times as much information is transferred for the same waiting time. By counting word-times it is possible to find out the exact angular position of the drum at any stage in a programme, and hence to determine the waiting time before any particular word or block can be transferred. This is, however, a laborious undertaking and is not often worth doing. It is usually adequate to allow half a drum revolution (i.e. about 8 milliseconds) waiting time for an isolated transfer order; to this should be added about 1½ milliseconds (for a 72- or 73-order) to allow for the time of the actual transfer. Very often we wish to transfer several consecutively-numbered blocks; in this case we can take the first transfer order to be an isolated one and allow about 9½ milliseconds for it. Subsequent transfers are rather faster though, and we need allow only 3 milliseconds for each of them; this is because consecutively-numbered blocks are arranged in a special way on the drum and become available fairly quickly after the first of them has been transferred (see Appendix 4). In fact the blocks are so arranged that there is always time between the transfer of two consecutive blocks for four simple orders (e.g. 00- or 10-orders) to be obeyed without losing any time. For example, the following sequence will require about 12½ milliseconds (assuming it to start at a random moment).



3.11 Logical operations

In a typical programme a high proportion of the orders will be those transferring information from one part of the store to another, or doing such operations as counting, testing, reading or punching tape, and generally ensuring that the useful arithmetical operations carried out are those appropriate to the data. These operations are often called "administrative" "organisational" or "red tape" operations, and frequently large sections of a programme will be concerned entirely with them. If, instead of programming the calculation for a digital computer, we were to specify it to the operator of a desk machine, many of these organisational operations would not be mentioned; they would be either unnecessary or implied. A computer programme is consequently much longer and more detailed than a sheet of instructions for human use. In order to facilitate the organisational parts of a programme the computer is equipped with a number of special orders. Certain of these are usually called *logical orders*. This term is not well defined; some authorities would apply it to many of the organisational orders, such as the jumps. But we shall restrict it to those orders where the numerical values of the words taking part are only of secondary significance and where the words are primarily thought of simply as strings of binary digits. We have already described the two logical shift orders, functions 52 and 53 (see Section 3.7); we shall now describe the remaining orders of groups 0.1 and 4, which were not included in Sections 2.7 and 2.10. We shall illustrate some of the uses of these orders by examples.

The simplest and most useful of these orders are the *collating* or *and* orders, with functions 05, 15 and 45; they can be used to pick out binary digits or groups of digits in words. They each operate on two words (operands) and produce one new word as the result; this resultant word has 1 digits only in those binary positions where *both* of the operands have 1's. This collating operation is sometimes called *logical multiplication* since the result may be got by a digit-by-digit multiplication. It is best described by an example: the result of collating

u = 0.11110 00011 10011 00010 10101 01111 10110 110

with v = 0.11000 11000 10101 10100 00000 10110 11111 010

is w = 0.11000 00000 10001 00000 00000 00110 10110 010

A digit of w is 1 only where the corresponding digits of u and v are 1's. The operation may be written symbolically as

$$w = u \,\&\, v.$$

Note that the operation is *symmetrical*, so that

$$u \,\&\, v = v \,\&\, u,$$

for any two words u and v. The operation is also *associative*, i.e., if x, y and z denote any three words, then

$$x \,\&\, (y \,\&\, z) = (x \,\&\, y) \,\&\, z,$$

so that we can leave out the brackets; in fact the word

$$x \,\&\, y \,\&\, z$$

will have a 1 digit only where x and y and z all have 1 digits. In most of the actual applications one of the operands will have a block of consecutive 1's and will have 0's elsewhere; the order will then pick out from the other operand the group of digits corresponding to the block of 1's. For example, if

u = 0.00000 00000 00000 00000 00000 11111 11111 000

and v = 1.10110 11011 10000 00110 10011 01101 00111 001

then $u \,\&\, v$ = 0.00000 00000 00000 00000 00000 01101 00111 000.

In this example 10 of the bits of v have been left unchanged, but the others have all been replaced by 0's. A word (like u in this example) which is used to pick out certain digits in other words is sometimes called a *collating mask* (or *collating constant*).

It is useful to introduce an abbreviated notation which can be used to write down collating masks and the like. We shall write 0^n or 1^n to indicate a group of n 0's or 1's, respectively. The word u above, consisting of 26 0's, 10 1's and 3 0's, can be denoted by $0^{26}1^{10}0^3$. A collating mask consisting of 20 1's followed by 19 0's will be written $1^{20}0^{19}$.

The orders with functions 05, 15 and 45 are analogous to the other orders in their respective groups and may be defined as follows.

F	*Effect*	*Description*
05	$x' = x \,\&\, n$	In the binary digital positions where the word in the register has 1's, leave the digits in the accumulator unchanged; elsewhere replace them by 0's.
15	$n' = n \,\&\, x$	In the binary digital positions where the word in the accumulator has 1's, leave the digits in the register unchanged; elsewhere replace them by 0's.

F	*Effect*	*Description*
45	$x' = x \,\&\, N$	In the binary digital positions where the integer written first in the order has 1's, leave the digits in the accumulator unchanged; elsewhere replace them by 0's (N.B. the result can have 1's only in the 7 right-hand digits).

None of these orders can cause overflow.

As an illustration suppose we wish to divide into two parts the word a in 5.0; the left half (b) is to consist of the first 30 bits of a, and the right half (c) is to consist of the last 9 bits of a. We need a collating mask having 30 1's followed by 9 0's (i.e., $1^{30}0^{9}$)

1.11111 11111 11111 11111 11111 11110 00000 000

which has the numerical value -2^{-29} on the fractional convention, or $-2^{9} = -512$ as an integer. Let us assume that b and c (made up to complete words by adjoining enough 0's) are to be put into 5.1 and 5.2 respectively.

```
0.0 | -512          |   collating mask (1^30 0^9)
    |---------------|
--> | 5.0  6 00     |   a to X6
1   | 0.0  6 05     |   replace ls 9 bits of a by zeros, result b.
    |---------------|
    | 5.1  6 10     |   b to 5.1
2   | 5.0  6 04     |   a - b = c
    |---------------|
    | 5.2  6 10     |   c to 5.2
3   |               |
    |_______________|
```

Alternatively the complementary collating mask ($0^{30}1^{9}$, i.e. 30 0's followed by 9 1's) could have been used; this could be written on the programme sheet as the integer +511, since it is $2^{9} - 1$. We shall describe in the next section how other sorts of collating masks may be written.

As another illustration, suppose we have to jump to 3.0 if the integer (k) in 5.7 is odd. Here we wish to examine the last digit of k, which will be 1 only if k is odd.

```
--> | 5.7  7 00     |   k to X7
0.0 |               |
    | (1)  7 45     |   leave last bit of k in X7
    |---------------|
<-- | 3.0  7 61     |   jump if last bit is 1
1   |               |
    |_______________|
```

Some calculations require the manipulation of many small integers. Now a small integer does not require many digits for its representation; for example if all the integers to be handled are known to be between zero and 1000 then they may be represented by only 10 binary digits (because $2^{10} = 1024 > 1000$); if the integers lie between -50 and +50 then 7 bits may be used to represent them (there are 101 different integers and this is less than $2^{7} = 128$). It is obviously wasteful of storage space to use a whole 39-bit word for a single such integer and one may instead *pack* several integers into one word. The collating orders can be used to unpack such a word into its component parts, or to pack up several components into a word. The packing and unpacking operations take time of course, but this may be more than counterbalanced by the more efficient use that can be made of such orders as block-transfers.

If a component of a packed word represents a signed number it is natural to use its left-hand bit to indicate the sign in much the same way as the ordinary sign-bit of a word, i.e. 0 for positive (or zero) and 1 for negative. This can easily be done, but it sometimes simplifies the packing and unpacking operations if the sign-bit is *reversed*, i.e. it is 0 in a negative number and 1 in a positive one (or zero). This corresponds to adding a constant (a power of 2 in fact) to every component, so as to ensure it is non-negative, and packing the components as unsigned; the constant is easily subtracted during the unpacking process. For example, suppose an integer k lies in the range

$$-64 \leqslant k \leqslant 63,$$

so that it can be represented by 7 bits. For simplicity let us assume the integer is stored as $k + 64$ in the 7 ls bits of a word in 5.0; note that $k + 64$ is non-negative. The following orders will place k in $X6$.

```
5.0   6 00
(127) 6 45     k + 64 in X6
(64)  6 43     k in X6.
```

As a rule, when unpacking the components of a word, we require to place each component at the ls end (or, sometimes, the ms end) of a word. The logical shifts are useful for this (their use can eliminate some collating), and they can also be used to shift the various components during the packing process. The double-length arithmetical shifts (functions 54 and 55) can often be used for logical purposes; though one should not forget that OVR may get set by a 54-order, that the sign-bit in *X*6 will be repeated in a 55-order, and that the sign-bit in *X*7 is cleared and does not take part in the shift (see Section 3.7). When using double-length shifts we can often dispense entirely with collating orders. For example, suppose that in 5.0 digits 1 to 9 and digits 10 to 14 represent two unsigned integers which are to be unpacked and placed at the ls end in 4.0 and 4.1 respectively:

```
5.0   7 00
  0   6 00
(9)   0 54     shift first integer into ls end of X6
4.0   6 10     first integer to 4.0
  0   6 00
(5)   0 54     shift second integer into X6
4.1   6 10     second integer to 4.1
```

A similar technique can be used for packing.

The packing of several items of information into a word is not, of course, restricted to integers; the components of such a word may be interpreted in a variety of ways, and may be of different length. Even single binary digits may sometimes be useful; for example, they can be used to indicate whether or not a certain event has occurred; the sign-bit of a word can easily be used in this way since a digit in this position is always available (in special register 32) and its presence or absence in a word can easily be sensed by a 62- or 63-order. Alphabetical information has often to be stored; for example, we may wish to print names of persons or the headings to columns of numbers. Since there are fewer than 32 letters in the alphabet, a letter can be represented by 5 bits (e.g. A = 00001, B = 00010, C = 00011, ..., Z = 11010), though there are some advantages in using 6 bits if other than purely alphabetical information is to be included. In general, any information which is to be represented inside the computer must be encoded in some way.

Another logical operation provided in Pegasus is the *not-equivalent* operation; this is perhaps less useful than the collating process, but has some applications. The orders are those with functions 06, 16 and 46. As with the collating orders there are two operands which are combined, digit-by-digit, to give a result; but in the not-equivalent process the resultant word has a 1 only in those positions where the corresponding bits of the operands *differ*. For example, the result of performing the not-equivalent operation between

u = 0.11110 00011 10011 00010 10101 01111 10110 110

and v = 0.11000 11000 10101 10100 00000 10110 11111 010

is w = 0.00110 11011 00110 10110 10101 11001 01001 100

The process is sometimes called *logical addition* since the result may be got by a digit-by-digit addition (or subtraction) with no carry. Where the digits of v are 0's the digits of w are the same as those of u; where the digits of v are 1's the digits of w are the reverse of those of u. The operation may be written symbolically as

$$w = u \not\equiv v$$

(read as "u not-equivalent v"). As with the *and* operation, this is *symmetrical* and *associative*, i.e. if x, y and z are any three words then

$$x \not\equiv y = y \not\equiv x,$$

and

$$x \not\equiv (y \not\equiv z) = (x \not\equiv y) \not\equiv z,$$

so that we can leave out the brackets. The word

$$x \not\equiv y \not\equiv z$$

will have a 1 digit where either all three operands have 1's or just one of them has a 1 (i.e. where an odd number of the operands have 1's).

The relevant orders may be defined as follows.

F	*Effect*	*Description*
06	$x' = x \not\equiv n$	In the binary digital positions where the word in the register has 1's, reverse the digits in the accumulator; elsewhere leave them unchanged.
16	$n' = n \not\equiv x$	In the binary digital positions where the word in the accumulator has 1's, reverse the digits in the register; elsewhere leave them unchanged.
46	$x' = x \not\equiv N$	In the binary digital positions where the integer written first in the order has 1's, reverse the digits in the accumulator; elsewhere leave them unchanged (N.B. at most the 7 right-hand digits will be changed).

None of these orders can cause overflow.

One of the most straightforward applications of the not-equivalent operation is the comparison of two words, u and v say, to determine whether or not they are equal. If u and v are equal, digit for digit, then $u \not\equiv v$ will be zero. Suppose, for example, that u and v are in 5.0 and 5.1, and we wish to jump to 2.0 if they are exactly equal.

```
5.0  6 00     u to X6
5.1  6 06     u ≢ v to X6
2.0  6 60     jump if u = v
```

Of course we could have subtracted the two words, the result would again be zero only if the operands were equal (even if they have no numerical significance); the disadvantage of subtracting is that overflow may occur.

The order 32 4 06 will reverse the sign-bit in accumulator 4 and leave all the other bits unchanged. Note that if this order, or, in fact, *any* not-equivalent order,† is obeyed twice then there is no change in any word (in this respect such an order resembles an 04, 14 or 44 order, except that these may cause overflow). This may be expressed symbolically as follows, if u and v denote any two words,

$$u \not\equiv (u \not\equiv v) = v,$$

or

$$v \not\equiv (u \not\equiv v) = u,$$

This process is useful if we wish to interchange two words. Suppose, for example, that we have to interchange $u = C(6)$ and $v = C(5.0)$.

```
5.0  6 06     u ≢ v in X6
5.0  6 16     v ≢ (u ≢ v)  =  u in 5.0
5.0  6 06     u ≢ (u ≢ v)  =  v in X6
```

Interchanging can be done in other ways but this method has the advantages that no working space is used and overflow cannot occur.

▼ It is convenient to introduce a special notation to indicate certain other logical operations which can be performed by combining several orders. We shall write 0 to mean a word all of whose digits are zero, and (−1) for the word whose digits are all 1's (these are the values of the words as integers). In the following, u, v and w represent any three words. Clearly

$$u \,\&\, u = u, \quad u \,\&\, 0 = 0, \quad \text{and} \quad u \,\&\, (-1) = u,$$

also

$$u \not\equiv u = 0, \quad u \not\equiv 0 = u.$$

A bar over a letter will denote the *not* operation, i.e. the reversal of all the digits, so that

$$u \not\equiv (-1) = \bar{u}, \quad u \,\&\, \bar{u} = 0, \quad u \not\equiv \bar{u} = (-1).$$

For example, to reverse all the digits of $C(5.0)$ we could use the orders

```
(1) 6 42     (−1) to X6
5.0 6 16     ū = (u ≢ (−1)) in 5.0
```

Note that $\bar{u} = -u - 1$.

---

† We exclude such orders as 4 4 06 of course.

The *and not* operation can be programmed by using either of the identities

$$u \;\&\; \bar{v} \;=\; (u \;\&\; v) \neq u \;=\; (u \neq v) \;\&\; u,$$

which can easily be proved; it is equivalent to collating with a complementary mask. If, for example, u and v are in 5.0 and 5.1 respectively, we can put $u \;\&\; \bar{v}$ in $X6$ by the orders

```
5.0  6 00    u
5.1  6 05    u & v
5.0  6 06    u ≠ (u & v)   =   u & v̄
```

The last order here could be replaced by an 04-order.

The *or* (or *mix*) operation is sometimes useful, it may be denoted by

$$u \vee v;$$

the resulting word has 1's where either or both of the operands have 1's. It can be programmed by using one of the identities

$$u \vee v \;=\; (u + v) - (u \;\&\; v),$$

$$u \vee v \;=\; (u \;\&\; \bar{v}) \neq v \;=\; \{(u \neq v) \;\&\; u\} \neq v \;=\; (u \;\&\; v) \neq u \neq v,$$

which can be easily proved. For example, if u and v are in 5.0 and 5.1, the following orders will put $u \vee v$ in $X6$.

```
5.0  6 00    u
5.1  6 06    (u ≠ v)
5.0  6 05    (u ≠ v) & u
5.1  6 06    u ∨ v
```

As another example, suppose u is in $X6$ and v in 5.0, and we have to mix the last 5 bits of u into 5.0, i.e. we wish to insert in the last 5 bits of $C(5.0)$ any 1's there may be in the last 5 bits of $C(6)$ without disturbing any other digits. The result may be written

$$v \vee (u \;\&\; 31),$$

and may be programmed in the form

$$[\{(u \;\&\; v) \neq u\} \;\&\; 31] \neq v.$$

```
5.0  7 00    v to X7
  6  7 05    u & v
  6  7 06    (u & v) ≠ u
 (31) 7 45   {(u & v) ≠ u} & 31
5.0  7 16
```

The algebra of these logical operations is called *Boolean algebra;* it has applications in the logical design of digital computers. Some elementary results are quite useful; the reader may like to consider the following relations.

$$u \;\&\; (v \vee w) \;=\; (u \;\&\; v) \vee (u \;\&\; w),$$

$$u \vee_v (v \;\&\; w) \;=\; (u \vee v) \;\&\; (u \vee w),$$

$$\overline{u \;\&\; v} \;=\; \bar{u} \vee \bar{v},$$

$$\overline{u \vee v} \;=\; \bar{u} \;\&\; \bar{v},$$

$$u \neq v \;=\; (u \;\&\; \bar{v}) \vee (\bar{u} \;\&\; v),$$

$$u \;\&\; (v \neq w) \;=\; (u \;\&\; v) \neq (u \;\&\; w),$$

$$u \;\&\; (u \vee v) \;=\; u \vee (u \;\&\; v) \;=\; u.$$

A certain operation that can be used with packed words is called *extract* (though this word is sometimes used for other purposes). This operation combines three operands to give a result; it may be defined symbolically as

$$(u \;\&\; w) \vee (v \;\&\; \bar{w}).$$

Here w can be thought of as a kind of mask; where the mask has 1's the digits of the resultant word are the same as those of u; where the mask has 0's the result has the digits of v. For example, if

$$u = 0.11110\ 00011\ 10011\ 00010\ 10101\ 01111\ 10110\ 110,$$
$$v = 0.11000\ 11000\ 10101\ 10100\ 00000\ 10110\ 11111\ 010,$$
and
$$w = 0.00000\ 00000\ 11111\ 11111\ 11111\ 00000\ 00000\ 000,$$

then the result is

$$0.11000\ 11000\ 10011\ 00010\ 10101\ 10110\ 11111\ 010.$$

This operation can be used to replace one component of a packed word by a new group of digits forming a part of another word. It can be programmed by using the identity

$$(u\ \&\ w)\ \vee\ (v\ \&\ \bar{w}) = \{(u \not\equiv v)\ \&\ w\} \not\equiv v.$$

To prove this we note that $(u\ \&\ w)\ \&\ (v\ \&\ \bar{w}) = 0$, so that

$$(u\ \&\ w)\ \vee\ (v\ \&\ \bar{w}) = (u\ \&\ w) \not\equiv (v\ \&\ \bar{w}),$$
$$= (u\ \&\ w) \not\equiv \{v \not\equiv (v\ \&\ w)\},$$

which follows from one of the *and not* identities. The result is therefore equal to

$$(u\ \&\ w) \not\equiv (v\ \&\ w) \not\equiv v = \{(u \not\equiv v)\ \&\ w\} \not\equiv v.$$

As an example of the use of this operation, let us replace digits 11 to 16 of the word (v) in 5.0 by the corresponding digits of the word (u) in 5.1. We shall need a mask (w) having 1's in digits 11 to 16 and 0's elsewhere (i.e. $0^{11}1^{6}0^{22}$).

0.0	0 0 77 0. 0	mask w (pseudo order-pair)
→ 1	5.0 6 00	v to $X6$ from 5.0
	5.1 6 06	$u \not\equiv v$
2	0.0 6 05	$(u \not\equiv v)\ \&\ w$ in $X6$
	5.0 6 16	$\{(u \not\equiv v)\ \&\ w\} \not\equiv v$ in 5.0

Note that if we add the order 5.1 6 16 we shall have interchanged digits 11 to 16 of u and the corresponding digits of v. The mask is here written as a *pseudo order-pair* since this is a very convenient way of specifying its individual binary digits; a pseudo order-pair may be defined as something which is written (and punched) according to the rules for order-pairs, but which is not intended to be obeyed by the computer. We shall discuss this subject further in the next Section.

There is a useful general theorem in Boolean algebra which can be used to prove or discover identities. We write $f(u)$ to mean a Boolean function of u, i.e. any finite combination of u with other letters and constants effected with the operations *not, and, or, not-equivalent*. The theorem asserts that

$$f(u) = \{u\ \&\ f(-1)\}\ _{v}\ \{\bar{u}\ \&\ f(0)\},$$

and can easily be generalized to include functions of more than one variable. It is the fundamental theorem of Boolean algebra. We can apply it to prove the *extract* identity above by writing

$$f(w) = \{(u \not\equiv v)\ \&\ w\} \not\equiv v,$$

so that $f(-1) = u \not\equiv v \not\equiv v = u$, and $f(0) = 0 \not\equiv v = v$. In this case the theorem shows that

$$f(w) = (w\ \&\ u)\ \vee\ (\bar{w}\ \&\ v),$$

which is the result of the *extract* operation. Note that if two functions $f(u)$ and $g(u)$ are to be proved equal for all u, it is enough to prove that $f(-1) = g(-1)$ and $f(0) = g(0)$.

Each of the orders introduced in this Section requires only the basic time to be obeyed, i.e. 3 word-times if it is an a-order and 2 word-times if it is a b-order.

3.12 Orders in binary, pseudo order-pairs

It is occasionally useful to know the way in which the written form of an order-pair is represented in binary inside the computer. This knowledge is chiefly useful when reading the monitor tubes, or when we wish to write collating masks or certain other constants on a programme sheet for input by the computer.

In Section 2.6 we described the way in which the 39 bits of an order-pair are used. We shall now complete this description by an account of the way the N-address of an order is represented by the 7 bits used for this purpose.

If a decimal number is written in the *N*-position of an order then it is directly represented in binary. Such numbers will usually correspond to one of the following.

(a) an accumulator address
(b) the address of a special register } groups 0, 1, 2.
(c) a small integer in a group 4 order,
(d) a shift-number in a group 5 order,
(e) a main store (or buffer store) block-number in a group 7 order.

If the *N*-address is the address of an ordinary register then the left-hand bit is always 1; the next three bits give the block-number in the computing store and the last three bits the position-number. The following examples should make this clear.

Written form of *N*	Binary digits representing *N*
0.3	1 000 011
2.6	1 010 110
5.2	1 101 010

In a jump-order the address of the register is followed by a + sign if the jump is to a *b*-order; in this case the *N*-address is represented as above except that the left-hand bit is always 0. These examples should clarify this.

Written form of *N*	Binary digits representing *N*
0.3+	0 000 011
2.6+	0 010 110
5.2+	0 101 010

The following are examples of the way the 19 bits of an order correspond with its written form.

Order as written	Binary form inside computer			
	N	*X*	*F*	*M*
(13) 5 42	0001101	101	100010	000
27 [2]72 6	0011011	010	111010	110
0.7 6 21	1000111	110	010001	000
3.2 3 66	1011010	011	110110	000
1.5+ 0 60 4	0001101	000	110000	100

When the written form of an order-pair is being converted to the binary form by the Initial Orders, the way in which the conversion is carried out is not affected by the function parts of the orders. A complete order-pair is assembled by shifting up the *a*-order and adding the *b*-order and stop/go digit.

We can, if we wish, write down "abnormal" forms of orders and get them converted to binary and stored; the following are examples.

"Order" as written	Binary form inside computer			
	N	*X*	*F*	*M*
0.3 6 40	1000011	110	100000	000
7.5+ 2 70 3	0111101	010	111000	011
48 0 36 7	0110000	000	011110	111

Usually orders that are written like this are not intended to be obeyed by the computer. They can however be used to express certain constants, e.g. collating masks, as *pseudo order-pairs*. A pseudo order-pair may be defined as a group of symbols written on a programme sheet and punched in accordance with the rules governing order-pairs, but which is not intended to be obeyed by the computer. A pseudo order-pair is usually used when we wish to specify each binary digit of a constant. In Table 3.2 *(page 57)* are given a few examples of such constants and the way in which they can be written as pseudo order-pairs. Note that the sign-bit (or stop/go bit) is 1 unless the pseudo order-pair is marked as a stop order-pair. For clarity we have written dots for 0's in the binary representation, and written the *a*-order above the *b*-order (this is the way words appear on the monitor tube).

Collating Mask		Pseudo Order-Pair
Abbreviated Notation	Binary	
$0^{10}1^{12}0^{17}$	. ...1 111111 111 11.....	0 1 77 7. 4.0 0 00 0
$1^{4}0^{16}1^{12}0^{7}$	1 111.... ... . 1111111 111 11.... ...	6.0 0 00 0 127 7 60 0
$0^{13}1^{13}0^{13}$	. ...1111 111 111111.	0 0 17 7. 7.6 0 00 0

Table 3.2 *Some representative pseudo order-pairs.*

It is customary to rule a heavy vertical line on the programme sheet to the left of a pseudo order-pair, as with other constants. Sometimes collating masks can be conveniently written as numbers, a few examples appear in the previous Section.

We have described earlier how a dummy order may be written simply as 0 in the *N*-position of the order, the other parts being left blank. In general we can, if we wish, write some integer (unsigned) in the *N*-position and leave the rest of the order blank; such an integer will be stored at the right-hand end of the corresponding *order*. For example the last of the collating masks in Table 3.2 could alternatively be written as follows:

```
127

7.6  0 00 0.
```

When writing a single-word transfer order (function 70 or 71) a decimal main-store address can be written in the *N*-position of the order and a *minus sign* in the *X*-position (see Section 3.10). The effect of such a minus sign is to cause the preceding part(s) of the order to occupy 3 extra binary digits to the right (i.e. to be shifted down 3 places). If desired, up to four minus signs may be written in an order, each occupying the space reserved for an octal digit in the written form of the order (i.e. *X*, either of the *F*-digits, or *M*). This facility is useful primarily in pseudo order-pairs used with modified orders and the *Assembly* part of the Initial Orders, which will be described later. The following are illustrations.

Order as written	Binary form inside computer			
	N	*X*	*F*	*M*
31 - 71	0000011	111	111001	000
3 7 71	0000011	111	111001	000
31 - -0 1	0000000	011	111000	001
31 - 01 -	0000000	011	111000	001.

Chapter 4

Some Simple Programmes

In order to put the subjects so far discussed into their proper relationship with one another we shall now describe some simple complete programmes. Before doing this we shall discuss briefly certain subjects which will be more fully covered in later Chapters.

4.1 Outline of output

The output device in a basic Pegasus installation is a paper tape punch (Plate 4). When certain orders are obeyed by the computer this punch perforates a row of holes across a blank strip of tape, and moves the tape forward a tenth of an inch in readiness for the next row of holes. Each row of holes is called a *tape-character;* there is room across the tape for up to 5 holes; each of the 32 possible combinations of hole and no-hole is regarded as a distinct character. The tape produced by the punch is called the *output tape,* in order to distinguish it from the *input tape,* which is the tape read by the input tape-readers. The output tape normally goes immediately into an *interpreter,* which senses the holes produced by the output punch and prints certain characters (e.g. decimal digits or letters) corresponding to the tape characters. When it has been printed out in this way the output tape is often no longer needed and is thrown away. There are two main reasons for using this indirect procedure for printing the results of a calculation:

(a) the punch is about six times as fast as the printer or interpreter,

(b) it is sometimes useful to feed the output tape back into the computer, i.e. to use it as an input tape, on a later occasion.

To avoid circumlocution it is common to talk of the computer printing its results, whereas strictly it only punches them for subsequent printing by an interpreter.

In order to print a number we usually have to cause the printer to print several characters; the individual decimal digits must each be printed and we shall probably have to include a few extra characters such as a sign (+ or −) or some spaces. Each character has to be printed as a separate operation.

For our present purposes we can think of special register 16 (see Sec. 2.9) as connected to the printer. In order to print any particular character all we have to do is to send a certain small integer to register 16. For example to print the character 5 the computer could obey the orders

```
(5) 3 40      5 to X3
16  3 10      print 5
```

The integer we must send to register 16 to cause any particular character to be printed is called the *value* of the character. The value of the character 5 is 5; the value of the character + is 10. The characters we shall need in this chapter are tabulated, with their values, in Table 4.1; it will be seen that values up to 9 are allocated to the decimal digits. Some of the characters call for further

Printed character	Value
0	0
1	1
2	2
⋮	⋮
9	9
+	10
−	11
decimal point (⊙)	12
line feed (LF)	13
space (Sp)	14
multiplication sign	24
equals sign	26
carriage return (CR)	30

Table 4.1 *Summary of printer code*

explanation. A decimal point is referred to as ⊙ to prevent confusion, it is of course printed simply as a point. The character *space* (abbreviated to Sp) causes the carriage of the printer to move forward unit distance without anything being printed; its effect resembles that of the space bar on a typewriter. The character *carriage return* (abbreviated to CR) causes the carriage of the printer to return so that the next printing occurs at the extreme left of the line; *line feed* (LF) causes the paper to be moved up one line so that the next printing occurs on a new line. These last two characters are usually printed successively (as CR followed by LF, denoted by CRLF) so that the next printing occurs at the extreme left of a new line; they could logically be combined but have to be kept separate for technical reasons connected with the design of the printer. The characters Sp, CR, LF are sometimes collectively called *layout characters*.

One should not assume that the carriage of the printer is in any particular position at the start of the programme, and consequently most programmes start off with some such sequence as the following.

```
(30)  6 40  }
            }  Print CR
 16   6 10  }

(13)  6 40  }
            }
 16   6 10  }  Print two LF's
            }
 16   6 10  }
```

Note that the first two orders in the above sequence send the integer 30 to register 16 and consequently cause the "printing" of a carriage return (CR). In order to print the integer 30 we should have to print the characters 3 and 0 separately, for example by means of the following orders.

```
(3)   6 40  }
            }  Print 3
 16   6 10  }

 16   0 10     Print 0
```

Of the 39 bits sent to register 16 to print a character only the ls 5 bits matter, the first 34 bits of the word do not affect the character printed.

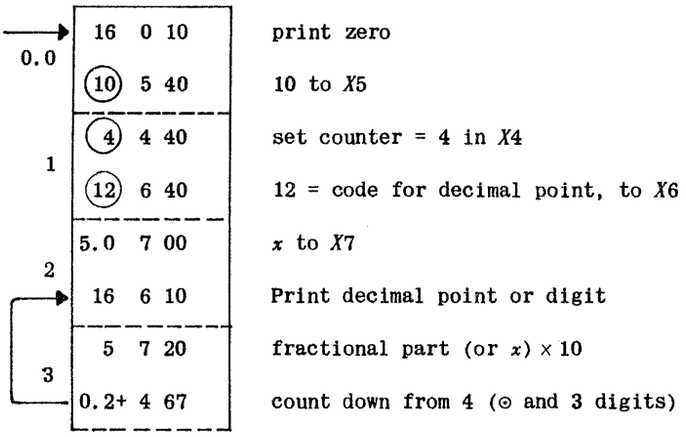
To print a number we have to calculate from the number the values of its individual decimal digits. To illustrate how this is done we shall construct a sequence of orders for printing a non-negative fraction x to 3 decimal places. It is, of course, less than 1.0 and can be written $0.rst$. where r, s and t are its first 3 decimal digits. For example, if x = 0.123456... then r = 1, s = 2, t = 3. We can easily find the value of r, the first digit, by multiplying x by 10; the integral part of the product is just r (e.g. if x = 0.123456... then $10x$ = 1.23456...). After printing r we can then replace x by the fractional part of the product and repeat the process. The following steps should make this clear.

x	=	0.123456...
$10x$	=	1.23456..., integral part = 1, fractional part = 0.23456...
10 × fractional part	=	2.3456..., integral part = 2, fractional part = 0.3456...
10 × fractional part	=	3.456..., integral part = 3, fractional part = 0.456...

After each multiplication the integral part is the next decimal digit which is to be printed. This process requires the multiplication of a fraction by an integer; if we use a 20-order we shall get the integral part of the product in $X6$ and the fractional part in $X7$ (see Sec. 3.1). The following sequence of orders can thus be used to print r, s and t if the fraction x is in 5.0:

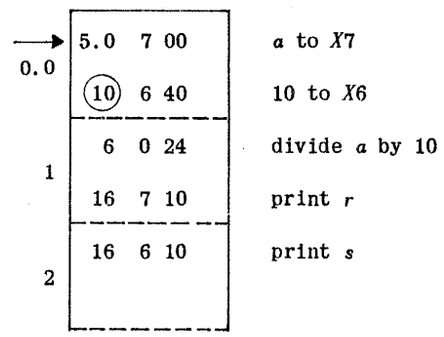
```
(10)  5 40     10 to X5
 5.0  5 20     10x to X6 and 7
 16   6 10     print r
  5   7 20     fractional part × 10
 16   6 10     print s
  5   7 20     fractional part × 10
 16   6 10     print t
```

It would be better to print a zero and a decimal point first and rearrange the above orders into a small loop as follows.

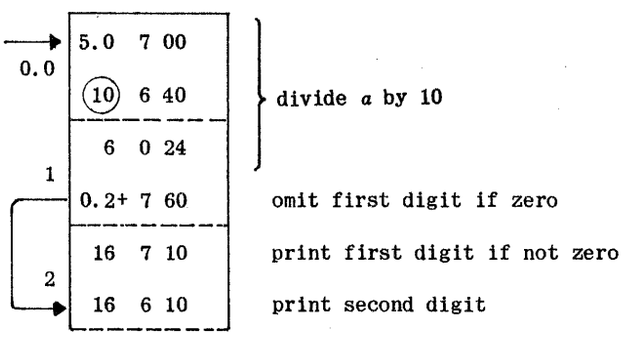


In practice other points would be taken into account. For example, we should print either CRLF or SpSp before the number; and we would probably arrange to print the sign so as to allow x to be negative (if $x < 0$ we must change its sign before entering the above loop of orders). We should also take care to round the value of x before printing (by adding 0.0005 to it) since we are only printing the first few digits of a long number (c.f. the use of rounding in multiplication, division and shifting). We must then consider the possibility of overflow caused by the rounding or the change of sign. When all these points (and others) have been taken into account we get a self-contained piece of programme of moderate complexity. Such a piece of programme is called a *subroutine;* a *routine* is simply another name for a programme, and a subroutine is a fairly independent, self-contained part of a routine. We shall discuss this subject further in the next Section.

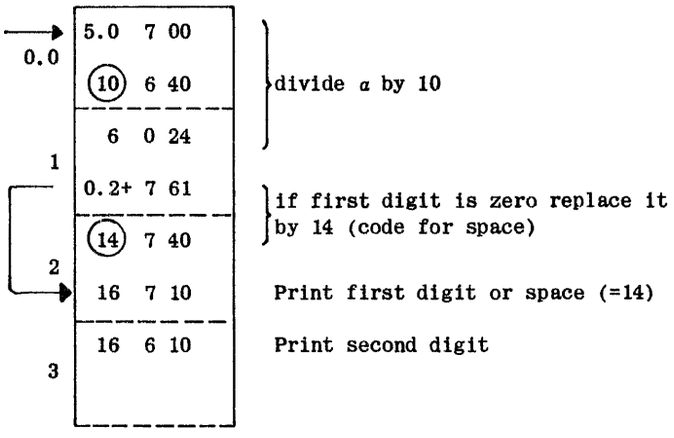
Let us now consider how integers may be printed. To start with we shall consider the printing of a non-negative integer a less than 100. One way is simply to divide the integer by 100 to convert it into a fraction and then to use a cycle of orders similar to the one given above; this method is awkward for larger integers so we shall not consider it further. If r and s denote the two digits of a, then $a = 10r + s$, and we can find r and s by dividing a by 10. For example, the following orders could be used, supposing a to be in 5.0 at the outset.



This sequence would probably not be used in practice since it does not incorporate any suppression of left-hand zeros, i.e. an integer such as 7 would be printed as 07. We must arrange that the first digit is not printed if it is zero. The following sequence shows how this can be done.



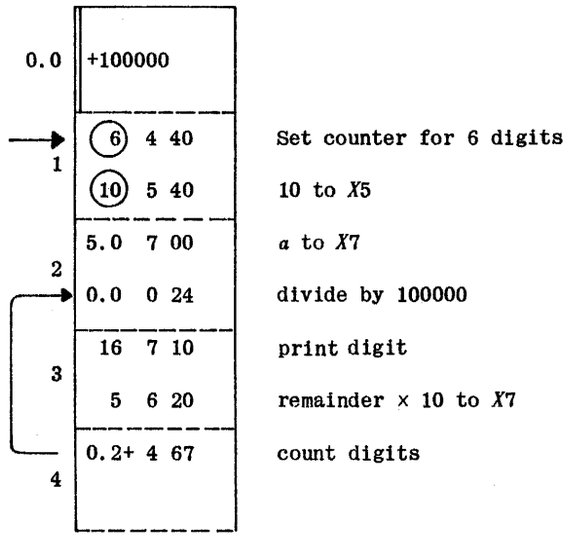
This device is known as left-hand zero *omission*. Even this sequence would not be acceptable in most programmes, since the *width* of the printing is not constant, i.e. sometimes the printing will be two units wide and sometimes only one. This is disadvantageous when several numbers are printed on the same line and similar lines are printed repeatedly: the layout of the printed page is then irregular. To obviate this we must arrange to print a space instead of the first digit if it is zero; this is called zero *suppression*.



Generally speaking we should want to print larger integers than can be handled by the above sequence of orders. To see how this can be done consider the printing of the integer a = 123456. If we divide it by 10 the remainder will be 6 and the quotient 12345; if this quotient is then divided by 10 the remainder will be 5 and the quotient 1234. This process can be repeated, but has the disadvantage of producing the digits (as the remainders) in the reverse order. A better way is to divide a by 100000 to give a quotient of 1 and a remainder of 23456. The quotient can then be printed and the remainder multiplied by 10 to give 234560. This number is in turn divided by 100000 to give a quotient of 2 and a remainder of 34560. The quotient is printed and the remainder multiplied by 10 again, and so on. The following table shows how the successive quotients give the required digits.

	quotient	remainder	remainder × 10
123456 ÷ 100000	1	23456	234560
234560 ÷ 100000	2	34560	345600
345600 ÷ 100000	3	45600	456000
456000 ÷ 100000	4	56000	560000
560000 ÷ 100000	5	60000	600000
600000 ÷ 100000	6	0	0

The following orders will carry this process out for any 6-digit non-negative integer.



As before, we must arrange for left-hand zero omission or suppression, the details of which are left to the reader. In Section 4.4 below we give an example of a sequence for printing single-length positive integers of any magnitude. As when printing fractions there are a number of complications and in general the whole process is best left to a subroutine.

As a rule the printing produced by a programme will consist of a great many numbers, some of these will be printed as integers, some as fractions, some as mixed numbers, some in sterling, some will be signed, etc. It is well to arrange that the printing is neatly laid out in columns, blocks, and so on, and this can be arranged by suitable counting and printing of layout symbols. It should be noted that there is room for printing up to 69 characters on one line of the printer.

In this section we have described the way in which numbers can be printed using the equipment in a basic Pegasus installation. One can also easily arrange to print alphabetical and some other characters, and we describe this in Section 6.2. The normal output punch operates at 60† characters

† Many installations will still be using the slower output punch which operates at 33 characters per second.

per second and this is adequate for most scientific and technical calculations. Sometimes, however, greater speed is necessary and the basic installation must then be supplemented by some ancillary equipment: this is most needed in large-scale commercial and statistical work. Some of the extra input and output facilities are described in Chapters 10 and 11.

4.2 Subroutines and the organisation of a programme

A reader new to the subject could be forgiven if, at this stage, he felt that the task of preparing a complete programme was altogether too difficult to be contemplated. It is true that the preparation of a large and complicated programme is an operation which may take a year or more; but there are many ways of lightening and spreading the labour, and much of it is, by any standards, interesting. Nearly all programmes can be subdivided into self-contained parts, each doing some job which can be regarded as more or less complete and independent: these parts are often called *subroutines*. If a flow-diagram of the whole programme is available then each box, or some convenient group of boxes, can be thought of as a subroutine. The importance of the idea is that, provided a sufficiently precise specification of its function can be written, each subroutine can be programmed and tested separately, without much reference to the rest of the programme. In this way the programme can be tackled piecemeal, if necessary by several people. Afterwards the various subroutines can be linked together to make up the complete programme.

This procedure is of the greatest value in the preparation of a complicated programme. We can visualize a programme organized in this way as a collection of subroutines so arranged that when one has finished its task it leads into (or *calls in*) the next. As a rule there will be a few "red tape" or organisational orders to be obeyed between one subroutine and the next, especially if they have been written by different people, and it is therefore often more convenient to have, in addition to the subroutines, a *master-programme*, (or *master-routine*) which includes most of the organisational orders and whose main purpose is to call in the various subroutines in the right sequence. If the programme is organised in this way, which is usually the case, the subroutines must be so written that they re-enter (or *return control to*) the master-programme when they have finished their work. In addition to calling in the various subroutines, the master-programme usually performs a few other operations, such as counting, moving numbers from one part of the store to another or doing a small amount of arithmetic. It often happens that some of the subroutines may themselves call in other subroutines for some subsidiary calculation (these are sometimes referred to as sub-subroutines) so that the whole structure of the programme may be on several levels as illustrated in the diagram (Fig. 4.1).†

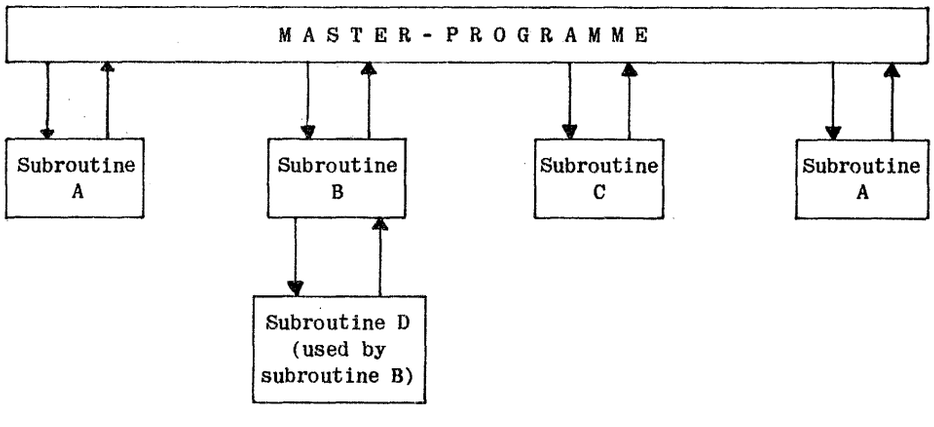
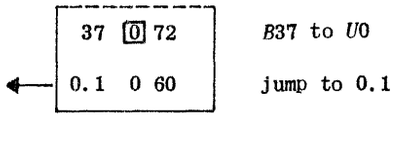


Fig. 4.1 *Structure of a typical programme*

It will be noticed that subroutine A has been called in (or *entered*) twice from two different points in the master-programme.

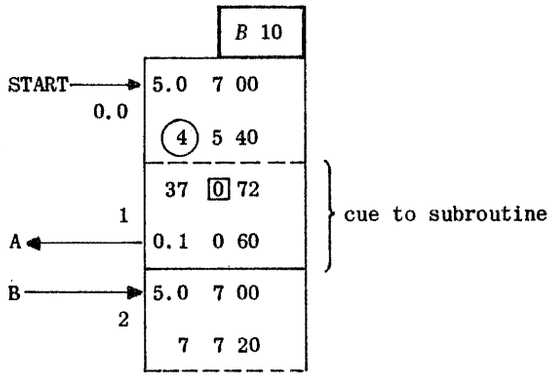
The orders used to call in a subroutine are called the *cue* to the subroutine. A cue is normally an order-pair consisting of a block-read order and an unconditional jump. For example, the following is a typical cue:



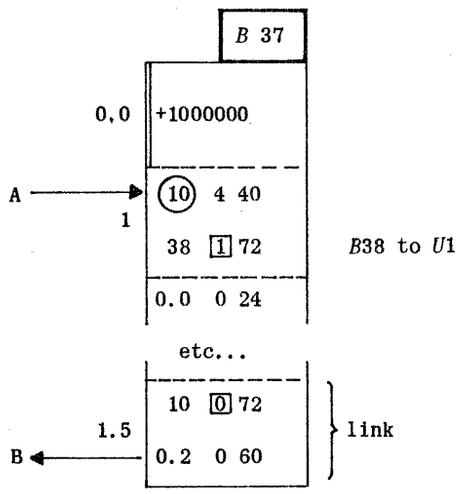
When such a cue is obeyed a block of the subroutine (usually, but not always, the first block) is copied into the computing store and entered (i.e. a jump to one of its orders occurs). After this the computer is obeying the orders of the subroutine, which will include any further block-read orders needed. Eventually the subroutine will have finished its task (e.g. printing a number, evaluating a square-root, reading in a list of numbers, etc.), and it then obeys a *link*, which is the order used to leave (or *exit from*) the subroutine, usually so as to return to the master-programme. The link is often an order-pair made up, like the cue, of a block-read order and a jump.

As an illustration, suppose that there is a subroutine for printing $C(7)$ as an integer which is to be entered by the cue written above. At the relevant point in the master-programme we might have the following orders.

† In this diagram the box representing the master-programme has, of necessity, been made larger than those representing the subroutines. This is not intended to reflect the amount of storage space occupied.

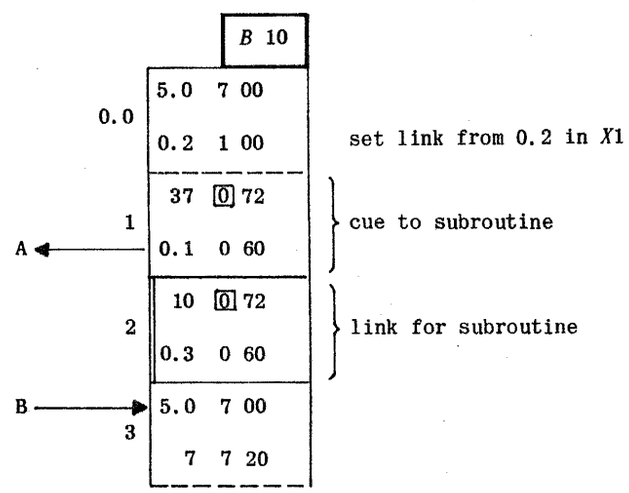


In the subroutine there could be the following orders.

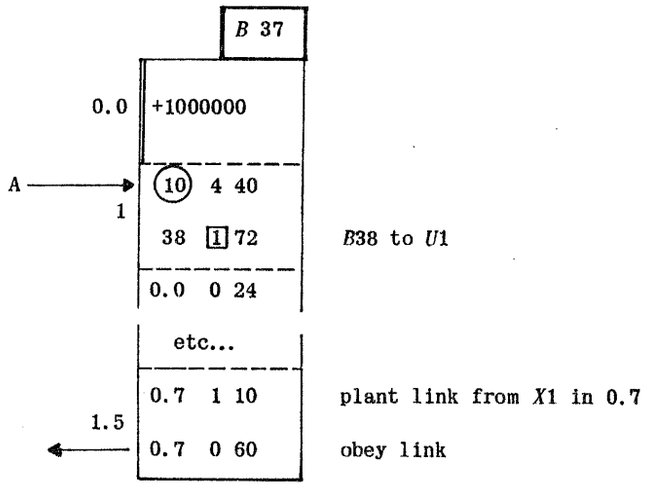


When the link is obeyed in the subroutine the original block (*B*10) of the master-programme is put back in *U*0 and re-entered at the *a*-order in 0.2 (marked B). A subroutine written like this can, of course, be entered from several different places in the master-programme but, since it always obeys the same link, it will on every occasion return to the same point in the master-programme. Frequently this would not be what was wanted and in a programme like that of Fig.4.1 would necessitate the storing of two (or more) copies of a subroutine, each with its own fixed link. It is better to arrange that the link obeyed by the subroutine can be changed so that return to the master-programme is to different points on different occasions.

The usual arrangement is as follows. The master-programme places the appropriate link in accumulator 1 just before obeying the cue which calls in the subroutine; this process is called *setting* the link. The subroutine is written in such a way that, when it has finished its task, it copies the link into an ordinary register (this is called *planting* the link) and obeys it. For example, the master-programme might contain the following sequence of orders.



And in the subroutine there could be the following orders.



It will be noticed that the link is written in the master-programme (in 0.2) immediately after the cue; this is often a convenient place to write it. The link is marked with a heavy line on the left like a constant or a pseudo order-pair; this is to draw attention to the fact that it is not obeyed in the master-programme, where it is written, but in the subroutine (actually in *U*0.7). With this device the subroutine can be entered from several different points in the master-programme; on each occasion the master-programme can set a different link in *X*1. Of course, there is no need to use *X*1 for the link in this way, another accumulator (or an ordinary register) could be used but it is conventional to use *X*1.

It is essential that the subroutine should carefully preserve the link until it is needed. If the subroutine requires to use *X*1 then it should copy the link elsewhere first; this may happen because the subroutine requires a single-word transfer order, or perhaps because it calls in a further subroutine.

When a subroutine has been written it is advisable to prepare a *specification* of it. This is simply a catalogue of the properties and effects of the subroutine as viewed from outside. The specification would state, for example, which blocks in the computing store were used by the subroutine, what cue (or cues) are to be used to enter it, which accumulators it uses as working space, what its precise effects are, and so on. There are a number of conventions relating to subroutines, among which the following are important.

(a) the link is to be in *X*1 on entry,

(b) OVR is to be clear on entry,

(c) nothing should be assumed about the contents of any registers or storage locations used as working space by the subroutine,

(d) if possible, the subroutine should use blocks *U*0 and *U*1 only in the computing store, and also as few accumulators as possible.

All these things are purely conventional. It is important to distinguish between the rules of programming, which are necessary because of the way the computer works, and conventions, which form a body of accepted practice. An adequate set of programming conventions is an essential prerequisite for the satisfactory use of a computer.

It is usually necessary to enter a subroutine by a proper cue as described above (i.e. a block-read and jump) even if the subroutine is used several times in quick succession. This is because most subroutines write over some of their own orders when these are no longer needed; consequently a fresh copy of the routine must be brought in from the main store on each occasion when it is used. Since some subroutines use *X*1 as working space, the link must be set each time the subroutine is used, even if the same link is required again.

The following are some imaginary, but plausible, brief specifications of subroutines.

(1) Print three-digit non-negative integer in *X*7, preceded by SpSp.

Cue:

```
40 0 72
0.0 0 60
```

Uses: *U*0, *B*0.

Link: In *X*1, obeyed in 0.3.

Time: about 100 milliseconds.

(2) Evaluate $\sqrt{(pq)_F}$ and leave result in *X*6.

Cue:

```
43 0 72
0.0 0 60
```

Uses: *U*0; *X*5, 6, 7.

Link: in *X*1, obeyed in 0.7.

Time: about 45 milliseconds.

(3) Read a sum of money in sterling from the input tape, convert it to pence and leave the result in *X*6.

Cue:

```
44 [0] 72
0.1  0 60
```

Uses: *U*0; *X*4, 5, 6, 7.

Link: in *X*1, obeyed in 0.7.

Time: about 108 milliseconds.

Some subroutines may usefully be written so as to have two or more alternative entry-points, each with its own cue. For example, a print subroutine could precede the main printing by CRLF if entered by the cue

```
40 [0] 72
0.0  0 60
```

or by SpSp if entered by the cue

```
40 [0] 72
0.2+ 0 60
```

Sometimes it is advantageous to use a device known as a *programme-parameter*. This is usually a number set in an accumulator by the master-programme just before calling in a subroutine. The effects of the subroutine are then determined to some extent by the parameter. A straightforward example is provided by a subroutine for printing a fraction; the subroutine could be so written that the number of decimal places printed is determined by, say, an integer in *X*2. The number of places required is then set in *X*2 by the master-programme at the time it sets the link in *X*1, i.e. just before obeying the cue to the subroutine. Like the link (which is really only a special kind of programme-parameter), the number of decimals wanted is set each time the subroutine is called in and can easily be changed.

We have already pointed out that the use of subroutines can greatly simplify the task of preparing a complete programme by enabling the whole job to be broken down into parts which can be dealt with more or less independently. There is a further advantage: some subroutines may be used in more than one programme. For example, many programmes will need a subroutine for printing an integer. Such a subroutine can be written by an experienced programmer in such a way as to be efficient and to have features making it widely useful. Many generally useful subroutines like this have been written, they together form the *Library* of the computer and there are volumes available which contain their specifications. The specification of a library subroutine provides all the information needed for the normal use of the subroutine (e.g. details of cues, working space, and so on); it does not indicate, nor need the user know, how the subroutine works. Since the use of the library requires a knowledge of the Assembly section of the Initial Orders, we defer further discussion until Chapter 8.

In most scientific and technical calculations functions of various kinds are needed. For example, square-roots, logarithms, exponentials, sines, cosines, and so on, enter into many calculations. When carrying out a computation by hand it is usual to use books of tables for such functions. With a digital computer one could, of course, store a table, together with a subroutine for looking up entries in the table and interpolating in it. This is sometimes a useful technique but it should be realized that such a table may occupy a considerable amount of storage space; furthermore the interpolation routine is by no means trivial and may be relatively slow. It is usually best to *evaluate* functions as they are needed; for example, sin x may be evaluated by means of a truncated Taylor series

$$\sin x \simeq x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots\ldots + (-)^n \frac{x^{2n+1}}{(2n+1)!}.$$

Only six or even seven terms are needed to give the full precision attainable with 39-bit words, and one can therefore construct a fast subroutine to evaluate sin$x$ (the numerical values of the coefficients being stored within the subroutine). (This is, in fact, not the best way to evaluate sin$x$.)

As a simple illustration of how subroutines can be used let us prepare a table of powers of 2. We shall print two columns giving the values of n and 2^n, respectively, starting at n = 1 and going on until 2^n is no longer a single-length number (the last value will be 2^{37}). We shall assume that a

subroutine is available with the following brief specification†:

(4) Print integer in *X*7 (with zero suppression).

Cues: (1)

50 [0] 72	to precede integer by CR LF
0.0 0 60	

or (2)

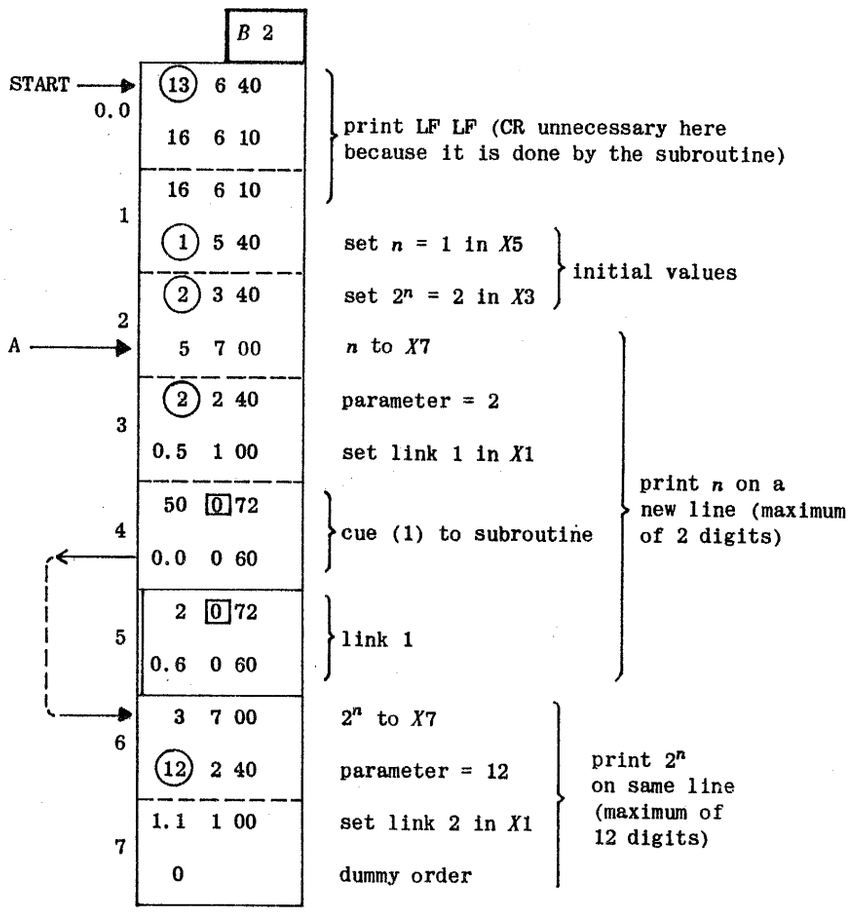
50 [0] 72	to precede integer by Sp
0.3 0 60	

Uses: *U*0, *B*0.

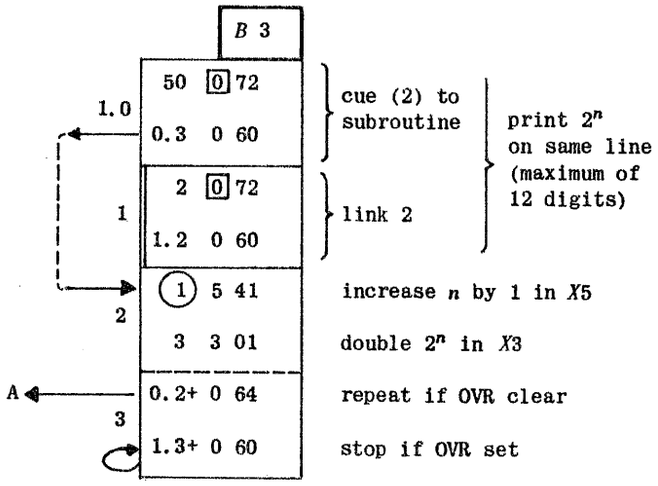
Link: in *X*1, obeyed in 0.3.

Programme-parameter: integer in *X*2 specifies maximum number of decimal digits in number being printed.

We know that n will have at most 2 digits and 2^n at most 12, so we can set the programme-parameter to these values. We shall keep the current value of n in *X*5 and that of 2^n in *X*3. The master-programme is a little under two blocks long; we shall suppose it occupies *B*2 and *B*3 and can be entered by copying these two blocks into *U*0 and *U*1 and jumping to 0.0; we shall describe how this is done, with the help of the Initial Orders, in the next Section.



† It will be noticed that this print subroutine uses only *U*0 in the computing store. This is typical of output subroutines in general, which are usually written so as to preserve all the accumulators by writing them into *B*0 on entry (there is an order 0 [7] 73 at the beginning of the subroutine); the accumulators are reset from *B*0 just before the link is obeyed. The time required by these extra block-transfers (and others in the subroutine) is generally small compared with the time needed to operate the output punch.



Note the dummy order (in 0.7+) inserted to make the second cue to the subroutine an order-pair (although this is strictly not necessary here). On each occasion the link restores *B*2 in *U*0 and this is all we need to do after using the subroutine since it uses only *U*0 in the computing store.

4.3 Putting a programme into the computer

We have so far described the effects of various orders and how they can be written, together with some numbers, on programme sheets. We have also discussed the way in which programmes can be organised with the aid of subroutines, and how the results are printed. Let us now consider how a complete programme and the numerical data for it can be supplied to the computer. We shall suppose that the complete programme (including all the subroutines) is available and that we wish to place it all in specified places in the main store, after which the beginning of the master-programme is to be copied into the computing store and the machine is to start obeying its orders at some specified point.

We must first of all prepare the *programme tape*. This is usually done with a teleprinter forming part of the tape-editing equipment (see Plates 7,10). This can be thought of as a kind of typewriter on which the orders and numbers of the programme can be typed out; as a result of this the teleprinter produces a printed sheet and a length of punched paper tape. With few exceptions, each time one of the keys on the teleprinter keyboard is pressed a single character is printed and the corresponding tape-character (i.e. a certain row of holes) is punched. We shall not at present describe the rules for operating the teleprinter and punching the tape; they are such that the way in which the orders and numbers appear on the printed sheet differs only slightly from the way they are written on the original programme sheets. The punching rules are consequently natural, and easy to learn and apply.

The printed sheet (known as a *print-out*) can be proof-read against the programme sheet, and this is usually an adequate way of checking the punching of the programme. Fig.4.3 shows a programme sheet, and Fig.4.4 the corresponding print-out. The data on which the programme is to operate are punched in much the same way but must be much more carefully checked; it is usual to punch the data twice and to compare the resulting tapes automatically in the *comparator* section of the tape-editing equipment. Should the checking of the tapes reveal any errors in the punching, these can easily be corrected in various ways with the help of the tape-editing equipment. If the programme is a long one it may be punched in sections (for example, each subroutine could be punched separately) and the several lengths of paper tape joined together (or *spliced*) afterwards. When the programme tape is complete it is spooled up on a hand spooler (Plate 6) and placed in a box (Plate 5).

When the programme is to be run on the computer the programme tape is usually put into a tape trough and its start is placed in the main tape-reader on the computer desk (Plate 3). There is always a *leader* of blank tape at the beginning of the tape and it is this which is placed in the tape-reader. The Initial Orders are then called in by operating the *Start key* on the control panel.

The *Initial Orders* are the principal means by which programmes and numbers are fed into the computer; they form a complicated programme stored permanently in the isolated part of the main store. They are complicated because they are both easy to use and comprehensive in function. To call them in we must first put the Run key to the STOP position and then push the Start key up to the position labelled START (this key springs back to the NORMAL position when released). This operation clears OVR and the external-conditioning relays (see Sec. 3.10) and certain stops (write-with-overflow, and unassigned order, see Sec. 3.9), prepares the computer to obey an *a*-order next, and places the *start order-pair*, which is

896 0 72

0.0 0 60

in the order-register. Now *B*896 is the first block in the isolated part of the main store; it contains the beginning of the Initial Orders programme, so that when the Run key is put back to RUN and the start order-pair is obeyed it causes the computer to start obeying the Initial Orders.† Assuming the handswitches to be clear, which is usually so, the Initial Orders then read the tape in the main tape-reader, taking in the various orders and numbers punched in it, converting them to the

† For the 4096-word store, the start order-pair is

512 0 72

0.0 0 60

internal form required by the computer and placing them in the main store. The process of calling in the Initial Orders in the way just described is called the *Normal Start*.

We must be able to direct the operations of the Initial Orders in at least two important ways:

(a) we must control where, in the main store, the orders and numbers read from the tape are placed,

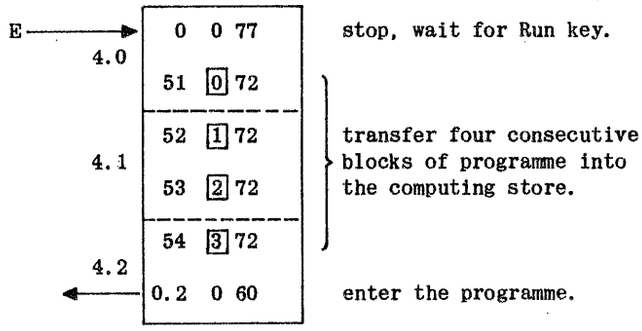
and (b) we must arrange that, on reaching the end of the tape, the computer starts to obey (i.e. *enters*) the programme just read in.

These and other operations of the Initial Orders are controlled by special groups of characters, called *directives* or *warning characters*, punched along with the programme in the tape. The directives are used exclusively to direct the operations of the Initial Orders, they are not stored. Before discussing them further we must introduce the Transfer Address.

The *Transfer Address* (usually abbreviated to T.A.) is simply the main store address where the next order-pair or number on the tape is about to be placed by the Initial Orders. Every time an order-pair or number has been read in and converted into a binary word, this word is written into the storage location specified in the Transfer Address: as soon as this has been done the Transfer Address is increased by unity in readiness for the next word. Consequently the order-pairs and numbers punched in the tape are placed in consecutive storage locations in the order in which they are punched. We can easily set the Transfer Address to any value we please at the beginning of the tape, or in the middle of it, by using the warning character T (for *transfer*). Suppose, for example, that we wish to place the first order-pair on the tape in *B*51.2; all we do is to punch T 51.2 at the beginning of the tape before the first order. The T 51.2 is a directive (in which T is the warning character and 51.2 the address) and is not stored; it merely causes the Initial Orders to set the Transfer Address equal to 51.2 and then carry on to read more tape. If there is a subroutine on the tape which has to be stored in *B*19.0 onwards, then we can punch the directive T 19.0 in the tape just before the subroutine.

The Transfer Address is set equal to 2.0 during a Normal Start; consequently if there is no T-directive at the head of the tape, the first order-pair or number read will be placed in *B*2.0.

When the Initial Orders reach the end of the tape we usually want the computer to enter the programme (i.e. to start obeying it); this can be done by using another directive consisting of the letter E (for *enter*) followed by an address. The effects of this are best explained by an example: consider, therefore, the result of the directive E 51.2, which means enter (start obeying orders) at the *a*-order in *B*51.2. As soon as this has been read by the Initial Orders (probably at the end of the programme tape) there is a 77-stop; this is to allow the input tape to be changed or the hand-switches to be set, etc. When the Run key is operated, *B*51 (the block containing the specified order) is copied into *U*0, and the next three blocks (*B*52, 53 and 54) are copied into *U*1, 2 and 3, respectively. The Initial Orders thus transfer the first four blocks of the programme into the computing store. After this there is a jump to the specified order, now the *a*-order in *U*0.2, and the computer leaves the Initial Orders and starts to obey the programme. There are a few other effects of less importance which we shall describe later, but it will be seen that when the Initial Orders read E 51.2 from the input tape the effect is similar to that of the following sequence of orders.



If we want to enter the programme at the *b*-order in 51.2, instead of the *a*-order, all we need do is to use E 51.2+, i.e. we write a plus sign after the address (like we do with jump orders).

Usually the directives associated with a programme are written in the appropriate places on the programme sheets; but it is important to realize that they do not form part of the programme when it is obeyed. The directives are concerned only with the process of input of the programme and not with its execution.

As an illustration of the use of these directives, suppose we have a programme consisting of three parts:

(a) a master-programme which is to go into blocks *B*5 to 24,

(b) a list of numbers which is to go into *B*233.5 onwards, and

(c) a subroutine to go into *B*50 and 51.

Suppose the programme starts at the *b*-order in 5.2. The tape could be punched as follows:

```
T 5.0
(master-programme)
T 233.5
(list of numbers)
T 50.0
(subroutine)
E 5.2+
```

The address in a directive always refers to the main store. The address in a T-directive can be the address of any location in the main store. That in an E-directive can be the address of any order in the main store, but this order must be obeyed in *U0*. There are many other directives which will be described in detail in later chapters, but we shall summarize three of them briefly here.

The J-directive (J for *jump*) is very similar to the E-directive; the only difference being that there is no 77-stop before the programme is entered. Thus the directive J 51.2+ punched in the tape causes a programme to be entered at the *b*-order in *B*51.2 as soon as it has been read.

In order to prevent wrong tapes being used every tape should have a name. This name should be written in ink or pencil on the blank leader at the beginning of the tape. As a valuable extra precaution the name is usually also punched before the first order or number in the tape; it is then preceded by N (for *name*). The warning character N, followed by the name and a short length of blank tape, form an N-directive (there is no address). When this is read by the Initial Orders the whole name is copied from the input tape on to the output tape, i.e. the name is printed; the end of the name is signalled by the short length of blank tape. The name itself can be made up of any of the available characters but must not, for obvious reasons, include any lengths of blank tape. It is advisable to have names for each part of a complete programme, e.g. for each subroutine; normally each name would start with CRLF so that it gets printed during input on a line to itself.

It is desirable that the output of the programme should bear the date, and to this end each complete programme tape should be headed by a D-directive, punched immediately after the blank leader before anything else in the tape. This consists simply of a letter D; there is no address. When the Initial Orders read a D-directive they cause the *date and a serial number* to be printed; the serial number is increased by one every time it is used. In this way the printed page produced by the programme is headed automatically by the date and serial number on each occasion the programme is used. It is then easy, on a later occasion, to determine the dates when various runs of a programme were made. This is especially useful when a new programme is being developed; the effects of various alterations made to the programme between successive runs can be studied without risk of confusing the runs.†

The directives so far described are summarized below; here the address is written as *a* if it is the address of a storage location, or as *a*(+) if it is the address of a single order.

T *a*	set Transfer Address = *a*
E *a*(+)	enter programme at *a*(+) after 77-stop
J *a*(+)	enter programme at *a*(+)
N	print the following name
D	print the date and serial number

As an illustration, suppose a programme is made up as follows:

(a) the master-programme is to occupy *B*2 to *B*12.

(b) a subroutine to go into *B*18 onwards has the name SPECIAL PRINT 3,

(c) a list of numbers with the name CONST. PZ is to go into *B*54.3 onwards.

If the name of the programme is DTRY 73 and it is to be entered in *B*2.0 then its tape could be punched as follows.

```
D
N
DTRY 73
T 2.0
(master-programme)
T 18.0
N
SPECIAL PRINT 3
(subroutine)
T 54.3
N
CONST. PZ
(numbers)
E 2.0
```

When this tape is read by the Initial Orders there is printing like this†† before the programme starts to be obeyed.

---

† The date and serial number (initially zero) are set in *B*895.7 and *B*895.6 respectively as a part of the daily maintenance process. These are the last two storage locations before the isolated part of the main store, and programmers should avoid writing into them. In the 4096-word store, *B*511.7 and *B*511.6 are used.

†† We assume here that *H0* = 1, i.e. that the sign-digit key of the handswitches is down. If *H0* = 0 then some extra printing occurs when certain directives are read; this is called *optional printing* and will be described later.

14/8/60----89

DTRY 73

SPECIAL PRINT 3

CONST. PZ

Here the top line consists of the date and serial number. The directive T 2.0 punched before the master-programme can be omitted because the Transfer Address is always set to 2.0 at the beginning by the Normal Start operation.

4.4 A simple complete programme - "Special Factorize"

We shall now describe a simple complete programme for factorizing whole numbers. For example, if supplied with the number 420 it will cause the computer to print.

$$420 = 2 \times 2 \times 3 \times 5 \times 7$$

The process used may be described by the flow-diagram of Fig. 4.2, in which the following notation is used.

- N At first this is the number to be factorized, but it is reduced by having factors removed as they are found,
- d Trial factor, initially 3, then successive odd numbers,
- q The quotient when N is divided by d,
- r The remainder when N is divided by d.

This process falls naturally into three stages, as follows.

Stage A The number (N) to be factorized is printed on a new line, and followed by an equals sign.

Stage B The number is repeatedly divided by 2 as many times as possible (if any), and '2' is printed for each factor. A multiplication sign is also printed after each factor if there are any more factors, i.e. if the quotient is not 1. As each factor is found it is removed from N, i.e. N is replaced by the quotient.

Stage C The number (N), which is by now odd, is divided by a trial factor (denoted by d). This trial factor is at first 3, and is then increased by 2 repeatedly so that we try dividing N by successive odd numbers 3, 5, 7, 9, ... If one of these trial divisions leaves a remainder of zero then d is a factor of N and it is printed and removed from N (i.e. N is replaced by the quotient). A multiplication sign is also printed after each factor if the quotient is not 1. After each unsuccessful trial division d is compared with the quotient q; if d is less than q we try the next value of d, but otherwise N is prime and is in turn printed (see the discussion in example (c) of Sec. 1.5).

To get a natural layout in the printing we must put a space on either side of the equals and multiplication signs, and we must arrange to omit left-hand zeros in the numbers (see Sec. 4.1). To show how integers may be printed with left-hand zero omission we have incorporated the necessary sequence of orders in the programme instead of using a proper subroutine of the kind described in Sec. 4.2. This particular sequence is especially simple because the integers to be printed are known to be positive (no special treatment of the last digit is needed) and it is called in at only two points in the programme, firstly to print N and later to print the factors. Instead of using a link, $X1$ is used as an indicator to steer the computer to the right part of the programme: $X1$ is clear when the the sequence is printing N, but thereafter $X1$ holds 14 (the value of Sp).

The whole programme fits into four blocks which are placed in $B2$ to $B5$ in the main store and are copied into $U0$ to $U3$ when the E-directive is read (see Sec. 4.3). Consequently the programme proper is concerned exclusively with the computing store. The number to be factorized is placed in $B2.0$ along with the programme and is therefore available in $U0.0$ on entry; this number is initially 420 but can be changed as we shall shortly describe. The programme is entered when the Initial Orders read E 2.1 or J 2.1. A facsimile of the programme sheet is shown in Fig. 4.3.

The three orders starting at the b-order in $U3.5$ at the end of the programme require some explanation. A jump to these orders occurs if N is negative or when N has been completely factorized. Since $U3.6$ holds a stop order-pair (see Sec. 3.9) the first effect is to cause an optional stop. When the Run key is operated to allow the computer to continue, the effect of these orders is to call in the Initial Orders, exactly as though a Normal Start operation had been done. Why this happens will be described in a later chapter since a knowledge of modification is needed. The reader will recall that in a Normal Start the Transfer Address is set to 2.0, after which the Initial Orders read tape. If therefore we now put in the tape-reader a tape such as the following

+ 996

J 2.1

then the Initial Orders will put the integer +996 (or whatever other number is on the tape) into $B2.0$ (overwriting the previous number there) and then enter the factorizing programme as soon as the J 2.1 is read. We can therefore factorize several numbers in succession by simply preparing an input tape on which each number is followed by J 2.1. If this is done then it is best to inhibit the optional stops by means of the special key on the control panel. This input tape should be placed in the main tape-reader when the programme tape has all been read in by the Initial Orders. Note that we are in effect using the Initial Orders as a kind of subroutine to read in the numbers; this subject is further discussed in Chapter 7.

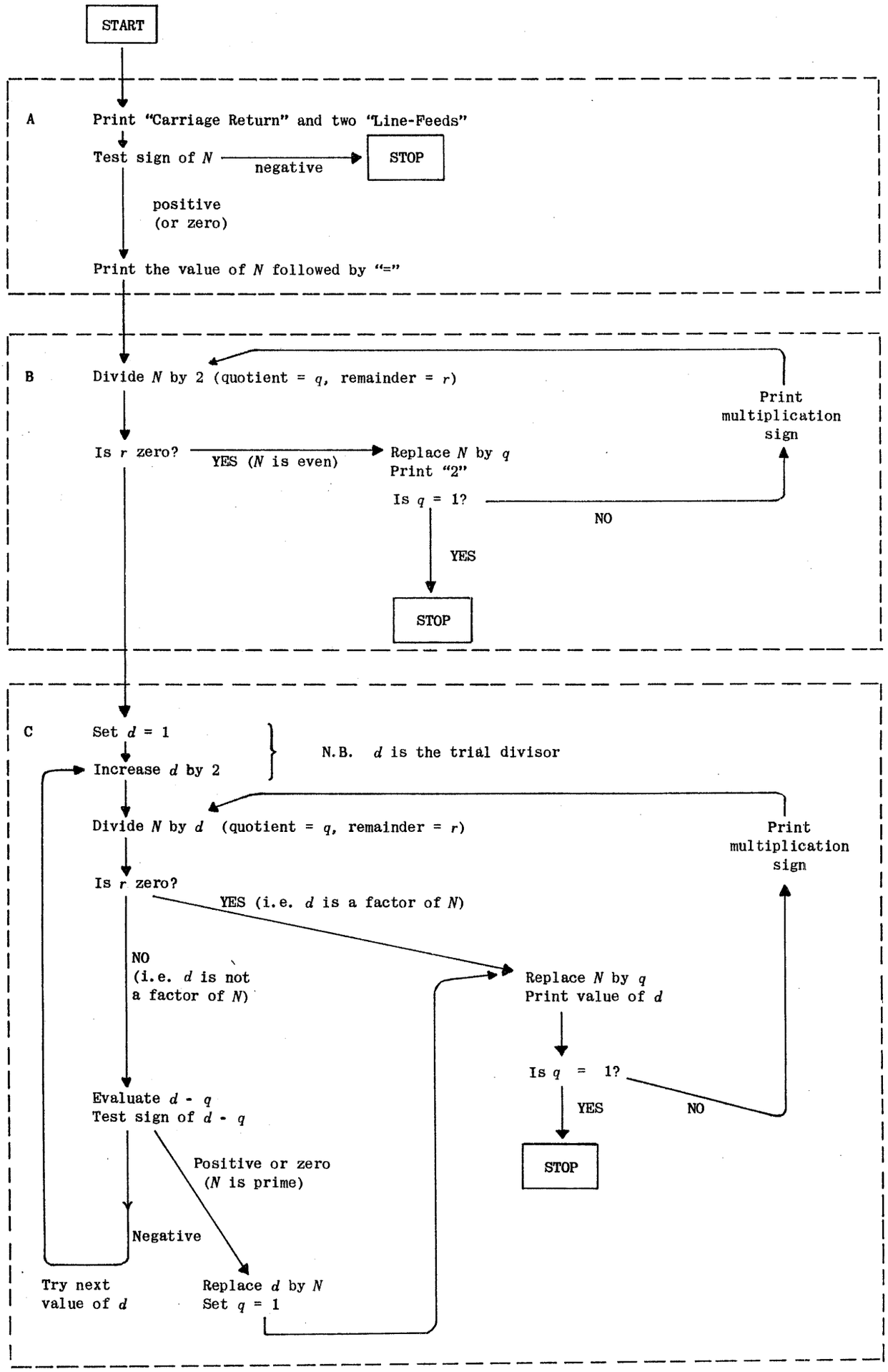


Fig. 4.2 *Flow-diagram of the programme 'Special Factorize'*

PEGASUS PROGRAMME SHEET

Date 18.5.57

Sheet 1

D
N
SPECIAL FACTORIZE

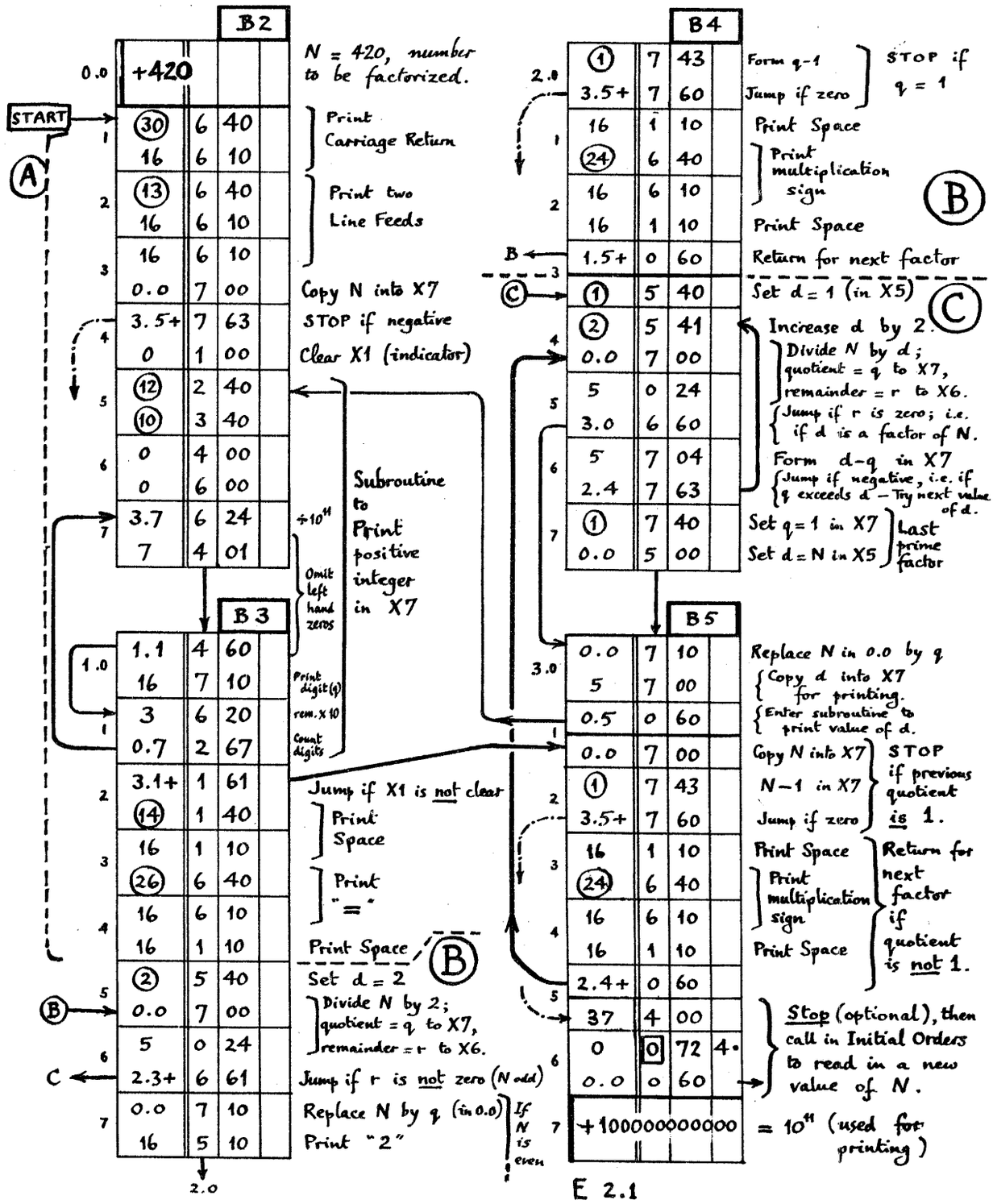


Fig. 4.3 *The programme sheet for 'Special Factorize'.*

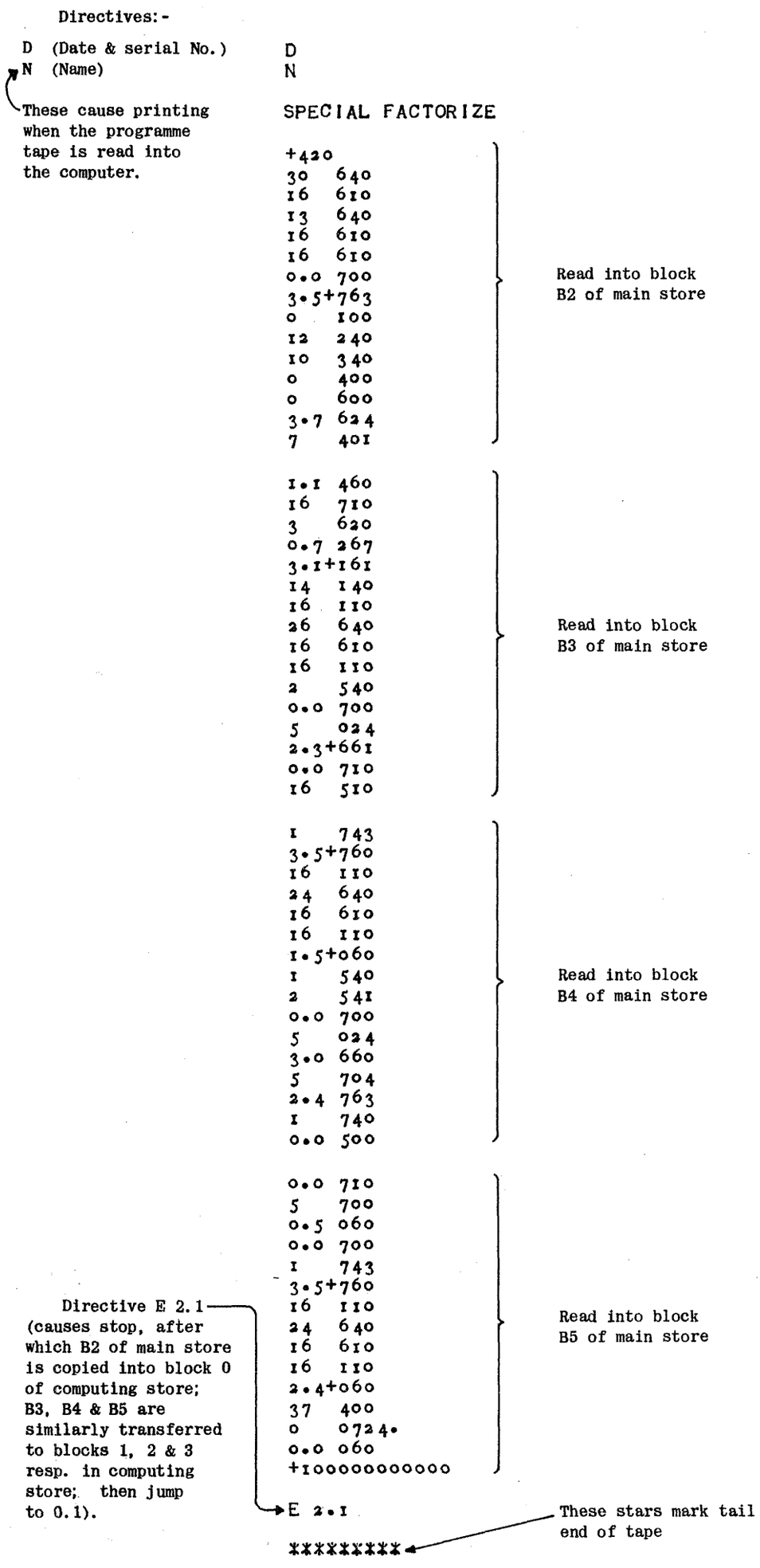


Fig. 4.4 *Print-out of the programme tape for 'Special Factorize'.*

Fig.4.4 is a reproduction of the print-out obtained when the programme is punched; this should be compared with the programme sheet of Fig.4.3. Note the Stars punched at the end of the tape - these are used to distinguish one end of the tape from the other. The print-out shown in Fig.4.5 is of a typical input tape carrying various numbers to be factorized: in this print-out each J 2.1 appears on the same line as the number; this is permissible, as will be explained later, since a number may be terminated either by CRLF or by Sp when it is to be read by the Initial Orders. After the last number on this tape is punched a Z-directive; the effect of this is simply to cause the Initial Orders to encounter a 77-stop; it is punched here just to stop the computer when all the numbers have been factorized. The results of running this programme are shown in Fig.4.6, which is a reproduction of the output.

```
N
TEST NUMBERS

+4      J 2.1
+5      J 2.1
+6      J 2.1
+10     J 2.1
+28     J 2.1

+98       J 2.1
+1001     J 2.1
+2662     J 2.1
+12345    J 2.1
+34567    J 2.1

+10001       J 2.1
+1234567     J 2.1
+7654321     J 2.1
+1000001     J 2.1
+99009901    J 2.1

-9        J 2.1
+9        J 2.1
+14641    J 2.1
+512      J 2.1
+96       J 2.1

Z
*******
```

Fig.4.5 *Print-out of an input tape for 'Special Factorize'.*

In view of the crudeness of the process used it might be thought that the programme would be very slow. This is not so, however. For most of the numbers shown the factorization takes a hardly perceptible time and the output punch is running nearly continuously. For 'difficult' numbers having large prime factors the running time can be easily calculated since it takes about 7 milliseconds to try each value of d. Thus all values of d less than 1000 (i.e. 3, 5, 7, ..., 999) can be tried in about 3½ seconds, and it therefore takes at most this time to factorize any number up to 1000000. The number 99009901 shown takes about 12½ seconds. †

4.5 Another complete programme

We shall now give an example of a programme using several subroutines. Suppose we have to tabulate the values of

$$z = \frac{7}{8}\sqrt{\frac{y^2-1}{y^2+1}}, \quad \text{where } y = \frac{3+2x^2}{1+4x}, \tag{1}$$

for x = 0, ½, 1, 1½, ... 6½.†† Since some of the quantities concerned cannot be represented directly as fractions in the range −1 to +1, we must manipulate the formulae to a certain extent and introduce scale factors. Instead of storing x we shall use the fraction $\xi = 1/8\ x$, which is always in range (a possible alternative would be to store $2x$ as an integer). The quantity y attains its largest value, of rather less than 3½, when x = 6½ so we can safely evaluate $1/4\ y$ without overflow; we shall write $\eta = 1/4\ y$ The value of y is a minimum, y = 1, when x = 1 so that

$$\tfrac{1}{4} \leqslant \eta < \tfrac{7}{8}.$$

---

† This number is actually a divisor of $10^{10}+1$ and more refined processes can be used on such numbers. In this case, for example, it is known that any factor must be of the form $20k + 1$, so we are actually trying ten times too many trial factors. With general numbers it is easy to omit multiples of 3 and 5, which roughly doubles the speed of factorizing.

†† This can usefully be written x = 0(½)6½, meaning that x takes values from 0 to 6½ in steps of ½.

```
13/5/60---6                Date & Serial No.  } Printed during
                                              } input of the
SPECIAL FACTORIZE          Name of programme  } programme tape.

420 = 2 X 2 X 3 X 5 X 7 <-- Test number

TEST NUMBERS               <-- Name of input tape
4 = 2 X 2
5 = 5
6 = 2 X 3
10 = 2 X 5
28 = 2 X 2 X 7
98 = 2 X 7 X 7
1001 = 7 X 11 X 13
2662 = 2 X 11 X 11 X 11
12345 = 3 X 5 X 823
34567 = 13 X 2659
10001 = 73 X 137
1234567 = 127 X 9721
7654321 = 19 X 402859
1000001 = 101 X 9901
99009901 = 3541 X 27961

9 = 3 X 3
14641 = 11 X 11 X 11 X 11
512 = 2 X 2 X 2 X 2 X 2 X 2 X 2 X 2 X 2
96 = 2 X 2 X 2 X 2 X 2 X 3
```

Fig. 4.6 *Output produced by the 'Special Factorize' programme.*

If we substitute $x = 8\xi$ and $y = 4\eta$ in the formula for y we get

$$\eta = \frac{\frac{3}{128} + \xi^2}{\frac{1}{32} + \xi}, \tag{2}$$

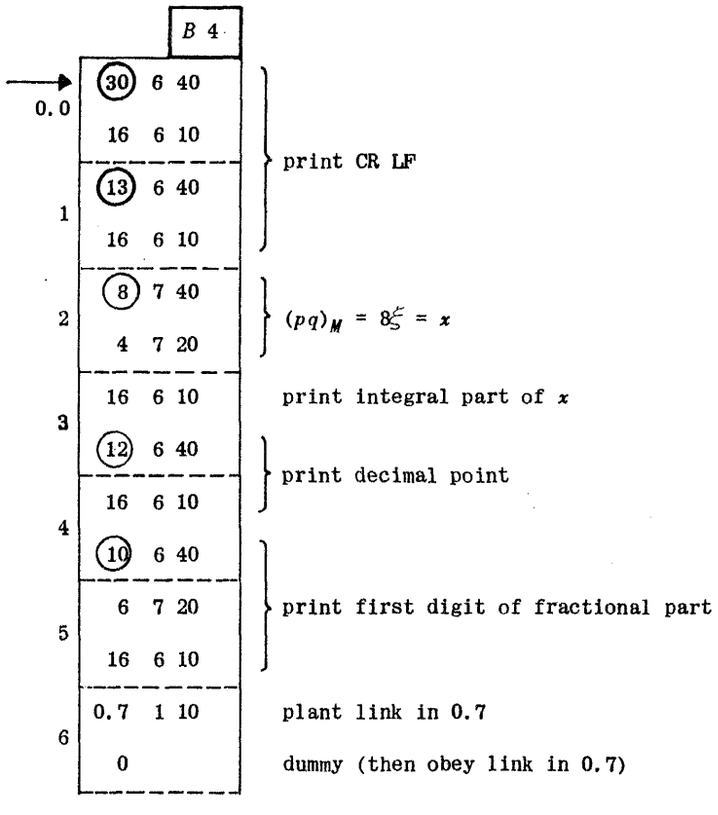
and we can see that the numerator and denominator and the result are all within the permissible range. In terms of η the formula for z becomes

$$z = \tfrac{1}{8}\sqrt{\frac{16\eta^2 - 1}{16\eta^2 + 1}} = \tfrac{1}{8}\sqrt{\frac{\eta^2 - \frac{1}{16}}{\eta^2 + \frac{1}{16}}}. \tag{3}$$

Since $1/16 \leqslant \eta^2 < \eta < 7/8$ both numerator and denominator are within range and are never negative. Clearly also the numerator is always less than the denominator. We shall therefore deal with the formulae (2) and (3) rather than (1).

There are 14 values of x and we shall therefore arrange the printing of the results on 14 lines and in two columns; on each line there will be a value of x and the corresponding value of z. We shall write a special subroutine for printing x (= 8ξ) preceded by CRLF, given the value of ξ, since this is very simple.† The reader should have no difficulty in understanding this subroutine, which is as follows. The value of ξ is held as a fraction in X4, and no rounding is needed since ξ is represented exactly and printed in full.

† Some library subroutines could be used for this but it would be inappropriate at present to describe them in the necessary detail.



The brief specification of this subroutine is:-

Print 8*C*(4) to one decimal (unrounded) preceded by CR LF.

Cue:

4	0	72
0.0	0	60

Uses: *U*0; *X*6, 7.

Link: *X*1, obeyed in 0.7.

Note: *C*(4) must be non-negative.

So that we can print z we shall suppose that a subroutine is available with the following brief specification.

Print *C*(7) as a fraction (unrounded) preceded by Sp.

Cue:

51	0	72
0.3	0	60

Uses: *U*0; *B*0.

Link: *X*1, obeyed in 0.7.

Programme-parameter in *X*2 specifies the number of digits to be printed after the decimal point. The sign is printed as Sp or a minus sign.

Since we have to evaluate a square-root we shall assume that the following subroutine is available.

Put into *X*6 the square root of the double-length fraction in *X*6 and 7 (or, briefly, $p' = \sqrt{(pq)}$).

Cue:

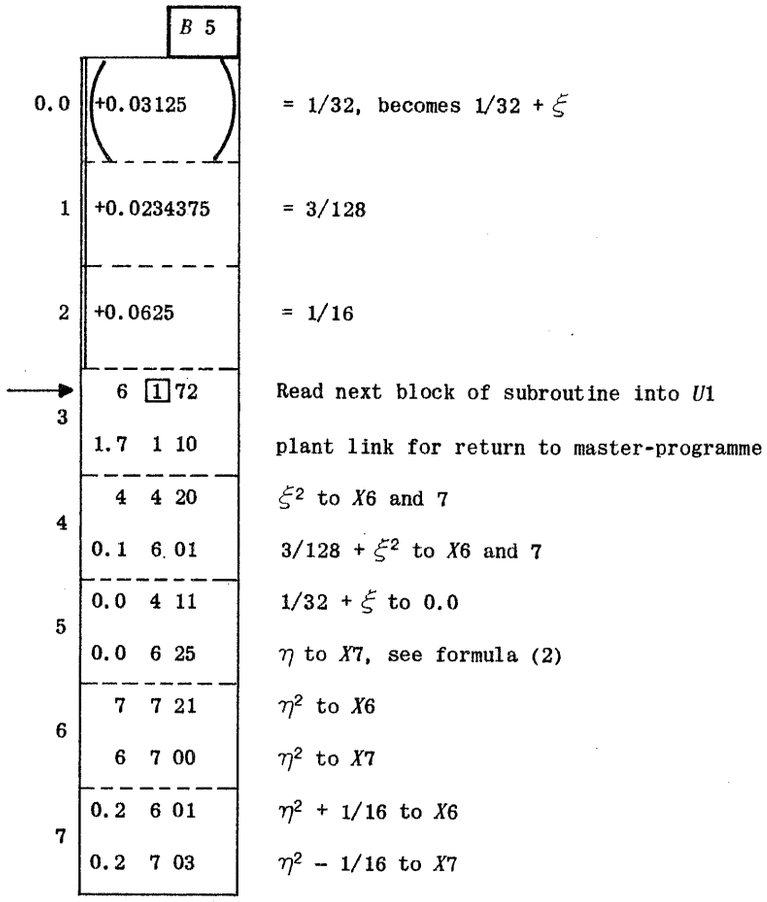
74	0	72
0.0	0	60

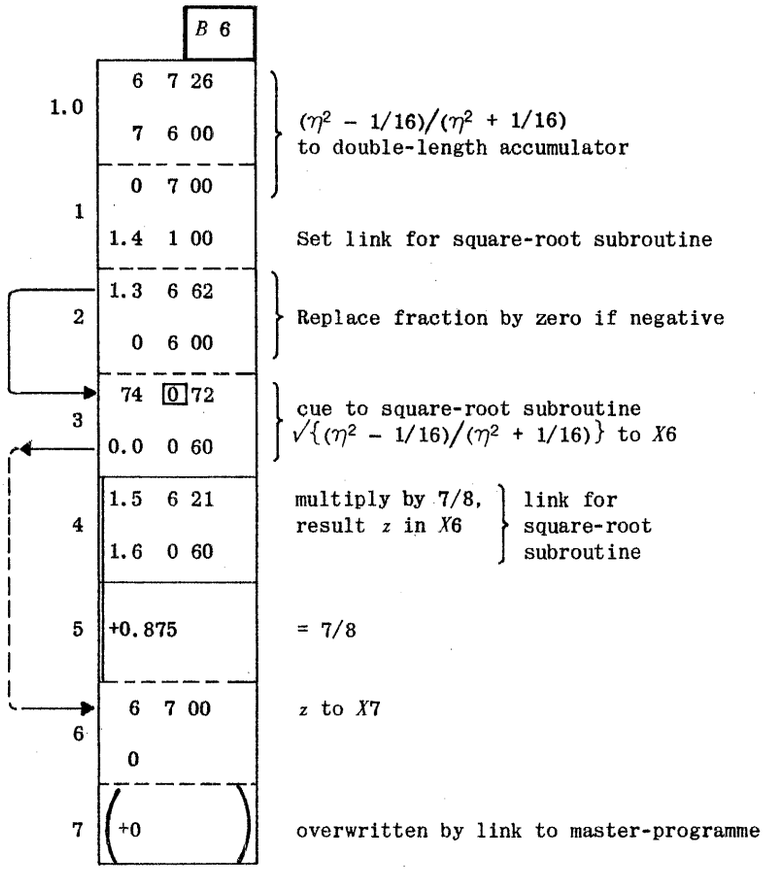
Uses: *U*0; *X*5, 6, 7.

Link: *X*1, obeyed in 0.7.

There is a loop stop if (pq) is negative.

We shall construct a *special subroutine* to evaluate z from ξ according to the formulae (2) and (3) above. It is convenient to keep the current value of ξ in $X4$ during this programme; this subroutine therefore takes ξ in $X4$ and places z in $X7$ (ready for printing). There are a few points to watch. First, the subroutine will itself have to call in a subroutine to evaluate a square root; it will therefore have to remove the link from $X1$ and preserve it until it is needed. Second, although ξ is stored exactly (in $X4$), η will have to be represented approximately, since it is calculated from the formula (2) above. Consequently the minimum value of η^2 may not be exactly 1/16 (as it should be) and the numerator of the fraction in formula (3) may go slightly negative when it should be zero. Since we do not wish to introduce spurious imaginary numbers we must make sure that the numerator in formula (3) is replaced by zero if it goes negative. Apart from these points the subroutine is simply a straightforward evaluation of formulae (2) and (3). We suppose the subroutine to occupy $B5$ and 6.





The brief specification of this subroutine is as follows.

Place z in $X7$, evaluated from ξ in $X4$ according to the formulae (2) and (3) for $0 \leqslant \xi \leqslant 13/16$.

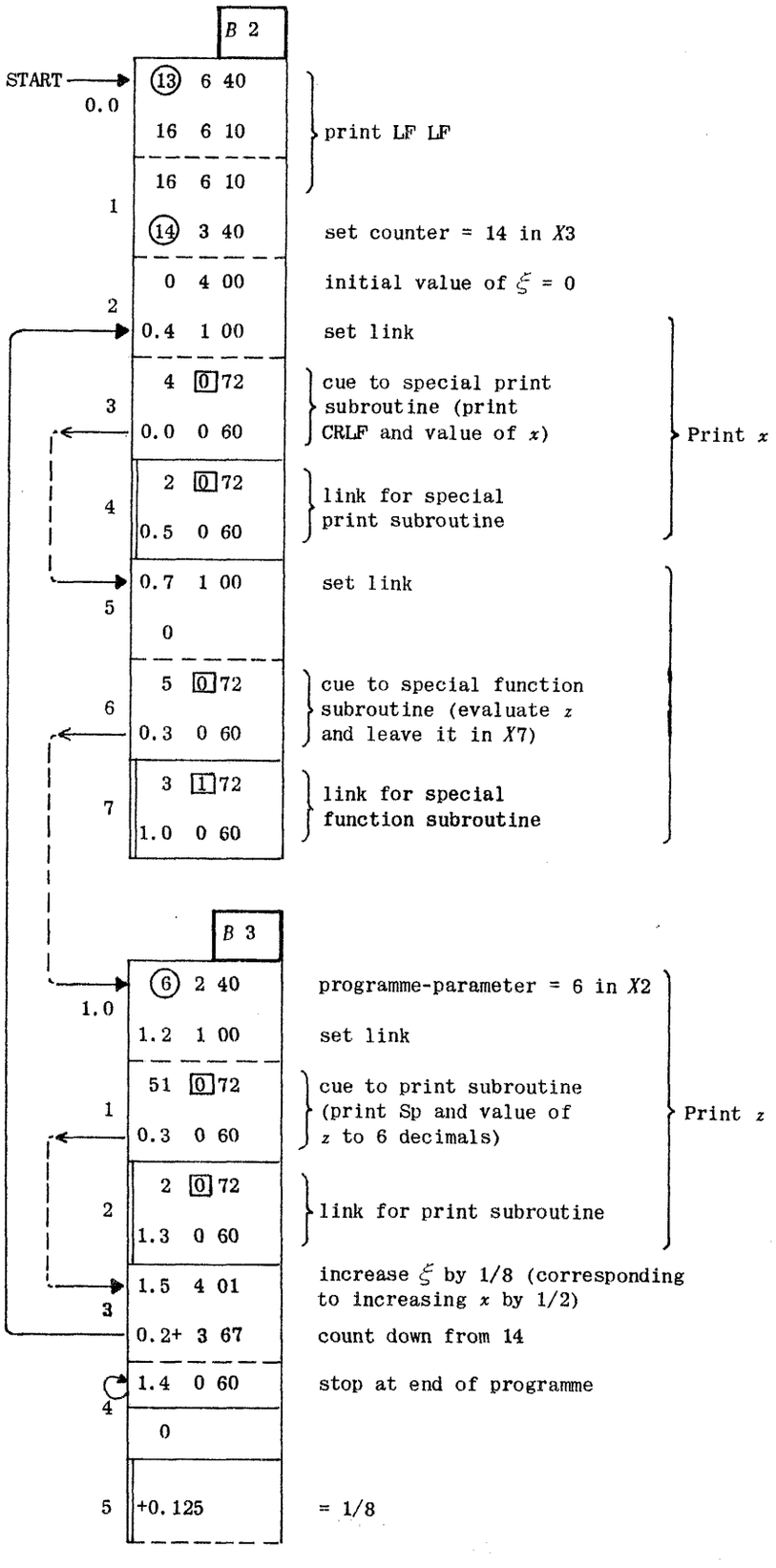
Cue:

5 [0] 72
0.3 0 60

Uses: $U0$, 1; $X1$, 5, 6, 7.

Link: $X1$, obeyed in 1.7.

We must now draw up the *master-programme;* this has merely to set a few starting values, call in the subroutines as required, move one or two numbers about and do some counting. The current value of ξ is kept in $X4$; it is initially zero. We keep a count in $X3$ to indicate the end of the programme after 14 cycles through the main loop. In following the master-programme the reader should carefully note the reasons for the various 72-orders resetting blocks of the programme in the computing store.



The tape for this programme is made up as follows.

D	Date and Serial Number
N TABULATE SPECIAL FUNCTION	Name of programme
(Master-programme)	Starts in *B*2.0
T 4.0	Set T.A. = 4.0
N SPECIAL PRINT	Name of subroutine to print x
(Subroutine to print x)	Starts in *B*4.0.
T 5.0	Set T.A. = 5.0
N EVALUATE Z	Name of subroutine to evaluate z from ξ
(Subroutine to evaluate z)	Starts in *B*5.0
T 51.0 (Print fraction subroutine) T 74.0 (Square-root subroutine)	Standard subroutines
E 2.0	Enter programme

The reader should note that no assumptions are made in this programme about the contents of any parts of the main or computing stores unless the programme has itself placed something there. One should take care not to assume, for example, that any unused registers or locations will be clear when the programme is entered. In fact, when the programme has been read in, the unused parts of the main store will contain whatever was left in them by the previous programme.

4.6 Relative addresses

A subroutine will usually occupy several consecutive blocks in the main store and the address of the first of these is called the address of the subroutine. Most subroutines are so written that they can be called in by a simple cue consisting of a block-read order and a jump. This cue brings only one of the blocks of the subroutine into the computing store, and there will consequently be further block-read orders in most subroutines which will bring in the remaining blocks as they are needed. In the last Section we gave an example of a subroutine to evaluate a function z determined by a number ξ; this subroutine occupied *B*5 and *B*6 and its first order (the a-order in *B*5.3) was 6 [1] 72 to bring in its second block. If we decided, for some reason or other, to place the subroutine in say *B*10 and *B*11 this order would have to be changed to 11 [1] 72 but we need make no other change in the subroutine. This is typical of most subroutines: as a rule the addresses in certain orders (usually 72-orders) will require changing in a simple way if the subroutine is moved. This is quite a serious disadvantage because it means that some of the orders in subroutines cannot be written in until it has been decided just where the subroutines are to be stored; and this is a decision which often cannot be made at the time the subroutine is written. Until we know where a subroutine is to be stored its tape cannot be prepared and it cannot therefore be tested. This is most serious in library subroutines since it is impracticable to allocate different parts of the main store to each subroutine.

To remove these difficulties a system of relative block-numbering has been introduced. The first (lowest-numbered) block of a subroutine is called 0+, the next block 1+, and so on. These *relative* block-numbers are written in the subroutine. For example, the order written 6 [1] 72 in the subroutine of the last Section would be written

1+ [1] 72

and *punched* like this. When the subroutine tape is read by the Initial Orders the relative block-numbers are converted into the correct absolute block-numbers; this is done by adding the address of the subroutine (i.e. the block-number of its first block) to each relative block-number. Thus if the subroutine is stored in *B*5 onwards then 5 has to be added to each relative block-number to convert it into the correct absolute number; if the subroutine is stored in *B*10 onwards then 10 must be added. The number to be added is called the *relativizer;* it is simply the address of the subroutine's first block.

As a rule there will be several subroutines in a programme, and each of them may contain orders such as

1+ [1] 72

requiring the addition of a relativizer. It is obviously essential that the Initial Orders should be informed of the start of a subroutine tape during input of the programme so that the appropriate block-number is used as a relativiser. This is done with the aid of a B-directive (B for *block*).

At the head of each subroutine tape the letter B is punched, before any of the orders or numbers of the subroutine. This directive has no address. When the B is read by the Initial Orders there are two main effects. First, the Transfer Address is increased (if necessary) to the beginning of a block; and, second, the number of this block is recorded as a relativizer for subsequent addition to relative block-numbers. For example, if the T.A. is 4.6 when a B is read, then the T.A. will be changed to 5.0 and the relativizer will be set equal to 5. If the T.A. is 11.0 it will be unchanged

but the relativizer will be set equal to 11. These two effects ensure that every subroutine starts at the beginning of a block, and that the correct relativizer is used within each subroutine. Provided the Assembly Routine is not being used, the B-directive is normally the only way of changing the relativizer: when once the relativizer has been set by a B then it will remain fixed in value until the next B is read. During a Normal Start (see Sec. 4.3) the T.A. is set to 2.0 and the relativizer is set to 2; the effect is the same as if every complete programme tape were headed by

T 2.0

B

The relative numbering of blocks can usefully be used not only in all subroutines but also in the master-programme. It gives a certain amount of flexibility and routines can be moved if necessary.

It should be noted that a relative address is something which exists only in the external form of the programme (i.e. in its written or punched form). The computer proper knows nothing of relative addresses, every order it obeys has an absolute address, which is represented by certain binary digits. Since there are only 7 bits in the *N*-address of an order we must be careful that the absolute block-number obtained by adding the relativizer to a relative block-number does not exceed 127. Note also that block 0+ in one part of a programme will often be different from block 0+ in another part.

The addresses in directives such as T, E or J can, if desired, be relative addresses. For example we could use a directive

T 3+.4

to set the T.A. so that the next word read from the tape goes into the location whose position-number is 4 in block 3+ (the relativizer used is, of course, that set by the last preceding B). If we use the directive

T 2+

this will set T.A. to the start of block 2+, i.e. it is equivalent to the directive

T 2+.0

Such directives are useful if, for example, some working space is required by a subroutine.

It is important to distinguish the various uses of the + sign. In a jump order such as

1.5+ 2 63

the + sign is used to indicate a jump to a *b*-order. In a transfer order such as

2+ 0 72

the + sign denotes a relative block-number. The Initial Orders can distinguish these during input of the programme tape because a ⊙ is punched before the + sign in the former order. In a number we must punch the sign before any of the digits, e.g.

+12 or +0.1234,

and it is this fact which differentiates order-pairs from numbers on the tape. A special "address-input" section of the Initial Orders is used to read the addresses in directives; if a + sign is read before a ⊙ is encountered, then the number is treated as a relative block-number, e.g.

T 1+.3 or T 4+ or E 0+.2

If a + sign is read after a ⊙ as in the directive

E 2.6+

the address is taken to refer to a *b*-order.†

We can summarize as follows the directives so far described.

T *a*	set Transfer Address = *a*
B	set T.A. to new block, record new block-number as relativizer
E *a*(+)	enter programme at *a*(+) after 77-stop
J *a*(+)	enter programme at *a*(+)
N	print the following name
D	print the date and serial number
Z	77-stop.

† Occasionally we need such directives as E 0+.2+ in which the address refers to the *b*-order in position 2 of block 0+.

Chapter 5

Modification

Because nearly all calculations are repetitive, almost every programme will contain loops or cycles of orders in which the same sequence of operations is repeated, perhaps hundreds of times or more, with different numbers. In this chapter we discuss these loops and the way in which their orders can be changed or modified so that slightly different operations can be performed on successive repetitions.

5.1 Modification and counting

Most programmes contain several loops or cycles which have to be traversed many times; and usually some of the orders in the loop will have to be changed slightly in successive repetitions. For example, when adding together the numbers in a list or set by means of a loop of orders, there will be an addition order, probably an 01-order, which adds one of the numbers into an accumulator. In order to add together all the numbers we must arrange that this basic order is obeyed the correct number of times and that it somehow adds in the next number of the set each time it is obeyed. The technique of *counting*, which has already been discussed (Sec. 3.8), can be applied here to ensure that the loop is traversed the correct number of times. We must also arrange that the addition order is *modified* so that on the first occasion when the loop is obeyed it adds in the first number, on the second occasion it adds in the second number, and so on. The facility of *modification* allows the addresses in orders to be readily changed in a systematic way so that different numbers of a set or list can be handled successively by the same sequence of orders.

We shall use the word *processing* to describe the systematic treatment of sets of numbers; for example the numbers may successively take part in arithmetical operations and may perhaps be replaced by other numbers. If we are processing successively the numbers in a fairly large set we cannot in general hold all of them in the computing store; as a rule a set of numbers like this will be spread over several blocks in the main store. Usually we shall keep in the computing store only one block of the numbers (i.e. eight of them); we proceed by working through these numbers, one at a time, until all eight of them have been used and we then bring in a further block from the main store. The whole process is then repeated until the last number of the set has been dealt with. The following facilities are therefore needed:

(a) We must be able to change the N-address in arithmetical orders (i.e. the orders of groups 0, 1 and 2) so that we can work systematically through a block of eight numbers in the computing store, processing each number in turn.
(b) We must be able to determine when the last number of a block has been processed.
(c) We must then be able to read the next block of numbers from the main store (we may also have to write back into the main store the numbers which have just been processed, since they may have been altered). This requires the facility of changing the first address in block-transfer orders.
(d) We must be able to determine when the last number of the whole set has been processed, in order to carry on with the next part of the programme.

These four facilities are provided in the following way:

(a) The orders of groups 0, 1 and 2 may be modified in a certain way which is described below.
(b) A special *unit-modify order* (the 66-order) has been introduced which can be used to determine when the last number of a block has been processed.
(c) The block-transfer orders (72 and 73) may be modified, but in a different way from the arithmetical orders.
(d) The unit-count order (the 67-order) can be used to determine when the last number of the whole set has been processed.

The techniques of counting and modification are usually needed together. We shall require two numbers, a *counter* and a *modifier*. At the start of a loop of orders we put the counter equal to the number of times we wish to obey the orders of the loop (i.e. to the number of numbers in the set), and we usually put the modifier equal to the main store address of the first number of the set. Every time we go round the loop we *reduce* the counter by one; when it reaches zero we have processed all the numbers. We also arrange to *increase* the modifier by one each time the loop is obeyed so that it is always equal to the main store address of the number being processed. We can arrange that a modifier, or a part of it, is added to the address in an order before the order is obeyed.

A single accumulator may be used to hold both a modifier and a counter. This is possible because only 13 binary digits are needed in the modifier to specify the address of any location in the main store. The modifier is made up of digits 1 to 13 in the accumulator. The counter consists of the right-hand 25 binary digits in the accumulator (i.e. digits 14 to 38). The sign-bit (digit 0) is not included in either the modifier or the counter. The diagram (Fig. 5.1) shows how the digits are allocated. The modifier in accumulator X will be denoted by x_M and the counter by x_C. If we are considering a particular accumulator, say $X4$, we shall write 4_M and 4_C for the modifier and counter.

Often the modifier will represent a main store address and we shall write x_B and x_P (or, for example, 4_B and 4_P) for the block-number and position-number respectively. Actually x_P is represented by the right-hand three bits of the modifier (digits 11, 12 and 13) and x_B by the left-hand 10 bits (digits 1 to 10).

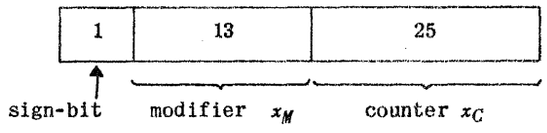


Fig. 5.1 *The counter and modifier in an accumulator*

It is convenient to introduce a notation for the composite number made up of a modifier and a counter. Suppose, for example, that the word in *X*5 has 51.6 in its modifier position (or modifier part) and 83 in its counter position, then we shall indicate this by writing

$$C(5) = (51.6, 83);$$

which means that $5_M = 51.6$, $5_B = 51$, $5_P = 6$ and $5_C = 83$. A similar notation can also be used if the modifier is an integer. The sign-bit is not indicated by this notation; it frequently does not matter whether it is a 1 or a 0.

A word having a 1 in the modifier part only may be denoted by (1, 0) or by (0.1, 0). Since the 1 appears in digit 13 such a word has the value 2^{-13} on the fractional convention. This word is permanently available in special register 35 (see Sec. 2.9); it can be used to increase a modifier by 1. The word in special register 34 has the value 2^{-10}; it may be denoted by (1.0, 0) since it has a 1 in the block part of the modifier.

Before we continue with modification we shall describe the 67-order (unit-count). The description of this order given earlier (in Sec. 3.8) was not quite accurate. The following is an accurate definition.

67 *Reduce by one the counter x_C in the specified accumulator, without affecting x_M; jump to the order specified in the N-address if the new value of the counter is not zero.*

For example, if $C(3) = (119.4, 17)$ then the order

```
1.3+ 3 67
```

will change $C(3)$ to (119.4, 16) and will cause a jump to the *b*-order in 1.3. If $C(3)$ had been (119.4, 1) then the above order would have changed it to (119.4, 0) and no jump would have occurred.

The 67-order cannot affect x_M, the modifier in the accumulator, because there is a *carry-suppression* between x_C and x_M when this order is obeyed. This matters only if the original value of x_C is zero, in which case the final value (x'_C) is $2^{25} - 1$ and a jump occurs. In fact the operation of the order does not affect, nor is it affected by, either x_M or the sign-bit in *X* or the overflow-indicator. The 67-order carries out a test on the 25 bits of x'_C only.

Consider now the orders needed to put into *X*3, say, the word in the main store location *B*51.6. The following two orders could be used.

```
51 [4] 72      B51 to U4
4.6  3 00      word in 4.6 to X3
```

In general the word in *B.P* in the main store can be put into *X*3 by means of the orders

```
B [4] 72
4.P  3 00
```

since the 72-order copies the eight words of block *B* into the eight registers of *U*4 *without changing their position-numbers*. This is the kind of operation we shall often have to do when we use modified orders. If a modifier represents *B.P*, the address of a location in the main store, we shall want *B* (the block part of the modifier) to appear as the first address in a 72-order; and we shall want the arithmetical order (00 in the above) to have the position part of its *N*-address equal to *P* (the position part of the modifier). This gives the clue to the way in which the modification of arithmetical and block-transfer orders takes place.

5.2 Modification of the arithmetical orders

The orders of groups 0, 1 and 2 are often called arithmetical orders, though this is not strictly accurate. These orders can be modified in such a way that their *N*-addresses have the *position* part of a modifier added to them. To do this we write the number of the accumulator holding the modifier in the *M* position (i.e. the modifier address) of the order. For example, suppose that $C(5) = (51.6, 83)$ and the computer obeys the order

```
4.0  6 00 5
```

Just before this order is obeyed the position part of the modifier in *X*5 (i.e. 5_P) is added to the *N*-address. In this case $5_P = 6$, so the order will have the same effect as if it had been written

```
4.6  6 00
```

With the same modifier in *X*5 the order

2.1 1 10 5

will have the same effect as

2.7 1 10

and the order

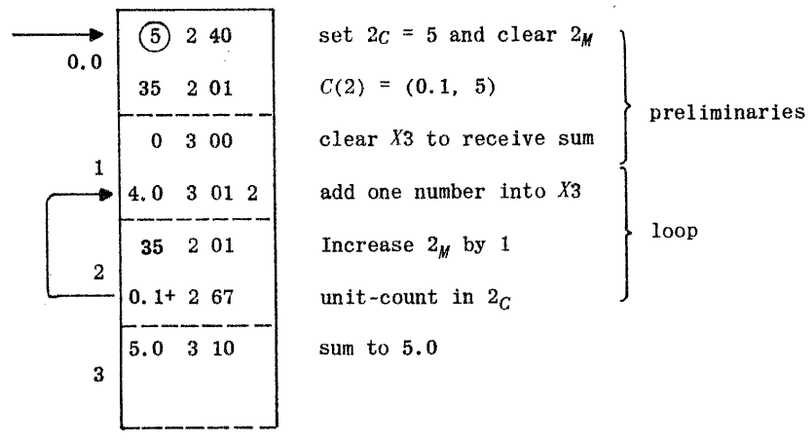
0.3 4 26 5

will have the same effect as

1.1 4 26

since 0.6 + 0.3 = 1.1 in octal arithmetic. It is important to realise that the addition of the position part of the modifier takes place *inside* the order-register just before the order is obeyed; the order is left *unchanged* in the computing store.

As a simple example, suppose we have five numbers in the ordinary registers 4.1 to 4.5, which have to be added up, and we then have to place the sum in 5.0. We shall do the addition in *X*3, which must first be cleared, and we shall use *X*2 to hold the modifier and counter. Before entering the loop we must set 2_C = 5 since the loop is to be obeyed five times, and we can put 2_M = 1 (or 2_P = 1, which is the same).



The first three orders simply clear *X*3 and set a counter and modifier in *X*2. The next order (in 0.1+) is the modified add order which adds one of the numbers into *X*3: when it is obeyed for the first time 2_P = 1 so it has the same effect as the order

4.1 3 01

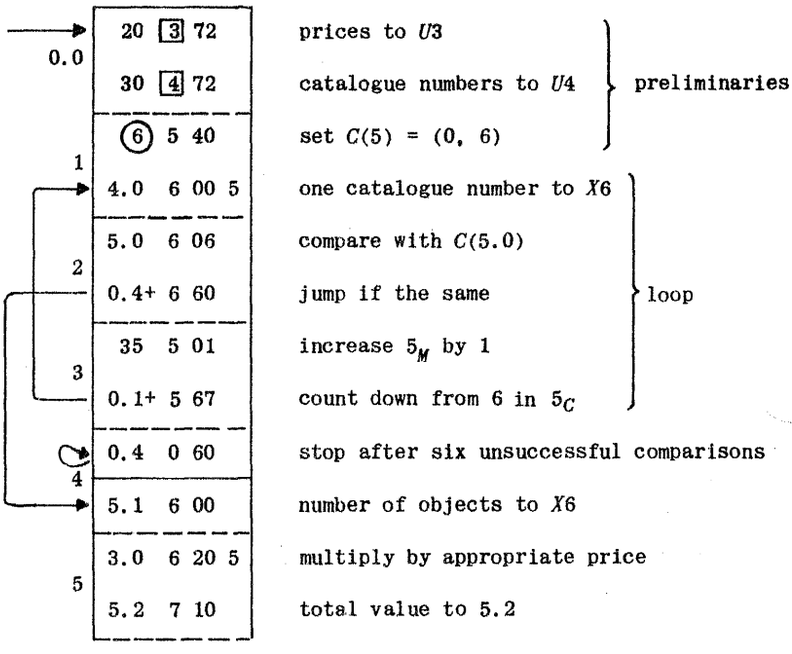
and the first number is added into *X*3. In the next order the modifier in *X*2 is increased by 1 to the value 0.2; the 67-order then reduces the counter to 4 and causes a jump back to the add order. This order now has the effect of the order

4.2 3 01

and adds in the second number. Note that the add order is stored unchanged in ordinary register 0.1; it is only in the order-register that it gets modified. The above cycle continues until the 67-order has reduced 2_C to zero, when no jump occurs. At this instant the sum of the numbers is in *X*3 and the word in *X*2 is (0.6, 0). The final value of the modifier has not been used to modify the add order in in 0.1+.

This sequence of orders is typical of most loops in that there are certain preliminary operations, followed by the loop proper and then some orders (here only one) concerned with finishing the process off before carrying on with the next part of the programme. In this example the operation to be carried out is quite trivial and it would in fact have been better (and faster) not to use a loop at all. It is instructive, however, to consider the effects of the orders as they are written above; furthermore the ideas involved can be applied to other, more complicated, loops which are better *not* written out in full.

As another small illustration, let us suppose that in 5.0 is held the catalogue number of an object, and in 5.1 is the number of these objects which are wanted. We wish to find the total value of these objects and store it in 5.2. Suppose that a price-list containing the prices of only six different objects occupies the first six locations in *B*20. The corresponding locations in *B*30 contain the corresponding catalogue numbers. What we have to do is to compare our catalogue number with those in the first six locations in *B*30; if we find one which is equal to our catalogue number then we know that the price we want is in the corresponding location in *B*20. If our catalogue number is different from all those in the list then we have to stop. The preliminary operations here consist of bringing *B*20 and *B*30 into the computing store and setting a modifier and a counter as (0, 6) in *X*5.



In the loop we use an 06-order, i.e. a not-equivalent order (see Sec. 3.11), to compare our catalogue number with the one put into *X*6 by the modified 00-order. If they are different we increase 5_M by 1 and repeat the loop up to six times. If the comparison is successful the 60-order in 0.2+ causes a jump: when this occurs the modifier in *X*5 is equal to the position-number of both the catalogue number (in *U*4 and *B*30) and the price (in *U*3 and *B*20).

5.3 Modification of the block-transfer orders; the unit-modify order

In the preceding Section we gave an example of a small loop of orders for adding up five numbers held in the computing store. By making minor changes it can be used to add up any number of numbers up to eight. We shall now show how the process can be generalized to add up a long list or set of numbers held in the main store. To do this we shall keep only one block of the numbers in the computing store and we must arrange that suitable block-read orders (72-orders) are obeyed whenever further numbers are required from the main store.

We shall at present assume that, when the computer is obeying the orders of the loop, the modifier will at each instant be the main store address of the number being added in. When the modifier block-number changes we must detect this fact and then obey a 72-order whose first address is equal to the new block-number. For this reason the block-transfer orders are modified by adding the *block* part of the modifier to the first address. As with the arithmetical orders, the address of the accumulator containing the modifier is written in the *M* part of the order, and the modification occurs only inside the order-register just before the order is obeyed.

Suppose, for example, that *C*(5) = (51.6, 83) so that 5_B = 51, and the computer obeys the order

0 [4] 72 5

Just before this order is obeyed the block part of the modifier (i.e. 5_B) is added to the first address. The order will consequently have the same effect as if it had been written

51 [4] 72

With the same modifier in *X*5 the order

27 [2] 73 5

will have the same effect as

78 [2] 73

since 27 + 51 = 78. With the same modifier in *X*5 the following two orders will put into *X*3 the word in *B*51.6 :-

0 [4] 72 5

4.0 3 00 5

since the first order will read into *U*4 the block containing the required word and the second order will pick the word out from the block. *In general, if 5_M is the address of any word in the main store, the above two orders will put the word into X3.*

When a block-transfer order has been modified, its first address is extended inside the order-register so that it can refer to any block in the main store.† The first address written in a block-

† The order as obeyed will have 10 binary digits in its first address instead of 7. There are 10 bits in the block part of the modifier; any carries beyond this are disregarded, i.e. the addition of the modifier is modulo 1024. These 3 extra digits increase the length of the order from 19 to 22 digits. The order is extended in this way only inside the order-register while it is being obeyed.

transfer order cannot exceed 127, so that unmodified orders have access to the first 128 blocks only (viz. *B*0 to *B*127). It is usual to keep the orders of a programme in the first 128 blocks together with any constants or small sets of numbers; the room available is ample for all but the largest programmes. Large sets of numbers are often stored in the rest of the main store, which can be reached only by modified orders: since we shall normally want to use modified orders with such sets of numbers, the restriction usually imposes little hardship.

As a rule we shall want to increase a modifier by unity every time a loop of orders is traversed. At the same time we usually have to take special action when the addition of unity causes the block part of the modifier (i.e. x_B) to change; when this happens we have used up all the numbers in the computing store and we must bring in a further block of them from the main store. At this moment the position-number of the modifier is zero. For example, suppose a modifier is initially 24.5 and is repeatedly increased by one, thus:

24.5
24.6
24.7 ◄——— addition of 1 causes x_B to change
25.0 ◄——— position-number zero
25.1
.
.
.

Then a new block-transfer is needed when the position-number of the modifier is zero, i.e. $x_P = 0$.

We can now describe the unit-modify order, which has the function 66.

66 *Add one to the modifier in the specified accumulator and then jump to the specified order if the new position-number in the modifier is* <u>not</u> *zero.*

Or, in terms of symbols†:

(a) $x'_M = x_M + 1$,
(b) jump to N if $x'_P \neq 0$.

This is a jump order and its N-address is written in the same way as that in any other jump order.

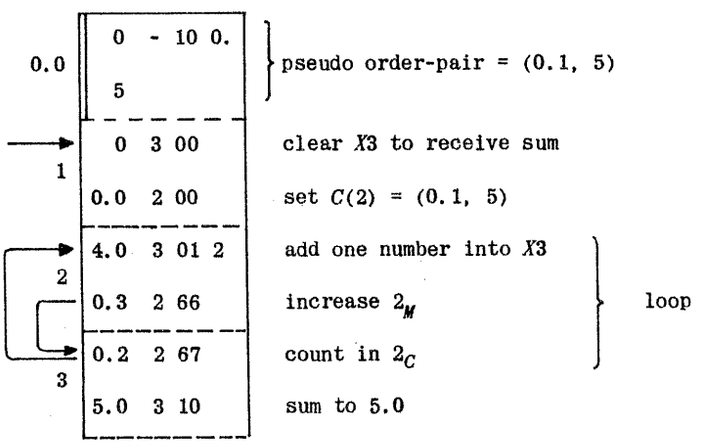
For example, if $C(2)$ = (24.5, 12) then the order

1.4+ 2 66

will leave $C(2)$ = (24.6, 12) and will cause a jump to the *b*-order in 1.4. If $C(2)$ = (24.7, 12) when this order is obeyed then it will leave $C(2)$ = (25.0, 12) and *no jump* will occur.

The 66-order senses when the end of a block has been reached because it will then not cause a jump. If we write a modified block-transfer order below the 66-order on the programme-sheet then it will be obeyed only when the addition of one to the modifier causes the block number to change.

We shall now illustrate the use of this order by extending to larger sets of numbers the example given previously for adding up five numbers in the computing store. We shall first rewrite this example using the unit-modify order, without extending its application.



The above sequence, like the one given earlier, will place in 5.0 the sum of the five numbers in 4.1 to 4.5. There are only two significant changes. The modifier and counter are set in *X*2 by copying a pseudo order-pair written in 0.0 (see Sec. 3.12). This pseudo order-pair is simply a constant which has the correct digits in its modifier and counter. We shall explain below (see Sec. 5.5) the way in which any counter and modifier can be written as a pseudo order-pair; for the present the reader is asked to accept the fact that this pseudo order-pair and those in later examples are correct. The 66-order (in 0.2+) is used here simply to increase 2_M; in this example the 67-order following it is

---

† Strictly speaking the first of these equations should be written

$$x' = x + 2^{-13},$$

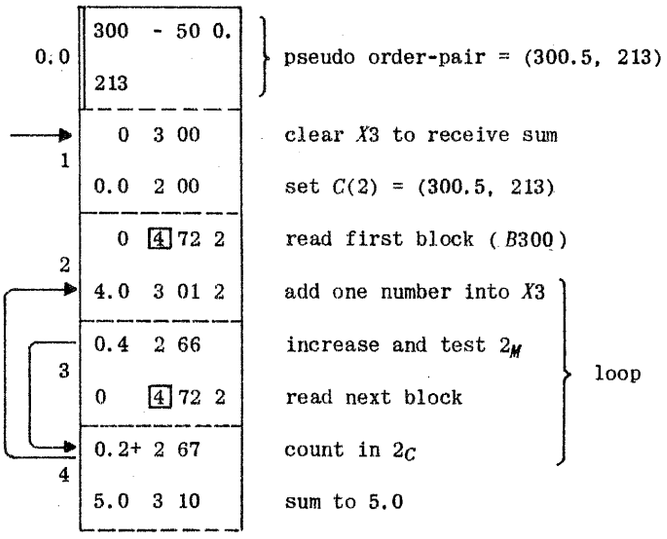
since the sign-bit in X may be affected by carry from x_M. The 66-order can set OVR, but only if the sign-bit is zero and x_M = 1023.7, i.e. "all ones".

always obeyed, whether or not the 66-order causes a jump. The only point in using a 66-order here instead of the order

35 2 01

is that the loop as given above can more easily be generalized.

Suppose now that we have a much larger set of numbers to be added together. The numbers must be held in the main store and block-transfer orders will be needed to bring them into the computing store when they are required. In order to fix our ideas, let us suppose we have to place in 5.0 the sum of the 213 numbers stored in consecutive locations starting at *B*300.5. We must now set the modifier and counter in *X*2 to the value (300.5, 213), i.e. we place in 2_M the address of the first number of the set, and in 2_C the number of numbers. We must insert a block-read order just before the loop is entered so that the first (incomplete) block of the set is copied into *U*4. We must also put in a block-read order after the unit-modify order so that further block-transfers will occur as each block of numbers is used. The appropriate sequence of orders is therefore as follows.



The first block-read order (in 0.2) will have its first address increased by the block-number in the modifier (i.e. 2_B) so that it will copy *B*300 into *U*4. When the add order in 0.2+ is first obeyed 2_P, the position-number in the modifier, is 5 and it will therefore add the word in 4.5 into *X*3. After this a unit-modify (66) order is obeyed; this increases the modifier from 300.5 to 300.6 and causes a jump since the new position-number (i.e. 2_P) is not zero. This jump goes to a unit-count (67) order in 0.4 which reduces the counter in *X*2 (i.e. 2_C) from 213 to 212 and jumps (back to the add order) since the new counter is not zero. At this stage *C*(2) = (300.6, 212) and the first number has been added into *X*3. Subsequent events are best exhibited in tabular form (Table 5.1).

Order	*C*(2)	Effective order (if modified)	Notes
4.0 3 01 2	(300.6, 212)	4.6 3 01	Second number added into *X*3.
0.4 2 66	(300.7, 212)		Jump occurs since $2_P = 7 \neq 0$.
0.2+ 2 67	(300.7, 211)		Jump since $2_C = 211 \neq 0$.
4.0 3 01 2	(300.7, 211)	4.7 3 01	Third number added in.
0.4 2 66	(301.0, 211)		No jump since $2_P = 0$.
0 4 72 2	(301.0, 211)	301 4 72	Second block of numbers brought into *U*4.
0.2+ 2 67	(301.0, 210)		Jump since $2_C = 210 \neq 0$.
4.0 3 01 2	(301.0, 210)	4.0 3 01	Fourth number added in.
0.4 2 66	(301.1, 210)		Jump sine $2_p = 1 \neq 0$.
0.2+ 2 67	(301.1, 209)	4.1 3 01	Jump since $2_C = 209 \neq 0$.
4.0 3 01 2	(301.1, 209)	4.1 3 01	Fifth number added in.
.	.	.	.
.	.	.	.
.	.	.	.

Table 5.1 *The orders in a simple loop.*

Points to note are:-

(a) only the position part of the modifier is added when the 01-order is modified,

(b) only the block part of the modifier is added when the 72-order is obeyed,

(c) the 72-order is obeyed on the average only once in every eight times round the loop,

(d) the 66-order and the 67-order are obeyed every time the loop is traversed.

The computer leaves the loop when the 67-order reduces 2_C to zero, when no jump will occur. At this instant the word in $X2$ will be (327.2, 0) but this will not have been used to modify any orders. The last modifier used is 327.1 and this causes the last number of the set to be added into $X3$.

In this example the modifier and counter can have any initial values; we need only write the appropriate pseudo order-pair in the programme. Apart from this change we can use the same sequence of orders to add up any number of consecutively stored numbers placed anywhere in the main store. Despite its generality, the above sequence of orders differs from that given earlier for adding together five numbers only by the insertion of the two block-transfer orders. This shows the power of this modifier-and-counter technique in linking together the computing store and the main store. It is important that only one modifier and one counter are needed to enable the programme to work progressively through both stores.

Another important point is that we need give no thought to the way in which the set of numbers is divided into blocks. In the example above the set of 213 numbers does not fit neatly into blocks; in fact the first number is in 300.5 and the last in 327.1. We could equally well write these addresses in their decimal form and say that the first number is in location 2405 and the last in 2617. *When dealing with sets of numbers we can visualize the main store as a continuous strip of storage locations numbered 0 to 7167.* The division into blocks can often be entirely disregarded. The orders which make up the programme are best thought of as grouped in blocks, however.

In the above example, the modifier is at any moment equal to the main store address of the number being dealt with and the counter shows how many numbers have still to be added in. This is a typical situation. The address was given in its block-and-position form because this made the explanation simpler. Since, however, the division into blocks can be disregarded here, the address could be given in decimal form; the only change in writing the programme is to write in a different way the pseudo order-pair giving the modifier and counter. In fact we can write it like this

$$0.0\left\{\begin{array}{|l|}\hline 2405\ \ -\ -0\ 0. \\ \\ 213 \\ \hline\end{array}\right\} = (2405,\ 213)$$

as we shall explain below (see Sec. 5.5).

5.4 Modification of the single-word transfer orders

Before giving any further examples of the use of modified orders we shall describe how the single-word transfer orders (functions 70 and 71) are modified, because they fit easily into the picture. As we have mentioned above, a modifier can usefully be thought of as the address of a location in the main store. Only the block part of this address is used in modifying block-transfer orders, and only the position-number is used in arithmetical orders. As we have just seen, these two methods of modification can be used together very satisfactorily. With the single-word transfer orders the whole of the modifier is used.

We described in Section 3.10 how the N and X addresses in a 70- or 71-order are used to specify the block-and position-number of the location involved in the transfer. When one of these orders is to be modified the address of the accumulator holding the modifier is written, as usual, in the M-part of the order. When the order is about to be obeyed the main store address specified by its N and X addresses is increased by the whole of the modifier. For example, if $C(5)$ = (51.6, 83) then the order

10 1 70 5

will have the same effect as the order

61 7 70

i.e. the word in $B61.7$ will be copied into $X1$. With the same modifier, the order

8 4 71 5

will have the same effect as the order

60 2 71

which places a copy of $C(1)$ into $B60.2$.

Note that, as far as $X1$ is concerned, the order

0 0 70 3

is equivalent to the orders

0 5 72 3

5.0 1 00 3

since the content of the location whose address is 3_M will be copied into $X1$.

If we like we can use the decimal forms of the modifier and the address in the single-word transfer order; this affects the way in which the orders and modifiers are written (but not, of course, their stored forms). The decimal forms of 51.6 and 10.1 are respectively 414 and 81. If $C(5)$ = (414, 83) then the order

81 -	70 5

has the same effect as the order

495 -	70

since 414 + 81 = 495.

As with the block-transfer orders, the first address is extended by 3 binary digits in the order-register. In this way the address has 13 digits and can refer to any location in the whole of the main store; whereas unmodified single-word transfer orders have access only to the locations in the first part of the store (i.e. $B0.0$ to $B127.7$).

As an example, let us rewrite, using single-word transfers, the sequence given earlier for placing in $U5.0$ the sum of 213 numbers stored in the main store in consecutive locations starting at $B300.5$. The modifier and counter needed are exactly the same as before, and the sequence reads as follows.

0.0	300 - 50 0. 213	pseudo order-pair = (300.5, 213)
1	0 3 00	clear $X3$ to receive sum
	0.0 2 00	set $C(2)$ = (300.5, 213)
2	0 0 70 2	read one number into $X1$
	1 3 01	add number into $X3$
3	35 2 01	increase 2_M by 1
	0.2 2 67	count in 2_C
4	5.0 3 10	sum to 5.0

We could alternatively have used a 11-order to build up the sum directly in 5.0.

This sequence will, of course, be much slower than the version using block-transfers since it requires access to the main store every time the loop is traversed. In fact the earlier sequence will be about 7 times as fast. For this reason we prefer block-transfers to single-word transfers whenever they can usefully be used. This is not always possible; for example we may not require access to the numbers in the order in which they are stored, or there may not be enough room in the computing store.

As an example of the combined use of modified orders of all the three kinds so far described let us generalize the second example of Sec. 5.2. We shall suppose we have to put in $U5.2$ the total value of n identical objects whose catalogue number is stored in 5.0; the number of objects (n) is in 5.1. Suppose that a list of the 491 possible catalogue numbers is stored in random order in the main store starting at the location whose decimal address is 2339 and that the corresponding prices are stored in the next 491 locations. Thus, for example, the object whose catalogue number is stored in 2400 has its price in 2891 = 2400 + 491. We must search the list of catalogue numbers until we find one which is the same as the one in 5.0 (if there is no such number in the list we must stop). We can then extract the appropriate price from the location 491 ahead of that containing the catalogue number, multiply it by n and put the result into 5.2.

0.0	2339 - -0 0. 491	pseudo order-pair = (2339, 491)	
1	0.0 5 00	set $C(5)$ = (2339, 491)	
	0 4 72 5	first few catalogue numbers to $U4$	
2	4.0 6 00 5	one catalogue number to $X6$	loop
	5.0 6 06	compare with $C(5.0)$	
3	0.5+ 6 60	jump if the same	
	0.4+ 5 66	increase and test 5_M	
4	0 4 72 5	more catalogue numbers to $U4$	
	0.2 5 67	unit count in 5_C	
5	0.5 0 60	stop after 491 unsuccessful comparisons	
	491 - 70 5	price of one object to $X1$	
6	5.1 1 20	multiply by number of objects	
	5.2 7 10	value to 5.2	

The average time per comparison is about 2.4 milliseconds. The reader should note the differences between this example and the more restricted earlier version.

5.5 Setting modifiers and counters

We often require to set particular modifiers and counters, usually just before entering a loop. Often the modifier is the main store address of the first of a set or list of numbers, or it may be a few blocks less than this address. For example, the first number of a set may be in 300.5 and we can set a modifier, say 2_M, equal to this and enter a loop containing orders such as

0 [4]72 2

But if we wished we could set the modifier equal to, say, 261.5 and write the block-transfer orders as

39 [4]72 2

since 261.5 + 39.0 = 300.5. Provided the modifier is a whole number of blocks less than the required address this will not upset the operation of the loop. Since, however, the first address written in a block-read order may not exceed 127 we must use a modifier which is at most 127 blocks less than the address. For example instead of 300.5 we could set 2_M = 173.5 and use the order

127 [4]72 2

but no smaller modifier could be used. If our list of numbers begins at the beginning of a block in the first part of the store the initial value of the modifier can be zero; this is very convenient since we can then often use a 40-order to set both modifier and counter simultaneously.

The best way of setting an arbitrary modifier and counter is often to use a pseudo order-pair. We have already given some examples of these; thus

300 - 50 0. 213	= (300.5, 213)
2339 - -0 0. 491	= (2339, 491)

In each case the "*b*-order" is simply the value of the counter and the "*a*-order" specifies the modifier. When the modifier is to be given in block-and-position form *B.P* we write the block-number *B* in the *N*-address part, a minus sign in the *X*-address part, *P* followed by zero in the *F*-part and zero in the *M*-part. If the block-number is relative we simply write a plus sign after it. The above pseudo order-pairs are written as stop order-pairs since this makes their sign-digits zero inside the computer. Frequently it does not matter whether the sign-digit is 1 or 0 since it does not form part of either the modifier or the counter. If a sign-digit of 1 is acceptable the first of the above pseudo order-pairs may be written

300 - 50 213	= (300.5, 213)

If the pseudo order-pair is to have its modifier part given in decimal form we write the modifier in the *N*-address part, a minus sign in the *X*-address part, the "function" is –0 and the *M*-part is zero. Inside the computer the modifier occupies 13 binary digits and so may refer to any location in the whole of the main store; there is no restriction to the first part of the store.

We often write 5.7_M and 5.7_C, for example, to mean the modifier and counter in ordinary register 5.7. Of course such a word would have to be copied into one of the accumulators before it could be used to modify any orders.

It should be remembered that a modifier in an accumulator is simply the number represented by digits 1 to 13. Sometimes we have to calculate the modifier we need (this happens, for example, in "floating-point" work) at the least-significant end of some accumulator. An upward shift of 25 places will then move it into the modifier position, for example, the orders

(93) 2 40 93 to *X*2

(25) 2 50 93 to 2_M

have the effect of setting *C*(2) = (93, 0). We could get a modifier of 93.0 by using a shift of 28 places. When forming a modifier in this way, we usually use the logical shift (52-order) for shifting up 25 places (termed the counter-to-modifier shift), as this is a fast order taking the same time as a group 0 order (whereas the order (25) 2 50 takes 25 extra word-times). This method has the added advantage that the 52-order cannot set OVR. It may happen that a modifier has been computed at the least-significant end of *X*2, say, from counters in other accumulators; this may result in an unwanted number appearing in 2_M which would cause overflow if shifted up 25 places with a 50-order.

The following orders set $C(2)$ = (328.0, 0):-

(82) 2 40

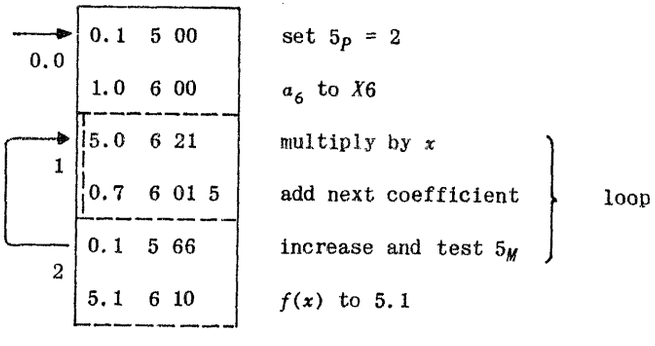
(30) 2 52

In this example, the 52-order causes a slow shift of 5 places followed by a 25-place fast shift, thus the order takes 5 extra word-times.

▼ Sometimes a modifier is used to modify arithmetical orders only, in which case only its position-number is relevant. Modifiers like this usually occur in small loops which have to be traversed at most eight times; in these loops we can often use the modifier for counting. The technique is to set the position-number in some modifier equal to 8-n, where n is the number of times the loop is to be traversed; at the end of the loop we count using a 66-order. As an illustration of this consider the evaluation of a function by means of a polynomial approximation

$$f(x) = a_0 + a_1x + a_2x^2 + \ldots + a_6x^6,$$

where all quantities concerned are fractions and there is no possibility of overflow. Suppose x is in 5.0, $f(x)$ is to be placed in 5.1, and the coefficients are stored in $U1$, with a_6 in 1.0, a_5 in 1.1, ..., a_0 in 1.6. The following sequence of orders can be used.



The first order copies $C(0.1)$ into $X5$; the only part of this word which is used is the first function digit in the a-order, viz. 2, which occupies the three bits in 5_P (digits 11, 12 and 13 in $X5$). The next order puts a_6 into $X6$. This gets multiplied by x, and we next obey the modified 01-order. When this order is obeyed for the first time 5_P = 2 so it will add a_5 from 1.1 into $X6$. The 66-order next increases 5_P to 3 and causes a jump. The whole process is repeated until, just after adding a_0 from 1.6 into $X6$, the 66-order does not cause a jump. Note that the word in 0.1 has a dual purpose; the broken line on its left shows that it is used as a constant, as well as an order-pair, and serves as a ▲ warning in case an attempt is made to alter it. This subject is further discussed in Sec. 5.10.

5.6 Some standard loops of orders

In Sec. 5.3 we showed how to construct a simple loop of orders for adding up a long list of consecutively stored numbers. In this example the numbers were brought by blocks into the computing store, where they were used to build up the desired result. It is clear that a very similar loop of orders could be used to obtain some other result depending on all the numbers; all we have to do is to replace the add order

4.0 3 01 2

by one or more other orders. For example, suppose we wish to find the sum of the squares of all the numbers. We can replace the add order by the following two orders.

4.0 3 00 2 place one number in $X3$

3 3 22 add square of number into (pq)

These will build up the result in the double-length accumulator, $X6$ and 7. Of course, before we enter the loop there is no need any longer to clear $X3$; but we must instead clear the double-length accumulator by the order

0 0 20

unless the result is to be a single-length rounded fraction in $X6$, when this preliminary order should be

0 0 21 $(pq)' = \frac{1}{2}\epsilon$

to place the round-off constant in the double-length accumulator.

The second example in Sec. 5.4 (to search a list for a given catalogue number) was also similar in construction. Here the relevant orders were

4.0 6 00 5 one catalogue number to $X6$

5.0 6 06 compare with $C(5.0)$

0.5+ 6 60 jump if the same

Any loop or cycle of orders such as these can be written in an abbreviated notation, which brings out the salient points:

```
Set initial conditions

Set (A0, n) in X2

0 □72 2                          read A

Operate (modified by 2M)

    2 66
    0 □72 2                      read A
    2 67

Deal with result
```

Here the first line refers to any necessary clearing of accumulators, etc. The second line indicates the setting of a modifier and a counter in *X*2, though any other accumulator could be used; A_0 is the main store address of the first number of the set (set *A*) and *n* is the number of numbers in the set. The word *operate* represents the add order, or whatever other orders are needed in the loop to process a single number of the set; some of these orders will be modified by 2_M.

We shall now use this notation to explain how we can construct more complicated loops. We shall assume throughout that the numbers of each set are stored consecutively.

For the present we shall ignore the possibility of overflow (the programme will, of course, come to a stop on overflow if the Stop On Overflow key is depressed). If overflow is considered likely then OVR can be tested by a 64- or 65-order written immediately after the *operate* orders and which may be considered a part of them. It is often better, however, to test OVR on exit from the loop after the 67-order. If the loop contains any 71- or 73-orders these will cause a stop if OVR gets set.

The first example concerns two sets of numbers, *A* and *B*. Let us denote the numbers of set *A* by $a_1, a_2, a_3, \ldots$, and the numbers of the equally long set *B* by $b_0, b_1, b_2, \ldots$. We may have to form the sum of the products of corresponding numbers:

$$\Sigma a_i b_i = a_0 b_0 + a_1 b_1 + a_2 b_2 + \ldots$$

We are not here concerned with the significance of this operation; the result may be a scalar product of two vectors, or (if the *a*'s are quantities and the *b*'s prices) it may be a total value. We shall need two modifiers, each associated with one of the sets of numbers, and each with its own 66-order and block-read order in the loop. There can be a single counter with its 67-order at the end of the loop.

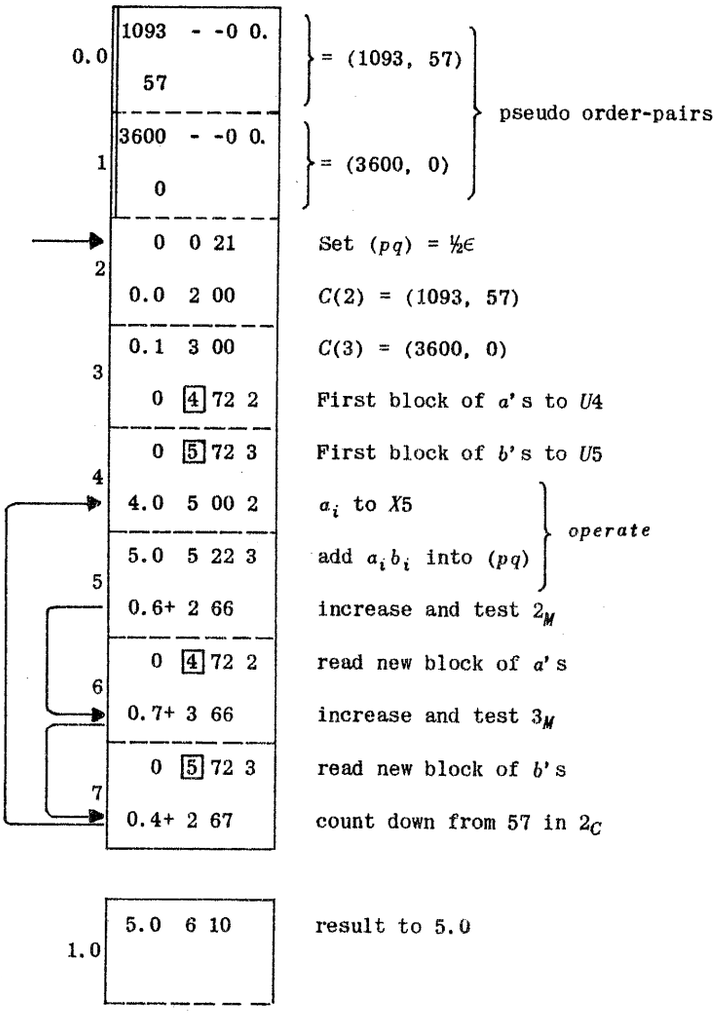
```
Set initial conditions

Set (A0, n) in X2

Set (B0,  ) in X3

0 □72 2                          read A

0 □72 3                          read B

Operate (modified by 2M and 3M)

    2 66
    0 □72 2                      read A
    3 66
    0 □72 3                      read B
    2 67

Deal with result
```

The counter can clearly be stored with either modifier, or indeed in a separate accumulator. There is clearly no necessity to use *X*2 and 3 to hold the modifiers; other accumulators could equally well have been chosen. The modifiers are each increased by unity every time round the loop; at the beginning they are set to A_0 and B_0, the addresses of the first numbers of the two lists.†

We shall now illustrate this by writing down a sequence of orders for the evaluation of the sum of the products when each set consists of 57 fractions, set *A* starts at location 1093 and set *B* at 3600. We shall use blocks *U*4 and 5 to hold the numbers of the sets and we shall place the result in *U*5.0.

---

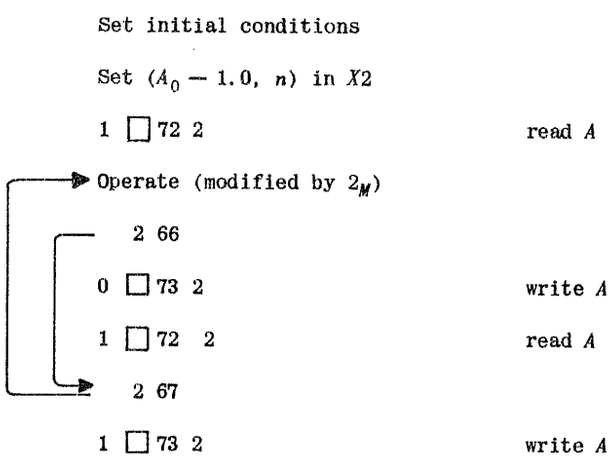
† The initial content of *X*3 is here indicated as $(B_0, \ )$. This means that 3_M must be B_0, and that the counter 3_C and the sign-digit may have any values.



In hand computation it is usual to round the results at the end of the process. In this sequence of orders it was found more convenient to add the rounding constant at the beginning; this will clearly not affect the result since it is immaterial in which order the additions are effected.

Very similar sequences can easily be constructed for evaluating some number depending (in a symmetrical way) on more than two sets of numbers. There will be a modifier and a block in the computing store associated with each of the sets. In the loop there must be a 66-order for each modifier; and at the end a single 67-order. Each 66- or 67-order must be obeyed every time the loop is traversed. In these loops there will be modified 72-orders, but no 73-orders.

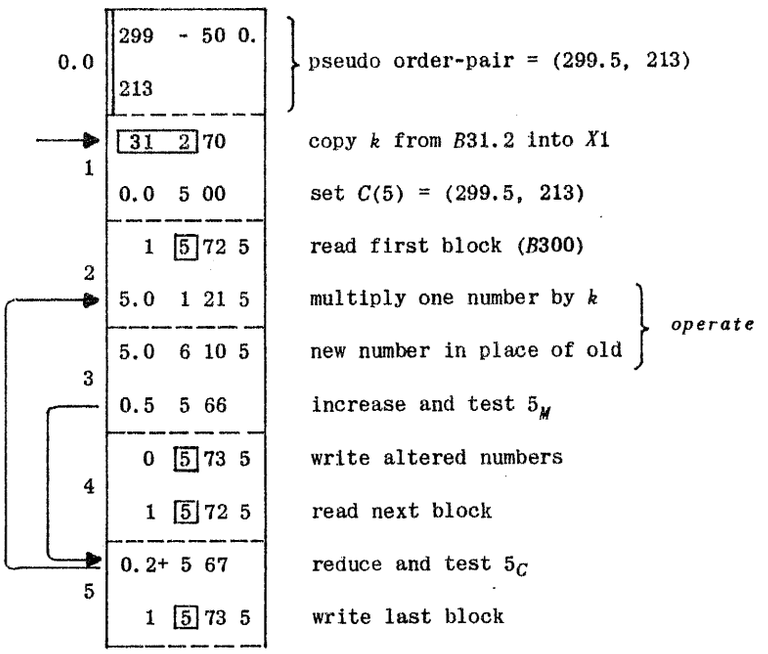
We shall now discuss loops involving writing transfers as well as reading ones. Consider, for instance, the problem of altering all the numbers of a set in some systematic way and replacing them in their original locations. For example, we may have to multiply each of them by a constant. The loop of orders will be generally similar to those described above but the 66-order will now be followed by a modified 73-order (to write up the newly altered numbers) as well as a modified 72-order (to read the next block of unaltered numbers). The effective address in the 73-order must be one block less than that in the 72-order. This means that the modifier will now have to be 1.0 less (i.e. one block less) than the address of the number being processed. Such a loop may be represented as follows.



In this loop the first 72-order reads in the first block (perhaps only partially used) of the set of numbers; the *N*-address written in the order must be 1 to compensate for the smaller modifier. The *operate* order(s), being arithmetical (if modified by 2_M), will not be affected by the block part of the

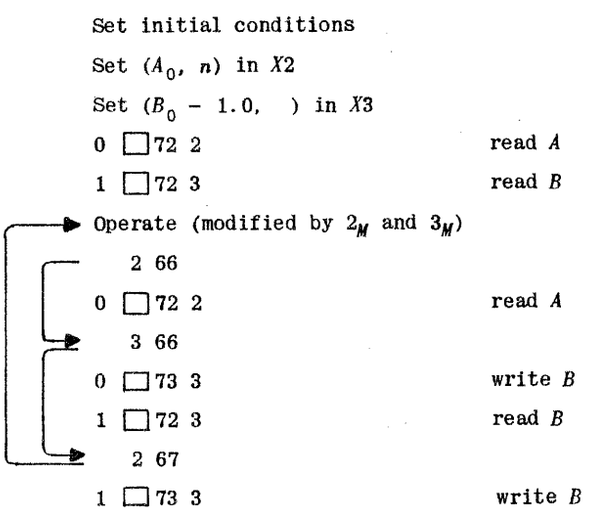
modifier. The two orders after the 66-order will be obeyed immediately after the processing of the last number in each block. At this instant the block part of the modifier (i.e. 2_B) will just have been increased by 1 so the 73-order (whose N-address is written as 0) will write the numbers back into the block from which they were originally read. The 72-order immediately following will then read in the next block. Note that we now need an extra 73-order at the end, after the 67-order, in order to write the last block of numbers back into the main store. This must have an N-address written as 1 since the block part of the modifier will not have changed since the last 72-order. This 73-order is actually redundant (but harmless) if the last number of the set happens to come at the end of a block; when this number has been dealt with neither the 66-order nor the 67-order will cause a jump and the three block-transfer orders will be obeyed successively without the modifier being altered.

As an example, suppose that the 213 fractions stored in consecutive locations starting at $B300.5$ have each to be multiplied by some constant factor k which is stored in $B31.2$. Since the modifier must be one block less than the address of the number being processed we shall need (299.5, 213) as our modifier and counter; this will be kept in $X5$. We shall use $U5$ to hold a block of the numbers and $X1$ to hold k, the constant multiplier.



The reader will find it instructive to draw up for this sequence of orders a table analogous to Table 5.1, showing the events near the beginning.† As with the earlier illustrations, this sequence of orders is quite general in the sense that the only change needed to process any number of consecutively stored numbers is a slight alteration in the word specifying the modifier and counter.

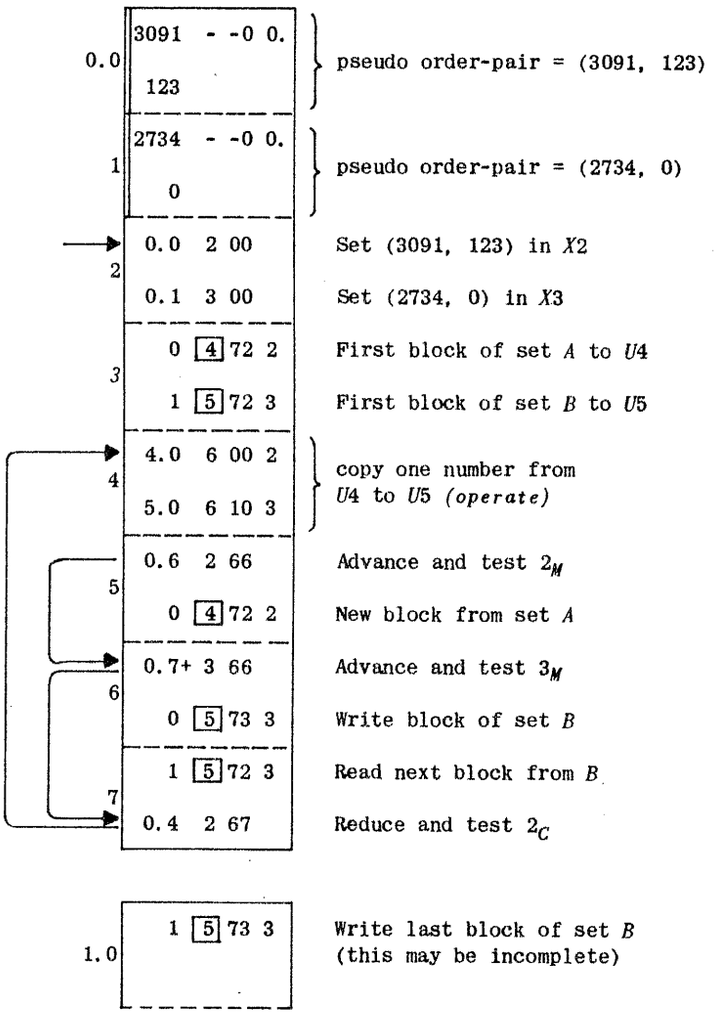
Another common loop is one involving the reading of one set and the writing of another; for example the numbers of set A may have to be copied into another part (B) of the store, perhaps being changed in some way before being stored. Because there are two addresses concerned we shall again need two modifiers. One of these we shall at first set equal to A_0, the address of the beginning of list A; we must set the other modifier equal to $B_0 - 1.0$, i.e. to an address one block before the start of set B; this is because we are writing with this set. As before, we shall need one counter, which can be put into the same accumulator as one of the modifiers. There will be a 66-order (unit-modify) associated with each of the modifiers, and the loop will be terminated with a 67-order (unit-count). The loop may be written as follows in abbreviated notation.



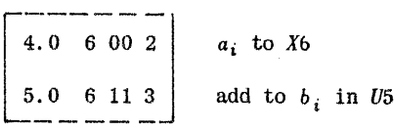
† The end is also interesting. The reader may like to take $C(5) = (325.6, 4)$ and start at the order 0.2+. The last number of the set is in $B327.1$.

If this loop is being used merely to copy n numbers from A to B, the two block-read orders reading from B are at first sight unnecessary. However, the numbers of set B will not as a rule fit neatly into blocks, and there will therefore be two incomplete blocks containing the first and the last numbers of set B. If the two orders reading from set B are omitted then the remaining locations in these two blocks will be overwritten.

As an example, suppose we have to copy 123 numbers from consecutive locations starting at that with decimal address 3091 into 123 locations starting at 2742. Here A_0 is 3091, B_0 is 2742 and n is 123; we shall use $U4$ and $U5$ to hold the numbers of sets A and B respectively. The modifier associated with set B must be initially $B_0 - 1.0 = B_0 - 8 = 2734$.



A sequence very similar to this can be used to combine the numbers of two sets. Suppose, for example, we wish to add to each number of set B the corresponding number of set A, i.e. each b_i is to be replaced by $a_i + b_i$. All we have to do is to replace the *operate* orders (the order-pair in 0.4 above) by the following orders:-



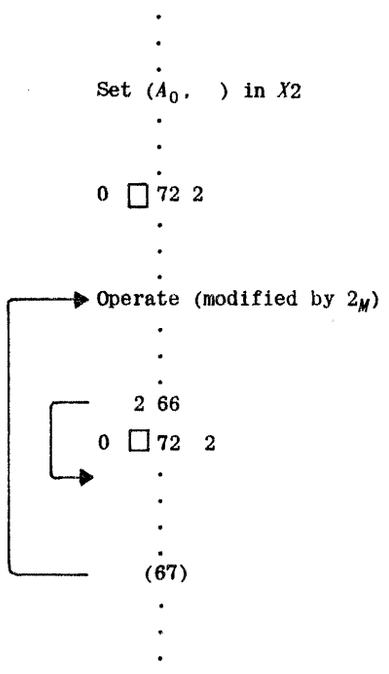
The rest of the sequence is unaltered. A sequence of orders similar to this can be used to add or subtract two vectors or matrices.

Study of these examples shows how we can easily build up many of the more common loops of orders. In practice we usually start by writing the *operate* orders, which deal with just one number from each set. After this we can add the organisational orders needed to count, advance the modifiers and transfer the numbers to and from the main store as needed. And finally we put in the preliminary orders to be obeyed before the loop can be entered, and the orders which have to follow the loop to finish off the process. When the sequence has been sketched out in this way it is usually easy to fit it into blocks and write in certain addresses, e.g. those in jump orders, which will have been perforce left blank.

In general, for sets of numbers starting at arbitrary addresses, there must be one modifier associated with each set. Each of the sets has certain organisational orders associated with it; these can largely be considered separately from those belonging to the other sets.

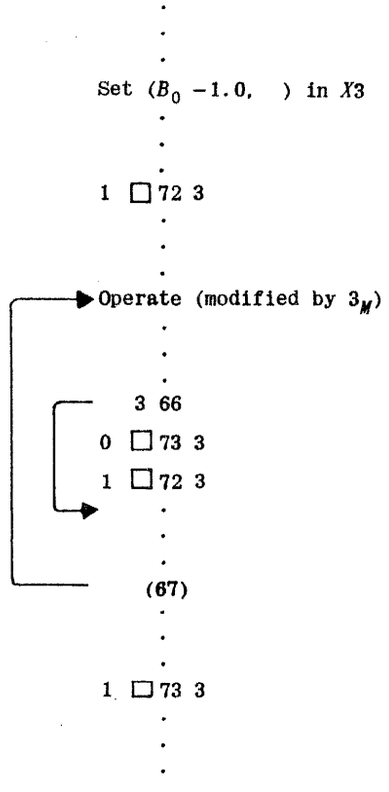
Consider first a set which requires only reading transfers (72-orders). There will be a modifier, in $X2$ say, which must initially be put equal to the address of the first number of the set. Before entering the loop we must also have a modified 72-order to read in the first few numbers of the set.

In the loop, after the *operate* orders, we must have a 66-order followed by another modified 72-order. We can therefore write down as follows the orders associated with this set of numbers:-



Here the dots represent organisational orders associated with the other sets of numbers (if any), which do not significantly affect the orders belonging to this set. Note that both the 72-orders have their *N*-addresses zero.

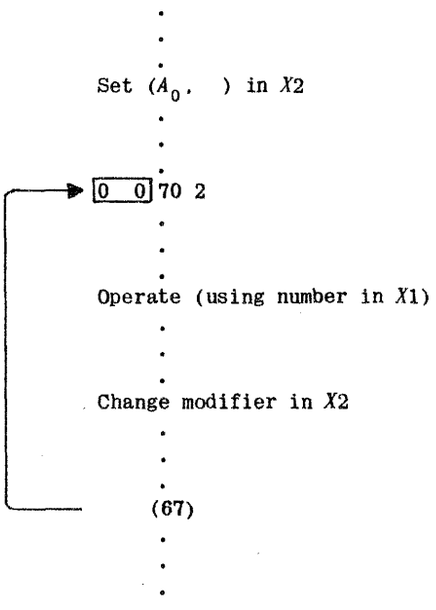
Consider now the orders associated with a set of numbers which has to be written into the main store. In general we must put in certain block-read orders as well as the block-write orders, even if we do not actually need the numbers being read in. This is because we must avoid writing over certain other locations in the main store, viz. those immediately preceding the first numbers to be written, and those immediately following the last. This means that the organisational orders for a set which has to be *written* differ in no way from those for a set which has to be both *read and written*. With either kind of set the modifier, in *X*3 say, must initially be put at one block (or eight locations) earlier than the start of the set. Before entering the loop we must have a modified 72-order to read in the block containing the first few numbers of the set. In the loop, after the *operate* orders, we must have a 66-order followed by a 73-order and a 72-order. At the end of the loop, after the 67-order, there must be a 73-order to write up the block which was last read in. The orders can be sketched out as follows.



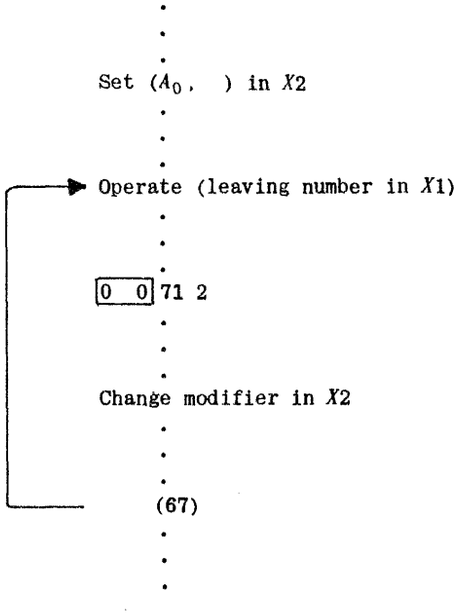
The dots again indicate the orders associated with any other sets of numbers. Note that both the 72-orders have their *N*-addresses equal to 1; the 73-order following the 66-order has zero *N*-address.

The last 73-order (after the end of the loop) can be thought of as writing up the block read by the last 72-order, and it must therefore have 1 for its *N*-address.

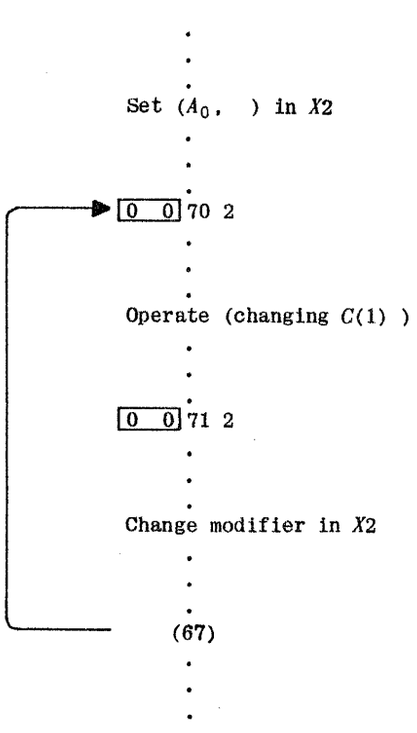
Sometimes it is necessary or desirable to use single-word transfers with one or more of the sets of numbers. This may happen because we do not want to use the numbers in the sequence in which they are stored, or perhaps because there is a large amount of calculation associated with each number and the extra waiting time entailed by the use of single-word transfers may not matter, or again because we may not be able to spare the use of a whole block in the computing store. The organisational orders associated with a set of numbers which is read by single-word transfers can be sketched out thus:



Here 2_M (or whichever modifier is to be used) is initially put equal to the address of the first number to be dealt with. The modified 70-order is written in the loop immediately before the *operate* orders. There is an example of this kind of loop in Sec. 5.4. If we are simply writing the numbers and not reading them (e.g. because they are being computed from other numbers) then we must put the single-word write order after the *operate* orders, thus;



Note that we do not now have to put in reading transfers, because there is no possibility of overwriting other numbers. If we wish to read the numbers, alter them in some way and then write them back we must write the organisational orders as follows:

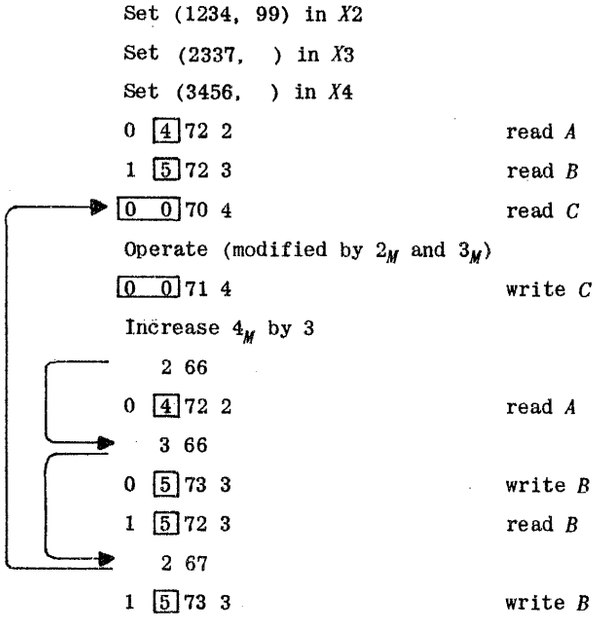


As an illustration of the way in which the various groups of organisational orders may be combined, we shall construct a loop of orders dealing with three sets of numbers A, B, C. The sequence will be a little artificial but will illustrate the ideas concerned. We shall write a_i, b_i, c_i to denote typical corresponding numbers of the three sets, and we shall assume they are all non-negative integers. The problem is to form the product $a_i b_i$ and compare it with c_i; we put the larger of these two numbers in place of c_i and the smaller in place of b_i; but if c_i is zero we leave b_i and c_i unaltered. Symbolically we can write

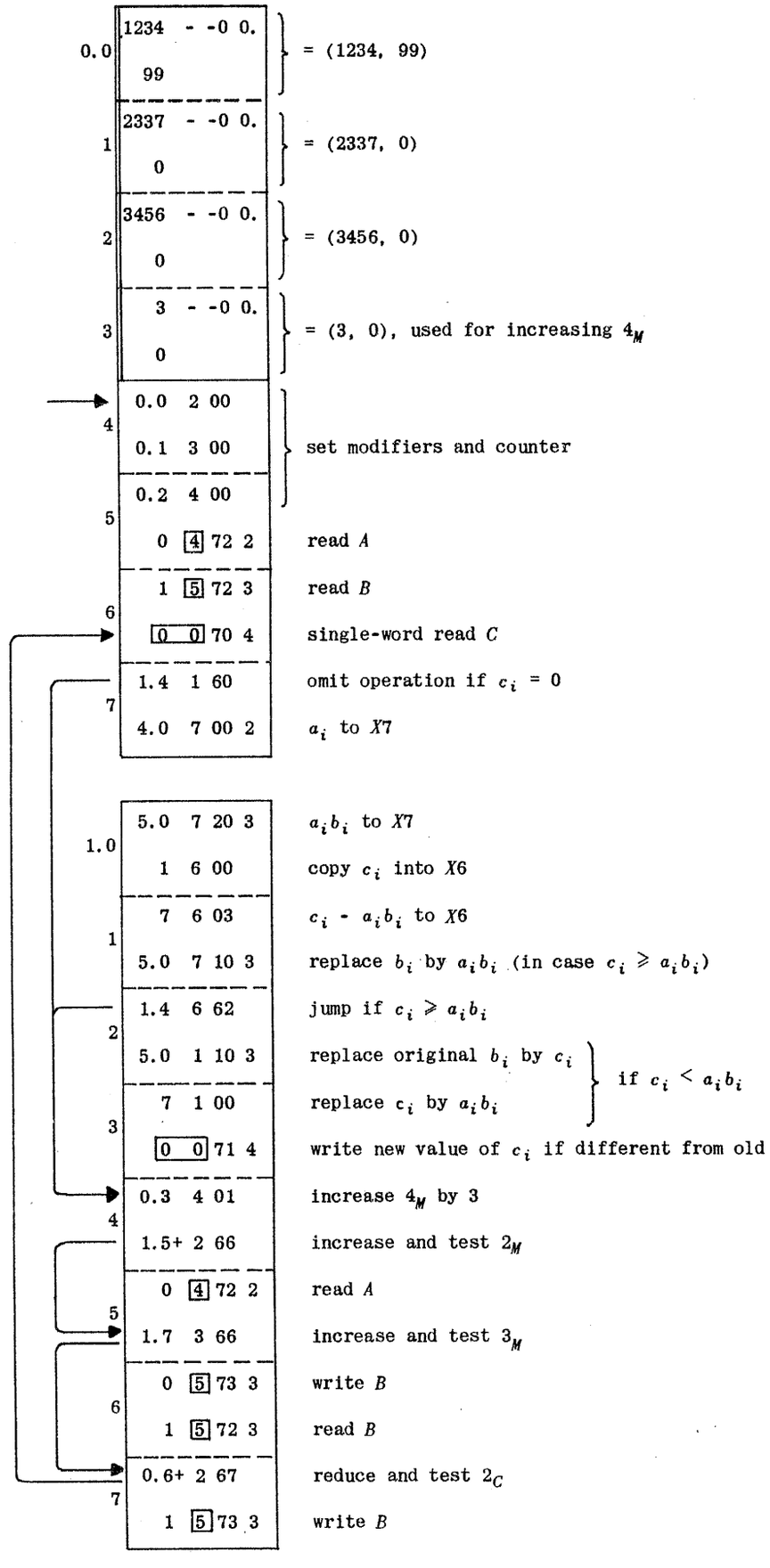
$$\left.\begin{aligned} c_i' &= \max\,(a_i b_i,\ c_i) \\ b_i' &= \min\,(a_i b_i,\ c_i) \end{aligned}\right\} \text{if } c_i \neq 0,$$

$$\left.\begin{aligned} c_i' &= c_i \\ b_i' &= b_i \end{aligned}\right\} \text{if } c_i = 0,$$

where, as usual, primes (dashes) denote values after the operation. We shall have to read the numbers of set A, but sets B and C will have to be read and written. We shall suppose that each set contains 99 numbers and the numbers of sets A and B are stored consecutively, but that the numbers of set C are spaced 3 locations apart in the main store. We assume that the first numbers of each set are in the locations with decimal addresses 1234, 2345 and 3456 respectively. Since the numbers of set C are not consecutively stored we shall use single-word transfers with them. The modifier for set B must be initially put equal to 2345−8 = 2337 since we are using 73-orders to transfer the numbers. In abbreviated notation the sequence may be written as follows.



Here we have indicated explicitly the fact that we are going to use *U*4 and *U*5 for the numbers of sets *A* and *B*. Of course there is a certain amount of freedom about the sequence of events, for example, we could interchange the groups of orders for sets *A* and *B*, or we could put the orders for writing up c_i (the 71-order and the order increasing 4_M) just before the 67-order, say. In the sequence of orders we assume that the product $a_i b_i$ is a single-length integer.



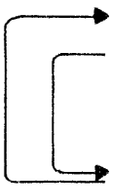
In practice, as mentioned above, we start writing such a loop with the *operate* orders (in this case the orders from 0.7 to 1.3), and then add the organisational orders and constants and fill in the *N*-addresses.

5.7 Some special loops

With few exceptions we have so far assumed that the sets of numbers being processed have started at quite arbitrary places in the main store and have been consecutively stored with their numbers in the sequence in which we needed access to them. In this Section we shall consider some loops in which none or only some of these conditions apply.

It is sometimes advantageous to store several sets of numbers *in parallel*, i.e. in such a way that corresponding numbers occupy corresponding positions in blocks - often the first numbers of each set will start a block. When this is done we can often use only one modifier for several sets, since all the transfer orders will need to be obeyed at the same moment. For example suppose set *B* starts exactly 80 blocks after set *A*; a loop involving the reading of both sets could have the following structure.

```
Set initial conditions
Set (A0, n) in X2
 0 □72 2                    read A
```



```
Operate (modified by 2M)
    2 66
 0 □72 2                    read A
80 □72 2                    read B
    2 67
Deal with result
```

This technique can be used whenever the starting addresses of the two sets differ by at most 127 blocks. If the numbers of set *B* have to be written then the loop is made up as follows.

```
Set initial conditions
Set (A0, n) in X2
 0 □72 2                    read A
80 □72 2                    read B
Operate (modified by 2M)
    2 66
79 □73 2                    write B
80 □72 2                    read B
 0 □72 2                    read A
    2 67
80 □73 2                    write B
```

Note that in either of these loops an order can be saved if the counter is initially set to $n + 1$ instead of n; the pair of 72-orders before the loop is entered can then be replaced by a jump to the first of the 72-orders inside the loop. With the above loops the 66-order and the 67-order are obeyed every time the loop is traversed.

If the *operate* orders are very quick it may be possible to speed up the loop still further by storing sets in such a way that the last number of each set occupies the last location in a block. We arrange the loop so that only the 66-order is obeyed every time, and the 67-order is used to count through the blocks. For example, if the two sets *A* and *B* are 80 blocks apart, as above, and we wish to write the numbers of set *B* then the loop can be built up as follows.

```
Set initial conditions
Set (A0, c) in X2
 0 □72 2                    read A
80 □72 2                    read B
Operate (modified by 2M)
    2 66
79 □73 2                    write B
    2 67                    count blocks
```

In this case the counter c must be set initially to the number of blocks occupied by each set (including the first block, which is perhaps only partially filled), i.e. it must be equal to the integral part of $(n + 7)/8$.

Sometimes we have an *inner loop* within an *outer loop* and we must then have two counters; the counter for the inner loop must be reset every time the outer loop is traversed. As an example, let us consider the evaluation of the product of a matrix **A** by a column-vector **y** to give a column-vector **z** = **Ay**. Suppose **A** is an $m \times n$ matrix, i.e. a set of numbers (or elements) a_{ij} which are stored in

A matrix stored in this way is said to be stored *by rows* since the elements a_{ij} in the same row of the matrix (i.e. having the same value of i) are stored in consecutive locations. If the first element (a_{00}) of the matrix is stored in location A_0 then the (i,j)-th element (a_{ij}) is stored in the location whose address† is

$$A_0 + ni + j.$$

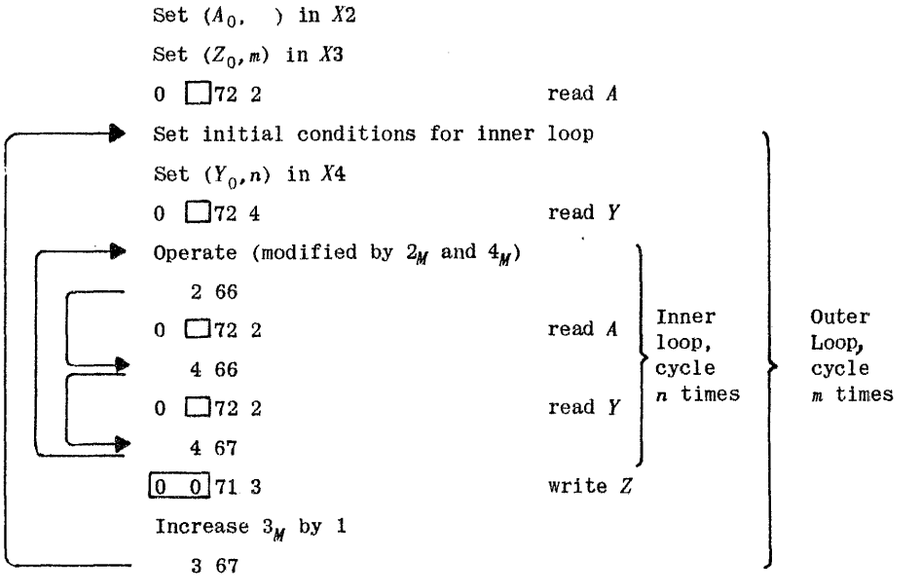
The n elements of the column vector **y** will be denoted by

$$y_0, y_1, \cdots, y_{n-1},$$

and we shall suppose they are in consecutive locations starting at Y_0. The m elements of **z**, $z_0, z_1, \ldots, z_{m-1}$ are to be placed in consecutive locations starting at Z_0. According to the rules of matrix algebra, the i-th element of **z** is to be evaluated by the formula:

$$z_i = \sum_{j=0}^{n-1} a_{ij}y_j \quad (i = 0, 1, 2, \ldots, m-1),$$
$$= a_{i0}y_0 + a_{i1}y_1 + \ldots + a_{i(n-1)}y_{n-1}.$$

This formula is similar to that for a scaler product given in Sec. 5.6 and the inner loop of the present sequence of orders will consequently be similar to the loop given earlier. We shall suppose that the addresses of the sets concerned are quite arbitrary and we shall therefore need three modifiers to work through the addresses of **A**, **y** and **z** respectively. Since there is a fair amount of work to be done in evaluating each element of **z** we shall write this element away with a single-word transfer (71-order). The sequence may be written in abbreviated form as follows:

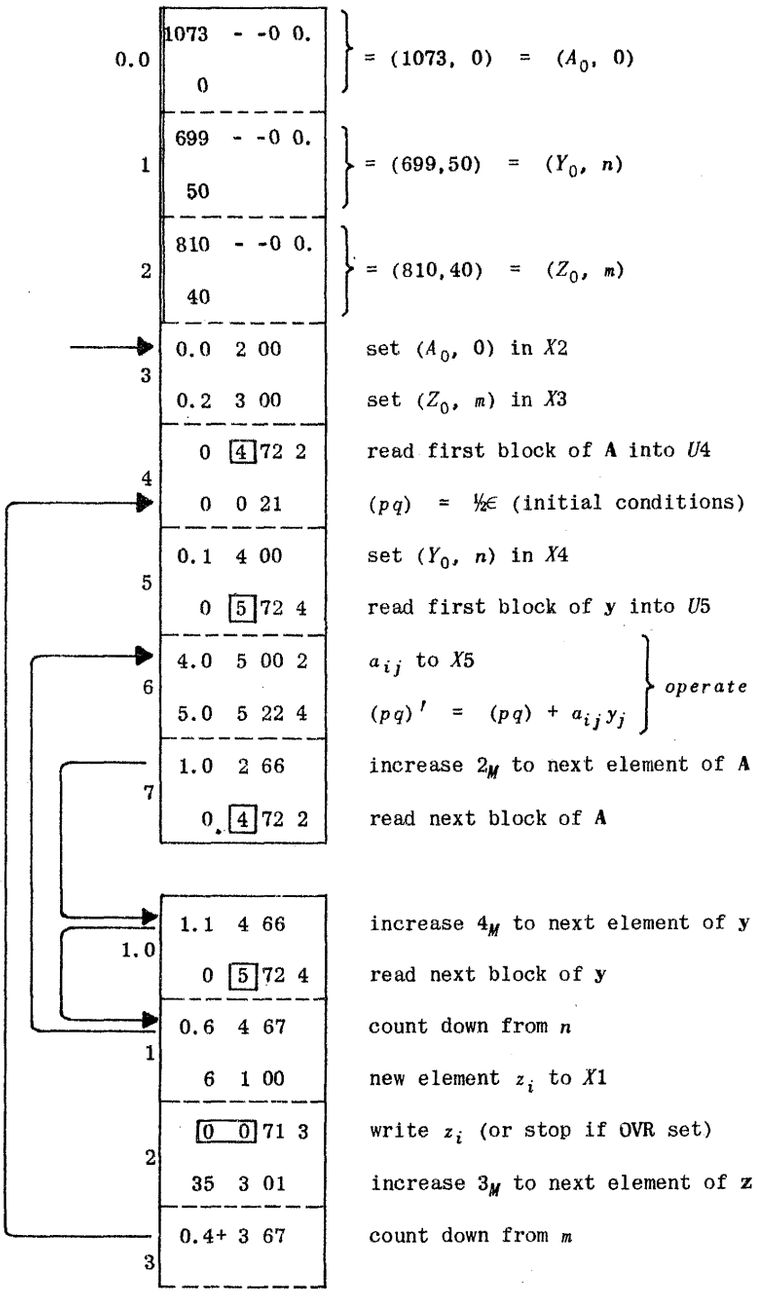


We shall now write out the sequence of orders assuming that the various constants have the following values:

A_0	=	1073	decimal address of first element of matrix **A**,
Y_0	=	699	decimal address of first element of vector **y**,
Z_0	=	810	decimal address of first element of vector **z**,
m	=	40	number of rows of **A** (or elements of **z**),
n	=	50	number of columns of **A** (or elements of **y**).

We assume that both blocks of the sequence are transferred to *U*0 and 1 in the computing store before entry at 0.3; the blocks *U*4 and 5 are used to hold elements of **A** and **y** respectively. In this sequence we have disregarded overflow; if the Stop On Overflow key is not down and an incorrect result is obtained because of overflow the computer will stop as soon as it encounters the 71-order.

† Note that i ranges over values from 0 to $m - 1$, and j from 0 to $n - 1$, so that the first row and column are numbered zero. This leads to a simpler formula for the address of the (i,j)-th element than does the conventional system of starting the numbering at 1.

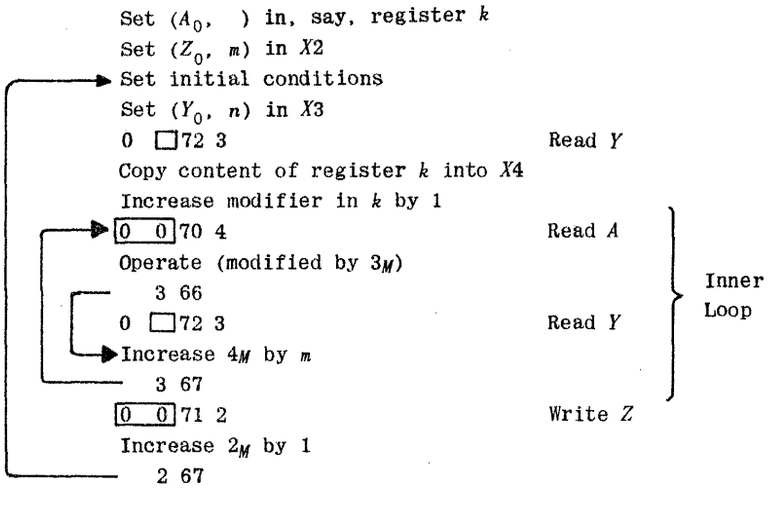


The running time for this sequence is about 18 seconds.

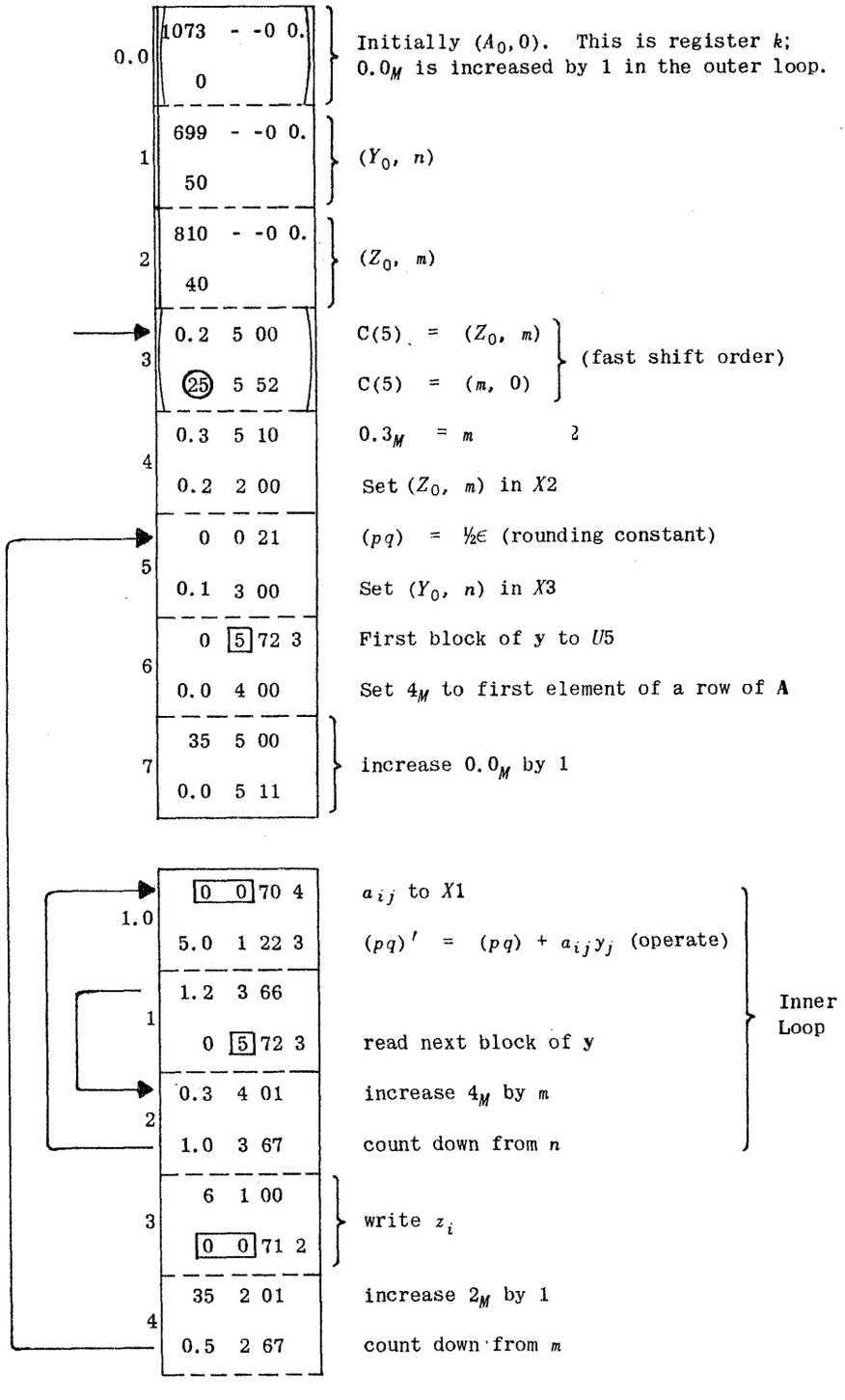
If the matrix is stored by columns instead of by rows then the address of a_{ij} is

$$A_0 + i + nj.$$

There is now no advantage to be gained by bringing in A by blocks because a 72-order will transfer elements of the same column and we need them by rows. If we use single-word transfer orders to read A and also to write the product vector z, we can use a sequence as follows:



Note that 4_M, which is used to pick out the elements of **A**, is increased by m each time the inner loop is traversed; in this way we can select the successive elements in a row. When the inner loop has been traversed n times we have evaluated one element of **z** and 4_M has been increased by mn. Before we can calculate the next element of **z** we must set 4_M to the address of the first element of the next row of **A**; we can do this either by subtracting $mn-1$ from it or, as we have done here, by storing it (in a register k) before entering the inner loop. In this example we use the same values of the constants as in the previous one. The ordinary register 0.0 is the "register k"; its content is initially $(A_0, 0)$. Register 0.3 is used to hold a useful constant when the order-pair in it is no longer needed. On the programme sheet these two words are written in brackets, the conventional way of indicating that they get changed during the course of the programme.



The running time for this sequence is about 33 seconds.

It is sometimes necessary to work backwards through a set of locations in the main store; this is to be avoided, if possible, since it is very easy to make mistakes. If we have to work backwards through one or more sets of numbers it is simplest, if computer time and storage space permit, simply to reverse the sets concerned, i.e. to copy them backwards, into a new set of locations, before performing any arithmetical operations on them. Alternatively we may use only single-word transfers.

Suppose, as an example, that a set of n numbers starting in location A_0 is to be copied backwards into a set of n different locations starting at B_0. This means that:-

a_{n-1} is to be copied from $A_0 + n-1$ into B_0

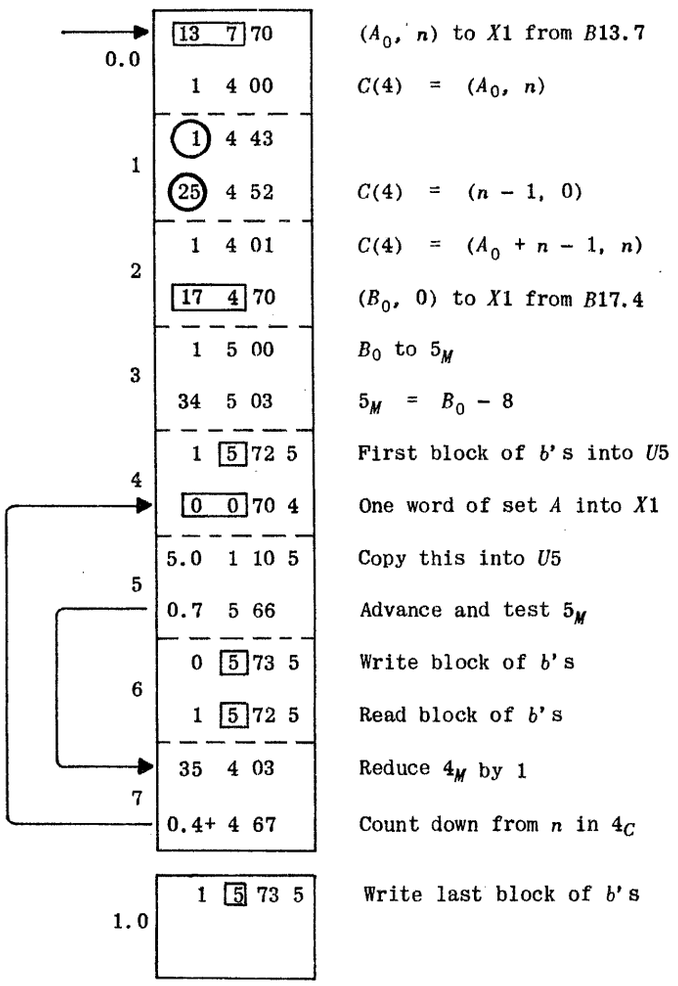
a_{n-2} " " " " " $A_0 + n-2$ " $B_0 + 1$,

.....

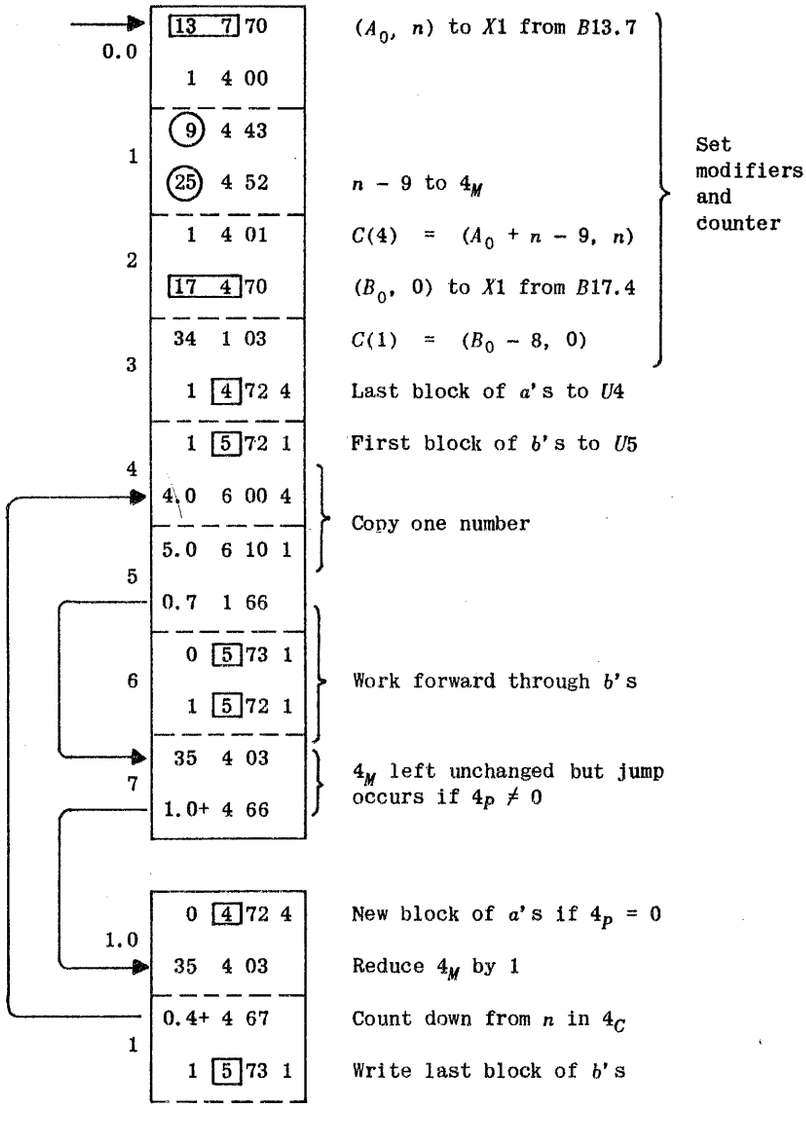
a_1 " " " " " $A_0 + 1$ " $B_0 + n-2$

a_0 " " " " " A_0 " $B_0 + n-1$

If block-transfers are used it is best to transfer the elements in the order indicated above. Suppose that the constant (A_0, n) is stored in $B13.7$ and that the constant (B_0, 0) is stored in $B17.4$. The simple method of using single-word transfers to read in the elements of set A will be illustrated first. We must construct a modifier equal to the address of the last element of A, this is $A_0 + n-1$. The programme will read; -

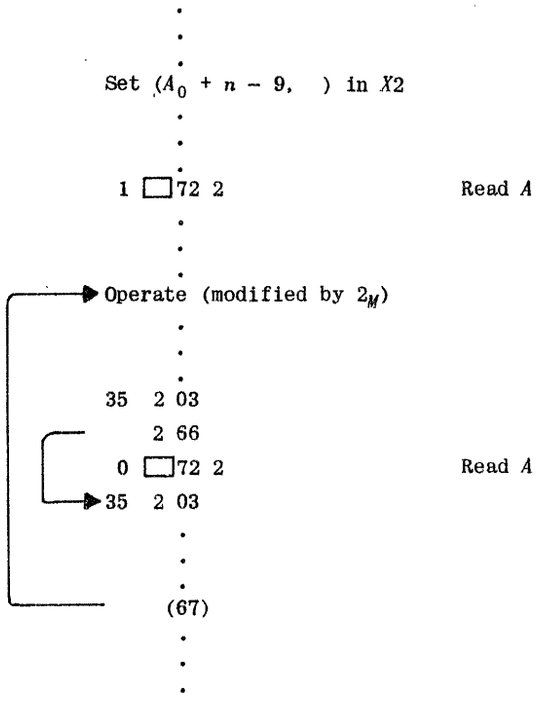


Since there is a single-word transfer order in the loop of the above programme it will be relatively slow (one number every 16 milliseconds). This may not matter. A rather faster programme may be written using block-transfers only. The following sequence will reverse the elements into their new places (it is assumed that the new locations are quite distinct from the old).



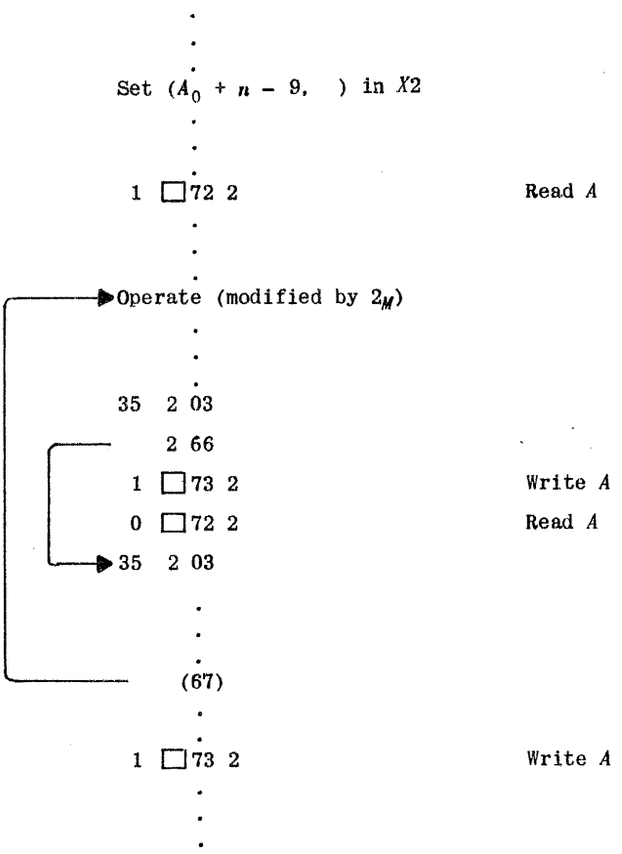
Note that the modifier 4_M has the initial value $A_0 + n - 9$; this is in fact 8 (or 1.0) less than the address of the first number to be handled, which is necessary when working backwards through a set with block-transfers just as it is when working forwards if we are writing and not simply reading. The 66-order in 0.5+ causes no jump when the position-number 4_P is zero, i.e. just before the block-number is reduced.

The administrative orders in a loop which works backwards through the n numbers of a set A (using block-transfers) can be written down in abbreviated notation as follows:



As before, the dots indicate the orders concerned with manipulating any other sets of numbers. Here the initial setting of 2_M is 8 (or 1.0) less than the address of the first number to be handled, i.e. the last number of the set. The first 72-order reads in the block containing this number. The 66-order and the 03-order preceding it can be regarded as leaving 2_M unchanged but sensing the moment when the last number of the set in the computing store (i.e. that in position 0 of the block) has been dealt with. The modifier is reduced by the 03-order after the 72-order.

If we are writing (or writing and reading) the numbers of the set the orders are as follows.



Note that in this sequence the modifier has the same starting value as in the previous sequence. The final 73-order outside the loop writes back the block containing the first number of the set.

As an example we will suppose that the n coefficients a_1, a_2, a_3, ..., a_n have been computed and stored in consecutive locations with

a_1 in location A_1,
a_2 in location $A_1 + 1$,
.....
a_n in location $A_1 + n - 1$.

We have to evaluate the $n + 1$ quantities x_0, x_1, x_2, ... x_n from the following recurrence relations:-

$$x_n = y,$$
$$x_{n-1} = a_n x_n,$$
$$x_{n-2} = a_{n-1} x_{n-1} + x_n,$$
$$x_{n-3} = a_{n-2} x_{n-2} + x_{n-1},$$
.....
$$x_0 = a_1 x_1 + x_2.$$

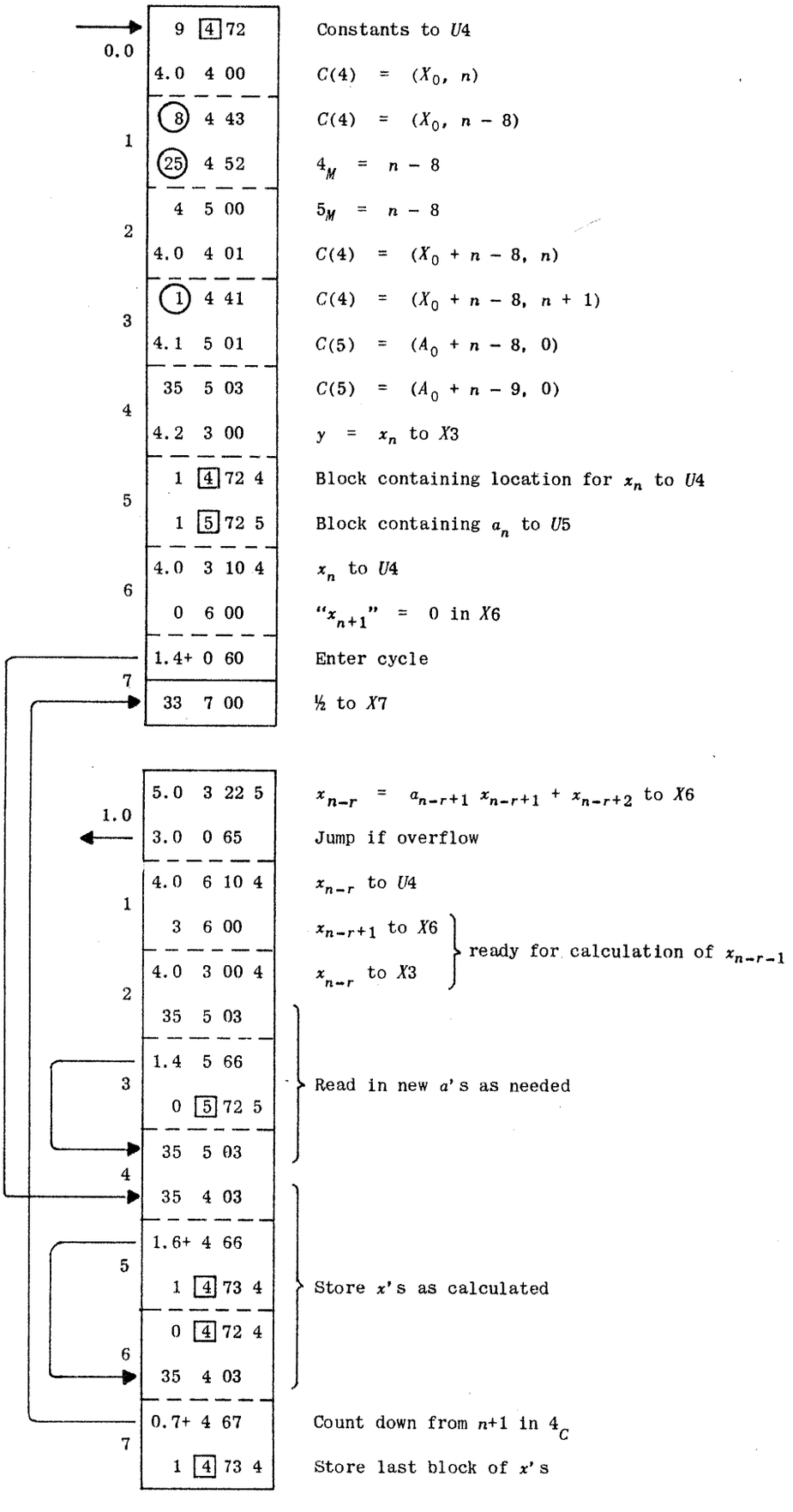
The values of the x's are to be stored in consecutive locations with x_0 in location X_0 in the Main Store, x_1 in $X_0 + 1$, ..., x_n in $X_0 + n$. The following constants are specified in $B9$:-

$B9.0$ contains (X_0, n),
$B9.1$ contains $(A_1, 0)$,
$B9.2$ contains y.

If overflow occurs a special overflow routine kept in $U3$ is to be called in by jumping to 3.0.

We need two modifiers and, since we shall have to work backwards through both sets (the a's and the x's), both modifiers must be 8 less than the addresses of the first numbers to be handled (this is so whether or not writing transfers are concerned). One modifier must be initially 8 less than the address of a_n, i.e. it must start at the value $A_1 + n - 9$. The other modifier must be 8 less than the

address of x_n, i.e. $X_0 + n - 8$. In the following sequence of orders x_{n-r+1} is in $X3$ and x_{n-r+2} is in $X6$ when the orders are encountered for evaluating x_{n-r}.



If speed is an important consideration in a loop consideration should be given to the possibility of completely or partially "unrolling" the loop. In this technique the orders required are written out one after the other instead of in the form of a loop; they are then often unmodified. Speed is gained because many of the orders concerned with counting or changing modifiers are not encountered every time the *operate* orders are obeyed. This device is chiefly useful when the *operate* orders are few in number, and when the sets concerned fit neatly into blocks.

5.8 Modification of other orders

We have so far described how the following kinds of orders are modified.

(a) *The arithmetical orders* (Functions 00 to 27), in which the *position part* of the modifier is added to the *N*-address in the order.

(b) *The block-transfer orders* (Functions 72 and 73), in which the *block part* of the modifier is added to the first address written in the order.

(c) *The single-word transfer orders* (Functions 70 and 71), in which *the whole* of the modifier is added to the address (represented by the *N*- and *X*- digits) in the order.

These orders are modified chiefly in loops or cycles of orders in which the modifier is increased or decreased (usually by 1) every time the loop is traversed. The modifier is usually best regarded as a main store address.

The other orders in the Pegasus order-code can also be modified, but usually for different reasons and hardly ever in a loop (except incidentally). With these orders it is usually best to regard the modifier simply as an integer.

The orders of groups 4, 5 and 6 are modified by having the 10 least-significant digits of the modifier added to the first address before the order is obeyed. Usually the other digits of the modifier (i.e. the 3 most-significant digits) are zero. For example, suppose $C(4) = (12, 243)$, so that $4_M = 12$; then the order

(3) 7 50 4

will have the same effect as the order

(15) 7 50

and $C(7)$ will be multiplied by 2^{15}. With the same modifier the order

(10) 5 40 4

will have the same effect as the order

(22) 5 40

and the integer 22 will be placed in *X*5.

As with the transfer orders (Sections 5.3 and 5.4), the *N*-address is extended inside the order-register by 3 binary digits at the instant the order is obeyed. The effective value of *N* may thus go up to 1023. Any carries beyond these 10 binary digits are disregarded.

As an example, suppose the number *y* in ordinary register 5.0 is to be shifted down arithmetically by *h* places. The quantity *h* is equal to an integer stored in 5.1, supposed non-negative. The following orders will do this.

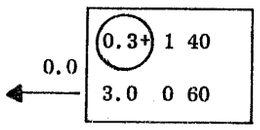
→ 0.0	5.0 6 00	*y* to *X*6
	5.1 7 00	*h* to *X*7
1	(25) 7 52	*h* to 7_M
	(0) 6 51 7	shift *y* down *h* places
2	5.0 6 10	result to 5.0

An important application of modified group 4 orders is to copy a number from the modifier of one accumulator into the counter of another, assuming that the modifier does not occupy more than 10 binary digits. On the whole it is rare to use modified orders of groups 4 and 5, and many of the applications can be considered as tricks.

Modified jump orders, although only occasionally needed, are important; when using them it is usually necessary to know the way in which the *N*-address of an order is represented in the computer (see Sec. 3.12). An important special case of the use of a modified jump is provided by what are known as *computing store links*.

Suppose we have a small group of orders or a subroutine which starts in *U*3.0. We wish to jump to 3.0 to enter the subroutine and we must arrange that the subroutine returns control to the appropriate place in the computing store when it has finished. This is normally done by setting a link in *X*1 but this is inconvenient if we wish to call in the subroutine frequently during the course of a short calculation. As we mentioned earlier, most subroutines use as working space some of the ordinary registers originally occupied by their own orders and constants; they do this, of course, only after the original contents of the registers are no longer required. This is permissible in a computer with a two-level store, such as Pegasus, since a fresh copy of the subroutine will normally be brought into the computing store every time it is needed. In some programmes this may waste some time because of the need to wait for the drum on most block-transfer orders. In order to avoid this it is possible to use *self-preserving* routines which may be entered repeatedly after they have been transferred only once. Such a subroutine must be so written that its orders are not spoilt by its own operation; for example by using only accumulators as working space, or by resetting itself (or perhaps only its first block) in the computing store before exit. It is often useful to design such a subroutine so that its last order is a modified jump order. This is most easily described by an example.

Suppose the subroutine has already been transferred to *U*3 and *U*4 of the computing store and it is to be entered at 3.0. Suppose further it is desired to call it in from 0.0 in a master-programme and that return to the master-programme is to be to 0.3+. The cue is written in the master-programme as follows.

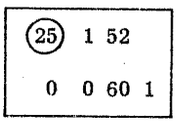


set link for return to 0.3+

enter subroutine

The first order of this cue is written in a rather unconventional way; when it is obeyed its *N*-digits are placed in *X*1. The actual digits, however, need not be known (they are specified in Sec. 3.12). The second order of the cue is simply a jump to the subroutine. Note that these two orders need not form an order-pair.

At the end of the subroutine there may be the following two orders:-

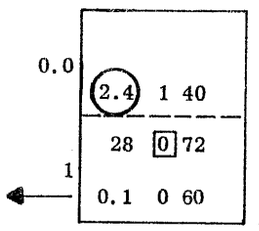


address to 1_M

return to master-programme

The first order here simply shifts up into the modifier position in *X*1 the digits originally put in by the first order of the cue, i.e. the digits of "0.3+". The second order gets these digits added to its *N*-address since it is modified by 1_M; the jump consequently has an effective address of 0.3+.

The main advantage of this technique is its ease of use, we need only two orders to set the link and call in the subroutine. Computing store links are not restricted to self-preserving subroutines; for example we may be able to enter a subroutine by a group of orders similar to the following.

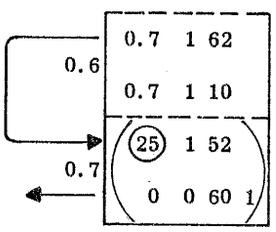


set link for return to 2.4

normal cue

Note that the subroutine must be specially written to accept such links; this would be explicitly stated in the specification of the subroutine.

Sometimes a subroutine may be so written that it can use either an ordinary order-pair link or a computing store link of the kind just described. These can easily be distinguished if the order-pair link is a go order-pair, as it nearly always would be, since it will then appear negative because of the 1 in the stop/go digit (see Sec. 3.9). A computing store link will appear non-negative since it was set by a 40-order. The terminating orders in the subroutine may therefore be as follows.



jump if computing store link

plant and obey order-pair link

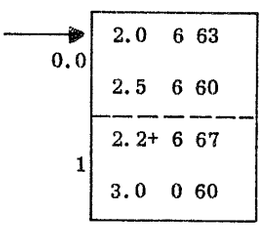
address to 1_M

exit

} may be overwritten by order-pair link

Note that this sequence is not self-preserving, and that it spoils *C*(1) if it is a computing store link. There are ways of avoiding these disadvantages should it be thought advisable.

The *multi-way switch* is another useful programming device which is made possible by modified jump orders. We have already seen how the sequence of orders obeyed by the computer can be interrupted by a conditional jump order so that one of two alternative sequences can be entered. If there are more than two alternatives we can often select the appropriate one by using two or more conditional jumps. For example, suppose that an integer a has just been calculated and placed in *X*6; it may have the values −1, 0, 1 or 2 and it has been arranged that the subsequent course of the calculation depends on the value of a (in fact a may have been artifically introduced specially to discriminate between the various possibilities). A possible method of selecting the appropriate sequence is to use the following orders.



jump to 2.0 if $a = -1$

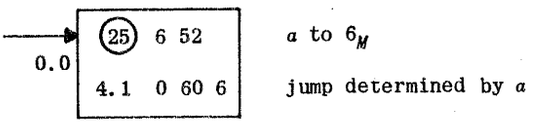
jump to 2.5 if $a = 0$

jump to 2.2+ if $a = 2$

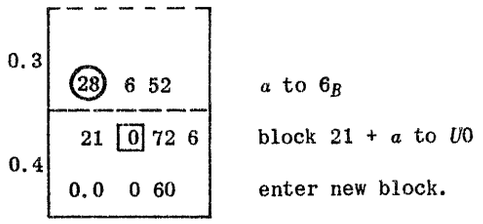
jump to 3.0 if $a = 1$

In a sequence of this kind the addresses in the jumps can be quite arbitrary. If, as often happens, the several sequences are too long be held simultaneously in the computing store, then the jumps can lead to cues for calling them in.

By using a modified jump we can get a similar effect, for example:



Here the jump will go to 4.0 if $a = -1$, to 4.1 if $a = 0$, and so on; and we can write the cues we need in these registers. An array of cues like this is often referred to as a *cue-directory*. Another possibility is to store the various sequences to be called in in consecutive blocks in the main store and to enter them with a modified cue or block-transfer order, thus for example:



Here the quantity a is shifted up into the *block* part of the modifier in $X6$ and used to modify a 72-order, which reads $B20$ into $U0$ if $a = -1$, or $B21$ if $a = 0$, $B22$ if $a = 1$, and so on.

We have now to describe how the remaining orders of group 7 are modified. It is seldom that any of these need to be modified in a basic Pegasus installation. The following summarizes the effects.

(a) the orders 70, 71, 74, 75 are modified by having the whole of the modifier added to the N- and X-digits written in the order,

(b) the orders 72, 73, 76, 77 are modified by having the block part of the modifier added to the N-digits written in the order.

Apart from the main store transfers, only the 74-order can usefully be modified in a basic installation and the 75-order is unassigned. The 76-order is often modified when magnetic tape or cards are fitted; this is described in Chapter 10. The addresses in a 77-order do not affect the operation of the order.

The way in which the various orders are modified can be conveniently summarized in the way shown in Fig. 5.2. Here the top line represents the order being modified; the lines below represent the modifier, of which the shaded part is added to those digits of the order which lie above them in the

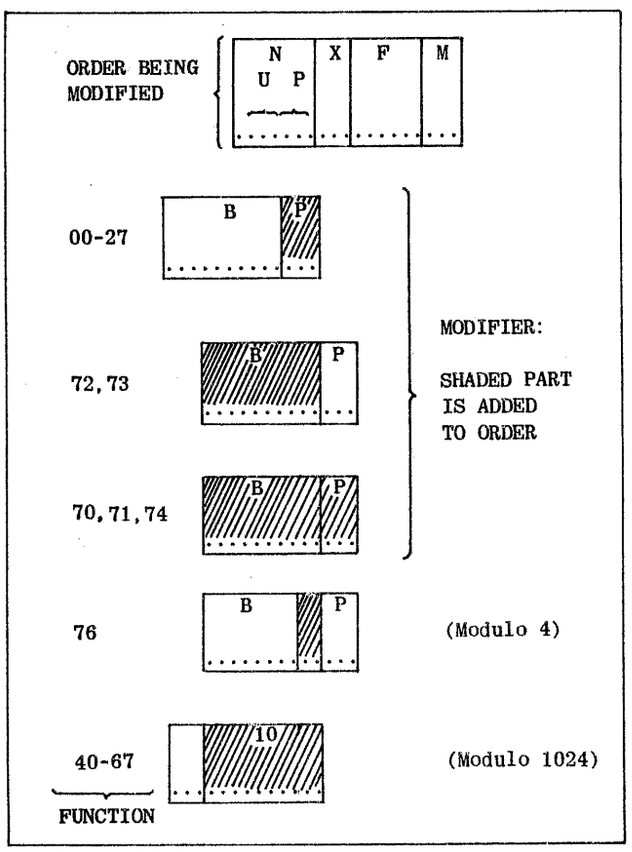


Fig. 5.2 *How various orders are modified.*

figure. Note that the order is extended on the left by 3 binary digits at the moment it is modified and just before it is obeyed. Any carries which occur beyond this will be disregarded, i.e. only the 10 least-significant bits of the modified *N*-address are relevant. In the jump orders and the arithmetical orders, i.e. in the orders of groups 0, 1, 2 and 6, it is only the 7 least-significant digits which are relevant.

The function part of the order is never affected by modification, and the *X*-part is modified only in the single-word transfer orders (70 and 71) and the external-conditioning order (74). If unconventional methods of modifying are ever needed (for example to change the function of an order) this can always be done by adopting a system often used in computers not equipped for modification; this is to change the order in its ordinary register by adding something to it, for example by using a 11- or 13-order. This technique is very seldom required and would normally rank as a trick.

We sometimes have to refer to a list or table of numbers or other data held in the store and extract a particular entry from it. This is usually referred to as a *table look-up* operation. Normally the table will be held in the main store and often each entry will occupy one word; if this is so we can extract any particular entry, say the *n*-th, by putting *n* into the modifier position of an accumulator and then obeying a modified single-word read order. For example, if *n* is held as an integer in 5.0 and the table is so stored that the entry corresponding to $n = 0$ is in *B*94.0, then the following orders will put into *X*1 the entry corresponding to the specified value of *n*.

```
5.0  1 00      n to 1_C
(25) 1 52      n to 1_M
[94  0]70 1    n-th entry to X1.
```

If each entry in the table occupies more than one word it may be advantageous to use a block-transfer. For example, suppose each entry occupies two words and the table starts in *B*94.0, then the following orders will place in *X*4 and 5 the two words corresponding to *C*(5.0).

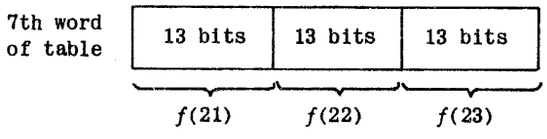
```
5.0  5 00      n to 5_C
(26) 5 52      2n to 5_M
 94 [4]72 5    block containing entry to U4
4.0  4 00 5    first word to X4
4.1  5 00 5    second word to X5.
```

Looking up such a table is particularly simple if each entry occupies a whole block.

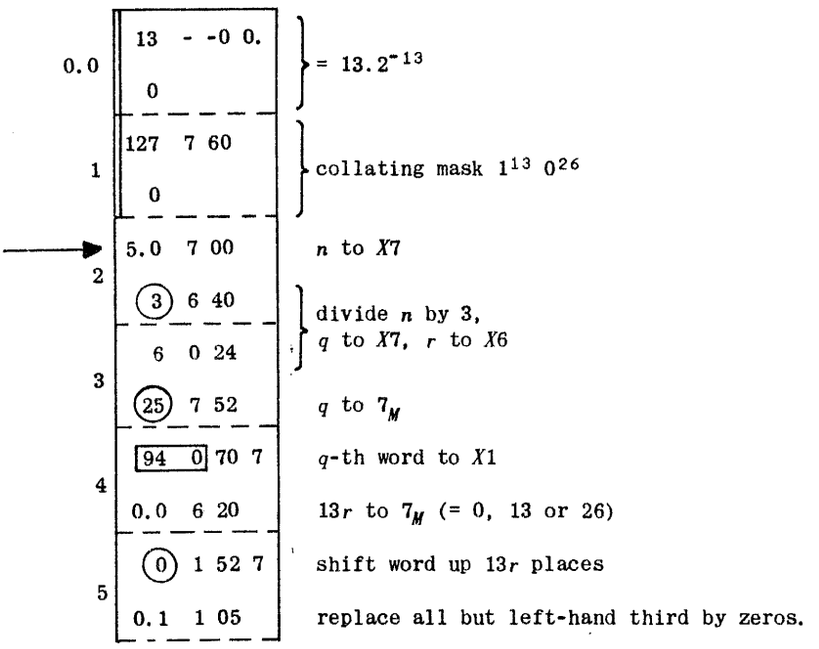
Often the entries in the table will be small integers or other items which can be specified adequately by using only a few binary digits. We can then pack several entries into a word and make the table more compact. As an example, suppose there are three entries in each word, each occupying 13 bits. Suppose we denote the *n*-th entry by $f(n)$ since it corresponds to the mathematical idea of a function of *n*. It is customary and convenient to number the entries in the table from zero, so that the first entry is $f(0)$ and not $f(1)$; to avoid circumlocution we shall talk of the *n*-th entry when we mean $f(n)$. To find out the number of the word in which $f(n)$ is stored we divide *n* by 3; suppose the quotient and remainder are *q* and *r*, respectively, so that

$$n = 3q + r.$$

The word we want is the *q*-th (starting from 0) and *r* (which will be 0, 1 or 2) indicates which part of the word contains $f(n)$. For example, the 7th word in the table (or rather word no. 7, starting from no. 0) will contain the values of $f(21)$, $f(22)$ and $f(23)$, packed in the following way.



If *n* is the non-negative integer in 5.0 then the following sequence will place $f(n)$ in the left-hand third of *X*1.



If the result is wanted in the right-hand third of the word instead of the left-hand third, we can omit the collating mask and replace the last order of the sequence by a logical shift down of 26 places; this is, however, relatively slow. A faster alternative is obtained by putting -13.2^{-13} in 0.0 so that the order in 0.4+ puts $-13r$ into 7_M (actually modulo 8192); the modified shift order should then be written as follows.

26 1 53 7

This order will then shift the word down $26-13r$ places. If a still faster process is wanted we can replace the division by 3 by a multiplication by 1/3, or by $2^{-13}/3$ (*rounded up* in either case).

A table look-up process is most useful when the entries in the table are not related in any simple or systematic way to the values of n. In fact the entries can be quite arbitrary. Sometimes it is possible to calculate from the value of n just what the corresponding entry is. If the time and complexity of this calculation are not too great then it may well be better to perform it instead of looking up a table. The programmer should always remember that looking up a table is often time-consuming (especially if interpolation is required) and that the table may occupy a large amount of storage.

Code conversion on input or output is an important application of the table look-up technique; a useful way of doing it is described in Sec. 6.3.

There is another kind of table look-up in which the entry to be extracted is determined by the entry itself and not (as in the processes described above) by where it is stored. This usually involves searching, and may therefore be referred to as a *table searching* process. In a typical application the entries may be word-pairs, and we require the word-pair whose first word (the *keyword*) is just greater than a specified number, the word-pairs being stored in order of magnitude of the keyword. Alternatively the entries may be blocks. In such a table not all possible keywords are present. One way of finding the appropriate entry is to determine first in which half of the table it lies, this can be done by examining the keyword of the central entry. We next look at the keyword of the central entry of this half of the table to find out the appropriate quarter of the table, and so on. In other words we proceed by repeated bisection of the table.† Special care has to be taken if the number of entries is not an exact power of 2. There are a number of ways of making the process faster. For example, we can store separately, in special blocks, copies of some of the keywords, grouped together so that the first few keywords we have to examine can all be brought into the computing store by a single block-transfer. In certain tables the distribution of the keywords may be such that it is possible to calculate approximately where a given keyword is to be found and so to narrow the search.

5.9 A complete programme using modification

We shall now describe a complete programme illustrating the application of modified orders and the use of subroutines. It is not intended to be suitable for practical use.

The programme is designed to read in a paper tape on which are punched account numbers, quantities and prices. Fig. 5.3 is an example of the print-out of an acceptable input data tape (see Sections 4.3 and 4.4). Here the first column contains the account numbers, the second contains the quantities, and the third the prices in sterling. The account number may be from 0 to 99; the quantities and prices may not be negative. Any number of lines may appear; after the last line the "account number" minus 1 is punched (as -1) to indicate the end of the tape. As the tape is read in by the programme a value is found for each line by multiplying the quantity by the price; this value is added into a total corresponding to the account number. When the end of the tape is reached there will therefore be up to 100 total values, one for each account number that has appeared on the tape. These totals are then printed out and the computer stops.

† This is sometimes called a *Weierstrass* bisection process, after the famous 19th Century mathematician who first applied it to mathematical analysis. It is also known as a *logarithmic search* since the time needed goes up as the logarithm of the number of entries in the table.

```
12        6      1.15.10
49      144     36.13. 0
90       30     10.10. 0
21       53      6. 7. 6
12       12      2.14. 0
 5       10      3. 6.11
90       10     51. 0. 0
12        3      4. 0. 0
21     2000      1. 1. 3
12       24     16.10. 6
-1
```

Fig. 5.3 *Print-out of a data tape.*

The process and the programme can best be described with the aid of a flow diagram, see Fig. 5.4†. At the start 13 blocks of the main store are cleared; these contain 104 storage locations, of which the first 100 are used to receive the totals as they are built up; the other 4 locations are not used. Next the *input loop* is entered: this starts at link-point 1. A subroutine (F) is used to read integers from the data tape; it is called in twice during the input loop, the first time to read in the account number and the second time to read in the quantity. These numbers are tested as soon as they have been read; if the account number is negative then the end of the tape has been reached and the computer leaves the input loop and goes to link-point 9 as described below. If the account number is between 0 and 99 and the quantity is not negative the input loop continues by calling in another subroutine (G) to read the price in from the tape. This price is multiplied by the quantity to form the value, which is then added into the total corresponding to the account number. The computer stops, however, if either the value or the new total exceeds the capacity of a register (about £1100 million since sums of money are stored as integral multiples of a penny). Control is then returned to link-point 1 to repeat the input loop.

Link-point 9 is reached when all the data have been read in; here two line feeds are printed (to separate subsequent printing from other printing on the page), after which a counter (c) is set equal to 100 and a modifier (n) is set equal to zero; the *output loop* is then entered at link-point 2. Here the totals are examined successively to see if they are zero. If any non-zero total is found a jump occurs to link-point 3, where subroutine A is called in to print a carriage return, a line feed and the account number (n); subroutine D is then used to print a space followed by the value of the total. When all 100 totals have been examined the computer stops (at no. 8).

The programme tape contains only the master-programmes. The four subroutines used (A, D, F and G) are assumed to be already in the main store; they are included in the *Lesser Library* and their specifications are given in Appendix 2. This small collection of subroutines can be put into the main store (in fixed locations) at any time by running in a certain tape. In order to read in the programme tape it is placed in the main tape-reader, and the Initial Orders are called in by operating the Start key (see Sec. 4.3). The Initial Orders then read in the tape; which starts, as usual, with a D-directive to cause printing of the date and serial number as the tape goes in. After this is an N and the name of the programme - ACCUMULATE TOTALS. Then come the four blocks of the master-programme, which are stored by the Initial Orders in $B2$ to $B5$ in the main store. At the end of the tape is punched the directive E 2.0, and a few "Erases" (used to mark the end of the tape). As soon as the Initial Orders read E 2.0 there is a 77-stop; when this occurs the programme tape is taken out of the tape-reader and the start of the data tape is placed in the tape-reader. The Run key is then operated and the first four blocks of programme, $B2$ to $B5$, are copied by the Initial Orders into the computing store (this is actually the whole of the master-programme), and a jump then occurs to the a-order in 0.0. In this way the computer starts to obey the master-programme.

The master-programme is given in full in Fig. 5.5; in order to facilitate study the numbers of the link-points and stops in the flow-diagram have been inserted on the programme sheet against the appropriate orders. At the moment when the computer starts to obey it the whole of the master-programme is in the computing store and the data tape is in the main tape-reader. The first four orders are those which clear 13 consecutive blocks in the main store, starting with $B110$, to receive the totals. This is done by copying $U6$, which contains eight zeros, into each block in turn (see Sec. 3.10). Note that 2_B is initially zero and is increased by 1 each time the loop is traversed.

When the 13 blocks have been cleared the input loop is entered (link-point 1, b-order in 0.2); this loop consists of the orders from 0.2+ to 2.1 inclusive. The first few orders are those which call in subroutine F to read in the first number on the data tape; this subroutine uses $U0$ and $U1$ and leaves the account number (n) in $X6$ before obeying the link (1), which restores $B2$ in $U0$ and jumps to 0.5.

---

† This flow-diagram is written according to one of the many sets of conventional rules which have been devised. The operations are all written in *operation boxes*, which have one entry arrow and one exit arrow. The *tests* are all written in ovals, which have one entry and two exits. *Stops* and *Starts* are indicated by large circles. The *link-points*, which are here numbered for convenience with the stops and starts, are the small circles; all the link-points bearing the same number can be thought of as joined together. A link-point can have any number of entries but may have one exit only; the numbering is useful for reference purposes. The arrows may lead only from one box (or link-point etc.) to another - they may not join up.

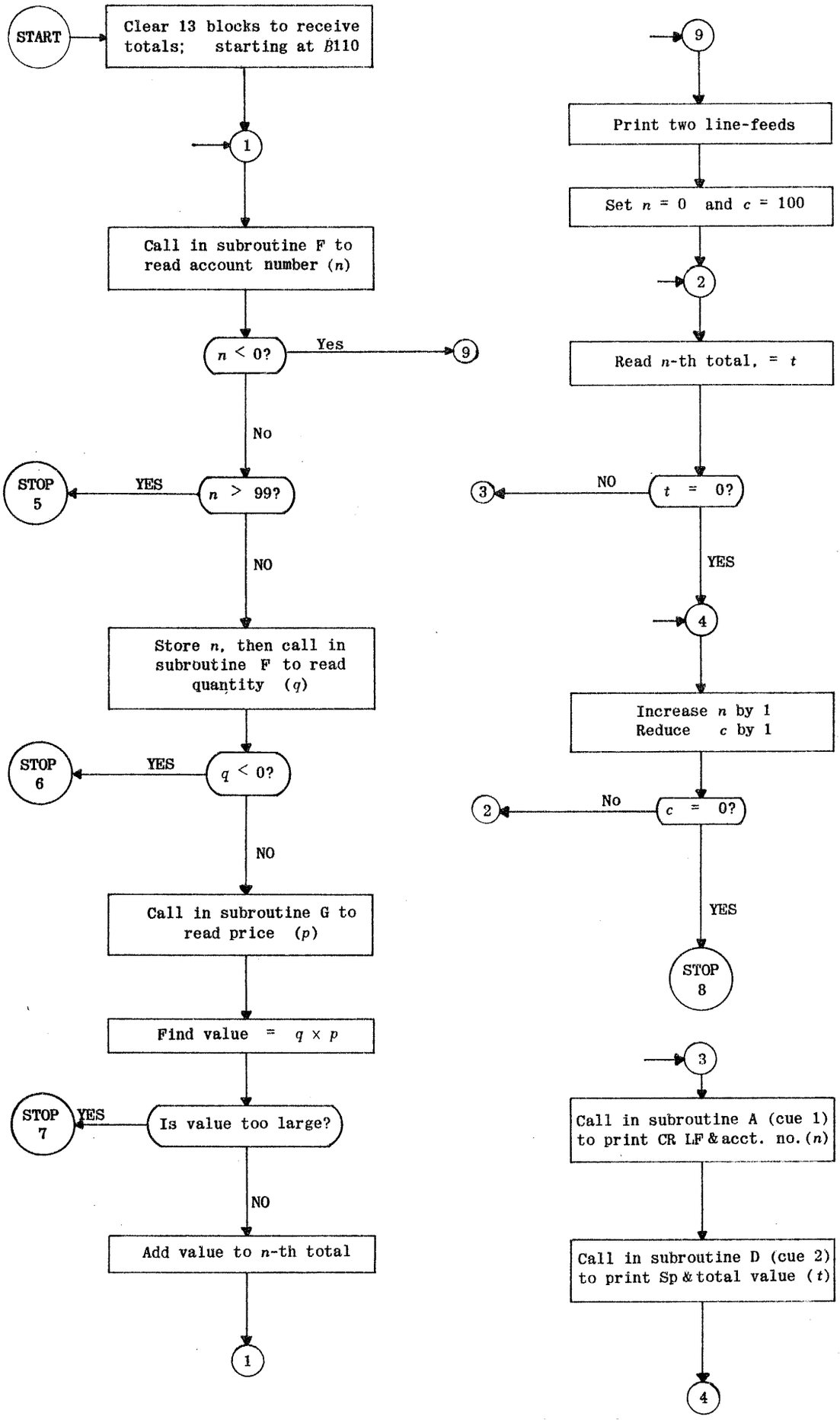


Fig.5.4 *Flow-diagram of the programme "Accumulate Totals"*

D

N

ACCUMULATE TOTALS

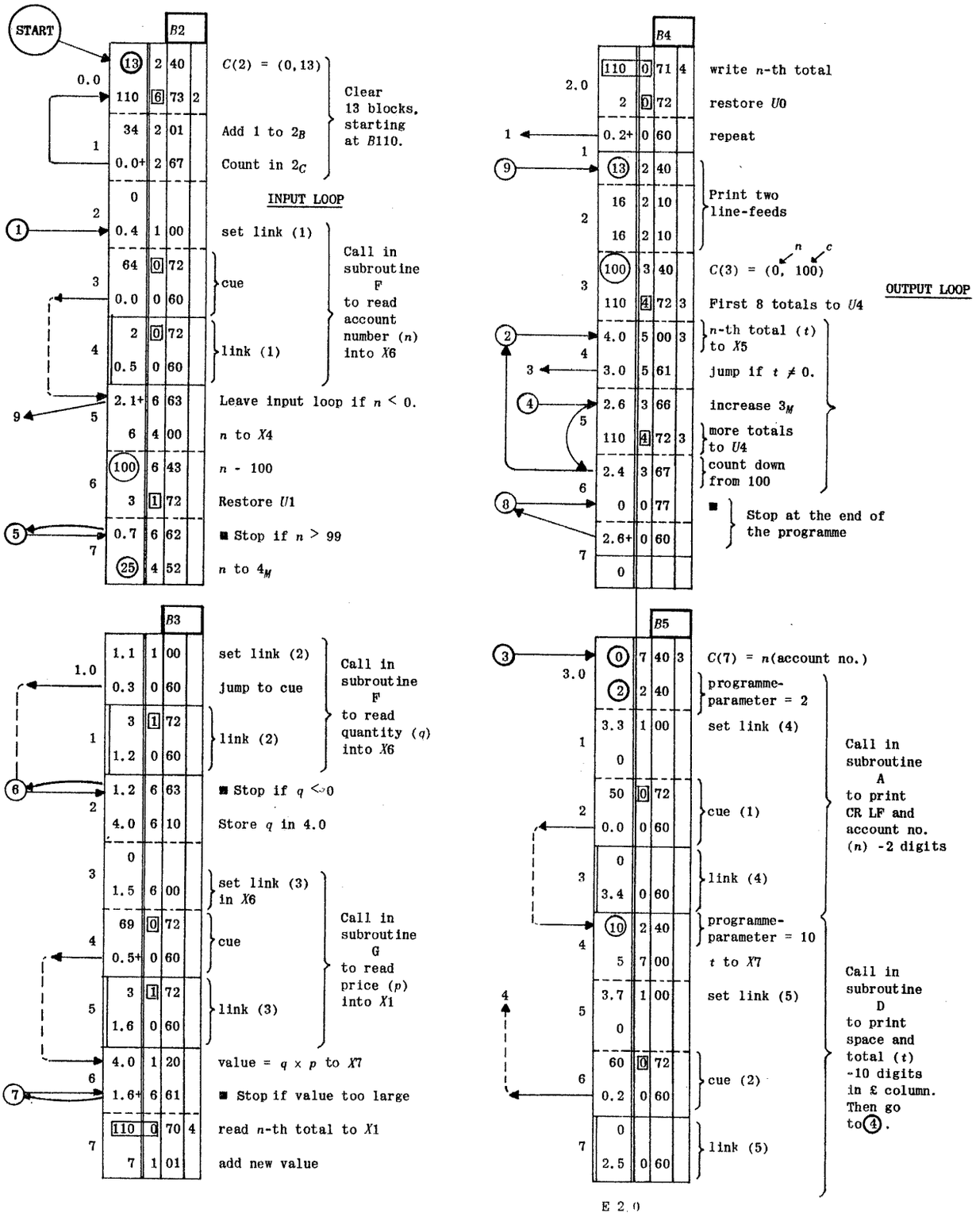


Fig. 5.5 *The master-programme of "Accumulate Totals"*

If the account number n is negative we have reached the end of the tape and there is a jump to 2.1+; but otherwise n is tested, to make sure it does not exceed 99, and is placed in 4_M by the fast logical shift for later use.

The first few orders in *U*1 call in subroutine F again to read in the next number on the tape (i.e. the quantity q). Note that the cue is not written out again; we have simply jumped to 0.3 to save a few orders. This time the link in *X*1 is the order-pair in 1.1, which restores *U*1 and jumps to 1.2 when the quantity has been placed in *X*6; if q is not negative it is temporarily stored in 4.0. The next few orders call in another subroutine (G) to read the price from the data tape. This subroutine is a little unusual in taking its link from *X*6 and leaving its result in *X*1.† When it has been called in it uses *U*0 and *U*1 and reads the price, converting it to an integral number of pence in *X*1. The link restores *U*1 and jumps to 1.6 where the value is found by a multiplication. If this value is a single-length number *X*6 will be clear and the n-th total gets the value added to it. This is done by a kind of table look-up, a modified 70-order extracts the n-th total from the location n ahead of *B*110.0 and puts it into *X*1, the value is added to this and the result put back by a modified 71-order. If the new total overflows the computer will stop at this point (see Sec. 3.10). After this *U*0 is restored (*U*1 was restored by the last link) and we jump to 0.2+ to repeat the input loop.

Eventually the "account number" minus 1 is read by subroutine F and the a-order in 0.5 then causes a jump to 2.1+ (i.e. link-point 9) where two line feeds are printed. The programme now has to print all the non-zero totals. First we put (0,100) into *X*3; at a later stage 3_M is the account number n and 3_C is the counter c which is used to stop the process when all the 100 totals have been dealt with. The next order (the b-order in 2.3) is

110 [4] 72 3

In fact when this order is obeyed 3_M is zero so the order need not have been written as a modified one at all. In other programmes the modification in this situation is necessary and it is therefore customary to write a modified order, even though the modifier may be zero, as it is in this programme. The effect of the order is to copy into *U*4 the eight totals corresponding to account numbers 0 to 7. The output loop is then entered at link-point 2, i.e. the a-order in 2.4.

The remaining orders in *U*2 are a special case of the simple standard loop given at the beginning of Section 5.6. The *operate* orders are

4.0 5 00 3

3.0 5 61

these simply pick out the n-th total and jump if any printing is required. This loop will not therefore be described further here. If the total is not zero we go to 3.0 (link-point 3); the orders in *U*3 are concerned with printing the account number n and the corresponding total t, using subroutines A and D in the Lesser Library. To use subroutine A we must put n into *X*7, place in *X*2 the number of digits (in this case 2), set a link and obey the first of the two cues given in the specification (to precede the printing by carriage return and line feed). The first order in *U*3 is

(0) 7 40 3

```
1/2/61----7
LESSER LIBRARY

1/2/61----8

ACCUMULATE TOTALS

 2       271308.15. 9
 3    465138 84.19. 3
12       840859.16. 2
14        34119 4. 9. 6
19  1081500023. 8. 8
21       775703.10.10
23          171.13. 4
36  538451476.15. 2
47      2556865. 3. 2
50        75939.19. 8
51       843870. 1. 6
66  594461666.13. 4
70        80710. 9. 7
84   1696979 1.13. 4
85     7070554. 3. 4
95       763781. 6. 4
```

Fig. 5.6 *Accumulate Totals - Computer Output.*

† This has been done in case it is required to put the result into the main store with a 71-order.

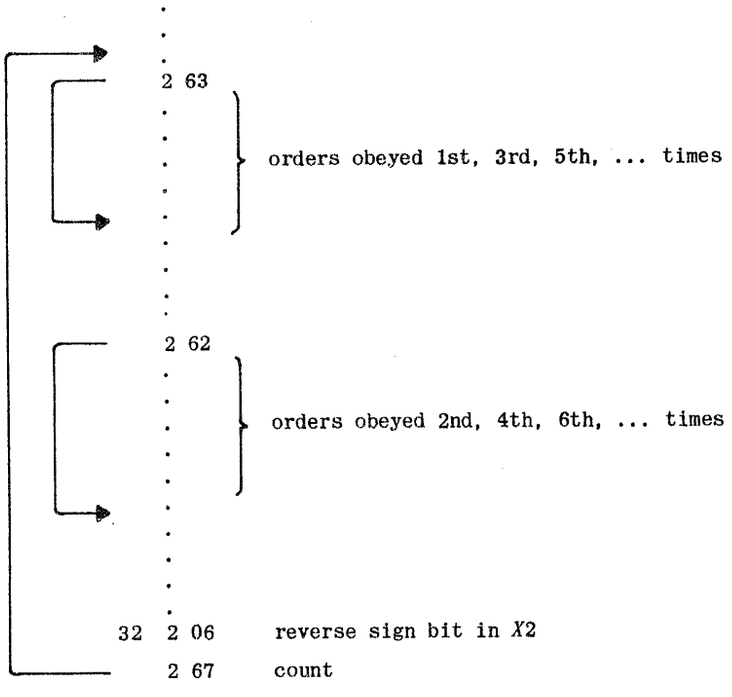
and has the effect of copying the account number from 3_M into the right-hand end of *X*7. When subroutine A has printed the value of n we call in subroutine D to print the total t (which is in *X*5) since we want it printed as a sterling sum of money. Here we may have as many as 10 digits in the £ column so we put 10 in *X*2 as required by subroutine D (see the specification in Appendix 2); we use the second cue since we wish the total t to be printed on the same line as the account number. When this subroutine has finished it obeys the link written in 3.7, which causes a jump back to 2.5 in the main output loop. When all the non-zero totals have been printed the 77-stop in 0.6+ is obeyed.

Figure 5.6 shows the computer output when the programme was run. The first two lines were printed by the Initial Orders when the Lesser Library tape was put in; the first line being the date and serial number printed because this tape has a D-directive on it. The third line is the result of the D-directive at the head of the "Accumulate Totals" tape (note that the serial number is one more than that printed on the first line). Next follows the name of the programme. The rest of the printing was produced by the programme itself; one line of printing for each non-zero total (the data tape was not that shown in Fig. 5.3). The first column contains the account numbers, and opposite each of these is printed the corresponding total.

5.10 Programming tricks

We shall now describe a few tricks, which may be defined as programming devices arising accidentally from the design of the computer and for which no provision was deliberately made. Every computer has its repertoire of tricks, though Pegasus has perhaps fewer than most†, and programmers generally find it satisfying to discover a trick or apply one to some process. On the whole tricks are to be avoided when writing a programme since one can easily spend much time in devising them. Their use is legitimate if the space occupied by a programme is important (for example, a certain section may have to be fitted into a block) or if a great deal of time can be saved. If a trick is used in a programme then it should be fully explained on the programme sheet so that someone unfamiliar with it (including, probably, the author after the lapse of a few months) can follow what is intended; such an explanation is valuable if, as often happens, the programme has to be altered during development or at a later date. Some tricks may make it exceedingly difficult to alter the programme and should not in general be used.

A useful device, which hardly ranks as a trick, is that known as a ***binary switch.*** This can be described as any technique for ensuring that certain effects occur on alternate cycles of a loop, or on alternate times that a particular sequence is obeyed. The simplest method is probably to use the sign-digit of an accumulator, this digit being reversed every time the loop is traversed. For example, suppose $C(2) \geq 0$ on entry:-



The rest of the accumulator can hold, for example, a modifier and a counter. There are many other ways of doing this; if an accumulator is available, for example, one can do it by counting or by using a modifier which is given two values alternately. Occasionally we may require that a certain number takes two different values on alternate cycles; here we can often use an 04-order or an 06-order in the loop to change the number from one value to another (the loop written above is only a special case of this). For example, suppose $C(2)$ is to be alternately 19 and 43; before entering the loop we set the value appropriate to the first cycle, and in the loop we have the order

(62) 2 44

which changes 19 into 43 or vice versa.

† This is not accidental. The computer was deliberately designed in such a way as to minimise the need for tricks. Most of the orders carry out the kind of operations one wants, and most of the elementary processes can be done with one or two orders used in a quite straightforward way.

Before passing on to the real tricks we shall draw attention to certain useful effects obtainable in the ordinary way. Consider, for example, the order

(1) 6 51

This order will clear *X*6 only if its original content was 0 or −1. The order

(1) 6 53

will clear *X*6 only if *C*(6) = 0 or +1 originally. We can sometimes use these ideas to test for two numbers simultaneously. For example, if *C*(6) is a small integer, the two orders

(1) 6 51

3.0 6 67

will cause a jump unless *C*(6) was +1 or +2.

The overflow-indicator can sometimes be used to store one bit of information. It can be set by the order

32 0 02

or cleared (and tested) by a 64- or 65- order. The order

0 0 23

clears OVR and the sign bit in *X*7. The order

N 0 02

where *N* is any register, will set OVR only if the content of the register is −1.0; if *N* = 15 we can in this way sense whether or not the setting of the handswitches is with *H*0 = 1 and *H*1 to *H*19 clear.

We can sometimes avoid the necessity for counting in processing a list of numbers by arranging that the last number is distinguishable in some way. For example, if all the numbers are positive we can add an extra negative or zero number at the end; or if they are all small we can put in an extra one which is very large. A useful number for this is the fraction −1.0 which can easily be detected by a pair of orders such as the following, for example:

5.0 0 02 3

2.0 0 65

Suppose we have two modifiers and counters in 5.0 and 5.1 then, provided the sign-bits are the same, the following sequence of orders will cause a jump only if the modifiers are equal.

5.0 6 00

5.1 6 06

(14) 6 50

3.0 0 64

We assume here that the counters are less than 2^{24}, and that OVR is clear on entry to the sequence.

It is sometimes necessary to mark or record the fact that a certain sequence of orders has been obeyed or a certain fact has occurred. This may happen if, for example, two branches of the programme join up and later the programme has to separate into two again. There are many ways of marking; a link may be set which is later obeyed (as in subroutines), or the sign-bit or the modifier or the counter in some accumulator may be given a certain value and later tested; perhaps OVR can be set by one branch and not by the other. Note that three markers can fairly easily be held in a single accumulator; the sign-bit can be tested by a 62- or 63-order and is the most convenient one to use. The order

2.0 6 66

will cause a jump unless 6_P = 7. The order

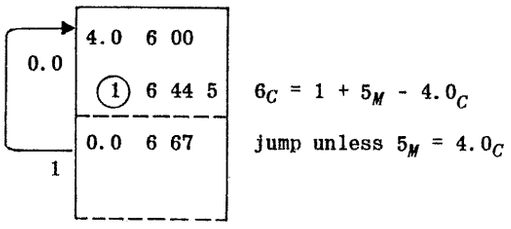
2.0 6 67

will cause a jump unless 6_C = 1. The orders

(13) 6 43

2.0 6 67

will cause a jump unless 6_C = 14, and so on. If the modifier in *X*5 does not exceed 1023 (or 127.7) then the following group of orders will cause a loop stop unless the counter in 4.0 is equal to the modifier in *X*5.



$6_C = 1 + 5_M - 4.0_C$

jump unless $5_M = 4.0_C$

This group of orders might perhaps be classified as a trick; note that 4.0_M and 5_C do not affect the test, and neither do the sign digits. The 67-order can, of course, jump elsewhere if a stop is not wanted. One can sometimes usefully mark that a certain group of orders has been obeyed by replacing an unconditional jump by a 66- or a 67-order which, under the circumstances, will always cause a jump. For example, suppose $5_M = 0$ and the order

2.0 5 66

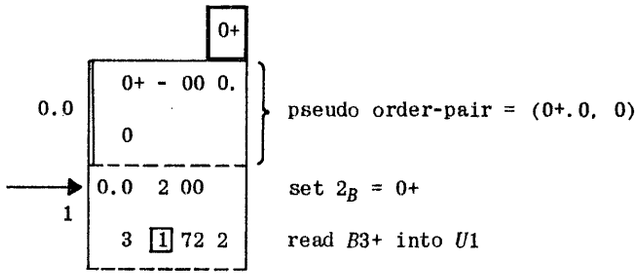
is obeyed; this is now effectively an unconditional jump, but it sets $5_M = 1$ and this can be sensed later, by such orders as the following, for example.

35 5 03

3.0 5 66

Here there will be no jump if 5_P was 0. Care should be taken lest such orders set OVR however.

We have described in Sec. 5.5 how pseudo order-pairs may be used for setting modifiers and counters. Sometimes a *relative modifier* may be used: this is especially useful if the part of the programme being considered starts beyond the first part of the store, and programme transfer-orders must be modified. For example, the following might be the start of a subroutine.



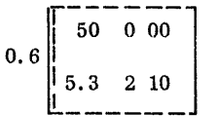
pseudo order-pair = (0+.0, 0)

set 2_B = 0+

read *B*3+ into *U*1

A subroutine written like this may be placed anywhere in the main store.

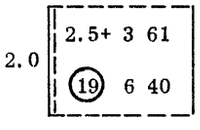
Sometimes only a modifier is required and we are not interested in the value of the associated counter; we may be able to use a dummy *a*-order to give us this. For example, the following order-pair might appear in a programme.



Dummy order

The modifier 0.6_M is here 400.0 (400 = 8 × 50), and the *a*-order has no effect (except to waste a very small amount of time). This word could be copied into an accumulator to give a modifier of 400.0; such an order-pair should be marked by a vertical broken line on the left to indicate that it is also used as a constant; this is to warn those who might wish to alter the routine.

We can sometimes use an actual order-pair to provide a modifier, particularly when only the position part matters (see Sec. 5.5), or when we are modifying only jump orders. For example, consider the following word:



The modifier 2.0_M can be worked out from a knowledge of how the parts of an order are represented (see Sec. 3.12); the block part 2.0_B is formed from the *N*- and *X*-digits of the *a*-order and is here equal to 253 in *octal*, or $\{(2 \times 8) + 5\} \times 8 + 3 = 171$ in decimal; the position part, 2.0_P, is 6; so the whole modifier 2.0_M is 171.6. Note that such an order-pair may be set either by an 00-order or, alternatively, by an 02-order. If the above order-pair is set by an 02-order the resulting modifier is 1023.7 - 171.6 = 852.1. Such devices should be used with caution. Note that we can remove an unwanted counter from a modifier by a pair of orders such as the following.

35 5 02

2.0 5 05

An ingenious application of this technique†, which is definitely a trick, is the *self-modified link*. This is a link which, when obeyed, is simultaneously in an ordinary register and in an accumulator (usually $X1$); consider, for example, the following sequence of orders.

```
        0
0.0
        0.1   1 00
      ---------------
        100 [3] 72 1      B903 to U3
1
        3.1   0 60
```

The order-pair in 0.1 is in $X1$ when it is obeyed in 0.1; the block part of the modifier 1_B is equal to (100 × 8) + 3 = 803 so that the modified 72-order actually transfers $B903$ = 803 + 100. Such an order-pair can be used to return from a subroutine to a programme held beyond the first part of the store, provided of course that $X1$ is untouched by the subroutine. Only certain blocks can be transferred by such a link; for, consider a general self-modified link as follows.

```
        B  [U] 72 1
        U.P   0 60
```

The block part of the modifier (which is all that is relevant) is $8B + U$ so that the block transferred is $9B + U$. For example only blocks whose numbers are divisible by 9 may be transferred to $U0$.

The above tricks are based on using an order as a modifier. Sometimes we can use a part of a number as an order; the only really useful trick consists in using the "a-order" of a word normally used as a small negative integer; this is a 77-stop.

There is a range of tricks based on using as a modifier a word which is numerically a small integer. The modifier is zero if the integer is non-negative, and is 1023.7 if the integer is negative.†† For all but the arithmetical orders a modifier of 1023.7 can often be treated as having the value −1. Consider, for example, the order

```
3+ [0] 72 6
```

If $C(6)$ is a small non-negative integer, this order reads $B3+$ into $U0$; if, however, the integer is negative then the above order will read $B2+$. In the same circumstances the order

```
0.5+ 0 60 6
```

will jump to 0.5+ if $C(6) \geqslant 0$, or to 0.4+ if $C(6) < 0$. We can sometimes in this way combine two tests in one order, such as

```
2.0  3 63 6
```

which will jump to 2.0 if $C(6) \geqslant 0$ and $C(3) < 0$, or to 1.7 if $C(6) < 0$ and $C(3) < 0$, and will not cause a jump if $C(3) \geqslant 0$; assuming $C(6)$ is not a large integer. It is sometimes useful to test OVR by an order of this kind.

Note that if we use this kind of trick with any of the arithmetical orders then it is only the position part of the modifier which is used; this has the value 7 if the small integer is negative. The address written in the order is never decreased by modification (unless it refers to $U7$). For example, the order

```
5.6  4 01 6
```

will add $C(5.6)$ into $X4$ if $C(6) \geqslant 0$; but if $C(6)$ is a small negative integer then the operand is $C(6.5)$, which is zero (see Sec. 3.9). Similarly, the order

```
5.7  4 05 4
```

will collate $C(4)$ and $C(5.7)$ if $C(4)$ is a small non-negative integer; but the order will clear $X4$ if it contains a small negative integer. Another useful order of this kind is the following

```
7.5  4 12 4
```

If $X4$ contains a small non-negative integer this order will have no effect; if however $C(4)$ is negative then $4_p = 7$ and the modification changes the effective address into 7.5 + 7 = 8.4 and this is equivalent to 4 because of the way the N-address is represented (see Sec. 3.12) and the fact that only the 7 least-significant digits in the modified N-address are effective (see Sec. 5.8). If, therefore, $C(4)$ is a small negative integer, the above order has the same effect as the order

```
4  4 12
```

---

† Due to Mr. D.G. Owen.

†† In fact $6_M = 0$ if $0 \leqslant C(6) < 2^{25}$, and $6_M = 1023.7$ if $-2^{25} \leqslant C(6) < 0$.

and $C(4)$ is changed in sign. The net result is that the order written above replaces $C(4)$ by its modulus. Similarly the order

```
0  7 12 6
```

can be used to change the sign of $C(7)$ if $C(6)$ is negative.

We can now explain the trick we referred to in Sec. 3.8 for correcting the sign-bit in $X7$ after the multiplication of two possibly negative integers. Suppose that we have two such integers in 5.0 and 5.1, then the orders

```
5.0  7 00     first integer to X7
5.1  7 20     multiply by second integer
4.0  7 10     product to 4.0
```

will put the product correctly into 4.0 provided it is non-negative and less than 2^{38} (see Sec. 3.1). If the product is negative we have to insert a 1 in the sign position in $X7$ before using the product; we gave in Sec. 3.8 the following orders for doing this.

```
    6 62
32  7 01
```

We can now see that these two orders can be replaced by the following single order,

```
25  7 01 6
```

If the product is non-negative 6_p is zero and the content of special register 25 is added into $X7$. This does not change $C(7)$ since $C(25)$ is zero (see Sec. 2.9). If the product is negative 6_p is 7 so that the effective address in the order is 25 + 7 = 32 and the 1 is inserted in the sign position. We can usefully apply another trick mentioned earlier in this section to detect overflow, which in this context means that the product is not single-length. If the product is single-length then $C(6)$ as an integer is either 0 or −1 and therefore $X6$ will be cleared by an arithmetical halving. The complete sequence may therefore be written as follows.

```
5.0  7 00      first integer to X7
5.1  7 20      multiply by second integer
 25  7 01 6    correct sign in X7
 (1) 6 51  }
 ↻   6 61  }   stop if product is not single-length
4.0  7 10      corrected product to 4.0
```

The overflow test may, of course, be omitted or adapted if appropriate.

The orders with functions 05, 06, 15, 16, 45 and 46 may often be used to provide trick effects.

The reader will have noted that none of the tricks described in this Section does anything that could not be done by more conventional methods. All that can be said for them is that they save orders and a little computer time, at the expense of increasing programming time and the difficulty of understanding or altering the programme later. They are sometimes useful when a small mistake has been discovered in a programme and an extra order or two are needed to put it right (this is quite a common situation): it may then be possible to make enough room by using a trick in an adjoining section of the programme.† To make these small corrections easy to put in it is recommended that those parts of a programme where timing and compactness are not important should be written in a most straightforward way, and that a few dummy orders should be left in each block. Quite often there will be two good ways of doing a small piece of calculation, but one of them might be slightly quicker and take one or two more orders. If this way is chosen, then we can always substitute the other one at a later date to make room for a correction, should this be necessary.

† There is a well known "theorem" to the effect that any programme can be shortened by one order!

Chapter 6

Input and Output

This chapter describes the input and output devices and the tape-editing equipment for handling punched paper tape. The rules are also given for preparing programme tapes. There is in addition a detailed description of the various controls, hand-switches and monitors and how to use them.

6.1 Punched paper tape

The equipment fitted in a basic Pegasus installation so that orders, numbers and other information can be inserted into the computer and extracted from it uses standard five-hole (or five-channel) punched paper tape. This tape is about 11/16 in. wide (1.746 cm.) and carries a row of small holes punched ten per inch along its length; these are the *sprocket holes*, which are used to move and position the tape. Each sprocket hole defines the position of a *tape character*, which is made up of a maximum of five holes (each slightly larger than a sprocket hole), punched in a row across the tape and in line with the sprocket hole. Three of the five channels are on one side of the sprocket hole and two are on the other.

The input equipment is normally two photo-electric tape-readers (Ferranti type TR5) to read the input tapes into the computer (plate 3). these read up to 300 characters per second. The output equipment is normally a single output punch (Teletype) which can punch up to 60 characters per second (plate 4). This output tape is usually fed immediately into a Creed *interpreter*, consisting of an automatic tape transmitter (a kind of tape reader for reading the tape) connected to a page teleprinter (Creed model 75), which we shall often refer to as a printer;† this interpreter prints 10 characters per second. The output punch is six times as fast as the interpreter and it usually operates in bursts. If a programme causes a great deal of output then a loop of tape will form between the output punch and the automatic transmitter of the interpreter. If the tape should become taut a taut-tape device on the tape transmitter stops the transmitter so that the tape is not torn.

Normally the printer will print a single character (e.g. a decimal digit or a letter) each time the automatic transmitter reads a tape character. With five-channel tape there are just $2^5 = 32$ different combinations of holes, each of which is a distinct tape character. Since we want to be able to use more than 32 different printed characters it is arranged that most of the tape characters correspond to *two* printed characters each. The character which is actually printed depends on whether the printer is in *figure shift* or in *letter shift*. There are two special characters which determine which shift is to be used for subsequent printing. When the printer receives the figure shift character (usually abbreviated to the Greek letter ϕ (phi), or sometimes to FS) then it will print all subsequent characters in figure shift until it receives a letter shift character (usually abbreviated to the Greek letter λ (lambda), or sometimes LS), when it will start printing in letter shift. The characters ϕ and λ do not themselves cause any printing. The two shifts resemble in some ways the upper and lower case shifts obtainable on a typewriter. We assumed in Sec. 4.1, where we described the output of numbers, that the printer was in figure shift; this is the normal or standard shift.

The complete tape code used with Pegasus is given in Table 6.1. The first column shows the tape characters and the last two columns the corresponding printed characters in figure shift and letter shift. The characters ϕ (figure shift), λ (letter shift), full stop and Erase are common to both shifts. The other 28 tape characters each correspond to two printed characters. The two columns headed $N(17,16)$ are explained below. The characters CR, LF and Sp are the layout characters described in Sec. 4.1; they are available only in figure shift. The Erase character has holes in all five positions on the tape; this character is printed as a star (✱) in either shift and is usually referred to as Er. Any tape character can be converted into Er by punching extra holes in it; for this reason it can be used for correcting tape-punching errors and we usually arrange to ignore any Er characters on the tape. Notice that the character ϕ (figure shift) is the blank character; it has only a sprocket hole. A number of successive ϕ's is usually called a length of *blank tape*, though it is not strictly blank because of the sprocket holes. All our tapes will start with a *leader* of about 6 or 8 inches of blank tape to facilitate their insertion into the tape readers and automatic transmitters. This automatically ensures that the first character after the blank tape will be read in figure shift (the normal shift). The output punch and the units of tape-editing equipment for preparing tape are each fitted with a *run-out* button, which causes blank tape to be punched as long as it is held down.

The tape code has the property that the most used characters (in figure shift) have each an odd number of holes. These *odd-parity* characters are the decimal digits (0 to 9), the signs (+ and −), full stop (⊙), LF, Sp and Er. The reason for choosing a code with this property is that it makes a certain amount of checking possible. The majority of errors due to faulty operation of the input or output

† Two telegraphy terms: an automatic tape transmitter is a device which reads punched paper tape and transmits electrical signals to a teleprinter, and a page teleprinter is a teleprinter which produces a print-out in page form - not in the strip form seen on Post Office telegrams.

TAPE	*N*		PRINTER	
	17	16	FIGS.	LETS.
·	0	16	**FIG. SHIFT** (ϕ)	
· ●	1	1	1	A
·●	2	2	2	B
·●●	3	19	*	C
●·	4	4	4	D
●· ●	5	21	(	E
●·●	6	22	)	F
●·●●	7	7	7	G
● ·	8	8	8	H
● · ●	9	25	$\neq$	I
● ·●	10	26	=	J
● ·●●	11	11	–	K
●●·	12	28	*v*	L
●●· ●	13	13	**L F**	M
●●·●	14	14	**Sp**	N
●●·●●	15	31	,	O
● ·	16	0	0	P
● · ●	17	17	>	Q
● ·●	18	18	$\geqslant$	R
● ·●●	19	3	3	S
● ●·	20	20	→	T
● ●· ●	21	5	5	U
● ●·●	22	6	6	V
● ●·●●	23	23	/	W
●● ·	24	24	×	X
●● · ●	25	9	9	Y
●● ·●	26	10	+	Z
●● ·●●	27	27	**LET. SHIFT** (λ)	
●●●·	28	12	.	.
●●●· ●	29	29	*n*	?
●●●·●	30	30	**C R**	£
●●●·●●	31	15	Erase (✱)	

Table 6.1 *The tape code.*

devices or the tape-editing equipment result in the gain or loss of a single hole when punching tape, or the misreading of a single hole when reading tape. This kind of faulty operation will always convert an odd-parity character into an even-parity character, but cannot, for example, convert one decimal digit into another. A single fault will usually produce an obviously wrong result; for example, a missing hole could turn a 3 into an asterisk, or a 9 into a multiplication sign. This checking is an example of a *parity check*, which finds many applications in computers (see also Sec. 6.7).

The design philosophy in Pegasus assumes that the majority of errors are due to the input and output equipment, the tape-editing equipment, and the stores. These devices are all subject to a parity check (see Sec. 6.7). Errors due to human frailty can often be detected or prevented by careful programming.

It should be noted that many different tape codes are in current use with different computers and other devices. The code described above is the standard Pegasus/Mercury tape code.

6.2 Output

The two special registers whose addresses are 16 and 17 are used for input and output. We have already described in Sec. 4.1 how most of the useful characters can be punched by sending a small integer (the *value* of the character) to register 16. In fact we can also punch a character in the same output tape by sending the integer to register 17, but the resulting character will in general be different.

When we send an integer (less than 32) to register 17 it can be thought of as going *directly* to the punch; its binary digits are punched as a hole for a 1 and no hole for a 0. For example, the orders

```
(3) 2 40
17  2 10
```

will cause the tape character for an asterisk to be punched (if this character is read in letter shift it corresponds to the letter C). This character has the two right-hand (or "least-significant") holes punched, it may be written as 000.11 if we write a 1 for each hole, a 0 for the absence of a hole, and a point for the sprocket hole. If, on the other hand, we send an integer (less than 32) to register 16, then it undergoes a conversion before being punched. If the integer is 15 or less, then it is adjusted (if necessary) so that the tape character punched has an odd number of holes - this is done by inserting the left-hand (or "most-significant") hole when required. For example, the orders

```
(3) 2 40
16  2 10
```

will cause the tape character for 3 to be punched. Since $3 \leqslant 15$ this character has odd parity; it may be written 100.11, and could alternatively have been punched by sending 19 to register 17. A similar process is applied to integers greater than 15 to ensure that the resulting tape character has an even number of holes. Thus the orders

```
(19) 2 40
16   2 10
```

will cause punching of the tape character for asterisk. Since $19 > 15$ this character has even parity. The main purpose of register 16 and its conversion equipment on output is to make it easy to punch the decimal digits (see Sec. 4.1).

It will be seen that a tape character has two values, one via register 16 and the other via 17; these two values are given in Table 6.1 in the columns headed N (17, 16). Generally we use register 16 since the decimal digits and a few other commonly used characters have convenient small values, as given in Table 6.2.

Printed Character (in figure shift)	0	1	2	3	4	5	6	7	8	9	+	−	.	LF	Sp	Er	ϕ	CR
Value via 16	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	30

Table 6.2 *Values of some commonly used figure shift characters via register 16.*

Register 17 is usually more convenient than 16 for the output of letters of the alphabet, because the value of the n-th letter is simply n (i.e. A has the value 1, B is 2, C is 3, and so on). Note, however, the useful order

```
17  0 10
```

which causes ϕ to be punched. If this order is obeyed repeatedly we get a length of blank tape.

Whenever a word is sent to the output punch, whether via 16 or 17, it is the five least-significant digits only which determine the character to be punched; these are digits 34 to 38. Digits 0 to 33 have no effect on the punch. The punch can be operated by any order of group 1 whose N-address is 16 or 17 (it is occasionally useful to use a 12-order in some tricks); such an order is called an *output order.*

The normal output punch operates at a maximum speed of 60 characters per second, so that it takes about 17 milliseconds (or about 133 word-times) to punch a single character. This time is long compared with the time required by the computer to obey a single order. When a character has been sent to the punch by an output order (via 16 or 17) the computer does not wait for the punch mechanism to operate but carries on and obeys orders in the usual way. In other words, the punch operates independently or *autonomously* while the computer is obeying further orders. Should another output order be encountered before the punch mechanism has finished, then the computer is made to wait. In fact during the 17 milliseconds that the punch is operating it sends a continuous *busy signal* to the computer; this signal holds up the computer if it is about to obey a further output instruction. In Sec. 4.1 we disregarded all timing considerations when dealing with the output punch; this cannot result in incorrect operation. There is a *paper tape* light on the control panel (Plate 11) which is lit while the punch (or the tape-reader) is operating.†

† On Pegasus 1, the output busy signal holds up the computer if it refers to registers 16 or 17 on an input or output instruction. However, on Pegasus 2, each of the standard input and output channels has its own individual "busy" line.

On the control panel of Pegasus 1 are two "busy" lights labelled "input busy" and "output busy" (although on Pegasus 1 with magnetic tape equipment the "output busy" light is replaced by "magnetic tape busy").

When the tape has been punched it normally passes straight to the interpreter for printing at 10 characters per second. The interpreter is for convenience mounted on the control desk of the computer (Plate 4), but can be considered as entirely separate from it since what happens to the output tape when it has been punched does not in any way affect the computer.

Apart from occasional punching of layout characters (CR, LF and Sp), most of the output requirements of programmes are met by subroutines. We discussed in Sec. 4.1 the way in which these subroutines can be prepared.

6.3 Input

Input can be thought of as the converse of output. There are two tape-readers, referred to as the *main* and *second* tape-readers, which can be used to sense photo-electrically the holes punched in the two input tapes and to move the tapes. The external-conditioning order (function 74 - see Sec. 3.10) selects which of the two tape-readers is to be used. We shall simply refer to "the tape-reader" when we mean the selected tape-reader, and to "the input tape" when we mean the tape in this reader. We can read a single character from the input tape by taking a word from registers 16 or 17. For example, the order

```
16  2 00
```

is called an *input order* or *read order;* it will place in *X*2 the value of the character in the reading position in the tape-reader. The value via 16 (as described in the previous Section) is the one used. As soon as the tape has been read the tape-reader automatically moves the tape forward so that the next input order reads the next character on the tape. For example, if the character in the reading position is 3 and the next character is Sp, then the orders

```
16  2 00
16  3 00
```

will place 3 in *X*2 and 14 in *X*3.

Suppose, for example, we have to read three characters representing decimal digits from the tape and build up the corresponding binary number in *X*7. For example, if the characters are 2, 4 and 9 then 249 is to be put into *X*7; we can use the following relationship.

$$(2 \times 10 + 4) \times 10 + 9 = 249.$$

The following sequence of orders will carry out this operation.

```
(10)  5 40    10 to X5
 16   7 00    first digit to X7 (e.g. 2)
  5   7 20    multiply by 10 (e.g. 20)
 16   7 01    add next digit (e.g. 24)
  5   7 20    multiply by 10 (e.g. 240)
 16   7 01    add last digit (e.g. 249)
```

Note how these orders effect the decimal to binary conversion. They do this mainly because the tape character for any decimal digit has a value equal to that digit when read via register 16.

The process of input via 16 is the exact converse of output via 16. Special checking equipment is used which converts the tape character into an integer of value 15 or less if it has an odd number of holes, or into an integer greater than 15 if it has an even number of holes. This greatly facilitates the detection of punching errors, tape-reader malfunctioning, or the blocking of holes by pieces of fluff.

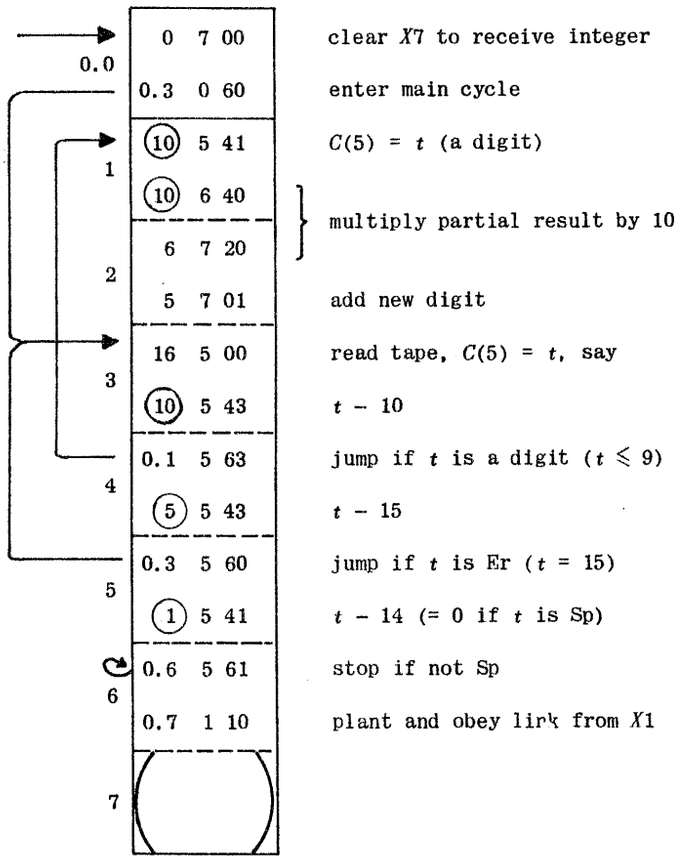
A sequence of orders like the above would not be used in practice, because it does not include any checks on the punching and because the preparation of a tape with many numbers would be very laborious and difficult to check. The sequence could be described as a *primitive* input routine, since the way in which the tape has to be prepared is rigorously prescribed. For example, there must always be just 3 decimal digits on the tape, with no other characters (left-hand zeros must be punched as such).

Generally speaking input, even more than output, is a process which is best left to a subroutine. Certain punching rules will have to be followed when preparing the tape for any particular subroutine; if possible the subroutine should be written in such a way that its punching rules are convenient and natural. What seems natural and simple to the person preparing the tape may be difficult and complicated when expressed in the form of precise rules. For this reason many input subroutines are complicated and difficult to write; they must give a certain amount of latitude to the operator of the perforator. For example, it is highly desirable to ignore Er (the Erase character) wherever it occurs. A certain amount of checking is useful so that spurious characters among the significant ones cannot give misleading results.

As a simple illustration of some of the ideas involved let us construct a subroutine to read in an unsigned integer from the input tape, convert it to binary and leave it in *X*7. The subroutine will read in one by one the characters representing the decimal digits of the number until it encounters a Sp, which is taken as terminating the number; Er will be ignored. As each character is read in it has to be tested because it may be any of the following:

(a) a decimal digit,
(b) Sp,
(c) Er,
(d) some other character.

If it is (d) we shall have a loop stop to indicate a fault or (more likely) a punching error. If the character is Er we must return to read the next character. If it is Sp we must obey the link. On entry, the subroutine clears $X7$ then, whenever a decimal digit is read, $C(7)$ will be multiplied by 10 and the new digit added to it. The following is a possible subroutine.



This subroutine would not normally be acceptable, for a variety of reasons. If, for example, the first character read is Sp then the link is immediately obeyed; as a rule it would be desirable to ignore Sp before the first digit is encountered, but to treat it as a terminating character afterwards. Again, we would usually want to treat CR LF (i.e. CR immediately followed by LF) in the same way as Sp, so that we could have numbers on several lines. Often we should want to handle signed integers and we must then arrange to accept the sign characters (of values 10 and 11) before (but not after) the digits and to change the sign of the number, if necessary, at the end just before obeying the link. It would probably also be desirable to ignore blank tape under certain circumstances. When an input subroutine is written to meet all these and other requirements it becomes quite complicated; each character is subjected to a whole series of tests and checks before any other action is taken on it. Subroutines like these are to be found in the Library; they are easy to use and it is easy to prepare tapes for them. Generally speaking, each subroutine will be designed for some particular kind of number - integers, fractions, mixed numbers, sterling, and so on; and different verions of certain subroutines are available.

Input via register 17 is the converse of output via 17 except that the integer resulting from reading the tape appears in the *modifier position*. For example, if the character in the reading position is 3 then the input order

17 2 00

will put 19 into 2_M and clear the rest of $X2$ (see the tape code in Table 6.1). Input via 17 is not often used but is very convenient when some sort of code conversion is to be applied to the characters as they are read in; this is why the modifier position is used.

A tape-reader of the type normally fitted operates at a maximum speed of about 300 characters per second, so that it takes about $3\frac{1}{3}$ milliseconds (or about 27 word-times) to move the tape from one character to the next. When a character has been read by an input order (via 16 or 17) the tape-reader immediately starts to move the tape to the next sprocket hole (or character); it does this quite independently of what orders the computer may be obeying - another example of autonomous operation. Should another input order be encountered before the tape has finished moving to the next character, then the computer is made to wait and the tape is not brought to rest, the character being read as the tape moves by. In fact during the $3\frac{1}{2}$ milliseconds that the tape is being moved the tape-reader sends a *busy signal* to the computer; this signal holds up the computer if it is about to obey a further input instruction.† If all timing is disregarded incorrect operation cannot occur. The paper tape light on the control panel is turned on by the busy signal from the tape-reader.†† We attempt to write

† On Pegasus 1, the input busy signal holds up the computer if it refers to register 16 or 17. Thus the busy signal sent by either the tape-reader or the punch inhibits all reference to registers 16 and 17. This means that the computer will be made to wait if it encounters an input order while the output punch is operating, and vice-versa. The fact that the busy signals are not distinguished within the computer does not often matter.

†† On Pegasus 1 a busy signal from the tape-reader lights an "input busy light".

input subroutines so that the computer takes rather less than $3\frac{1}{3}$ milliseconds between input orders; the input tape is then kept running at full speed and the input busy light is on continuously while the tape is being read in. Note that the tape will always stop in time if there is a long interval between one input order and the next.

A code conversion can often be achieved by a table look-up operation (see Sec. 5.8). There is an interesting process† for doing this which is well suited to the kind of conversion often required on input. Here we have a 5-bit integer t ($0 \leqslant t \leqslant 31$) which will normally be the value of a tape character read via register 17. We have to convert this integer into another integer u by looking up the t-th entry in a 32-entry table. Often we shall not want to use a whole word for each entry and we can therefore pack several entries into one word. If we can pack four entries into each word the whole 32-entry table can be held in the 8 words of a single block. Consider the following sequence of orders, where the table is supposed to be in $U4$.

```
17  2 00      t to 2_M
4.0 6 00 2    extract k-th word of table
(0) 6 52 2    shift up t places
```

The first order simply reads the tape in the "direct" way and puts t into 2_M. The next order is an arithmetical order and will therefore be modified by the last 3 bits of t only (let us call this k; it is the remainder when t is divided by 8). The last order is modified by the whole of t since it is an order of group 5. For example if $t = 20$ then the remainder k is 4 and the above orders will pick out the word in 4.4, copy it into $X6$ and shift it up 20 places; if $t = 28$ the same word will be put into $X6$ but it will be shifted up 28 places. The 8 most-significant binary digits in $X6$ constitute u, the result of the table look-up. The rest of $C(6)$ can be discarded if required by shifting or collating. These 8 bits can be arbitrarily assigned for each of the 32 entries. The whereabouts of each entry are given in Fig. 6.1; each numbered space represents an 8-bit entry, the numbers being the value of t needed to extract the entry in the above way. The shaded parts are not accessible.

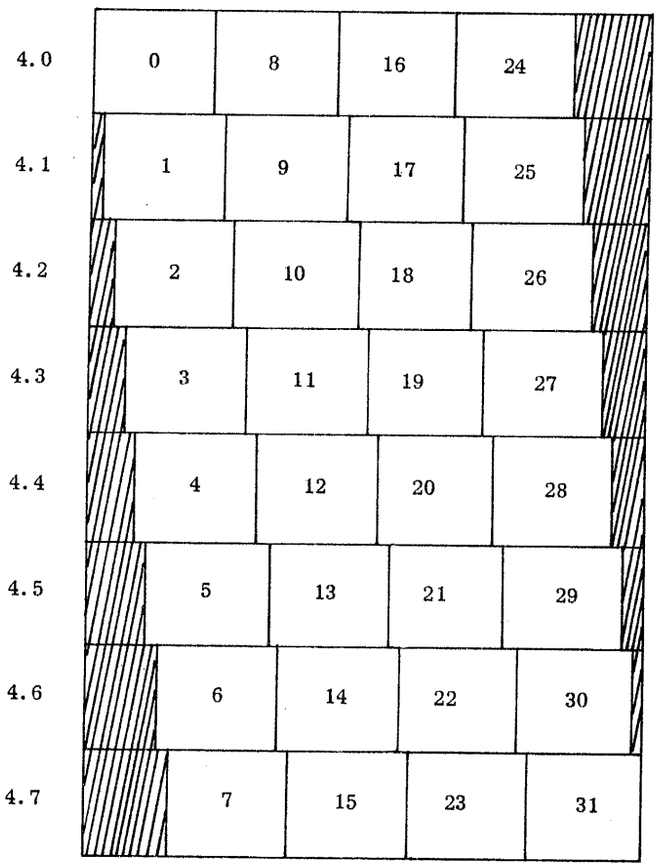


Fig. 6.1 *A packed 32-entry table.*

This block would normally be written on the programme sheet as eight pseudo order-pairs (see Sec. 3.12).

6.4 The tape-editing equipment

The equipment for the manual preparation, editing and printing out of the punched paper tape is called tape-editing equipment. The following facilities are provided.

(a) Punching tapes manually from a keyboard, with or without simultaneous printing.

(b) Printing out tapes.

(c) Copying tapes with simultaneous printing and, if required, the insertion of corrections from a keyboard or another tape.

(d) Comparison of two tapes to detect discrepancies due to punching errors, with or without the simultaneous preparation of a print-out and a third "perfect" tape free from errors.

† Due to Mr. D.G. Owen.

The simplest item of equipment is the *keyboard perforator* (Plate 9) which consists simply of a keyboard which operates a tape-perforator. It can be used only for the manual preparation of tapes without printing, and is consequently used only for very short tapes. There is usually a keyboard perforator near the computer.

An important unit in all installations is the *page teleprinter* (Creed Model 75) with keyboard and reperforating attachment. It can be used by itself for the manual punching of tapes with simultaneous printing, or it can be connected to other devices to provide full editing facilities. It provides the most convenient means for preparing tapes manually because a printed record (a print-out) is obtained while the tape is being punched. The keyboard of this instrument (Plate 10) resembles that of a typewriter except that the keys for the decimal digits (and certain other characters) have been grouped together at the right. This grouping makes it possible to prepare most programme and data tapes with one hand only. The *run-out key* is at the top right corner of the keyboard; if it is depressed, continuous punching of any character may be obtained by also depressing the key of the character required. The run-out will cease when either key is released.† The operation of any other key causes a single tape character to be simultaneously punched and printed, except that a few characters produce no actual printing (these are ϕ, λ, CR, LF, Sp). Figure shift (ϕ) is punched by pressing the bar labelled FIGS/BLANK and letter shift (λ) is punched by operating the key on the left labelled LTRS (for "letters"). The keys are engraved with the printed characters, and there are consequently two keys for each tape character (with a few exceptions). The keys for the characters available only in letter shift (chiefly the letters A, B, ..., Z) are locked when the teleprinter is in figure shift, and vice-versa. Thus the key engraved B is free only in letter shift and the key engraved 2 is free only in figure shift; these keys cause the same tape character to be punched. The keyboard perforator has a similar keyboard.

Most punching errors are noticed immediately they have been made; the operator can back-space the tape and over-punch incorrect characters with Erase (Er, which has all 5 holes punched); the over-punching (but not the back-spacing) is printed out simultaneously. The keyboard perforator has a similar facility.

The printing is on a continuous roll of paper wide enough for 69 characters. There are 10 characters to the inch horizontally, and there are 6 lines to the inch vertically. The tape is produced by the reperforating attachment and comes out from the side of the teleprinter; the action of this attachment can be suppressed if desired, when the teleprinter will print without punching.

The *simplified tape-editing equipment* (Plate 8) consists of a teleprinter of the kind just described connected to an automatic transmitter (a kind of mechanical tape-reader) and a small control box; the equipment is sometimes called a *reproducer*. The automatic transmitter can be used to read a tape and operate the teleprinter, which then prints what is on the tape and punches a new copy of it (unless punching is suppressed). At any time the automatic transmitter can be stopped and characters inserted from the keyboard. When the teleprinter is operated from the automatic transmitter the printing is in black; when the keyboard is used the printing is in red (the latter is usually the result of manually inserted corrections). The tape in the automatic transmitter can be *inched* forward character by character by operating a key on the control box; the characters inched over in this way can be copied or ignored as desired.

This set of equipment is adequate for the preparation of most tapes. The print-out which is obtained makes it easy to prepare and edit a tape since it gives a visible record of operations. The usual procedure for preparing a programme tape is as follows. Firstly, the tape is punched by an operator directly from the programme sheets (the punching rules are described in Sec. 6.5). Secondly, the print-out obtained during the punching is proof-read against the original programme sheets and any errors are marked on the print-out. Lastly, the tape is reproduced with the aid of the automatic transmitter, and the resulting print-out is watched. Just before a punching error is reached the copying is stopped and the tape inched up to the last correct character; the incorrectly punched characters are then inched over one by one without copying and corrections are put in on the keyboard. This procedure is repeated for each error. When the end of the tape is reached the new print-out will have all corrections in red and can be quickly checked. Note that one can prepare and correct a tape in this way without learning the tape code; nearly everything can be done by watching the print-out.

The simplified tape-editing equipment can also be used for joining up a number of short tapes into a longer one by copying. As a rule this is not recommended since it can take a long time and errors may be inserted during the copying. It is better to splice the several tapes together, as described below.

The *full set of tape-editing equipment* (Plate 7) provides all the facilities of the simplified set and can in addition be used for comparing two tapes, and correcting errors discovered as a result. It is made up of the following items of equipment.

(a) a page teleprinter with keyboard and reperforating attachment, as described above,

(b) an automatic transmitter (as in the simplified set),

(c) a triple-head tape-reader or comparator (two of the heads are used when comparing tapes automatically, while the third can be connected to the teleprinter for copying or printing out tapes),

(d) a control unit,

(e) an independent keyboard perforator.

The equipment also includes a power unit. The keyboard perforator is an entirely independent item but the other units mentioned above are all connected together.

The method described above for the preparation of programme tapes is not suitable as a rule for data tapes. Programme tapes are often prepared by the programmer himself, who can easily read and

---

† On the previous model of the teleprinter (Creed Model 54), it was only possible to run-out the blank character (i.e. ϕ or figure shift).

understand the programme sheets and who will be on the watch for slips and other minor errors, many of which he will probably detect and put right while he is punching the tape. Programme tapes are also fairly short and easy to punch and check. The punching usually comes as a welcome relaxation to the hard-pressed programmer! Any errors which slip through the check are usually picked up quite quickly when the tape is fed into the computer. It is quite otherwise with data tapes. These are nearly always produced from lists of figures with no apparent meaning and may be very long. It is also difficult to detect errors by proof-reading and the consequences of the errors may be serious. It is therefore recommended that data tapes should be prepared twice, preferably by different people using different equipment but working from the same data sheets. The two tapes should then be identical but, in all probability, each will contain a number of errors (perhaps one error in every 300 to 2000 characters). These two tapes can be automatically compared by the comparator section of the tape-editing equipment, this is normally connected to the teleprinter which produces a print-out and a third tape (the so-called *perfect* tape). The comparator reads both of the hand-punched tapes, character by character; as long as they agree the third tape is punched. The device stops as soon as there is disagreement and the print-out can then be inspected to determine the correct next character, which will usually be on one of the two input tapes. This tape can then be chosen as the "master" and copied on to the final correct tape. Alternatively a correction can be put in from the keyboard of the teleprinter, or from the automatic transmitter. The comparator can also be disconnected from the teleprinter and used simply to compare two tapes without producing a third; this facility is mainly used for comparing a copy of a tape with the original.

The full set of tape-editing equipment also includes an automatic carriage-return and line-feed facility to return the carriage of the teleprinter after a preset number of characters have been printed across each line.

The *interpreter* mounted on the desk of the computer (Plate 4) is conveniently described here, although it cannot be used for editing tapes. It resembles a simplified set of tape-editing equipment in that it includes an automatic transmitter and a page teleprinter. However, there is no control box and the teleprinter has no keyboard and no reperforating attachment.

The comparator works at 4.4 characters per second when comparing and producing. The model 75 printer works at 10 characters per second and the model 54 printer at 6.6 characters per second.

In addition to the major items of equipment described above there are a number of *tape-handling accessories.*

The motor-driven *tape-spooler* (Type A1) can be seen near the second tape-reader in Plate 3. It is primarily intended for feeding a long tape into a high-speed tape-reader or for winding up a tape issuing from it. The *hand-operated tape-spooler* (Type A3 - see Plate 6) is for spooling tapes out of bins; there are usually one or two near the computer and the tape-editing installation. Special *tape boxes* (type A4 - see Plate 5) are used to hold tapes at all stages of their preparation and use; each box has a *core* (type A5) which rotates on a pin in the box. These boxes have a slot in the side to allow the tape to be fed out or spooled up inside the box. The spooling up can be done either with the hand-operated spooler or by a spooling knob (type A6). Tapes are usually fed into the tape-readers on the computer from a *plastic trough* (Type A7), one of which can be seen in Plate 3; if desired the tape box can be put into one of these troughs and the tape pulled out through the slot in the box. A number of *tape bins* (Type A8) mounted on castors are normally used to receive the tape coming out of the tape-readers and punch on the computer, and also from the tape-editing equipment. Small cardboard boxes are used to hold short tapes, chiefly programme tapes. The *tape-correcting punch* with splicing facilities or *unipunch* is a useful device (see Plate 13); it has two functions, as follows;

(a) It can be used to insert extra holes or a few extra characters in a tape.

(b) It assists the *splicing* or joining of tape.

A partly spliced tape is shown in Plate 13. The two tapes to be joined are placed in the device, which holds them firmly. The two ends are then cut straight and are covered with opaque adhesive tape (PVC tape) to make a butt joint (end to end); the tapes are then taken out of the device and the adhesive tape is wrapped round the underside of the two punched tapes. This kind of joint is normally made only where there are lengths of blank tape; it covers over the sprocket holes but this does not affect the operation of the high-speed tape-readers on the computer. Such a join cannot, however, be passed through the tape-editing equipment. In general this method of joining is preferable to copying with the tape-editing equipment; it is quick and easy to follow.

By using all these various items of equipment it is possible to prepare and edit punched tapes for the computer. Great care should be taken that the tapes are correct before they are fed to the computer and, if at all possible, numerous short sections of tape should be spliced into a long tape in order to save computer time, and prevent errors.

6.5 The preparation of programme tapes

We shall now describe how a programme tape is prepared. We shall assume that the punching is done on a page teleprinter (with keyboard and reperforator attachment) as described in the last Section. We therefore get a print-out while the tape is being punched. If the tape is instead punched on a keyboard perforator then the only difference is that no print-out is obtained; the actual punching procedure is unchanged. The description which follows is given in the form of rules, which should normally be adhered to. Sometimes departures from the rules are allowed, but these will not usually be mentioned and the beginner is advised to follow the rules carefully. In general it will be found that the punching rules are natural and easy to follow and will result in an acceptable print-out for proof-reading. We assume that the complete programme or subroutine is all written out, complete with directives, ready for punching. A few additional rules are needed to cover certain punchings required by Assembly, and some special directives; these will be described later. We assume that the programme is to be read in by the Initial Orders (see Sec. 4.3).

A complete tape is made up chiefly of the following;

(a) Lengths of blank tape,

(b) Order-pairs,

(c) Numbers,

(d) Directives.

When the tape is read in by the Initial Orders the blank tape is, by and large, ignored (i.e. it has no effect); the order-pairs and numbers are placed in consecutive storage locations as determined by the Transfer Address, and the directives control the operation of the Initial Orders and are not stored. These four kinds of punching are distinguished from one another by their *first* characters, apart from a few which have no effect, like Er; these first characters are as follows.

(a) Figure shift (ϕ) introduces blank tape,

(b) A decimal digit (0 to 9) introduces an order-pair,

(c) A sign (+ or −) introduces a number,

(d) Letter shift (λ) introduces a directive.

The reason for treating blank tape in a special way is primarily to detect faults in the tape-reader or the tape-editing equipment. Full use is made of a check rendered possible by the special features of the tape code and register 16 (see Sec. 6.3). Most of the characters used for punching are the odd-parity characters (see Sec. 6.1), viz.

0 1 2 3 4 5 6 7 8 9 + − . LF Sp Er

which have each an odd number of holes in the tape. An error of a single hole in reading or punching cannot turn any one of these 16 odd-parity characters into another one. Blank tape (ϕ) is not an odd-parity character, and in fact a single error could cause confusion between ϕ and any of the characters 0, 1, 2, 4 or 8. For this reason it is not desirable to accept blank tape among the digits of an order-pair or number, and some preliminary search is required. In fact if ϕ is encountered in the middle of an order-pair or a number the Initial Orders will encounter a loop stop.

The following are useful rules about blank tape.

A1. All tapes should start with a leader of at least 6 inches (15 cm.) of blank tape (obtained by pressing the run-out key); the name of the tape, the date and the programmer's (or punch operator's) name or initials should be written on this leader in ink or pencil. This leader facilitates the insertion of the tape into tape-readers or tape-editing equipment.

A2. About two inches (5 cm.) of blank tape should be punched at any natural breaks in the information being punched; for example, after any of the following:

(a) each block of the programme,

(b) each directive (or warning-character sequence),

(c) any group of numbers.

These short lengths of blank tape greatly facilitate splicing the tape and identifying portions of it.

A3. The characters CR and LF should always be punched before anything else after any blank tape. Thus each directive or each block of programme is normally preceded by blank tape CR LF.

A4. If there are several routines or long lists of numbers on the same tape it is advisable to punch about 6 inches (15 cm.) of blank tape between them, and to write here the name of the immediately following routine or list of numbers. In this way we can more easily find a routine in a long tape should this be necessary.

A5. At the end of the tape after the last significant character there should be a tail, consisting of about 6 inches (15 cm.) of blank tape, followed by half-a-dozen Erase characters (Er) and a very short piece of blank tape. The little group of Er characters right at the end of the tape is very conspicuous and effectively prevents our feeding the wrong end of the tape into a tape-reader.

A6. Blank tape should not appear except where indicated above. Isolated figure shifts (ϕ's) are allowed in a few places only; these are described below (in the rules for punching directives).

The reader should particularly note rule A3 above; if CR LF is omitted after blank tape the computer will, in most cases, stop when reading the tape. Incidentally, it will also stop if the wrong end of the tape is placed in the tape-reader.

To avoid repetition in describing how order-pairs, numbers, and directives are punched we can state two *general rules* applicable to all of them.

B1. The character CR should always be followed immediately by LF; the combination is regarded as a composite character and is denoted by CR LF.

B2. The character Er may be punched anywhere except between CR and LF.

The character CR is not one of the 16 odd-parity characters; in fact ER can turn into it by the loss of a single hole, and CR can similarly turn into ⊙ + 6 or Sp. As a check, therefore, the Initial Orders require that the character immediately after CR should be LF (an odd-parity character). The Er character (Erase) has all five holes punched and can be used for correcting punching errors, especially those which the operator realises he has made at the moment he presses the key. †

We shall now describe the rules for punching orders.

C1. Orders must always be punched as order-pairs; single orders cannot be read in.

Each order of a pair is punched in the same way, as governed by the following rules. Note that circles, boxes, vertical lines, etc. are never punched; they are merely guides in reading the programme sheet.

C2. The N-address in an order is punched as written, and followed by enough spaces to make the total number of characters up to four (e.g. 2.0 Sp or 12 Sp Sp or 5 Sp Sp Sp or 2 + Sp Sp). No space is necessary if N has four characters, provided the last is + (e.g. 0.7+ or 100+), if it is not + then a single space should be punched (e.g. 1023 Sp).

The effect of this rule is generally to make the *N*-address have a *width* of four characters on the print-out; the orders will then be printed in well laid-out columns for easy checking. The last character of *N must* be either Sp or +, so that in the exceptional case when *N* is a decimal number of four or more digits we must put in a single Sp after it (this is likely to occur only in pseudo order-pairs but may also occur in single-word transfer orders in which the address is written in decimal - see Sec. 3.10).

---

† He can back-space the tape, cover the incorrect characters with Er, and then punch the correct ones. If he should press a key other than LF after punching CR then he must erase both the incorrect character and the CR, and then punch CR LF.

C3. The X-address, function digits and modifier of an order are punched as written; no spaces are allowed between them. The modifier need not be punched (or written) if it is zero.
C4. Every order must be followed immediately by CR LF.
The following are a few illustrations of these rules:

Order as written	Order as punched	Order on print-out
4.0 6 01	4.0Sp601CR LF	4.0 601
10 2 43	10SpSp243CR LF	10 243
1+ [1] 72	1+SpSp172CR LF	1+ 172
5.0 6 00 4	5.0Sp6004CR LF	5.0 6004
1.2+ 3 63	1.2+363CR LF	1.2+363
[100 -] 70	100Sp-70CR LF	100 –70
[1000 -] 71	1000Sp–71CR LF	1000 –71

In fact it is permissible to punch spaces before the *N*-address; but this is not recommended. The following special rule applies to null orders, i.e. the order which is written as simply 0 on the programme sheet and which represents the dummy order 0 0 00.
C5. A null order must be punched 0 CR LF; no spaces are allowed between the 0 and the CR LF.
The fullowing rules govern order-pairs.
C6. Order-pairs are made up of the a-order followed immediately by the b-order, both orders being terminated by CR LF. Before the a-order LF and CR LF may be inserted if desired.
C7. If an order-pair is a stop order-pair then a full-stop (⊙) should be punched after either the a-order or the b-order, but not both. In this order the modifier must be punched, even if it is zero.
If just one of the orders of a stop order-pair is modified we normally punch the stop after that one. Otherwise we usually punch it after the *a*-order. If a modifier is zero it can, if desired, be punched as Sp but this is not recommended.

Pseudo order-pairs (see Sec. 3.12) are punched according to the same rules as order-pairs, but they usually require more care in punching. Often the "*b*-order" in a pseudo order-pair will be an unsigned integer (see Sec. 5.5), when the following rule applies.
C8. If one of the "orders" in a pseudo order-pair is an integer it should be punched as written and followed immediately by CR LF; no spaces are allowed after the integer.
Such an integer is stored by the Initial Orders at the least-significant end of the order.

The following are some examples of the way order-pairs may be punched.

Order-pair as written	Order-pair as punched	Order-pair on print-out
1.4 7 24	1.4Sp724CR LF	1.4 724
15 5 00	15SpSp500CR LF	15 500
0.2 1 00 0.	0.2Sp1000.CR LF	0.2 1000.
2+ [3] 72	2+SpSp372CR LF	2+ 372
229 - 30 0.	229Sp-300.CR LF	229 –300.
48	48CR LF	48
3274 - -0 0.	3274Sp--00.CR LF	3274 --00.
122	122CR LF	122

Between the blocks of a programme there should be about two inches (5 cm.) of blank tape. The punching should be as follows:

Last order of block (a *b*-order) ending with CR LF,

Blank tape CR LF,

First order of next block CR LF, etc.

See rules A2 and A3 above.

If a block of the programme is not completely filled with orders and numbers then the length of blank tape may be punched after the last order or number in the block. There is no need to fill up the block with dummy orders or numbers, but care should be taken that the last order punched is a *b*-order (i.e. the orders must be punched as order-pairs). Such a partially filled block would normally be followed (after the blank tape) by a directive.

The rules for punching *numbers* are quite straightforward. Two kinds of numbers are recognized by the Initial Orders, *integers* and *fractions;* they are distinguished from one another by the presence or absence of a decimal point (punched as a full stop ⊙).

D1. Every number is introduced by its sign (+ or −), which must always be punched.
A number occupies a whole word in a programme, whereas an order is only half a word; the conversion from the punched form into the internal binary form required by the computer is quite different for orders and for numbers. The sign, punched before any digits, initiates the correct processes.
D2. Every number must be immediately followed by either CR LF or Sp.
Thus the *terminating character* for a number may be Sp or CR LF, whereas orders must be terminated by CR LF (see Rule C4). The reason for allowing Sp as the termination of a number is that numbers are often arranged in small groups; this grouping can conveniently be reflected in the layout of the printing.
D3. Between the sign and the terminating character of a number the only permissible characters are the decimal digits of the number, erases and a single stop (if the number is a fraction); no spaces are allowed between the sign and the first digit or among the digits (if desired Er may be used to group the digits).
D4. Between numbers (i.e. between the terminating character of one number and the sign of the next) the following characters may appear: Sp, LF, CR LF, Er. Blank tape (ϕ) is also allowed provided it is followed by CR LF (punched before the next number).
D5. The absolute value of an integer punched in the tape may not exceed 274877906943 (i.e. $2^{38}-1$).
D6. The sign of a fraction is immediately followed by 0⊙ and up to 11 decimal digits; the fraction will be correctly rounded before being stored (exceptionally the fraction −1.0 is so punched).
The integer -2^{38} should be punched as −1.0. Note that an integer may have up to 12 digits (left-hand zeros need not be punched), but a fraction may not have more than 11 digits after the point. The following are examples of numbers punched in accordance with these rules.

Numbers as punched	Appearance on print-out
Blank tape CR LF	
+92SpSp+0 CR LF	+92 +0
Sp−6SpSp+4 CR LF	−6 +4
−10Sp−22CR LF LF	−10 −22
+0.1234Sp−0.9876 CR LF	+0.1234 −0.9876
+0.01122339 CR LF	+0.001122339
−1.0 CR LF LF	−1.0
Blank tape CR LF etc.	

We shall now explain how *directives* are punched. A directive usually consists of a warning character, which is a letter of the alphabet, and sometimes an address (see Sec. 4.3). We shall later encounter directives which contain two addresses, and the rules for punching these are included below for convenience of reference. Certain directives are in some way special, for example the N-directive, which contains a name.
E1. A warning character is always punched as follows: letter shift, warning character, figure shift.
For example a B-directive is punched λBϕ and this is the complete directive, although it is usual to punch CR LF before it.
E2. A single address in a directive should be punched as written, preceded by Sp (optionally), and followed by CR LF.
The space is helpful to separate the address from the warning character on the print-out but is strictly not necessary. If desired extra spaces may be inserted to align the addresses in a number of directives, but note that only a single ϕ is allowed between the warning character and the address. The address *must* always be followed by CR LF. The following are examples of the punching of directives.

Directive as written	Directive as punched	Directive as printed out
Z	CR LF λ Z ϕ	Z
T 4.3	CR LF λTϕSp4.3 CR LF	T 4.3
E 2.2+	CR LFλEϕSp.2.2+CR LF	E 2.2+
T 2+	CR LFλTϕSp2+ CR LF	T 2+

E3. If there are two addresses in a directive punch the following after the warning character
Sp, first address, Sp, minus, Sp, second address, CR LF.
Here again the spaces are entirely optional but they do improve the appearance of the print-out. For example the directive

F 14.6 − 15.3

Would be punched as follows,

CR LF λ F ϕ Sp 14.6 Sp − Sp 15.3 CR LF.

E4. The following punchings are recommended for the names after the warning character N; in both examples the name is taken to be SYSTEM 3D. A length of blank tape must appear after the name.†

(a) *Names of subroutines and other parts of a programme: blank tape CR LF λ N φ CR LF λ SYSTEM φ Sp 3 λ Dφφφ...*

(b) *Name of a complete programme: blank tape CR LF λ Nφ CR LF LF λ SYSTEM φ Sp3 λ Dφ CR LF φφφ ···*

In the above a sequence of φ's represents blank tape. Single figure shifts (φ) may be inserted where necessary in the middle of a name. Two or more consecutive φ's (normally punched as a short length of blank tape) mark the end of the name; it is important to *punch a length of blank tape after a name.* If this blank tape is omitted the next directive or piece of programme on the tape will be included in the name and be simply printed out when read by the computer. A few extra LFs included in the name of a complete programme serve to make it more conspicuous on the printed output of the computer and on the print-out of the programme tape.

The reader is reminded that Er is ignored everywhere on the tape except between CR and LF (and in names); so that an order may be punched, for example, as

Er 25 + Sp Er 2 Er Er 73 Er CR LF

and the directive T 5.2 may be punched as

CR LF Er λ Er T Er Er φ 5 Er. 2 Er CR LF

though these are rather extreme examples: The rule (A3) that blank tape should always be followed by CR LF is relaxed with directives! strictly the rule should also allow λ after blank tape without any intervening CR LF, but it is advisable to follow the form of the rule given above. It is emphasized that some of the above rules are recommendations which are not strictly binding - they can be regarded as a guide to good practice, and will give an acceptable tape and a legible print-out. Many of the rules must be followed precisely however, and departures are treated as punching errors; these will cause a loop stop when the tape is read by the Initial Orders. This subject is discussed further in Chapter 7.

A fairly common punching error is the omission of an entire order; this is sometimes not detected when proof-reading the print-out against the programme sheet. The error will usually be detected when the tape is fed into the computer, however, since it results in a block containing an odd number of orders. The blank tape at the end of the block will therefore be read at a time when the Initial Orders are expecting a *b*-order and this causes a loop stop (it violates rules A6, C1, and C6 above). The error will be detected at a slightly earlier stage if there are any numbers in the block after the missing order; the Initial Orders will then encounter a sign (+ or −) when they are expecting a *b*-order.

As an illustration of these punching rules we shall now describe the punching of the tape for the programme "Special Factorize" which was described in Sec. 4.4. The reader should compare this description with the programme sheet given in Fig. 4.3 and the print-out in Fig. 4.4. The tape starts as follows.

Eight inches (20 cm.) of blank tape (leader) CR LF

λDφ Two inches (5 cm.) of blank tape CR LF

λNφ CR LF LF λ SPECIAL φSp λ FACTORIZE φ CR LF

Two inches (5 cm.) of blank tape CR LF

+ 420 CR LF

30 Sp Sp 640 CR LF

16 Sp Sp 610 CR LF

13 Sp Sp 640 CR LF

16 Sp Sp 610 CR LF

etc....

The punching at the end of the first block and the beginning of the second is as follows.

7 Sp Sp Sp 401 CR LF (last order of first block)

Two inches (5 cm.) of blank tape CR LF

1.1 Sp 460 CR LF

16 Sp Sp 710 CR LF

etc....

Near the end of the tape the punching concludes as follows (starting here with the *a*-order in *B*5.6).

---

† The name is taken as starting with the φ after the N. The Initial Orders copy this φ and all succeeding φ's until a non-blank character is found; the copying then continues to include the first of two consecutive φ's. For example, in the following sequence the characters underlined are printed as the name.

CR LF λ N φφφφφ CR LF LF λ SYSTEM φ Sp 3 λ D φ CR LF φφφφφ ....

Note that Er is not ignored in this process.

0 Sp Sp Sp 0724.CR LF

0.0 Sp 060 CR LF

+100000000000 CR LF

Two inches (5 cm.) of blank tape CR LF

λ E ϕ Sp 2.1 CR LF

Six inches (15 cm) of blank tape and five Er's
followed by one inch (2½ cm.) of blank tape (tail).

When the tape has been punched it is spooled up using a hand spooler (see Sec. 6.4). The name of the tape, the date and the programmer's name are then written on the leader, and a rubber band is put round the tape. If it is a short programme tape it is then put into a small white cardboard box; a long tape would be put into a larger box. The length of the tape can be estimated in advance by allowing about 16 inches (41 cm.) for each block, including a little blank tape.

6.6 The design of subroutines for input and output

In most calculations input and output deserve study, particularly where there are large amounts of either. It is always advisable to identify the input and output information in some way, preferably automatically. The Initial Orders can provide some help here, if properly used. The date and serial number and the name of each programme and subroutine can be printed automatically as the programme tape is read in; these facilities should be fully used as the record they provide can be invaluable, particularly during the development phase of a programme when errors are being detected and removed.†

Care should be taken to arrange the output material of a programme so that it is as convenient as possible to read. For example, it should be printed in columns of regular layout and divided into blocks by extra line feeds at suitable intervals. Each number should be printed in the form in which it is needed; all internal scaling factors should be removed and the decimal point should be printed in its proper place. It must be easily possible to identify each number; the layout can be helpful for this, and consideration should be given to printing extra information, for example each line or block of printing may be preceded by an identifying number (e.g. the value of some variable). It is often helpful to print names at the heads of columns of figures, or carefully chosen symbols or letters in front of some of the numbers. As a check on the calculation it may be advisable to print extra numbers, such as check-sums. The use of only the odd-parity characters (see Sec. 6.1) and a regular layout of the output matter can be very useful in detecting faults in the output punch. A single error in punching or printing an odd-parity character will result in an obviously wrong character being printed or else in a disturbance of the regular layout of the page.

A good way of checking the output tape is to use a special programme to read it in again and apply certain checks, for example on the layout, the number of digits in each number and its approximate size, the presence of special symbols, and the accuracy of check-sums. In some problems it may be possible to repeat the whole calculation while the output tape is being read in and to check every character on it; this kind of check is particularly useful when the output time is large compared with the computing time since output can be suppressed during the checking run. A check like this is thorough but, of course, the labour of doing it may not be warranted in most problems.

Both output and input subroutines should be carefully written or selected from Library routines. They should be appropriate to the programme and as easy as possible to use correctly. An output subroutine should yield results which are easy to read. An input subroutine should have punching rules which are as simple and as natural as possible. An output subroutine for general use should be accompanied, if possible, by an input subroutine capable of reading in the output tape, complete with all its layout characters. All special cases should be considered when the subroutine is being written. For example, what are the effects of reading or printing the fraction -1.0? When printing fractions, the addition of a rounding constant may cause overflow. It is vital that special numbers such as these should not cause misleading results on input or output.

An input subroutine should give accurate results and incorporate as many checks as possible on the punching. As each character is read in it will normally go through a chain of tests; these should be arranged to reject impossible combinations of characters. For example a subroutine for reading in signed fractions should indicate a failure if it reads more than one sign or decimal point, or if overflow occurs. It is advisable to include as many checks as is reasonably possible on the input to a programme. The hand-punching may well include a number of errors. A particularly powerful check is provided by check-sums, which can be the sum of all the numbers in any convenient group. The numbers can be added, regardless of their significance, on a desk machine and the sum written on the data sheet, for example at the foot of each column of figures. This sum can be punched with all the numbers, and read in and checked by the programme. The programme should also count the numbers as they are read in, if it is known how many there should be. It should also check in some way that it has been supplied with the right tapes in the right order; this can be done by copying names on to the output tape or by using certain identifying numbers or symbols punched in the input tapes.

The punching rules for the Initial Orders (see Sec. 6.5) should be taken as standard. So far as possible or appropriate an input subroutine should be so written that its punching rules are similar to them or are adapted from them to suit particular circumstances.

Any subroutine should be adequately described in a *specification*. This would probably be sketched out before the subroutine is written and all necessary details included afterwards. In an output subroutine it would be necessary to include in the specification a fair amount of detail about the exact printing produced in all cases. The specification of an input subroutine would include the punching rules for the tapes designed to be read by the subroutine and a description of the effects (e.g. loop stops) which occur if the rules are violated. The aim of the specification is to provide full information about the use of the subroutine in such a form that the user does not have to refer to the detailed programme.

† This operation is sometimes known by the picturesque, if inelegant, name of *debugging!*

▼ We described in Sec. 6.3 the way in which an integer may be read in and converted to binary; the orders dealing with a digit may be as follows.

```
(10)  5 41     C(5) = t ⩽ 9
(10)  6 40  }
       6 7 20 }   multiply partial result by 10
       5 7 01     add new digit from X5
```

Here the integer is being built up in $X7$; $X5$ is used to receive the character (t) from the input tape and we suppose that the orders given above are obeyed when $t - 10$ is in $X5$. This is because the character will probably have been identified as a digit by subtracting 10 from its value (via 16) and testing the sign of the result. After obeying the above orders we return to read the next character.

If the number being read in is not a decimal number (e.g. it might be a sum of money in sterling) we can often use an adaption of the above sequence in which the partial sum is not necessarily multiplied by 10 but by the appropriate radix.

In order to read in a decimal fraction we need a different action when a digit is read. The general principle is to read the digits in as if they were the digits of an integer, but we simultaneously build up a power of 10, which starts as $1(=10^0)$ when the decimal point is read and gets multiplied by 10 each time a digit is read. Thus, if we read the number .123 from the tape we build up (a) the integer 123 and simultaneously (b) the integer $1000(=10^3)$. The value of the fraction can be found by a rounded division when the end of the number is reached. For example, suppose initially that $C(0.0) = 0$ and $C(4) = 10$ and that $C(0.1)$ is put equal to 1 when a decimal point is read. The following orders can deal with a digit of the fraction, they are supposed to be entered with $t-10$ in $X5$.

```
(10)  5 41     C(5) = t ⩽ 9
0.0   4 20     multiply partial result by 10
  5   7 01     add new digit
0.0   7 10     replace result in 0.0
0.1   4 20  }
0.1   7 10  }  multiply C(0.1) by 10
```

These orders build up an integer in 0.0 and a power of 10 in 0.1. Alternatively the multiplication by 10 can be done by a process of shifting and adding; this is sometimes desirable since multiplication is a relatively slow process and it is preferable to take less than $3\frac{1}{3}$ milliseconds to handle each digit (we can run the tape-reader at full speed). In the following sequence an integer (x say) is built up in $X7$ and a power of 10 (y say) in $X6$. As before, we enter with t-10 in $X5$.

```
(10)  5 41     C(5) = t ⩽ 9
 (1)  0 54     2y in X6 and 2x in X7
0.0   6 10     2y to 0.0
  7   5 01     2x + t to X5
 (2)  0 54     8y in X6 and 8x in X7
  5   7 01     x' = 10x + t in X7
0.0   6 01     y' = 10y in X6
```

This sequence of orders is quite a good one; it resembles the one used in the Initial Orders. Note that the sign-bit in $X7$ is cleared by this sequence of orders† and does not affect the result (unless overflow occurs). If therefore we set $C(7) = -1.0$ initially (instead of zero) we can test the sign to find out if a digit has been read.

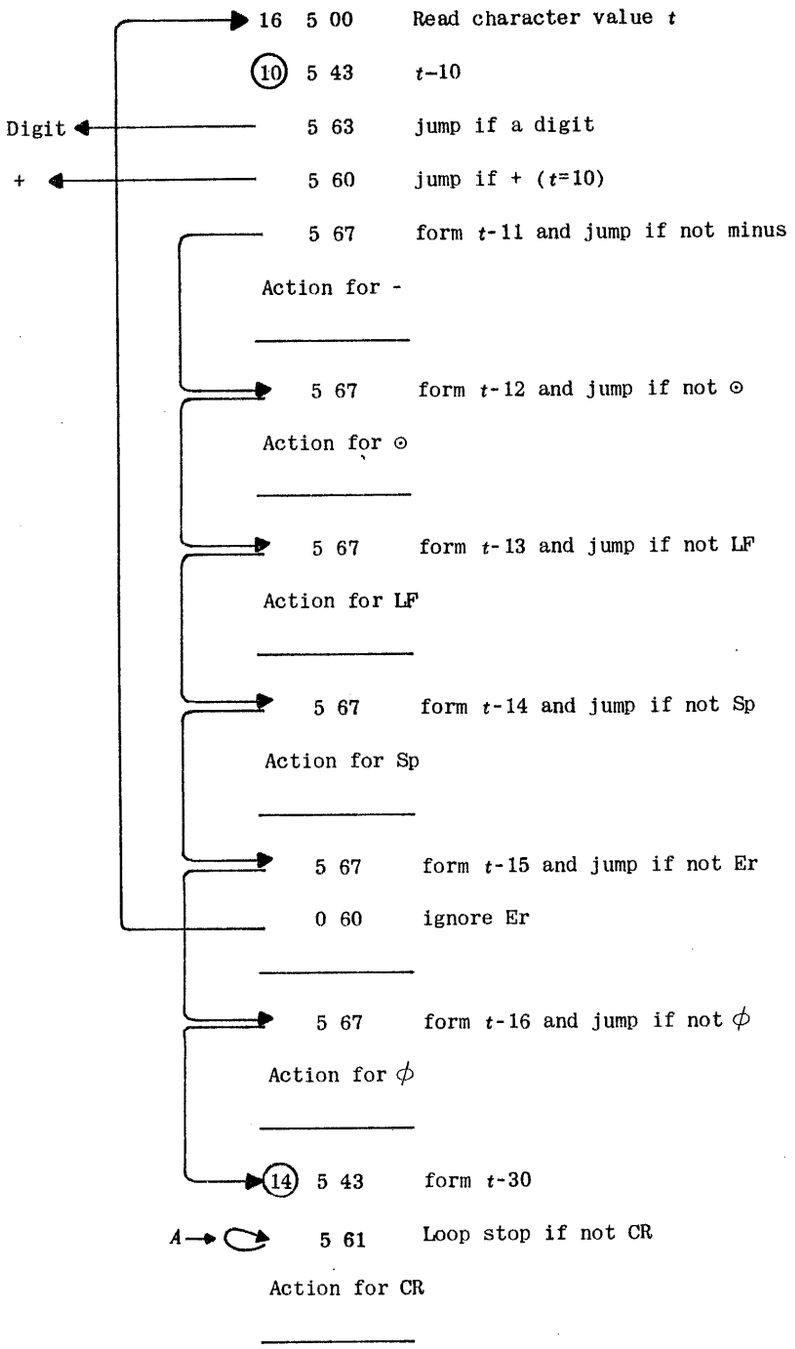
As an illustration of the filtering process to which each character can be subjected as it is read from the tape let us suppose that the following characters are possible, any other indicating a fault (the values via register 16 are written below each character):

digits and	+	−	⊙	LF	Sp	Er	ϕ	CR
0 to 9	10	11	12	13	14	15	16	30

Having read a character we first subtract 10 from it to see if it is a digit, we then test for each possibility in turn in the sequence given. Erase (Er = 15) is to be ignored but each other possibility leads to some special action. A sequence of the following kind may be used.

---

† Because of a property of the double-length shift orders (see Sec. 3.7).



In this sequence we have not specified the action to be taken on each possible character, which is bound to depend on the exact purpose of the subroutine. As a rule in each case there should be some kind of a test that the character is permitted before the action is taken. After this there is in most cases a jump back to the input order at the top to read the next character. This jump can be a conditional one because of an interesting property of this kind of sequence; if the chain of 67-orders is entered at some point with $X5$ clear then ultimately the loop stop A will be encountered. The conditional jump can therefore be an order which causes a jump only if the character is a permitted one (for example, ϕ should not be allowed after a digit has been read).

A number of programming tricks can be used with input and output orders (see also Sec. 5.10). For example, there is the order

```
16  0 11
```

which has the effect of copying a character from the input tape on to the output tape. It reads the number in 16 (i.e. the input tape character), adds $C(0)$, and sends the result to register 16 (i.e. to the output punch). This order is of rather limited usefulness since the character is not available inside the computer for testing. In some programmes it may be desired to print a character or not depending on certain circumstances; use can here be made of modified orders or conditional jumps. Suppose, for example, that we can arrange that $5_p = 0$ if no printing is wanted, or $5_p = 1$ if we have to print CR LF. The following sequence can be used.

```
30  6 40
15  6 10 5
13  6 40
15  6 10 5
```

Sending a number to register 15 (the handswitches) has no effect. Suppose again that $C(5)$ is a small integer which may be positive (or zero) or negative. Then the order

```
10  0 10 5
```

will print ϕ if $C(5)$ is negative (when $5_p = 7$) and will do nothing otherwise. Under the same circumstances the following orders will print the sign of $C(5)$ (as + or −).

```
(22) 6 42 5
16  6 10
```

Here the first order puts −22 into $X6$ if $C(5) \geqslant 0$ and −21 into $X6$ if $C(5) < 0$; these numbers are congruent respectively to 10 and 11 (modulo 32) so that their five least-significant bits are those of the sign characters. The following orders have the same effect.

```
(22) 6 40 5
16  6 12
```

If input via 17 is acceptable the following group of orders can be used to ignore a length of blank tape and Erases.

```
      17  6 02     -t to 6M (in 17 code)
0.0
      0.0 6 60     jump if φ (t=0)
      (31) 6 42 6  -(31-t)=t-31 to 6C
1
      0.0 6 60     jump if Er (t=31)
```

The code-conversion table-look-up technique described in Sec. 6.3 can be of great help in a complicated input scheme where almost any character can occur. There can, for example, be several tables which are used according to the stage which input of an item has reached, or according to the shift.

6.7 The monitors and control panels

There are three separate panels of controls on the computer. Located centrally in front of the control desk (see Plate 2) is an inclined panel known as the *programmers' switches* or, simply, the *control panel*. The keys or switches on this panel are the ones of chief concern to the operator and the programmer; they are shown in Plate 11. Above these is the vertical *monitor panel* on which are visible the faces of the two monitor tubes and a number of associated controls (see Plate 12). The third panel, known as the *engineers' switches*, is mounted under a hinged flap in the centre of the control desk; it has been 'hidden away' because the controls on it are not often used and only a few of them are of interest to the programmer or operator. The labels and numbers of the controls are engraved in two colours, black and white; we are concerned only with those in white since the others are intended for use by the maintenance engineers.

The most important controls are on the *programmers'* switches or control panel (Plate 11). The row of 22 keys along the top of this panel are known as the *handswitches*, and were briefly described in Sec. 2.9. These keys are numbered (from left to right) −2, −1, 0, 1, 2, ..., 19. Of these, the 20 keys numbered 0, 1, 2, ..., 19 can be sensed by the programme; they control the digits $H0$ to $H19$ of special register 15 (key 0 is coloured red and is the sign-digit key of the handswitches). The two keys numbered −2 and −1 are not available in register 15; they are used in some manual operations (see below). The 19 keys 1, 2, ..., 19 are divided into four groups labelled *N, X, F, M* to correspond with the four parts of an order (see Sections 2.6 and 3.12); this has been done chiefly to facilitate manual operations.

The *Start key* and the *Run key* are located centrally under the handswitches. The Run key is described in Sec. 3.9; it can be used to stop the computer, or to cause a single order to be obeyed, or to allow the programme to continue after a 77-stop or an optional stop. The Start key is described in Sec. 4.3; it selects the source of the orders which the computer obeys. In its NORMAL (central) position these orders are selected from the ordinary registers in the computing store in the usual way. In the START (upper) position the start order-pair (see Sec. 4.3) flows into the order-register to call in the Initial Orders. In the MANUAL (lower) position the computer obeys orders set up (in binary) on the hand-switches; this is called *manual operation* and will be described below. Manual operation is used chiefly to test the computer and also to read in the Initial Orders when the computer is being commissioned.

To the left of the Start key are ten small neon lights which indicate the principal kinds of stop, the state of the overflow indicator, the paper tape and magnetic tape busy signals, and an indication when one of the engineers' switches is not in the normal position. The lights are labelled as follows:-

Optional stop

Stop order (77)

Main store or Buffer parity failure

Computing store parity failure

Paper tape

Overflow

Writing with overflow

Unassigned order

Engineers' switch set

Magnetic tape†

The stops are summarized in Table 3.1. The parity failure lights are discussed below.

At the extreme left of the control panel is the key which controls the hooter. If this key is down the hooter will be sounded when most kinds of stop occur; in fact it will sound if the Run key is up on any stop except a loop stop, an input busy stop or an output busy stop (the last can occur only if the punch breaks down).

Next to this is a key labelled PUNCH ON BLOCK TRANSFERS. This key can be used in the development stage of a programme; if it is down the computer operates normally (except for a reduction of speed) but any 72- or 73-order causes the main store block-number concerned to be punched in binary. This useful feature is described in Sec. 7.6.

The next key labelled STOP ON OVERFLOW is described in Sec. 2.11. The key labelled INHIBIT OPTIONAL STOP is described in Sec. 3.9.

In the centre of the control panel are two lights which indicate whether the next order to be obeyed is an *a*-order or a *b*-order. These lights are useful chiefly if the computer is stopped or if manual or single-shot operations are being performed.

The keys labelled MONITOR 2 SELECTOR and DRUM TRIGGER are used with the monitor tubes and are described below.

▼ Before describing the monitors we must first discuss the question of the *parity checks* in Pegasus. We have already described the parity check applied to input and output (see Sec. 6.1). On input this is really a programmed check since all that is done by the computer is to convert the property of having an odd (or even) number of holes into the property of being not greater than (or greater than) 15; the programme has to do the rest. On output reliance is usually placed on visual checking of the printed material; it is normal to use only odd-parity characters for the bulk of the output. Both the main store and the computing store are subject to a parity check. A word is normally regarded as containing 39 binary digits, but there are in fact 42, three of which are not accessible to orders. Of these three bits two are gap digits (normally both zero) and one is a *parity digit*. The parity digit is immediately to the left of the sign digit. When a word is placed in the store its parity digit is adjusted so that the word has an odd number of 1 digits (and, therefore, also an odd number of 0's). When a word is extracted from the store its parity is automatically checked to make sure that it still has an odd number of 1's. Should this check fail the computer stops and one of the stop lights on the control panel is lit; there are separate lights for the main store *(drum parity failure)* and the computing store *(computing store parity failure)*, and in addition the hooter will sound if it is switched on.

The parity of all words read from the main store by a 70- or 72-order is checked before the words are put into the computing store; and conversely, when words are written into the main store by a 71- or 73-order their parity is checked before they are written. When a word enters the mill from an accumulator or one of the ordinary registers its parity is checked and the parity digit removed; when the result of the order is put back into the computing store a new parity digit is automatically computed and inserted into the word. For example, when the order

2.7 3 01

is obeyed the words in 2.7 and *X*3 are checked and the result in *X*3 has a new parity digit manufactured. Note that in the order

2.7 3 00

both *C*(2.7) and *C*(3) enter the mill, even though *C*(3) is not used, and a parity check is applied to both of them. When a word enters the mill its parity digit is replaced by a copy of its sign digit; this is then used to detect overflow.

The parity of an order-pair is also checked when it is extracted from an ordinary register and copied into the order-register to be obeyed.

These checks are highly effective in ensuring the early detection of faulty components or incorrect adjustments in either of the stores. Their use is based on the fact that the most common faults in a storage device result in the alteration of a single binary digit, or else in the replacement of a complete word by a string of 0's or 1's (either of which have even parity). Complete reliance may be placed on the functioning of the stores.

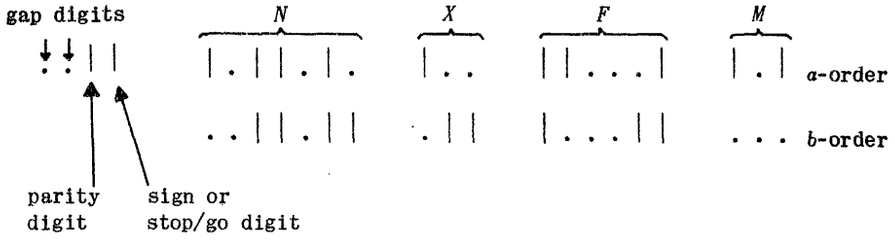
---

† The arrangement of the neon lights is slightly different on Pegasus 1. There are two paper tape busy lights, input busy and output busy (except on machines with magnetic tape- which have a magnetic tape busy light and no output busy), and there is no light to indicate that one of the engineers' switches is set.

It should be noted, however, that accumulators 6 and 7 are not checked in this way. The reason is that these accumulators are part of the complex circuits associated with multiplication, division, double-length shifts and the justify and normalize orders; it was deemed impracticable to apply a parity check to them. A parity digit is, of course, supplied whenever the accumulators are written into the main store with a block-write order.

Whenever access is not required to the main store, the equipment used for the parity check in it is instead used to check continuously the parity of the *address track* on the drum. This track is used chiefly to identify the addresses of words round each track of the drum and so to initiate main store transfers. It is thus possible for the computer to stop and light the *drum parity failure* light even when the programme is not using the main store.

The *left-hand monitor tube* (Plate 12) can be used to display the content of any accumulator or ordinary register (and also a few other things chiefly of interest to the engineers). The two rotary switches below the screen and to the left select the register to be displayed. The upper switch (N BLOCK) selects one of the blocks of ordinary registers in the computing store. When the lower switch is set to one of the numbers marked N UNITS the register with the corresponding position-number is selected and its content displayed. As shown in Plate 12 the display gives the word in ordinary register 1.6. To display the content of an accumulator the lower switch is simply turned to one of the numbers marked X, the setting of the upper switch need not be changed. The word displayed on the screen is arranged as an order-pair with the *a*-order above the *b*-order and the digits conveniently grouped. Each 1 digit appears as a short vertical line and each 0 digit as a dot. To the left of the *a*-order are the sign-digit (or stop/go digit), the parity-digit and the two gap digits. Thus the display shown in Plate 12 gives the following word in *U*1.6.



This represents the go order-pair:-

3.2 4 61 5

(27) 3 43

The parity digit is a 1 to make the total number of 1's odd. Reference should be made to Sec. 3.12 for a detailed description of the binary representation of order-pairs. When reading an order displayed on the monitor screen it is usually best to read the function part first since this determines how the *N*-address is to be read. Note that the parity digit and the gap digits cannot be touched by the programme.

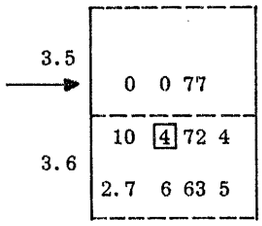
It is possible for the display on the monitor screen to be in seven groups, each of six bits, for use with mixed radix conversion operation. This display is given by the "Character" selection of the three-position engineers' switch labelled 'Scale', which is just above the monitor tube.

The *right-hand monitor tube* can be used to display the content of the order-register, the order-number register or the previous order number only (these can also be inspected on the left tube); the three-way key labelled MONITOR 2 SELECTOR determines which of them is visible. The display is generally similar to that on the other monitor. As shown in Plate 12 the order-number is 2.3, i.e. the computer was about to obey the order-pair in 2.3 when it was stopped. The order-number or previous order-number is displayed in the position corresponding to the *N*-address in the *a*-order; its left-most digit is always 0 and is *not* used to indicate whether an *a*-order or a *b*-order is to be obeyed next; this is shown instead by the two lights in the centre of the control panel. The right-hand monitor tube is of use only when the computer is stopped, otherwise the display is a blur. The content of the order-register depends on whether the computer, when stopped, was about to obey an *a*-order or a *b*-order. The display is as follows in these two states:

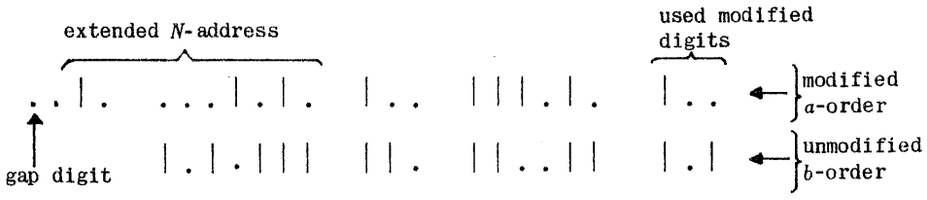
(a) When the computer is about to obey an *a*-order the whole order-pair is in the order-register. The upper line of the display shows the *modified* *a*-order which is about to be obeyed, and the lower line shows the *unmodified* *b*-order.

(b) When the computer is about to obey a *b*-order the *a*-order is no longer in the order-register. The upper line shows the *modified* *b*-order which is about to be obeyed, and the lower line shows the original *unmodified* *b*-order.

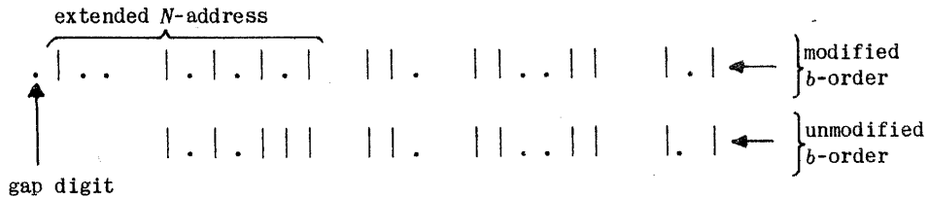
It must be emphasized that it is only in one of these two states that the computer can be stopped. The upper line of the display always shows the next order in the form in which it is about to be obeyed, i.e. modified. The *M*-digits in the order are still visible and are held in the order-register, but they have already been used to select the modifier and have no further effect. The stop/go and parity digits have already served their purpose and are *not* present; they have been replaced, together with one of the gap digits, by the 3-bit extension of the *N*-address which results from the modification (see Sections 5.3 and 5.8). Suppose, for example, that there is the following sequence of orders in the computing store.



When the computer obeys the *b*-order in 3.5 it stops. The next order to be obeyed is the *a*-order in 3.6 so that, if the display is examined when the 77-stop occurs, we shall find that the order-number is 3.6, the next order is an *a*-order, and the content of the order-register will be the modified *a*-order from 3.6 on the upper line and the unmodified *b*-order on the lower. If 4_M = 256.0 the order-register display will therefore appear as follows.



If a single-shot is now given by moving the Run key down to SINGLE SHOT and releasing it, the *a*-order will transfer *B*266 into *U*4. After this the *b*-order will get modified and, in its modified form, will replace the *a*-order. Thus, if 5_M = 63.6 the following display will result.



The lights will show that a *b*-order is to be obeyed next. When obeyed the order is effectively

2.5 6 63

The sequence of events would have been exactly the same if the 77-order in 3.5+ had been omitted and the order-pair in 3.6 had been a stop order-pair. If the *b*-order in 3.5 were another order causing a stop (e.g. an unassigned order, or a 73-order encountered with OVR set) then the machine would stop in the state described above, with the order-number equal to 3.6, an *a*-order next, and the modified *a*-order on the upper line of the order-register display.

Along the top of the monitor panel are three sets of neon lights. The seven lights on the left, labelled EXTERNAL CONDITIONS, show which of the external-conditioning relays are switched on. These are the relays controlled by the 74-order (see Sec. 3.10); in a basic Pegasus installation the right-most light (no. 7) shows which of the two tape-readers is in use. The next six lights, labelled SELECTED TRACK, show which part of the main store is in use. Each track on the drum contains 16 blocks. If a block number be divided by 16 the quotient is the number of the track containing the block and the remainder is the number of the block on the track. For example *B*243 is block number 3 on track 15, since 243 = 15 x 16 +3. The lights always show the number of the track containing the block or word used in the last main store transfer; this is called the selected track. In most programmes these lights are continually flickering as various parts of the main store are referred to. It is sometimes useful to note the condition of these lights when either a programme failure or parity failure occurs. The six neon lights on the right show whether the power is on, whether the drum is turning at the right speed, and so on. *These lights should all be on.* The computer should not be used if any of them are off.

We can arrange to stop the computer at a certain stage in a programme and then use the monitors to examine the contents of various registers, and perhaps work through a small part of the programme one order at a time. This process is called *peeping* and can sometimes give information about why a programme is not working which it would be difficult to get in any other way. On the whole, however, peeping is to be discouraged strongly. It is very slow to go through even a tiny part of a programme and it is very easy to make a mistake when reading the monitor tubes. It is preferable to use the computer itself to print out extra information in a more convenient form, which can then be studied at leisure; one way of doing this is described in Chapter 7. If a programme fails to work when it is first put on the computer then the first thing to do is to hand over the machine to the next user, after noting down the state if a stop has occurred. A considered decision can then be taken about detecting the error, and the necessary alterations made to the programme. There is a tendency among beginners to assume that if they cannot get their programme to work quite quickly then the computer is faulty, which is only seldom so in fact.

The *engineers' switches,* under a hinged flap in the control desk, are only occasionally of use to the programmer. Here are some of the buttons used for switching on the machine, some keys and knobs which should not be touched, and the following sets of keys which are sometimes useful:

(a) keys for inhibiting stops (77-stop, unassigned order, write-with-overflow, and parity failure),

(b) a key which prevents the main store being written into (inhibit drum write), which has one or two uses.

In various places around the computer are some red emergency buttons to shut off the power should an accident occur.

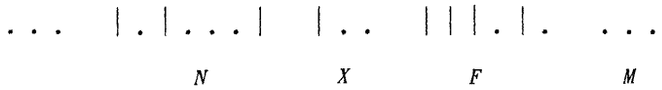
We shall now describe the technique of *manual operation;* it is not often needed and should always be used with great care since it is easy to go wrong. By manual operation we mean the technique of getting the computer to obey orders set up on the handswitches; it should not be confused with manual directives, which concern the Initial Orders; this is quite a different matter and is described in Sec. 7.5.

When the Start key is down, in the position marked MANUAL, the computer takes its orders from the handswitches, instead of from the ordinary registers as it does when the key is at NORMAL. In all manual operation it is advisable to watch the order-register on the right-hand monitor tube and the *next order* lights on the control panel. Orders should normally be obeyed one at a time, i.e. by pressing down the Run key to SINGLE SHOT and releasing it (this operation is called *giving a single shot*). The order to be obeyed next can always be determined by reading the upper line on the monitor tube; the lower line will always show a dummy order† while manual operations are being performed, and it may be disregarded.

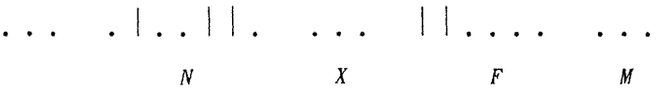
Let us suppose the computer is stopped *when it is about to obey an a-order,* and that we then set up an order on the handswitches (using digits −2, −1 and 0 to extend the *N*-address; there is no stop/go digit in manual operation). If we now move the Start key†† to MANUAL then the order-pair in the order-register will be replaced by a *manual order-pair,* in which the *a*-order is that on the handswitches and the *b*-order is a dummy. Giving a single shot now causes the *a*-order to be obeyed and the dummy *b*-order to be moved up; the next order lights will indicate a *b*-order. When another single shot is given the dummy order is obeyed and a new order-pair enters the order-register; this will be a manual order-pair if the Start key is still at MANUAL, but will be taken from the computing store if the key is at NORMAL. If we leave the Start key at MANUAL we can obey a whole sequence of orders from the handswitches in this way, giving two single shots to obey each one. We must set up the handswitches afresh whenever a *b*-order is shown on the lights as next order; this setting will enter the order-register on the next single shot and will be obeyed on the one after that.#

If we wish we can return to normal operation by returning the Start key to NORMAL after the last manual order has been put into the order-register or when it has just been obeyed. The process of moving the Start key to NORMAL does not of itself affect the order-register but it ensures that the next order-pair to enter the order-register comes from the computing store. Provided there have been no manual jumps the programme will be resumed at the point where it was originally left, since the order-number is preserved during all manual operations (except jumps).

As an illustration of the technique we shall describe the procedure to transfer blocks 81 and 82 of the main store into blocks 4 and 5 of the computing store and them to jump to the order in 4.6+. We must first put the Run key to STOP and we shall suppose that the computer is then about to obey an *a*-order. The next step is to set up the order 81 [4] 72 on the handswitches, thus:-



We then put the Start key to MANUAL and this order (with a dummy *b*-order) enters the order-register, where we can examine it on the right-hand monitor tube. Next we give a single shot and the order is obeyed, copying *B*81 into *U*4. We now set up the order 82 [5] 72 on the handswitches. A single shot puts it into the order-register and another one causes it to be obeyed, copying *B*82 into *U*5. We then set up the order 4.6+ 0 60 on the handswitches, thus:-



When we give another single shot this enters the order-register and we then return the Start key to NORMAL. We can now go to RUN, but it is probably advisable to give one or two single shots and inspect the order-register to make sure that the proper orders are being obeyed.

If the computer is about to obey a *b*-order at the time it is stopped the procedure is similar in general, but one of the single shots at the beginning should be omitted. In this case it is advisable to finish a sequence of manual orders with a jump, as was done above, to make sure that the computer starts obeying normal orders at the right point.

As the reader will probably have realized, the whole procedure is somewhat complicated and there are many opportunities to make mistakes. There is still one trap to be described; this concerns modified manual orders. The need for modified manual orders is rare and they are best avoided, but for those who wish to try them we must point out that a manual order which is put into the order-register by the act of putting the start key to MANUAL is *not* modified (i.e. its *M*-digits are ineffective). In other words the first of a sequence of manual orders is not modified; later orders in the sequence will be put into the order-register when giving a single shot and these will be modified in the usual way.

A *simplified procedure for manual operation* in which *a*-orders and *b*-orders are irrelevant is probably best for general use. Here we simply stop the computer and go through the following three steps for each manual order:

---

† Actually the order 0.0 0 00, not a null order.

†† The Start key should never be moved unless the Run key is at STOP.

The fact that two single shots must be given for each manual order allows us to set up and obey orders such as

15 3 00

which can be put in with one single-shot (*b*-order next) and obeyed with another (*a*-order next), when the handswitch setting can have been changed.

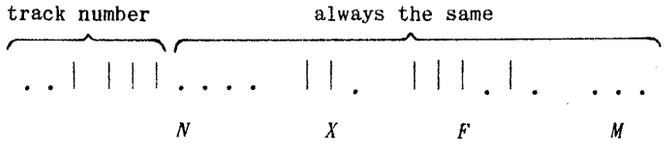
(a) set up the order on the handswitches,
(b) put Start key to MANUAL and then back to NORMAL,
(c) give a single shot.

Here step (b) puts the order into the order-register and step (c) causes it to be obeyed. We repeat the above steps as often as required; we can finish in one of two ways, either we call in the Initial Orders by going to START, or the last manual order must be a jump. Note that if this simplified procedure is used then none of the manual orders may be modified.

As a rule all manual operation is to be avoided, it can lead to waste of much computer time, it is very easy to make a mistake, and when a mistake has been made it is often difficult to determine what happened, as there is no record of the orders obeyed. It is in fact a stage worse than the extensive use of the monitors, which cannot at any rate lead to a wrong result. These remarks apply to any high-speed computer. It is much better to control operations by the input tape where this is at all possible. To this end there should be a keyboard perforator near the computer and reserved for the exclusive use of the operator. We shall describe in the next chapter some of the facilities which are provided in the Initial Orders for changing a programme or stopping it in the middle or for printing out the contents of parts of the main store. Manual operation is usually best left to the maintenance engineers, for whom it is mainly intended.

There is perhaps one application of manual orders which is very useful, reasonably simple to operate, and which yields results which might otherwise be very difficult to obtain. If a programme stops or produces meaningless results it is often helpful to know the content of every register in the computing store. Perhaps the best thing to do is to copy these contents into a convenient place in the main store by obeying a sequence of 73-orders manually (using the simplified procedure); the Initial Orders can then be used to print out the results (see Sec. 7.5).

Manual operation combined with the use of the monitors can allow us to inspect the content of any storage location in the main store should this be necessary. In general it is best to use the Initial Orders to print out such information but there may be exceptional occasions when this is impossible. The simplest technique is to transfer manually the word or block required into the computing store and use the monitors to examine it. If none of the computing store is available we can display the word in the main store by switching the left-hand monitor tube to the position marked SELECTED TRACK. We must then stop the computer and select the appropriate track by a manual 72-order and use the DRUM TRIGGER keys on the control panel. The procedure is best illustrated by an example. Suppose we wish to examine the word in *B*243.5. This word is in position number 5 of block number 3 on track number 15 (243 divided by 16 yields a quotient of 15 and a remainder of 3). We therefore stop the computer and set up 3.5 (i.e. 0011 101) on the DRUM TRIGGER keys, and switch the left-hand monitor to SELECTED TRACK. We now set up on the handswitches an order to read any block of track 15 into *U*6 (a non-existent block), for example the order 240 6 72 which is set up as follows on the handswitches.



This amounts to setting up the track number on the left-hand six keys of the handswitches (i.e. the keys numbered −2, −1, 0, 1, 2, 3) and setting the other keys as shown. We then obey this order by putting the Start key down to MANUAL and back to NORMAL and then giving a single shot. This operation selects track 15 by reading *B*240 harmlessly into *U*6. The left-hand monitor tube now shows the required word. This procedure may be used to monitor any location on the drum, but cannot be used to look at a word in the delay line storage.

The first 16 blocks of the main store (or 32 blocks in some installations) are held on delay lines. In order to examine the contents of this store, the following procedure must be used:-

1. Set the monitor selector switch to SELECTED TRACK.
2. Obey the manual order 0 6 72 to select track 0 or 16 6 72 to select track 1. In these orders tracks 0 and 1 represent the main store blocks 0-15 and (optionally) 16-31 that are held on delay-lines.
3. Set up the required block address in the N-address position on the handswitches.
4. Go to MANUAL and back to NORMAL. (This selects the appropriate block and there is no need to single shot).
5. Select the required word by setting up its position number on the last three drum trigger keys.
6. To strengthen the image on the monitor screen, the three left-hand drum trigger keys should be UP.

In connection with the use of the monitor tubes it is occasionally useful to know that some *unassigned orders* clear certain registers before causing the computer to stop. The effects may be summarized as follows.

07	x'	= 0
17	n'	= 0
30-36	n'	= 0 (these orders are unmodified)
47	x'	= 0
75	no effect	

Chapter 7

The Initial Orders

In this chapter we give a full account of the directives and their principal uses, and also a description of some of the various sections of the Initial Orders, including Binary Input. We also describe some ways of detecting programming errors, and the facilities which have been provided in the computer and the Initial Orders to help detect and correct them.

7.1 General description

The Initial Orders are a programme which permanently occupy 128 blocks starting at *B*896 in the isolated part of the main store.† Their primary purpose is to read in programmes for the computer. They can be regarded as made up of a number of more or less separate routines or sequences. One of these is called *Assembly*, and it has the function of assembling a complete programme from a master-programme and subroutines; it is described in Chapter 8. Another important section is called *Input;* this is responsible for reading in order-pairs and numbers from the tape and placing them in the main store.†† Most of the other sections of the Initial Orders are concerned with reading in directives (see Sec. 4.3), decoding them, and taking the appropriate action.

The *Input* section is operating during nearly all the time that a programme tape is being read in; it is left, in fact, only when a letter shift (λ) is read. The character λ always introduces a directive on the tape. These directives are important features on a tape, since they control the operation of the Initial Orders. So as to provide a record of the directives it is arranged that many of them cause some informative printing as soon as they have been read from the tape; this is known as *optional printing* (or *optional punching*) since it can be suppressed if it is not wanted. For example, the T-directive, which is used to set the Transfer Address (see Sec. 4.3), causes optional printing of a letter T followed by the new value of the Transfer Address; this printing is preceded by CR LF so that it will appear at the beginning of a new line. By inspecting the printing we can find out the value of the Transfer Address at any stage of the input of a programme, and hence determine where the various sections of the programme were stored. Whether optional printing occurs or not is determined by the setting of the red sign-digit key of the handswitches. If this key is up (*H*0 = 0) optional printing occurs; this is usually done with programmes that are under development, since the information given by the printing may be of great value in finding programming errors. If the sign key of the handswitches is down (*H*0 = 1) the optional printing is suppressed; this is normally done only with fully developed programmes.

The T-directive and the others described in Chapter 4 are included in Table 7.1, which also gives the optional printing; each directive in the table has been given a typical address, where appropriate. The optional printing produced by an E-directive occurs before the 77-stop. Note that the printing caused by N and D is not optional; it always occurs, regardless of the setting of *H*0. The optional printing produced by a B-directive includes the new value of the Transfer Address; thus if a tape carrying the two directives

```
T 24.3

B
```

is read in, then there will be optional printing as follows.

```
T 24.3

B 25.0
```

These directives leave the Transfer Address at 25.0 and the relativizer at 25 (see Sec. 4.6). The B-directive on the tape must not include an address, or there will be a loop stop.

An address in a directive is always a main store address; it may refer to a word, as in a T-directive, or to an order, as in an E-directive. The address of a word is usually written in block-and-position form (with a relative block-number if desired), but it may be written in decimal form;

---

† In the Pegasus 1 with small drum, the Initial Orders occupy about 60 blocks in the isolated store, starting from B512.

†† *Input* reads in the programme tape slightly faster than one block per second.

the following are examples.

24.6	word 6 in *B*24
2+.5	word 5 in *B*2+
3+	word 0 in *B*3+
709	word 709 (decimal address)

Directive	Effect	Optional Printing
T 14.3	Set T.A. = 14.3	T 14.3
E 13.5+	Enter programme at *b*-order in *B*13.5, after 77-stop, by copying *B*13 into *U*0, and next three blocks into *U*1,2,3 and then jumping to 0.5+	E 13.5+
J 13.5+	As for E but no 77-stop	J 13.5+
N	Print the following name	none
D	Print the date, increase the serial number by 1 and print it	none
B	Increase T.A. (if necessary) to the start of a block; set relativizer equal to new block-number	B followed by new T.A.
Z	77-stop	none
Y	Optional stop	none

Table 7.1 *Summary of some important directives.*

The address of a single order is usually written in block-and-position form, sometimes with a relative block-number; an *a*-order address (but not a *b*-order address) may be written in decimal form. Any of the above examples can be interpreted as *a*-order addresses; the following are *b*-order addresses.

24.6+	*b*-order in *B*24.6
2+.5+	*b*-order in *B*2+.5

The address in a directive is always punched exactly as it is written. An attempt to use a *b*-order address in any directive requiring the address of a whole word, such as a T-directive, will result in a loop stop. Note that the optional printing always gives, in block-and-position form, the actual (absolute) address used. If, for example, either of the directives

T 1+.3 or T 91

is read when the relativizer is 10, then the optional printing will be

T 11.3.

The Y and Z directives simply cause an optional stop or a 77-stop respectively; this allows the input tape to be changed if desired. Operating the Run key at one of these stops calls in the *Input* section of the Initial Orders to read in more tape, exactly as though the directive had not been there. There is no optional printing with either of these directives. These stops are sometimes useful in the development stages of a programme when the programme tape may be in separate pieces, each of which can be terminated by a Y or a Z. If this technique is used it is advisable for each piece of tape to be headed by a name-sequence (i.e. an N-directive followed by a name); this helps to prevent feeding in the tapes in the wrong order.

The 77-stop when Z is read can be distinguished from that when an E is read by inspecting the order-number on the right-hand monitor tube and the "next order" lights; these show 0.6(*b*) for a Z and 4.0(*a*) for an E.

The Initial Orders are called in to read a programme tape by the *Start* operation: the Run key is set to STOP, the Start key is pushed up to START and allowed to spring back to NORMAL, and the Run key is then raised to the RUN position. This causes the start sequence of the Initial Orders to be entered, which immediately tests the five right-hand keys of the handswitches (keys *H*15 to *H*19); if these are *all up* the operation is called a *Normal Start*. In a Normal Start the Transfer Address is set equal to 2.0 and the relativizer to 2; there is then optional printing of three asterisks, which consequently appear at the top of the computer output, above the date and serial number, unless optional printing is suppressed. These asterisks show that the T.A. and relativizer were set to their starting values†.

† The Reference Address is also set to its starting value by a Normal Start; this subject is discussed in Chapter 8.

When all this has been done the Input section of the Initial Orders is called in to read the tape. If, on the other hand, the five right-hand keys of the handswitches are *all down* when the start process is carried out, then Input is immediately called in without the T.A. and relativizer being touched; there is then, of course, no optional printing of the asterisks. Other settings of these five keys are described below (see Sec. 7.5).

The current value of the Transfer Address is kept in $U5.7_M$ (i.e. in the modifier part of the last of the ordinary registers) and can be inspected on the monitors while the tape is running in. When each word has been read in by *Input* and converted to binary, it is placed in $X1$; the Transfer Address is copied from 5.7 into one of the other accumulators and there is then a modified single-word write order to put the new order-pair or number into the main store; after this the T.A. is increased by unity in 5.7_M. The relativizer is kept in $U5.6$ as an integer (i.e. at the right-hand end of the register). Normally these quantities are changing too fast to be read but it is sometimes useful to examine them when the tape has stopped; their final values are still in these registers when the programme is entered, e.g. by an E-directive.

The Transfer Address is kept in 5.7_M. The sign-bit of 5.7 is always 0; the counter 5.7_C is not disturbed by a T-directive or a B-directive (or their binary equivalents, see Sec. 7.7). This counter may be useful if the Initial Orders are called in as a subroutine (see Sec. 7.2) but it is cleared automatically if Assembly is used (see Ch.8).

7.2 The use of the Initial Orders by a programme

The main purpose of the Initial Orders is to read in programme tapes, on which there may be numbers as well as order-pairs. It follows that the Initial Orders can be used to read in lists of numbers at the same time as the programme tape; because, for example, there may not be enough room for them in the main store. The programme can be so written that when the next list of numbers is required, it calls in an input subroutine to read in the list. Instead of supplying a special input subroutine to do this it is possible for the programme to call in the Initial Orders *as a subroutine*. If this is done then the Initial Orders will be used with two quite different objects; firstly to read in the programme tape and enter the programme, and secondly to read in some numbers under the control of the programme.

When a subroutine has finished its work it normally obeys a link to return control to the master-programme. A special directive has been introduced to signal the end of the numbers being read by the Initial Orders, and to cause a link to be obeyed; this directive is made up of the warning character L (for *link*) - there is no address.

The part of the master-programme which calls in the Initial Orders as a subroutine to read in some numbers should do the following:

(a) set in 5.7_M the main store address where the first number is to be placed, i.e. the Transfer Address,

(b) set a link in $X1$ for return to the master programme when all the numbers have been read in,

(c) transfer $B906$ to $U0$ and jump to 0.0.

This last operation must be done with a modified 72-order. It is convenient to use special register 37 to provide the modifier; this contains the fraction 7/8, which corresponds to the word (896.0, 0) if interpreted as a modifier and a counter.† For example, to read a list of numbers into consecutive locations starting at $B38.7$, the master-programme could contain the following sequence of orders.

```
        0.3  1 00
0.0                    } set T.A. = 38.7
        5.7  1 10
        ------------
        0.4  1 00      set link in X1
1
        37   4 00      set 4_M = 896.0
        ------------
        10  [0] 72 4    B906 to U0
2
0.0     0.0  0 60      jump to 0.0
        ============
        38   - 70 0.
3                      } = (38.7, 0)
        0
        ------------
        2+  [0] 72
4                      } link, obeyed when L is read.
        0.5  0 60
```

We are here using the *subroutine entry* to the Initial Orders, and the following sequence of events will occur. First, the Initial Orders preserve the accumulators by writing them into $B0$; the Input section is then entered to read the tape. As each number is read from the tape it is, as usual, placed in the main store in the location determined by the Transfer Address in 5.7_M, and the latter is then increased by unity. After the last number on the tape an L-directive is punched, the effect of which is to set the accumulators from $B0$, and to obey the link, i.e. the order-pair in $X1$ (and in $B0.1$)††.

† On the small drum Pegasus 1 the 4096 Initial Orders may be entered as a subroutine by transferring $B522$ to $U0$ and jumping to 0.0. In this case the modifier for the 72-order is conveniently provided by special register 33; this contains the fraction ½, or (512.0, 0) as a modifier and counter. It will be seen, therefore, that the piece of programme in the next example may be made to work with the 4096-word store by changing the order in 0.1+ to 33 4 00.

†† The link is obeyed in $U0.3$.

There is no optional printing associated with an L-directive. The list of numbers to be read in in this way must be punched according to the usual rules for the Initial Orders (see Sec. 6.5); they are terminated by the L (punched as CR LF λ L ϕ).

It is not, of course, necessary that the tape should have only numbers on it; since it is being read by the Initial Orders any of the usual directives may be used. For example, it is a good plan to have a name-sequence on the tape before the first number. It is also possible to set the Transfer Address by a T-directive on the tape, instead of setting it in the master-programme. We can also read in order-pairs as well as numbers, though we may then have to set the relativizer (in 5.6 as an integer) as well as the Transfer Address. †

The final value of the Transfer Address is in 5.7_M (and that of the relativizer in 5.6) when the link is obeyed. This may be of some use to the programmer, for example it is easy to check how many numbers have been read in. Apart from the accumulators, the whole of the computing store may get altered when the Initial Orders are called in. It is important that none of the numbers should be put into *B*0.1 if the L-directive is to be used (this is where the link is kept).

The subroutine entry to the Initial Orders, which we have just been describing, is to be carefully distinguished from the start entry at *B*896.0, which is the entry used when we call in the Initial Orders by using the **Start** key. When the subroutine entry is used the accumulators are preserved in *B*0 and Input is then immediately called in to read tape. When the start entry is used the accumulators are not stored away, keys *H*15 to *H*19 of the handswitches are tested (see Sec. 7.1) and, if these are all up, the T.A. and relativizer are set to their starting values, there is optional printing of three asterisks, and only then is Input called in. It is the start entry which is used in the 'Special Factorize' programme described in Sec. 4.4. It is also possible to call in Input directly to read a tape without using the subroutine entry; if this is done the accumulators are *not* copied into *B*0 but the effects are otherwise similar to the subroutine entry. This direct entry to Input can be used if, for example, a link for the L-directive has already been put into *B*0.1 or if there is no L-directive on the tape. To call in Input in this way we must transfer *B*903 to *U*3 and jump to 3.1; this is described below.

The main application of the direct entry to Input arises in *interludes*. An interlude may be defined as a short piece of programme which is read into the store along with a main programme, but which is obeyed as soon as it has been read in (by a J-directive on the tape). It can, for example, perform a small calculation or some piece of organisational work connected with the input of the main programme. After being obeyed, an interlude calls in Input directly and is then no longer required; we usually arrange that it gets overwritten by the next routine on the tape. In order to facilitate this, two effects of the directives E and J have been provided which have not so far been mentioned. Either of these directives, just before entering the programme, sets the accumulators from *B*0 and then sets a special link in *X*1; when this link is obeyed Input is directly called in. This link is *self-modified* (see Sec. 5.10) and consequently must be in *X*1 at the moment it is obeyed; it is the following order-pair.

100 [3] 72 1	
3.1 0 60	

The modifier part of this word is 803.7 (N.B. 803 = 100 x 8 + 3) so that the *a*-order reads *B*903 (N.B. 903 = 803 + 100) into *U*3. An interlude can be usefully regarded as a subroutine of the Initial Orders, which gets called in at the appropriate moment during input of a programme to do some calculation and then obeys the link in *X*1. The tape may be made up as follows.

B	set relativizer
Interlude tape	
T 0+	reset T.A. so that interlude is overwritten.
J 0+	enter interlude.

The fact that an E-directive sets the accumulators (except for *X*1) from *B*0 on entry to a programme may be useful; we simply place in *B*0 the words we want to have in the accumulators when the programme is entered. This can be done as a part of the input process of the programme. This procedure is, however, *not recommended* because most programmes use *B*0 as temporary storage (for the accumulators) while they are running; the original words in *B*0 consequently get overwritten and it is then impossible to restart the programme by a simple E-directive. To restart the programme we would have to restore the original contents of *B*0 first; this would be a serious handicap in the development stages of the programme. It is always advisable to arrange that a programme may be restarted by an E- or J-directive, if this is reasonably possible.

7.3 The detection and correction of blunders

We use the word *blunder* as meaning a mistake in writing a programme. It is useful to have a special word for this kind of mistake in order to distinguish it from mistakes in operating procedure, rounding errors, truncation and other errors due to the use of approximate formulae, incorrect data, faults in the computer and punching errors in the preparation of tapes. Blunders arise for a variety of reasons; a wrong order may be written inadvertently, a short sequence of orders may not do exactly what was intended, the problem may have been misunderstood, the same register or storage location may accidentally be used simultaneously for two different purposes, some feature of the computer or a subroutine may have been misunderstood, and so on. When a programme is first put on the computer it will probably contain a

† If the tape to be read in has any binary-punched information on it (see Sec. 7.7) it is essential that the word in *U*5.5 should not be negative.

number of blunders and these will all have to be discovered and corrected: in a large programme this may be quite difficult. However, it is usually worth trying a programme out once first - after all, it may be free of blunders! The process of discovering blunders and correcting them is called *developing* the programme.

The symptoms produced by blunders are as varied as the blunders themselves; some will result in an unexpected or irregular layout of the results, or perhaps all the numbers printed may be zero, or there may be no results at all, or the programme may start to read tape or punch blank tape indefinitely, or it may stop or go into a loop of orders and stay there. These are all fairly obvious symptoms, and are usually not too difficult to correct. The most difficult blunders to put right usually produce results which differ only slightly from the correct ones. A carefully worked hand solution of one or two cases is a great help in developing a programme. Any differences between the results of the programme and those of the hand computation can be investigated in detail; not infrequently the hand computation is the one shown to be in error, but it is encouraging to get results from the programme which are known to be right.

When a new or partly developed programme is put on to the computer it may produce some results or it may stop in some way. A careful note should be made, with the help of the monitors, of where the programme stopped (if it did) and whether OVR is set; it may also be useful to print out the contents of the computing store (see below, and Sec. 7.5). This should all be done quickly and then the output tape or the printed results should be taken away for study. This makes the computer available for the next user with as little wasted time as possible. No attempt should be made to diagnose possibly complicated and misleading situations manually on the computer; in the event of the programme not working the machine should be vacated at the earliest possible moment. It is advisable, however, that all programme sheets and other relevant papers should be readily available at the computer during the development of a programme.

Quite often a preliminary run will disclose a number of small blunders, and perhaps show that not all is well. The small blunders or slips can probably be put right quite easily, by altering the programme tape in some way with the tape-editing equipment or by using X-directives as described below. When preparing for a further run of the programme it may be useful to alter the programme slightly so that it stops at a certain point. When this point is reached in running the programme it may be desirable to note, for example, the contents of certain accumulators or perhaps to examine or print out intermediate [illegible] which can be compared later with a hand computation. To this end the following directives have been provided in the Initial Orders:

S to make a specified order-pair into a *stop order-pair*,

G to make a specified order-pair into a *go order-pair*,

F to print *fractions*,

I to print *integers*,

†K to print contents of a location as 6-bit characters,

P to print parts of a *programme* (as orders),

X to *correct* orders in a programme.

All these directives contain addresses which, as always, refer to the *main store*.

Suppose, for example, that we wish to make the programme stop when the order-pair in 19.6 is encountered. We can take the programme tape and insert (by splicing, see Sec. 6.4), just before the E-directive at the end, a short length of tape bearing the directive S 19.6. When read by the Initial Orders this causes the order-pair in *B*19.6 to be made into a stop order-pair, regardless of what it was

Intermediate results obtained by a programme can, if desired, be examined on the left-hand monitor tube. We described in Sec. 6.7 how to use this tube to display words in the main store; this procedure is, however, rather inconvenient, involving as it does, the setting up of addresses in binary on keys and the reading of binary information from the monitor tube. This *peeping* is generally discouraged as it is not only inconvenient, but also time-wasting and does not provide any automatically printed record of the operation or its results. It is preferable to punch a short tape carrying certain directives and to read this in with the Initial Orders when information from the main store is wanted.

The F- and I-directives cause the Initial Orders to print words in the main store as fractions or integers, respectively. For example the directive F 83.1 causes the word in *B*83.1 to be interpreted as a fraction and printed out on a new line; the fraction is printed, unrounded, to 11 decimal places and is preceded by its sign. In a similar way the directive I 83.1 causes the word in *B*83.1 to be interpreted as an integer and printed on a new line; left-hand zeros are suppressed (i.e. replaced by spaces) and the sign appears immediately before the first digit to be printed. It is, of course, up to the programmer to decide how any particular word should be interpreted and to use the appropriate directive. When the printing is finished the Input section of the Initial Orders is called in as usual to read more tape. It is quite possible, and in fact not unusual, to have several F- or I-directives on the tape. If desired, either of these directives may be followed by *two* addresses separated by a dash (i.e. a minus sign), in which case all the words from the first specified location to the second will be printed out (the second address must not be less than the first). For example, the directive

F 43.6 - 44.2

will cause the five words in the locations *B*43.6 to *B*44.2 to be interpreted and printed as fractions, each on a line to itself. The first number is preceded by CR LF and each number is followed by CR LF. An extra line feed is printed between blocks for convenience in reading the results. There is a certain amount of optional printing as follows:

(a) the printing begins with F or I (on a line to itself), and

(b) each number is preceded by its main store address.

Note that the printing of the numbers always occurs, whereas that of the addresses (and the warning character) is optional. If optional printing is suppressed the output tape is suitable for reinput by the Initial Orders.† In all cases the numbers are printed in a single column, the right-hand end of the integers being vertically aligned. The F- and I-directives can even be used to print the final results of a programme, but this should be regarded as an expedient for temporary use. The technique is to arrange that the programme calls in the Initial Orders when it has finished (or it can stop and the Initial Orders can be called in manually); we must put a tape bearing the required directives into the tape-reader (these could be punched after the E directive on the main programme tape).

The P-directive is generally similar to the F- and I-directives but it prints part of the *programme* in the form of orders. The address or addresses in a P-directive are those of individual orders. For example, the directive P 2.2+ will cause the *b*-order in *B*2.2 to be printed, and the directive P 12.6+ - 13.0 will cause printing of all the orders from the *b*-order in *B*12.6 to the *a*-order in *B*13.0, inclusive. Each order occupies a line to itself, the first order being preceded by CR LF and every order being followed by CR LF, with an extra LF between blocks. The optional printing is similar to that produced by F or I:

(a) the printing begins with P (on a line to itself), and

(b) each order is preceded by its main store address.

The individual orders are printed according to much the same rules as they are punched for input. The *N*-address, for example, is printed in the way it is usually written on programme sheets††; this is followed by the *X*-address and function and by the modifier-address (or Sp if this is zero). The output tape is in fact suitable for reinput by the Initial Orders, provided

(a) optional printing is suppressed, and

(b) only order-pairs are punched.

Note that relative block-numbers will be punched as they are stored, i.e. in their absolute form; this is generally desirable as we get information always about the actual orders in the programme as obeyed. The *a*-order of a stop order-pair is followed by a full stop. As with F and I the Input section of the Initial Orders is called in to read more tape as soon as the printing is finished. The P-directive is very useful in telling us the details of the programme as obeyed by the computer.

As an illustration suppose that when the programme stops we wish to print the fraction in 3225 (decimal address), the integers in *B*29.7 to *B*30.2, and the orders in *B*9.7 to *B*10.1+. We first prepare the following *steering tape:*

F 3225

I 29.7 - 30.2

P 9.7 - 10.1+

Z

---

† It will have to be provided with a length of blank tape at each end, which can be obtained by pressing the run-out button on the output punch. Exceptionally the integer -2^{38} cannot be read in (see Sec. 6.5).

†† There are a few small and unimportant departures from the usual punching rules described in Sec. 6.5. The address in a single-word transfer order is always printed in block-and-position form. A decimal *N*-address (e.g. that of a special register or in an order of groups 4, 5 or 7) is aligned on the right instead of on the left, i.e. such an address is preceded by enough spaces to make its width equal to 3 and then followed by a single space.

This tape is punched as follows (see Sec. 6.5):-

Blank tape CR LF

λF∅Sp3225 CR LF

Blank tape CR LF

λI∅Sp29.7Sp - Sp 30.2 CR LF

Blank tape CR LF

λP∅Sp Sp 9.7Sp - Sp10.1+ CR LF

Blank tape CR LF

λZ∅ CR LF

Blank tape, erases.

When the programme stops we place this steering tape in the main tape-reader, clear the handswitches, put *H0* (the red, sign-digit key) down if we wish to suppress optional printing, and then call in the Initial Orders (by a Normal Start) to read in the tape. The computer then produces an output tape which yields a print-out similar to those shown in Fig. 7.1; here the left-hand column shows the printing with optional printing; the right-hand column gives the result if optional printing is suppressed. At the top is a print-out of the steering tape.

```
F 3225

I 29.7 - 30.2

P  9.7 - 10.1+

Z
******
```

```
F
403.1 +0.12345678900

I
 29.7 +123456789012

 30.0        -987654
 30.1           +992
 30.2             +0

P
  9.7  10 172
  9.7+1.2+060

 10.0 3.0 4012.
 10.0+4.2 500
 10.1    4 740
10.1+  5 720
```

```
+0.12345678900

+123456789012

        -987654
           +992
             +0

 10 172
1.2+060

3.0 4012.
4.2 500
  4 740
  5 720
```

Fig. 7.1 *Typical output from F-, I-, and P-directives.*

The K-directive causes the computer to print the contents of the specified address (or addresses) as 6-bit characters. It operates in the same way as the F- and I- directives. The content of a location is taken to consist of six 6-bit characters laid out as follows:-

Number of bits	3	6	6	6	6	6	6
Contents	S	C_1	C_2	C_3	C_4	C_5	C_6

where S indicates spare bits
C_i is the i-th character printed.

As far as possible the teleprinter characters printed by the K-directive correspond to the letters or numbers which would be printed, using a standard 6-bit code, if the characters were sent to a line-printer. Letter shift and figure shift characters may appear on the paper tape as well as the 6 printing characters, but the word always terminates in figure shift.

The code is given in Table 7.2. Characters not listed are printed on the teleprinter as * (asterisk). This code is the same as that used in the Ferranti Converter (Magnetic Tape ⟷ Printer or Cards) for Pegasus and Mercury, except that on the converter the following four characters are different:

000000 010000 100000 110000

If desired, the K-directive at a given installation could be set permanently to use a different code.

Figure 7.2 shows the print-out of a steering tape to print the contents of two locations with a P-directive and a K-directive, followed by the output: (a) with optional printing in the left-hand column, (b) without optional printing in the right-hand column.

```
P 10.3 - 10.4+

K 10.3 - 10.4

Z
******
```

```
P
 10.3    8 4173.
 10.3+   1 0003
 10.4    4 0002.
 10.4+  72 075

K 10.3 A3X103
  10.4 805879
```

```
8 4173.
1 0003
4 0002.
72 075
A3X103
805879
```

Fig. 7.2 *Typical output from a K-directive*

If a blunder is discovered in a programme it is often possible to put it right by use of the tape-editing equipment. The simplest kind of blunder may be called a slip; for example, writing the wrong address in a jump order, specifying the wrong accumulator in an arithmetical order, or writing the wrong function (a common slip is to write 00 instead of 40, e.g. to write (10) 7 00 instead of (10) 7 40). Usually a slip can be put right by simply changing a single order. A slightly more complicated blunder may involve the rewriting of a part of a block because the orders actually used may not do quite what was intended. This may be troublesome because room may have to be found for one or two extra orders; considerable ingenuity and the use of conventional tricks (see Sec. 5.10) may be needed. If we insert a few dummy orders in each block when writing the programme, these can be a great help later. Often a small sequence of orders may have to be moved up or down bodily so as to provide the space to insert the extra orders; this will usually require changes in the addresses in jump orders. If room cannot be found for any extra orders that are needed we may be able to *patch* the programme, i.e. replace an order-pair by a block-read and a jump order to bring in and enter a spare block from the main store, in which can be placed any orders needed. If a portion of a programme has to be rewritten it is possible to punch new versions of the blocks containing the blunders and to splice them into the programme tape in place of the original blocks. Short lengths of blank tape left between the blocks can be a help here, and we may be able to reproduce part of the new blocks by reproducing from the original tape. It is undesirable, however, to tamper extensively with a long programme tape; it may not be easy to find the right section of tape, and one will have to read the tape-characters to identify the blocks (actually not at all difficult after a little practice). It is thus easy to introduce extra errors, particularly as there may be a fair amount of unwinding and re-spooling to do.

Usually the best way to correct a small blunder is to use a small tape carrying a correction sequence which can be fed in after the main programme tape and has the effect of correcting the programme in the main store. In other words we read the original wrong programme into the computer and then put it right. A number in a programme is simply altered by a small tape headed by a T-directive. For example, a tape such as the following may be run in by a Normal Start operation after the main programme tape has been read in (but not entered).

```
T 17.2

+ 443

E 2.0
```

The best way of correcting orders is to use the special X-directive which has been provided for this purpose; with it we can easily change single orders or small groups of orders. An X-directive containing the address of a single order is followed immediately on the tape by an order, which will replace the one whose address is specified. For example, suppose we wish to change the *b*-order in *B*28.4 from 0.3 6 62 to 1.3 6 62; we can use the following directive:

```
X 28.4+

1.3 6 62
```

We can alternatively have two addresses after the X, when a number of orders read from the tape will replace those in the main store between the specified locations (inclusive). For example the following

directive will replace orders from the *b*-order in *B*8.6 to the *a*-order in *B*9.0 by the orders punched in the tape:

```
X 8.6+ - 9.0

0.4  1 20

  9  0 72

0.0  0 60

1.7+ 6 63
```

Character	Card Code U.C.	Card Code L.C.	6 - Bit Character	Teleprinter Character
Zero	0	-	000000	0
1	-	1	000001	1
2	-	2	000010	2
3	-	3	000011	3
4	-	4	000100	4
5	-	5	000101	5
6	-	6	000110	6
7	-	7	000111	7
8	-	8	001000	8
9	-	9	001001	9
Word Inhibit	Not Available		001010	)
Space	-	-	010000	Erase
A	10	1	010001	A
B	10	2	010010	B
C	10	3	010011	C
D	10	4	010100	D
E	10	5	010101	E
F	10	6	010110	F
G	10	7	010111	G
H	10	8	011000	H
I	10	9	011001	I
Block Inhibit	Not available		011010	(
Full Stop	10	-	100000	,
J	11	1	100001	J
K	11	2	100010	K
L	11	3	100011	L
M	11	4	100100	M
N	11	5	100101	N
O	11	6	100110	O
P	11	7	100111	P
Q	11	8	101000	Q
R	11	9	101001	R
Paper Throw	Not available		101010	≥
Hyphen	11	-	110000	-
S	0	1	110001	S
T	0	2	110010	T
U	0	3	110011	U
V	0	4	110100	V
W	0	5	110101	W
X	0	6	110110	X
Y	0	7	110111	Y
Z	0	8	111000	Z
Minus	0	9	111001	-
Line Feed	Not available		111010	>

Table 7.2 *6-Bit Character Code for K-Directive*

Each order is punched in the usual way (actually according to the rules for punching *a*-orders) so that this directive would be punched as follows:

```
Blank tape CR LF
λXφ Sp 8.6+Sp - Sp 9.0 CR LF
0.4 Sp 120 CR LF
9 Sp Sp Sp 072 CR LF
0.0 Sp 060 CR LF
1.7+663 CR LF
Blank tape
```

When an X-directive has finished making corrections it returns to the Input section of the Initial Orders to read in more tape; the Transfer Address will not have been altered. The optional printing associated with an X-directive is very full and informative. First there is a letter X on a line by itself; then there is a line of print for each order replaced, consisting of the address of the order, the old order and the new order. The first line is preceded by CR LF, and each line is followed by CR LF, with an extra LF between blocks. This is illustrated in Fig. 7.3; on the left is a print-out of the X-directive given above, and on the right is an example of the computer output with optional printing.

```
X 8.6+ - 9.0
0.4 I20
9   072
0.0 060
1.7+663
```

```
X
  8.6+0.4 I21   0.4 I20
  8.7   9 I72     9 072
  8.7+0.0+060   0.0 060

  9.0 1.7+662   1.7+663
```

Fig. 7.3 *Print-out of an X-directive and the optional printing produced by it.*

If optional printing is suppressed (*H*0 = 1) there is no printing at all, and the relevant sections of the Initial Orders have been made as fast as possible. This means that any corrections made by X-sequences can be spliced into the programme tape and left there in the final form of the tape; they will be read in at high speed. This is better than attempting to edit the programme tape when development is finished because of the possibility of introducing errors; any edited tape must always be carefully tested again.

In an X-directive the orders are read in one by one and the corrections made; this is the only way in which individual orders (instead of order-pairs) can be read in by the Initial Orders. An *a*-order can, if desired, be marked with a full stop to make the order-pair a stop order-pair; a stop punched after a *b*-order will have no effect. It is advisable to punch a short length of blank tape after a complete X-directive; this will help to prevent errors due to punching the wrong number of orders and assists in splicing the tape. The optional printing is very valuable since it provides a complete record of every order which was changed. Furthermore, it shows the original form of the order as well as its changed form, so that it is easy to undo a correction at a later stage and also to make sure that any correction was put into the right place.

Relative addresses may be used in X-directives but great care should be taken that the relativizer is correct. For example, suppose we wish to alter the *b*-order in *B*1+.3 of a subroutine starting in *B*12 so that it reads 2+ 1 72. We can put the following directive *at the end of the subroutine tape.*

```
X1+.3+

 2+ 1 72
```

Alternatively we could put the following directives at the end of the programme tape.

```
T 12.0 }
       } set relativizer = 12.
B      }

X 1+.3+

2+ 1 72
```

A sequence of this kind will, of course, change the Transfer Address.

Further help in detecting blunders is provided by the computer itself by means of the facility of punching on block-transfers. This is best used in conjunction with another directive, the ?-directive (read as "query-directive"), which is not normally punched on tape. The process is described in Sec. 7.6 below.

7.4 Summary of the directives on tape

Table 7.3 summarizes the effects of the various directives when they are read from the tape by the Initial Orders. In the column headed *addresses* indication is given of the kind and number of addresses which normally follow the warning character. A single whole-word address is generally written as *a*, and if two such addresses are permissible they are written a_1 - a_2. The addresses of individual orders are indicated as *a*(+) or as a_1(+) - a_2(+). For the sake of completeness a number of directives are included which are described in later sections of the book. The details given in the table are of necessity abbreviated, and the reader should refer to the sections indicated in the *reference* column for a fuller description.

The directives E, J and L cause the Initial Orders to be left and a programme to be entered. All the other directives (except ? and A0) cause the Input section of the Initial Orders to be called in to read more tape when they have done their work.

The directives F, G, I, S and T cause a loop stop (after optional printing) if they are followed (erroneously) by a *b*-order address, e.g. T 17.3+. A B-directive has similar effects if it is read with a negative word in 5.7. These and other loop stops in the Initial Orders are included in Table 7.7 in Sec. 7.8.

The ϕ-directive has been included in Table 7.3; this directive is punched $\lambda\phi\phi$ and can consequently be punched in a length of blank tape simply as λ. The effect is the same as Z, i.e. a 77-stop. This directive can conveniently be inserted manually with a unipunch (see Sec. 6.4) in any stretch of blank tape where a stop is desired.

The character λ and all other letter shift characters not mentioned in the Table 7.3 are unassigned directives, they cause a loop stop in 0.5 after optional printing (except that there is no optional printing for λ).

If desired, an E- or J-directive may be followed by *two* addresses; the second address is disregarded but will be found in 5.4_M when the programme is entered; this is occasionally useful.

	Addresses	Effects	References	Optional Printing
A0	none	Enter Binary Input.	7.7	none
A1	none	Prepare to use Assembly.	8.2	A1
A2	none	Change to second tape-reader.	8.2	A2
A3	none	Process the programme.	8.2	A3 and details of routines
A4	a or a_1 - a_2	Punch out the word in a or the words in a_1 to a_2 in binary.	7.7	none
†A5	none	Prepare to read MAGLIB.		A5
†A6	none	Search for required routine on MAGLIB.		A6
B	none	Start new block with T.A. and set relativizer.	4.6, 7.1	B and new T.A.
C	r	Set relativizer to that of routine number r.	8.8	C r
D	none	Print date and serial number after increasing serial number by 1.	4.3	none
E	$a(+)$	Enter programme at $a(+)$ in *U*0 after 77-stop.	4.3, 4.6, 7.2	E $a(+)$
F	a or a_1 - a_2	Print fraction in a or fractions in a_1 to a_2.	7.3	F, addresses
G	a	Make word in a a go order-pair.	7.3	G a
I	a or a_1 - a_2	Print integer in a or integers in a_1 to a_2.	7.3	I, addresses
J	$a(+)$	Jump to $a(+)$ in *U*0.	4.3, 4.6, 7.2	J $a(+)$
K	a or a_1 - a_2	Print 6-bit characters in location a or locations a_1 to a_2.		K, addresses
L	none	Transfer *B*0 to accumulators, then obey link in *X*1.	7.2	none
N	none	Print name of tape.	4.3, 6.5	none
P	$a(+)$ or $a_1(+)$ - $a_2(+)$	Print order in $a(+)$ or orders in $a_1(+)$ to $a_2(+)$.	7.3	P, addresses
†Q	n	Call for complete programme, number n, on magnetic tape.		
R	none	Reference. Add T.A. to next order-pair and store in tag-list and not in T.A.	8.2	none
S	a	Make word in a a stop order-pair.	7.3	S a
T	a	Set T.A. equal to a.	4.3, 4.6, 7.1	T a
X	$a(+)$ or $a_1(+)$ - $a_2(+)$	Replace order in $a(+)$ or orders in $a_1(+)$ to $a_2(+)$ by orders read from tape.	7.3	X, addresses old and new orders
Y	none	Optional stop.	7.1	none
Z	none	77-stop.	4.4, 7.1	none
?	none	Convert block-transfer tape.	7.3, 7.6	?
ϕ	none	77-stop.	7.4	none
λ		Loop stop.	7.4	none
All others		Loop stop.	7.4	Letter

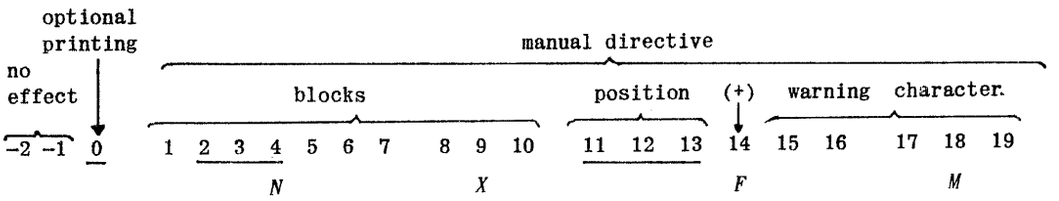
Table 7.3 *Summary of the directives on tape.*

† A5, A6 and Q are only available on machines which have magnetic tape; in the absence of magnetic tape these three directives lead to a loop stop.

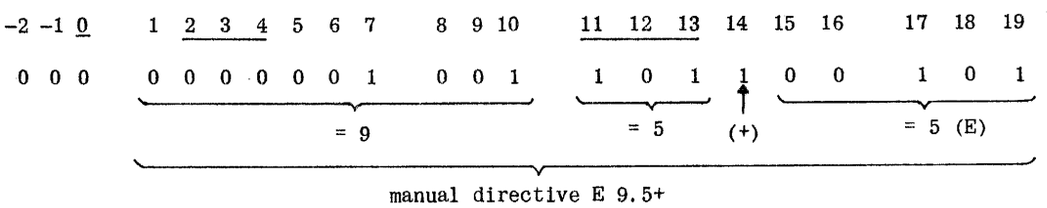
7.5 Manual directives

When the Initial Orders are called in by use of the Start key the five right-hand keys of the handswitches (*H*15 to *H*19) are tested, and their setting determines subsequent action. Usually these keys are all up, a setting we shall describe as ϕ, and we enter Input to read tape after setting the Transfer Address and the relativizer (and the Reference Address) to their starting values; this is the Normal Start, which has optional printing of three asterisks. If the five keys are all down, a setting we describe as Er, we enter Input immediately, and there is no optional printing.

If the five right-hand keys of the handswitches are neither all up (ϕ) nor all down (Er) when the start operation is performed, then they are treated as if they were a *warning character* read from the tape (via register 17 - the direct input). If an address is required then it is taken from keys *H*1 to *H*14; in this address *H*1 to *H*10 give the block-number, *H*11 to *H*13 give the position-number, and *H*14 is used as an *a*/*b* digit in single-order addresses (*H*14 = 1 indicates a *b*-order address). *H*0 is used exclusively to control the optional printing. The keys *H*(-1) and *H*(-2) have no effect. The setting of the handswitches thus determines a directive with a single address; it may be called a *manual directive*. The way the keys are used may be shown diagrammatically as follows.



In this diagram the numbers which are underlined correspond to red keys. As an illustration suppose that the setting of the handswitches is as follows when the Start key is operated (keys marked 1 are down).



The five right-hand keys represent the integer 5 in binary; this corresponds to E, the fifth letter of the alphabet. The address is 9.5+. The effect of the operation is therefore exactly the same as if the directive E 9.5+ had been read from the input tape, and we get this printed, before the 77-stop associated with an E-directive, because optional printing is not suppressed (*H*0 = 0). In this way we can enter a programme which is already in the store. Note that the address which is set up is always an absolute one, relative addresses cannot be used in manual directives.

With few exceptions (principally F, I, K, P, Q and X) any manual directives have almost exactly the same effects as the corresponding tape directives. The main difference lies in what happens when the appropriate action has been completed; most tape directives then call in the Input section of the Initial Orders to read tape, whereas most manual directives cause a 77-stop, after which an operation known as *restart* occurs and a new manual directive is taken from the handswitches.

For example, suppose there is a programme in the store and we wish to make the order-pairs in *B*3.6 and *B*10.2 into stop order-pairs, and then enter the programme at *B*2.0. The procedure is as follows.

(a) Run key to STOP.
(b) Set up S 3.6 on handswitches (S is the 19th letter so the five right-hand keys are set to 10011; of the other keys *H*9 to *H*12 are down).
(c) Start key to START and release it.
(d) Run key to RUN (the manual directive is carried out and there is a 77-stop).
(e) Change *address* in manual directive to 10.2 (no change in five right-hand keys; of the other keys *H*7, 9 and 12 are down).
(f) Run key to STOP, then to RUN (new manual directive S 10.2, then a 77-stop).
(g) Set up E 2.0 on handswitches (key *H*9 down and five right-hand keys set to 00101).
(h) Run key to STOP, then to RUN (77-stop on E-directive).
(i) Clear handswitches or set them as required by the programme and operate Run key to enter the programme.

Note that the Start key need be operated only once (step (c) above), but the sequence of events would be exactly the same if it were operated again after steps (e) and (g). This is because the restart operation which occurs after the 77-stops in steps (d) and (f) is almost identical with the start operation at the beginning of the Initial Orders.

We can now define more precisely the effects of the *start operation*, which is usually the result of operating the Start key but may occur if a programme uses the *start* entry (*B*896 to *U*0, jump to 0.0). The handswitches are immediately read. If the manual directive is ϕ (five right-hand keys all up) then the Transfer Address, relativizer and Reference Address are set to their starting values, there is optional printing of three asterisks, and *Input* is called in to read tape. If the manual directive is Er (five keys all down) then *Input* is immediately called in with no other effect. Any other setting of the five right-hand keys causes the appropriate manual directive to be carried out.

After most manual directives there is a 77-stop; when we operate the Run key to allow the computer to continue the *restart* operation occurs. The handswitches are immediately read and subsequent effects are the same as those in the start operation, *except* that the manual directive ϕ has exactly the same effects as Er, i.e. Input is called in without the Transfer Address, etc. being

altered. This difference has been introduced to facilitate the use of manual T- and B-directives. For example, to set the Transfer Address to 6.0 and then read in a tape the procedure is as follows.

(a) Run key to STOP.
(b) Set up the manual directive T 6.0 (T is the 20th letter so the five right-hand keys are 10100; of the other keys *H*8 and 9 are down).
(c) Start key to START and then release it.
(d) Run key to RUN (77-stop after T.A. is set to 6.0).
(e) Clear handswitches (except possibly for *H*0).
(f) Run key to STOP, then to RUN.

The last operation causes the tape to be read without any change being made in the Transfer Address. Here we must *not* operate the Start key after step (e) or we shall do a Normal Start and the T.A. will be reset to 2.0. A manual B-directive is similarly used; here the address set on keys *H*1 to *H*14 is entirely disregarded, as in any manual directive which does not need an address (e.g. D or L).

As was mentioned above, most manual directives lead to a 77-stop and the restart operation when they have been carried out. The principle exceptions (apart from ϕ and Er) are, of course, E, J and L, which cause a programme to be entered. The manual R-directive is entirely without effect; it simply leads directly to the 77-stop and restart: the R-directive read from tape is used with the Assembly Routine and is described in Chapter 8. The manual N-directive can be used to print a name from the input tape; when the end of the name is reached there is a 77-stop and restart. This and the manual directives Y and Z are of little value.

The manual directives F, I, P and X are very useful; the procedure for using them is rather different from that for most manual directives. Suppose, for example, that we set up the manual directive F 16.4 and do a start operation. The fraction in *B*16.4 will be printed out as usual and there is then an *optional stop*. If we operate the Run key at this stage, *without changing the setting of the handswitches*, then the fraction in the *next* location *B*16.5 will be printed, and again there is an optional stop. This procedure may be repeated indefinitely. If optional stops are inhibited the Initial Orders will continue to print fractions from consecutive locations in the main store until stopped, for example by raising the key for inhibiting optional stops. The process continues in fact until the setting of the handswitches is *changed* at one of the optional stops (this can only be done if optional stops are not inhibited); when this happens the restart operation immediately occurs and a new manual directive is taken from the handswitches. The manual I- and K-directives can be used in exactly the same way for printing integers or six-bit characters respectively.

The following procedure can be used for printing numbers in randomly placed locations in the main store.

(a) Run key to STOP.
(b) Optional stop inhibit key UP (i.e. stops).
(c) Set up F (or I) and first address on handswitches.
(d) Start key to START and release it.
(e) Run key to RUN (first number is printed, then optional stop).
(f) Change *address* on handswitches to that of the next number.
(g) Run key to STOP, then up to RUN (next number is printed, then optional stop).
(h) Change *address* on handswitches to that of the next number.
(i) Run key to STOP, then up to RUN (next number is printed, then optional stop).

The last two steps are repeated as many times as necessary. When the last number has been printed we can set up a new manual directive and operate the Run key. Note that at any time when the address is being changed we can also change F to I or vice versa. There is the usual optional printing for F or I provided *H*0 = 0; this gives the address of each number printed and should not as a rule be suppressed since it provides a valuable check on the correct setting of the addresses.

The procedure for printing out a series of numbers from consecutive locations in the main store is as follows.

(a) Run key to STOP.
(b) Optional stop inhibit key DOWN (i.e. no optional stops).
(c) Set up F (or I) and the address of the first number on the handswitches.
(d) Start key to START and release it.
(e) Run key up to RUN.

Printing will continue. To stop it the best procedure is to raise the key for inhibiting optional stops and then set a new manual directive, after which the Run key can be operated or the start operation carried out. As usual with F or I the numbers are printed in a single column with the blocks separated by an extra line feed; if optional printing is not suppressed each number is preceded by its address. The extra optional printing takes a certain amount of time and if many consecutively stored numbers are to be printed it is best to suppress it. A useful procedure to make sure that the original handswitch setting was correct is the following.

(a) Run key to STOP.
(b) Optional Stop inhibit key UP (i.e. stops).
(c) Set up F (or I) and address of first number *and* *H*0 = 0 to allow optional printing.
(d) Start key to START and release it.
(e) Run key up to RUN (first number and its address are printed, then optional stop).
(f) Suppress optional printing (*H*0 = 1).
(g) Inhibit optional stops (key DOWN).
(h) Run key to STOP then up to RUN.

The last stop causes the first number to be printed again, this time without its address, and then the printing of the following numbers starts. This is because the handswitch setting was changed (*H*0 was depressed) at the first optional stop at step (f). Consequently the restart operation occurs at step (h) and the first number gets printed again. The optional printing with the first number (at step (e)) gives its address as set up on the handswitches; it is then easy to determine the addresses of subsequent numbers, since they are grouped into blocks. The address of the number actually being printed can at any moment be found by examining on the monitor tube the word in *U*5.3; the modifier part of this word gives the address. When it has reached or got beyond the last number wanted we raise the key for inhibiting optional stops (alternatively we can put the Run key to STOP but this may stop the printing in the middle of one of the numbers).

The manual K-directive operates in the same way as F and I. The manual P-directive is similar to F and I except that individual orders are printed. If the handswitches are not altered at an optional stop then printing of the next order occurs. If optional stops are inhibited consecutive orders are printed out starting at the address set on the handswitches. The various procedures described above for F and I can be applied also to P.

The manual X-directive can be used to change orders in the store; it is in some ways similar to F, I, K and P but there is in addition a 77-stop in the process, at which point a new *order* is set up on the handswitches. For example, if we wish to change the *b*-order in *B*15.6 to read 1.2 0 60 we set up the manual directive X 15.6+ and do a start operation. There is than a 77-stop and we now set up the order 1.2 0 60 on the handswitches,† and operate the Run key; this causes the order in the main store to be changed and an *optional stop* occurs. If we operate the Run key at this stage, *without changing the setting of the handswitches*, then the computer will prepare to change the next order in the store, i.e. the *a*-order in *B*15.7, and there is a 77-stop to allow us to set up this next order. The procedure may be repeated indefinitely. If we inhibit optional stops the computer will continue to replace consecutive orders in the main store by orders taken from the handswitches; there is a 77-stop each time to allow us to set up the next order. This process continues in fact until we *change* the setting of the handswitches at one of the optional stops (this can be done, of course, only if the optional stops are allowed to happen), at which point the restart operation immediately occurs and the new setting of the handswitches is taken as a manual directive.

For example, suppose we wish to change the *b*-order in *B*10.2 to read 12 6 40 and then enter the programme by E 2.0. The procedure is as follows.

(a) Run key to STOP.
(b) Optional stop inhibit key UP (i.e. stops).
(c) Set up X 10.2+ on the handswitches (X is the 24th letter so the five right-hand keys are 11000; of the other keys *H*7, 9, 12 and 14 are down).
(d) Start key to START and release it.
(e) Run key to RUN (there is optional printing of a letter X; then a 77-stop).
(f) Set up the *order* (12) 6 40 on the handswitches; the setting is as follows:

000 0001100 110 100000 000

(g) Run key to STOP and then up to RUN (the order is changed, there is optional printing of the address, the old order and the new order, after which there is an optional stop).
(h) Set up the directive E 2.0 on the handswitches.
(i) Run key to STOP and then to RUN (there is optional printing of E 2.0 and then a 77-stop before the programme is entered).

If desired the Start key could be operated just before step (h) above and the effects would be the same.

The following procedure can be used if we wish to change several orders in arbitrary places in the main store.

(a) Run key to STOP.
(b) Optional stop inhibit key UP (i.e. stops).
(c) Set up X and the first address on the handswitches.
(d) Start key to START and release it.
(e) Run key to RUN (there is a 77-stop).
(f) Set up first order on handswitches.
(g) Run key to STOP and then to RUN (there is an optional stop).
(h) Set up X and address of next order on the handswitches.
(i) Operate Run key (there is a 77-stop).
(j) Set up next order on handswitches.
(k) Operate Run key (there is an optional stop).

Steps (h) to (k) are then repeated as often as required. After each optional stop there is optional printing of the letter X, and after each 77-stop the address, the old order and the new order are printed optionally (i.e. if *H*0 = 0). When the last order has been changed (i.e. at an optional stop) we can set up a new manual directive and operate the Run key. Note that at any optional stop we can operate the Start key without affecting the results; this is indeed probably to be preferred as less likely to cause confusion. It is unwise to suppress optional printing when carrying out manual X-directives; the printing shows exactly what was actually done and it is then easy to put any errors right.

The procedure for changing a sequence of consecutive orders in the main store is as follows.

(a) Run key to STOP.
(b) Optional stop inhibit key DOWN (i.e. no optional stops).
(c) Set up X and the address of the first order.
(d) Start key to START and release it.
(e) Run key to RUN (there is a 77-stop).
(f) Set up first order on handswitches.
(g) Operate Run key (there is a 77-stop).
(h) Set up next order on handswitches.
(i) Operate Run key (there is a 77-stop).

Steps (h) and (i) are repeated as often as required. At any 77-stop the address of the order about to be changed can be read from the monitors by displaying C(5.3); the modifier part of this word is the address and the sign-bit is a 1 if it refers to a *b*-order. When the last order has been set up on the handswitches we can raise the key for inhibiting optional stops before operating the Run key; when the next optional stop occurs we can set up a new manual directive. Alternatively we can break the sequence by doing a start operation.

It should be noted that any X-directive always treats the stop/go digit of an order-pair as belonging to the *a*-order; we should therefore, normally set *H*0 = 1 when setting up an *a*-order; this key is *disregarded* in *b*-orders so it can safely be left down (after step (f), when it will have no effect on optional printing) when changing a sequence of orders in the way described above. Another small point to note is that it is necessary to change the setting of the handswitches in some way at the optional stop before a new manual directive can be read. Exceptionally it may happen that the

† This order is set up on the handswitches as though it were to be obeyed manually (see Sec. 6.7); the digits being set up on the groups *N X F M* of the keys. The setting for this order is as follows (see Sec. 3.12):-

000 1001010 000 110000 000
N *X* *F* *M*

The *N*-address in any such order must be an absolute one.

handswitch setting for the last order is exactly the same as the setting for the new manual directive. This is likely to occur only if a null order is put in last and it is then desired to read tape. It is easy to avoid this difficulty by doing a start operation instead of simply operating the Run key (alternatively we can use a manual ϕ-directive with an arbitrary non-zero address).

On the whole, manual directives of any kind should be used sparingly, and then always with optional printing. If the operator is in doubt about the procedure then only the simplest operations should be used. The safest directives are F, I, K and P since they cannot affect the contents of the main store.

If a programme goes quite wrong and, perhaps, stops after a few seconds it is often possible to get valuable information by printing out the contents of the computing store. The procedure is to obey manually (see Sec. 6.7) a sequence of orders such as either of the following.

(a)		or	(b)	
	0 [0] 73			0 [0] 73
	1 [1] 73			2 [2] 73
	2 [2] 73			3 [3] 73
	3 [3] 73			1 [1] 73
	4 [4] 73			5 [5] 73
	5 [5] 73			4 [4] 73
	6 [7] 73			6 [7] 73

These orders copy away the whole of the computing store, which can then be printed out, probably by the manual directive P 0.0. (Optional stops should be inhibited, and the printing can be stopped at the appropriate moment by consulting the modifier in U5.3. This contains the address of the word being printed, as has already been described.) Before doing this a note should be made of the state of the overflow-indicator and of the order that the machine was about to obey. The overflow-indicator can be cleared, just before obeying the above manual orders, by putting the Start key up to START before putting it down to MANUAL. Note that if the manual orders are obeyed in the sequence (b) given on the right above, fewer key changes are needed. It is usually best to use a P-directive for printing out the computing store in this way because it is usually orders that one is interested in and the information printed is complete, in the sense that the value of every binary digit can be determined. (It is easy to find the values of modifiers and counters.)

In table 7.4 are summarized the effects of the manual directives. Some directives are included which are described in later sections of the book. The optional printing is not given because it is the same as that caused by the corresponding tape directives, except for the manual ϕ-directive which has no analogue on tape (the tape ϕ-directive is equivalent to Z). In this table the following abbreviations are used:

RESTART	The restart operation occurs.
77 RESTART	A 77-stop occurs, followed by the restart operation.
TAPE	The Input section of the Initial Orders is entered to read tape.

The setting of the five right-hand keys of the handswitches is shown by a five-digit group in which a 0 represents a key which is raised and a 1 a key which is down. ▲

▼ **7.6 Block-transfer punching**

There is a key on the left of the main control panel (Plate 11) which is labelled PUNCH ON BLOCK TRANSFERS. When this key is depressed the computer operates normally except that every time a block-transfer order (i.e. a 72- or 73-order) is obeyed the computer will punch, in the output tape, the number of the main store block concerned in the transfer. The punching consists of three characters for each block-transfer order; the first character is always Er and the next two give the block-number in binary (these two characters each give 5 bits of the 10-bit block-number - in the *direct* code used in register 17). This punching is interspersed with the ordinary output (if any) of the programme being obeyed; it results in a fair reduction of speed as a rule.

This facility is a powerful aid in the diagnosis of blunders; it is provided by special circuits in the computer and is not connected in any way with the Initial Orders. Since, however, the punched information is in a binary code, it cannot be easily read or printed out. A special facility has therefore been put into the Initial Orders to convert the binary information into decimal; this is done with the ?-directive as described below.

If a programme is known to be correct up to a certain stage, after which its behaviour is difficult to account for, the facility of punching on block-transfers may provide a useful clue since the block-transfer orders determine the contents of the computing store and control most of the vital transfer of information to and from the main store. The following procedure, or some adaptation of it, may be used.

(a) Read the complete programme into the store in the usual way and then alter it (e.g. by an S-directive) so that the computer will stop when the last correct stage has been reached.

(b) Start the programme as usual (e.g. by an E-directive).

(c) When the computer stops because of the alteration made in the programme, tear off the output tape and press the RUN OUT button on the punch to obtain a leader of blank tape.

(d) Press down the PUNCH ON BLOCK TRANSFERS key.

(e) Allow the computer to carry on with the programme (e.g. by operating the Run key). The computer will now punch whenever a block-transfer order is obeyed.

	Last 5 keys		Effects	Reference
		Setting		
ϕ	0	00000	*Normal Start*: Set T.A. = 2.0, relativizer = 2, R.A. = 882.7, then TAPE after optional printing of three asterisks. *Restart*: TAPE (no optional printing)	7.5
A	1	00001	Decimal digit is taken from the red keys *H*11 to 13; keys *H*4 to *H*10 should be clear (other keys are disregarded). Effects as follows: A0 No use (enters Binary Input). A1 Prepare to use Assembly, then TAPE (after B operation). A2 Change to second tape-reader; 77 RESTART. A3 Process the programme, then TAPE (after B operation). A4 No use (reads addresses from tape and enters Binary Punch). A5 Search for Library on MAGLIB; 77 RESTART T A6 Search for routine on MAGLIB;	 8.2 8.2 8.2 8.7 8.7
B	2	00010	Set T.A. to new block, set relativizer; 77 RESTART.	7.5
C	3	00011	Set relativizer to start of routine specified; 77 RESTART.	8.8
D	4	00100	Print date and serial number after increasing serial number by 1; 77 RESTART.	4.3
E	5	00101	Enter programme at $a(+)$ after 77-stop.	7.5
F	6	00110	Print fraction in a. After optional stop repeat with next number from store if handswitches unchanged; otherwise RESTART.	7.5
G	7	00111	Make order-pair in a a go order-pair; 77 RESTART.	7.3
I	9	01001	Print integer in a. Otherwise as F.	7.5
J	10	01010	Enter programme at $a(+)$; no stop.	7.5
K	11	01011	Print 6-bit characters in a. Otherwise as F.	7.5
L	12	01100	Set accumulators from *B*0; obey link in *X*1 (and *B*0.1); no stop.	7.5
N	14	01110	Print name from input tape; 77 RESTART.	7.5
P	16	10000	Print order at $a(+)$. After optional stop repeat with next order from store if handswitches unchanged; otherwise RESTART.	7.5
Q	17	10001	Read complete programme (number tapped out after optional stop) from magnetic tape.	8.7
R	18	10010	No effect; 77 RESTART.	7.5
S	19	10011	Make order-pair in a a stop order pair; 77 RESTART.	7.5
T	20	10100	Set T.A. equal to a; 77 RESTART.	7.5
X	24	11000	Replace order at $a(+)$ by new order read from handswitches after 77-stop. After optional stop repeat with next order in store if handswitches are not again changed; otherwise RESTART.	7.5
Y	25	11001	Optional stop; 77 RESTART.	7.5
Z	26	11010	77-stop; 77 RESTART.	7.5
?	29	11101	Convert block-transfer tape.	7.6
Er	31	11111	Leave T.A. relativizer and R.A. untouched; TAPE (no optional printing)	7.5
All others			Loop stop.	

Table 7.4 *Summary of the manual directives.*

(f) Stop the computer manually when it is thought that enough information has been punched out (this can alternatively be programmed).

(g) Press the RUN OUT button on the output punch to obtain a tail of blank tape, *mark the end* of this tape and tear it off. This tape is referred to as a *block-transfer tape*.

(h) Return the PUNCH ON BLOCK TRANSFERS key to its normal position. *It is important* not to forget this operation.

(i) Place the leader of the block-transfer tape in the main tape-reader. Press the RUN OUT button on the punch for a moment.

(j) Set up a manual ?-directive by setting the five right-hand keys of the handswitches to 29 (i.e. 11101 in binary).

(k) Do a *start* operation to call in the Initial Orders. These will now read the block-transfer tape and produce a new tape with the block-numbers in decimal as described below. The process stops when the block-transfer tape runs out (or it can be stopped manually).

This is the way in which a block-transfer tape can be obtained and converted to a readable form with the aid of the ?-directive (read as "query directive"). This directive is almost always used manually; it causes the block-transfer tape to be read and ignored as far as the first Er character; the next two characters are then interpreted as a block-number and printed in decimal (preceded by CR LF) after which the next Er is sought. We therefore get a single column of the block-numbers; this is preceded by optional printing of "?". If there is any punching on the block-transfer tape between a binary block-number and the next Er then an asterisk is printed (preceded by CR LF); each asterisk printed among the block-numbers therefore corresponds to one or more characters on the block-transfer tape which will have been produced by the programme under test.

Note that if the normal output of the programme being tested contains any Er characters then misleading results will be obtained when the block-transfer tape is converted. This is unlikely to happen, however, since few programmes require to punch Er.

7.7 Binary input and output

The ordinary way of writing and punching a programme has been designed largely for the convenience of the user of the computer. It is clearly valuable to be able to prepare programme tapes easily and to print them out and edit and splice them in various ways. These features are less important, however, when a programme gets beyond the development stage and is used regularly for production work. The programme is then treated simply as a length of tape which causes the computer to do a certain job and, provided an adequate specification has been written, there is seldom any need to alter it nor to enquire what orders it contains.† At this stage what matters is that the programme tape should be complete and convenient to use, and that it should be short and capable of being read into the computer in the least time possible.

These requirements can usually be met by converting the programme tape from its ordinary Initial Orders form into a new abbreviated form; in fact we use the computer to prepare a new *binary tape* carrying the whole of the original programme in a highly compact form. This new tape can be read into the computer and is, for most purposes, the equivalent of the original tape; it is handled as though it were the original tape except that it cannot usefully be printed out on the tape-editing equipment. The words on a binary tape are punched in such a way that they closely resemble their form inside the computer: roughly, each binary digit corresponds to a single hole on the tape so that a single tape-character corresponds to 5 bits and eight characters are enough for a whole word (compared with the 18 characters needed to punch an order-pair in the usual way if both orders are unmodified). Such tapes are unintelligible to a human being but they can be read in at high speed (nearly 5 blocks per second) by a very simple input routine called *Binary Input*, which forms part of the Initial Orders.

There are two main programmes for preparing a binary tape, since it is, of course, almost impossible to punch such a tape by hand. The *Binary Punch* routine is a section of the Initial Orders; it can be used to punch out in binary a programme held in the main store; we shall describe below how it can be used. The special programme known as *Binary Translation* reads a tape punched in the ordinary Initial Orders notation and simultaneously punches out the corresponding binary tape. The tapes produced by either of these programmes are handled as if they were in the Initial Orders notation since a special directive (A0) appears at the start of the binary-punched information; this directive calls in Binary Input. A complete programme is normally converted into binary by using the Binary Punch routine since the result is a complete, self-contained tape. The Binary Translation programme is most often used to convert Library subroutines into binary.

Binary Input has its own set of *binary directives* corresponding to the most important ordinary directives (and a few others); they are of course punched in binary and are produced automatically as required by the Binary Punch and Binary Translation programmes. The ordinary user of the computer need not concern himself with them. The Transfer Address and relativizer (and the Reference Address) are used by Binary Input in the usual way.

The *A-directives* must now be described; they are concerned with the Assembly routine, Binary Input and Binary Punch, which are all sections of the Initial Orders. The letter A which introduces one of these directives is always followed by a single decimal digit, which may be 0, 1, 2, 3, 4, 5 or 6. This digit is considered as combined with the A to form a composite warning character; it is not an address. The effect of an A-directive is determined by the digit. The A0-directive calls in Binary Input; it is always immediately followed by information punched in the special binary code required by Binary Input. This directive is never punched manually, but is automatically put in where required when the binary tape is prepared. The directives A1, A2, A3, A5 and A6 are normally used with Assembly and are described in Chapter 8. The A4-directive calls in Binary Punch to punch out in binary some information held in the main store. This directive includes two whole-word addresses (as with F or I for example), which specify the first and last words to be punched out. For example, the directive

A4 2.0 - 23.7

causes the contents of *B*2 to *B*23 inclusive to be punched in binary. This directive would normally be punched as follows††.

Blank tape CR LF

λ Aϕ4 Sp 2.0 Sp - Sp 23.7 CR LF

Blank tape.

---

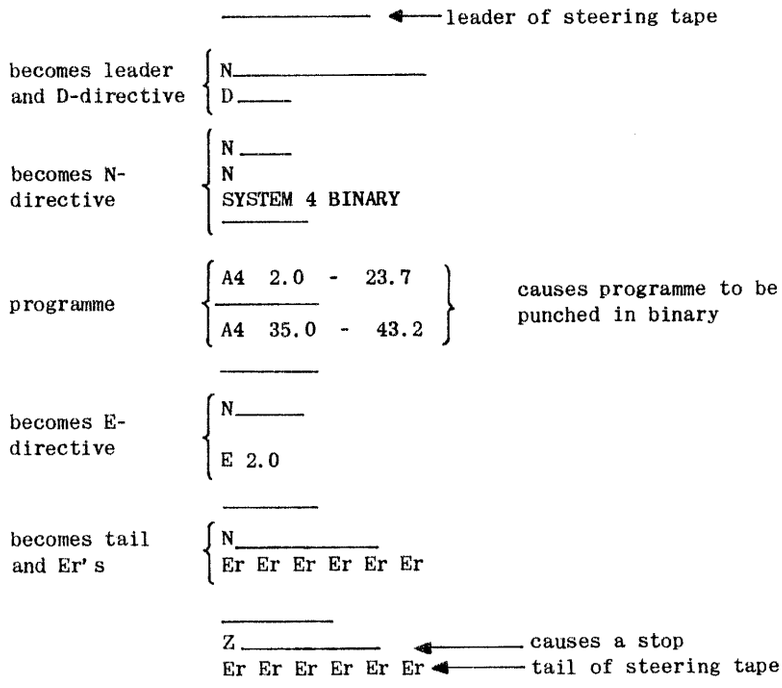
† It goes without saying that properly annotated programme sheets should be prepared for any complete programme; these supplement the specification of the programme, which should give adequate and full details of (a) the operating procedure, and (b) how the data are prepared, and (c) the form and nature of the results.

†† Here all the spaces are optional but they contribute to the legibility of the print-out and should be included (especially the Sp before the first address). The CR LF before the A is not strictly required but that after the two addresses must be put in. It should be noted that the digit 4 must immediately follow the λAϕ; *no spaces* are allowed between the A and the digit. As usual, Er characters may appear anywhere except between CR and LF.

The A0- and A4-directives must be read from tape and not from the handswitches; there is no optional punching with either of them. If an A4-directive includes only one address then the binary tape will contain only one word; this is sometimes useful. When the binary punching produced by an A4-directive is finished *Input* is called in to read more tape.

The binary tape resulting from the use of an A4-directive is suitable for reinput; when it is read in each word will be put back into the location it originally occupied; there is no arrangement for putting the words anywhere else. In fact at the start of the binary-punched section is an A0-directive to call in Binary Input; immediately after this is the binary equivalent of a T-directive, which sets the T.A. so that the information all goes into its original locations (this is usually what is wanted in a complete programme). At the end of the binary information there is a *checksum*, which is simply the sum of all the words and binary directives on the tape (disregarding overflow). This checksum is punched automatically when the binary tape is prepared; while the tape is being read in by Binary Input a new checksum is built up and this is compared with the one punched at the end of the tape. If there is disagreement then a loop stop occurs (in *U*3.4), but otherwise *Input* is called in to read more tape (in Initial Orders notation). A checksum disagreement is usually caused by a torn or dirty tape but may be due to a malfunctioning of the tape-reader; it is usually necessary to read the whole programme in again. This sensitive check is automatically carried out on all binary tapes.

While the binary tape produced by an A4-directive can be used as it stands for reinput, it is usually preferable that it should carry a few ordinary directives in addition to the binary information. Presumably the binary tape is to replace the original programme tape; it should therefore start with a leader and a D-directive and an N-directive at least; and it should finish with an E-directive and a tail of blank tape and some Er characters. It is quite possible to splice all these on to the "bare" binary tape, but it is preferable that they should be put in automatically at the time the binary tape is prepared. This is quite easy to do by using N-directives for putting information on to the output tape. Let us suppose, for example, that the programme to be punched out occupies locations *B*2.0 to *B*23.7 and also locations *B*35.0 to *B*43.2 inclusive; its name is to be SYSTEM 4 BINARY and it is to be entered by a directive E 2.0. We first of all prepare a *steering tape* as follows (horizontal lines indicate lengths of blank tape terminated by CR LF).



Most of the items on this steering tape correspond to something on the binary tape. The steering tape is read by the Initial Orders and produces the binary tape; before doing this we must of course put in the programme which is to be converted to binary. It is as well also to suppress optional printing (*H*0 = 1) and to tear off the tape at the output punch. Consider the sequence of events as the steering tape is read. First there is an N-directive and this causes the following "name" to be copied on to the output tape; the "name" here consists of a length of blank tape and a letter D. Thus we get the leader of blank tape and the D-directive at the start of our binary tape. Next on the steering tape is another N-directive; this causes copying on to the binary tape of the following "name" consisting of N and the actual name. Then come two A4-directives to punch out the actual programme from the main store. This is followed by two more N-directives, which copy E 2.0 and a tail on to the binary tape. This steering tape could be punched as follows (the characters underlined will be copied on to the binary tape - see the footnote to Rule E4 in Sec. 6.5).

Blank tape CR LF
λ N φ blanks CR LF
λ D φ blanks CR LF
λ N φ blanks CR LF
λ N φ CR LF LF
λ SYSTEM φ Sp 4 Sp λ BINARY φ CR LF φ

blanks CR LF

λAϕ 4 Sp Sp 2.0 Sp - Sp 23.7 CR LF

blanks CR LF

λAϕ 4 Sp 35.0 Sp - Sp 43.2 CR LF

blanks CR LF

λNϕ blanks CR LF

λEϕ Sp 2.0 CR LF ϕ

blanks CR LF

λNϕ blanks CR LF

Er Er Er Er Er Er ϕ

blanks CR LF

λZϕ blanks CR LF

Er Er Er Er Er Er

The A4-directive causes an inch of blank tape to be punched before the A0 and the binary information. The above steering tape will produce the following binary tape.

Blank tape CR LF

λDϕ blanks CR LF

λNϕ CR LF LF

λ SYSTEM ϕ Sp 4 Sp λ BINARY ϕ CR LF

blanks CR LF

λAϕ0 and binary information from *B*2.0 to *B*23.7

blanks CR LF

λAϕ0 and binary information from *B*35.0 to *B*43.2

blanks CR LF

λEϕ Sp 2.0 CR LF

blanks CR LF

Er Er Er Er Er Er

If the steering tape is run with optional printing (*H*0 = 0); the only difference is that the binary tape will start with three asterisks, which would have to be cut off. The sole purpose of the Z-directive at the end of the steering tape is to cause the computer to stop when the binary tape has been prepared.

The time taken to prepare a binary tape on the teletype punch is about 1 second per block of the programme. A fairly short programme can thus easily be converted into binary. With a long programme it may be advisable to produce a binary tape before development of the programme is complete; one must try to balance the time saved by using a binary tape for input during the rest of the development against the time wasted by preparing a binary tape. A binary tape is very convenient to use, and one may be able to give up the possibility of printing out and editing one's programme tape at a later stage of development. We can of course always splice corrections on to the end of a binary tape, and these can quite well be left there in the final version of the programme if they are not too numerous and extensive. ▲

▼ **7.8 Detailed description of the Initial Orders**

The Initial Orders may be usefully considered as made up of the following sections.

(a) The *Start* sequence, which is called in by a start operation; it will reset the T.A. etc. if the manual directive is ϕ and give the optional printing of three asterisks. The restart operation is also done here.

(b) A *Decode* sequence for reading warning characters and dealing with most of the optional printing. This sequence is entered when λ is read by Input or when a manual directive other than ϕ is to be carried out. It reads in a warning character and the ϕ following it and calls in Address Input if required. It also carries out the simpler directives (B, G, L, N, R, S, T, Y, Z, ϕ) and most of the X-directive and includes a few checks.

(c) *Input*. This reads the whole of the tape except for directives and binary-punched information. It is called in by a Normal Start operation and, usually, when a tape directive has been carried out. Input is left only when λ is read from the tape.

(d) *Address Input*. This reads in the one or two addresses in a tape directive. (The address is a manual directive is read in by the *Decode* section.)

(e) *Number Print* can be used for printing fractions or integers. It is used by F, I and D (for the serial number). It is also used to print the block-number in addresses in the optional printing for most directives, and by Order Print to print decimal *N*-addresses.

(f) *Order Print* is used by P and also for the optional printing for X.

(g) *A-directives.* This routine reads the single digit associated with A, does the optional printing and calls in the appropriate routine. It deals completely with A2.

(h) *C-directive.* This calls in Address Input to read the routine number, does the optional printing and sets the new relativizer (see Sec. 8.8).

(i) *D-directive.* This prints the data from *B*895.7, increases the serial number in *B*895.6 by unity and then calls in Number Print.

(j) *E- and J-directives.* This routine organises the transfer of four blocks of the programme into the computing store, sets the accumulators from *B*0, sets a link in *X*1 and enters the programme in *U*0 (the 77-stop associated with E is in the Decode sequence).

(k) *X-directive.* This is really only a part of the sequence of orders obeyed when an X-directive is read; it replaces the order in the store. The optional printing is mostly done by Order Print.

(l) *?-directive.* This routine converts the block-transfer tape, using Number Print to print the block numbers.

(m) *Next Operand.* This routine is used by F, I, K, P and X to advance the process to deal with the next number or order in the store and to test for the end. It deals with both tape and manual directives.

(n) *Binary Input* reads binary tapes; it is called in by an A0-directive.

(o) *Binary Punch* prepares binary tapes when called in by an A4-directive. It calls in Address Input to read the addresses.

(p) *Q-directive.* This routine is used to read and place on the drum complete programmes which are stored on the same magnetic tape as the library.

(q) *Assembly.* This is a complex programme which is described in Chapter 8; it may itself be regarded as made up of a number of sections:

Part 1
- A1, A5, A6 and L entries,
- Process tag-list,
- Process cue-list,
- Search index,
- Exit,
- Optional interludes,

Part 2
- Process programme (A3 entry),
- Assembly printing.

There are a number of routines used in connection with magnetic tape and cards: these will be discussed later.

It is desirable to describe the Input section in a certain amount of detail; it is then possible to determine the effects of certain slightly unusual punchings (e.g. in pseudo order-pairs), should this be necessary. While reading tape, Input may be in any of six different states, as determined by the nature of the item being read in and the stage reached in reading the item. These states may be numbered from 1 to 6.

Input is always entered in *state 1*, which is also known by the name *β-search* (beta-search). In this state the following characters are ignored:

Er, Sp, LF, CR LF.

Certain other characters determine the nature of the item to be read in; these are as follows.

Decimal digit (0-9): an order-pair.

Sign (+ or -): a number.

Figure shift (ϕ or blank): blank tape.

Letter shift (λ): a directive.

Any other character is treated as a fault and causes a loop stop. If a decimal digit is read state 3 is entered, which deals with the decimal digits of orders. A sign leads to state 5, which deals with numbers. A figure shift calls in state 6 (called the *α-search*), and a letter shift causes Input to be left for the Decode section dealing with directives.

In *state 2*, Input searches for the first character of a *b*-order, which should be a decimal digit (0-9). The characters

Er, Sp, CR LF

are ignored and all other characters cause a loop stop. If a digit is found then state 3 is entered.

In *state 3* the decimal part of the *N*-address is read in. This part of the address is made up of decimal digits; as each is read in, the partial address is multiplied by 10 and the new digit added. This state is left when one of the following characters is read:

⊙ (as in register addresses such as 3.7); here 8 is added to the partial address.

Sp (as in the *N*-address of an order of groups 4, 5 or 7); the *N*-address is now complete.

\+ (as in a relative block number); the relativizer is added to the *N*-address, which is then complete.

When any of the above characters has been dealt with, state 4 is entered to read the remaining (octal) digits of the order. If CR LF is read it is treated as terminating the order (e.g. a null order punched as 0 CR LF, or the counter in a pseudo order-pair consisting of a modifier and counter). In this state only Er is ignored; other characters not mentioned cause a loop stop.

In *state 4* the octal digits of an order are dealt with. When this state is first entered (from state 3) an octal digit counter c is set equal to 4. As each digit is read the value of c is reduced by 1, the partially formed order is shifted up 3 places (i.e. multiplied by 8) and the new digit is added. There should be either 3 or 4 octal digits after the *N*-address. Apart from digits the following other characters may be expected.

+ This normally occurs only in jump addresses (e.g. 3.7+); only the 6 least-significant bits of the address are retained and c is reset to 4. The *N*-address is now complete.

Er This is ignored.

Sp If c = 0 this is ignored, so we can punch spaces at the end of an order.
If c = 1 this is taken as replacing a zero *M*-address; the partial order is shifted up 3 places and c is made zero.
If c is neither 0 nor 1 the Sp is taken as coming between the *N*- and *X*- addresses and c is reset to 4.

– If a minus sign is read the only effect is to reduce c by 1; the partial order is *not* shifted up. (Thus all information before the minus appears 3 binary places to the right of its expected position.)

⊙ This indicates a stop order-pair provided it is read when c = 0.

CR LF This terminates the order; if c = 1 the partial order is first shifted up 3 places (to put *M* = 0). If c is not 0 or 1 there is a loop stop.

In *state 5* numbers are read in. As each digit is read in, the partly formed number is multiplied by 10 and the new digit is added. The character ⊙ indicates a fraction; it initiates a process whereby a power of 10 is built up as the succeeding digits are read (this starts as 1 and is multiplied by 10 as each digit is found). The end of the number is shown by Sp or CR LF. Erase is ignored, and other characters cause a loop stop.

State 6 is also known as the *α-search* (alpha-search); in this state Input ignores blank tape (ϕ) and the Er character only. The characters CR LF or LF cause the *β*-search (state 1) to be entered, and λ causes Input to be left for the Decode section dealing with directives. All other characters cause a loop stop. Note that it is only in this state that ϕ is ignored.

The above details are summarized in Table 7.5, which also gives a little extra information. In this table x represents the partially built up order or number; it is always zero on entry (to state 1 or 2). The quantity y is the power of 10 which is built up during input of a fraction; it can be taken as zero initially. The quantity c is the octal digit counter used to make sure that there are either 3 or 4 octal digits after the *N*-address of an order. The symbols 'TO 4' mean *enter state 4*, and so on. The expression END OF ORDER means that the following events take place:

a-order: x is now the *a*-order and is kept; x is set equal to 0 and state 2 is entered.

b-order: x is now the *b*-order; the *a*-order and the stop/go digit are added and the resulting order-pair is stored in the location specified by the T.A., which is then increased by 1. State 1 (*β*-search) is then entered (with x = 0).

The phrase END OF NUMBER means that firstly the sign of x is changed if a minus sign was read, secondly x is divided by y (the power of 10) if a point ⊙ was read, and finally the result is stored in the place specified by the T.A., which is then increased by 1. We then return to state 1 (*β*-search) after setting x = 0.

If overflow occurs at any stage of the operations of Input it is not immediately detected, but OVR remains set as there are no 23-, 64- or 65-orders anywhere in Input. The computer will consequently stop when attempting to write the result into the main store. Such overflow is most likely to be caused by one of the following:

(a) two many digits in a number,
(b) number too big,
(c) relativizer plus *a*-order too big (e.g. if relativizer $\geqslant$ 128).

Note that the integer -2^{38} may *not* be punched as –274877906944, since the intermediate result 2^{38} will be built up before the sign is changed at the end. Fractions may not have more than 11 digits after the point, or the power of 10 (i.e. y) will overflow. If we punch a number such as +1.2 overflow will occur when Input divides x by y (i.e. 12 by 10 for this number); the number –1.0, on the other hand, leads to the permissible division of –10 by 10. Zeros in front of the ⊙ in a fraction have no effect.

The *γ-search* (gamma-search) is not a part of Input, but is in the Decode sequence. It is called in whenever λ is read by Input in states 1 or 6, and has the function of reading in the next character, ignoring only Er; this is a warning character and is then dealt with appropriately (after a check has been made that the next character is ϕ, again ignoring Er).

The various stops which occur in Input are included in Tables 7.7 and 7.8 which show the error stops in the Initial Orders. These stops are powerful aids to the detection of punching errors.

Input is called in in a special way after an X-directive so as to read in single orders. These orders are read in as though they were *a*-orders (e.g. LF is ignored) but a loop stop occurs if a letter shift or a sign is read in state 1. The action when the order is complete is rather different, of course.

The *Address Input* section of the Initial Orders has the function of reading the one or two addresses punched in a tape directive. Each of these addresses is stored in the same form: as a modifier, with the sign-digit of the word indicating whether the address is that of an *a*-order or a *b*-order (the sign-bit is 1 in a *b*-order address). A single address is left in *X*3 and in *U*5.4; if there are two addresses the first is left in *X*3 and *U*5.3 and the second in *U*5.4. Address Input can be in any of three states at the various stages of reading each address; the states can be numbered 1, 2 and 3. The effects of various characters in each state are summarized in Table 7.6, in which x represents the address being built up. The word (STOP) in brackets means that OVR gets set, which causes a stop when CR LF is read at the end of the directive.

Tape character	① β-search First character	② First digit of b-order	③ Remaining decimal digits of orders	④ Octal digits of orders	⑤ Numbers	⑥ α-search Blank tape
0 - 9 (decimal digit d)	First digit of a-order. $x' = d$ TO 3.	First digit of b-order. $x' = d$ TO 3.	Next decimal digit of order. $x' = 10x + d$. TO 3.	Next octal digit of order. $x' = 8x + d$, $c' = c - 1$. TO 4.	Next digit of number. $x' = 10x + d$, $y' = 10y$. TO 5.	STOP
+	Mark + TO 5.	STOP	Add relativizer to x, $c' = 4$. TO 4.	Retain only 6 l.s. bits of x, $x' = x$ & 63 $c' = 4$. TO 4.	STOP	STOP
−	Mark − TO 5.	STOP	STOP	$c' = c - 1$. TO 4.	STOP	STOP
⊙	STOP	STOP	$x' = x + 8$, $c' = 4$. TO 4.	If $c = 0$ mark stop order-pair. TO 4. If $c \neq 0$ STOP	Set $y = 1$ and mark as fraction, but STOP if already so marked. TO 5.	STOP
CR LF	Ignore. TO 1.	Ignore. TO 2.	END OF ORDER	If $c = 0$, END OF ORDER If $c = 1$, $x' = 8x$ then END OF ORDER If $c \neq 0$ or 1, STOP	END OF NUMBER	Enter β-search TO 1.
LF	Ignore. TO 1.	STOP	a-order: Ignore. TO 3. b-order: STOP	a-order: Ignore. TO 4. b-order: STOP	STOP	Enter β-search TO 1.

Tape character	① First digit of address	② Remaining decimal digits of address	③ Octal digit of address
0 - 9 (decimal digit d)	$x' = d$, TO 2.	$x' = 10x + d$, TO 2.	$x' = x + d$, TO 3.
	(STOP)	$x' = 8$ (x + relativizer), TO 3.	If marked as b-order address: (STOP) Otherwise: mark as b-order address: TO 3.
–	(STOP)	If a second address has been marked: (STOP). Otherwise: x to 5.3 (first address), mark second address to come, TO 1.	
⊙	(STOP)	$x' = 8x$, TO 3.	Ignore, TO 3.
LF Sp Er	Ignore, TO 1.	Ignore, TO 2.	Ignore, TO 3.
CR LF	(STOP)	If one address: x to $X3$ and 5.4 (address) If two addresses: first to $X3$ (and 5.3), x to 5.4 (second address), Then: If OVR set: STOP. If OVR clear: Return to *Decode*.	
All others	STOP	STOP	STOP

Table 7.6 *The effects of various characters in Address Input*

Order number	Order	Cause of stop
0.0	0.0 3 63	b-order address after T, or B with negative T.A.
0.5	0.0 7 60 6	Unassigned warning character.
0.5	0.5 1 63	t > 4 in sequence λAφ t (No magnetic tape)
0.5+	0.5+ 5 61	Wrong character, or no φ after warning character.
†0.6	0.6 1 63	t > 6 in sequence λAφ t
0.6	0.6 7 62	λ in order pair or number or after X or R; may be due to omitted order.
0.6+	0.6+ 2 62	
†1.2+	1.2+ 7 60	Q-directive, routine not available.
1.5	1.5 5 61	CR not followed by LF after warning character address.
1.6	1.6 2 62	Sign (+ or –) at start of b-order or after X or R.
2.2+	7 1 01	Binary input. Order overflow due to addition of Relative Address (9½ hr. stop)
2.3	2.2+ 4 67	
2.6	2.6 5 61	Directive address overflowing or wrongly punched.
†3.1	3.1 1 63	Assembly cue-list address too small.
3.3	3.3 3 63	F, G, I or S followed by b-order addresses.
3.4	3.4 6 61	Binary input. Checksum disagreement.
††3.5	3.5 1 61	Assembly. L and φ's after cue list not followed by λN or CR LF.
4.1	4.1 3 63	Either (a) T followed by b-order address or (b) B read with C(5.7) negative.
†4.2-4.3+		Loop after handswitch Q, if optional stops inhibited (see table 7.10, order number 4.2). Clear handswitches, operate Run key, and tap out the routine number.

Table 7.7 *Loop stops in the Initial Orders*

† Not applicable to 4096 store. †† In the 4096 Assembly this error causes a loop stop in 3.4.

There are four places in the Initial Orders where a writing-with-overflow stop may occur; these are shown in Table 7.8. It should be noted that if a 71- or 73-order is encountered with OVR set the computer stops *instead of* writing into the main store, and it stops as if it were about to obey the *next* order. This next order is consequently shown in the table and may help to identify the stop on the monitor. The states of the order-number register and the "next-order" lights when the stop occurs are given in the first column.

Order number	Orders	Reason for overflow
1.1 (*b*)	(0 4 73 5) 2.3 4 67	Binary input. Relative Address too large after binary T.
1.2 (*a*)	(0 0 71 3) 1.7 5 63	After X-directive. New order overflowing.
†1.3 (*b*)	(20 7 11) 2 3 76	Spurious information in block 0, section 0, of a magnetic tape.
3.0 (*b*)	(0 0 71 6) 35 2 02	Input. Tape word overflowing; may be due to, for example:- (a) too many digits in a number (b) number too big (c) order plus relativizer too big.

Table 7.8 *Writing-with-overflow stops in the Initial Orders*

Order number	Cause of stop and usual action required
†0.0 (*b*)	Magnetic tape isolation or 16/32 word key wrongly set.
†0.5 (*b*)	Routine read by a handswitch Q. Operate Run key to enter the Initial Orders.
0.6 (*b*)	Z-directive. Operate Run key after changing tapes (or handswitch setting).
†0.7 (*b*)	Magnetic tape library not isolated or not mounted on the selected mechanism after Q-directive.
4.0 (*a*)	E-directive. Operate Run key to enter programme; entry address is in 3_M.
4.0 (*b*)	Manual X-directive. Set new order on handswitches and operate Run key.
†4.1 (*b*)	Magnetic tape library not isolated or not mounted on the selected mechanism.
4.5 (*a*)	Most manual directives. Set new manual directive and operate Run key for restart.

Table 7.9 *77-stops in the Initial Orders*

The 77-stops and optional stops in the Initial Orders are listed in Tables 7.9 and 7.10. In the first column of these tables are given the order-number, as shown on the monitor, and the "next-order" light visible when the stop occurs.

Order number	Cause of stop and usual action required
0.7 (*a*)	Y-directive. Operate Run key after changing tapes (or handswitch setting)
3.7 (*a*)	C-directive. Specified routine not read in (see Sec. 8.8).
†4.2 (*a*)	After Q from handswitches, clear handswitches and Run. (There will be a loop stop in 4.2 to 4.3 if optional stops are inhibited. See table 7.7).
4.6 (*a*)	Manual F, I, K, P or X. Operate Run key without changing handswitches to deal with next order or number in the store; or set new manual directive and operate Run key for restart.

▲ Table 7.10 *Optional stops in the Initial Orders*

† Not applicable to the 4096-word store.

Chapter 8

Assembly

The building up of a complete programme from its component routines can be a complicated task. The Assembly section of the Initial Orders is designed to do it automatically and to introduce as muc flexibility as possible in the writing of subroutines and master-programmes.

8.1 The purpose of Assembly

A complete programme is usually made up of some or all of the following items.

(a) A master-programme,

(b) Subroutines from the Library,

(c) [illegible]

There may also be lists of numbers or other constants but we shall disregard these for the present. T subroutines which are not in the Library, and which will probably have been drawn up specially for the programme, will be referred to as *programmer's subroutines* (or private subroutines). Generally speaking, most of the detailed work is done by the subroutines; the task of the master-programme beinı to organize the calculation, by counting, setting numbers and modifiers needed by subroutines, obeying cues to call in subroutines, dealing with the results, and so on. The master-programme may, of course, undertake some calculation itself. The whole question is discussed in Section 4.2.

The Assembly routine is a section of the Initial Orders which assists in building up inside the computer a complete programme made up of a master-programme and subroutines as described above. One of the problems in assembling a complete programme is the allocation of space in the main store for each part of the programme. It is most undesirable to have to fix in advance the address of each subroutine; and yet the cues in the master-programme (and elsewhere) will depend on these addresses. In fact, using only the facilities so far described in this book, it is not possible to write the cues in the master-programme until the whole of this programme has been written; for it is only then that we know exactly how much storage space it occupies and so can allocate definite addresses to all the subroutines. A possible technique which we can adopt during the writing of the master-programme is to leave blank any word where a cue is needed, and simultaneously to make a note in a special list to record which cue is needed to which subroutine, and where it is to be written. When the master-programme has been completed (apart from the cues) we can allocate the main store and fix the address of each subroutine (it is only at this stage that we can be sure which subroutines are needed). We can then go through the list of cues wanted and fill in the cues in the blank spaces left for them in the master-programme.

Since this last process is quite automatic and is subject to definite rules we can programme the computer to do it; it is in fact one of the things done by Assembly. The allocation of the store among the various parts of the programme is also undertaken by Assembly. This is a straightforward task; the parts of the programme are simply placed in the store consecutively as they are read in, except that each part is made to start at the beginning of a block. The routines are stored as though there were simply a B-directive before each. Assembly will also select the Library subroutines needed by a programme and incorporate them in the programme. To use Assembly we must punch certain special directives on our tapes as explained below (see Sec. 8.2). We prepare a programme tape containing these directives together with the master-programme and the programmer's subroutines, but *not* including any Library subroutines: this tape is then placed in the main tape-reader. In the second tape-reader we put the *Library Tape*, which contains all the subroutines in the Library. After the programme tape has been read in by the Initial Orders (acting under the control of Assembly) the Library Tape is scanned; as each subroutine is found it is examined to see whether or not it is needed by the programme. Unwanted subroutines are rejected but the others are put into the store, each starting at the next available block. Usually all the subroutines required are found fairly quickly and Assembly then stops scanning the Library Tape. At this stage the whole of the programme, including all the subroutines, is held in the Main store but there are blanks here and there where cues are required. Assembly now *processes* the programme by putting in all these cues; the programme is then complete and can be entered in the usual way, for example by an E-directive.

The operations of Assembly thus fall into two parts: the first is the input of the master-programme and the subroutines (including Library subroutines), and the second is the processing of the programme, which consists primarily of filling in the cues (which could not be put in earlier). The first part usually takes a minute or so and the second part less than a second; when this has all been done the complete programme is ready to be obeyed and we have finished with Assembly altogether.

In addition to dealing with cues, Assembly also provides the facility of *preset-parameters*. A parameter is a constant or number which affects in some way the operation of a subroutine, but whose value cannot be specified at the time when the subroutine is written. A programme-parameter is a parameter which is set (i.e. given a particular value) in the programme; usually it is set by the

Plate 1: A general view of Pegasus 2

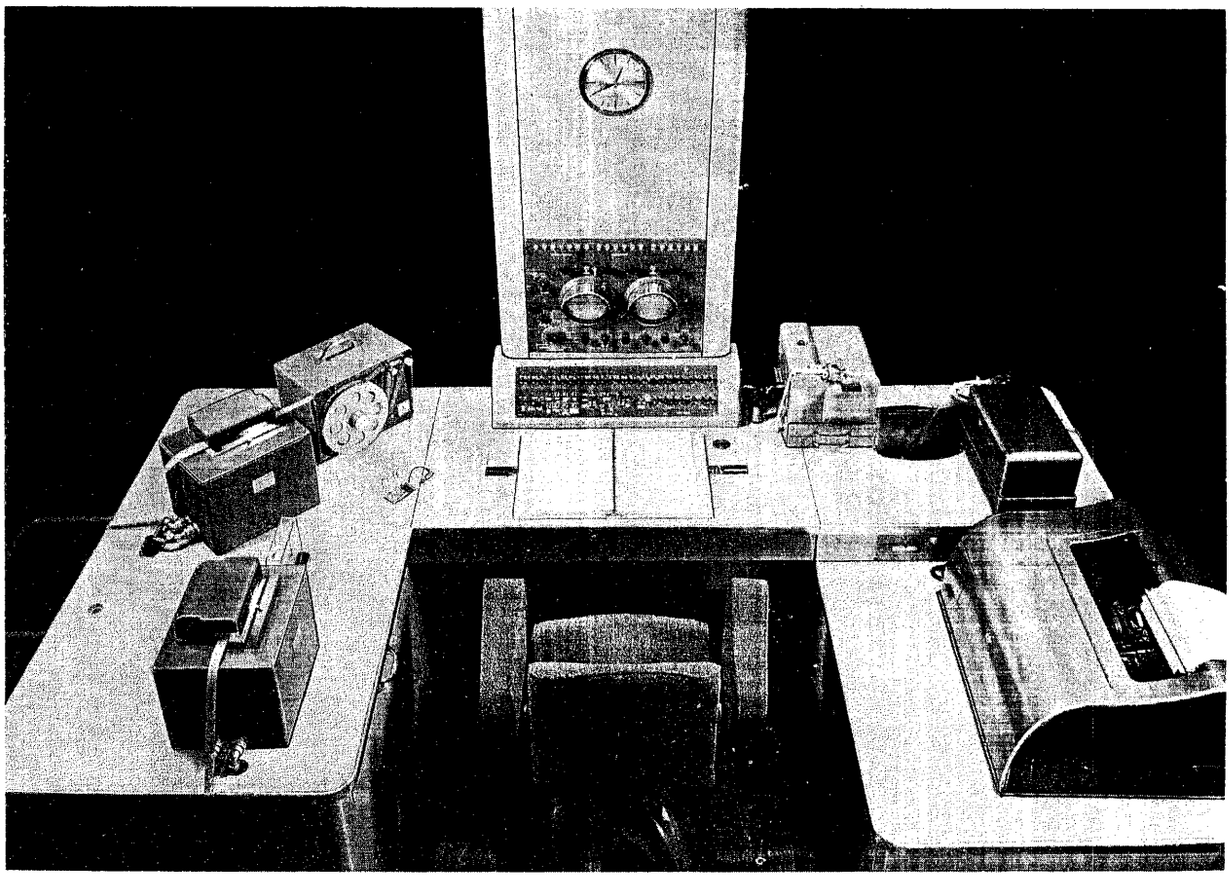


Plate 2: The control desk of Pegasus 1

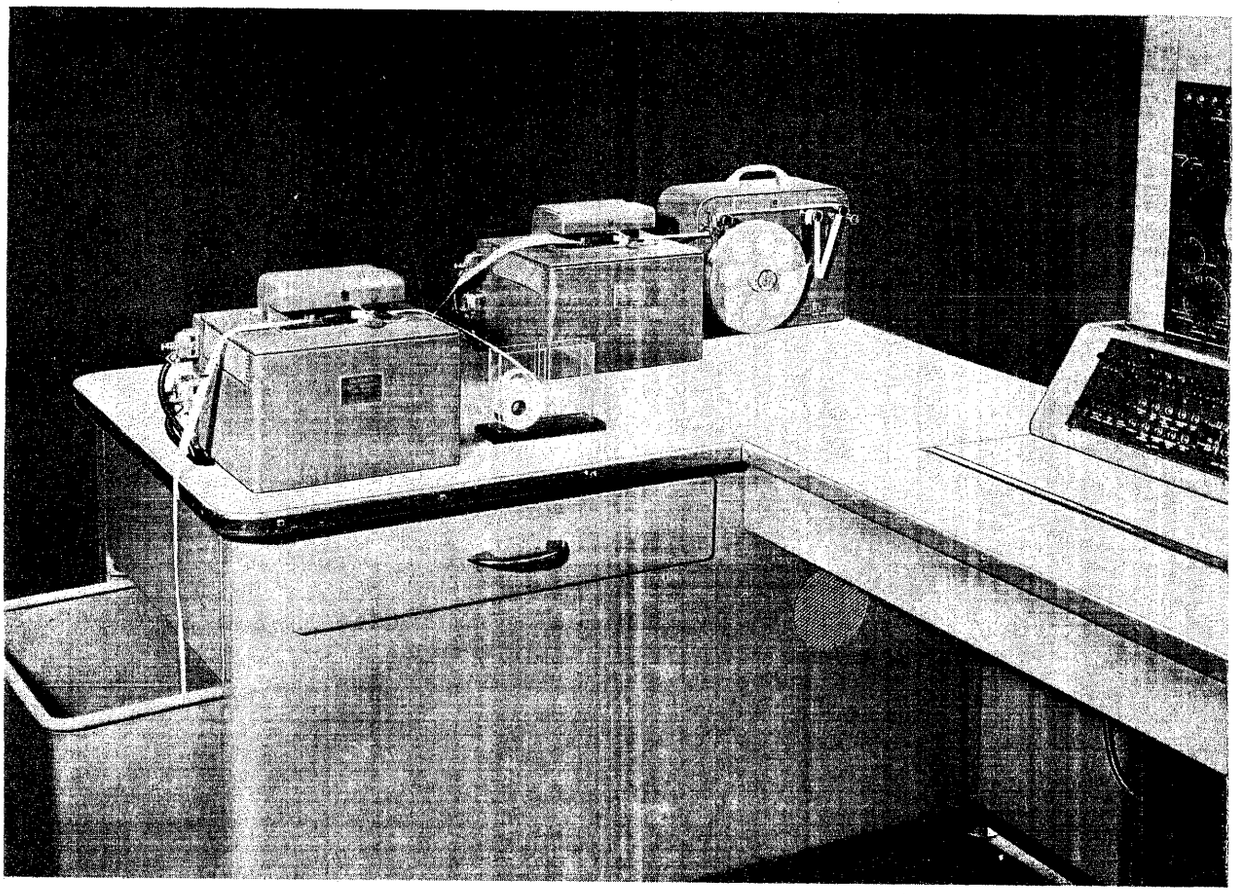


Plate 3: The paper tape input equipment

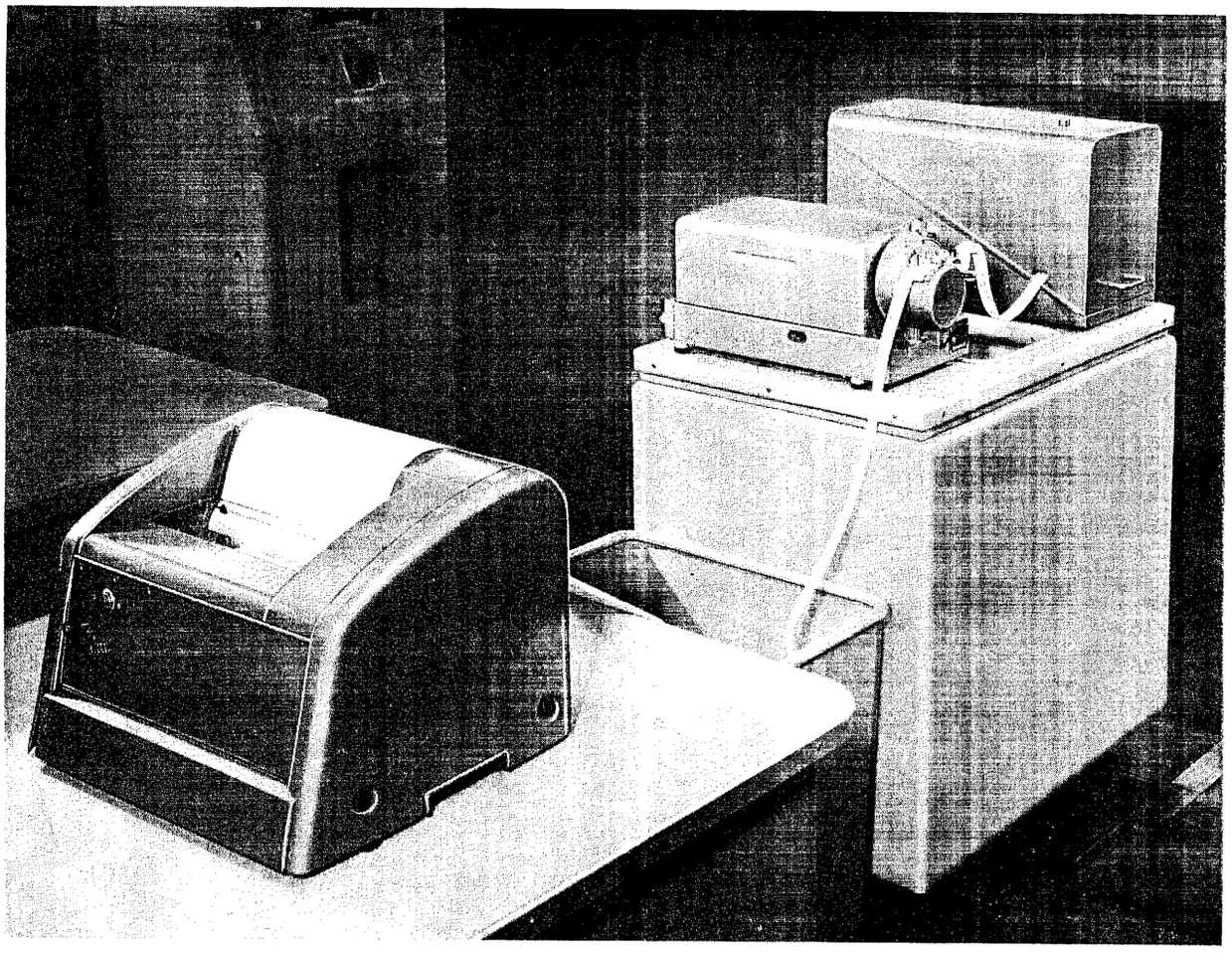


Plate 4: The paper tape output equipment of Pegasus 2

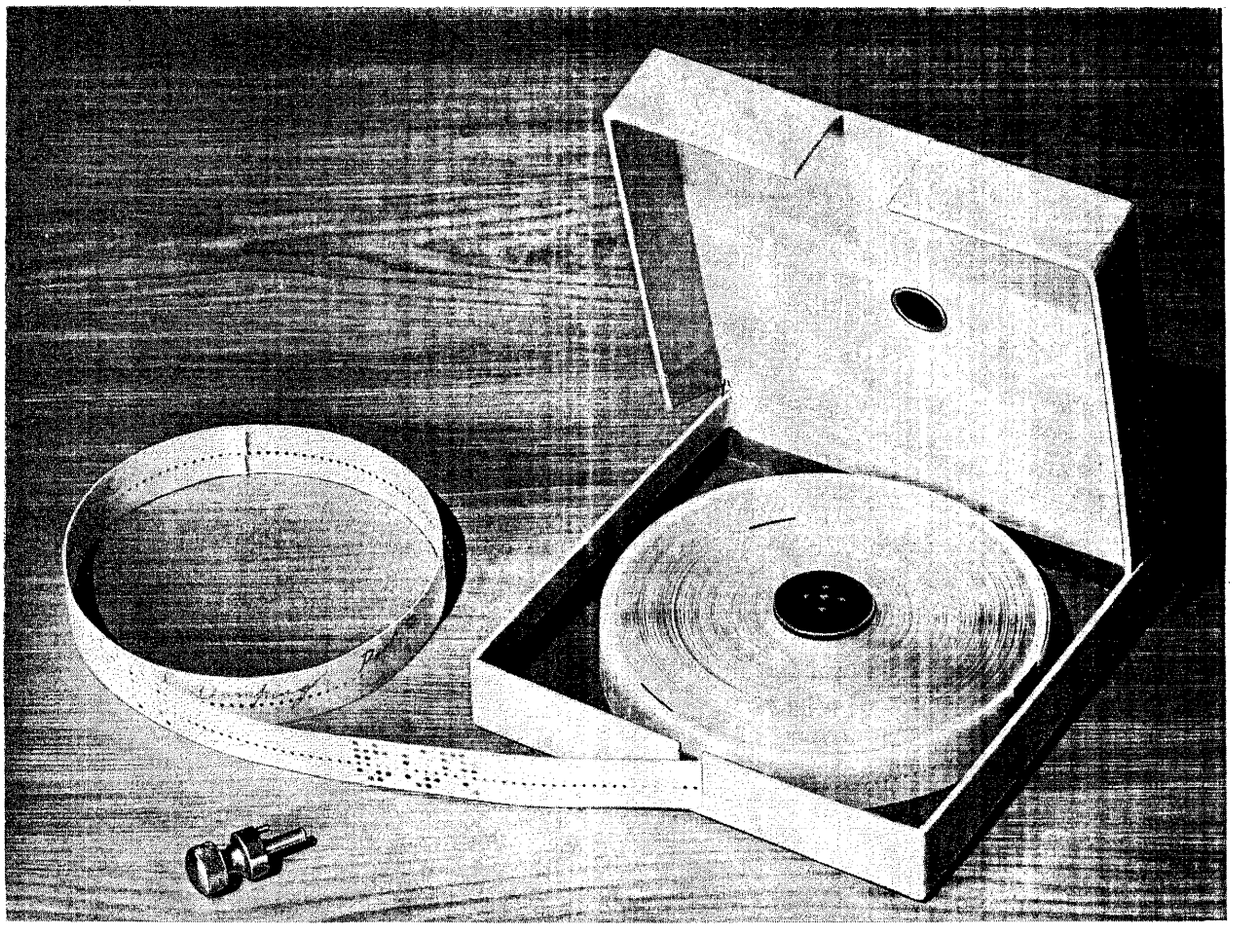


Plate 5: A tape box type A4 with punched paper tape

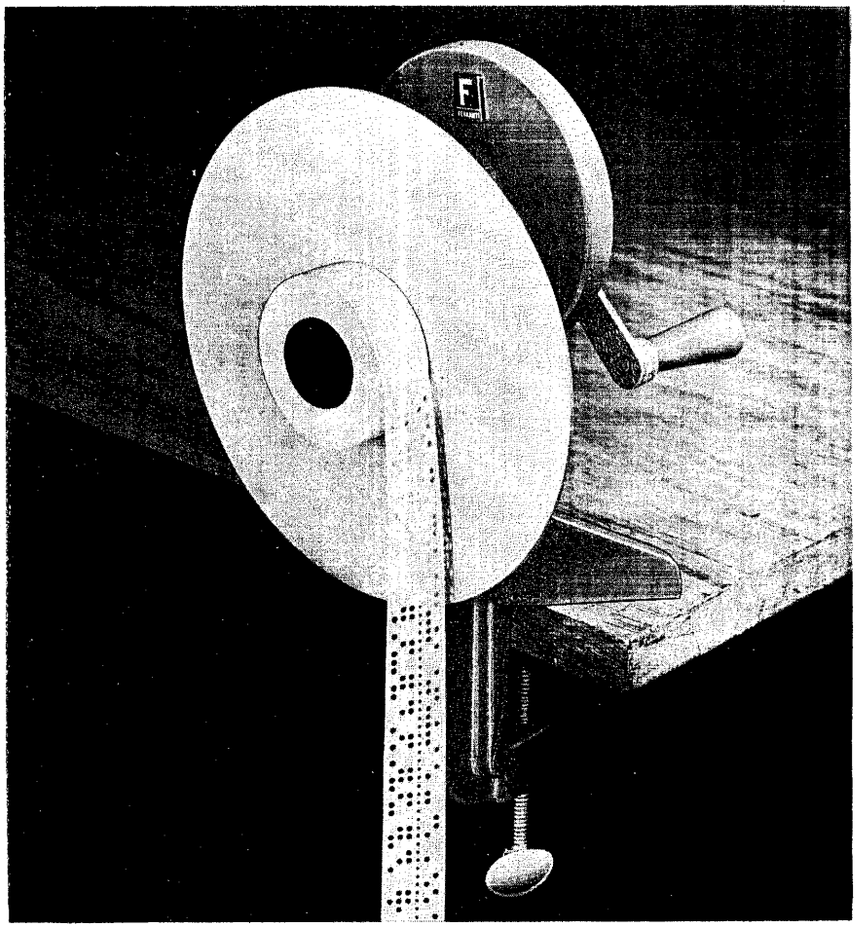


Plate 6: A hand spooler type A13

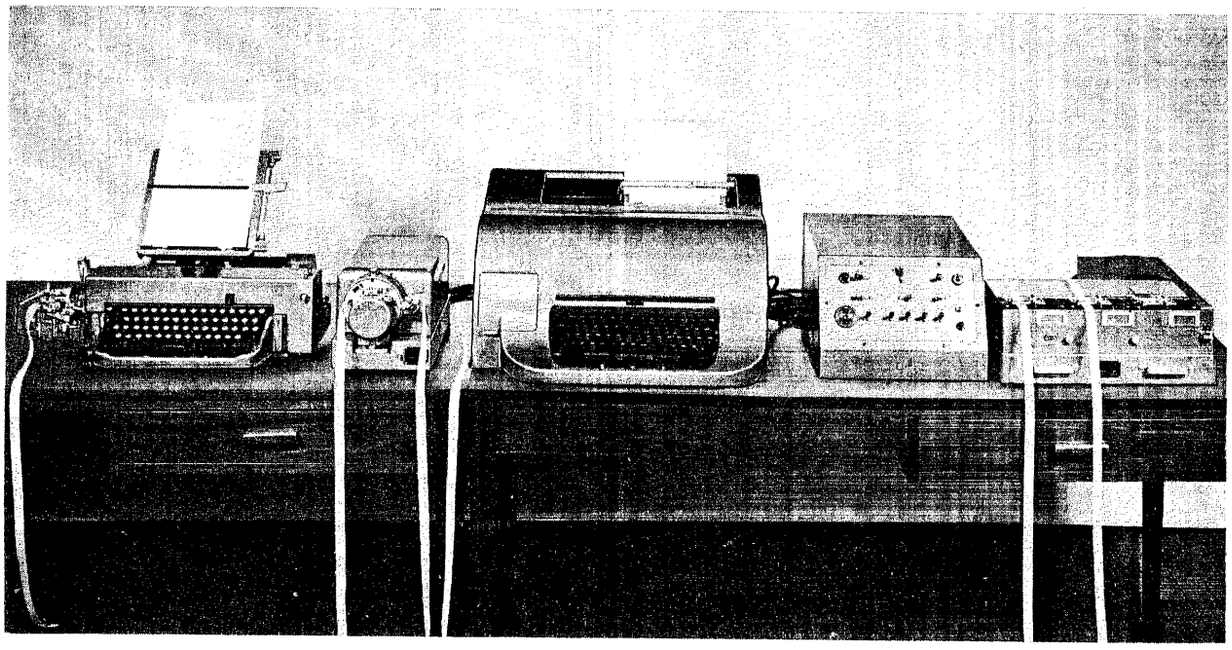


Plate 7: The full set of tape-editing equipment

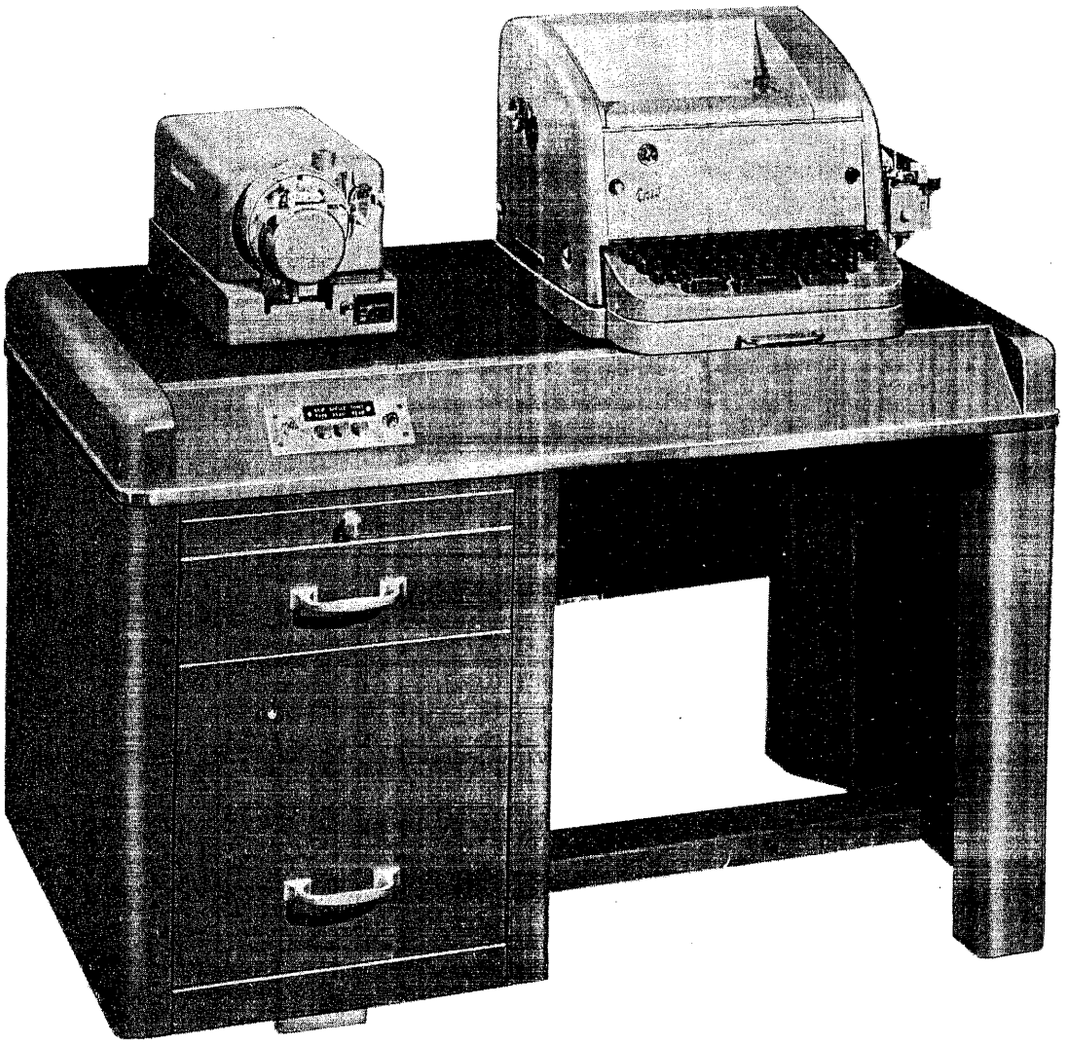


Plate 8: The simplified set of tape-editing equipment

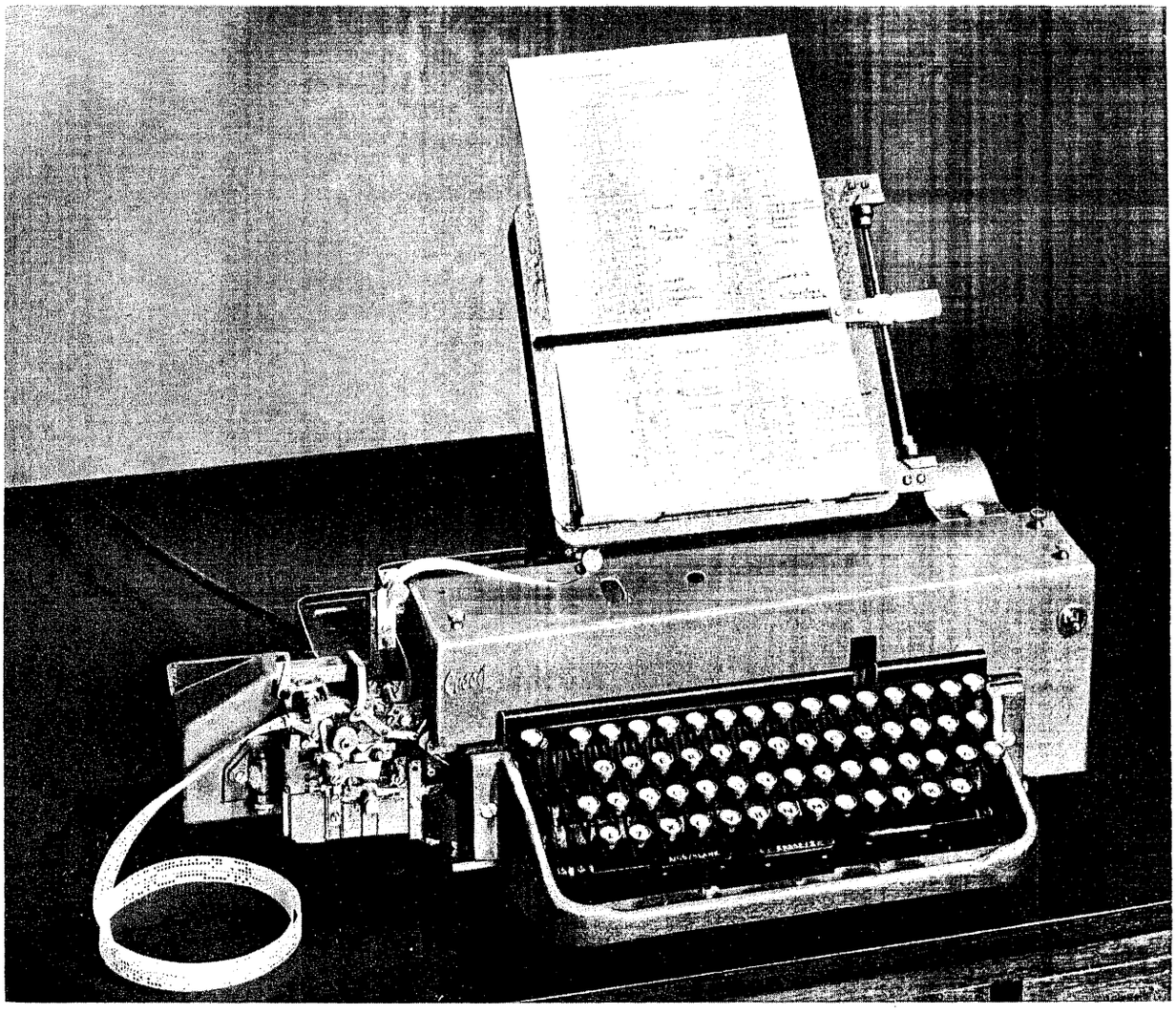


Plate 9: A keyboard perforator

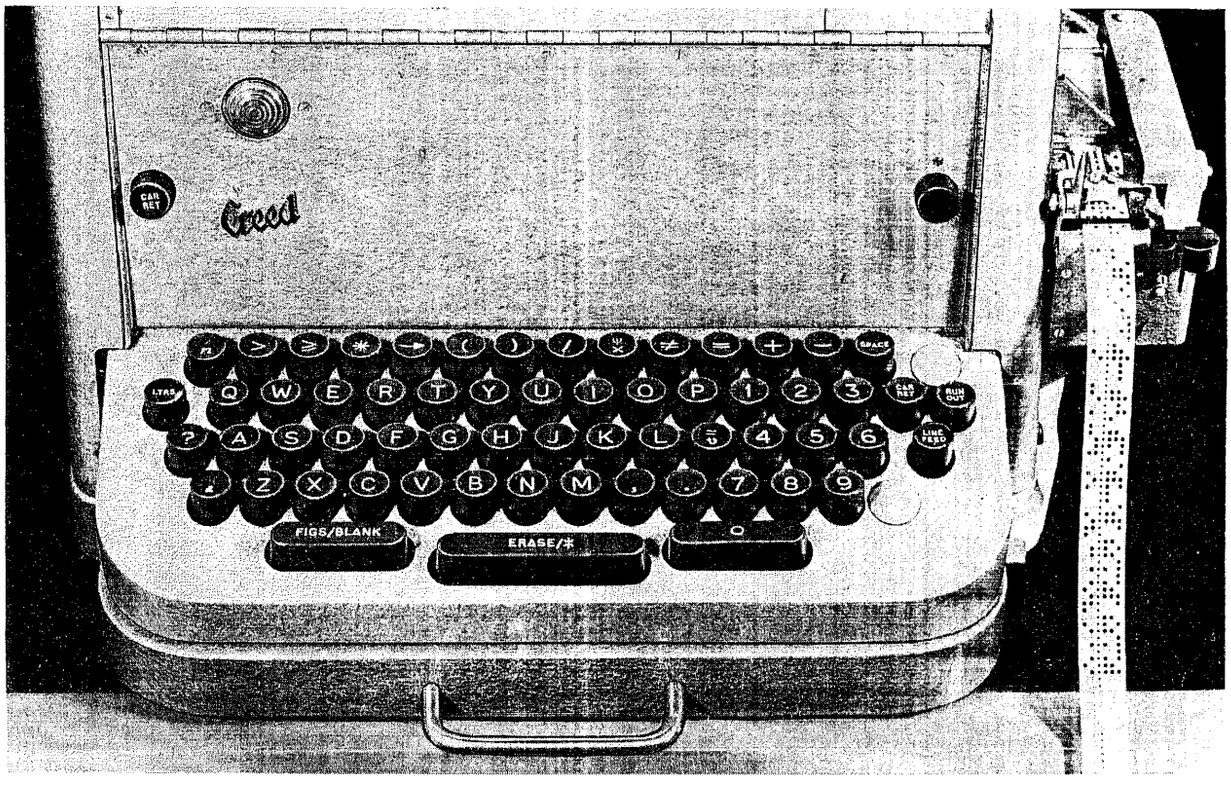


Plate 10: The keyboard of a Creed Model 75 teleprinter

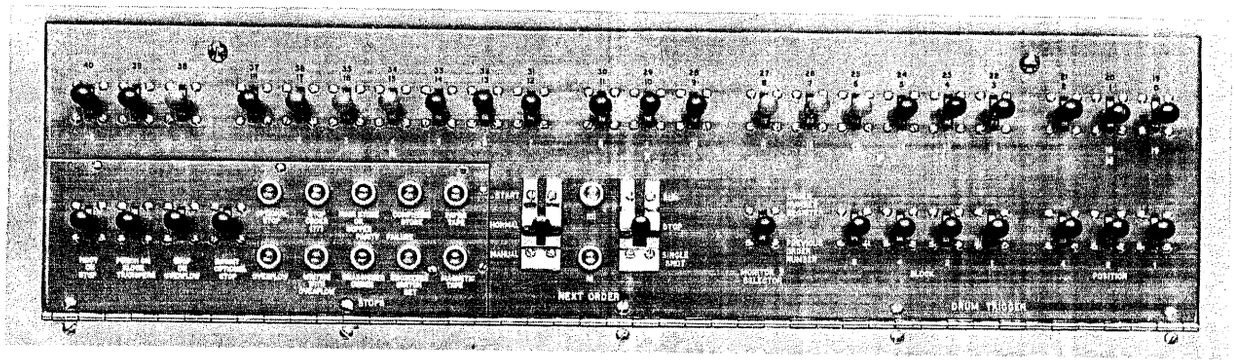


Plate 11: The programmers' control panel of Pegasus 2

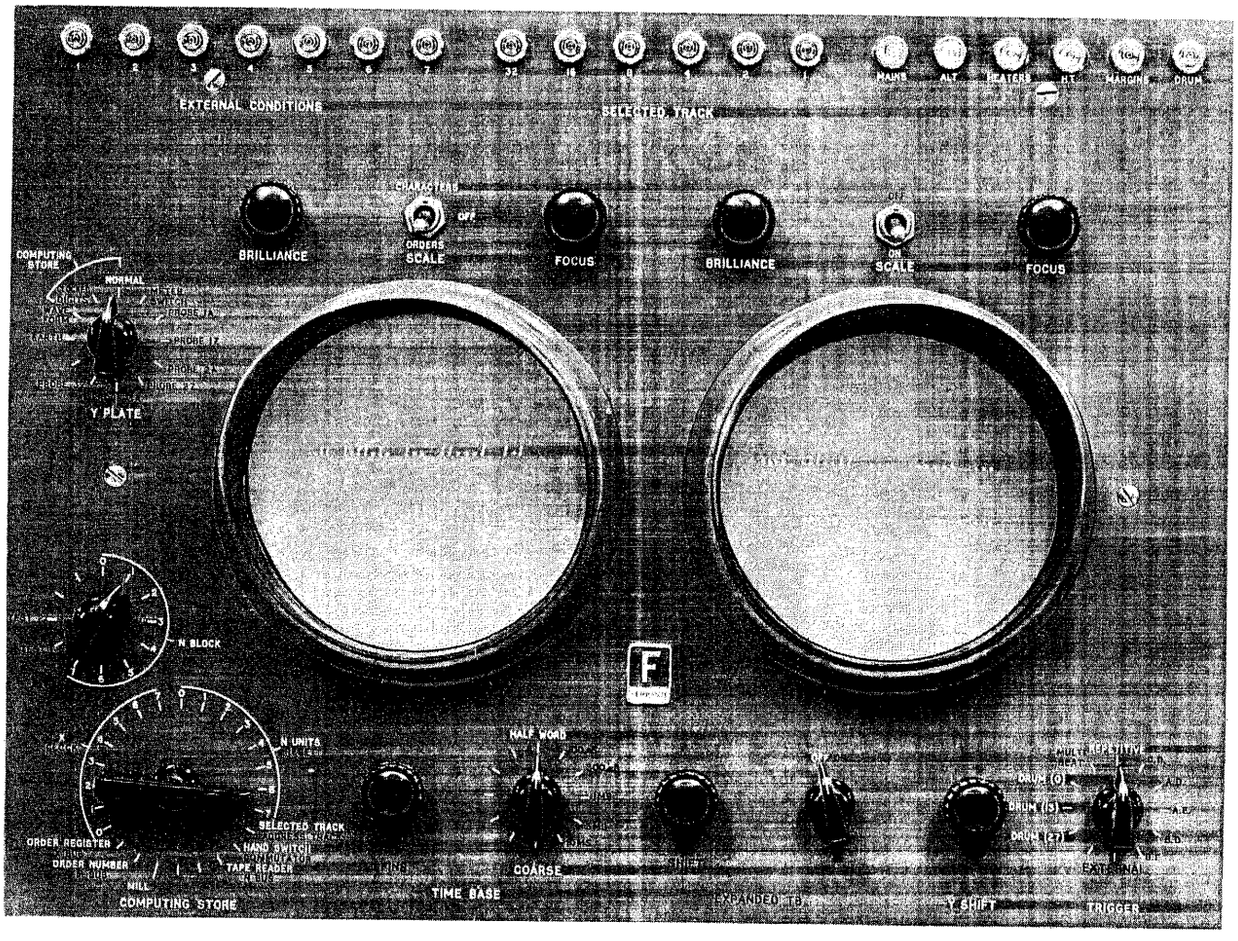


Plate 12: The monitor panel of Pegasus 2

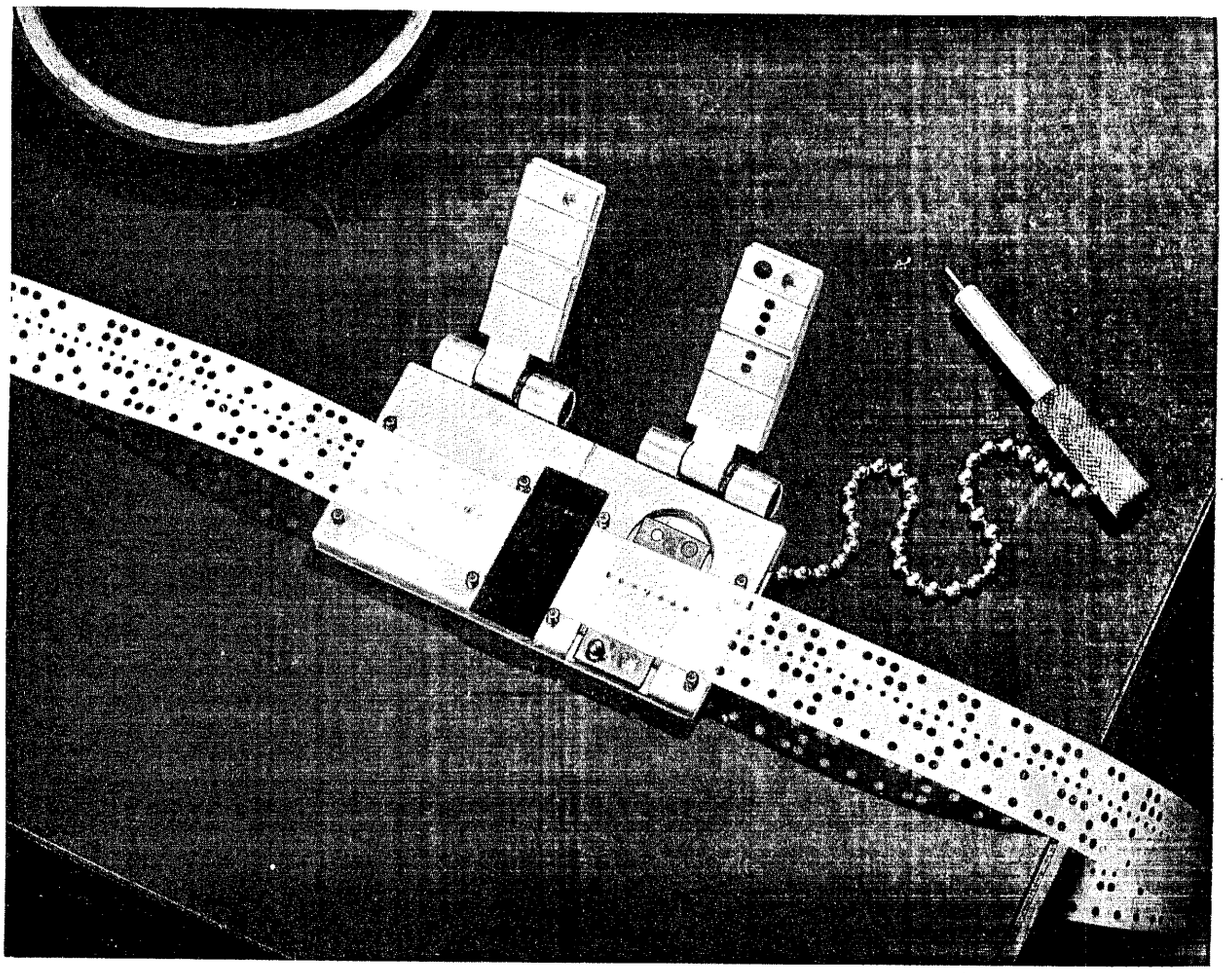


Plate 13: A unipunch in use for tape splicing

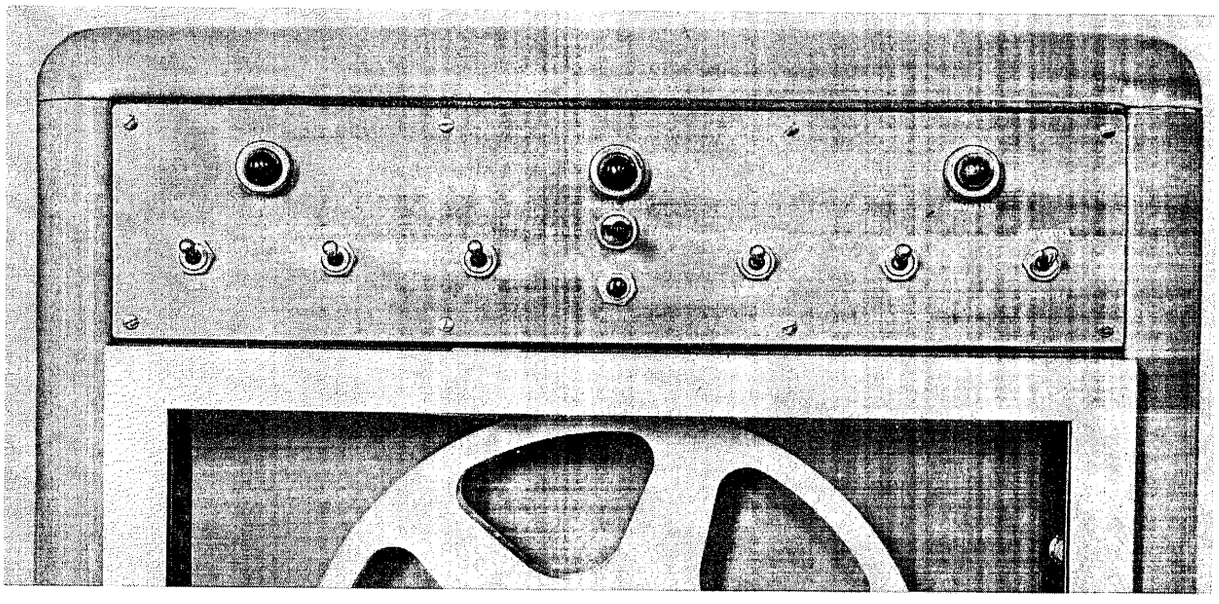


Plate 14: The controls of an ElectroData magnetic tape mechanism

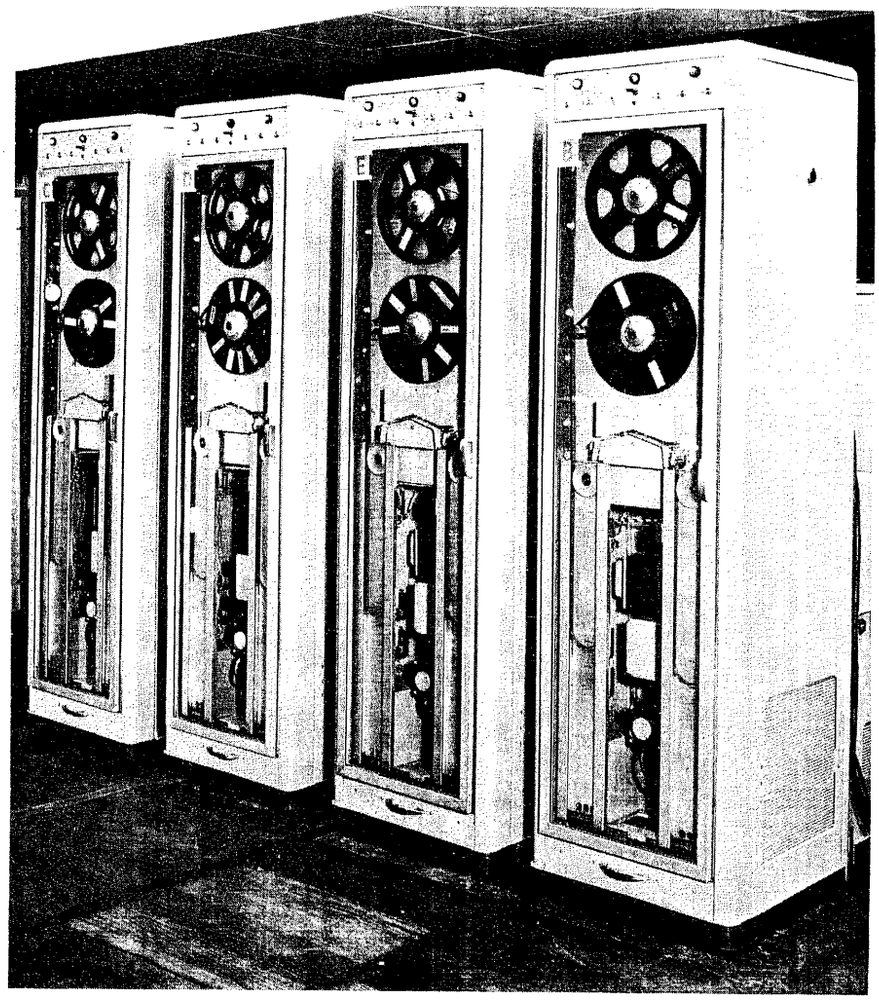


Plate 15: ElectroData magnetic tape mechanisms attached to Pegasus 2

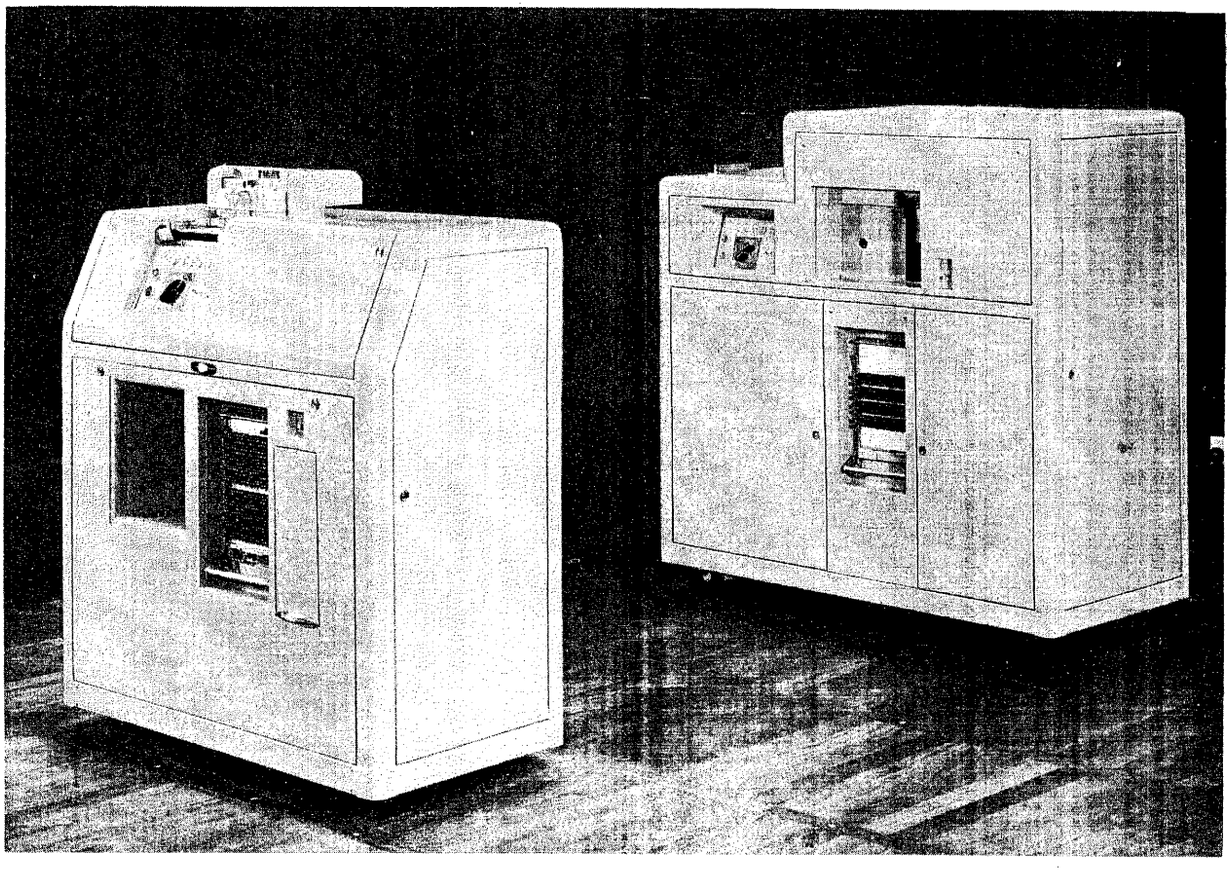


Plate 16: A card reader (left) and punch (right) attached to Pegasus 2

the course of the programme. A preset-parameter is a word which has to be incorporated in a subroutine during input of the programme; it will be punched elsewhere on the tape, normally with the master-programme. The preset-parameters are eventually put into their subroutines during the Assembly processing. The kind of parameter appropriate to a particular subroutine is something which is decided when the subroutine is written; instructions concerning parameters appear in the specification of the subroutine.

For the purposes of Assembly the subroutines in a programme must be numbered, and no two of them may bear the same number. This *routine number*, as it is called, is used to identify the subroutine; it is simply an integer between 0 and 1023 and is normally given in the specification preceded by a letter R. Thus, for example, we talk of R 240 when we mean routine number 240. Routine numbers less than 1000 are permanently allocated to Library routines according to a plan given in Appendix 7. Routine numbers between 1000 and 1023 are conventionally used for programmer's subroutines, these numbers are used only temporarily and have no permanent significance outside any particular programme. Routine numbers greater than 1023 are sometimes allocated to programmes for convenience of reference; they cannot be used with Assembly.

It is not absolutely necessary to use Assembly; indeed in earlier chapters we have described some simple complete programmes which were put together "by hand". The main Library subroutines are, however, built up round the facilities which Assembly provides and we must use Assembly if we require any of them. Many programmes are split up into a fair number of subroutines and the labour of manually inserting all the cross-references would be very heavy, whereas we can use Assembly to do the whole process automatically. Assembly also makes it simple to alter a programme; for example we can replace incorrect versions of programmer's subroutines by new versions which need not occupy the same amount of storage space; or we can change the values of preset-parameters. In fact the programme is flexible.

8.2 Cues and tags

A cue is normally an order-pair for calling in a subroutine. The cue is obeyed in the master-programme (or perhaps in another subroutine), but we cannot write it there until we know where the subroutine is stored. To take a simple example, suppose a subroutine is so written that it is to be entered by bringing its first block into *U*0 and jumping to 0.2. If this subroutine is stored in *B*12 onwards (i.e. the *address of the subroutine* is *B*12.0) then we must write the following cue in the master programme to call it in:

12 [0] 72

0.2 0 60

But if the address of the subroutine is *B*18.0 the first order of the cue has to be changed to 18 [0] 72. In general we can write the cue in the relative form

0+[0]72

0.2 0 60

provided it is understood that the relative address refers *to the subroutine* and not to the master-programme. We may *not* write or punch the cue in this form in the master-programme, because, if we did, the relativizer when the cue was read from the tape would then be that of the master-programme. If, however, the cue is punched (in its relative form) together with the subroutine then the correct relativizer will be added to it. This is what is done when Assembly is used.

In the master-programme we leave a gap for the cue by writing the integer +0 (which occupies a whole word) instead of the cue order-pair. In addition we make a note (a *call for a cue*) in a special list, indicating which subroutine is to be called in by the cue (and also which of the cues is required, since many subroutines will have more than one possible cue each). Along with each subroutine is punched its cue (or cues) in relative form, and Assembly later picks out the appropriate cue and puts it in the gap left in the master-programme. In fact the cues get *added* in but this makes no difference as a rule, since they get added to the zero stored where the cue is wanted.

The cue or cues for a particular subroutine are punched in the form of a *cue-list* accompanying the subroutine on the tape. The various cues are numbered 01, 02, 03, etc... in the order in which they appear in the cue-list.† The specification of a Library subroutine shows what each particular cue is used for. The following are two abridged specifications of Library subroutines.

▼ *R* 5 Double-length number print.
Prints (pq) as a number whose integral part and fractional part are in *X*6 and 7 respectively.

Name: D.L. PRINT Store: 7 blocks

Uses: *U*0; *B*0

Cues: 01 to precede the number by CR LF
02 to precede the number by Sp

Link: In *X*1, obeyed in 0.5

Programme-parameters: The number of digits to be printed before the point (n_1') and after it (n_2) are to be set in 4_C and 5_C respectively. If $n_1 = 0$ the fraction in *X*7 is printed; if $n_2 = 0$ the integer in *X*6 is printed (in either case the content of the other accumulator is ignored).

† The numbering system for the cues is *octal*, so that the cue following cue 07 is cue 10, the next is cue 11, and so on. This usually does not matter since few subroutines have more than two or three cues. In fact 08 and 09 have the same effect as 10 and 11 respectively.

R 200 Square root. $p' = \sqrt{(pq)}$ (fractions).

Puts into *X*6 the square root of the double-length fraction in *X*6 and 7.

Name: SQ.ROOT Store: 1 block

Uses: *U*0; *X*5, 6, 7

Cue: 01

Link: In *X*1, obeyed in 0.7

Error: Not greater than 2^{-39} (i.e. ½∈)

Note: There is a loop stop if (pq) is negative.

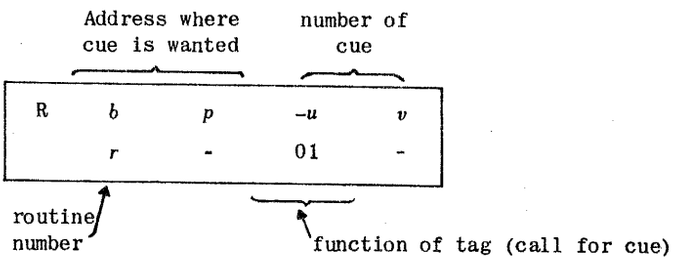
It should be noted that the actual cues themselves are not specified; we do not need to know what they are, only what they cause the subroutine to do. If, for example, we use cue 02 to R 5 then the number will be printed preceded by a space. We can do this by writing +0 in the master-programme in the place where we want the cue; we must also call for cue 02 to R 5 and specify where we want it to be added in by Assembly. This is done by a tag.

A *tag* is a pseudo order-pair preceded by the warning character R (for *reference*). Various kinds of tag are used by Assembly, but for the present we consider only *tags calling for cues.* Along with our master-programme we punch a number of tags calling for cues, one tag for each of the cues we want. Suppose, for example, that in *B*1+.4 of the master-programme (i.e. in the word in position 4 of block 1+) we want cue 02 to R 5. In this word we write +0 instead of the cue; and we must punch, at the head of the master-programme, a tag calling for cue 02 to R 5 to be added into *B*1+.4. This tag is written as follows.

R	1	4	−0	2
	5	-	01	-

In this tag we write, in the *N* and *X* parts of the "*a*-order", the *address* (here 1+.4) of the place where the cue is wanted, just as though it were an absolute address (the relative block-number is in the *N*-address and the position-number in the *X*-address). After this comes the number of the cue (here 02) preceded by a minus sign; usually the first digit of the cue number is 0 so that we write −0 in the F part of the *a*-order and the second digit of the cue number in the *M* part. In the *b*-order of the tag we put the routine number (5 in this tag) in the *N* column. The rest of the *b*-order is 01 in the *F* part with minus signs on either side of it (in the *X* and *M* parts); this 01 is called the *function of the tag* and shows that this is a tag calling for a cue. This tag serves only to give information to Assembly; it is *never obeyed* as an order-pair.

In general a tag calling for cue number *u v* to the subroutine with routine number *r* is written thus,



The cue is to be added into the word in position *p* of block *b*+ of the master-programme; this word is called the *tagged word,* and since it is normally +0 it will in effect become the cue when Assembly has processed the programme. As a rule there will be quite a number of these tags punched in front of the master-programme proper; there will be one tag for every place where a cue is required (several tags may refer to the same cue, of course, if this cue is wanted in several places). These tags may be punched in any order. As they are read in they are automatically stored directly in Assembly's working space in the high-numbered locations in the main store; they are *not* stored with the rest of the programme. In fact as soon as a tag has been read in it has the Transfer Address added to it (in the modifier-position) and the resulting word is then stored in the place specified by the *Reference Address,* instead of the Transfer Address (which is left unaltered). The Reference Address (abbreviated to R.A.) is set initially to *B*882.7 and is *decreased* by 1 every time a tag is read.† Thus the tags are stored backwards in the upper reaches of the main store without disturbing in any way the programme they accompany, which gets stored (forwards) in the lower end of the store as usual. The Reference Address is held during input of the programme in the modifier part of *U*5.5 and can be inspected there on the monitor tube. The fact that the T.A. gets added into each tag has the effect of converting the written address (*b.p*) into the *absolute* address of the tagged word, i.e. the address of the word where the cue is wanted. This is because the T.A. remains constant while the tags are read; and, since they are all punched before the master-programme, the T.A. will be the address of the beginning of the master-programme.

The directives A1, A2 and A3 play an important role when we use Assembly; L-directives also have a special use. The A1-*directive* is punched at the beginning of the programme tape and signifies that we intend to use Assembly. This directive must be punched before the first tag, and is normally put just after the D and N at the head of the tape. Next will follow all the tags calling for cues, and then the master-programme, which must be followed by an L-directive. Assuming for the moment that we need only Library subroutines then the next item on the tape is an *A2-directive,* which causes the

† The Reference Address is set initially to *B*506.0 by the 4096-word store Assembly.

Initial Orders to start reading the Library tape in the second tape-reader. The next item on the main programme tape is an *A3-directive*, which is read when all the Library subroutines needed have been found and stored; this directive causes Assembly to process the programme, i.e. to examine all the tags and insert the cues where required in the master-programme. The last item on the tape is normally an E-directive such as E 2.0 to enter the programme. The make-up of the programme tape is therefore as follows.

```
D                          <------  date and serial number
N                          <------  print name of programme
name of programme
A1                         <------  prepare to use Assembly
tags calling for cues
master-programme
L                          <------  end of master-programme
A2                         <------  read the Library tape
A3                         <------  process the programme
E 2.0                      <------  start obeying the programme
```

To illustrate this we shall now describe a complete programme using Assembly. This programme tabulates the square roots of all the integers from 1 to 100; it uses R 5 to print the integers and the square roots, and R 200 to evaluate the square roots. Abridged specifications of these two Library subroutines are given above. The master-programme is given in full below.

The integer whose square root is currently being evaluated is denoted by n and is held in $X2$. The output of the programme consists of two columns of figures, the first giving n and the second $\sqrt{n}$; the lines are grouped into blocks of five lines, and a counter is kept in $X3$ for this. The square root subroutine R 200 evaluates the square root of the double-length fraction (pq). We use it by setting this fraction equal to $n/256$, which is always in range; the answer is then $(\sqrt{n})/16$ and is multiplied by 16 before printing.

In the master-programme there are three places where cues are wanted (0+.5, 1+.0 and 1+.4). The integer +0 is written in each of these places and there are three tags calling for cues at the head of the master-programme. All the necessary directives are included.

All this programme is punched on one tape. The course of events when the programme is run is as follows. First we put the beginning of the programme tape in the main tape-reader and the beginning of the Library tape in the second tape-reader and do a Normal Start operation. The T.A., relativizer and R.A. are set to their starting values and the computer starts to read the programme tape. The D-directive causes the date and serial number to be printed and the N-directive causes the name to be printed. Next the A1-directive is read and there is optional printing of A1; the first part of Assembly is then entered, which causes a few counters and modifiers to be set in Assembly's working space (actually in $B895$†) and a *special link* to be placed in $B0.1$ (ready for subsequent L-directives). The Reference Address is then set to its starting value of 882.7, but with this programme this operation is actually redundant. Next a *B-operation* occurs, i.e. the T.A. is set to the beginning of a block and the relativizer is put equal to the block-number. Assembly then calls in the rest of the Initial Orders as a subroutine to read the tape (every time an L is read Assembly is re-entered). The next items on the tape are the three tags calling for cues; the relative addresses in these tags are converted into absolute ones by having the *Transfer Address* (here 2.0) added to them (note that the relativizer is not used for this). The tags are stored in $B882.7$, 882.6 and 882.5 respectively, and the T.A. is left untouched at 2.0. Then the master-programme is read in and placed in $B2$, $B3$ and a part of $B4$; the L-directive at the end causes Assembly's special link (in $B0.1$) to be obeyed and Assembly is re-entered. At this stage Assembly examines each of the tags to determine which subroutines are required; it builds up an *Index* (starting at $B883.0$) containing one word for each subroutine called for†† #. Next there is a B-operation, which here has the effect of moving the T.A. forward to 5.0 and setting the relativizer equal to 5 (there is no optional printing), and Input is called in again to read more tape.

The next item on the tape is the A2-directive, which has the effect of changing over to the second tape-reader, where the Library tape is waiting. This is the only effect of an A2-directive, apart from optional printing of A2. At the beginning of the Library tape is an N-directive to identify the tape. After this comes the subroutines, each preceded by its cue-list, which includes a certain tag identifying the subroutine. By inspecting this tag and searching through its index Assembly can determine whether or not the subroutine is one of those which were called for in the programme. If the subroutine is not wanted it is rejected and Assembly passes on to the next subroutine on the tape. When a wanted subroutine is found Assembly first copies its cue-list up into the top end of the main store (among the tags) and then reads the subroutine in. In this programme the first accepted subroutine to be read from the Library tape gets put into $B5.0$. Assembly then scans its index to determine whether any more subroutines are required and, if so, it does a B-operation and returns to read in more of the Library tape. Ultimately all the wanted subroutines will have been found and Assembly then returns to the main tape-reader (after a B-operation); the rest of the Library tape is not read.

The next item to be read is the A3-directive on the programme tape, which causes the second part of Assembly to be entered after optional printing of A3. At this stage the master-programme and all the subroutines have been read in. Assembly now examines every tag and uses certain information built up in its index during input. When each call for a cue is found the appropriate cue is extracted from the

---

† $B511$ in the 4096-word store.

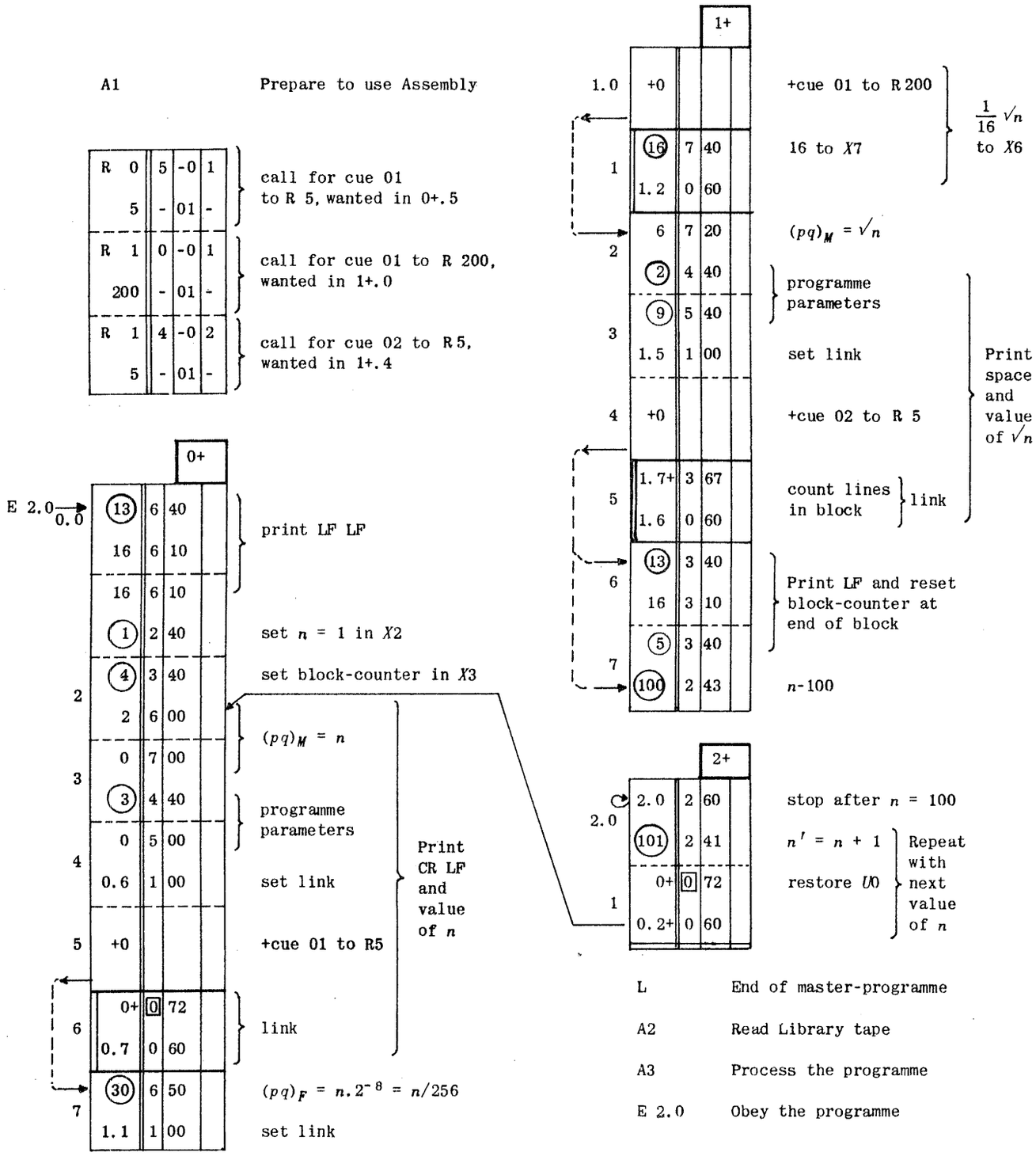
†† In the 4096-word store, the Reference Address is set to a starting value of 506.7, and the Assembly index begins in $B507.0$.

The tags are also changed slightly, the routine number being replaced by the address of the index-word.

D

N

TABULATE SQUARE ROOTS OF n = 1(1)100



cue list for the specified subroutine and added into the tagged word in the master-programme (overflow is disregarded in this process). In this way all the cues get put in. When all this processing is complete there is some optional printing which shows where the various subroutines have been put and gives a little extra information. There is then a B-operation and more tape is read in. The next item on the tape is the directive E 2.0, which causes the programme to be entered in the usual way.

It will be noted that Assembly always does a B-operation whenever it calls in Input to read more tape. There is consequently no need to use any B-directives when we are using Assembly.

The main advantages which result from using Assembly in the way just described are as follows.

(a) We do not need to select the tapes for Library subroutines or to feed them into the computer individually, or to join them on to our own tapes.

(b) We do not have to specify in advance (or, indeed, at all) where the subroutines are to be stored.

(c) The cues are inserted automatically.

There are other advantages in using Assembly, which will become clear later.

The subroutines will all be stored head-to-tail after the master-programme, each one beginning a new block. It is easy to find the total amount of storage space occupied by the programme, when it has been written, by simply adding up the requirements for each subroutine as given in its specification. (If this is too much trouble we can always punch a B-directive after the A3 on the programme tape; this will will cause optional printing of the final value of the T.A., i.e. the address of the start of the next free block after the programme.) The Library subroutines are all written on the assumption that they are placed in the first 128 blocks of the store; if an attempt is made to place any block of such a subroutine beyond *B*127 then a relative address in the subroutine will overflow.

8.3 Cue-lists and programmer's subroutines

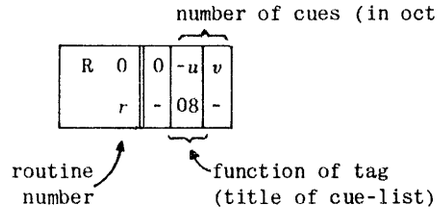
The subdivision of a complete programme into a master-programme and subroutines introduces a great amount of flexibility and makes it possible for several people to work simultaneously on different parts of the programme or, in general, for one part to be written without close attention to the other parts. We shall now describe how a subroutine may be drawn up to use the facilities of Assembly. We assume for the present that it is a programmer's subroutine, i.e. one required in some particular programme and not likely to be needed elsewhere. In the programme it will consequently be given a routine number between 1000 and 1023.

We first of all write the subroutine proper, i.e. the orders and constants which will ultimately go into the programme. This is done in exactly the same way as if Assembly were not going to be used, but all references to the main store must be relative; in other words we number the blocks of the subroutine itself 0+, 1+, etc. We disregard at present the possibility of using preset-parameters, which does affect the way the subroutine is written; this subject is discussed in Sec. 8.4. When the subroutine itself has been prepared we can specify what its cues are (in relative form), and we can draw up the cue-list for the subroutine. This is simply a list of all the possible cues arranged in some convenient order, which must be adhered to throughout. The cues themselves are numbered 01, 02, 03, ..., as described above, in the order in which they are written. Suppose the last cue is numbered 03 and the subroutine has been allotted the routine number R 1009, then the following tag, called the *title of the cue-list* must be written before the first cue.

R 0	0	-0	3
1009	-	08	-

This tag should be compared with those given earlier, which were tags calling for cues. The function of this tag is 08 (written in the *F* part of the *b*-order) and indicates to Assembly that this is the title of a cue-list. The routine number is written, as before, in the *N* part of the *b*-order, and the number of the last cue (i.e. the number of cues, in octal) is written in the *F* and *M* parts of the *a*-order. The address part of the tag is always zero (in the *N* and *X* parts of the *a*-order); minus signs are written in the same position as before.

In general the title of the cue-list containing $8u + v$ cues for routine number r is written thus.



This tag is punched immediately before the first cue in the cue-list. In practice most subroutines have only a few cues and we seldom have to specify any octal number of cues greater than say 04, in fact many subroutines have only one cue. In principle the number of cues may be as great as 63 (i.e. 77 in octal) but this is of rather academic interest.

The tape for the subroutine is made up of the cue-list (headed by its title), which is followed by an L-directive. Then comes an N-directive† and the name of the subroutine (if required). This is followed by the subroutine proper and another L-directive, thus:

Title of cue-list

Cue-list

L N

Name of subroutine

Subroutine proper

L

We can illustrate this by an example of a programmer's subroutine to evaluate a simple function. At a certain stage in a programme suppose we wish to evaluate one of the functions

† This, if present, must be separated from the L-directive preceding it by blank tape *only*; no CR LF is allowed. This is discussed below and in Sec 8.5.

$$f(x) = \frac{x^2 + 0.327x + 0.113}{x + 0.141}$$

or

$$g(x) = \frac{x^2 - 0.327x + 0.113}{x + 0.141}$$

Since these functions are very similar we can write a single subroutine which will evaluate either of them. Here x is a positive fraction, less than ½, in $X6$ and the function is to be left in $X6$ by the subroutine. There are two entry-points to the subroutine, each with its own cue, so that we may evaluate either $f(x)$ or $g(x)$ as we please. The subroutine may be written and punched as follows.

```
        R  0    0  -0  2      } Title, indicating
        1003    -  08  -      } two cues to R 1003.
    - - - - - - - - - - - -
        0+      0  72         } Cue 01, to
01                            } evaluate f(x).
        0.3     0  60         }
    - - - - - - - - - - - -
        0+      0  72         } Cue 02, to
02                            } evaluate g(x).
        0.7     0  60         }
                                                    } Cue-list
                                                    } (followed by L)
        L  N                                        }
                                                    } Name of routine
        FUNCTIONS F AND G                           }
```

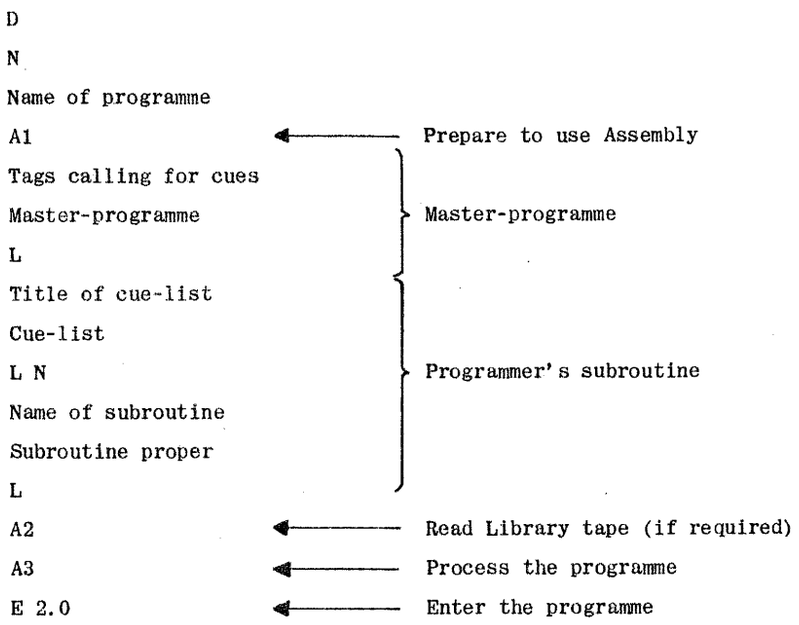
```
                0+                                  } Coefficients
0.0     +.141                                       }
    - - - - - - - - - - - -
1       +.327                                       }
    - - - - - - - - - - - -
2       +.113                                       }
    - - - - - - - - - - - -
01 ->   0.1     5  00       +.327 to X5             } Subroutine proper
3                                                   } (followed by L)
        6       5  01       x + .327 to X5          }
    - - - - - - - - - - - -
        0.0     6  11       x + .141 to 0.0         }
4                                                   }
        5       6  20       x² + .327x to X6 and 7  }
    - - - - - - - - - - - -
        0.2     6  01       add .113                }
5                                                   }
        0.0     6  25       divide by x + .141      }
    - - - - - - - - - - - -
        7       6  00       quotient to X6          }
6                                                   }
        0.7     1  10       plant and obey link     }
    - - - - - - - - - - - -
02 ->   0.1     5  02       -.327 to X5             }
7                                                   }
        0.3+    0  60                               }
        L
```

Note that the cues are written in the cue-list in relative form, the addresses in them get converted into the correct absolute form when they are read in because the relativizer will be added to them. If the master-programme obeys cue 01 to this subroutine then x in $X6$ will be replaced by $f(x)$ and the link in $X1$ is obeyed; if cue 02 is used $g(x)$ will be evaluated instead. These cues can be obtained by writing the appropriate tags calling for them at the head of the master-programme.

Both the cue-list and the subroutine proper must be terminated by L-directives. The reason for this should become clear when we consider what happens when this subroutine is read in, which we shall do shortly. We shall first describe how the programme tape is made up for a programme consisting of

(a) a master-programme,
(b) the above programmer's subroutine,
(c) some Library subroutines.

We write the master-programme in the way described in Sec. 8.2, writing +0 in place of each cue. The tags calling for cues at the head of the master-programme will now include a few referring to R 1003, i.e. to the above subroutine, as well as those referring to Library subroutines. The whole programme tape is made up as before, but the programmer's subroutine is included after the master-programme and before the A2 on the tape. The programme tape may be described as follows.



If there are several programmer's subroutines they are each made up in much the same way and are inserted (in any order) between the master-programme and the A2-directive. If no Library subroutines are required the A2-directive should be omitted entirely.

When this programme is read in, the sequence of events is as follows. First, the T.A., relativizer and R.A. are set to their starting values by a Normal Start operation and the D and N directives cause their appropriate printing. Next the A1-directive sets the counters and modifiers required to start Assembly's operations, sets the link in *B*0.1 for later L-directives, does a B-operation and returns control to Input to read in more tape. The tags calling for cues are next read in and placed in the upper end of the main store as determined by the Reference Address. The master-programme goes, as usual, into *B*2 onwards and the L-directive is then read. Here Assembly examines all the tags read since the A1-directive and builds up its index of the subroutines required. This index will contain an entry relating to the programmer's subroutine. Assembly then does a B-operation and returns to read more tape.

The next item on the tape is the tag which is the title of the cue-list, and as usual this goes into the place specified by the R.A., leaving the T.A. untouched. The cues in the cue-list are then read in; the first of them will occupy the beginning of a block (because of the B-operation at the end of the master-programme) and the others follow it in the order in which they are punched. The L-directive at the end of the cue-list is then read. Assembly now inspects the last tag and determines that it is the title of the cue-list to R 1003; the entry in Assembly's index is thereupon amended so as to record the fact that the routine has been found. The whole of the cue-list is then *copied* into a place determined by the R.A. (and the number of cues in the list), so that it is now up amongst the tags in the higher-numbered locations of the main store. The *T.A. is then put back* to the value it originally had before the cue-list was read in, and Input is called in to read the subroutine (after a redundant B-operation). The name of the subroutine is printed and the subroutine proper is then put into the main store; its first few order-pairs will *overwrite the cue-list* in its original locations; this does not matter because the cue-list has been safely copied away for later use. When assessing the amount of storage space occupied by a subroutine we need not allow anything for its cue-list. The L-directive at the end of the subroutine causes Assembly to note that input of the subroutine is finished; a B-operation occurs and more tape is read in. The A2 and A3 directives have the effects described earlier.

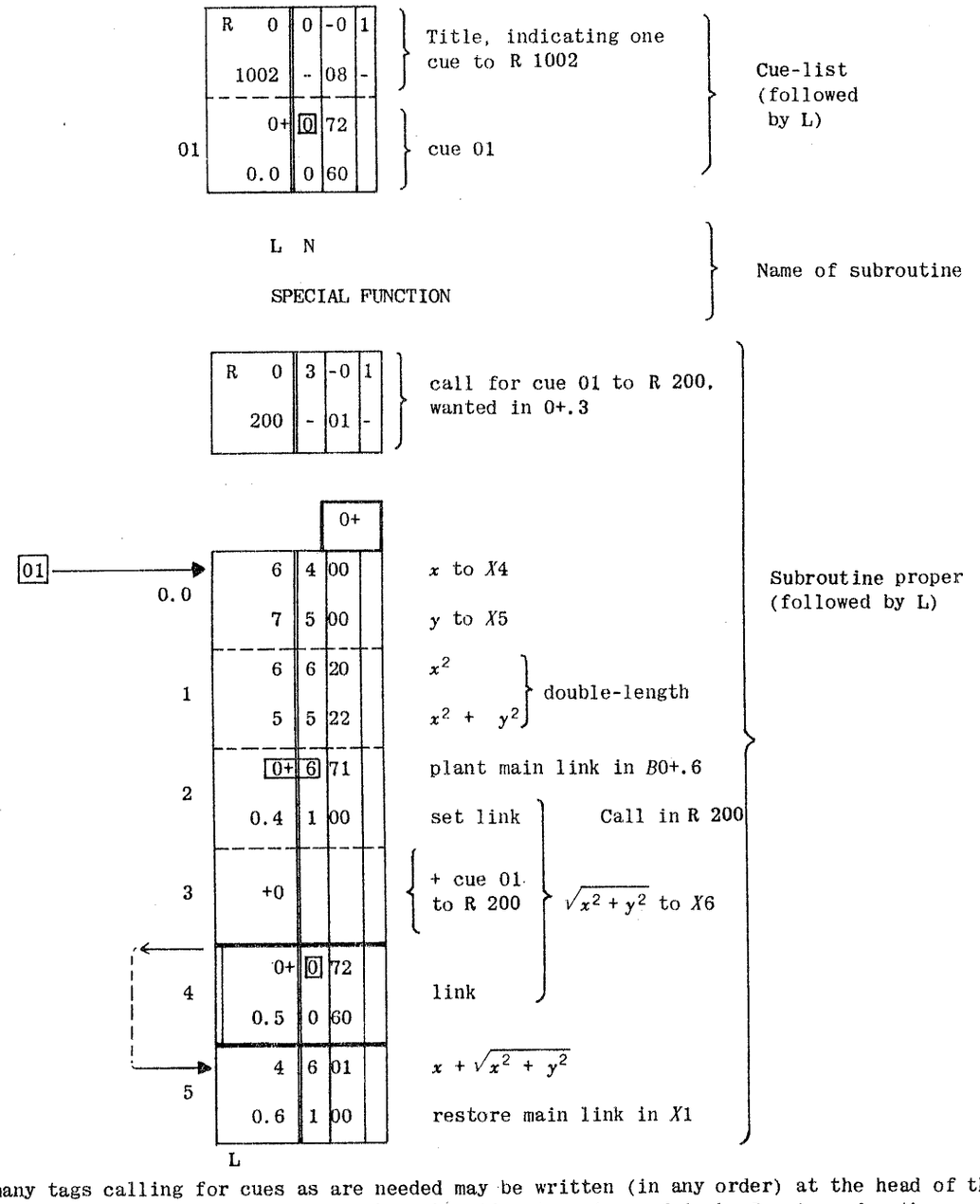
A Library subroutine is punched in the same way as a programmer's subroutine with only one important difference; the function part of the tag which is the title of its cue-list is 28 instead of 08. This is because of the slightly different treatment accorded to Library subroutines. Assembly assumes when it reads a programmer's subroutine that this subroutine is wanted, and it therefore always accepts it. A Library subroutine, on the other hand, is accepted only if a call for a cue to it has been read in. However, Assembly never accepts two routines with the same number. Assembly determines whether to accept a subroutine or to reject it when the L is read at the end of the cue-list. At this point Assembly inspects the last tag, which identifies, by the routine number, both the cue-list and the immediately following subroutine (not yet read in). If the subroutine is to be accepted then the cue-list is copied away and the subroutine is read in as described above. If on the other hand the subroutine is to be rejected then it is read in but the T.A., relativizer, and R.A. are reset at the end of the subroutine to the values they had before the cue-list was read; the subroutine is therefore overwritten by the next one on the tape.

The name of a subroutine is punched, with its N, immediately after the cue-list and is treated in a special way. The name is printed only if the subroutine is accepted and if optional printing is allowed to occur (*H*0 = 0). If optional printing is suppressed, or if the subroutine is rejected then the part of the tape carrying the name is read and ignored by Assembly itself (this is the only occasion when Assembly reads tape; it otherwise uses the rest of the Initial Orders, chiefly Input).

Not infrequently a subroutine may itself require to call in other subroutines, and we then write tags calling for cues at the head of the subroutine proper. An example should make this clear. Consider a programmer's subroutine to evaluate the function

$$x + \sqrt{(x^2 + y^2)}$$

from values of x and y given in $X6$ and 7 respectively, the result to be left in $X6$. We need the Library subroutine R 200 to evaluate the square root. We first of all write out the subroutine in the ordinary way but putting +0 instead of the cue to R 200; we then put a tag calling for the cue at the head of the subroutine and put an L at the end. The construction so far is just like that of the master-programme. We now write the cue-list and its title at the top, above the tag calling for the cue to R 200. This is shown in the subroutine as given below, in which the routine number is R 1002. This subroutine has only one cue, which is therefore referred to as cue 01; and, as usual, the N-directive giving the name of the subroutine is punched immediately after the cue-list.



As many tags calling for cues as are needed may be written (in any order) at the head of the subroutine proper, as shown above. When the subroutine tape is read in by Input, under the control of Assembly, the tag calling for the cue to R 200 (and any other similar tags) will be stored in the place specified by the Reference Address, and not with the subroutine. When the L at the end of the subroutine is read, Assembly examines each of these tags and extends, if necessary, its index of wanted subroutines. When the above subroutine is read, for example, Assembly will record the fact that R 200 is now needed in the programme, even if it was not referred to by any tags associated with the master-programme. During the processing of the programme caused by the A3-directive the cue to R 200 will be inserted in the programmer's subroutine.

8.4 Preset-parameters

Many subroutines can be made more versatile if they are made to use parameters. The detailed operation of a subroutine can be made to depend on some constant or number, whose value is not fixed when the subroutine is written. Parameters are of particular value in the construction of a Library of subroutines because the increased flexibility they provide enables the total number of subroutines

needed to be reduced; a single subroutine can do the work of several. A programmer's subroutine is written for a particular programme and there is sometimes less need to make it readily adaptable. The subject of programme-parameters has already been discussed (in Sec. 4.2); these are usually integers placed in accumulators by the master-programme just before obeying the cue to the subroutine. Programme-parameters have the advantage that they can be readily changed during the course of the programme. Sometimes this advantage is not wanted and it then becomes a nuisance to have to set the parameters every time the subroutine is called in. Consequently some subroutines are written to use preset-parameters, whose values are fixed for any particular programme but may be changed from one programme to another. We shall first describe how preset-parameters are used with Library subroutines.

The preset-parameters (if any) for a particular subroutine are grouped together in the form of a *parameter-list*, at the head of which is punched a special tag called the *title of the parameter-list*. The specification of any Library subroutine requiring such a parameter-list always gives this tag in full and explains how the parameter-list is to be drawn up. The parameter-list and its title are normally punched as a *part of the master-programme*, but they may appear elsewhere if convenient.

For example, there is in the Library a subroutine for the evaluation of natural logarithms which has the routine number R 224. This subroutine works out a scaled down value of the logarithm of the double-length fraction (pq). This fraction must be positive and is, of course, less than 1; its logarithm is consequently negative. If the fraction is very small its logarithm is large and negative (e.g. if $(pq) = 1/20$ the logarithm is about −3; if $(pq) = 1/400$ the logarithm is about −6). To prevent overflow the subroutine consequently evaluates

$$2^{-n} \log_e (pq),$$

i.e. the logarithm divided by 2^n, where n is chosen to prevent overflow. If we make n large there is no possibility of overflow but the logarithm is scaled down heavily and precision will be lost if (pq) is not small. If we know that (pq) is not going to be small in a particular programme then we should choose as small a value of n as possible. The subroutine has consequently been written so that n is determined by a parameter-list (containing one preset-parameter); and the specification† gives this parameter-list as follows.

	R 0 224	0 -	-0 04	1 -	Title, indicating one parameter for R 224.
01	n 0	0	00	0.	Preset-parameter 01

Parameter-list for R 224

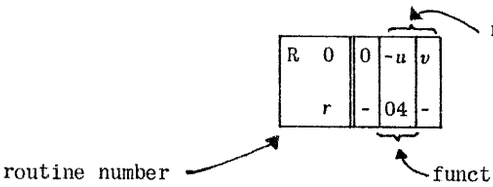
When we wish to use R 224 we simply have to copy this parameter-list into our master-programme, exactly as it is given here, except that we must of course put in the value of n we require. Note that we do not put an L at the end of a parameter-list (unless, of course, the parameter-list is at the end of the master-programme, when it will be followed by the L terminating the master-programme). When the parameter-list is read in, together with the master-programme, the title, being a tag, will be stored in the place specified by the R.A., along with the other tags; but the parameters themselves will occupy space in the ordinary part of the main store - in other words, they remain where they are put originally. They are *not* copied away elsewhere like a cue-list. Note that a parameter-list may be put anywhere into the main store; there is no need for it to start a new block.

In a parameter-list the preset-parameters are referred to by the numbers 01, 02, 03, etc... in the same way as the cues (we sometimes abbreviate "preset-parameter 02" to PP02, and so on). In the title of the parameter-list the function (in the F part of the "b-order") is 04; the routine number is written in the usual place and the number of the last parameter (i.e. the octal number of parameters) is written in the top right-hand corner. The address in the tag is given as zero above, and in all titles shown in Library specifications; this means simply that the parameter-list follows immediately after the tag (the T.A. is added to the written address before the tag is stored).

If desired the tag can be written with all the tags calling for cues at the head of the master-programme, leaving the parameter-list somewhere later on the tape; if this is done the correct (relative) address of the first parameter should be written in the address part of the tag (just as we write the address of the tagged word in a tag calling for a cue - in fact in a parameter-list the first preset-parameter is the tagged word).

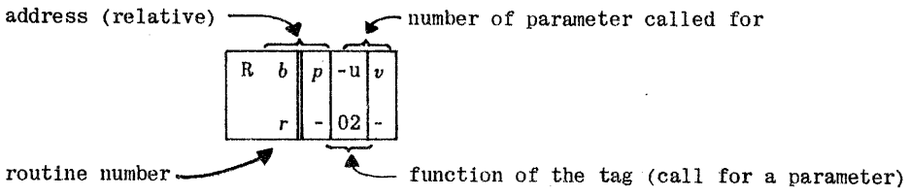
In the subroutine there are *tags calling for parameters*, which resemble tags calling for cues, except that the function used is 02 instead of 01. During input of the programme tape Assembly notes in the index-word of a subroutine the address of any parameter-list found; it derives this address from the title of the parameter-list. When the programme is processed (i.e. after the A3-directive has been read) Assembly deals with a call for a parameter in much the same way as a call for a cue, extracting the parameter wanted from the appropriate parameter-list and adding it into the tagged word in the subroutine (disregarding overflow).

In general the tag forming the title of the parameter-list to routine number r is written, if the list contains $8u + v$ preset-parameters and is immediately preceded by the title, as follows.



† An abridged specification of R 224 is given on page 181.

And a tag calling for the addition of preset-parameter number *uv* in the parameter-list of routine number *r* is written as follows.



The address in this tag is that of the word which is to have the parameter added to it; like all the addresses in tags it is a relative address. There may be any number of calls for parameters but there is only one effective parameter-list for a routine. If more than one parameter-list is supplied for some routine then the last to be read in will be used.

Preset-parameters may be applied just as well to programmer's subroutines as to Library subroutines. The parameter-list normally forms part of the master-programme though it may be read in at some other stage if this is desirable. The tags calling for parameters will normally appear at the head of the subroutine proper, that is after the cue-list and the name of the subroutine.

To illustrate these points we shall construct a rather trivial programmer's subroutine to evaluate the fraction

$$2^{-n}\sqrt{(p^2 + q^2)},$$

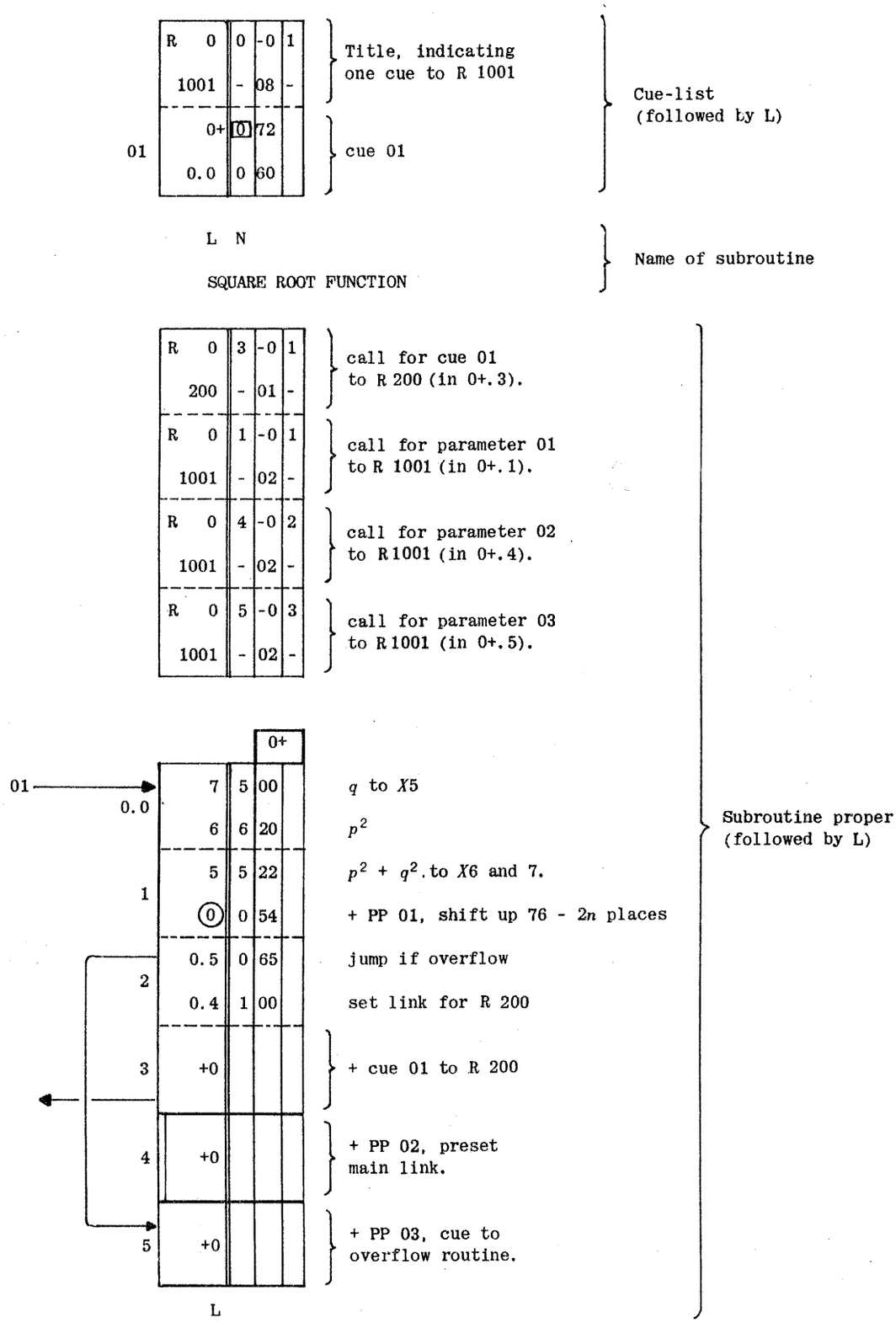
where n is specified by a preset-parameter, and p and q are integers in $X6$ and 7 on entry; the result is to be left in $X6$. We can suppose that n is left unspecified when the subroutine is written so that it can be made as small as possible (without overflow occurring) to suit the requirements of the problem. For example if p and q are both less than 100 in absolute value we can safely set $n = 8$; but if p and q are allowed to be as large as 1000 we must set $n = 11$ to prevent overflow. We shall further assume that the subroutine is called in at only one point in the master-programme, so that there is no need to adopt the usual technique of setting a link in $X1$; instead, we shall write the subroutine so that when it has done its work it obeys a *preset link*, i.e. a link which is built into the subroutine and is not changed. This link will be another preset-parameter, as we do not wish to specify it before writing the subroutine. We shall also arrange that, if overflow should occur during the subroutine (because n was made too small), then another order-pair is obeyed, which can call in an *overflow-routine*. We do this because we do not wish to specify in advance what action is to be taken if overflow should occur. This order-pair is another preset-parameter (it can be called a cue to the overflow-routine).

The subroutine is given in full below; it has been given the routine number R 1001. The cue-list contains only one cue and is, as always, terminated by L. Then comes the N-directive including the name of the routine. This is followed by four tags, one calling for the cue to R 200 to evaluate the square root, and the other three calling for the parameters to be added to certain words in the subroutine, which comes next. Preset-parameter 01 is of the following form.

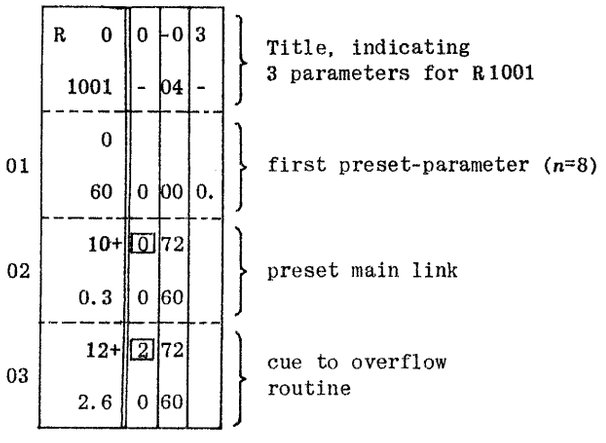
0			
76-2*n*	0	00	0.

This specifies the value of n; when it is added to the order-pair in 0+.1 of the subroutine it inserts the required shift number in the 54-order so that it will shift up $p^2 + q^2$ by $76-2n$ places. We have to do this because p and q are integers and the answer is to be a fraction (p'_F). Let us write p_F and q_F for the fractions ϵp and ϵq, then

$$\begin{aligned} p'_F &= 2^{-n}\sqrt{(p^2 + q^2)} = \sqrt{\{2^{-2n}\ (p^2 + q^2)\}}, \\ &= \sqrt{2^{-2n}\{(2^{38}\ p_F)^2 + (2^{38}\ q_F)^2\}}, \\ &= \sqrt{\{2^{76-2n}\ (p_F^2 + q_F^2)\}}. \end{aligned}$$



The following is a typical parameter-list for this subroutine. The value of n is 8, which determines preset-parameter 01. When the subroutine has evaluated the required result, it returns to the master-programme by transferring its block 10+ to $U0$ and jumping to 0.3; this is done by preset-parameter 02 (the main link). Preset-parameter 03 is obeyed if overflow occurs in the subroutine; it transfers block 12+ of the master-programme to $U2$ and jumps to 2.6.



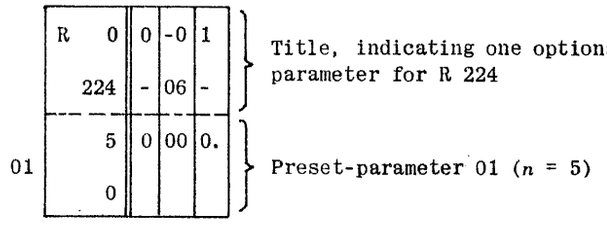
There are a few points to note here. Preset-parameter 01 is a pseudo order-pair which has zeros everywhere except in the *N*-address of the *b*-order (its stop/go digit is 0 because it is marked as a stop order-pair); when Assembly adds it to the order-pair in 0+.1 of the subroutine it has the effect of putting the *N*-address in the *b*-order. This order-pair will read as follows after the Assembly processing:

5 5 22

(60) 0 54

The parameter-list is punched as a part of the master-programme; this means that when it is read in the relativizer will be that of the *master-programme* and the parameters 02 and 03 will have the correct main store addresses in their 72-orders. The overflow routine is simply a part of the master-programme. Notice how a parameter-list allows us to write and punch a word with the master-programme for later insertion into a subroutine. A cue, on the other hand, is punched with a subroutine but is later put into the master-programme. By these two devices Assembly allows in effect cross-references from one routine to another.

When writing a subroutine we may wish to provide the facility of preset-parameters, and yet we may be able to supply values for every parameter which will probably be those chosen by the user of the subroutine. Here an *optional parameter-list* can be useful. An optional parameter-list is one which is used by Assembly only if no other parameter-list is supplied. For example, the logarithm subroutine R 224, described above, is accompanied on the Library tape by an optional parameter-list which has $n = 5$. This fact is mentioned in the specification. If the user of the subroutine does not provide a parameter-list of his own then Assembly will use the optional list and set n equal to 5. An optional parameter-list is written in the same way as an ordinary parameter-list except that the tag which is the title of the optional list has the function 06 instead of 04. Thus the optional parameter-list supplied with R 224 would be written as follows.



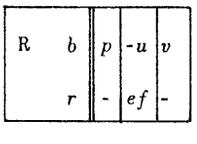
Note that in an optional parameter-list all the parameters must be given values; it is not possible to specify values for only a few of them.

Since a parameter-list is always stored along with the programme with which it is punched, allowance must be made for the storage space occupied by an optional list, whether it is used or not. The storage space required by a Library subroutine is given in the specification (under the heading *Store*). Since the L at the end of the subroutine always causes Assembly to do a B-operation we must allow a whole number of blocks for the subroutine, including any partially filled blocks occupied by the optional parameter-list.

8.5 The preparation of a programme for use with Assembly

We shall now collect together some of the information given in earlier sections of this chapter, add a little new matter and set it all out systematically. We shall also indicate how the directives and tags used with Assembly are punched.

A general tag for use with Assembly may be written as follows.



This tag is a *go* pseudo order-pair preceded by a letter R. Its component parts are as follows:

(a) *b.p* is the relative *address* of the tagged word,
(b) *uv* is an octal number ($8u + v$),
(c) *r* is the *routine number* of the subroutine referred to,
(d) *ef* is the *function* of the tag.

The various *functions (ef)* in Assembly tags are as follows; all these refer to the subroutine with routine number *r*:

01 Call for cue number *uv*,

02 call for preset-parameter number *uv*,

04 title of a parameter-list containing $8u + v$ parameters,

06 title of an optional parameter-list,

08 title of the cue-list for a programmer's subroutine, containing $8u + v$ cues,

28 title of the cue-list for a Library subroutine.

When the tag is read (by *Input*) the address has the T.A. added to it; the resulting word is then stored in the location specified by the Reference Address (R.A.), which is then reduced by 1; the T.A. is not changed. The address *b.p* is therefore an address relative to the present value of the T.A. (and *not* to the relativizer). Suppose, for example, that the following tag is read from the tape when the T.A. is 12.4, the relativizer is 2, and the R.A. is 880.3:-

R 1 7 -0 3
1000 - 01 -

Before the tag is stored the T.A. (12.4) is added to its address (1.7) to give 14.3. The following pseudo order-pair is therefore stored in *B*880.3:-

14 3 -0 3
1000 - 01 -

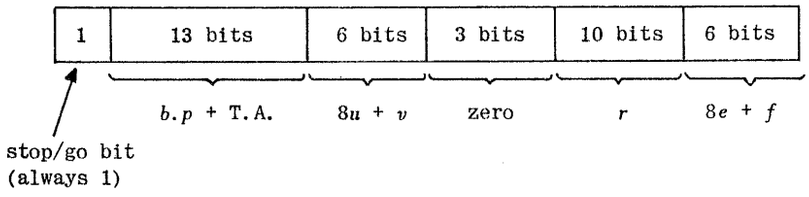
After this has been done the R.A. is reduced to 880.2 but the T.A. is still 12.4. The relativizer plays no part. This particular tag will cause Assembly to add cue 03 for R 1000 into the word in *B*14.3, which is the tagged word. The addresses in the various kinds of tag give the following information to Assembly:

01 to 02 Address of word to which a cue or parameter is to be added.

04 or 06 Address of the first parameter in a parameter-list.

08 or 28 Address of a subroutine (and also of the first cue before the cue-list is copied away).

The component parts of the tag word are stored as follows:



It is from this stored form of the tag that Assembly derives all its information. During input of the programme Assembly examines the tags whenever an L is read; the information is then used to build up, or to extend or amend, an *index* of subroutines. Each entry in this index is a single word containing the essential information relating to one subroutine. When the tag is examined the routine number *r* is replaced by the address of the corresponding index-word. The index-word contains the routine number, the address of the cue-list (when copied away), the address of the parameter-list (if any), and a record of the functions of the tags relating to the subroutine (see Sec. 8.9). The index may not contain more than 97 entries; consequently no programme may include more than 97 subroutines.†

The usual tags occurring in a master-programme are those calling for cues and those which are titles of parameter-lists. The tags calling for cues are normally all grouped (in arbitrary order) at the head of the master-programme, in which case the address in each tag is simply the relative address of the word which is to have the cue added to it††. The titles of parameter-lists are usually written immediately before the first preset-parameter in the list, when the address in the tag should be zero. Alternatively the title tag may be written at the head of the master-programme with the other tags, when the address in the tag should be the relative address of the first parameter. An L-directive must be punched at the end of the master programme.

---

† The index may contain not more than 34 entries in the 4096-word store.

†† An alternative scheme is to write each tag calling for a cue immediately in front of the place where the cue is required; if this is done the address part of the tag should be zero.

A subroutine is always immediately preceded by its cue-list, which is headed by its title tag (no other tags are allowed in a cue-list) and followed by an L. The address in the title tag should always be zero; and the function is 08 in a programmer's subroutine or 28 in a Library subroutine. The subroutine proper is terminated by L; it may contain tags calling for cues or for parameters or the titles of parameter-lists; these are all written in much the same way as tags in the master-programme. The name of the subroutine is punched immediately after the L following the cue-list; this name is printed optionally (if $H0 = 0$) provided the subroutine is accepted.

The tape for a programme starts with D and N directives and the special directive A1, which calls in Assembly to do certain preliminary operations, including setting the R.A. to its starting value of 882.7. This directive always causes a B-operation to occur before more tape is read. It must appear on the tape before the first tag.

The A2-directive is punched after the last programmer's subroutine, or after the master-programme if there are no such subroutines; it causes the Library tape to be read in on the second tape-reader, after optional printing of A2. This directive is not restricted to programmes using Assembly, its only effect being to switch to the second tape-reader. On a normal programme tape (using Assembly) the A2 can be thought of as representing all necessary Library subroutines.

When the last Library subroutine needed for the programme has been read from the Library tape, Assembly returns to the main tape-reader where there is normally the A3-directive to initiate the processing of the programme. Assembly now examines all the tags. If a tag is a call for a cue then the index-word is used to direct Assembly to the place where the cue-list is stored (among the tags), and the appropriate cue is extracted and added into the tagged word (overflow being disregarded). A call for a parameter is treated in much the same way. The titles of parameter-lists are ignored at this stage as the necessary information (i.e. the address of the parameter-list) is contained in the index word. The title of a cue-list causes Assembly simply to avoid the following cues. (This examination of the tags is done in a forward direction through the store i.e. starting with the *last* tag read in.) The optional printing associated with the A3-directive is very useful; it is as follows.

(a) As soon as the A3 is read there is optional printing of A3.

(b) When the processing is complete there is a line of printing for each subroutine accepted; this gives

(i) the routine number,
(ii) the address of the subroutine,
(iii) the address of the cue-list,
(iv) the address of the parameter-list (or an asterisk if there is none).

Thus all essential information about the subroutine is available, and we can, for example, print out the cue-list if we wish, and also determine just where each subroutine has been put. The parameter-list address is that of the actual list used in the processing (if several lists are supplied for a given subroutine only the last one read in is effective; an optional list is used only if no other is supplied). In this printing the routine number consists always of three digits (except that spaces are printed instead of left-hand zeros); if there is a programmer's subroutine then the first digit of its routine number is printed as + (representing 10), for example routine number 1009 is printed as +09.

Assembly ceases to read through the Library tape as soon as it is *satisfied*. Normally this is when all the specified subroutines have been found. Strictly speaking, Assembly is satisfied only when it has found each of the following:

(a) a cue-list (and its subroutine) corresponding to each call for a cue,
(b) a call for a parameter for every parameter-list supplied (other than optional lists),
(c) a parameter-list corresponding to each call for a parameter,
(d) a call for a cue for each programmer's subroutine supplied.

If only Library subroutines are used then the last is irrelevant. For example, Assembly will never be satisfied if a non-existent subroutine is called for, or if a parameter-list is not supplied for a subroutine needing one, or if a parameter-list is supplied for a routine which does not need one. In such circumstances Assembly will continue reading the Library tape seeking satisfaction in vain, and it will eventually reach the special *End of Library* routine which appears, after a Y-directive, at the end of the tape. This routine is entered as soon as it has been read in (it is a kind of interlude) and prints out information showing which kinds of tag are missing. If, for example, a non-existent R 199 has been called for in error it will show that a tag with function 08 is missing (this is equivalent to 28); it is at this stage possible to supply any missing information, if it is available, by putting it into the second tape-reader and operating the Run key.

The whole of the programme (except for the Library subroutines) should be punched before the A2, or Assembly may not be satisfied on reading the Library tape.

The Library tape has to be scanned as rapidly as possible and it is consequently punched in the special binary code used by Binary Input (see Sec.7.7). This is read at the rate of nearly 5 blocks per second.

The way in which a programme tape is punched is mostly described in Sec. 6.5, but there are a few points concerning Assembly which should be amplified. The A1, A2 and A3 directives are punched in a straightforward way, for example the A1 directive is normally punched as follows:

Blank tape CR LF

λ Aϕ1 CR LF

blank tape

As usual Er (erase) may appear anywhere except between CR and LF. In fact the single digit after the A is not an address and need not be followed by CR LF; either of the CR LF's indicated above may therefore be omitted but this is not recommended. It is important to note that, apart from Er, no character is allowed between the A and the 1, except for a single ϕ. For example, it would be wrong to put a Sp in here and this would lead to a loop stop on input. The L-directives which appear at the

end of every routine and cue-list are normally punched simply as

λL blank tape

without any CR LF.

A tag is simply a pseudo order-pair preceded by an R, and is consequently punched as λ Rϕ followed by the usual punching for pseudo order-pairs. It is customary, however, to make the *N*-addresses up to *five* characters (i.e. four plus a space) to cater for tags referring to programmer's subroutines (which have a four-digit routine number). For example, the tag

R 3 4 -0 2
456 - 01 -

would usually be punched as follows:

λ Rϕ Sp Sp 3 Sp 4 - 02 CR LF

Sp 456 Sp - 01 - CR LF

Note that the letter R counts as one character.

If we punch in this way a tag having a two-digit block-number (in the *N*-address of the *a*-order) we get a space between the R and the block-number, which improves the appearance of the print-out. Great care should be taken not to omit any of the minus signs in punching tags.

The name of a subroutine is punched after the cue-list according to the usual rules for N-directives but the N itself must *not* be preceded by CR LF. In fact only blank tape (ϕ) may appear between the L at the end of the cue-list and the λN introducing the name; no character other than ϕ is permissible (not even Er). If there is no name (this would be unusual) then the L should be followed by blank tape and the CR LF before the first word or tag of the subroutine. Generally speaking, the name of a subroutine should start with CR LF to ensure that it is printed on a new line; for example if a subroutine has the name ABCD it should be punched as follows (the L is that at the end of the cue-list):

L blank tape λ Nϕ CR LF

λ ABCD blank tape CR LF

First word or tag of subroutine

If the subroutine is accepted the following characters of the name are printed optionally:

ϕ CR LF λ ABCD ϕ

Should the N before the name be preceded by CR LF then the name will always be printed, even if the routine is rejected. Other unacceptable characters (such as Er) cause a loop stop.

▼ Lists of constants and other data may be dealt with in various ways. The simplest is usually to compute the total storage space occupied by the programme (this can be confirmed by the optional printing produced by Assembly or by a B-directive written after the A3). Various sections of the main store can then be allocated, each starting at a fixed address. This method is usually adequate but sometimes we may programme the computer to determine where data are to be put and which blocks of locations are to be used as working space. The final value of the T.A. in 5.7_M may be of use. Occasionally it is best to use Assembly in an unconventional way, for example by supplying dummy subroutines consisting of just a cue-list followed by two L-directives; the cue-list can be used to ▲ provide correct relative addresses for automatic insertion into the master-programme.

We conclude this section with a description of a complete programme for tabulating the function

$$z = \text{argsech } y = -\log_e \left\{ \frac{1 - \sqrt{(1 - y^2)}}{y} \right\},$$

for $y = 0.01\ (0.01)\ 0.99$. There will be a master-programme to organise the calculation, and a programmer's subroutine to evaluate the function. When writing a programme of this sort it is usually best to write the programmer's subroutines first and the master-programme last. The master-programme will use the Library subroutine R 5 to print the values of y and z. The programmer's subroutine may be numbered R 1000; it will call in two Library subroutines, R 200 for the square roots and R 224 for the natural logarithms. Abridged specifications of the Library subroutines R 5 and R 200 are given on pages 167 and 168.

▼ The following is an abridged specification of R 224:-

R 224 Logarithm, variable range. $p' = 2^{-n} \log_e (pq)$.

Puts into *X*6 the scaled down natural logarithm of the double-length fraction in *X*6 and 7. The positive integer n is fixed by a preset-parameter.

Name:	LOG MK. 2	Store:	3 blocks.
Uses:	*U*0, 1; *X*5, 6, 7.	Cue:	01
Time:	42 or 58 msec.	Error:	about 2ϵ.
Link:	In *X*1, obeyed in 1.7.		

Notes: Only the left half of the normalized (pq) is used; q must not be negative. If (pq) is negative or the result overflows there is a loop stop.

The value of n should be set by a parameter-list of the following form:

R	0	0	-0	1
224		-	04	-
	n	0	00	0.
	0			

If no parameter-list is supplied by the programmer, n will be set as 5 by an optional parameter-list.

The fraction y is always in range. As y increases from 0.01 to 0.99 it can easily be found that z decreases steadily from about 5.3 down to about 0.14. We therefore evaluate $z/8$ instead of z. In evaluating the formula we can adopt a quite straightforward approach since the values $y = 0$ and $y = 1$ are excluded, and overflow cannot occur. The logarithm subroutine R 224 evaluates

$$p' = 2^{-n} \log (pq).$$

where n is fixed by a preset-parameter. We put $n = 3$ and the result is then $-z/8$. The programmer's subroutine R 1000 will be called in from one point only in the master-programme and we shall therefore write it to obey a preset link when it has done its work. In R 1000 there will consequently be tags calling for cues to R 200 and R 224, a tag calling for a preset-parameter (the preset link) and a parameter-list (with its title) for R 224. The subroutine has only one cue in its cue-list; when this cue is obeyed in the master-programme the subroutine replaces the value of y in $X6$ by $z/8$.

The master-programme has to call in R 5 to print y, call in R 1000 to evaluate $z/8$, call in R 5 again to print z and repeat with the next value of y. It starts with $y = 0.01$; and this has to be increased 98 times by 0.01. This could be done by repeated addition, but this is inadvisable in a binary computer since the value of 0.01 will be held only approximately, and repeated addition would multiply the error. In this programme we store $x = 100\,y$ as an integer; x has to take all integer values from 1 to 99, this can be done by repeated addition of 1 without any error. The value of y can be obtained by dividing x by 100 (using a rounded division), and this is not subject to the accumulation of rounding errors.

The complete programme is as on pages 183, 184 and 185.

8.6 The Library

The Library includes, so far as possible, all subroutines and complete programmes of general use. The subroutines are included on the Library tape; separate tapes of the complete programmes are available. The Library is continually expanding as new routines are written and included in it; and from time to time a new Library tape becomes necessary as a result. Subroutines on the Library tape are democratically arranged, the most popular coming first; this minimizes the time spent in scanning the tape looking for wanted routines.

Some subroutines appear two or more times on the tape; this is because they are needed by other subroutines. For example, R 200 (square root) is a frequently needed subroutine and consequently appears first on the tape. The routine R 242 (arcsin and arccos) is less often wanted and therefore appears further down the tape; but it uses R 200 as a subroutine and must therefore be followed by a copy of R 200 in case the latter was not called for by the master-programme (in this event Assembly would reject the first copy of R 200 as not wanted, but would accept the second, if R 242 were called for, since there is a tag in R 242 calling for the cue to R 200). Assembly never accepts two copies of a routine.

The subroutines in the Library are each identified by their routine numbers, which are allocated according to the function or purpose of the routine. Each routine number is less than 1000; the allocation scheme is given in Appendix 7. Complete programmes are allocated routine numbers 7000† greater than those of corresponding subroutines; such routine numbers are used only for reference purposes, and cannot be used with Assembly.

Various documents are available concerning the Library. The *index* to the Library gives brief particulars of each routine, so as to enable a programmer to see which routines are available to meet any particular need. A copy of this index is given in Appendix 7. A printed *specification* of each routine is prepared for insertion into loose-leaf volumes. The specifications give full details of how the routines are used. Printed *programme sheets* for the more frequently used routines are also prepared, so as to provide full information in case a programmer wishes to adapt a routine to do a slightly different job, or if he wishes to use some of the techniques employed in it. A document is also prepared from time to time showing which routines are available on the current Library tape.

Certain *conventions* are adopted in writing and using Library subroutines and their specifications; these should also be used in programmes and programmer's subroutines, if at all possible. The following are the main conventions; any departure from them must be indicated explicitly in the specification.

1. The *link* for a subroutine is to be set in $X1$ before the subroutine is entered.
2. In a specification any registers or storage locations whose contents may be altered by a routine are stated to be *used*. Nothing should be assumed about their contents after the routine has finished.

† Four-digit routine numbers beginning with 7 are allocated to complete programmes to be run on the 7168-word store: complete programmes for the 4096-word store have corresponding numbers beginning with 2.

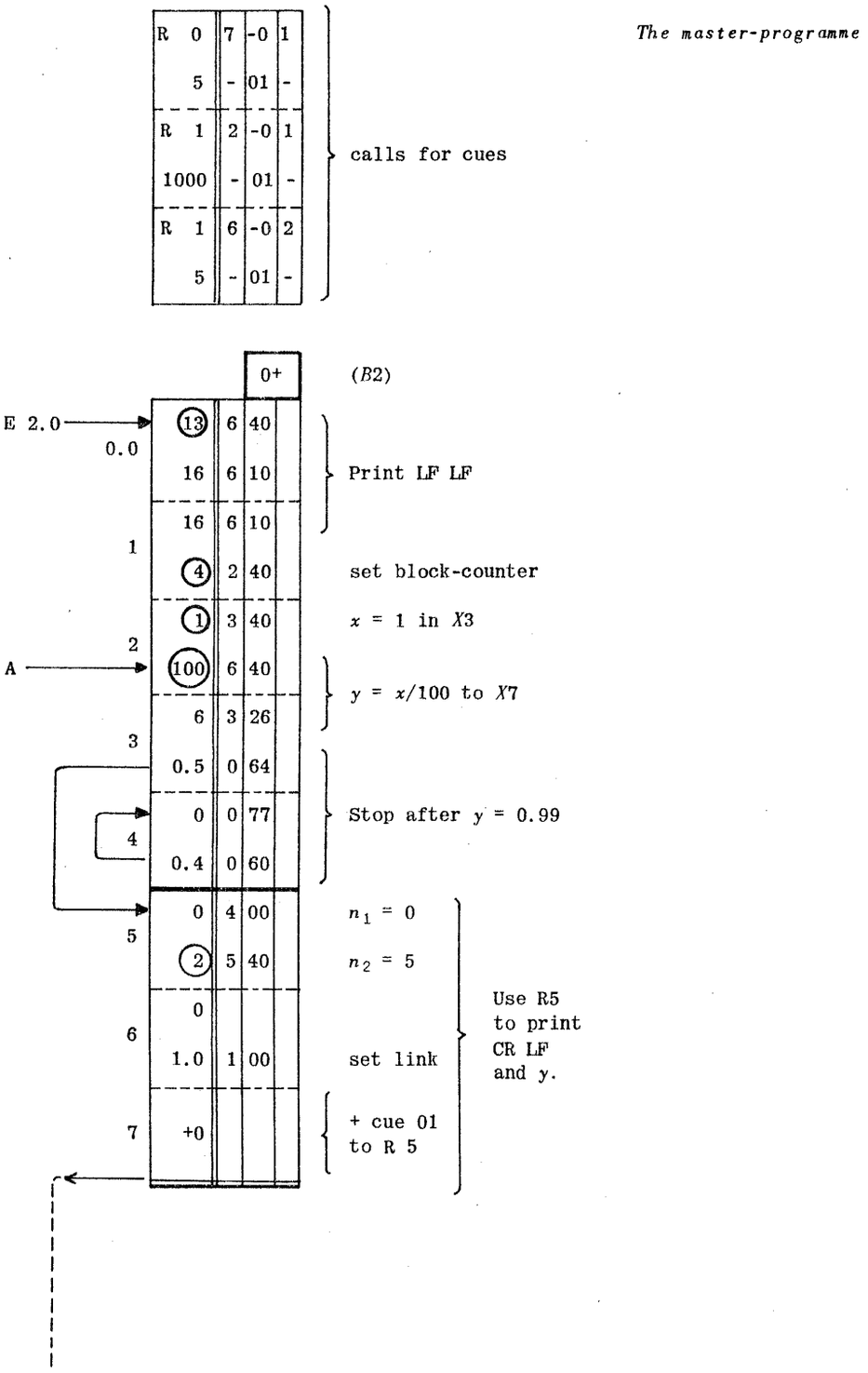
D

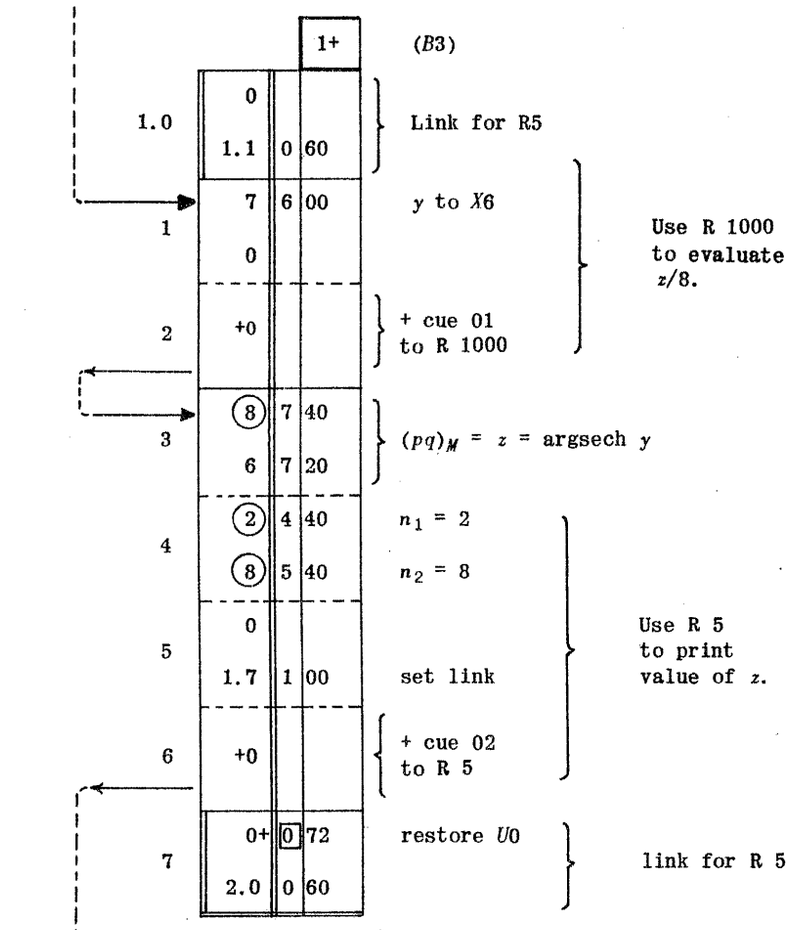
N

TABULATE ARGSECH

A1 (prepare Assembly)

The master-programme




1+
(B3)
1.0
0
1.1 0 60
Link for R5
1
7 6 00
0
y to X6
2
+0
+ cue 01
to R 1000
Use R 1000
to evaluate
z/8.
3
8 7 40
6 7 20
(pq)M = z = argsech y
4
2 4 40
8 5 40
n1 = 2
n2 = 8
5
0
1.7 1 00
set link
6
+0
+ cue 02
to R 5
Use R 5
to print
value of z.
7
0+ 0 72
2.0 0 60
restore U0
link for R 5
2+
(B4)
2.0
2.2 2 67
13 2 40
count lines in block
1
16 2 10
5 2 40
print LF
reset block-counter
2
1 3 41
0.2+ 0 60
increase x by 1
repeat loop
A
R 0 0 -0 1
1000 - 04 -
title
1+ 1 72
1.3 0 60
Preset main
link
Parameter-
list
for R 1000
L
End of master-programme
The programmer's subroutine
R 0 0 -0 1
1000 - 08 -
Title
01
0+ 0 72
0.1 0 60
Cue 01
Cue-list
L N
End of cue-list
ARGSECH

```
R  0   3  -0  1      } call for cue 01
200    -  01  -      } to R 200 (in 0+.3)
---------------------
R  0   7  -0  1      } call for cue 01
224    -  01  -      } to R 224 (in 0+.7)
---------------------
R  1   2  -0  1      } call for parameter 01
1000   -  02  -      } to R 1000 (in 1+.2)
---------------------
R  1   3  -0  1      } Title of parameter-list for
224    -  04  -      } R 224, starting at 1+.3
                     } and containing one parameter
```

```
                    0+
          0+   0  72      } link for R 200
0.0       0.4  0  60      }
01 ------>6    4  02      -y to X4
1         4    6  20      -y² to X6 and 7
---------------------
          32   6  03      1-y² to X6 and 7
2         0.0  1  00      set link
          +0             } + cue 01 to R 200
3 <-------                } √(1-y²) to X6
          1+   0  70      set link for R 224
4 ------->32   6  01      √(1-y²) - 1
---------------------
          4    6  26      divide by -y
5         7    6  00      } (pq) = (1-√(1-y²))/y
---------------------
          0    7  00      }
6         0               }
---------------------
7         +0             } + cue 01 to R 224
                         } - z/8 to X6.
<-------
```

```
                    1+
          1+   0  72      } link for R 224
0.0       0.1  0  60      }
          6    6  02      z/8 to X6
1 ------->0
---------------------
2         +0             } + preset-parameter 01
                         } = preset main link.
Exit <----
---------------------
          3    0  00  0.  }
3         0               } Parameter for R 224.
```

```
L       End of subroutine
A2      Read Library tape
A3      Process the programme
E 2.0   Enter the programme
```

3. A subroutine must be entered in one of the specified ways (normally by a cue) every time it is used. Only self-preserving subroutines (see below) are left intact in the computing store ready to be obeyed again. If *X*1 is stated to be used the link will have to be set again, even if the same one is required.
4. The *overflow-indicator* (OVR) is to be clear on entry to a Library subroutine and will be left clear on exit. It is recommended that OVR should be clear at any well-marked division or break between one part of a programme and another.
5. A subroutine should use as few blocks of ordinary registers as possible. Generally a Library subroutine will use *U*0 and *U*1 only; some will use only *U*0.
6. As few accumulators as possible are to be used. Clearly *X*6 and *X*7 will be needed by most subroutines and *X*1 holds the link. If other accumulators are needed they are used in the order 5, 4, 3, 2.
7. Output routines for the normal 60 char./sec. punch are usually written to use only *U*0 and to preserve the accumulators in *B*0.
8. Any print routine will leave the teleprinter in figure shift, and will assume that it is in figure shift on entry.
9. Any subroutine requiring a single *operand* is normally written to take it from *X*6; if there are two operands they may be in *X*6 and *X*7. If the result is a single number it will normally be left in *X*6.
10. *Preset-parameters* required for addition to order-pairs should normally be positive (i.e. they will be written as stop order-pairs). This allows the addition of several parameters to a single order-pair.
11. Programmes and subroutines should be made as simple as possible to use; in particular any awkward punching of tape or any special and unusual ways of entry should be avoided.
12. All Library subroutines are written on the assumption that they will be stored in the *first 128 blocks of the main store* (*B*0 to *B*127). The master-programme and programmer's subroutines should also be stored here if possible.

These conventions are not arranged in order of importance. A few notes on them may be desirable. Concerning Nos. 1 and 9, it is sometimes advantageous to interchange the roles of *X*1 and *X*6, i.e. to put the link in *X*6 and a single operand in *X*1. The operand can then be the subject of single-word transfers. Sometimes it may be very difficult to write a subroutine in such a way that it uses *U*0 and *U*1 only (see No. 5 above); it may then be desirable for the subroutine to preserve *U*2, for example, by obeying some such order as

4+[2]73

which copies *U*2 into *B*4+. At the end of the routine the corresponding 72-order can be used to restore *U*2. If this technique is adopted *U*2 is not included in the blocks of ordinary registers stated to be used by the routine (nor is *B*4+ included in the locations used; though of course storage space must be allowed for it).

A *self-preserving* subroutine is one which may be entered and re-entered by simple jumps when once it has been transferred into the computing store. Such a subroutine may either be held entirely in the computing store and be so arranged that it does not write over any of its own orders, or else it may arrange to read in its own first few orders again just before obeying the link. Self-preserving subroutines are often used with computing store links (see Sec. 5.8) and *partial cues*, which we shall now describe.

Consider a subroutine which may be transferred into *U*3, 4 and 5 and is then self-preserving. Having once transferred it, the master-programme can repeatedly enter it, for example, by orders like the following.

(1.3+) 1 40	set computing store link
3.0 0 60	

When the subroutine has finished it will obey a pair of orders like these.

(25) 1 52	shift address to 1_M
0 0 60 1	jump to specified address

In this case the subroutine will return to the *b*-order in 1.3 of the master-programme. A subroutine like this is easy to use repeatedly when once it has been transferred into the computing store. To facilitate the transfers the subroutine is provided with two partial cues (sometimes called 0+ cues), for example its cues 01 and 02 may be as follows.

01	0+ 0 00 0. 0	*a*-order partial cue
02	0 0+ 0 00 0.	*b*-order partial cue

The first of these cues contains zeros everywhere except in the *N*-address of the *a*-order, where there is the block-number of the first block (*B*0+) of the subroutine. The other cue is similar but has the address

of the subroutine in the *N*-address of its *b*-order. These cues can be added to 72-orders in the master-programme so as to make them transfer the subroutine.

Suppose, for example, that the following orders appear in block 4+ of the master-programme.

```
4+.5 |  0 [3] 72  |
     |-------------|
     |  1 [4] 72  |
4+.6 |             |
     |  2 [5] 72  |
     |-------------|
```

By calling for the appropriate cues we can get the address of the subroutine *added* to the *N*-addresses in these orders. This can be done by writing the following tags calling for cues at the head of the master-programme (we suppose that the routine number is 1005).

```
| R  4  5 -0 2 |
| 1005  - 01 - |   call for cue 02 to R 1005, to be added into B4+.5

| R  4  6 -0 1 |
| 1005  - 01 - |   call for cue 01 to R 1005, to be added into B4+.6.

| R  4  6 -0 2 |
| 1005  - 01 - |   call for cue 02 to R 1005, to be added into B4+.6.
```

We simply write one tag for each of the orders requiring alteration, a call for cue 01 if it is an *a*-order, or for cue 02 if it is a *b*-order. Note that the word in *B*4+.6 is tagged twice; both partial cues will be added to it. This technique finds other applications; this is the reason why Assembly *adds* in the cues and parameters, instead of simply placing them where required.

A few Library subroutines include interludes which do a certain amount of organisational work when they are read in; these interludes occasionally refer to such things as parameter lists which are assumed to have been read in previously. It is therefore essential that these lists should appear before the A2 on the programme tape.

When Assembly has finished the processing of a programme the parameter-lists are normally no longer needed and may be overwritten. One or two Library subroutines do actually refer to their parameter-lists (by a table look-up) during their operation. Such lists are properly not preset-parameters, and they may not of course be overwritten; the specifications of these routines indicate this fact.

8.7 The magnetic tape Library†

On an installation with magnetic tape equipment is is possible to read Library routines either from paper tape, or, more rapidly, from 16-word magnetic tape.

In order to call for subroutines from the magnetic tape Library, the warning characters A5 and A6 are used. A loop steering tape containing this sequence is placed in the second tape-reader in place of the paper tape Library, and is read as soon as the A2 on the programme tape is encountered. There need be no difference in the master programme tape whether the paper tape or magnetic tape Library is used. Alternatively the A5 may be punched anywhere after the A1 on the programme tape, but before the A6, and the A6 may be punched in place of the A2. The method has the disadvantage that the programme cannot be so easily used if the magnetic tape Library is not immediately available.

The magnetic tape Library is normally mounted on the highest numbered magnetic tape mechanism available, as this mechanism is least often required for other purposes. The A1 directive places in B895.1 a tape order to search for the first section of the Library on the highest numbered tape mechanism. The A5 directive obeys this tape order, and the A6 enters the part of the Initial Orders which reads the magnetic tape Library. When the required subroutines have been read, control is returned to the main tape-reader and Assembly initiates a search for the beginning of the magnetic tape Library.

If it is required to mount the Library on another mechanism the following sequence must be read after A1 and before A5:

```
X 895.1+
64m      (where m is the mechanism number)
```

The A5 and A6 are normally punched as follows:

```
Blank tape CR LF
λ A φ 5    CR LF
λ A φ 6    CR LF
Blank tape
```

The punching conventions are the same as for A1, A2 and A3. The single digit after the A is not an address and need not be followed by CR LF although it is recommended that CR LF should be punched. Apart from Er, no character other than a single φ is allowed between the A and the 5 or the A and the 6. Er may appear anywhere except between CR and LF.

† On the 4096-word store there is no facility in the Initial Orders for reading the Library from magnetic tape.

The warning character Q is used to call for complete Library programmes, such as the Autocode, which are stored on the same 16-word magnetic tape as the subroutine Library. The Library of complete programmes, known as the Q-Library, starts one section before the subroutine Library and extends backwards down the magnetic tape.

The warning character Q followed by the number of the routine required should be punched at the head of the appropriate tape: this may be a programme tape for an interpretive routine such as the Autocode or a data tape for a complete programme such as "Simultaneous Equations". When the required routine has been transferred from magnetic tape to its normal position in the main store the computer will return to the Initial Orders to read paper tape, except when there is an E or J at the end of the routine.

Q may be obeyed from the handswitches in the following manner:

(a) Set up Q(10001) on the last five handswitches in the same way as other warning characters. The address should normally be zero.
(b) START and RUN.
(c) The computer should reach an optional stop in 4.2. If optional stops are inhibited there will be a loop in 4.2 and 4.3 until the handswitches are cleared.
(d) Clear the handswitches and operate the Run key to pass the optional stop (if any).
(e) Tap out the routine-number digit by digit, using handswitches 0 to 9 for the digits 0 to 9. As each digit is tapped it will be punched on the output tape.
(f) If an error is made in tapping the number handswitch 11 should be tapped: this will print * CR LF λ Q ϕ, cancel the number, and return to stage (c) above.
(g) Terminate the routine-number by tapping handswitch 10.
(h) When the required routine has been transferred from magnetic tape to its normal position in the main store there will be a 77 stop in 0.5 before returning to Initial Orders to consult the handswitches, except when there is an E or J at the end of the routine.

Most routines are transferred from magnetic tape to a standard position in the main store in the same way as a binary punched programme tape. When the whole of the selected routine has been transferred to the main store, the Initial Orders initiates a search for the beginning of the subroutine Library.

If the selected routine includes E- or J- directives these are obeyed in the normal way.

When Q is being read from paper tape or set up on the handswitches, handswitch 14 should normally be clear. If the Library is mounted on some mechanism other than the highest numbered, handswitch 14 should be down and the mechanism number set up as an octal digit on handswitches 11 to 13. These handswitches should be set before operating the Start and Run keys: they may be left set if Q is on paper tape, but must be cleared before the tapping routine if Q is on the handswitches.

The magnetic tape containing the Library should always be isolated so that no part of the Library may be accidently overwritten. Every time an A6 or Q is obeyed, a check is made to ensure that the tape is isolated.

8.8 The C-directive

It is sometimes difficult to make corrections or alterations to subroutines, particularly if the programme includes many of them. One method is to find out from the Assembly printing where the particular subroutine has been stored and then to insert corrections by using X- or T-directives. The short tape effecting the corrections can be inserted near the end of the programme tape, after the A3 but before the E for entering the programme. In this way corrections and alterations can be made to Library subroutines as well as to those written by the programmer. This method suffers from the disadvantage of using absolute addresses, so that the corrections may have to be changed if the subroutines are later stored in different places in the main store. This disadvantage is most serious when there are many programmer's subroutines, since some of these may well be re-written during the development of the programme (this may involve moving many of the others) and it may, in any case, be desirable to test some of them by themselves before the whole programme is put together.

To help overcome these difficulties, the C-directive (C for *correction*) has been introduced. This is punched as a letter C followed by a routine-number; it may be used at any time after the subroutine referred to has been read in, provided that the information stored by Assembly in its working space has not been spoiled. The effect is simply to set the relativizer to the value it had when the specified routine was being read, i.e. to the address of the first block of the subroutine; the Transfer Address is not altered. When the relativizer has been set in this way, corrections to the specified subroutine can be made by using directives (such as T, X or S) with relative addresses.

For example, to change the α-order in 2+.3 of the programmer's subroutine R1002 we could punch a sequence such as the following:

```
C 1002          Set relativizer to start of R1002
X 2+.3
0.7  2 61       new order
```

This sequence could appear anywhere on the tape after R1002.

This directive can sometimes be used to facilitate cross-references from one routine to another, for example we can insert a cue to R1002 in 1+.3 of R1005 by some sequence such as the following.

```
C 1005          set relativizer to start of R 1005
T 1+.3          set T.A. to 1+.3 of R 1005
C 1002          set relativizer to start of R 1002
0+[0]72  }
0.1  0 60 }     cue to R1002, containing a relative address.
```

This sequence would have to be read in after both of the subroutines referred to.

The optional printing associated with this directive consists of a letter C followed by the routine number.

If the required subroutine has not been read in when a C-directive is encountered there is an optional stop in 3.7; if the Run key is then operated the relativizer is set to the beginning of the isolated store so that most relative corrections will have no effect (the second kind shown above would then, however, cause a writing-with-overflow stop).

8.9 Detailed description of Assembly

Assembly is divided into two main parts. The first is concerned with everything on the input tapes up to the A3-directive, it is mainly concerned with the selection of Library subroutines and the preparation of information to make the processing possible later. The second part is called in by the A3-directive and inserts all the cues and parameters, and prints (optionally) information concerning the subroutines accepted. These two parts of Assembly are quite distinct except for an *exit sequence*, which is obeyed whenever Assembly returns to read tape. This exit sequence will reset the T.A. and R.A. after a rejected subroutine; it always does a B-operation and calls in *Input*.

The first part of Assembly is called in by the A1-directive; it then calls in the rest of the Initial Orders as a subroutine to read in nearly all the programme and Library tapes. These tapes will be read by *Input* or *Binary Input*. Assembly is re-entered whenever an L-directive or its binary equivalent† is read. As far as Assembly is concerned the tape is therefore regarded as divided into sections by L-directives. These sections are of three sorts:

(a) *Subroutine cue-lists.* These are always headed by a title tag with the second function digit 8, and contain no other tags.

By inspecting the title Assembly decides whether to accept or to reject the subroutine. It will reject a subroutine only if it has previously accepted the same subroutine (judging by the number) or if it is a Library subroutine and no other tag has been read with the same routine number. If it is accepted then the whole cue-list is copied away.

(b) *Subroutine programmes.* These are always immediately preceded by subroutine cue-lists, and may contain any tags other than cue-list titles. If the cue-list was accepted then the subroutine programme is read into the next available locations, except that a new block is always started; these locations will be the same as those occupied temporarily by the cue-list. If the cue-list was rejected the subroutine proper is read in as before, but the T.A. and R.A. are then put back to their original values so that the subroutine will be over-written by the next section of tape. The N-directive which may appear before the subroutine is treated specially by Assembly.

(c) *Other programme.* A section of tape which is neither a cue-list nor immediately follows one is treated as a part of the master-programme and read into the next available locations, starting at a fresh block.

Assembly uses extensive working space at the upper end of the main store (and *B*0.1, where the link is kept), which may be divided into three parts:††

(i) Five words in *B*895 to hold a tape word, modifiers and counters.
(ii) The index, which occupies space from *B*883.0 onwards.
(iii) The reference list, made up mostly of tags, which occupies locations working backwards from *B*882.7.

The five words in *B*895 record the following information:-

895.1 The tape control word planted by A1 to search for the first section on the magnetic tape Library.
895.2 Assembly modifier and index counter:
895.2_M is always 882.0,
895.2_C is one more than the number of words in the index.
895.3 Reference Address } values after the last accepted subroutine or section of
895.4 Transfer Address } master programme, and the B-operation following it.
895.5 After an accepted cue-list and during the following subroutine, 895.5_M is 1.0 less than the address of the index-word relating to the subroutine.

The index is compiled by Assembly with one index-word for each subroutine accepted or to be accepted. Each tag is examined when the L is read at the end of its section of tape; it is then compared with each index-word. If no index-word has the same routine number a new index-word is formed, unless the tag is the title of the cue-list of a Library subroutine (its digit 34 is then 1), when the subroutine is rejected. If an index-word is found with the same routine number as the tag or if a new index-word has to be formed, then the address of the index-word is written into the tag in place of the routine number, and the function digits (35 to 38) of the tag are *mixed* (by an *or* operation) with those already in the index-word from previous tags (if any). In this way the last 4 bits in the index-word contain 0's in those positions where the corresponding kinds of tag have not yet been encountered.

If the tag is a call for a cue or parameter nothing further is done to it by the first part of Assembly. If it is the title of a parameter-list, the address of the parameter-list is written into the index-word, unless it is an optional parameter-list and another list has been found previously. Thus a non-optional parameter-list (title tag with function 04) always supplants earlier lists, whereas an optional parameter-list (title function 06) is only used if it arrives first. The sign of a

† The binary equivalent of an L-directive always has the effect of an Assembly L; it does not in fact use the word in *B*0.1.

†† The 4096-word Assembly uses the following working space:
(i) Four words in *B*511, 511.2 - 511.5 (which record information corresponding to that held in 895.2 - 895.5 on the 7168-word store).
(ii) The index, which occupies space from *B*507.0 onwards.
(iii) The reference list, made up mostly of tags, which occupies locations working backwards from *B*506.7.

parameter-list title is also changed at this stage, so that it becomes a stop tag and is ignored in future by Assembly (tags whose stop/go digit is 0 are entirely ignored by Assembly).

If the tag is the title of a cue-list, then the cue-list (and the following subroutine) is rejected if either

(a) the title is that of a Library subroutine and there is no corresponding index-word, or

(b) the corresponding index-word shows that a cue-list has already been found for this subroutine.

If Assembly requires to reject the subroutine programme which follows the cue-list it makes the R.A. negative (actually -2^{-10} is put in *U*5.5). This has the effect that when Assembly is next re-entered the R.A. and T.A. are set back to what they were before the cue-list was read (these values are stored in *B*895, see above). If a rejected subroutine is punched in the ordinary Initial Orders notation, it is placed in the store as usual but later overwritten, but its tags (e.g. calls for parameters) are put harmlessly into locations starting at 1023.7 and working backwards; but if it is punched in binary no storing of programme or tags takes place since the word in *U*5.5 is negative (Binary Input ignores everything except for a binary L if *C*(5.5) is negative; it continues to check the checksums however).

If the cue-list is accepted, it is copied into the reference list with the title preceding it, so that the last cue occupies the location formerly occupied by the title; and the new address of the title is entered into the index-word. The number of words copied is determined by the change in T.A. since Assembly was last left. In addition a word is placed in *B*895.5 which contains, in the modifier position, the address (minus one block) of the index-word of the accepted subroutine; this is for convenience in writing interludes (see below).

If a cue-list is rejected or if optional printing is suppressed (*H*0 = 1), Assembly reads the N-directive and name (if any) at the head of the subroutine, so that it shall not be printed by the Initial Orders. To do this Assembly reads and ignores blank tape until it finds λN or CR LF; λN indicates the start of the name and CR LF that there is none, and most other combinations cause a loop stop.

Whenever the section last read is not a cue-list, the index is examined before returning to read more tape to see whether every subroutine called for has been found; if so the next tape will be read from the main tape-reader. If the Library tape was being read this stops, and the main tape next read in normally starts immediately after the A2.

It will be seen that when the first part of Assembly stops reading the Library tape it has accumulated enough information in the reference list and the index for the second section to process the programme as required. This is done, on reading A3, in a more or less straightforward manner by working forwards† through the reference list up to *B*882.7. Each call tag (with function 01 or 02) contains the address of the word tagged and the address of the index-word, and the latter contains the addresses of the cue-list and the parameter-list (if any). On adding parameters and cues to the tagged words overflow is ignored. At this stage parameter-list titles are ignored (being non-negative) and cue-list titles cause the following cues to be avoided (the number of them before the next tag is in the title). When the processing is complete informative printing occurs (if *H*0 = 0). This is done by working through the index, the subroutine address being obtained from the cue-list title. The T.A. and R.A. are unaffected by the second part of Assembly, except for a B-operation.

Interludes may be needed for some subroutines. Such an interlude should not be entered by a simple J-directive such as J *a*(+) but by a J-directive with *two* addresses, in fact

J 928.0+ - *a*(+) ††

This is because the subroutine may be rejected, in which case an ordinary J-directive would still cause the interlude to be entered. Other directives are unlikely to have undesirable effects, apart from N, which is specially treated. The above J-directive causes a sequence starting at *B*928.0+ to be entered; if the subroutine is being accepted the second address *a*(+) is treated as if it were the first, otherwise Input is immediately re-entered to read the rest of the rejected subroutine. This J-directive must appear on the tape before the L terminating the subroutine programme. An interlude of this kind is usually called an optional interlude. Certain information used by Assembly may be useful to such an interlude, in particular the word in *B*895.5 (see above) which enables the index-word of the subroutine to be found. The structure of this index-word and of tags may also be useful; this is shown in the diagram below. Only 11 bits in the index-word are allotted to the cue-list address, but as this is in Assembly working space the two most-significant bits are assumed to be ones. All the tags read in will have been processed by Assembly (when the L was read at the end of their section of tape), except for those tags, such as calls for parameters, appearing in the subroutine containing the interlude (since the interlude is entered before the L is read which terminates it).

Unprocessed Assembly Tag:

Digits	0	1 to 13	14 to 19	20 to 22	23 to 32	33	34 to 38
Contents	1	Tagged address	*uv*	0 0 0	Routine number	0	Function

† It should be noted that it is only when tags are read by Input or Binary Input that the reference list is worked through backwards; Assembly always works forwards.

†† For the 4096-word store the J-directive is J 560.0+ - *a*(+).

Processed Assembly Tag: †

Digits	0	1 to 13	14 to 19	20, 21	22 to 34	35 to 38
Contents	1	Tagged address	*uv*	0 0	Index word address -1.0	Function

This digit is zero in a parameter-list title.

Index word: ††

Digits	0	1 to 10	11 to 21	22 to 34	35 to 38
Contents	0	Routine number	Cue-list address 768.0	Parameter-list address	Mix of functions

When an optional interlude has finished its work it should obey the link which was in *X*1 on entry. There are two alternative links which may be set, depending on whether the subroutine was punched in the ordinary Initial Orders notation or in binary; care should be taken to obey the actual link set. Either of these links is self-modified (see Sec. 5.10) and must be in *X*1 at the time it is obeyed. The ▲ interlude should also not disturb the Assembly link stored in *B*0.1.

On the 4096-word store, the processed tag is packed in the same way as on the 7168-word store, except that the index word address occupies only 12 bits (23-34) and bit 22 is spare.

† On the 4096-word store, bits 11 to 22 in the index word hold the exact cue-list address and the parameter-list address is held in the 12 bits 23 to 34. The rest of the index word is the same as on the 7168-word store.

Chapter 9

Some Programming Techniques

In this chapter are described some advanced methods of programming, including the so called automatic programming methods.

9.1 Floating-point operations

In most programmes the problem of scaling arises. The numbers entering into a calculation usually have to be scaled to bring them in range or to prevent loss of accuracy. In some problems there may be a physical quantity, for example a pressure denoted by p. Suppose that this pressure p can range over values from 10 up to 50 lbs./sq. inch, and we wish to measure it to within 0.1 lbs./sq. inch. Inside the computer we can store $10p$ as an integer and arrange our programme accordingly; alternatively we can store $p/64$ or $p/100$ as a fraction, which is probably to be preferred. Either scheme really amounts to measuring the pressure in different units (such as lbs. per 100 sq. inches, if we store p as $p/100$). The various quantities entering into a calculation must all be scaled in some way unless they can be conveniently represented directly as integers or fractions. This scaling involves some rearrangement of the formulae to be programmed. As a rule quantities are scaled by powers of 2 rather than powers of 10 because we can multiply by a power of 2 (i.e. shift) in any accumulator, whereas we can multiply directly by a power of 10 only in X6 and 7. Usually we write a programme in such a way that all the numerical data for it are punched in their natural, unscaled form, and we use a special input subroutine to read them in, scale them appropriately and store them. In a similar way the scaled results of the calculation are 'unscaled' before being printed by a suitable output subroutine. It should not be forgotten, however, that all the intermediate quantities arising in a calculation, even if they exist only momentarily, must also be scaled so that they are in range.

In certain problems scaling may be difficult. This may be because the calculations to be done are very complicated and we may not be able to determine readily the size of all the intermediate quantities. Or it may happen that certain numbers may vary over a very wide range. In either kind of difficulty we may be able to assign an upper bound for the numbers concerned, but if we scale them so that the upper bound is brought within range we may lose a great deal of accuracy, especially if an upper bound can be assigned only roughly. It is worth pointing out that we can often choose what seems to be a reasonable upper bound for each quantity and scale it accordingly. If the bound is ever exceeded overflow will occur, and this can be used to prevent wrong results passing undetected. The overflow-indicator is a powerful aid to correct scaling. The scaling difficulty is reduced in computers having a long word; in Pegasus we have 38 binary digits after the point (corresponding to rather more than 11 decimal digits) and we can often afford to scale down fairly generously without losing too much accuracy. If necessary we can work double-length and represent all our numbers by two words, giving the equivalent of nearly 23 decimal digits. The alternative approach of *floating-point*† working is usually to be preferred for general use, since it usually solves all scaling problems; in fact we need not scale the numbers at all. In contrast, the ordinary mode of operation may be called *fixed-point*.

It is common in scientific work, and in engineering, to represent very large or very small numbers by introducing a power of 10 as a scaling factor. For example, we may write numbers in the form

$$-3.457 \times 10^{12} \quad \text{or} \quad 8.664 \times 10^{-6}.$$

Here we arrange to write the decimal point immediately after the first significant digit of the number and we introduce a suitable power of 10 to correct the value. We can use a similar technique inside the computer; but since Pegasus is a binary machine it is usually easier to associate a power of 2 with the number, instead of a power of 10. A quantity x is consequently represented in the form

$$x = A.2^a$$

where a is an integer chosen so as to bring A within some convenient range. We shall call A the *argument* of the number x, and a the *exponent*††. Each quantity x is represented inside the computer by its argument and its exponent, i.e. we store two numbers to represent one quantity; there is no need to store the *base* 2 of the quantity because this will be the same for all. We shall restrict the argument A to one of the following ranges:

(a) $\frac{1}{4} \leqslant A < \frac{1}{2}$ if $A > 0$,

or (b) $-\frac{1}{2} \leqslant A < -\frac{1}{4}$ if $A < 0$.

† The binary point is allowed to *float*, its position being automatically adjusted to suit the size of the number. In French this is *virgule flottante*, in German *gleitendes Komma*.

†† There is unfortunately no generally accepted terminology for these two parts of a floating-point number. The quantity A is sometimes called the *mantissa* or the *numerical part*; and a is sometimes called the *index*.

The number zero requires special treatment in all floating-point work. The advantage of choosing these ranges is that two such arguments can be added or subtracted without overflow. The exponent a is a suitably chosen signed integer.

A sequence of orders will be needed to do an ordinary arithmetical operation on two floating-point numbers. Suppose, for example, we are given two such numbers

$$x = A.2^a \quad \text{and} \quad y = B.2^b,$$

and we wish to form their product $z = C.2^c$. We have

$$z = x.y = (A.2^a)\ (B.2^b) = AB.2^{a+b},$$

so that we can put $C = AB$ and $c = a+b$. This is quite simple to work out, but the operation is not yet complete. The quantities C and c must now be adjusted so that, provided C is non-zero, either

(a) $\frac{1}{4} \leqslant C < \frac{1}{2}$ if $C > 0$,

or (b) $-\frac{1}{2} \leqslant C < -\frac{1}{4}$ if $C < 0$.

If this is not done the argument of the result will be less than the argument of either of the original operands; and after a sequence of operations severe scaling down may occur. In other operations the argument of the result may be larger than that of either of the operands. The argument of the result must, in general, be shifted up or down to bring it into one of the above standard ranges, and the exponent must be simultaneously adjusted by subtracting from it the number of upward shifts. This operation is called *normalizing* the floating-point number and, after it has been carried out, the number is then said to be *in normal form.*† The following are two examples of floating-point numbers which are not in standard form:

$$\tfrac{3}{4} \times 2^8 \quad \text{and} \quad -\tfrac{1}{16} \times 2^{10},$$

The result of normalizing these numbers is, respectively,

$$\tfrac{3}{8} \times 2^9 \quad \text{and} \quad -\tfrac{1}{2} \times 2^7.$$

The normalizing process can, of course, be programmed by using only the orders so far described. This is, however, very inconvenient and, what is much worse, relatively slow; a special order, with function 56, has therefore been provided. This order does the whole process automatically; it deals with a double-length argument. We put the argument of the floating-point number into the double-length accumulator, $X6$ and 7, and the exponent into one of the other accumulators, say $X4$. When the order

(N) 4 56

is obeyed the floating-point number is automatically normalized. The N-address in the order shows the maximum number of places through which the argument is to be shifted; it is in fact shifted up at most N-1 places. This limit has been introduced primarily to allow for the argument being zero. The 56-order may be described verbally as follows.

56 (normalize) *Normalize the floating-point number whose argument is the fraction (pq) in X6 and 7, and whose exponent is the integer in the specified accumulator. But do not shift the argument more than N-1 places up (N being the number written first in the order).*

The 56-order may shift the argument down one place (and add one to the exponent), or it may leave the argument and exponent unaltered, or it may shift the argument up a variable number of places and subtract this number from the exponent. Usually the possibility of a zero argument must be specially allowed for (by including a 60-order to a special sequence of orders, for example), and it is usually ruled out by the time the 56-order is encountered. Consequently it is normally enough to write a large number (such as 80) in the N-address of the order.

The effects of the order may be defined algebraically as follows (here X is not 6 or 7):

$$(pq)' = 2^{\mu}\ (pq). \qquad (q \geqslant 0),$$

$$x' = x - \mu \qquad \text{(integer convention)},$$

where μ (Greek letter *mu*) is an integer chosen so that

(a) $\frac{1}{4} \leqslant (pq)' < \frac{1}{2}$ and $-1 \leqslant \mu \leqslant N-1$,

or (b) $-\frac{1}{2} \leqslant (pq)' < -\frac{1}{4}$ and $-1 \leqslant \mu \leqslant N-1$,

or (c) $-\frac{1}{4} \leqslant (pq)' < \frac{1}{4}$ and $\mu = N-1$.

Only one of these three statements can hold; the integer μ is uniquely determined. In these inequalities (pq) is, of course, a double-length fraction. The shifting which occurs in the double-length accumulator is similar in its effects to that produced by a suitable 54- or 55-order (the

† The terms *standardizing* and *in standard form* are also used.

sign-bit of 7 is always cleared by a 56-order, however)†. A 56-order can set the overflow-indicator, but only if the exponent overflows (this is very rare).

As an illustration of the use of a 56-order, we give a sequence of orders for multiplying two floating-point numbers whose arguments are in *X*6 and *X*7, respectively, and whose exponents are in *X*4 and *X*5.

```
    6  7 20      multiply arguments
    5  4 01      add exponents
(3) 4 56        normalize
```

We assume here that the two operands are in normal form and are not zero. There is then no need to put *N* greater than 3 in the 56-order. The above sequence leaves the exponent in *X*4 and the argument double-length in *X*6 and 7; to put into *X*6 the best single-length value we must round it, which can be done by adding the following two extra orders (see Sec.3.4):

```
   33  7 01  }
                  add ½ϵ
    6  0 23  }
```

But if we do this we should normalize the result again.

The division of two floating-point numbers is done similarly, we have now to divide the arguments (first doubling that of the divisor to prevent overflow) and to subtract the exponents. In fact

$$(A.2^a)/(B.2^b) \;=\; (A/B).2^{a-b} \;=\; (A/2B).2^{a-b+1}.$$

The following orders can be used, given the two operands as above:

```
↷   7 60        loop stop if divisor is zero
    7  7 01     double argument of divisor
    7  6 26     divide arguments
    7  6 00  }
                  (pq) is rounded quotient A/2B
    0  7 00  }
    5  4 03  }
                  a - b + 1
(1) 4 41     }
(2) 4 56        normalize quotient
```

We assume that the operands are in normal form and that the dividend is not zero. There is no need to put *N* greater than 2 in the normalize order. This sequence leaves the argument (single-length)†† in *X*6 and the exponent in *X*4 in normal form.

The addition or subtraction of two floating-point numbers is a little more complicated than multiplication or division. Suppose we have to add

$$x \;=\; A.2^a \quad \text{and} \quad y \;=\; B.2^b.$$

We first of all compare a and b, the two exponents. If $a \geqslant b$ we proceed, but if $a < b$ we interchange the arguments and exponents first. Assuming therefore that $a \geqslant b$ we must shift B down through $a-b$ places (thus effectively aligning the binary points) and then add it to A. We can write

$$C.2^c \;=\; A.2^a + B.2^b,$$

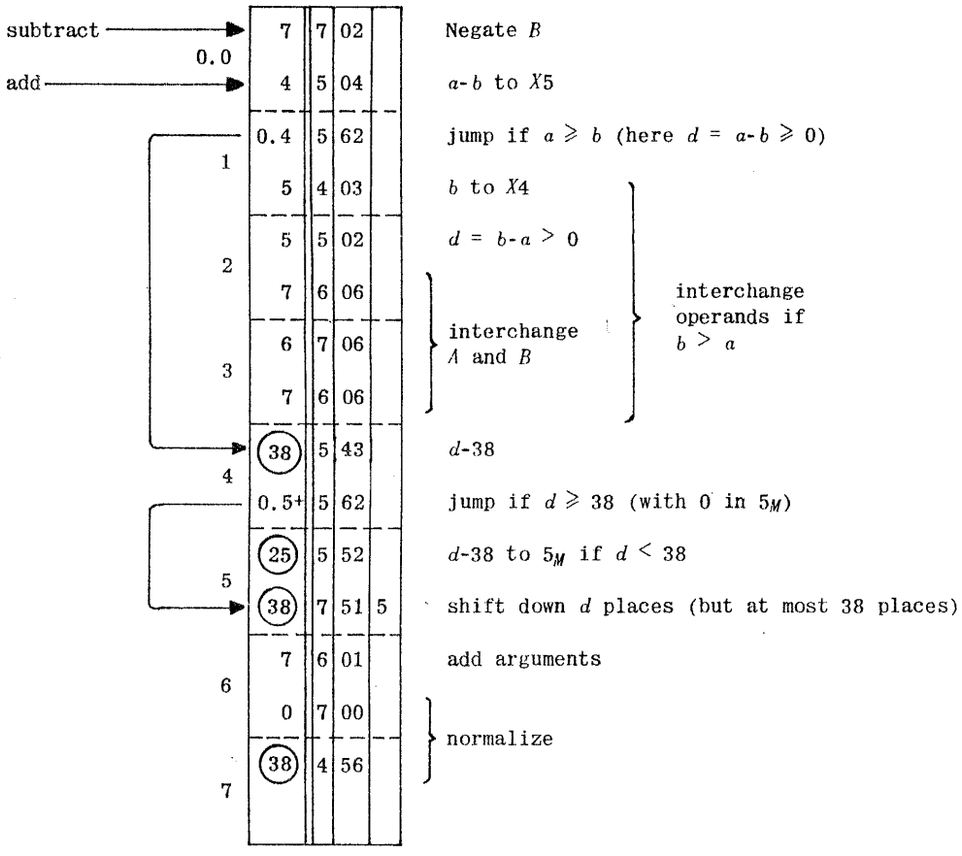
$$\;=\; (A + B.2^{b-a}).2^a.$$

We can therefore put the exponent c equal to a, the larger of the two original exponents. The process for subtraction is similar but we must change the sign of B first. The following sequence can be used to add or subtract two floating-point numbers (as above, the exponents are in *X*4 and *X*5 and the arguments are in *X*6 and *X*7). In this sequence d denotes $a-b$ or $b-a$, whichever is not negative; there is a trick to prevent a shift of more than 38 places. This sequence shows a good application of a modified shift order, perhaps the most important one.

---

† The sequence of events when a 56-order is obeyed is roughly as follows: the exponent enters the mill and 2 is added to it; the double-length argument is shifted up (if necessary) until its two most-significant bits are different, it is then halved; the exponent is reduced by 1 for every shift (including the last) and is finally sent back to its accumulator; the value of *N* is tested and, if non-zero, reduced by 1 for each doubling of the argument; if *N* is ever zero the order is immediately terminated by halving (*pq*) and subtracting 1 from the exponent. Overflow of the exponent is sensed only at the end of the order, when the exponent is sent back to its accumulator. The net result if *N* is 0 is that (*pq*) is always halved and one is added to the exponent. If *X* is 7 then p' is the left half of the normalised (*pq*) and $q' = q-\mu$. If *X* is 6 then $p' = p-\mu$ and q' is the right half of the normalized (*pq*). The 56-order can be modified in the same way as the other orders of group 5, but this is only rarely useful. The time taken is similar to that of a shift order plus two word-times, i.e. a total of $\mu + 5$ word times if it is an *a*-order, or $\mu + 4$ if it is a *b*-order.

†† Strictly speaking we should round the argument as was done after a multiplication since the normalize order may have shifted the quotient down one place, thus putting one binary digit into *X*7.



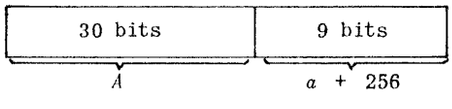
We again assume that the result is not zero. This may occur however because of exact cancellation, and we may get nearly complete cancellation of the two arguments leaving a result much smaller than either of the operands (this can happen with normalized operands only if the two exponents are equal or differ by 1). In this way serious loss of accuracy can arise.

The sequences of orders given above are meant to illustrate some of the ideas which can be applied to floating-point work. They are not complete because the number zero has not been provided for. In these sequences we have assumed that the arguments and exponents of the operands are available in certain accumulators. In practice it is usually wasteful to allocate two words to each floating-point number. This is because the exponent is seldom very large in absolute value. It would be unusual, for example, to have a number as big as 2^{100} (which corresponds to about 10^{30}) or as small as 2^{-100}. We can therefore represent the exponent quite adequately with only 8 or 9 binary digits. Again, in most applications the argument need not be given a whole 39-bit word; this is because it is normalized, which means that nearly all its digits are significant digits - we do not get strings of zeros or ones before the first significant digit as we do in fixed-point work. It is therefore possible for the argument and exponent to be packed into a single word; usually the argument occupies the first 30 binary digits, and the remaining 9 represent the exponent. With 9 bits to represent the signed exponent, a, this can be allowed to assume values in the range

$$-256 \leqslant a \leqslant 255.$$

This corresponds to allowing numbers to get as large as about 10^{76}. (This is about the number proposed by Eddington as the total number of protons in the universe!) The argument is represented with a precision corresponding to a little more than 8 *significant* decimal digits.

A standard *packed floating-point number* is represented by a single word in which digits 0 to 29 represent A, the argument of the number, directly: and digits 30 to 38 represent $a + 256$, thus:-



There are several advantages to be gained by packing $a + 256$ into the word, rather than a; the main point is that $a + 256$ is never negative. Such a number will of course, have to be split up, or unpacked, into its component parts before we can do any arithmetic with it. The result of an arithmetical operation will also have to be packed up into this form before being stored. These packing and unpacking operations are simplified if the exponent is held as $a + 256$, and the extra complication in the arithmetical sequences is slight (in fact the addition and subtraction sequence given above need not be altered at all).

There are a number of Library subroutines and programmes which handle these packed floating-point numbers; there are subroutines for input and output, for arithmetical operations, elementary functions and certain processes. With many of these the number of binary digits, n, used to represent the exponent is determined by a preset-parameter and can be given any reasonable value (with all these routines there is an optional parameter-list to set n equal to 9, the most generally used value). If n bits are used, they represent

$$a + 2^{n-1},$$

where a is the exponent which can take values in the range

$$-2^{n-1} \leqslant a \leqslant 2^{n-1} - 1.$$

The number zero is represented in all these routines by an argument of zero and an exponent which is as small as possible. Thus if 9 bits are used for the exponent (n = 9), zero is represented in the form

$$0.2^{-256},$$

which is called the *standard zero*. The word representing this has 0 in the part allotted to the argument; the exponent digits represent 256 - 256 = 0 and are also zero; the word therefore has 0 in every digital position and is a null word. In fact the standard floating-point zero is represented by the same word as a fixed-point zero. We can still use *X*0 to supply zero even if we are doing floating-point work. The first four jump orders (60 to 63) also have their usual effects, even if the number being tested is a packed floating-point number.

If, during the course of a calculation, the exponent of some quantity becomes larger than 255 (or, more generally, larger than $2^{n-1} - 1$ if n bits are used for the exponent) then *floating-point overflow* is said to have occurred. This is an unlikely event and is usually due to a mistake in the programme. If an exponent less than -256 (or, in general, -2^{n-1}) is obtained this is called *underflow;* it is then usual to set the result equal to a standard zero. If there is exact cancellation of the arguments the result is also made into a standard zero.

We give below an example of a complete floating-point programme. This is designed to read in a list of k numbers x_i followed by a single number y from an input data tape and to evaluate and print the function

$$z = \sqrt{\{y(x_1 + x_2 + x_3 + \ldots + x_k)\}} = \sqrt{\{y \sum_{i=1}^{k} x_i\}}.$$

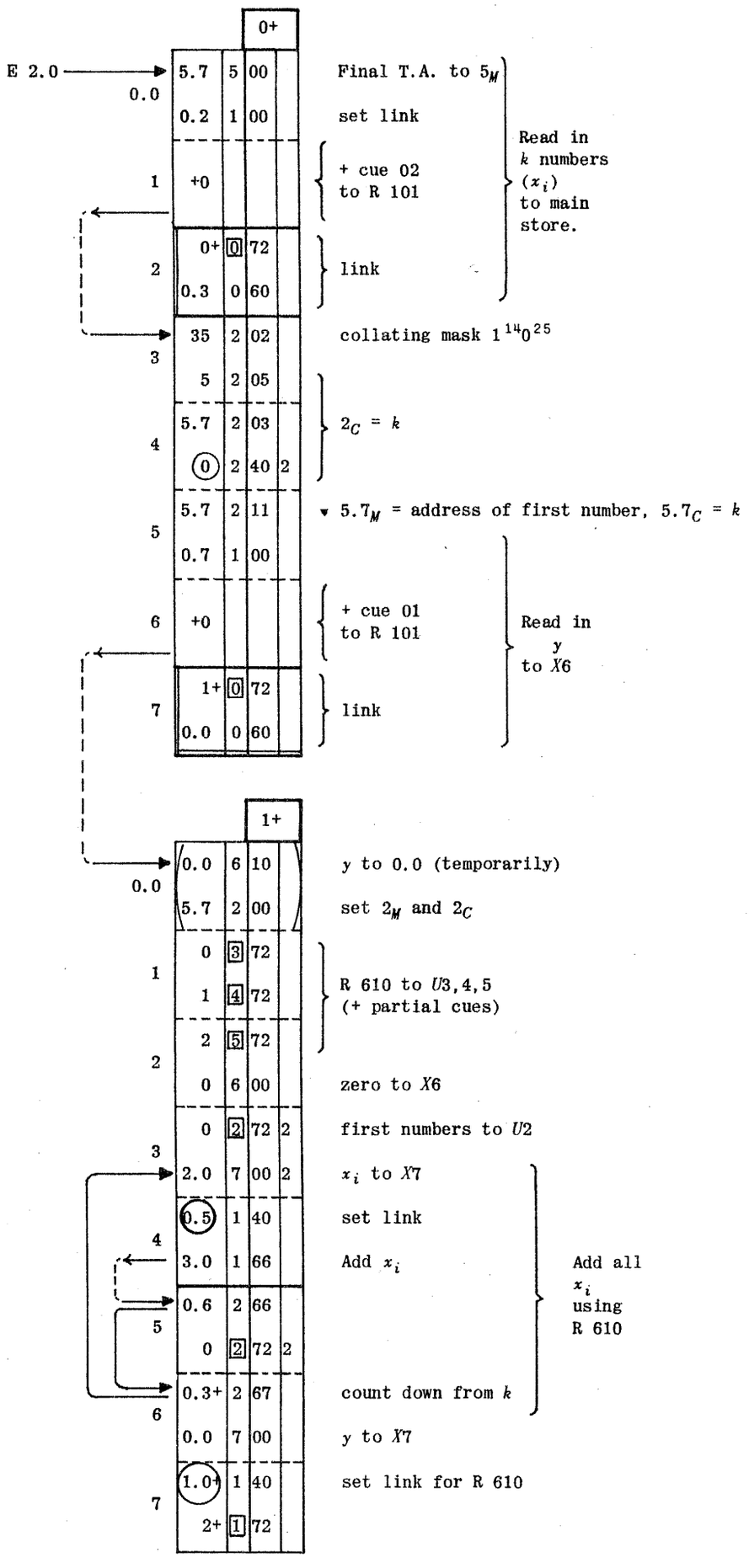
The value of k is determined by counting the numbers x_i read from the tape; after the last of them (x_k) an L is punched. This is followed by the single number y. These numbers are all punched in their usual form with the decimal point where required among the digits; they are read in by Library subroutine R 101. The floating-point additions and the multiplication are done by R 610 and the square-root is found by R 611. The answer is finally printed by R 11 and there is then a loop stop.

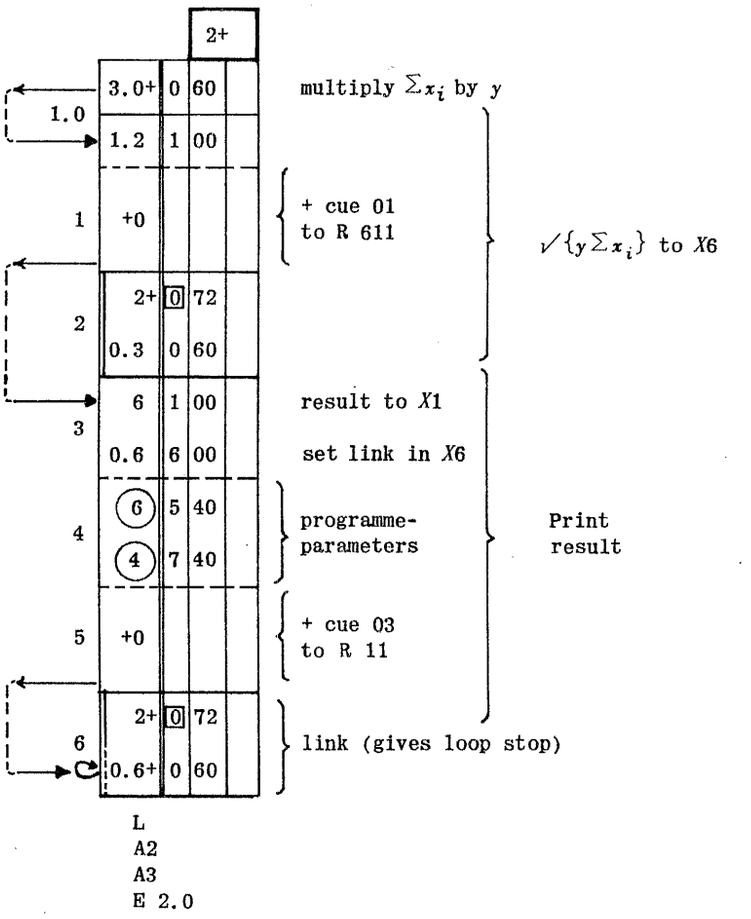
When R 101 is used first in this programme it reads in numbers until the L on the data tape is found. Each of these numbers is stored in the location whose address is 5_M; this is increased by unity after each number has been stored. Note the computing store links and partial cues used with R 610.

```
D
N
FUNCTION Z
A1
```

R 0	1	-0	2
101	-	01	-
R 0	6	-0	1
101	-	01	-
R 1	1	-0	1
610	-	01	-
R 1	1	-0	2
610	-	01	-
R 1	2	-0	1
610	-	01	-
R 2	1	-0	1
611	-	01	-
R 2	5	-0	3
11	-	01	-

} Calls for cues.


0+
E 2.0
0.0
5.7 5 00
0.2 1 00
Final T.A. to 5_M
set link
1
+0
+ cue 02 to R 101
Read in k numbers (x_i) to main store.
2
0+ 0 72
0.3 0 60
link
3
35 2 02
collating mask $1^{14}0^{25}$
5 2 05
4
5.7 2 03
0 2 40 2
$2_C = k$
5
5.7 2 11
5.7_M = address of first number, $5.7_C = k$
0.7 1 00
6
+0
+ cue 01 to R 101
Read in y to $X6$
7
1+ 0 72
0.0 0 60
link
1+
0.0
0.0 6 10
5.7 2 00
y to 0.0 (temporarily)
set 2_M and 2_C
1
0 3 72
1 4 72
2 5 72
R 610 to $U3,4,5$ (+ partial cues)
2
0 6 00
zero to $X6$
0 2 72 2
first numbers to $U2$
3
2.0 7 00 2
x_i to $X7$
0.5 1 40
set link
4
3.0 1 66
Add x_i
Add all x_i using R 610
5
0.6 2 66
0 2 72 2
6
0.3+ 2 67
count down from k
0.0 7 00
y to $X7$
7
1.0+ 1 40
set link for R 610
2+ 1 72



Note that in this programme the main set of numbers x_i is placed in the main store in consecutive locations starting at the final value of the Transfer Address at the end of the input stage of the programme. We thus avoid fixing any main store addresses; indeed the only absolute address in the whole programme is that in the final E-directive. This means, however, that we cannot safely restart the programme by a simple E-directive; but we can easily use instead a pair of directives such as the following, if the need should arise:

```
T 100.0
E   2.0.
```

9.2 Interpretive and conversion programmes

Floating-point work is an example of what is often called *programmed arithmetic*. We have to obey quite a sequence of orders, in fact a subroutine, to do even an addition or a subtraction. This is because the numbers on which we are operating are not the simple numbers round which the computer was built.† There are other examples of awkward kinds of quantity, complex numbers, or double-length or multi-length numbers for example, where the position is similar. By the use of carefully written subroutines most of the serious difficulties can be removed, and the main programme becomes almost entirely made up of organisational (or *red tape*) orders setting links and parameters, calling in the subroutines in the appropriate way, effecting the transfers of programme and numbers, counting, modifying and so on. The actual programme is in fact quite complicated, even if the original flow-diagram was simple. The maze of organisational orders may conceal the essential simplicity of structure of the programme; so much, in fact, that it may become quite difficult to prepare, test and correct it. Yet an attempt to write the programme without the use of proper subroutines would introduce other, more intractable, problems and make the programme even more lengthy.

A possible approach is to sketch out the programme first in some suitably chosen abbreviated notation; this can later serve as a guide while the actual programme is written. For example, in floating-point work we could number all the locations used to hold floating-point numbers (assuming these are all held in the main store); the contents of these locations could be referred to as variables and written v_0, v_1, v_2, v_3,.. We could then indicate the floating-point addition of say v_5 and v_{12} by writing down

$$v_8 = v_5 + v_{12},$$

which means that the 8th variable is to be set equal to the sum of the 5th and the 12th. A whole programme may be written down in this sort of way, one "order" below another. Later on we can go through it and translate each of the "orders" in it into the proper computer orders to extract the variables, set any parameters and links, call in the subroutine appropriate to the operation in the "order" and dispose of the result. If we suitably choose the way in which the original "orders" were written down, this translation process can be made systematic and subject to definite rules; in fact it can be carried out by someone who is not familiar with the technique of programming, provided only

† Some machines, e.g. the Ferranti Mercury Computer, have built-in floating-point operations; on such machines a single order can be used for an arithmetical operation.

that the rules to be followed are precise, clearly expressed and cover all cases. Indeed we can get the computer itself to effect the translation, if we supply a suitable programme incorporating all the rules.

The general procedure is to write our programme in a convenient language well adapted to the nature of the work, and to punch it on tape much as it is written. This is the *pseudo-programme;* the tape is then read by the computer under the control of a special *conversion programme* and converted into a programme of machine orders†. The result of the conversion is an ordinary programme and the computer can either obey it immediately or punch it out (probably in binary) for later use. The Initial Orders are in effect a rudimentary conversion programme, because we write and punch our orders and numbers in a fairly convenient way, and these are converted into the binary form required by the computer. There is in fact quite a lot of conversion done by the Initial Orders (consider the effects of reading A3, for example), but a full conversion programme does a great deal more.

There is another way of handling the problem of translating a programme from a convenient language for a particular problem into the ordinary machine language. As before, the programme is punched as written and read in, but this time the "orders" on the tape are not converted into machine orders; instead they are stored with only the minimum of conversion. For example the "order" given above could be stored in the form of three addresses specifying the variables together with a single integer specifying the function - all these could be packed into one or two words. When the end of the tape is reached the whole programme will have been stored, but only in a simple coded form corresponding closely to the written pseudo-programme. At this point a special programme is called in, which can extract each order from the store and interpret it, i.e. carry out the operation called for. For example, the addresses in the order could be used as modifiers to select the operands and to write away the result; the function number could be used to modify a special cue which would then bring in the appropriate subroutine. In this *interpretive* scheme the "orders" are each carried out as soon as they have been translated, and the programme then passes on to interpret the next "order". Thus if there is a loop in the programme being interpreted, each "order" in the loop will be translated each time the loop is traversed; whereas in a conversion scheme the translation process is done once and for all before any of the "orders" are obeyed. An interpretive scheme is consequently slower, in general, then a conversion scheme. On the other hand each "order" for an interpretive scheme occupies only a word or two in the store whereas the converted programme of machine orders resulting from the use of a conversion scheme will occupy a considerable amount of space. The extra time spent interpreting orders in an interpretive scheme is the penalty paid for compactness of the programme; there are circumstances when this time does not matter, for example when the individual operations carried out take much longer than the interpretation time.

In either type of scheme a convenient external language is ultimately translated into the internal machine language. There are other ways of doing this. All such schemes are often described collectively as *automatic programming,* though perhaps *automatic coding* is a term to be preferred since the calculation usually has to be carefully planned. Some automatic programming schemes can be considered as directed at removing some difficulty or disadvantage in a particular computer, but it is not easy to make any firm definitions. Some of these schemes are so successful that they are regarded as the normal way of putting a programme into the computer; usually, however, the machine-made result is inferior to that produced by hand. An experienced programmer can make use of many special features of a problem to make his programme efficient; it is very difficult to cater for such individual features in an automatic programming scheme designed as a rule to handle a wide variety of problems. In general, therefore, the use of an automatic programming scheme can be expected to slow down the computer - and by a factor which may be as large as 100 in unfavourable cases. The programming time for a particular problem may well, however, be reduced in about the same ratio and the programme will often be correct at the first attempt. As an important consequence, the *overall* time from beginning to study a problem to the obtaining of accurate results is usually dramatically reduced, perhaps from months to hours. This is especially true of what might be called small problems, which can be fairly simply stated and do not involve large masses of data or results.

The kind of external language chosen for one of these schemes depends very much on the operations to be carried out, the nature of the operands, the frequency with which the scheme is to be used and the programming experience of the users. It is, of course, restricted to the characters which can be punched in the paper tape and printed. A number of schemes are available for Pegasus and some of these are described below. If a great deal of a certain kind of work is to be undertaken then it is worth considering the writing of some special automatic programming scheme. Such a scheme, if written with some special aim in view, need not be a very elaborate one; and some technique might well be found peculiarly well adapted to the subject. For example, what is known as a *tape-steered programme* finds some applications: this kind of programme is designed to read in an "order", or some parameters, or perhaps just a J-directive from the input tape, and to carry out some operation (on internally stored information) the nature or details of which depend on the information read in. When the operation is complete another "order" or other information is read in. A scheme like this is reasonably flexible and has the advantage that the pseudo-programme does not occupy any storage space in the computer; though of course loops are impossible.

The user of an automatic programming scheme will usually never know into what machine orders his programme is ultimately translated; he is, as it were, removed or insulated from the actual machine and thinks only in terms of the external language of the scheme. In fact to him the computer appears as quite a different machine, perhaps capable of doing the most complex operations when it obeys a single order; he need not know anything of ordinary programming at all††. There is, however a difficulty which can arise if the user's programme and that actually obeyed by the computer are widely different: if the programme does not work then it may be quite difficult to find out why. However, the programme is much more likely to be free from error than an ordinary computer programme,

---

† A machine order is simply an ordinary order as obeyed by the computer.

†† An amusing distinction has been drawn between the *Primitives* of programming, who think always in binary or octal and like to keep as close as possible to the computer way of doing things, and the *Space Cadets*, who like to punch algebraic equations, English words and integral signs on their tapes.

and most schemes provide facilities for extra informative printing at the start which can later be omitted.

9.3 The Autocode

The Pegasus Autocode is a conversion scheme which greatly facilitates the programming of certain kinds of problem†. It is especially suitable for such technical and scientific calculations as the evaluation of formulae, the tabulation of special functions and small *ad hoc* calculations; but it can also be used in commercial and industrial work for the testing out of complex flow-diagrams, and has many other applications. The user of the Autocode need not know anything of ordinary Pegasus programming, though there are certain facilities which he cannot otherwise use.

To use the Autocode on a particular problem the calculation must first be broken down into a sequence of steps, each of which is the calculation of a number from one or two previously calculated numbers. Each step is written as an order or *instruction* and the whole sequence forms the *programme* of the calculation. The programme is punched, as written, on a teleprinter or a keyboard perforator and the resulting tape is then read into the computer by the Autocode scheme (Instruction Input section), which converts the programme into a suitable internal form, which is stored. At the end of the tape are some special symbols which cause the programme to be obeyed. At various stages while it is being obeyed the programme can cause printing of results (they are actually punched, as usual, and printed later), or it can read in numerical data from tapes in either of the tape-readers.

There are two kinds of numbers which can be handled by Autocode instructions:

(a) *Variables*, denoted by $v0$, $v1$, $v2$, $v3$, ... These are actually standard, packed, floating-point numbers with 9 bits for the exponent, and may therefore lie in the range

$$-2^{254} \leqslant v \leqslant 2^{254}\,(1 - 2^{-28}).$$

They are represented with a precision corresponding to between 8 and 9 significant decimal digits, except that numbers in the range

$$-2^{-257} \leqslant v \leqslant 2^{-257}\,(1 - 2^{-28})$$

are treated as zero.

(b) *Indices*, denoted by $n0$, $n1$, $n2$, ... These are signed integers in the range

$$-8191 \leqslant n \leqslant 8191.$$

They are actually stored in the modifier-position of words and are handled in fixed-point form.

Most of the numbers entering into a calculation are variables; the indices are introduced as auxiliary quantities, mostly to facilitate counting and modifying. The v and n symbols are available in figure-shift on the teleprinter; the digits which follow them are to be thought of as suffixes although they cannot be printed as such on the teleprinter. The total available number of variables and indices is adjustable (by preset-parameters, in fact), but they are normally fixed so that they are

(a) 1380 variables written $v0$, $v1$, ..., $v1379$, and

(b) 28 indices written $n0$, $n1$, ..., $n27$.

Most of the Autocode instructions take the form of an equation giving the new value of a variable (or index) in terms of one or two numbers or previously calculated variables (or indices). For example, the instruction

$$v1 = v2 + v3$$

means that the new value of $v1$ is to be the sum of $v2$ and $v3$. The instruction

$$v5 = v5 + v1$$

means that the new value of $v5$ is to be the sum of $v1$ and the old value of $v5$. Numbers may be put instead of variables on the right of the equality sign; thus the instruction

$$v8 = v2 + 35.771$$

means that the new value of $v8$ is the sum of $v2$ and 35.771. As an example of a sequence of these instructions dealing with variables let us evaluate

$$31.41v5 - v92^2 + \{(-6.535v8 + v97 - 9)/v323\},$$

and put the result in $v0$. The following sequence of instructions can be used (here we use $v1$ as working space):

$$\begin{aligned} v0 &= 31.41 \times v5 \\ v1 &= v92 \times v92 \\ v0 &= v0 - v1 \\ v1 &= -6.535 \times v8 \\ v1 &= v1 + v97 \\ v1 &= v1 - 9 \\ v1 &= v1/v323 \\ v0 &= v0 + v1 \end{aligned}$$

† The scheme is based on that of Dr. R.A. Brooker for the Ferranti Mark 1 computer at Manchester University, although it differs greatly in detail.
Brooker, R.A. *An attempt to simplify coding for the Manchester Electronic Computer*, Brit. J. App. Phys. (1955), **6**, p.307.

Note how each instruction involves only two variables or numbers on the right. We can also use instructions like the three following ones, which have only one variable or number on the right:

$$v8 = v4, \qquad v6 = -v2, \qquad v6 = 43.$$

There are exactly analogous instructions for handling indices, for example:

$$n3 = n8 + n3, \qquad n2 = -79 - n5,$$

but it should be remembered that indices can take only integral values. In general we cannot mix indices and variables in the same instruction; a few simple instructions like this have, however, been provided.

The purely arithmetical instructions in the Autocode are summarized in Table 9.1. In this table $v1$, $v2$ and $v3$ are given simply as representative variables, and $n1$, $n2$ and $n3$ represent any three indices. The instruction $n1 = n2/n3$ gives the integer quotient when $n2$ is divided by $n3$, and the instruction $n1 = n2*n3$ gives the corresponding remainder. (The analogous instructions with a minus sign give the same numbers with their signs changed). In any of the instructions any variable (or index) on the right-hand side of the equality sign may be replaced by a positive number (or integer).

Certain elementary *functions* may be evaluated by a single instruction; for example we can write

$$v6 = \text{SQRT } v9$$

to evaluate a square root, or

$$v3 = \text{-SIN } v99$$

to evaluate a sine. The available functions are tabulated in Table 9.2; they all apply to variables only, except for MOD which can be used to form the modulus of an index or a variable. Again, any variable written on the right may be replaced by a positive number; for example:

$$v37 = \text{LOG } 5.1058209.$$

Variables		Indices	
$v1 = v2$	$v1 = -v2$	$n1 = n2$	$n1 = -n2$
$v1 = v2 + v3$	$v1 = -v2 + v3$	$n1 = n2 + n3$	$n1 = -n2 + n3$
$v1 = v2 - v3$	$v1 = -v2 - v3$	$n1 = n2 - n3$	$n1 = -n2 - n3$
$v1 = v2 \times v3$	$v1 = -v2 \times v3$	$n1 = n2 \times n3$	$n1 = -n2 \times n3$
$v1 = v2/v3$	$v1 = -v2/v3$	$n1 = n2/n3$ (quot)	$n1 = -n2/n3$
		$n1 = n2*n3$ (rem)	$n1 = -n2*n3$
Mixed:	$n1 = v2$	$n1 = -v2$ (nearest integer)	
	$v1 = n2$	$v1 = -n2$	
	$v1 = n2/n3$	$v1 = -n2/n3$	

Table 9.1. *Autocode instructions - arithmetic.*

$v1 =$ MOD $v2$	$v1 =$ -MOD $v2$	modulus
$v1 =$ INT $v2$	$v1 =$ -INT $v2$	integral part
$v1 =$ FRAC $v2$	$v1 =$ -FRAC $v2$	fractional part
$v1 =$ SQRT $v2$	$v1 =$ -SQRT $v2$	square root
$v1 =$ SIN $v2$	$v1 =$ -SIN $v2$	circular functions
$v1 =$ COS $v2$	$v1 =$ -COS $v2$	
$v1 =$ TAN $v2$	$v1 =$ -TAN $v2$	
$v1 =$ CSC $v2$	$v1 =$ -CSC $v2$	
$v1 =$ SEC $v2$	$v1 =$ -SEC $v2$	
$v1 =$ COT $v2$	$v1 =$ -COT $v2$	
$v1 =$ ARCSIN $v2$	$v1 =$ -ARCSIN $v2$	inverse circular functions
$v1 =$ ARCCOS $v2$	$v1 =$ -ARCCOS $v2$	
$v1 =$ ARCTAN $v2$	$v1 =$ -ARCTAN $v2$	
$v1 =$ LOG $v2$	$v1 =$ -LOG $v2$	natural logarithm
$v1 =$ EXP $v2$	$v1 =$ -EXP $v2$	exponential
$v1 =$ EXPM $v2$	$v1 =$ -EXPM $v2$	exp (-$v2$)
$n1 =$ MOD $n2$	$n1 =$ -MOD $n2$	modulus of index

As usual, Autocode instructions are obeyed in the order in which they are written, until a *jump instruction* is encountered. If a jump occurs, the next instruction to be obeyed is identified by its *label*, which is simply a small positive integer written in front of the instruction and separated from it by a right bracket. For example,

$$7)\quad v4 = -9.44/v5$$

is labelled 7. Any instruction can be labelled, but the same label should not be used twice. If required we can attach two or more labels to the same instruction, thus

$$9)2)\quad v3 = \text{LOG } v0$$

The first instruction in an Autocode programme is always automatically labelled 0 (there is no need to write this label in). A jump instruction always includes an arrow; the simplest is the unconditional jump, for example

$$\rightarrow 7$$

is the Autocode instruction meaning: jump to the instruction labelled 7. Most of the jumps are conditional, however; consider the following instruction

$$\rightarrow 8,\ v1 \geqslant v6$$

This means: jump to the instruction labelled 8 if $v1 \geqslant v6$, otherwise carry on with the next instruction as usual. The following three instructions resemble this one closely:

$$\rightarrow 8,\ v1 \geqslant -v6$$

$$\rightarrow 8,\ -v1 \geqslant v6$$

$$\rightarrow 8,\ -v1 \geqslant -v6$$

These four instructions can be summarized as

$$\rightarrow 8,\ \pm v1 \geqslant \pm v6$$

All the available jump instructions are summarized in this way in Table 9.3; as usual the numbers 1, 2, 3 merely stand for any convenient numbers. There is a great variety of jumps, and the scheme is consequently very flexible. In a jump instruction, any variable may be replaced by a number, or any index by an integer. The following are examples of this:

$$\rightarrow 4,\ 0 \geqslant v62$$

$$\rightarrow 8,\ n6 \neq -20$$

The jumps including tests of approximate equality are intended to allow for the effects of the rounding errors inevitable in most floating-point work; care should be taken to set the value of $n0$ before using them (e.g. by an instruction such as $n0 = 20$).

→ 1 (unconditional jump)	
→ 1, $\pm v2 \geqslant \pm v3$	→ 1, $\pm v2 > \pm v3$
→ 1, $\pm v2 = \pm v3$	→ 1, $\pm v2 \neq \pm v3$
→ 1, $\pm n2 \geqslant \pm n3$	→ 1, $\pm n2 > \pm n3$
→ 1, $\pm n2 = \pm n3$	→ 1, $\pm n2 \neq \pm n3$
→ 1, $\pm v2 = * \pm v3$	(jump if approximately equal; more exactly, jump if the two variables agree to $n0$ significant binary digits - i.e. to about $0.3 \times n0$ significant decimal figures)
→ 1, $\pm v2 \neq * \pm v3$	(jump if not approximately equal)

Table 9.3 *Autocode instructions - jumps.*

As an example, let us suppose that we are given two positive numbers $v1$ and $v2$ and we have to replace them, respectively, by

$$\tfrac{1}{2}(v1 + v2) \quad \text{and} \quad \sqrt{(v1 \times v2)}.$$

This process is to be repeated until the two quantities are nearly equal, say until they agree in their first 20 binary places. The result is in fact an approximation to the Gauss arithmetico-geometric mean of the two quantities. The following sequence of orders can be used.

```
    n0 = 20
3)  v3 = v1
    v1 = v1 + v2
    v1 = 0.5 × v1
    v3 = v3 × v2
    v2 = SQRT v3
    →3, v1 ≠ * v2
```

The first instruction sets $n0$ for subsequent "approximately equal" tests. The variable $v3$ is used to hold intermediate quantities, it is first used to keep the value of $v1$.

The technique of *modification* can be applied to any Autocode instructions involving variables. For example, we can write the instruction

$$v8 = v998 + vn6$$

which means that the new value of the 8th variable is the sum of the 998th variable and the $n6$-th variable. The $n6$ in the variable $vn6$ is to be thought of as a suffix (v_{n_6}) though it cannot be typed in this way on a teleprinter. Thus the instruction written above is equivalent to the instruction

$$v8 = v998 + v28$$

if $n6 = 28$ at the time the instruction is obeyed. Any variable can have this kind of suffix, for example

$$vn0 = vn3 + vn4$$

We can also write more complicated kinds of suffix, for example

$$v(8 + n2) = v(53 + n4) + v(-2 + n11)$$

in which each bracketed expression is regarded as a suffix. The integer (h, say) in this kind of suffix must be written before the n and can have any value in the range

$$-2048 \leqslant h \leqslant 2047,$$

but the result after the addition of the index n must not be negative. The index must be *added*, never subtracted.

We can also have modified jump instructions, of which the following are specimens:

```
→ n7, v0 > v67
→ n9, -n8 ⩾ -21
→ (-4 + n8), n0 ≠ 8
```

Instructions like these are not often needed.

A small group of instructions is provided for the *input of numbers*. A simple one is the following:

$$v6 = \text{TAPE}$$

This causes $v6$ to be set equal to a number read from the input tape in the main tape-reader. The instruction

$$v5 = \text{TAPE } 13$$

causes 13 numbers to be read in and placed in $v5$, $v6$, ..., $v17$. The instruction

$$v28 = \text{TAPE } *$$

causes numbers to be read in and placed in $v28$, $v29$, $v30$, ... until an L-directive is read. There are a number of other input instructions; they are all included in Table 9.4. It should be noted that *any* input instruction has the following properties:

(a) input ceases and the next instruction is obeyed when L is read,
(b) $n0$ is always put equal to the number of numbers read in.
(c) directives N, Z and Q are recognized in addition to L; Q fixes a decimal block exponent (it cannot be used with indices).

Suppose as an example, that the instruction

$$v1 = \text{TAPE } *$$

is obeyed and that the following tape is in the main tape-reader:

```
N
DATA 3
+1.5
Q +10          -0.657      +0.9876
Q -15
+0.55321
- 13
L
```

When the instruction has been obeyed the following will be stored:

$$v1 = +1.5, \quad v2 = -0.657 \times 10^{10}, \quad v3 = +0.9876 \times 10^{10},$$

$$v4 = +0.55321 \times 10^{-15}, \quad v5 = -13 \times 10^{-15}, \quad n0 = 5.$$

The name DATA 3 will be printed when the tape is read. Each number on the tape is punched as written, immediately preceded by its sign, and must be terminated by Sp or CR LF (if indices are to be read in, by an instruction like $n3$ = TAPE, for example, they must be punched without a decimal point).

$v1$ = TAPE $v1$ = TAPEB	read in one number and set in $v1$
$n1$ = TAPE $n1$ = TAPEB	read in one integer and set in $n1$
$v1$ = TAPE $n2$ $v1$ = TAPEB $n2$	read in $n2$ numbers and set in $v1$, $v2$, ...
$n2$ = TAPE $n1$ $n2$ = TAPEB $n1$	read in $n1$ integers and set in $n2$, $n3$, ...
$v1$ = TAPE * $v1$ = TAPEB *	read in numbers up to L on tape; set in $v1$, $v2$, ...
$n1$ = TAPE * $n1$ = TAPEB *	read in integers up to L on tape, set in $n1$, $n2$, ...

TAPE instructions refer to the main tape-reader,
TAPEB to the second tape-reader.

Table 9.4 *Autocode instructions - input*

There are two main methods of *output* in the Autocode. The most used consists of a special instruction, of which the following is an example,

PRINT $v6$, 3045

This instruction causes the value of $v6$ to be printed in a way determined by the *style-number* 3045. To find the style-number to write in a print instruction we must first decide whether the number is to be printed on a new line or on the same line as the previous number, and whether it is to appear in floating-point form (i.e. as a number and a decimal exponent) or in fixed-point form (with the decimal point in its proper position). This determines the first digit (a) of the style-number, which can be found from the following table:

	Floating-point	Fixed-point
Print on a new line (CR LF ϕ before number)	$a = 1$	$a = 3$
Print on the same line (Sp before number)	$a = 2$	$a = 4$

The rest of the style-number is determined by the number of digits (b) to be printed before the decimal point and after it (c); the style-number is in fact

$$1000a + 20b + c,$$

so that style 3045 causes the number to be printed in fixed-point (or normal) form on a new line with *two* digits before the decimal point and five after it. The number is preceded by its sign and is, of course, properly rounded. If desired we can write the instruction in the form

PRINT $v3$, $n9$

for example, when the current value of $n9$ is taken as the style-number. We can also use a similar

instruction for printing indices; here only the first digit of the style-number matters and the index is always printed as a 4-digit signed integer. Thus the instruction

PRINT *n*3, 4000

causes the value of *n*3 to be printed, preceded by a single space. In all printing, left-hand zeros in the integral part are, of course, suppressed (i.e. replaced by spaces).

An alternative method of output is mainly useful to provide extra printing in the development stage of an Autocode programme. We can write XP (for printing on a new line) or SP (for the same line) *before* an instruction; this will cause the result of the instruction to be printed. Thus the instruction

XP *v*9 = *v*3 + *v*5

causes the new value of *v*9 to be printed on a new line after the rest of the instruction has been obeyed. The printing caused by XP and SP is optional; it can be suppressed by depressing the sign-digit key of the handswitches. We can alternatively put X or S in front of an instruction; these cause printing (non-optional) of CR LF or a single Sp, respectively, and can be used to lay out the printing. We may put XP, SP, X or S in front of any instruction which carries out an arithmetical operation or evaluates a function.

The output instructions are given in detail in Table 9.5.

PRINT *v*1, *n*2	Print *v*1 in style number *n*2 = 1000*a* + 20*b* + *c*.
If *a* = 1	print CR LF ϕ then number in floating-point form with *b* digits before point in argument and *c* digits after point, then Sp and two-digit exponent, then Sp Sp. Width of printing 8 + *b* + *c*.
If *a* = 2	print Sp then number as for *a* = 1. Width 9 + *b* + *c*.
If *a* = 3	print CR LF ϕ then number in fixed-point form with *b* digits before point and *c* digits after. Width 2 + *b* + *c*. But if *b* is too small print in floating-point form as if *a* = 1.
If *a* = 4	print Sp then number as for *a* = 3. Width 3 + *b* + *c*. If *b* is too small print as if *a* = 2.
PRINT *n*1, *n*2	Print *n*1 in style number *n*2 = 1000*a*
If *a* = 3	print CR LF ϕ then 4-digit index. Width of printing 5.
If *a* = 4	print Sp then 4-digit index. Width = 6.
XP	before an instruction (arithmetical or functional) the result of which is a *variable:* Obey instruction, then print CR LF ϕ and result in floating-point form; the argument to 9 decimals with its first significant digit after the point, then Sp and two-digit exponent, then Sp Sp. Width of printing 18. There is no printing if *H*0 = 1.
XP	before an instruction the result of which is an *index:* Obey instruction, then print CR LF ϕ and result as four-digit integer. Width 5. There is no printing if *H*0 = 1.
SP	is similar to XP but Sp is printed instead of CR LFϕ. Width 19 for variables, or 6 for indices.
X, S	before any arithmetical or functional instruction: Print CR LF or Sp respectively, after obeying the instruction.

Table 9.5 *Autocode instructions - output.*

The following are typical results of some instructions causing output.

PRINT *vn*10, 1064	CR LF ϕ + 345.6789 Sp Sp + 1 Sp Sp
PRINT *v*3, 2064	Sp - 100.0000 Sp Sp + 6 Sp Sp
PRINT *v*4, 2044	Sp + 12.3456 Sp + 12 Sp Sp
PRINT *v*9, 3062	CR LF ϕ - 123.45
PRINT *v*(1+*n*2), 4026	Sp + 9.876543
PRINT *n*2, 3000	CR LF ϕ Sp Sp + 12
PRINT *n*5, 4000	Sp - 7732
XP *v*4 = *v*3 + *v*2	CR LF ϕ + 0.12345678901 Sp Sp + 2 Sp Sp
SP *v*7 = *v*9 + 10	Sp - 0.98765432109 Sp + 11 Sp Sp
XP *n*3 = 1 + *n*3	CR LF ϕ Sp + 103
X *n*2 = 0	CR LF
S *n*4 = *n*7 - *n*1	Sp

There are a few other, miscellaneous, instructions in the Autocode. We shall describe all these later except for one simple one; the instruction

STOP

causes a 77-stop. If the Run key is operated the next Autocode instruction will be obeyed.

The *programme tape* for an Autocode programme is headed as follows:

```
D
N
(name of programme)
J 1.0
```

The tape up to and including the J-directive is read in by the Initial Orders in the usual way. The rest of the tape is then read in by the Autocode instruction-input routine (the Autocode itself must have been put into the store previously, of course). The autocode instructions making up the programme are then punched (as written) in their correct sequence after the J 1.0. They are read in and each one is converted into a block of ordinary machine orders and numbers and stored (actually in *B*300 onwards). At the end of the tape we must punch certain special symbols to cause the programme to be started. This is done by *brackets*.

An instruction or a group of instructions written in brackets is obeyed as soon as it has been read in. Since the first instruction in the programme is always automatically labelled 0 we can enter the programme (i.e. start obeying it) by punching

(→0)

at the end of the tape. This is an unconditional jump to the beginning of the programme, and since it is written in brackets it is obeyed the moment it has been read in. If we like we can punch a sequence such as the following on the tape:

```
(v1 = 1.76
 n2 = 10
  → 3)
```

This sequence is all read in but a note is made of the left bracket; when the right bracket is read the instruction after the left bracket is obeyed, and then the next, and so on. The last instruction is here an unconditional jump to the instruction labelled 3 and the main programme will be entered at this point. A sequence of instructions like this is called a *bracketed interlude*. If the last instruction of the interlude does not cause a jump then further instructions, punched after the interlude on the tape, will be read in and will obliterate the interlude. If we wish, the bracketed interlude can be quite a complicated programme including loops and print instructions, for example. In fact we can even put the whole of our programme in brackets, when it will be entered as soon as it has been read in; this technique is not recommended, however.

As an example of a complete Autocode programme we give below a programme to read in some numbers from a tape in the main tape-reader and to evaluate and print the sum of their squares, and also the square root of this number.

```
D
N
SUM OF SQUARES, SQUARE ROOT
J 1.0

STOP                              wait for number tape
v2 = TAPE *                       numbers to v2, v3, ... and set
                                        n0 = number of numbers
v0 = 0
2)v1 = v(1 + n0) × v(1 + n0)      square v(1 + n0)
v0 = v0 + v1                      accumulate sum in v0
n0 = n0 - 1                       count numbers
→ 2, n0 ≠ 0
PRINT v0, 1025                    print sum of squares
v0 = SQRT v0
PRINT v0, 2025                    print square root
STOP
(→ 0)                             enter programme
```

The instructions making up a programme are read into the store and converted at the rate of about 2 instructions per second, they are obeyed at the rate of about 15 per second. These figures are necessarily approximate.

As another illustration we give an Autocode programme to tabulate the function

$$z = \text{argsech } y = -\log\left[\{1 - \sqrt{(1 - y^2)}\}/y\right],$$

for $y = 0.01\ (0.01)\ 0.99$. We generate the current value of y by dividing an integer $n1$ by 100; $n1$ will start at 1 and go up to 99. This is to prevent the accumulation of rounding errors.† The numbers y and z are printed in two columns and an extra blank line is inserted (by printing CR LF) whenever y is a multiple of 0.05 (i.e. $n2$ is a multiple of 5).

```
D
N
TABULATE ARGSECH - AUTOCODE
J 1.0

  n1 = 1
1)n2 = n1 * 5          } n2 is the remainder when n1 is
  → 2, n2 ≠ 0          } divided by 5, if it is zero
  X n2 = 0             } we print CR LF
2)v1 = n1/100          y = n1/100
  PRINT v1, 3022       print y
  v2 = v1 × v1         y²
  v3 = 1 - v2          1 - y²
  v4 = SQRT v3         √(1 - y²)
  v5 = 1 - v4          1 - √(1 - y²)
  v6 = v5/v1           {1 - √(1 - y²)}/y
  v7 = -LOG v6         z
  PRINT v7, 4027       print z
  n1 = n1 + 1
  → 1, n1 ≠ 100
  STOP
  (→ 0)                enter the programme
```

As at present arranged there is room in the scheme for 594 Autocode instructions.†† This includes the instructions in the last bracketed interlude (in such an interlude the right bracket occupies the space of one instruction). Each instruction in fact occupies one block in the main store starting at B300. (The starting block can, however, be altered if necessary.)

We shall now describe a few other facilities provided in the Autocode scheme which may occasionally be useful. The instruction

```
TAPE
```

causes more instructions to be read in from the input tape (in the main tape-reader) and to be added at the end of the programme (in fact they overwrite the last bracketed interlude). This instruction can be used in a programme designed to do a rather complicated calculation on a few numbers or parameters. At the end of this calculation a TAPE instruction can be used to read in a short bracketed interlude, such as the following,

```
(n4 = 3
 v1 = 41.509
 → 6)
```

This interlude can set new values of the numbers or parameters and cause the calculation to be repeated.*

A variant of the TAPE instruction is written, for example,

```
TAPE 6
```

---

† We could alternatively have used a variable ranging from 1 to 99 and divided it by 100 to give y. This is also exact.

†† In the version of the Autocode for the 4096-word store there is room for only 210 instructions.

* The TAPE instruction should be carefully distinguished from the instructions such as $v1$ = TAPE and so on, which were described earlier; these latter read in *numbers* from the tape.

This reads in more instructions from the input tape and puts the first one over (i.e. in place of) the instruction labelled 6. Similarly the instruction

TAPE 6, 3

will place the first of the new instructions over the 3rd instruction after that labelled 6. We can also use instructions such as

TAPE *n*1

or

TAPE *n*1, *n*2

with similar effects.

It is sometimes desirable to stop obeying an Autocode programme and to start obeying ordinary Pegasus machine orders; after some special calculation has been done we may then wish to return to the Autocode programme, either at the next instruction or at some other specified instruction. There are consequently facilities for *exit to machine orders* and *entry from* machine orders. The instruction

→ M1682

means: start obeying machine orders at decimal address 1682 (i.e. at *B*210.2). The effect is similar to a J-directive with decimal address 1682, so that *B*210 will be transferred to *U*0, the next three blocks to *U*1, 2 and 3, and a jump will then occur to *U*0.2 (*a*-order). A link is set in *X*1 and a special word in *X*2; if the link is obeyed with *C*(2) undisturbed then the machine will return to the next Autocode instruction (i.e. the one after the →M1682). In this way we can jump to any *a*-order. Similarly the instruction

→ M1682, *v*2

will cause exit to machine orders at decimal address 1682, but in addition the variable *v*2 will be unpacked - *X*7 will contain its argument and *X*5 will contain 256 + its exponent. We can also use instructions such as

→ M *n*3, *v*5

or

→ M1891, *n*1

This last will leave *n*1 in 7_M. With any → M instruction the settings of the accumulators will be reproduced in *B*0. Entry from machine orders to an Autocode programme at the instruction labelled 15 (for example) can be done by putting 15 into *X*2 as an integer and obeying the cue

37 [4] 72

4.4 0 60

Before obeying this cue we can set a link in *X*1; this is obeyed by the Autocode instruction

→ L

If we wish to use machine orders together with an Autocode programme we have to know how the space in the main store is allocated. This is at present as follows:-

*B*1 to 111	Autocode scheme
*B*111.2 onwards	Labels (see below)
*B*124.0 to 127.3	Indices (*n*0 to *n*27 - one location each)
*B*127.4 to 299.7	Variables (*v*0 to *v*1379 - one location each)
*B*300 onwards	Instructions (one block each)

The labels each occupy one location, thus the word in *B*111.4 gives the address of the instruction labelled 2. If labels 0 to 25 are used, then locations *B*113.2 to 114.3 must not be used for machine orders. This allocation of the store is liable to alteration if extra facilities are included. It is probably safest to use for machine orders the space normally occupied by the higher-numbered variables.

The Initial Orders may be called in as a subroutine by the Autocode instruction

→ IO

A link is set which causes return to the next Autocode instruction when an L-directive is read, provided the contents of *B*0.1 and 0.2 are undisturbed. Similarly the instruction

→ IO, 1200

calls in the Initial Orders after setting the Transfer Address to decimal address 1200 (i.e. to *B*150.0), and the instruction

→ IO, 1200, 153

will in addition set the relativizer to 153. If we wish we can also use instructions such as the following.

→ IO, *n*1

→ IO, *n*1, *n*2

▲ These various Autocode instructions are summarized in Table 9.6

STOP	Stop (77), proceed with next instruction when Run key is operated.
TAPE	Read in more instructions and add them at the end of the programme.
TAPE *n*1	Read in more instructions, the first to replace that labelled *n*1.
TAPE *n*1, *n*2	Read in more instructions, the first to replace the instruction *n*2 after that labelled *n*1.
→ M *n*1	Exit to machine orders at decimal address *n*1. } Set link in *X*1
→ M *n*1, *v*2	The same, and leave *v*2 unpacked in *X*7 and *X*5. } Set link in *X*1
→ M *n*1, *n*2	The same, but leave *n*2 in 7_M } Set link in *X*1
→ L	Obey link set in *X*1 before entry from machine orders by cue.
→ IO	Call in Initial Orders as a subroutine.
→ IO, *n*1	The same, but set T.A. = *n*1,
→ IO, *n*1, *n*2	The same, and set relativizer = *n*2 in addition.

Table 9.6 *Autocode instructions - miscellaneous*

At the beginning of the Autocode programme tape the directive J 1.0 is punched; this causes the input programme to treat the following instructions as a new Autocode programme and to assign the label 0 to its first instruction. Sometimes we may wish to *restart* an Autocode programme that has already been read in, perhaps after changing one or two parameters or after making some alterations to the programme (this is described below). This can be done by reading in a bracketed interlude ending with a jump, for example

(*n*3 = -48
→ 1)

We must *not* punch J 1.0 at the beginning of this tape for the input programme would then treat it as the start of a new programme and store it over the beginning of the original programme. We must instead punch J 1.2 at the head of the tape; this has the same effect as the Autocode instruction TAPE so that any instructions on the tape will overwrite the last bracketed interlude.

Occasionally it may be found desirable to alter an instruction in an Autocode programme. This can be done by using the tape-editing equipment to alter the tape but this may be laborious if the programme tape is long. Instead of doing this we can insert a correction near the end of the tape, before the bracketed interlude causing entry to the programme. This correction is written, for example

ALTER 6, 3

followed by an Autocode instruction (on a new line); this instruction will then replace the one which is three after the instruction labelled 6. This sequence forms a kind of *Autocode directive* and is not stored; it does not form a part of the programme. When the alteration has been made the input section of the Autocode scheme will return to read more instructions or alterations. If the alteration is to be made by a separate tape then it should be headed J 1.2 as described above. In this connection it is worth noting that a bracketed interlude can be considered as a kind of Autocode directive.

▼ **9.4 The Matrix Interpretive Scheme**

Many of the calculations arising in industry can be expressed in matrix form.† It is therefore important that the preparation of these calculations for the computer should be as quick and easy as possible. The matrix Interpretive Scheme is a means of specifying in easy and concise terms the transformation between the matrix form of the problem on paper and the actual operation of the machine. The problem to be solved must first be expressed in matrix form. Then a method of solution has to be thought out and expressed as a sequence of elementary matrix operations, e.g. the input of a data-matrix into the computer, or the multiplication of one matrix by another. Each of these elementary steps is then written down as a single instruction, called a *matrix-instruction*, which defines the operation required and the positions in the computer's store of the matrices involved. The whole sequence of matrix-instructions is called a *matrix-programme*.

† Good introductions to Matrix Algebra will be found in the following books. AITKEN, A.C., *Determinants and Matrices* (Oliver and Boyd, Edinburgh, 19). FERRAR, W.L., *Algebra* (Oxford University Press). There is also a useful discussion of some of the applications in: FRAZER, R.A., DUNCAN, W.I. and COLLAR, A.R., *Elementary Matrices* (Cambridge University Press, 1938). LANCZOS, C., *Applied Analysis* (Pitman, London, 1957).

In use the programme of the Matrix Interpretive Scheme is first read into the store of the computer by the Initial Orders in the ordinary way.† The Interpretive Scheme programme then reads in the tape on which is punched the matrix-programme. When the whole tape has been read and stored the individual matrix-instructions making up the matrix-programme are examined and decoded in turn by the Interpretive Scheme programme, which then carries out the operations called for. The notation used for the matrix-instructions has been designed to be simple. The aim has been to make it possible for the person originating the problem to write the programme in this form, which is directly acceptable by the computer; the user need not be familar with ordinary Pegasus programming.

The user of the scheme can visualise the store of the computer as a continuous strip of 6142 locations, each of which can hold one element of a matrix.†† The elements of a matrix occupy a block of consecutive locations on this strip. It is an important practical point that a stored matrix consists of nothing more than its individual elements arranged in a certain order, and that each element can stand on its own as a number. Thus there are no row or column checksums or overall scale factors, and the dimensions of the matrix are not stored explicitly with it. This makes it possible to change or extract individual elements, or groups of elements, to regard a rectangular matrix as a number of columns, to deal relatively easily with partitioned matrices, and in general to make efficient use of the storage space available. When a particular matrix is no longer needed the space it occupies in the store may be used again at a later stage, giving great economy of storage space.

An important feature of the scheme is that the user can consider all numbers to be in their normal decimal form (with the decimal point wherever he wishes); when these numbers are taken into the computer they are converted automatically into packed floating-point numbers ($A.2^a$), which are used for all subsequent operations (the converse applies on output). This eliminates difficulties in scaling or overflow and enables the numerical data to be read into the machine in the form in which they occur. All the numbers are represented with a precision corresponding to the use of between eight and nine significant figures; this is adequate for most applications.

The instructions and the numerical data are punched on to paper tape. The elements of the matrices are punched as written, each preceded by its sign and followed (optionally) by a decimal exponent. Thus an element of value 123.456 may be punched in any of the following ways:-

+ 123.456

+ 0.123456 + 3

+ 1.23456 + 2

+ 123456 - 3 (the spaces are optional).

Each element may be labelled (if desired) by its row or column number, punched as a number without a sign.

Matrices are normally punched by columns (this giving fewer errors) and after each element the CR and LF characters are punched. This permits a data tape to be typed out automatically by teleprinter equipment for checking or record purposes. It is normal to punch a checksum at the end of each column; this is used by the computer to check the input when the tape is read in. It should be noted that this checksum is not stored, it is merely compared with the sum of the elements read in, and input carries on as usual if the check agrees. Letters, numbers or other characters may be punched at the head of the tape as a name or label. When the tape is read in these will be punched on the output tape, which thus bears a record of the matrices which have been put in.

Results are normally presented in columns, in a form similar to that used for input. Each column is headed by its column number, each element is preceded by its row number, and there is a checksum for each column. By using scissors and paste the result can be displayed as a complete matrix in its conventional form. There are two alternative forms of output:

(a) each element has one decimal figure before the point and is followed by a decimal exponent;

or (b) the whole matrix is automatically scaled (by a power of 10) so as to print the largest element with a pre-selected number of figures before its decimal point, and the decimal exponent of the scaling factor is printed at the head of the matrix.

The punched tape which appears as the output may be printed out automatically; it is in such a form that it may be read in again if it is being used to carry intermediate results.

In the interests of simplicity no checks on the accuracy of computation have been incorporated. Experience with the computer has shown that there is no real need for these, and in any case they would require a considerable amount of rather difficult programming. However, the computer itself incorporates a number of automatic checks against errors, particularly concerning the input, output and the store. There is a programmed check against errors in the input equipment. The column checksums will help detect wrongly-punched data. During the input of any matrix the elements are counted and checked against the dimensions of the matrix.

The matrix interpretive scheme enables a complete matrix operation to be performed by writing one single matrix-instruction. To enable a complete matrix-programme to be built up it is necessary to have certain other functions. These have been provided for in the scheme and a complete matrix-programme, comprising many separate matrix-instructions, can be built up quickly and easily. The list of functions is comprehensive for most matrix work. A single matrix can be copied, transposed or normalised. Two matrices can be added, subtracted, multiplied or divided. Matrix-instructions are also used for the input and output of matrices. In addition to rectangular matrices, the scheme is designed to operate on diagonal matrices, row-vectors, column-vectors, and scalar matrices. The row- and column-vectors are regarded simply as special cases of rectangular matrices.

---

† The scheme bears the Library routine-number R 7500 (or R 2500 for the 4096-word store), which is used only for reference purposes.

†† On the 4096-word store there is room for 3070 matrix elements.

Each matrix element stored in the computer occupies one storage location, of which 6,142 are available. A matrix is stored within this space *in columns;* a matrix of order $m \times n$ occupies mn storage locations. For a diagonal matrix, only the diagonal line is stored, so that a diagonal matrix of order $n \times n$ occupies n storage locations. A scalar or a scalar matrix (i.e. a scalar multiple of the unit matrix) is stored as a single element. The scheme has been written so that in most operations the results may be written over one or other of the operands; exact details of what may be done are given later.

The following is a typical matrix-instruction:

$$(304,\ 10 \times 12) \times (1195,\ 12 \times 16) \rightarrow 5$$

When obeyed this will cause the 10×12 matrix starting at location† 304 to be post-multiplied by the 12×16 matrix starting at 1195, the product will be a matrix of 10 rows and 16 columns, and this is placed in consecutive locations starting at 5. The whole operation is done in floating-point form. Each expression in brackets on the left can be thought of as representing a matrix. A matrix may be rectangular, for example (2107, 29×25) stands for a matrix with 29 rows and 25 columns whose first element is in location 2107; this matrix will occupy 725 locations of which the last is 2831. A diagonal matrix with 19 rows and columns starting at location 100 is written (100, 19/); since only the diagonal elements are stored this matrix occupies only 19 locations. No order need be specified for a scalar quantity or a scalar matrix; for example (300) can mean a scalar stored in location 300 or a scalar multiple of a unit matrix of any order.

In a general instruction three matrices may occur; the operands will be referred to as **A** and **B** and the result as **C**. The addresses of these matrices will be denoted by N_a, N_b, N_c, respectively (these are the addresses of the locations holding the first elements of the matrices). The letters m and n denote the number of rows and columns respectively. In general the following three kinds of matrix are used.

(a) A rectangular matrix is written (N_a, $m \times n$),

(b) A diagonal matrix is written (N_a, m/),

(c) A scalar matrix of any order is written (N_a).

Row- and column-vectors are regarded as special cases of rectangular matrices, for example (1230, 12×1) means a column-vector with 12 elements. In any matrix, N, the address, must lie between 1 and 6142, and the dimensions m and n must not exceed 255. The address N is zero in certain orders used for input and output.

The available types of matrix-instruction are shown in Table 9.7; for each one a general symbolic form is given, and also a typical actual matrix-instruction. Apart from a few obvious exceptions, any matrix may be rectangular, diagonal or scalar. In the symbolic form, **A**, **B**, **C** represent the three matrices concerned; **A** and **B** are written in the way described above and **C** is specified merely by its address. There are two instructions which are not included in this table (written with letters O and J); these are described later.

Function	General form	Typical matrix-instruction
Copy	**A** → N_c	(314, 15×9) → 2653
Input	0 → N_c	(0, 58×9) → 932
Output	**A** (x, y) → 0	(384, 6×2)(6, 4) → 0
	(x and y specify the number of digits - see below)	
Transpose	**A** * → N_c	(33, 8×32) * → 795
Transpose in situ	**A**/ → N_c	(348, 25×25)/ → 348
	(**A** must be square, **C** must have the same address as **A**)	
Add	**A** + **B** → N_c	(28, 8/) + (41, 8×8) → 971
Subtract	**A** - **B** → N_c	(6, 9×3) - (99, 9×3) → 751
Multiply	**A** × **B** → N_c	(375, 10×5) × (58, 5/) → 209
Divide	**A**, **B** → N_c	(749, 4×4), (592, 4×2) → 30
	(in this order $\mathbf{A}^{-1}\mathbf{B}$ goes to N_c, **A** and **B** are both in general spoiled)	
Normalize	**A** n **B** → N_c	(78, 16×40) n (62) → 862
	(a scaled form of the matrix **A** goes to N_c the largest element being made 1, the scaling factor, which must be a scalar, goes to N_b)	
Convert to fixed-point	**A** v **B** → N_c	(8, 9×9) v (862) → 901
	(the scale factor goes to N_b; this is the exponent of the maximum element)	
Binary output	**A** v → 1 (or 9)	(34, 21×17) v → 1
Binary input	0 v → N_c	(0, 6×7) v → 982
	(these last two functions are complementary; they are described below)	
Stop (77)	*	*

Table 9.7 *Matrix-instructions*

† The 6142 available locations (or 3070 on the 4096-word store) are numbered straight through (starting at 1), these numbers are not the ordinary decimal addresses of the locations but differ from them only by addition of a constant (actually 1023).

As an example, we give below a complete programme for the calculation of the column-vector **y** given by

$$\mathbf{y} = \{\omega\mathbf{D} + \Omega(\mathbf{E} + f\mathbf{I})\,\mathbf{B}\}\mathbf{x},$$

where ω, Ω and f are scalars, **D** is a diagonal matrix, **E** and **B** are square matrices, **I** is the unit matrix, **x** is a column-vector, and all matrices and vectors are of order 10. For checking purposes, the last row and column of the intermediate square matrix

$$\mathbf{C} = \Omega(\mathbf{E} + f\mathbf{I})\,\mathbf{B}$$

are to be printed. The data for the calculation are

$$\mathbf{E},\ \mathbf{B},\ f,\ \Omega,\ \omega,\ \text{and}\ \mathbf{x}$$

and are assumed to be available on an input tape in this order. The complete programme tape reads as follows:-

D	Date and serial number
N	name of programme
SPECIMEN MATRIX-PROGRAMME	
J64.0	Call in matrix scheme
(0, 10 × 10) → 1	Read in **E**
(0, 10 × 10) → 101	Read in **B**
(0) → 201	Read in f
(0) → 202	Read in Ω
(1, 10 × 10) + (201) → 1	Replace **E** by (**E** + f**I**), and
(202) × (1, 10 × 10) → 1	then by Ω (**E** + f**I**)
(1, 10 × 10) × (101, 10 × 10) → 203	Form **C**
(203, 10 × 10) * → 101	Replace **B** by **C**′ (transpose of **C**)
(191, 10 × 1) (6) → 0	Print out last column of **C**′
(293, 10 × 1) (6) → 0	Print out last column of **C**
(0, 10/) → 101	Replace **C**′ by **D** (diagonal)
(0) → 202	Replace Ω by ω
(101, 10/) × (202) → 101	Replace **D** by ω**D**
(101, 10/) + (203, 10 × 10) → 203	Replace **C** by ω**D** + **C**
(0, 10 × 1) → 1	Read in **x**
(203, 10 × 10) × (1, 10 × 1) → 11	Form **y** = (ω**D** + **C**) **x**
(11, 10 × 1) (6) → 0	Print out **y**
*	Stop
S	Directive to start obeying matrix-programme

To run this programme the tape of the Matrix Interpretive Scheme is first run into the computer; it ends with a Z-directive to cause a stop. The above tape is then put into the main tape-reader and the Run key is operated. The D and N-directives have their usual effects and the special directive J 64.0 is then read; this simply causes the Matrix Interpretive Scheme to read in the following programme and store it. The S at the end of the tape is not a matrix-instruction but a directive (S for *start*) which causes the matrix-programme to be obeyed. The first order is, as usual, an input instruction; this kind of instruction always causes an optional stop before any tape is read (to allow changing of tapes, if required). At this point the data tape is placed in the main tape-reader and the Run key is operated. The actual running time for the above programme, including input of data and all output, is about 75 seconds.

The instructions making up a matrix-programme are, of course, punched exactly as they are written, each being terminated by CR and LF. It is usual to include a number of spaces in each instruction to improve the legibility of the print-out, these spaces (as well as any Erase characters) are ignored when the tape is read. There is room in the store for 80 matrix-instructions, these being numbered 0 to 79. Should the matrix-programme be longer than this, one can either use some of the space normally used for numbers or else read in more instructions after the first 80 have been obeyed. To assist in reading in the matrix-programme three *directives* have been provided in the *order-read* section of the Matrix Interpretive Scheme; these are as follows.

I This causes the Initial Orders to be called in (as a subroutine). When the Initial Orders read an L-directive the interpretive scheme will resume reading in matrix-instructions at the point it had reached when the I was read.

S This causes the interpretive scheme to start obeying the matrix-programme. If desired we can write a number after the S; for example, S 12 causes the scheme to start at the matrix-instruction no.12 (note that the instructions are numbered from 0).

X*n* This directive sets an address (the equivalent of the Transfer Address) in the scheme so that the next matrix-instruction on the tape will replace matrix-instruction number *n*. If *n* is zero it may be omitted.

The *numerical data* read in by input matrix-instructions can be punched as normally written. Each number can be punched with or without a decimal exponent. A number is introduced by its sign (punched as + or -); spaces and erases are ignored throughout the number. The decimal point is punched where required; no decimal point need be punched if the number is an integer. The number of decimal digits which can be accommodated in one number is 9. If more than this are punched they will be ignored, but care should be taken not to punch more than 11 digits if the number is being punched as an integer. If the number is fractional then the extra digits will in any case be insignificant. The exponent (if there is one) is punched after the number proper and is introduced by its sign; it must not exceed 77. The end of the number is indicated by CR LF. If desired, unsigned integers representing row or column numbers may be punched in front of any number, they will be ignored on input. The data-input section of the matrix scheme recognises the following directives.

n This is punched to introduce a decimal exponent (a block exponent) which is to be carried through the matrix. It is followed immediately by a sign and then the exponent. This will be added to the exponent of each number until an asterisk or another *n* is read. If no exponent is put in this number will of course be zero. Space and erase are ignored. The exponent is terminated by CR LF.

→ It is possible to print out a name or title for each matrix. This is done by punching an → on the data tape. Each character read after this will be printed on the output tape until two consecutive figure shifts are encountered.

= This is punched immediately before a checksum. Thereafter the checksum is punched in the same way as any other number. A certain tolerance is allowed between the checksum computed from the elements, and the checksum on the tape; this tolerance depends on the number of digits actually punched in the tape checksum. If the checksum is not exact, i.e. if the elements of which it is the sum have been rounded off after computing the checksum, it should be punched to as many digits as are punched in the element which is punched to the fewest number of digits (disregarding zero elements). If however the checksum is intended to be exact, as is more common in practice, it should be punched with two more digits than are given in the separate elements, subject to one restriction. The restriction is that the sum of the moduli of the checksum and the largest element should not contain more than 9 digits. This principle can best be illustrated by examples:-

(a)
+.2031
+.101
+.123
Sum +.4271
Punch = +.427100

(b)
+10210
+320
Sum +10530
Punch = +10530.00

(c)
+1023
+.3641259
-1000
Sum +23.3641259
Punch = +23.36413

(d)
-.12345
+.12345
Sum +.00000
Punch = +.0000000

(If +0 is punched, even such a gross error as the wrong sign for either element will not be detected.)

* This must be punched after each matrix. It causes the next matrix-instruction to be obeyed, after first checking that the correct number of elements has been taken in.

There are two programmes within the scheme for the *output of results;* they are independent of one another, and the scheme has been written so that only one may be in the machine at any one time. One programme prints numbers in floating-point form, that is, with one decimal digit preceding the decimal point and the whole argument followed by an exponent to base 10. The other programme prints numbers in a fixed-point form with a specified number of digits before the decimal point and a block exponent for the whole matrix (which is printed first). There are thus two matrix-schemes, identical except for form of output, and they are known as "floating-point matrix scheme" and "fixed-point matrix scheme".

A tape produced by the floating-point output programme may later be read in by the input programme (provided that not more than 9 significant figures have been punched). A tape produced by the fixed-point output programme can also normally be read in again, but if this is to be done not more than 8 figures should be punched in each number (if 9 are punched there may be occasional checksum failures on input).

A name or title is printed to distinguish each matrix which is punched out. It consists of → and a decimal number specifying which matrix instruction caused output to occur. For example, →16 means that the 16^{th} instruction is being obeyed. The elements of the matrix are printed in columns. Each column is preceded by two line feeds and the column number, which appears centrally over the column. Each element is preceded by a row number and two spaces. These may be suppressed at any time during punching by pressing down the sign-digit key of the handswitches.

If the floating-point output is used each number is printed as an argument and a decimal exponent. The number of significant digits in the printed argument is x, the number written in the output instruction (if none is written x is automatically set to 9). A typical element with $x = 7$ is

12 +1.234567 +16

meaning 1.234567×10^{16}; the 12 is the optionally printed row number. At the foot of each column a checksum is printed, preceded by an extra LF and an equals sign. The number of figures printed in this checksum is automatically adjusted to suit the tests applied on re-input.

If we use fixed-point output each number is printed with at most x significant figures, of which at most y appear before the decimal point (there will always be x-y digits after the point); here x and y are written in the output instruction. (If x and y are not written in this instruction they will be set to 9 and 1 respectively). Two elements might appear as follows, if $x = 8$ and $y = 4$

15 -8765.4321

16 +45.6789

provided the row numbers are not suppressed. A checksum is printed at the foot of each column. Fig. 9.1 shows specimens of the two kinds of output. The fixed-point output includes on overall scale-factor or block exponent (printed at the head of the matrix); the programme can easily be amended to give a normal fixed-point output with no scale factor.

```
→
21

      0

0   +1.3742   +3
1   +2.1840   +2
2   -7.9336   +2
3   -6.0011   +0
4   +3.6100   -10

=   +7.932    +2

      1

0   +2.6098   -1
1   -1.1122   +0
2   +1.1132   +0
3   +5.3142   -2
4   +4.1592   +1

=   +4.1907   +1

      2

0   -8.9793   +3
1   +0
2   -2.3846   -1
3   -2.6433   +1
4   +8.3279   +2

=   -8.1732   +3
*
```

```
→
21
n+2

      0

0   +13.742
1    +2.184
2    -7.934
3    -0.060
4    +0.000

=   +7.932

      1

0    +0.002
1    -0.011
2    +0.011
3   +0.001
4    +0.416

=    +0.419

      2

0   -89.793
1    +0.000
2    -0.002
3    -0.264
4    +8.328

=   -81.732
*
```

Fig. 9.1 *A typical matrix with 5 rows and 3 columns as printed by the Matrix Interpretive Scheme, using the two alternative forms of output.*

The only instructions not so far described are those for exit to machine orders (written as a letter O followed by the decimal address of an *a*-order) and for jumping; there is only an unconditional jump, written as a J followed by the number of the matrix-instruction to which the jump is to go. The O-instruction causes the block containing the specified *a*-order to be transferred to *U*0 and entered; 5_M will be the address of this order and may be used to read in more blocks from the main store. To return to the next matrix-instruction we must obey the cue

61 [0] 72

0.0 0 60

To return to matrix-instruction number *n* we put the integer *n* into *X*7 and obey the cue

65 [1] 72

1.2+ 0 60

The O-instruction is useful if a matrix-programme is very long and there is room only for a part of it in the store; at the end of each part we may have the instruction O 512 to read more matrix-programme over the top of the old, or O 7168 to enter the Initial Orders.† The Matrix Scheme occupies blocks up to *B*107 inclusive, and the matrix-programme starts at *B*108.0 (with two words per matrix-instruction); care must be taken not to overwrite these by machine orders. If there are only 60 matrix-instructions in the programme then *B*123 to *B*127 inclusive are available for machine orders, or we can use part of the space normally holding numbers. Each matrix element occupies one word and location 1 corresponds to *B*128.0.

There is an extension to the scheme to permit the use of *preset-parameters*. For a problem of a given type this allows one matrix-programme to be written which can be used with matrices of different sizes. When this facility is to be used the programme is written in general terms, and a small subsidiary programme (called the *Interlude*) is used to insert into the matrix-instructions the dimensions of all the matrices for the particular calculation, and also to allocate the storage locations. To use this facility we write the matrix-programme in the ordinary way except that any dimensions (i.e. numbers of rows and columns) which cannot be fixed are written in as numbers between 241 and 255, and any addresses which cannot be fixed are specified as numbers greater than 5000. In fact if *m* denotes any dimension and *N* any address of a matrix *as written* in an instruction, then

(a) if $1 \leqslant m \leqslant 240$ *m* is the actual dimension, but
if $241 \leqslant m \leqslant 255$ the dimension to be used is specified separately by the programmer,

and (b) if $1 \leqslant N \leqslant 5000$ *N* is the actual address, but
if $N \geqslant 5001$ the address to be used is specified separately by the programmer.

While writing the matrix-programme we build up two lists, an *address-list* containing values of *N* and a *dimension-list* containing values of *m*. In the actual instructions we write 5001, 5002, etc. for unspecified values of *N*, and 241, 242, etc. for *m*. For a particular case we can punch actual address and dimension lists to suit the matrices being used and we feed these into the store together with the matrix-programme. The Interlude then processes the matrix-programme, replacing any *N* exceeding 5000 and any dimension exceeding 240 by the appropriate address or dimension from the lists. After this the resulting matrix-programme is obeyed in the usual way. The advantage of the preset-parameter scheme is that the actual matrix-programme will apply to all problems of a given type, and yet we retain the possibility of overwriting matrices when they are no longer needed.

As an example, we give a short matrix-programme to read in two matrices

Q, of order $m \times n$ $(m \leqslant n)$,
and **A**, of order $m \times m$,

and to evaluate and print the $n \times n$ matrix $\mathbf{Q}'\mathbf{A}\mathbf{Q}$ This programme is written as follows.

(0, 241 × 242) → 1	Read in **Q** ($m \times n$)
(0, 241 × 241) → 5001	Read in **A** ($m \times m$)
(5001, 241 × 241) × (1, 241 × 242) → 5002	Form **AQ** ($m \times n$)
(1, 241 × 242) * → 5001	Form $\mathbf{Q}'$ ($n \times m$)
(5001, 242 × 241) → 1	Copy $\mathbf{Q}'$ over **Q**
(1, 242 × 241) × (5002, 241 × 242) → 5001	Form $\mathbf{Q}'\mathbf{A}\mathbf{Q}$ ($n \times n$)
(5001, 242 × 242) → 0	Print result
*	Stop

† The instruction O 4096 is used to enter the Initial Orders in the 4096-word store version.

We prepare the dimension and address lists in the following form while this programme is being written.

Dimension list	*Address list*
241 m	5001 $2mn + 1$
242 n	5002 $mn + 1$

The allocation of the store is as follows:

1 to mn	Matrix **Q**, later $\mathbf{Q}'$
$mn + 1$ to $2mn$	Product **AQ**
$2mn + 1$ to $2mn + m^2$	Matrix **A**
$2mn + 1$ to $3mn$	Matrix $\mathbf{Q}'$ (overwrites **A**)
$2mn + 1$ to $2mn + n^2$	Matrix $\mathbf{Q}'\mathbf{AQ}$ (overwrites **A** and $\mathbf{Q}'$)

The programme can be used for any matrices **A** and **Q** provided

$$2mn + n^2 \leqslant 6142.$$

In use actual numbers are put into the dimension and address lists, and these are then read into decimal address 1600 and 2000 respectively (i.e. into *B*200.0 and *B*250.0); this preset-parameter tape is made up as follows (for the above matrix-programme with $m = 30$, $n = 31$, $2mn + 1 = 1861$, $mn + 1 = 931$):-

```
T 1600
+ 30      } dimension-list
+ 31      }
T 2000
+ 1861    } address-list
+ 931     }
J 40.0    entry to Interlude.
```

This tape is to be read in *after* the matrix-programme has been read in as usual. At the end of the matrix-programme tape we should punch the Matrix Scheme directive I to call in the Initial Orders (there can then be an Initial Orders Z-directive to cause a stop so that we can put the parameter tape in the tape-reader). When the Interlude has been entered by the J 40.0 at the end of the parameter-tape it first finds the address of the last matrix-instruction read in and then processes all the matrix-instructions, replacing any addresses greater than 5000 or dimensions greater than 240 by numbers taken from the lists. When it has done this the computer will start to obey matrix-instructions starting at number 0.

If a programme like the above example is used a great deal we can arrange to feed in only the dimension-list and get the address-list automatically computed by a small programme of machine orders.

If storage space for matrices is at a premium it may be desirable to punch out certain intermediate results for later re-input. This is best done by using the *binary input and output* instructions. To punch out in binary the 10 × 6 matrix in location 100 we can use the instruction

$$(100,\ 10 \times 6)v \rightarrow 1$$

which precedes the main punching by a leader of blank tape; a checksum is punched at the end of the matrix. To read this matrix in again to location 947 we use the instruction

$$(0,\ 10 \times 6)v \rightarrow 947$$

If we use the following instruction to punch out the matrix

$$(100,\ 10 \times 6)v \rightarrow 9$$

a checksum is punched at the end of each column and the matrix can be read in either one column at a time, or as a complete matrix. If desired we can punch out the successive columns of a matrix (e.g. as they are computed) by several $0v \rightarrow 1$ instructions and later read it in as a complete matrix.

In most matrix-instructions the address of the result (**C**) can be the same as that of either of the operands (**A** and **B**), *provided the result does not occupy more storage space* than the operand which gets overwritten. The following are the exceptions to this rule.

(a) *Transposition*

$\mathbf{A}^* \rightarrow N_c$	**C** may never overwrite **A**,
$\mathbf{A} / \rightarrow N_c$	**C** must always have the same address as **A**.

(b) *Multiplication*

$$\mathbf{A} \times \mathbf{B} \rightarrow N_c$$

If either operand is rectangular (but not a vector) then the other one may not be overwritten.

(c) *Division*

$$\mathbf{A},\ \mathbf{B} \rightarrow N_c \text{(i.e. } \mathbf{C} = \mathbf{A}^{-1}\mathbf{B})$$

If **A** is square it may not be overwritten; if in addition **B** is rectangular (but not a vector) it may not be overwritten either.

In the division instruction **A** is spoiled if it is square, and is replaced by $\mathbf{A}^{-1}$ otherwise (unless overwritten). If **A** is square, **B** is replaced by $\mathbf{C}'$ (if **B** is rectangular) or by **C** (if it is a column-vector); in all other cases **B** is untouched provided, of course, it is not overwritten by the result. Note that partial overwriting of an operand is *forbidden* in general for any instruction.

There are a few useful tricks which can be done with the *transpose* instruction **A** * → **C**; in fact an order of this kind is fundamental to any scheme for manipulating matrices. It is easy to pick out the individual columns of a matrix; by transposing first we can pick out the rows. This idea can be generalised to *partitioned matrices*. Suppose a 10 × 8 matrix has address 1 and is partitioned thus

$$\left(\begin{array}{c|c} \mathbf{A} & \mathbf{B} \\ \hline \mathbf{C} & \mathbf{D} \end{array}\right),$$

where **A** is 4 × 5, **B** is 4 × 3, **C** is 6 × 5 and **D** is 6 × 3. This matrix occupies locations 1 to 80. Suppose we have to put the component matrices **A**, **B**, **C**, **D** into addresses 81, 101, 113 and 143 respectively. The following sequence of matrix-instructions can be used:

(1, 10 × 5)* → 123	Matrix $(\mathbf{A}' \mid \mathbf{C}')$ to 123-172
(123, 5 × 4)* → 81	**A** to 81-100
(143, 5 × 6)* → 113	**C** to 113-142
(51, 10 × 3)* → 149	Matrix $(\mathbf{B}' \mid \mathbf{D}')$ to 149-178
(149, 3 × 4)* → 101	**B** to 101-112
(161, 3 × 6)* → 143	**D** to 143-160

A similar technique can be used to assemble a partitioned matrix. The transpose instruction can also be used to pick out the *diagonal elements* of a matrix. Suppose a 4 × 4 matrix **A** is at address 1. The following instruction puts into locations 21 to 24 the diagonal elements of **A** (and incidentally overwrites locations 25 to 40):

$$(1,\ 5 \times 4)^* \rightarrow 21$$

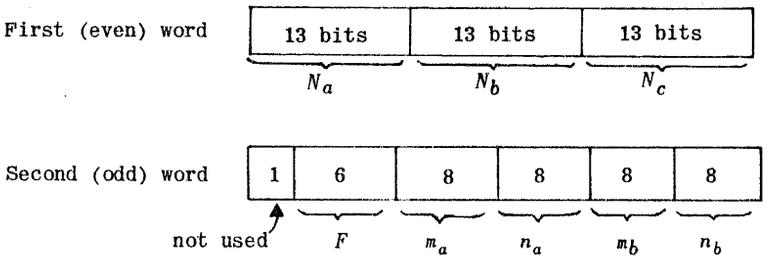
The diagonal matrix (21, 4/) is now made up of the diagonal elements of **A**.

The *speed of operation* can be found from Table 9.8. The approximate time for obeying an order can be determined from the formulae given. The last three columns give some typical times for square matrices and vectors of various orders. The bulk of the calculation time in most problems is likely to be occupied by multiplications of one rectangular matrix by another or by non-trivial divisions (i.e. those in which **A** is square); output time may also be significant. All the times given are approximate because the time taken by a floating-point operation depends on the actual numbers involved and because the access time to the main store cannot be accurately forecast. Usually the formulae give times within about 10% of the actual times. In this table the following abbreviations are used:

Rect.	Rectangular matrix,
Diag.	Diagonal matrix,
Col.vector	Column-vector,
Scalar	Scalar or scalar matrix.

The elements making up a matrix are each packed floating-point numbers similar to those described in Sec. 9.1 (with 30 bits for the argument and 9 bits for the exponent + 256). When used in the Matrix Scheme, however, these numbers are only partially unpacked when operated on, in fact the whole of the 39-bit word is used as the argument. The 9 bits used for the exponent are included and are used to round the argument; the resulting bias is normally quite negligible (because the exponent is stored augumented by 256) unless the elements are known exactly (e.g. if they are integers). Underflow (i.e. an exponent becoming less than -256) causes the number to be replaced by zero; there is no check on floating-point overflow, but this, of course, is unlikely to occur.

Each matrix-instruction occupies two words, the first of which must have an even address; the digits in these words are used as follows:



Here the figures show the number of binary digits used for each part of the instruction; these parts are as follows:

N_a, N_b, N_c	the three addresses in the instruction,
F	six bits indicating the function,
m_a, n_a	the number of rows and columns in the first matrix,
m_b, n_b	the number of rows and columns in the second matrix.

Operation		Formula giving time in milliseconds	Examples of times in seconds		
A matrix	B matrix		16	24	32
COPY					
Vector or Diag.		$5m + 150$	0.2	0.3	0.3
Rect.		$5mn + 150$	1.4	2.9	5.3
TRANSPOSE					
Ordinary		$12mn + 130$	3.2	7.0	12.4
'In situ'		$13n^2 + 150$	3.5	7.6	13.4
ADD or SUBTRACT					
Rect.	Rect.	$16mn + 190$	4.3	9.4	16.6
Diag. Vector	Diag. Vector	$16m + 190$	0.5	0.6	0.7
Square	Diag. Scalar	$5m^2 + 21m + 200$	1.7	3.6	6.0
Diag. Scalar	Square	$5m^2 + 21m + 200$	1.7	3.6	6.0
Diag. Scalar	Scalar Diag.	$15m + 190$	0.4	0.6	0.7
MULTIPLY (all these times will be reduced if there are zeros in A or B)					
Rect. ($m \times n$)	Rect. ($n \times r$)	$[(21n + 12)r + 12]\ m + 150$	1m 29.4	4m 57.7	11m 41.0
Rect.	Col. vector	$21mn + 24m + 150$	5.9	12.8	22.4
Row vector	Rect.	$21mn + 12n + 170$	5.7	12.5	22.0
Row vector	Col. vector	$21n + 180$	0.5	0.7	0.9
Col. vector	Row vector	$33mn + 12m + 150$	8.8	19.4	34.3
Rect.	Diag.	$10mn + 11n + 170$	2.9	6.2	10.8
Row vector	Diag.	$21n + 170$	0.5	0.7	0.9
Rect.	Scalar	$10mn + n + 180$	2.8	6.0	10.5
Row vector	Scalar	$11n + 180$	0.3	0.4	0.5
Diag. Scalar	Rect.	$12mn + 11n + 170$	3.4	7.4	12.8
Diag. Scalar	Col. vector	$12m + 180$	0.4	0.5	0.6
Diag.	Diag.	$12m + 180$	0.4	0.5	0.6
Scalar Diag.	Diag. Scalar	$11m + 180$	0.4	0.5	0.6
DIVIDE					
Square ($n \times n$)	Rect. ($n \times r$)	$19n^2r + 8.5n^3 + 30nr + 131n^2 + 181n + 170$	2m 37.0	7m 57.0	17m 52.0
Square	Col. vector	$8.5n^3 + 150n^2 + 211n + 170$	1m 17.0	3m 29.0	7m 19.0
Diag.	Rect.	$12nr + 11n + 11r + 170$	3.6	7.7	13.2
Diag.	Diag.	$27n + 180$	0.6	0.8	1.0
Scalar	Rect.	$12nr + 11r + 180$	3.4	7.4	12.8
NORMALIZE		$15mn + 190$	4.0	8.8	15.6
CONVERT		$12mn + 170$	3.3	7.1	12.5
INPUT (the specimen times assume 6 digits, sign and point in each number, and that there are no checksums or row- or column-numbers).					
Rect. Vector		5 msec. per character read, including layout characters, signs etc. plus 70-90 msec. per number	33.3 2.1	1m 14.9 3.1	2m 13.1 4.2
OUTPUT (the specimen times assume 6 digits etc. as for the input times).					
Rect. Vector	Fixed-point	30 msec. per character punched including layout characters etc. plus about 60 msec. per number, including checksums	1m 43.0 6.7	3m 43.5 9.6	6m 30.1 12.5
Rect. Vector	Floating-point		2m 14.6 8.7	4m 55.4 12.5	8m 36.7 16.3
BINARY INPUT		$40mn$	10.2	23.0	41.0
BINARY OUTPUT		$250mn$	1m 4.0	2m 24.0	4m 16.0

Table 9.8 *Times of operation of matrix-instructions.*

The function F indicates the kind of operation and the nature of the operands; the following are some examples:

Function F	*Operation*
0	Transpose,
2	Multiply rectangular matrix by rectangular matrix,
6	Multiply diagonal matrix by rectangular matrix,
7	Multiply scalar matrix by rectangular matrix,
10	Add rectangular matrix and diagonal matrix,
12	Add diagonal matrix and rectangular matrix,
22	Subtract scalar matrix from diagonal matrix,
28	Stop,
38	Divide scalar matrix by square matrix.

This function is evaluated during input of the matrix-instruction; it is later used to cause entry to the appropriate subroutine for carrying out the actual matrix operation.

If the storage space provided by the Matrix Interpretive Scheme is not enough to hold all the matrices required we may be able to punch out (in binary) some intermediate results which can later be read in again. This amounts to using the punched paper tape as an auxiliary backing store. If the computer is equipped with magnetic tape it is better not to use the ordinary matrix scheme but to use instead the *Magnetic Tape Matrix Scheme* (R 7502). This scheme is basically similar to the original scheme but includes a few extra orders for transferring information between the main store and the magnetic tape store. The scheme uses two magnetic tape mechanisms (see Chapter 10), on each of which about 200,000 locations are available for storing matrix elements (more can be made available by changing tapes).

Occasionally the precision of the numbers represented in the matrix scheme may be inadequate. If the calculation is not too large it may be possible to use the *Double Length Matrix Scheme* (R 7503), in which the precision is equivalent to about 22 decimal digits. All calculations are done in a double-length floating-point mode in which each number is represented by three words. The conventions for writing programmes are very similar to those for the ordinary matrix scheme except that only about 2000 matrix elements may be stored. Operations take roughly three times as long as those in the ordinary scheme.

9.5 Double-length floating-point arithmetic

For certain calculations high precision is required. The *Double-Length Floating-Point Interpretive Scheme*, R 650, has been written for them.† The programme is written in the form of a subroutine, which can be called in when it is required to do double-length floating-point operations; the user has therefore to be familiar with Pegasus programming and with the use of the Assembly routine. The interpreted orders resemble ordinary Pegasus orders and can consequently be read in, together with them, by the Initial Orders. Entry to the interpretive scheme and exit from it are very quick and simple; it is thus easy to do special counting operations and the like with ordinary machine orders. This routine occupies only 32 blocks in the main store.

The numbers used with R 650 each occupy three words, the first two of these represent the double-length normalized (or zero) argument and the third holds the binary exponent as an integer. There is a *floating-point accumulator* which holds one number; this occupies *U*1.1, 1.3 and 1.5. All the other numbers are held in the main store and are referred to by decimal addresses, which are always multiples of three. The address of a number is actually that of its first word (the left half of its argument), counted from the start of the R 650 working space. This working space is a section of the main store set aside to hold double-length floating-point numbers; there is a preset-parameter to fix the start of the working space. The second and third words representing a number are stored in the same position as the first word, but in the two following blocks.

Interpreted orders (i.e. orders causing R 650 to carry out operations on double-length floating-point numbers) are always obeyed in *U*0. As usual they are obeyed sequentially until a jump occurs. If the last order in *U*0 (i.e. in 0.7+) does not cause a jump then the next order to be obeyed is always in 0.0 (whether it is to be interpreted or not). Each interpreted order is made up of four parts *N X F M* like an ordinary order; but often a minus sign is written in the *X* part and the single address in the order is then denoted by *S*. This is usually the address of a number and will therefore be a multiple of three; the number itself is the content of *S* and is denoted by *s*. In explaining the effects of the various orders the double-length floating-point accumulator will be denoted by *A* and its content by *a*. For example, the order

12 - 00

has $S = 12$ (a multiple of 3), $F = 00$ and $M = 0$; its effect is to add the double-length floating-point number from location 12 into the floating-point accumulator. The effect of a 00-order can in general be written

$$a' = a + s.$$

The order

150 - 44

† This programme was written by Mr. J.G.F.Francis while with the National Research Development Corporation.

writes the number in the floating-point accumulator into location 150. The effect of a 44-order may be written

$$s' = a.$$

The whole order-code of R 650 is summarized in Table 9.9, many of the details are explained later: the times given in the last column are expressed in milliseconds; they are approximate only. Extravagant effects may occur if any unassigned order is obeyed.

F	Operation	Time (msec)
00	$a' = a + s$	51
02	$a' = a + s$, and set indicator if $a' = a$ or s	51
04	$a' = a + s$ (same as 00)	51
06	$a' = a + s$, and set indicator if $a' = a$ or s (same as 02)	51
10	$a' = a - s$	51
12	$a' = a - s$, and set indicator if $a' = a$ or $-s$	51
14	$a' = s - a$	51
16	$a' = s - a$, and set indicator if $a' = s$ or $-a$	51
20	$x' = N$-th constant in list ($N \geqslant 1$)	32
22	$a' = s' =$ number read from input tape	
24	punch CR LF and value of a } S indicates the style	
26	punch Sp Sp and value of a }	
30	test indicator: jump *in* to N if not set. Clear indicator	24
32	test indicator: jump *out* to N if not set. Clear indicator	26
34	count: $x'_M = x_M + 3$, $x'_C = x_C - 1$; jump *in* to N if $x'_C \neq 0$	26
36	count: $x'_M = x_M + 3$, $x'_C = x_C - 1$; jump *out* to N if $x'_C \neq 0$	27
40	$a' = s$	38
42	$a' = -s$	38
44	$s' = a$	38
46	$s' = -a$	38
50	block-read and jump: Block $N(+)$ to $U0$, jump to $0.X$	34
52	$a' = S$	26
54	set indicator if exponent of $a <$ exponent of s	30
56	set indicator if $\lvert a \rvert \leqslant 2^{-S}$	24
60	$a' = a \times s$	48
62		
64	$a' =$ integral part of a	29
66		
70	$a' = a/s$	74
72		
74		
76	subroutine operation S	

Table 9.9 *Double-length floating-point orders.*

As an illustration, we give below a sequence of interpreted orders for reading in a number x and placing in the floating-point accumulator the quantity

$$x(x + 3)/(x - 3).$$

The sequence is as follows.

3	-	52	$a' = 3$
0	-	44	write 3 into location $S0$
3	-	22	x to A and $S3$
0	-	10	$x - 3$
6	-	44	$x - 3$ to $S6$
3	-	40	x to A
0	-	00	$x + 3$
3	-	60	$x(x + 3)$
6	-	70	$x(x + 3)/(x - 3)$ to A

It will be seen that all the functions included in Table 9.9 are even (i.e. they have an even second digit). If the second function digit in any order is made odd by the addition of 1 then the order is obeyed as usual but *the next order is not interpreted;* it is treated in the normal way as a machine order. This is the simplest way of leaving R 650 and entering machine orders.

The *indicator* referred to in Table 9.9 can be used to facilitate convergence tests in iterative processes; it is actually represented by the sign bit of $U5.7$, which is 1 if the indicator is set (the rest of 5.7 is used for other purposes). In the jump orders, jump *in* means that R 650 is to start interpreting orders at the specified address, and jump *out* means that the jump is to an ordinary machine order. All the jumps can be only to orders in $U0$; the block part of the jump address is not used. The two orders for counting are of interest, they advance the modifier by 3 because of the way numbers are addressed; there is no carry from x_C to x_M (as in an ordinary 67-order); no jump occurs if x_M overflows and a jump will always take place if $X0$ is specified. The 50-order jumps *in*, and the 51-order jumps *out*. Machine orders may use $U0$ and $U4$ and all the accumulators without disturbing the interpretive routine.

Entry to the scheme is normally by obeying its cue 02. Before doing this we must place in $X6$ the address of the first order to be interpreted (which must be in $U0$); for example to start interpreting orders at 0.1+ we obey the order

(0.1+) 6 40

and then obey cue 02 to R 650. If instead we use cue 01 then the specified order will be treated as an ordinary machine order. If we leave the scheme to obey machine orders which do not disturb the interpretive routine (in $U1$, 2, 3 and 5) we can at any time re-enter it by jumping to 1.7+, after placing in $X6$ the address of the next order to be interpreted. Thus to start interpreting at 0.3 we can use the following pair of orders.

(0.3) 6 40

1.7+ 0 60

If we wish to start interpreting at 0.0 we can simply jump to 1.7 without setting $X6$. On exit from interpreted orders $X1$ contains the most-significant word of the floating-point accumulator (this may be tested in the usual way by ordinary orders 60 to 63); and $X2$, 3, 4, 5 will be unchanged since entry, except for the result of counting (or using an interpreted 20-order).

As an example of some of the techniques which may be used we give on page 222 a complete programme using R 650 which reads from the tape fifty sets of numbers x, y, z, w and calculates and prints for each set the quantity

$$-(x + y)/(zw + 2).$$

The data tape is to be placed in the main tape-reader when the E-directive causes a 77-stop.

The *output orders* are 24 and 26; they print the number in the accumulator, 24 preceding it by CR LF, and 26 by Sp Sp. The number S specifies the style. If b decimal digits are to be printed before the point and c after it then $S = 32b + c$; for example $140 = 32 \times 4 + 12$, so that style 140 causes 4 digits to be printed before the point and 12 after it. If $b = 0$ (i.e. $S \leqslant 31$) or if the number is greater than about 10^{22}, then it is printed in standard floating decimal form; this has a signed argument with the point after the first significant digit, and then c digits, a space, and a signed decimal exponent. Not more than 21 or 22 significant digits are printed in the argument, and the exponent is printed as zero if the argument is zero. The number is always correctly rounded and left-hand zeros in the integral part are replaced by spaces. If the number is an exact integer then the decimal point and the fractional part are replaced by spaces. During output $B0$ is used to store the accumulators. (A number whose exponent exceeds 511 causes a 77-stop and then printing of * 0).

The *input order* has the function 22; this order reads a single number from the input tape, converts it to double-length floating-point form and places it in location S and the accumulator. The number may be punched as an integer or a fraction or a mixed number and can, if desired, be followed by a decimal exponent; both the argument and the exponent must be signed and the whole number is always terminated by CR LF. Numbers punched by either of the output orders (24 or 26) are always suitable for re-input provided each number is termined by CR LF. In front of the number CR LF, LF, Sp and ϕ are ignored and Er may, as usual, appear anywhere except between CR and LF. Spaces may be punched anywhere except between digits or between the exponent and the terminal CR LF; there must be at least one space between the argument and the sign of the exponent (if there is one). A single decimal point

may be punched anywhere in the argument and may precede or terminate it. Any number of digits may be punched in the argument but only 21 or 22 significant digits are used, the rest being ignored. The number zero is stored with an exponent -2^{19}. If a number is read whose decimal exponent is 128 or larger in absolute value there is a 77-stop, but the programme will continue if the Run key is operated; this has been devised to detect punching errors. During input the programme uses *B*0.

```
D
N
FIFTY NUMBERS
A1
```

R 0	1	-0	2	Call for cue 02 to R 650 (in 0+.1)
650	-	01	-	

			0+		
E.2.0 → 0.0	(0.2)	6	40		prepare to interpret at 0.2
	(50)	2	40		set counter
1	+0				+ cue 02 to R 650
2	(2)	-	52		2 to *S*0
	0	-	44		
3	1+	0	50		*B*1+ to *U*0, jump to 0.0
	0				

	T1+		1+		
0.0	3	-	22		*x* to *S*3
	6	-	22		*y* to *A* (from data tape)
1	3	-	00		*x* + *y*
	3	-	46		-(*x* + *y*) to *S*3
2	6	-	22		*x* to *S*6
	9	-	22		*w* to *A* (from data tape)
3	6	-	60		*zw* to *A*
	0	-	00		*zw* + 2
4	6	-	44		*zw* + 2 to *S*6
	3	-	40		-(*x* + *y*) to *A*
5	6	-	70		-(*x* + *y*)/(*zw* + 2)
	140	-	25		print result (style 140), then obey machine orders
6	1.7	2	67		count in 2_C, start interpreting at 0.0
	0.6+	0	60		

```
L
A2
A3
E2.0
```

Interpreted orders can be *modified*, but only by a modifier in *X*3, 4 or 5. When any order is modified all thirteen bits of the modifier are added to the first ten bits of the order, i.e. to *S*(or else to *N* and *X*), which are extended by 3 bits (except for the 56-order); this modification cannot cause overflow. Modification can be used in a loop with either of the *count* orders (34 and 36). The counters and modifiers are normally set by a 20-order, which is described later.

The start of the working space used to hold numbers is fixed by a preset parameter; if no parameter-list is supplied an optional list will set it to *B*125.7 (this is the largest address that may be set). To set the start of the working space to, say, *B*110.0 the programmer must include the following *parameter-list* in his programme.

	R 0	0	-0	1
	650	-	04	-
01	110	0	00	0.
	0			

A number of *subroutines*, mostly for matrix work, are available for R 650 and it is sometimes useful to use programmer's subroutines also. The use of all subroutines is greatly facilitated by the special 76-order. If this order is to be used the programmer must supply a parameter-list, followed immediately by an index of the subroutines (and cues) which are to be used. The preset-parameter in this list is the one fixing the start of the working space, and is written as above except that its "*b*-order" is now the number of different subroutine operations. After this parameter we put a number of words each specifying one subroutine operation; we write the routine-number in the "*a*-order" and the cue-number in the "*b*-order". Consider the following parameter-list and index, for example.

	R 0	0	-0	1	title	Parameter-list
	650	-	04	-		
	120	0	00	0.	working space *B*120.0	
	2				two subroutine operations	
1	680				R 680 cue 01	Index of subroutine operations
	1					
2	680				R 680 cue 02	
	2					

The interpreted 76-order carries out the operation defined by the *S*-th entry in the index; for example, if the above index is supplied then the order

2 - 76

causes R 680 to be called in by cue 02. When using subroutines in this way no link need be set; after the subroutine operation is complete the next order is interpreted (the 77-order causes the following order to be treated as an ordinary machine order when the subroutine operation is finished). The parameter-list and index are treated in a special way by R 650; they must not be overwritten if subroutines are to be used. They are actually processed by an interlude and do not contain what was written there by the programmer.

The following are brief specifications of the available subroutines. Most of them require the setting of certain modifiers and counters as programme-parameters; this can be done by using the 20-order, which is described below.

R 680 *Linear combination*

Cues: 01 $a\mathbf{v}$ replaces **u** (time 126 + 43n msec), Store: 4 blocks.

02 **u** + $a\mathbf{v}$ replaces **u** (time 126 + 58 n msec).

Before entry put 4_M = address of vector **u**, 5_M = address of vector **v**, 5_C = number of elements in each vector.

R 681 *Scalar product*

Cues: 01 $a' = \mathbf{u}.\mathbf{v}$ (time 117 + 59n msec), Store: 3 blocks.

02 $a' = a + \mathbf{u}.\mathbf{v}$ (time 131 + 59n msec).

Before entry set accumulators as for R 680.

R 684 *Square root*

Cue: 01 $a' = \sqrt{a}$ (time about 224 msec). Store: 3 blocks.

R 685 *Matrix multiplication*

Cues: 01 **B** × **C** replaces **D** } (time about 60 $pr(q + 3)$ msec) Store; 2 blocks + 3 more for R 681.

02 **B** × **C** is added to **D** }

Before entry set (d, p) in $X3$, (c, r) in $X4$ and (b, q) in $X5$. The matrices **B**, **C** and **D**, which are stored *by rows*, have their first elements at b, c, d and dimensions $p \times q$, $q \times r$ and $p \times r$ respectively. The matrix **D** must be different from **B** or **C**. The content of the floating-point accumulator is lost. This routine uses R 681 as a subroutine.

R 686 *Matrix division*

Cues: 01 $\mathbf{B}^{-1}\mathbf{C}$ replaces **C**, $\mathbf{B}^*$ replaces **B**.

02 $\mathbf{C}.\mathbf{B}^{-1}$ replaces **C**, $\mathbf{B}^*$ replaces **B**.

03 $\mathbf{B}^{-1}\mathbf{C}$ replaces **C**, where $\mathbf{B}^*$ is given.

04 $\mathbf{C}.\mathbf{B}^{-1}$ replaces **C**, where $\mathbf{B}^*$ is given.

Store: 5 blocks + 9 more for R 680, 681, 685.

Time: $60pq(p + 3)$ msec for cues 03 and 04; for cues 01 or 02 add $p^2(29p + 240)$ msec. to this.

Before entry set (c, q) in $X4$ and (b, p) in $X5$. The matrices **B** and **C**, which are stored *by rows*, have their first elements at b and c; **B** has dimensions $p \times p$ and **C** has dimensions $p \times q$ (cues 01 and 03) or $q \times p$ (cues 02 and 04). A special method of division is used in which **B** is replaced by $\mathbf{B}^*$, the basic information needed to reduce it to the unit matrix by straightforward elimination above and below the diagonal (no search is made for pivots); this process requires about $\frac{1}{2} p^3$ arithmetical operations. This information $\mathbf{B}^*$ is all that is required to divide another matrix by **B**, and operating with it resembles a matrix multiplication in needing only about p^2q arithmetical operations. Once $\mathbf{B}^*$ has been found by use of cue 01 or 02 it can be used again by cue 03 or 04 (for example to find $\mathbf{B}^{-2}\mathbf{C}$). The content of the floating-point accumulator is lost. This routine uses R 680 and R 685 (which uses R 681).

A subroutine for *Power Series Economisation* is also available for R 650; details of this may be found in the library specification, R 360.

The 76-order automatically restores the whole of the computing store, except for $X1$, 6, 7 and the floating-point accumulator A, after a subroutine operation is complete. It is quite possible to have subroutines within subroutines (i.e. sub-subroutines), and so on; each 76-order proceeding to a lower level causes R 650 to use extra blocks to store certain information temporarily. Blocks 894 and 893 are used with a simple subroutine; if this calls in its own subroutine $B892$ and 891 are also used, and so on.

Programmer's subroutines may also be used with R 650; they should be written in the usual way for Assembly. They may consist of machine orders only, or partly of interpreted orders (the latter must, as always, be in $U0$). The cues must be machine orders, for example

```
0+ [0] 72

1.7  0 60
```

will enter the subroutine at 0+.0 as an interpreted order (see above). Blocks $U1$, 2, 3 and 5 may not be disturbed if interpreted orders are to be used later, unless they are restored by a cue to R 650. Note that $U1.1$, 1.3 and 1.5 are the floating-point accumulator. During the process of entering a subroutine $X1$, 6 and 7 are used but the rest of the computing store is unaltered (except for the block overwritten by the action of the cue); however $X1$ will not contain the most-significant word in the accumulator. When the subroutine has finished its work it should obey cue 05 to R 650, which is called for by a tag in the usual way. When this cue is obeyed the whole of the computing store (except $X1$, 6 7 and the accumulator A) will be restored to its condition on entry to the subroutine. Ordinary subroutines may be used by machine orders within programmer's subroutines in the usual way; but if R 650-subroutines are to be called in by interpreted 76-orders a special technique must be adopted. If the sub-subroutine is included in the index of subroutine operations then the usual 76-order will call it in; it may, however, be undesirable to have to draw up the index at the time when the programmer's subroutine is being written. If this is so we must arrange that $U0.7$ contains a cue to the sub-subroutine at the moment it is to be entered; the special order 0 - 76 is then used to enter it. This technique ensures that correct action is taken about the *level* of the subroutine.

The interpreted 20-order can be used for setting modifiers and counters (or other constants) in $X3$, 4 or 5. If this order is to be used the programmer must supply a special *list of constants;* this always follows immediately after the working-space address parameter and the subroutine index (if there is one). For example, the order

```
2  5 20
```

puts the second constant in the list into $X5$ (the first constant is numbered 1). This order is a very useful one, as will be seen from the next example.

Suppose we are given three square matrices **A**, **B** and **C**, each with 6 rows and columns, and a 6-vector **u** and we have to evaluate the vector

$$\mathbf{A}^{-1}(\mathbf{A}^{-1}\mathbf{B} + \mathbf{C})^{-1}\mathbf{u}$$

and print it with three digits before the point and five after. We assume that the matrices are punched by rows on a data tape in the order **A**, **B**, **C** and are followed by **u**. There are altogether 114 numbers on this tape; these are read into consecutive locations starting at $S0$. The matrix **A** occupies $S0$ to $S105$ (since every address is a multiple of 3), the matrix **B** occupies $S108$ to $S213$, **C** occupies $S214$ to $S321$ and the vector **u** is placed in $S324$ to $S339$.

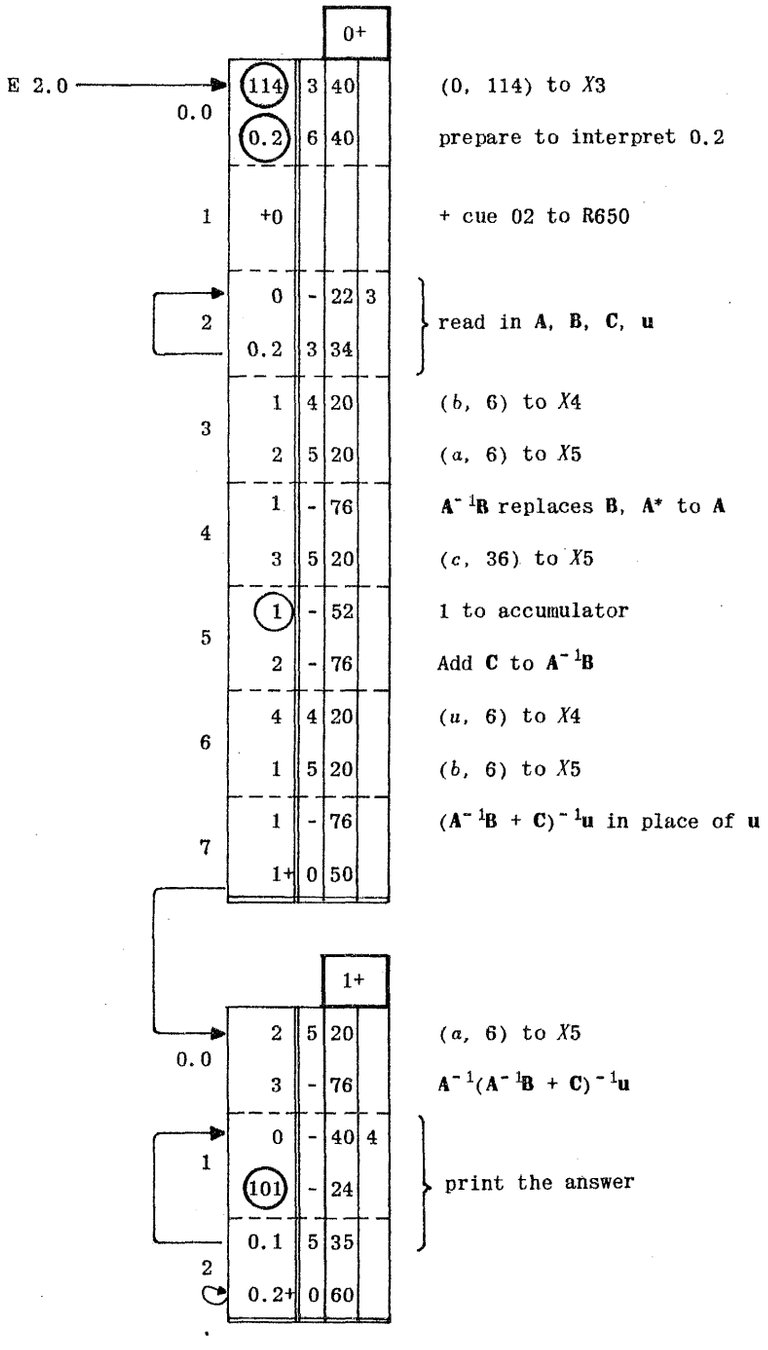
D

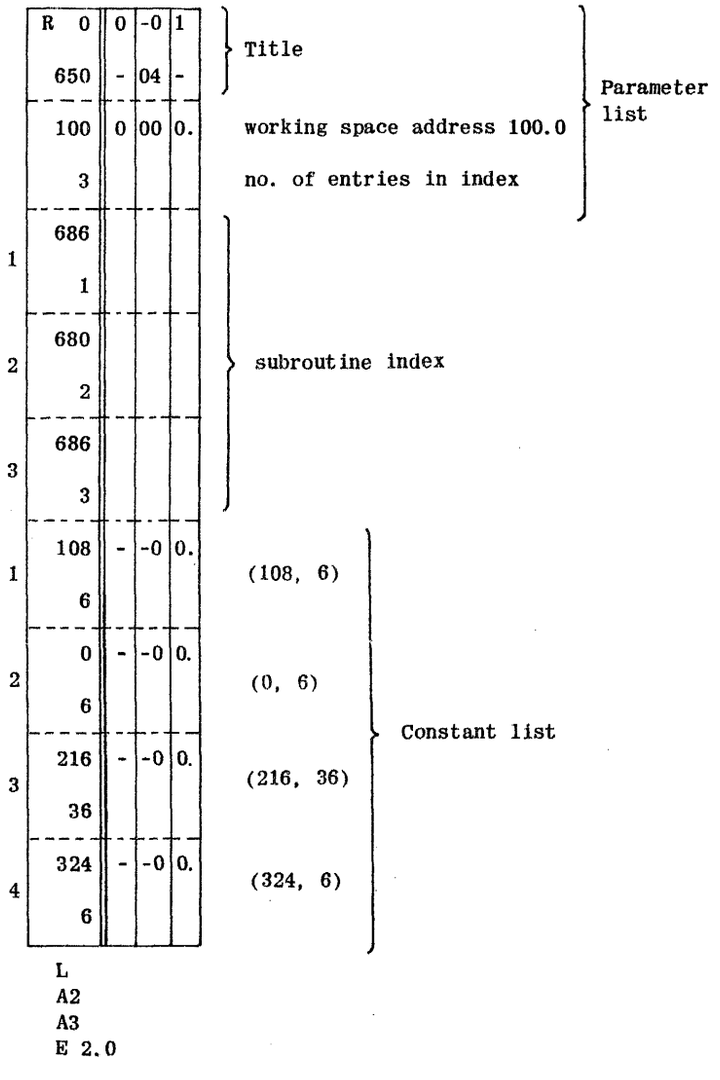
N

VECTOR V

A1

R 0	1	-0	2
650	-	01	-





The tape for R 650 is not included in the Library tape but is placed in the second tape-reader when required. It contains R 650 itself, followed by a special 4-block interlude and the subroutines. The interlude tampers with the parameter-list in a special way and ensures that the appropriate subroutines are accepted; in addition the optional parameter-list is replaced (if necessary) by two words containing the working-space address in the *a*-order and *b*-order respectively, and the programmer's parameter is overwritten by information concerned with subroutine entries. The parameter-list actually used by Assembly is therefore always in the place of the optional list. If the user needs the address of the working space he should call for R 650 parameter 01 or 02. The address of R 650 itself may be obtained by calling for cues 03 and 04; these are ordinary partial cues containing the address in the *N*-part of the *a*-order and the *b*-order respectively. While R 650 and its subroutines are being read in *B*0.0 and 0.2 are used.

When interpreted orders are being obeyed $U5.4_M$ normally holds 3 and the rest of the word is clear (thus $C(5.4) = 3.2^{-13}$); this constant is added to the specified accumulator by the count orders 34 and 36. The user can change this word to another value; for example, he can set it negatively for working backwards through the store, or perhaps to some multiple of 3.2^{-13} for referring to a column or the diagonal of a matrix. It is however used by the subroutines doing matrix and vector operations and must normally contain 3.2^{-13} before the 76-order calls in the subroutine. The subroutine R 680 may be used if the elements of the vectors have addresses differing by a constant difference other than 3; this difference should be set in 5.4_M; similarly in R 681 the vector **u** (but not **v**) may be so handled. In any event $C(5.4)$ is always reset to 3.2^{-13} after all 76-orders.

No provision is made for the possibility of any exponent overflowing. The overflow-indicator OVR is always cleared by R 650 on entry and will be left clear on exit. The order 0 - 52 sets zero in the accumulator with an exponent -2^{19}; this kind of zero is also stored if a 22-order reads zero from the tape. Zero resulting from the subtraction of equal numbers has an exponent 76 less than that of the numbers. The exponent of zero resulting from multiplication or division is the sum or difference, respectively, of the exponents of the operands.

If we make the order-pair in *B*23+.2 of R 650 a stop order-pair then an optional stop will occur before each interpreted order; at this instant *X*6 will contain the order-pair and 7_M will contain its address (and a negative sign-bit for an *a-order*). On entry initially, this stop will occur once before *X*6 and 7 have been set. In general, if the programme stops while interpreting an order then $U1.6_M$ will contain the address of the next order. Stops are liable to occur on input or output (for punching errors or numbers with very large exponents), or on dividing by zero, or if a 20- or 76-order is interpreted and no parameter has been supplied.

Chapter 10

Magnetic Tape

A basic Pegasus installation can be augmented in various ways and the ancillary equipment is so designed that it is still possible to run, without change, programmes prepared for the basic installation. In this chapter we describe magnetic tape equipment and its associated programming techniques.

10.1 General description of magnetic tape equipment

Magnetic tape equipment can be added at any time to a basic Pegasus installation; it provides an auxiliary store of very large capacity. The equipment can also be used with the converter fitted at some installations, when the tape can be regarded as a medium for input and output. The magnetic tape itself is ½ inch wide (13 mm) and is normally wound on metal spools having a diameter of 9¾ inches (24.8 cm). The tape on a spool is called a *reel*; it is normally 3000 feet long (914 metres), but lengths of 600 feet (182 m.) and 1800 feet (549 m.) are also used. The tape is made of thin, flexible, inextensible plastic material coated on one side with a magnetic iron oxide; information can be recorded (or *written*) on to it and read from it by electromagnetic heads in much the same way as with a magnetic drum. Any information written on the tape will remain there indefinitely and can be read as many times as desired until it is overwritten by new information. The process of writing on to a part of the tape automatically erases anything previously written there, whereas the reading process does not disturb the recorded information. Each reel of tape is permanently marked out into *sections*, on each of which either 16 or 32 words can be recorded, depending on the way the tape was originally marked out. The sections marked on a reel of tape are all either of 16 words or of 32 words; no mixture of section-lengths is possible on a reel. Each section has an *address*, which is used to identify it and is permanently recorded on the tape. This address is simply an integer; the first section on any reel always has the address 0, the next 1, and so on. The number of sections on a reel depends on the exact length of the reel, on certain physical properties of the coating, and on the section-length. A 3000 ft. reel has about 10976 16-word sections (a total of 175616 words), or alternatively about 8416 32-word sections (269312 words). The storage capacity of a single reel of tape is thus very large - about thirty times that of the main store of the computer (or the equivalent of about 10976 or 16832 80-column punched cards).

When a reel of tape is to be used with the computer (or the converter) it is mounted (or *loaded*) on a *tape mechanism*, which incorporates the reading and writing heads and equipment for moving the tape at high speed in either direction and for spooling it up. It is quick and easy to load a reel on to a mechanism or to unload it; in fact it is usual to allow 1½ minutes to change a reel. Each mechanism has a small control panel; one of the switches on this panel must be set to show whether a 16-word or a 32-word tape is in use on the mechanism.

There are several tape mechanisms in an installation; and also one or two *tape control units*, each of which contains the electronic circuits for controlling up to five mechanisms, and for reading, writing, checking and storing information. The tape mechanisms are numbered from 0 to 4 (if there are five), in the order determined by a plugboard on the tape control unit which may be set up by the programmer. The arrangement of the plugboard should be altered only when the control unit is *not* obeying a tape operation, and when the Run key on the computer is in the STOP position. Each tape control unit has its own control panel, which is chiefly of concern to the maintenance engineers. On the right of this control panel is a set of five keys which may be used by the programmer to isolate any of the mechanisms, i.e. to prevent writing, thus safeguarding the information on a tape. If a key is in its normal position (tape write) information can be written on to a tape on the corresponding mechanism; if the key is depressed (write inhibit) information cannot be written. These keys provide complete protection from programming errors and faults.

10.2 Programming with magnetic tape equipment

From the programming point of view the tape control unit is capable of obeying tape orders as described below; it contains a *buffer store*, which can be regarded as a waiting room between the magnetic tapes and the computing store.

When magnetic tape equipment is fitted to the computer two additional special registers with addresses 20 and 21 are provided; these are described below. Also provided when magnetic tape, punched cards or a line-printer are used, is a buffer transfer order (76) in which the *N*-digits specify a function as well as an address.

The magnetic tape buffer store can hold 32 words divided into four blocks *W*0, *W*1, *W*2 and *W*3. The 76-order with an *N*-address between 0 and 23 is used to transfer blocks of 8 words between this buffer store and the computing store, as shown in the following table; information in brackets is relevant only to an installation in which there are two tape control units and therefore more than five tape mechanisms, and an extra buffer store, *W*4, *W*5, *W*6 and *W*7.

N-address	*76-order*
0 - 3	Interchange any buffer block of tape control 1 with any computing store block.
4 - 7	Interchange any buffer block of tape control 2 with any computing store block.
8 - 11	Transfer any buffer block of tape control 1 to any computing store block.
12 - 15	Transfer any buffer block of tape control 2 to any computing store block.
16 - 19	Transfer any computing store block to any buffer block of tape control 1.
20 - 23	Transfer any computing store block to any buffer block of tape control 2.

The 76-order with an *N*-address greater than 23 is described in Chapter 11. †

The 76-order is written in a similar way to a 72- or 73-order, the first address specifies the function and, modulo 4, the buffer store block, and the second address specifies the computing store block. For example, the order

2 [5] 76

interchanges the contents of *U*5 in the computing store and *W*2 in the buffer store, and the order

16 [3] 76

writes the contents of *U*3 in the computing store to *W*0 in the buffer store. Four of these orders are needed to change all the words in the buffer store. When using a 32-word tape the whole content of the buffer store is written on to a section of the tape or is replaced by a section read from the tape. When using a 16-word tape either half of the buffer store (i.e. either *W*0 and *W*1 or *W*2 and *W*3) is concerned and the other half is not affected; we can then consider the buffer store as containing two sections of information which are referred to as section 0 (*W*0 and *W*1) and section 2 (*W*2 and *W*3). Fig. 10.1 shows the buffer store and its relationship with the other stores.

A *tape order* is initiated by sending a word to the special register 20, for example by an order such as

20 5 10

The word sent to register 20 is called a *tape control word;* it determines the kind of operation to be carried out. In a large installation with two tape control units, a tape order referring to a mechanism linked to the second tape control unit is initiated by sending the tape control word to special register 21; register 20 refers only to tape control unit 1. In what follows we shall assume that only one control unit is available.

The tape control word may be written on a programme sheet as a pseudo order-pair. A general tape control word may be written thus:

A	- --
0	0 *fg* *h*

Here *A* is the address of the section on the tape,
f is the number of the tape mechanism (0, 1, 2, 3 or 4),
g specifies the section of the buffer (0 or 2),
h indicates the kind of operation to be done, according to the following code;

h = 0 Search,
h = 1 Read,
h = 2 Write,
h = 3 Rewind.

More details are given below, but we first give an example. Suppose that a 16-word tape has been loaded on to mechanism No. 1 and we wish to write on to section 5432 the contents of *U*4 and *U*5 in the computing store. The following sequence of orders may be used.

† In Pegasus 1 with magnetic tape, the 76-order is available only as an interchange order, that is with an N-address 0-3; an N-address greater than 3 in interpreted modulo 4.

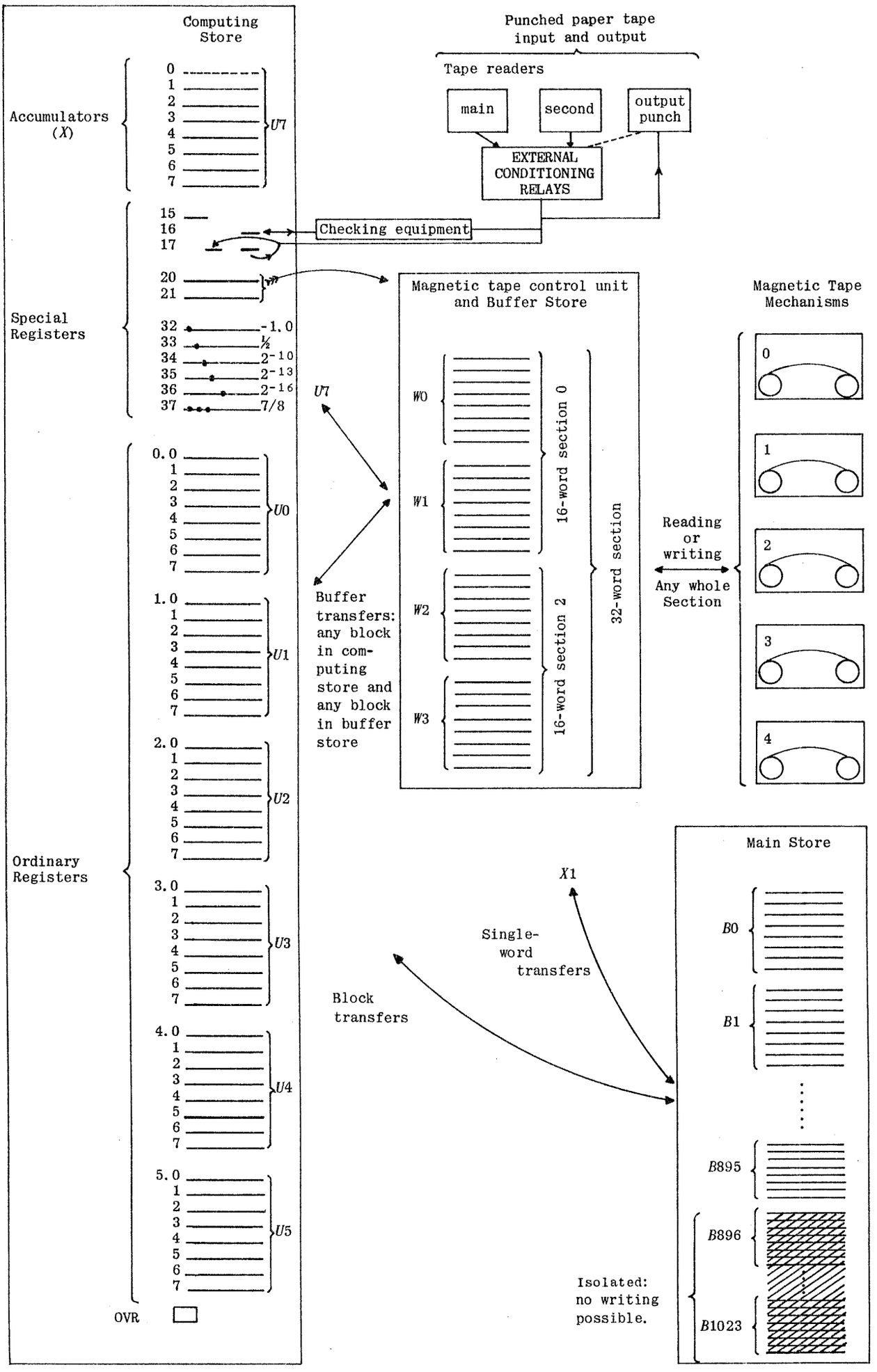
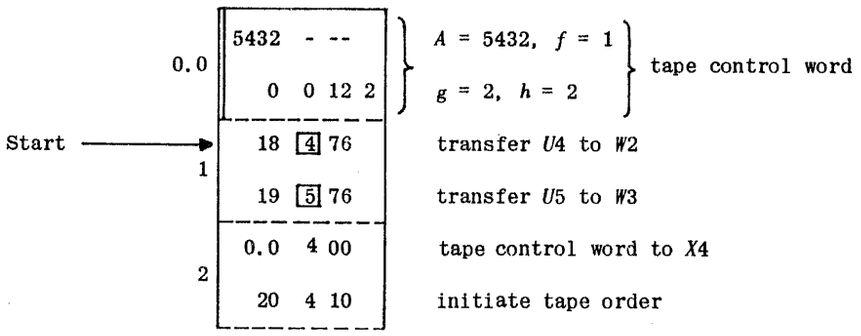


Fig. 10.1 *Diagram of Pegasus and magnetic tape.*



In the tape control word, which is written for convenience in 0.0, the address is 5432, $f = 1$ specifies the mechanism to be used, $g = 2$ means that the transfer to or from tape concerns section 2 of the buffer store (*W*2 and *W*3), and $h = 2$ shows that we wish to write on to the tape. The computer first obeys two 76-orders, which put the contents of *U*4 and *U*5 into *W*2 and *W*3 respectively; the original contents of *W*2 and *W*3 will be lost.† The next two orders send the tape control word to register 20 to initiate the writing operation: the magnetic tape control unit will promptly start searching the tape on mechanism No. 1 for section 5432 and will write the information on that section as soon as it has been found (without disturbing the contents of the buffer store). As soon as the tape order has been initiated by sending the tape control word to register 20 the rest of the process is carried out quite automatically by the tape control unit; the computer is free to carry on with the rest of the programme. The computer will be made to wait if it encounters an order referring to register 20 before the tape operation is complete. This will also happen if a 76-order is found referring to *W*2 or *W*3; but the other section of the buffer store may be freely used. Complete disregard of all such considerations cannot cause faulty operation. Magnetic tape operations are *autonomous* since they are carried out while the computer is doing other operations.

The kind of operation which is carried out by the tape control unit depends on the quantity h in the tape control word. The four operations are as follows.

$h = 0$ (search). Move the tape (forwards or backwards) to the prescribed address, then await a further tape order.

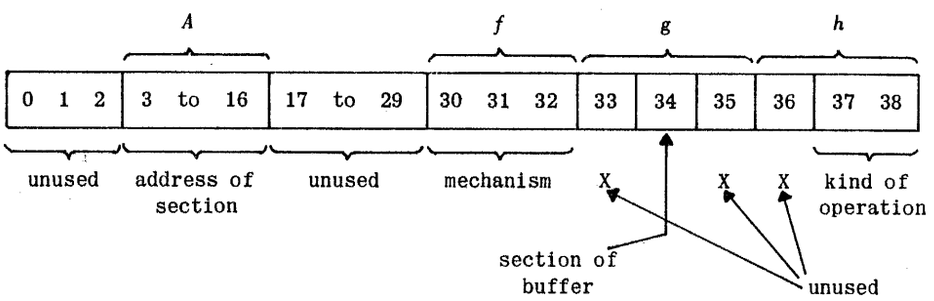
$h = 1$ (read). Read the information in the specified section of the tape and copy it into the buffer store.

$h = 2$ (write). Write the information in the buffer store on to the specified section of the tape.

$h = 3$ (rewind). Rewind the tape on the specified mechanism.

The *search* order may be used when we know that the section we require is some way along the tape; to be able to use it we have to anticipate which section we shall need next, and also be able to carry on with other operations while searching. The *read* and *write* orders include a preliminary search for the section required, and we are therefore not obliged to precede them by a *search* order. The tape control unit is capable of carrying out only one tape order at a time; it cannot, for example, be searching the tape on one mechanism while writing on another. If we initiate one tape order before the previous one is complete then the computer is held up. The *rewind* operation is, however, quite autonomous; as soon as rewinding has been initiated on one mechanism then the tape control is free to operate with the other mechanisms while the first tape is being rewound. Note that g is the part of the tape control word used to indicate which section of the buffer store is to be used in 16-word reading and writing operations; the value of g is immaterial when searching or rewinding or when we are using a 32-word tape.

The binary digits in the tape control word are allocated as follows:



The values of the unused digits are entirely without effect on the tape control unit. The address in the tape control word is an integer with its least-significant digit in digit-position 16; if we have a tape control word in an accumulator and we add 2^{-16} to it the word will have its address increased by

† Provided that it is not necessary to keep the contents of *U*4 and *U*5 in the computing store, the 76-orders could be written as interchange orders:

2 4 76 interchange *W*2 and *U*4

3 5 76 interchange *W*3 and *U*5

When these orders are obeyed, the contents of *U*4 and *U*5 will be transferred into *W*2 and *W*3 and the original contents of *W*2 and *W*3 will be copied into *U*4 and *U*5. These orders would normally be written in this form for Pegasus 1, as the 76-order is available only with an *N*-address between 0 and 3. If an *N*-address greater than 3 is encountered on Pegasus 1, the 76-order will be obeyed as an interchange order i.e. the *N*-address will be interpreted modulo 4.

one. This is a common requirement in magnetic tape programmes; the constant 2^{-16} has therefore been provided in a special register, whose address is 36. The order

36 5 01

will increase by one the address part of a tape control word in *X*5. The orders of group 4 can often be used to change the other digits; for example, the order

(1) 5 41

will add one to *h* and can therefore be used to turn a *read* word into the corresponding *write* word.

Register 20 is made up of a nickel delay line in the tape control unit. When an order of group 1 is obeyed by the computer the final content of this register determines the tape order to be carried out. Other orders may refer to it without initiating a tape order; for example, the order

20 5 00

will place in *X*5 the tape control word which initiated the last tape order, and will have no effect on the tape control unit. An order like this will hold up the computer if the last tape order is not complete. Orders like the following are sometimes useful.

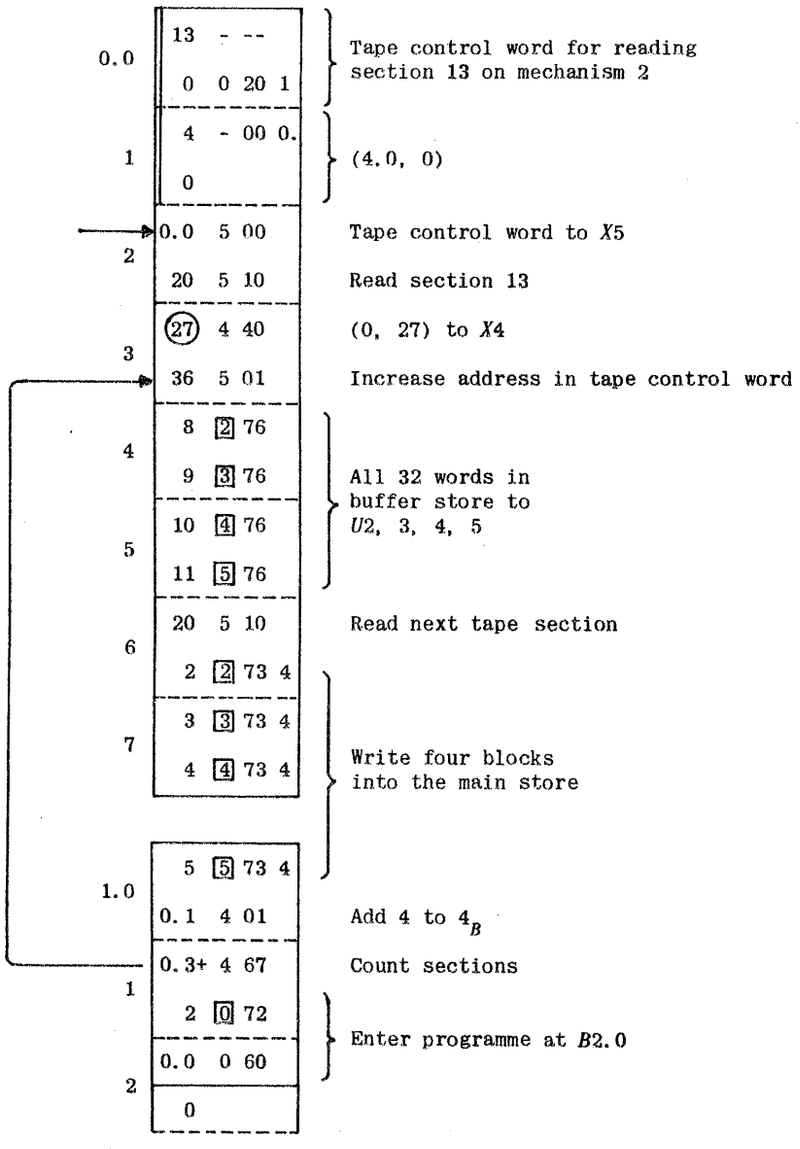
36 3 00

20 3 11

These cause the last tape order to be repeated, but with its address increased by one. They are thus equivalent to an order for reading the next section, if the last tape order was a read order.

Programmes which write information on to magnetic tape should not use a tape address below section 1, as it is customary to record in section 0 special identifying information about the tape. The use of section 0 is described further on Page 234.

As an example of a short sequence using magnetic tape, let us suppose that we have a programme recorded on 27 32-word sections on tape mechanism 2 starting at section 13. This programme is to be put into the main store in blocks *B*2 to *B*109, after which *B*2 is to be copied into *U*0 and a jump to 0.0 is to take place. The following sequence will do this in 2.3 seconds (if section 0 of the tape is under the reading head initially); of this time 0.7 seconds is spent searching for section 13. Note that the block transfers to the main store are done while the tape control unit is reading the next section.

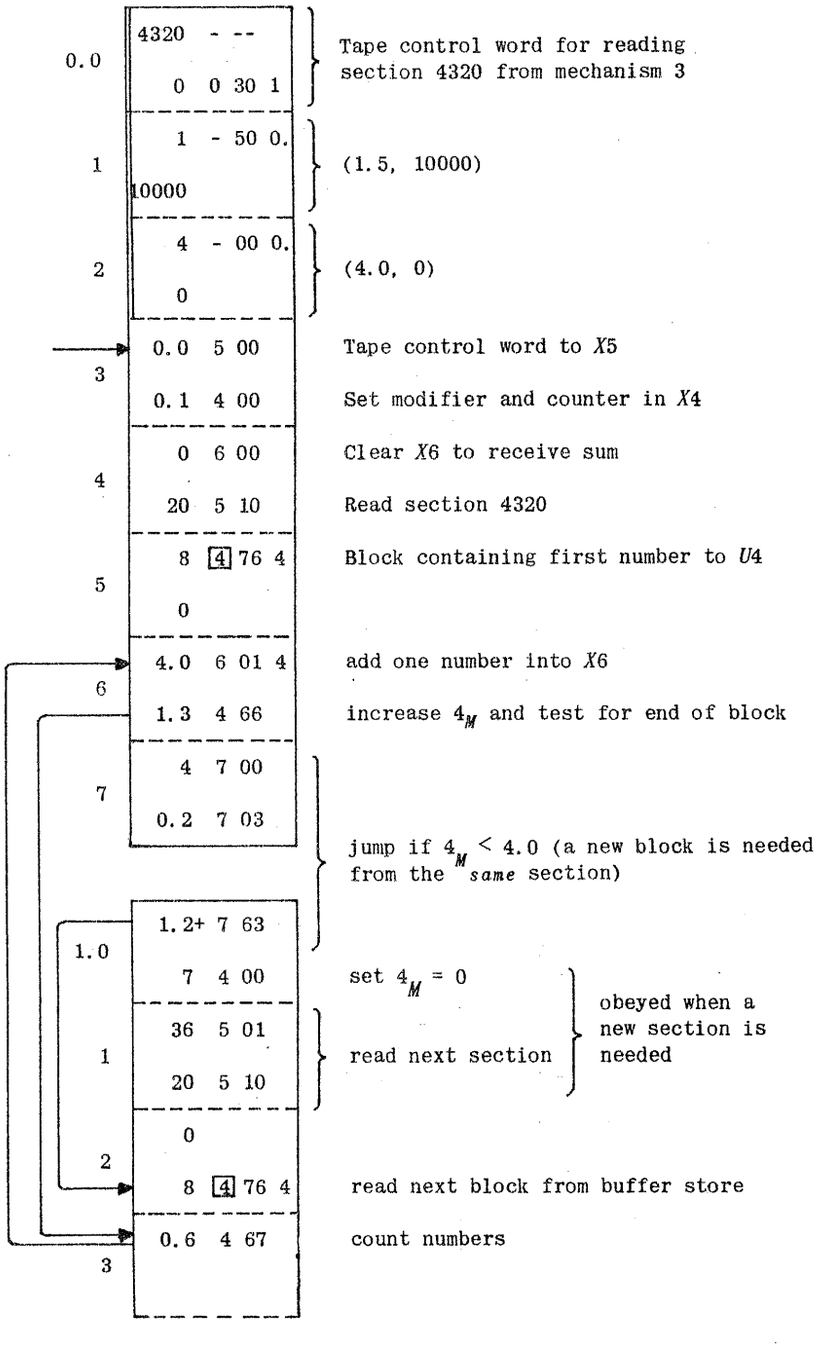


The 76-order may be modified, and indeed often will be in many programmes. Modification of this order is similar to that of the 72- and 73-orders in that the block part of the modifier is added to the first address, i.e. to the address of the buffer block. Since there are not more than four blocks in the buffer store associated with a certain piece of ancillary equipment, only the two least-significant binary digits of the block modifier are actually used, the remaining digits are without effect. Any carry over from the addition of the two least-significant digits of the block modifier to the *N*-address is ignored, i.e. the modification is carried out modulo 4. This is important because we can often use the same modifier for two purposes:

(a) for modifying 76-orders to deal with successive blocks in the buffer store,

(b) for modifying 72- or 73-orders to deal with successive blocks in the main store.

To provide an illustration of modified 76-orders, let us consider the problem of adding up 10000 numbers stored in consecutive locations on a 32-word tape on mechanism 3; the first of the numbers being in location 1.5 in section 4320. The following sequence can be used; it takes about 31 seconds if the first section is immediately available. Note that this sequence does not make use of the autonomous properties of the tape control.



We must now consider the *timing* of magnetic tape operations, because it is usually possible to speed up certain operations by paying some attention to it. The order sending the tape control word to register 20 takes only the normal time of an order of group 1 (i.e. 3 word-times if it is an *a*-order, or 2 if a *b*-order, an average of about 0.3 msec). If the tape is in position to be read or written on the operation will be complete after 43 milliseconds (for a 16-word section) or 56 milliseconds (for a 32-word section); these times apply whether the tape is moving or not. When the operation is complete the tape will come to rest in the right position for dealing with the next section (i.e. that with an address 1 greater) unless another tape order sets it in motion again, but this makes no difference to the timing. We normally try to arrange the programme so that consecutive sections are dealt with in increasing order of their addresses, the tape then moves forwards only. The time for searching the

tape for a specified address is also 43 msec (or 56 msec) per section; i.e. the searching rate is 23 sections per second (or 18 per second). No time need be allowed for changing from one mechanism to another. The 76-order takes about 1.32 msec (actually 11 word-times if it is an *a*-order, or 10 if a *b*-order), assuming of course that the required block of the buffer store is not in use for a tape order. Rewinding a full 3000 ft. reel takes about 4 minutes; it is possible to rewind several reels simultaneously if programming considerations permit.

Searching for a specified address can take place with the tape running in either direction, but reading and writing occur only with the tape moving forwards. If we have just read section 50 on a reel and we then wish to read section 49 the sequence of events is as follows: first the tape is started up in a forwards direction and the next address (51) is read, the motion of the tape is then reversed and scanned until the required address (49) is found, the tape is then again reversed and the section is read. The whole process is quite automatic; it takes roughly three times as long as reading section 51 after section 50.

In many technical and scientific applications it is preferable to use 16-word tapes, the processing of one section in the buffer store can be carried out while the other section is being filled from the tape (or written on the tape). It is often found that the calculation is then only slightly slowed by the tape, because the time required to process 16 numbers is often greater than the time to read them in (about 43 msec). If the programme can keep the tape running at full speed then the whole reel is dealt with in about 8 minutes (this time is the same for 32-word or 16-word sections). This can hold in a changed version of the programme given above for adding 10000 numbers; it then takes only 17½ seconds instead of 31 seconds (still with 32-word sections).

When the tape control unit has completed a tape order, we can spend a short time calculating before initiating a tape order referring to the next section on the reel; if this time is not more than about 20 word-times (2½ msec) the tape will not be slowed appreciably. In this time we can usefully obey 1 or 2 76-orders. If the time is slightly more than 20 word-times the tape will be slowed only slightly. Thus with no intervening 76-orders or other calculation the interval between the end of the first tape operation and the end of the next will be the minimum of 43 msec. (or 56 msec.); with 5 msec. calculation this interval is increased by only 3 msec. †

Before running a magnetic tape programme it is usually important to check that the write switches on the tape control unit and 16/32 word switches on the tape mechanisms are correctly set, and that the correct tape is mounted on each mechanism to be used. It is not sufficient to rely on the computer operator for this check. There is a Check Tape subroutine, stored with the Initial Orders in the isolated part of the main store, which checks the settings of the write and 16/32 word switches on a magnetic tape mechanism. It leaves certain information in the computing store about the tape, and will print this information if required. It has the following specification:- ††

Name: CHECK TAPE

Store: B962, 969, 970, 975, 976, 977

Uses: *U*0, 1, 2; $X6_M$, 7; *B*0;
*W*0, 1, 2, 3.

Entries:

37 6 01
74 [0] 72 6
0.2 6 66

Check and print

37 6 01
74 [0] 72 6
0.2 0 60

Check without printing

Time: Approximate times in milliseconds:

	16-word	32-word
With Printing	1200	1300
Without Printing	500	600

These times assume that, on entry, section 0 of the tape is under the reading head.

Link: Obeyed in *U*1.0 and left unaltered in *X*1.

Initial Settings

Before entering Check Tape a programme must first set *X*6 as follows:

$$x_6 = m + 8i + t$$

where m = mechanism number
i = 1 if the write switch should be off (writing inhibited)
= 0 otherwise
t = 16 or 32, according to the setting of the 16/32 word switch.

---

† The times given above for magnetic tape operation were obtained experimentally on ElectroData mechanisms and are therefore approximate. There are variations between mechanisms, especially in the amount of calculation which may be done without slowing the tape.
The details of the timing for the Decca mechanisms are slightly different from the ElectroData although the minimum tape transfer times are the same.

†† There is no Check Tape programme in the 4096 Initial Orders, but the programme is available as a library subroutine, *R*930.

The modifier and sign of X6 should normally be clear, but if it is desired to ignore the setting of the write switch the sign bit should be 1 and $i = 0$.

Checking

Check Tape first checks the settings of the write and 16/32 word switches on the specified mechanism. If either is wrong there will be a 77 stop in 0.0; if the 16/32 word switch is wrong the OVR will also be set. On correcting the settings and operating the RUN key the tape will be checked again.

A tape read failure will occur if the setting of the 16/32 word switch is not consistent with the type of tape mounted. On correcting the fault and operating the REPEAT key on the tape control unit the tape will be checked again.

Section 0 on Magnetic Tape

Check Tape assumes that the following information has been written in the first three words of section 0 on each magnetic tape.

0.0	$+L$	Length of tape in feet (nominal)
0.1	$+S$	Serial number of tape
0.2	A - -- 0.	Address of last section on the tape $\times 2^{-16}$
	0	(A = Number of sections - 1)

This information is usually written on to a tape before it is brought into use. Thus, all programmes using magnetic tape should be checked to ensure that they do not overwrite the first three words of section 0.

Printing

If the printing entry to Check Tape is used, and if the settings of the write and 16/32 word switches are correct, the following information will be printed on a new line:

$m\ t/L/S/n\ =$

where m = mechanism number (1 digit)
t = 16 or 32: the setting of the 16/32 word switch
L = nominal Length of tape in feet (4 digits)
S = Serial number of tape (5 digits)
n = number of sections on the tape (5 digits)
= indicates that the write switch is on
≠ indicates that writing to the tape is inhibited

Check Tape printing occupies 22 character positions across the page, leaving room for the master-programme to print further identifying information on the same line if required. The layout may be seen from the following example:

3 16/3000/00192/10816≠

Exit

On exit from Check Tape the contents of section 0 will be available as follows:-

Section 0, block 0 in $U2$
1 in $W1$ (Buffer block 1)
2 in $W2$, 3 in $W3$ } if checking 32-word tape

The master programme may therefore easily transfer further identifying information to or from section 0 and print it if required.

6_C is not changed by Check Tape and may afterwards be used by the master programme to determine m and t, but if the sign of $X6$ is 1 it is not possible to use i to determine the setting of the write switch.

A writing marker is left in $X7$ and $U1.1$. This marker will be zero if writing is inhibited but non-zero otherwise.

Steering Tape

It is possible to check a magnetic tape by using a steering tape punched as follows:-

T 6
896 - 100.
$m+8i+t$ } setting of $X6$
J 970.2

Stops in Check Tape

*U*0.0

0	0	77
20	7	13

77 stop if the 16/32 word or write switches are incorrect.

If OVR clear, x_4 = ½: write switch wrong

If OVR set, x_4 = -1.0: 16/32 word switch wrong

If OVR set, x_4 = -½: both switches wrong

On correcting the fault and going to RUN the tape will be re-checked.

*U*1.0

0	[2]	76
2.2	4	00

Tape failure stop if the 16/32 word setting is not consistent with the type of tape mounted. On correcting the fault and operating the REPEAT key the tape will be re-checked.

*U*1.3

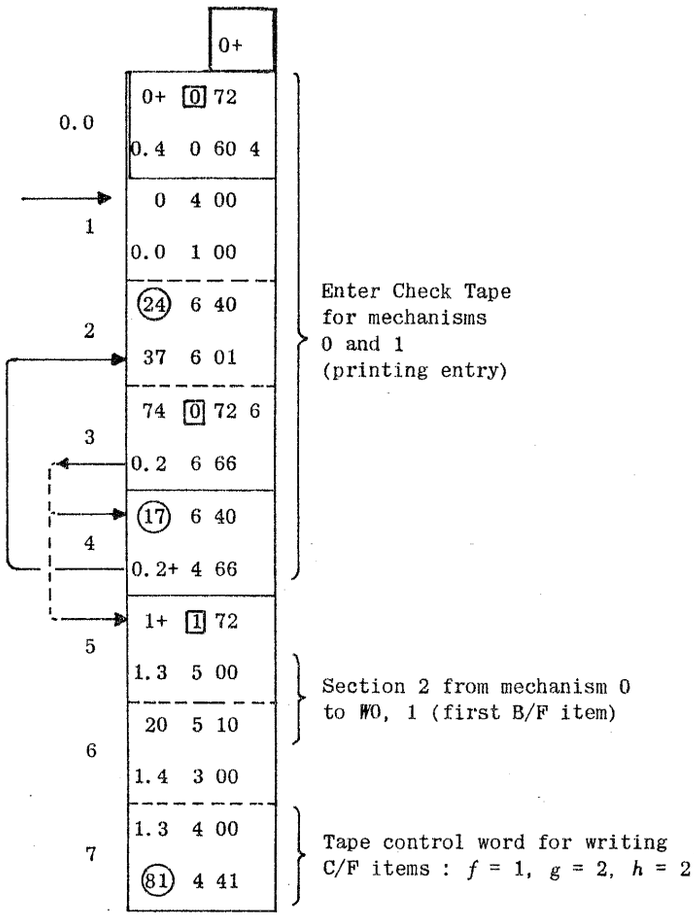
20	7	11
2	[3]	76

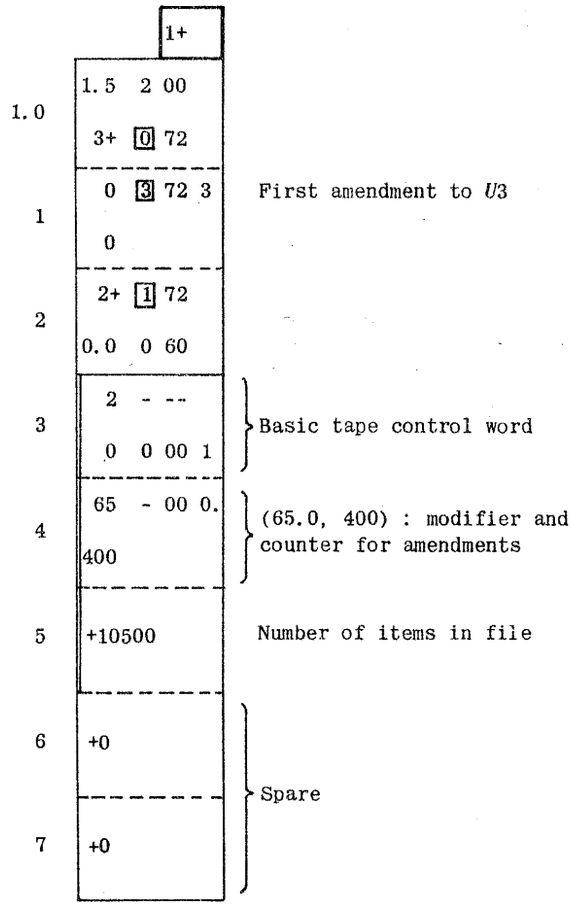
Tape failure stop, with writing inhibited OR writing with OVR stop. These may occur if the first three words of section 0 do not contain the information specified above.

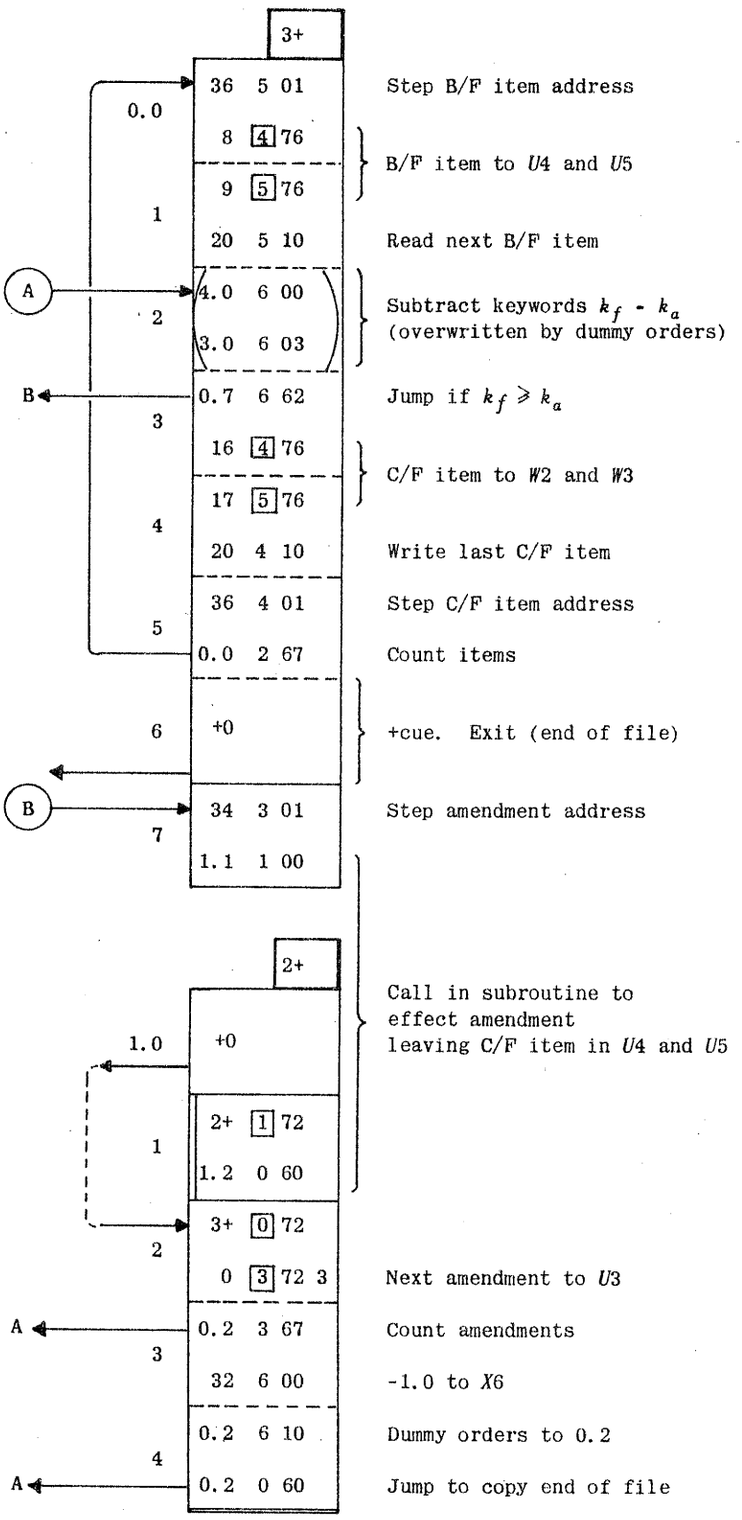
As an example of a slightly more complicated programme let us suppose we have a 16-word tape carrying information, for example about items in stock in a warehouse; this will be called the brought-forward tape (or B/F tape). We have to effect a number of amendments to some of the recorded information and produce a new, up-to-date tape called the carried-forward (or C/F) tape. This new tape is to be a copy of the old, except for a few amendments. To simplify the problem we shall suppose that each item of information occupies exactly one section; the first word of each section is a *keyword* identifying the item (e.g. it could be a part-number). The tape can be regarded as a file, and we shall assume that there are 10500 items on it arranged in ascending order of their keywords. We shall suppose that the information relating to one amendment occupies one block, the first word of which is the keyword identifying the item to be amended in the main file. We assume these amendments are held in the main store, properly sorted in ascending order of keywords; we suppose there are 400 of them and that the first is stored in *B*65. We shall not specify exactly how an amendment is to be carried out since this is left to a subroutine to be drawn up to suit the details of the problem; it should have the following brief specification:

> Amend the brought-forward (B/F) item in *U*4 and 5, using the amending information in *U*3, and leave the carried-forward (C/F) item in *U*4 and 5, then obey link in *X*1. Do not disturb *X*2, 3, 4 or 5.

The main programme is given below. The B/F tape is to be put on mechanism 0 and the C/F tape on mechanism 1. Check Tape is entered at the beginning of the programme for each of the two mechanisms: this will ensure that the 16/32 word switches on both mechanisms are switched to 16 and that the write switch in OFF for mechanism 0 and ON for mechanism 1. It is most important that the B/F tape should be isolated (write key switched OFF), so that this information is safeguarded should an accident occur. The keywords are denoted by k_f (file item) and k_a (amending information). The cues to the amending subroutine and the exit sequence are to be written in 2+.0 and 3+.6 respectively (this would normally be done by Assembly). Two tape control words are kept in *X*4 and *X*5.


0+
0.0
0+ 0 72
0.4 0 60 4
1
0 4 00
0.0 1 00
2
24 6 40
37 6 01
3
74 0 72 6
0.2 6 66
4
17 6 40
0.2+ 4 66
5
1+ 1 72
1.3 5 00
6
20 5 10
1.4 3 00
7
1.3 4 00
81 4 41
Enter Check Tape for mechanisms 0 and 1 (printing entry)
Section 2 from mechanism 0 to #0, 1 (first B/F item)
Tape control word for writing C/F items : f = 1, g = 2, h = 2


1+
1.0
1.5 2 00
3+ 0 72
1
0 3 72 3
0
First amendment to U3
2
2+ 1 72
0.0 0 60
3
2 - --
0 0 00 1
Basic tape control word
4
65 - 00 0.
400
(65.0, 400) : modifier and counter for amendments
5
+10500
Number of items in file
6
+0
7
+0
Spare



It would be possible to write the amendment subroutine so that it could insert or delete items in the file, but this would normally be arranged by providing extra facilities in the main programme. The amendment subroutine can detect when an insertion is needed because $C(6) = k_f - k_a$ is not then zero. Often there will be more amendments than can be held in the main store and provision would then have to be made for reading in further amendments.

In most data-processing applications the main file is kept on tape with 32-word sections. This is because it is often copied to produce an up-to-date file and a given amount of information occupies less space and can be transferred faster than if a 16-word tape were used. Amending information is often on 16-word tapes. By suitable programming *variable-length items* can be handled; the usual technique is to make the first word of each item (normally a keyword) negative by adding -1.0 to it and to ensure that none of the other words are negative. The programme has to examine the sign of every word and is thus able to fix the beginning and end of each item. A number of programmes have been written using this technique; they are useful not only if the items have variable length but also if the items do not fit neatly into sections or blocks, for example if the natural length of an item is 21 words. The variable-length technique minimises the amount of tape needed to hold a given quantity of information and hence the time required to read, write or rewind the tapes; it also enables one general purpose programme to be used in several different applications.

When bringing a file up to date it is important not simply to alter the information on a tape, but to produce a new tape which can be used in place of the old; in this way vital information can be safeguarded because the original brought-forward file is never written on. If a power-failure or other calamity should occur in the middle of the up-dating process then it is always possible to go back to the beginning and repeat the whole process. As an extra precaution the switches provided on the tape control unit can be used to isolate any of the mechanisms, i.e. to prevent writing; this provides complete protection from programming errors and faults. There is also a stop which occurs if an attempt is made to write on a tape when the overflow-indicator (OVR) is set; this is similar to the stop when writing into the main store. It occurs if a tape control word with $h = 2$ is sent to register 20 with OVR set; nothing happens, of course, if OVR becomes set in the middle of a writing operation. In all rewinding operations the tape is moved clear of the heads and rewinding proceeds at full speed; the tape is then left ready for unloading or for reading or writing the first section.

Magnetic tape can be effectively used for *sorting*. A common process uses four mechanisms; a number of items for sorting are read into the main store and sorted by merging; these items are then written on to one of the reels of tape, and form a *string* of sorted items about 700 blocks long. The process is then repeated but the next string is put out on another tape. The next string after this is written on to the first tape, and so on, so that ultimately we get two reels of tape each bearing strings of sorted items about 700 blocks long. These two tapes are then rewound, and the strings read back and merged to form new strings of about twice the original length, which are written alternately on to two other tapes. The process is repeated until all the items have been merged into one long string.

The magnetic tape equipment is fully interlocked and it is consequently impossible for lack of attention to timing details to cause incorrect operation. The following is a summary of the interlocks in detail.

▼ (a) When the tape control unit is busy, an order initiating another magnetic tape operation will be held up until the previous order has been completed (in fact reference to registers 20, 21 or to the unused registers 22 and 23 has this effect).†

(b) A buffer transfer order (76) referring to a part of the buffer store being used for reading or writing on the tape will be held up until the reading or writing is complete. In 16-word transfers one section of the buffer store may be used for a read or write order, and the other simultaneously for 76-orders. All the buffer store may be used for buffer transfer orders during search or rewind operations.

(c) An order to rewind a particular mechanism does not occupy the tape control unit for more than a few ▲ milliseconds; other mechanisms are then available.

A number of *checks* are applied during magnetic tape operations; they are all done quite automatically by special circuits and *there is no need for any programmed checking whatsoever*. It is important that these checks are applied because reading or writing errors are liable to occur occasionally (once every few reels) due to external causes, such as specks of dust. These errors are not often due to the equipment being faulty or out of adjustment; they will usually not occur again if the tape operation is immediately repeated. To attain the same high standard of reliability in the magnetic tape equipment as in the computer itself thorough checks and automatic repeating are therefore provided.

▼ Details of these checks are as follows.

(a) The parity digit stored with each word in the computing store (see Sec. 6.7) is carried with the word into the buffer store and on to the tape when writing. The parity of each word is checked as it is written on the tape; failure to pass the check stops the computer (and the writing process) and lights the *buffer parity failure* light on the tape control unit.

(b) While a section is being written on to the tape a 6-bit checksum is formed by adding together each group of 6 bits in each word††; this checksum is written on the tape at the end of the section and is also stored. Another head in the mechanism reads back the section just written and builds up the checksum again from it. The checksum stored in the writing circuits, the one built up during the check-reading, and the one on the tape are then compared. If these three are not identical the section is automatically written and checked again. At most five attempts at writing are made, and if there is still failure the computer stops. (This kind of failure will usually occur if an attempt is made to write on an isolated tape.)

(c) While a section is being read from the tape a checksum is formed and compared with the one on the tape; disagreement causes the section to be read again automatically. At most five attempts are made, and the computer then stops.

(d) During an order transferring information from the buffer to the computing store the eight words entering the computing store are subjected to a parity check; the *main store parity failure* lamp comes on and the computer stops if this check fails.

(e) Each address on the tape has a parity digit associated with it, which is checked every time the address is read.

(f) A special check is applied to ensure that no *clock-digits* are missed (these digits correspond in function to the sprocket holes in punched paper tape).

▲ (g) A further check is used to detect distortion of the tape or misalignment of either of the heads.

If any error is detected a bell in the tape control unit will sound once; this is normally the only indication of the error since the operation will usually be successful the first time it is automatically repeated. If at the end of five attempts there is still a failure, the bell will ring continuously.

---

† On Pegasus 1, the initiation of a magnetic tape order will also be held up if paper tape (input or output) is busy. Paper tape orders are *not* held up while magnetic tape is busy.

†† This addition is done in a special 6-bit checksum register which has *end-around carry*, i.e. a carry off the left end of the register is added into the right-hand end.

When a tape order is being obeyed the *magnetic tape* busy light on the control panel of the computer is lit.

10.3 Dealing with magnetic tape failures

Types of Faults

Magnetic tape faults fall into 4 categories:-

(i) A tape failure (f), indicated by the ringing of a bell and the read failure light on the tape control unit.

(ii) A tape busy stop (b), identified by the fact that the 'tape busy' light is on but no tape unit is operating.

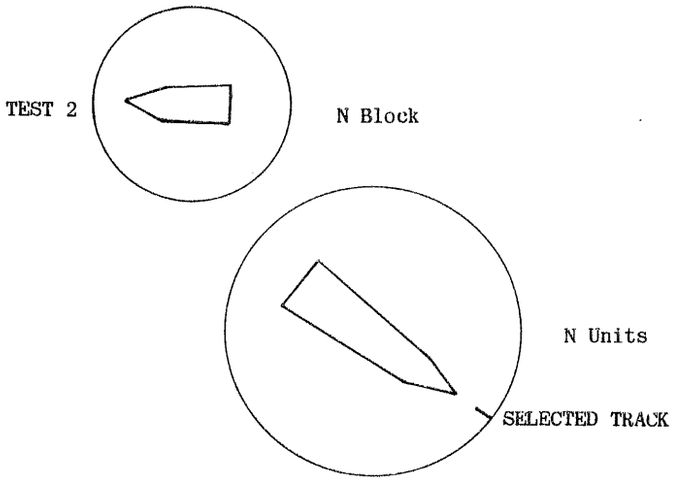
(iii) A buffer parity failure (p), which may show up as a drum parity failure after a 76-order, or as a buffer parity failure after a tape write order. The latter failure lights the magnetic tape busy lamp on the computer and also the buffer parity lamp on the tape control unit.

(iv) A tape wreck (w).

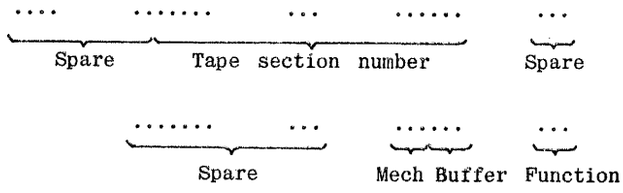
Register 20

Contents: The current magnetic tape instruction.

Selection on Monitor Switches: Test 2, Selected Track.



Interpretation:



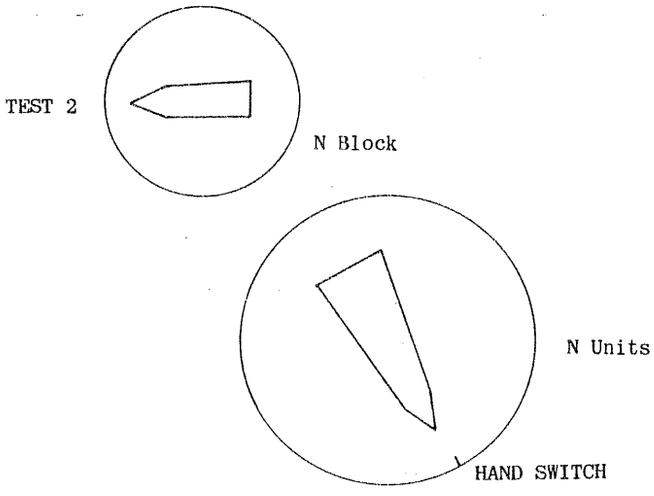
Function: 0 = Search
1 = Read
2 = Write
3 = Rewind

Buffer: 0 = First half of buffer } 16-word sections only
2 = Second half of buffer }

Tape Failure Indicator

The content of this indicator is present only whilst the tape failure bell is still ringing; it is cleared when the 'Clear' key on the tape control unit is operated, or when the instruction is successfully repeated.

Selection on Monitor Switches: Test 2, Handswitch.



Interpretation:

FRS			IB	
111.	.....	...	.11...	...
			0	
	.....	...	...1..	...

A 1-bit in the display indicates a fault according to the following code:

F Address Forward failure
R failure Repeated five times
S check Sum failure
I Inter-character gap failure
B address Backward failure
O clock Omission failure

Operator Checks

When a magnetic tape fault occurs, the operator should check the following points. The letters in brackets indicate the types of faults to be expected.

1. Is the write switch for the selected mechanism correctly set? It should be ON normally, if the tape is being written on; but note that 'Check Tape' writes to section 0, Assembly to section 512 and directive Q to section 511 of an isolated tape. (f)

2. Is the 16/32 word switch correctly set? (f)

3. Is the correct tape mounted on the selected mechanism? (f)

4. Is the tape correctly threaded and running free from contact with the edge of the spool? Is the spool firmly clipped to the hub? (b, f, w)

5. Is the movement of the tape being obstructed by a bad join? (b, f, w)

6. Has the mechanism reached an 'end of tape' stop? On the Decca mechanism an 'end of tape' stop occurs when a transparent patch comes under the left detecting light. (b, f)

7. Is the key on the tape control unit corresponding to the selected tape mechanism in the 'Auto' position? (b)

8. Is the selected mechanism switched on and in the 'remote' state? The green 'remote' light should be on. (b)

9. If the fault is a buffer parity failure, check that the buffer has been cleared since the computer was last switched on and since the last magnetic tape reading failure. (b)

10. Is the tape on the other mechanism of a Decca twin-unit loaded on the perforated leader? (b, f)

Programme Checks

The following checks should be applied to the programme.

1. Is the programme intended to use the selected mechanism? (b, f)

2. Is the programme calling for a high numbered section not present on the tape (or not yet reached)? (b, f)

3. Is the programme reading from a section on the tape which it has not previously written on? (f)

Action

Bring the Run key to STOP. Further action depends on the type of fault as follows.

Tape Failure

If a tape failure is due to an operator's error the error should be corrected and the REPEAT key on the tape control unit depressed. However, care should be taken if the 16/32 word switch is wrongly set. †

† If a tape order has failed owing to the 16/32 word switch being set to 16 instead of 32, it is possible that part of the buffer which is not locked out may have been overwritten since the tape order by a 76-order. In this case some restart procedure must be used.

If the failure is due to a programming error the order may be cancelled by operating the CLEAR key. If the order is a tape reading order there will be wrong parity in the buffer store which must be cleared (see below) before proceeding.

If the failure is not due to an operator or programming error the REPEAT key should be depressed once or twice. If the order still fails the tape may be moved back and forth once or twice and the REPEAT key again operated. If the fault is still present the maintenance engineer should be called. If the order succeeds the RUN key may be operated to proceed with the programme.

Tape Busy

If a tape busy failure is due to an operator's error the error should be corrected and the tape order can then proceed. If an 'End of Tape' stop is reached the tape should be run backwards several yards under manual control.

If the failure is due to a programming error the order may be cancelled by operating the CLEAR key.

If the failure is not due to an operator or programming error the maintenance engineer should be called. If he is not available the fault may sometimes be cleared by re-loading the mechanism and moving the tape forward past the first few addresses. If the order succeeds the RUN key may be operated to proceed with the programme.

Buffer Parity Failures (p)

If the failure is due to an operator error, or to clearing a tape failure, the operator may insert words with correct parity in the buffer store by the following procedure:-

Inhibit parity failure stops

Obey the following four orders by setting each in turn on the handswitches and going to -

MANUAL	NORMAL	SINGLE SHOT
0 [6] 76		
1 [6] 76		
2 [6] 76		
3 [6] 76		

Uninhibit parity failure stops.

If the failure shows up as a drum parity failure on a 76-order, there will be wrong parity in one word of the computing store block, X, referred to in the 76-order. Words with correct parity may be brought into this block by inserting the order

0 [X] 72

in the above sequence of manual orders.

After a buffer parity failure it is not normally possible to continue with the programme merely by operating the RUN key. It is usually necessary to enter the programme at some Restart point so that the lost information is regenerated.

Tape Wreck (w)

Call for the maintenance engineer. Do not attempt to use the unit again until the cause of the tape wreck has been established.

10.4 Magnetic tape programmes in the Initial Orders†

The following three programmes are available in the isolated part of the main store, and work under the control of a steering tape.

1. *Read from Magnetic Tape*

A programme to read information from magnetic tape into the main store. A steering tape to use this programme should be punched as shown on the left below.

T2	
+*m*	Tape mechanism number
+*a*	Number of first tape section
+16 (or 32)	Number of words in a section
+*n*	Number of sections
J958.0 - B.0	*B*.0 = first main store address

After transferring the specified sections from magnetic tape to the main store, the programme will re-enter the Initial Orders to read paper tape. The J-directive will often be followed by F-, I- or P-directives to print the required information.

† Only the first of the programmes in this section, Read from Magnetic Tape, is in the 4096 Initial Orders. The steering tape for the 4096 version of the programme is similar to that for the 7168 version, but should end with J513.0 - *B*.0 in place of J958.0 - *B*.0. The other two programmes in this section are available as complete programmes which may be read into the unisolated part of the store.

2. *Write to Magnetic Tape*

A programme to write information from the main store on to magnetic tape. A steering tape to use this programme should be punched as follows:

T2	
$+m$	Tape mechanism number
$+a$	Number of first tape section
+16 (or 32)	Number of words in a section
$+n$	Number of sections
J959.0 - *B*.0	*B*.0 = first main store address.

After writing the required information to magnetic tape, the programme will re-enter the Initial Orders to read paper tape.

3. *Copy Magnetic Tape*

A programme to copy information from one magnetic tape to another. It copies one section at a time, but the section numbers may be different on the two tapes. The programme is not intended for copying directly from one part of a tape to another part of the same tape: this is most efficiently done by routing the information via the main store or a second tape mechanism.

A steering tape to use this programme should be punched as follows:

T2	
$+m_1$	Tape mechanism to be read from
$+a_1$	Number of first tape section on m_1
$+m_2$	Tape mechanism to be written to
$+a_2$	Number of first tape section on m_2
$+n$	Number of sections to be copied
+0 (or -1)	Rewind after copying if +0; no rewind if -1
J979.0	

The routine will work with 16- or 32-word sections, but care should be taken to ensure that the 16/32 word switches are correctly set before a tape is copied.

After copying the specified sections, the programme will re-enter the Initial Orders to read paper tape.

Chapter 11

Punched Cards and Line Printer

Punched card equipment may be attached to a basic Pegasus 2 computer but not to a Pegasus 1. The following chapter contains a description of this equipment and the details of the relevant programming techniques.

11.1 Pegasus 2 Punched card system

As an optional extra, equipment for punched card input and output can be added to Pegasus 2. An I.C.T. card reader (Hollerith type 581) is used, there being versions for Hollerith, I.B.M., Bull or Powers-Samas 80- or 65-column cards. An I.C.T. card punch (Hollerith type 582) is used, of which there are versions for the same types of cards. 65- and 80-column readers or punches can be interchanged by the use of plugs and sockets, a valuable facility in installations using both types of cards. The card reader runs at 200 cards per minute and the card punch at 100 cards per minute.

As an alternative, the Bull PRD machine can be attached, this being a reader and an independent punch in the same housing. It runs at 120 cards per minute and is suitable for Hollerith, I.B.M. and Bull 80-column cards.

When punched card input and output equipment is attached to the computer, a card control unit consisting of a double-bay free standing cabinet is also added. When a card is read, the data from the card passes from the card reader, by way of conversion and checking circuits, into a buffer store where it is held until transferred into the computing store by programme by means of 76-orders (see section 11.4). In the same way, when cards are being punched, the data to be punched is sent to a buffer store where it is held until the card punch is ready to punch the card, when the information passes by way of conversion circuits to the punch. The two buffer stores associated with the card reader and card punch are quite separate; each consists of two blocks of eight words. Each word is divided into 6 six-bit characters numbered as shown in the diagram; the sign-bit and the two most significant bits of each word are usually clear. Thus there are 96 character positions in each card data buffer which can be used for the storage of the data contained in one card. Facilities are provided for automatic code conversion,

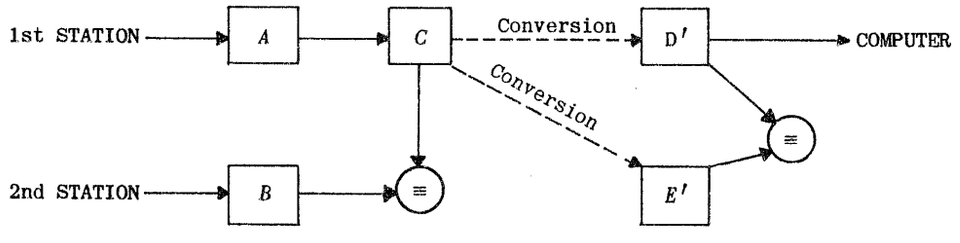
...	Ch. 1	Ch. 2	Ch. 3	Ch. 4	Ch. 5	Ch. 6

using virtually any code, there being up to 32 different modes of conversion. Usually, each column is converted into one six-bit character, there are, however, also modes of conversion by which the punching in the upper curtate (see section 11.2.1) is converted into one character and the lower curtate is converted into another character (this can be done using either the 2 + 10 or the 3 + 9 column split) or a twelve-bit copy of the column can be obtained.

The relation between the layout of columns on the card and the position of characters within the data buffer is determined by the contents of the Card Distribution Table, which specifies, for each of the 96 character positions in the card data buffer, the card column from which the character is to be obtained on input. The type of conversion which is to be used to obtain the characters from the punching in the columns is determined by the contents of the Interpretation Table, which specifies, for each of the 96 character positions, the mode of conversion which should be used. The Card Distribution Table and the Interpretation Table are combined together to form four blocks of table. There are separate tables for the reader and the punch. These are stored in buffer stores and are formed and loaded by programme (see section 11.6). The transfer of the tables from the computing store to the buffers is done by means of 76-orders (see section 11.4).

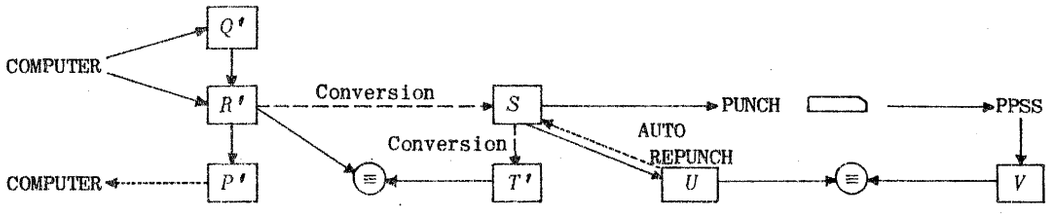
Although the actual conversion from the holes in the columns to the characters in the data buffer is carried out by electronic circuits this is controlled by a Code Table specified by the programmer. (Input and Output use the same table.) This table can be visualised as having entries specifying, for each permissible punching in a code, the corresponding six-bit character, so that the programmer has complete freedom of choice in the computer code used. The table has sufficient capacity to hold two different codes at any one time and is filled from the computer under programme control.

A block diagram of the reading system is shown below:-



The card reader has two reading stations. The data from the first reading station are assembled in card data buffer *A*, according to the distribution table, in a code determined by the mode of interpretation being used. After assembly in *A* is complete the data are transferred to buffer *C* as soon as *C* is free. The card is read again at the second reading station and the data assembled in buffer *B*. (Meanwhile the following card is being read into *A*). Conversion takes place as soon as *C* is filled and *D'* is free: it may take place before or after check reading, depending on the relative speeds of the programme and the card reader. By means of one set of conversion circuits the card data are transferred to card buffer *D'* (the prime denotes the fact that the data are now in computer code). By a second set of conversion circuits the data are transferred to *E'*. When assembly in *B* is complete the equivalence of *B* and *C* is tested. When conversion is complete the equivalence of *D'* and *E'* is tested. If these tests succeed the data from *D'* are allowed to be transferred to the computing store by 76-orders (see section 11.4). If there is a failure, an indication is set in a fault register and the card may be offset in the stacker (see section 11.7.2).

The punch also has two stations, a punching station and a reading station (the post-punch sensing station or P.P.S.S.). A block diagram of the punch system is shown below:-



Data from the computer are loaded from the computing store into card buffer *R'* by means of 76-orders. If *R'* is already loaded and the contents are awaiting punching the same 76-orders will transfer the data into card buffer *Q'*. If both *R'* and *Q'* are already loaded the computer will be held up until *R'* has been emptied and the contents of *Q'* transferred to *R'*. As soon as *R'* is loaded, conversion into card code takes place and a converted image is assembled in *S*. Check-conversion now proceeds, giving a card image in computer code in buffer *T'*. The contents of buffers *R'* and *T'* are tested for equivalence. The data from *S* are punched into the card and also transferred to a buffer *U*. When the card is read at the post-punch sensing station, a card image is assembled in buffer *V*. When complete, the equivalence of this and the buffer *U* is checked. Note that this is a check only on the information which has been sent to the punch and there is no check on other columns and curtates of the card, which could contain pre-punched information. Errors cause the card to be offset in the stacker and the card to be re-punched, but if desired the repunching may be suppressed and replaced by an immediate fault indication. Since by the time the punched card has been read at the post-punch sensing station the next card will also have been punched, this too is offset and will be repunched after the first card has been repunched. Repeated errors cause a lockout and an indication to be set in a fault register (see section 11.7.3). When the contents of buffers *R'* and *T'* have passed the equivalence test the contents of *R'* are transferred to *P'* (and the contents of *Q'* to *R'*) so that if there is a punch failure, the contents of the card buffer *P'* may be recovered by the computer. When this is transferred to the computer *R'* is transferred to *P'* and *Q'* to *R'* so that all the card information waiting to be punched may be recovered in the correct order by transferring the contents of buffer *P'* (the card output recovery buffer) to the computer three times.

11.2 Card Usage

11.2.1 *Introduction*

The only types of punched cards which we shall consider here, though there are others, are those with 12 rows and either 80 or 65 columns. The notation adopted by Ferranti for the positions in the card column is based on the usage of the new British Standard card code. Its relation to the other notations is shown in Table 11.1.

Ferranti, I.C.T., British Standard usage	British Standard (official)	Hollerith	Powers-Samas	I.B.M., Bull
10	24	Y	A	12
11	22	X	B	11
0	20	0	0	0
1	18	1	1	1
2	16	2	2	2
3	14	3	3	3
4	12	4	4	4
5	10	5	5	5
6	8	6	6	6
7	6	7	7	7
8	4	8	8	8
9	2	9	9	9

Table 11.1 *Card Column Notations*

A card column is often considered as being made up of two parts, allowing a difference of treatment of the two parts. These parts are called the *upper-curtate* (the upper part) and the *lower-curtate* (the lower part). There are at present, in this country, only two ways in use of splitting a column into upper and lower curtate. In one of these ways, the division is between the top two positions and the bottom ten: this is called a *2 + 10 column split*. In the other way the division is between the top three positions and the bottom nine: this is called a *3 + 9 column split*.

Usually a character may have a single hole in each curtate or in only one; the space character has no holes. Except in certain special codes, there is never more than one hole punched in either curtate. Table 11.2 gives the codes in common usage in this country.

Column Punching	3-Zone Codes		BULL	4-Zone Codes		IBM	Hollerith 5-Zone and Hollerith New 4-Zone
	Hollerith 3-Zone	Powers 32-ch.		Hollerith 4-Zone	Powers 39-ch.		
- -	Space	Space	Space	Space	Space	Space	Space
- 0	0 0(11.2.2)	0 0(11.2.2)	0 0(11.2.2)	See "0-"	See "0-"	See "0-"	See "0-"
- 1	I 1	I 1	I 1	1	1	1	1
- 2	2	2	2	2	2	2	2
- 3	3	3	3	3	3	3	3
- 4	4	4	4	4	4	4	4
- 5	5	5	5	5	5	5	5
- 6	G 6	G 6	6	6	6	6	6
- 7	7	7	7	7	7	7	7
- 8	S 8	S 8	8	8	8	8	8
- 9	9	9	9	9	9	9	9
10 -	Z (11.2.2)	A(11.2.2)	(11.2.2)(11.2.3)	(11.2.2)	(11.2.2)	(11.2.2)	(11.2.2)
10 0	N	C		NONE	NONE	NONE	NONE
10 1	P	D		A	A	A	A
10 2	Q	E		D	B	B	B
10 3	R	F	NONE	G	C	C	C
10 4	T	H		J	D	D	D
10 5	U	J		M	E	E	E
10 6	V	K		P	F	F	F
10 7	W	L		S	G	G	G
10 8	X	M		V	H	H	H
10 9	Y (11.2.2)	N		Y	I	I	I
11 -	A	B (11.2.2)	(11.2.2)	(11.2.2)	(11.2.2)	(11.2.2)	(11.2.2)
11 0	B	P		NONE	NONE	NONE	NONE
11 1	C	Q		B	J	J	J
11 2	D	R		E	K	K	K
11 3	E	T		H	L	L	L
11 4	F	U		K	M	M	M
11 5	H	V	NONE	N	N	N	N
11 6	J	W	But see	Q	0	0	0
11 7	K	X	(Table 11.4)	T	P	P	P
11 8	L	Y		W	Q	Q	Q
11 9	M	Z		Z	R	R	R
0 -				(11.2.2)	(11.2.2)	(11.2.2)	(11.2.2)
0 1				C	S	/	&
0 2				F	T	S	S
0 3				I	U	T	T
0 4				L	V	U	U
0 5	NONE	NONE		0	W	V	V
0 6				R	X	W	W
0 7				U	Y	X	X
0 8				X	Z	Y	Y
0 9				&	(11.2.2)	Z	Z

Table 11.2 *Codes in common usage*

Each column of Table 11.2 gives the standard interpretation, in the code specified, of the punchings listed in the left-hand column. In some of the codes a punching can represent both a letter and a number, in which case both are given. In places where the situation cannot be summarised within the table reference is made to these notes.

The left-hand column of the table indicates the column punchings. In each case the punching in the upper curtate is given first, followed by that in the lower curtate. No punching in a curtate is indicated by "-". Thus, "-7", means that a hole in the 7-position only is punched, and "10 3" means that holes in the 10-position and the 3-position are punched. To avoid confusion the letter 0 is written "0".

Table 11.2 is arranged to show how far the different codes correspond; for instance, the decimal digits 1 to 9 are the same in all codes. Codes are often thought of as being made up of a number of

zones. The *zone* to which a particular punching belongs is determined by the punching of the upper curtate, though this may be extended in some codes to include other rows than those given above in the standard ways of splitting. Thus a 2 + 10 column split gives rise to a *3-zone* code where the zone is determined by the three alternative punchings -, 10, 11 in the upper curtate. A 3 + 9 column split gives rise to a *4-zone* code with the zones determined by the punchings -, 10, 11, 0. Punchings outside the common range are given in Tables 11.4, 11.5 and 11.6.

11.2.2 *Spare punchings*

In the 4-zone codes and the new codes in which the punchings 10 -, 11-, 0 - are spare these punchings can represent special characters. In all these codes, if zero is required to be represented by something other than a blank column it will be represented by 0 -. Otherwise, each computer user or punched card installation operator has freedom to choose what special characters he wishes for each column position, although his freedom is limited by the need to have special type on tabulators wherever special characters are used. His choice will also affect the preparation of the cards and the programmes used for processing the cards in the computer. In the new codes the introduction of a number of special punchings into the code eases the shortage of punchings available for representing special characters.

When a card column is being used for punching pence or months the punchings 10 -, 11 - and 0 - are used, in some permutation, to represent 10, 11, 0 or 10, 11, 12. Actually, when a column is being used for this purpose it is strictly not being punched according to an alpha-numeric code. Nevertheless, the preferred ways of representing pence and months for some codes are given in Table 11.3.

Pence	Hollerith 3-zone	Bull	Hollerith Old 4-zone	Powers 39	IBM	Hollerith New 4-zone
0d.	0-	0-	0-	0-	0-	0-
10d.	11-	11-	11-	10-	11-	10-
11d.	10-	10-	10-	11-	10-	11-
Months						
Oct. (10)	11-	0- or 0- / 8 3	11-	10-	11-	10-
Nov. (11)	10-	11- or 8 3	10-	11-	10-	11-
Dec. (12)	0-	10- or 7 2	0-	0-	0-	0-

Table 11.3 *Representation of Pence and Months*

11.2.3 *Bull Code*

The punchings in the Bull code which do not correspond to the standard punchings are given in Table 11.4. In this code zero can be represented either by 0 - or by 10 -. The latter, which is called "Mechanical Zero", is used when zero suppression on printing may be required; the Bull tabulators can be set up so that non-significant zeros are not printed provided they have been punched as 10 -; but this punching is accepted by the accounting mechanism as zero.

Cards punched in the Bull code can be read into Pegasus 2 and automatically converted into 6-bit code provided that they are inverted before being fed into the card reader. This enables the computer to treat rows 9, 8 and 7 as the upper curtate. It also inverts the significance of all the row numbers listed in the code table described in section 11.6.2.

11.2.4 *IBM Code*

The permissible range of punchings in the IBM code extends beyond the usual range of punchings, as it also allows -, 10, 11 or 0 in the upper curtate, one of the digits 2 to 7, together with 8 in the

lower curtate. These punchings together with the two standard allocations of special characters, one for commercial use and one for scientific use are shown in Table 11.6. The most common of the special punchings are the eight listed which contain either a 3 or a 4. Note that the special characters give characters corresponding to 10 - and 11 - (and 0 - represents zero) so that strictly speaking these punchings are not then available for representing pence or months. But the use of the special characters does not imply that particular columns cannot be treated differently.

Overpunching, Control Punching, etc.

When a column is used to hold a decimal digit, the upper two positions (or three if zero is not punched) are not really needed. Sometimes these spare positions are used to hold other data. In the usual IBM and I.C.T. (Hollerith) practice only the 11 position is used for this purpose. It is used particularly to indicate the sign of a number, or 10/-. These two companies refer to the practice as *control-punching*. In Powers-Samas practice, in addition to similar uses of a spare hole position, it is common to use the top three positions to hold a digit 0, 1, 2 or 3 corresponding to punchings -, 10, 11, 0. This latter practice is termed *overpunching*. Sometimes three adjacent upper curtates (of three positions each) are used to hold a decimal digit, i.e. in the first column a hole in one of positions 10, 11, 0 could mean 1, 2, 3, in the second column 4, 5, 6, and in the third column 7, 8, 9, no hole in these positions in any of the columns would mean zero. This practice is termed *block overpunching* or *boxing*.

Field

A set of columns used to hold, say a number, or a name and address, is usually referred to as a *field*. The term is a flexible one, and may be used to refer to any set of columns being considered at the time as a whole. Fields may be given names, such as "Date", "Part Number", "Price", "FI". Thus, for example, the date might be punched as six digits in columns 1 to 6 and these six columns constitute a field. Care has to be taken to distinguish between reference to a field and to its contents (as in the computer one refers to a register as "*X*" and its contents as "*x*").

Interstage Working

Powers-Samas double the amount of data they can store on their 65- and 80-column cards by use of *interstage working*, in which holes are also punched and read half a row below the standard position. They are able to do this because of the small holes. Code conventions of every sort are just the same with interstage working as with normal working.

The following are illustrations of punched cards showing the punchings in each code in a more graphic form.

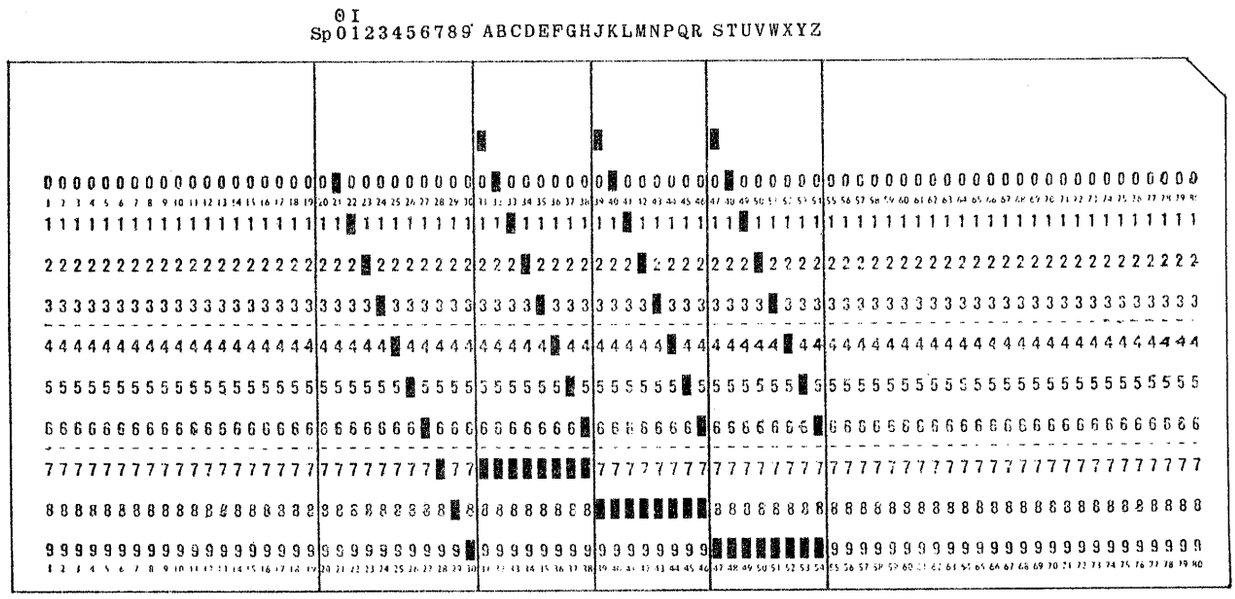


Figure 11.1 *Specimen card - Bull*

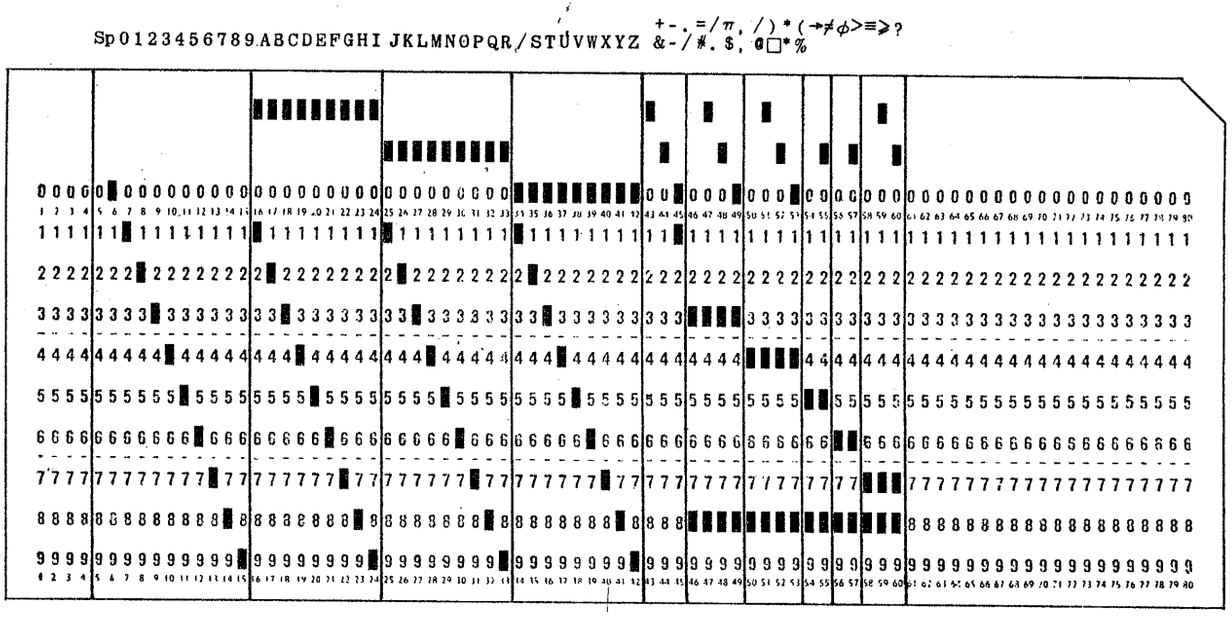


Figure 11.2 *Specimen card - I.B.M.*

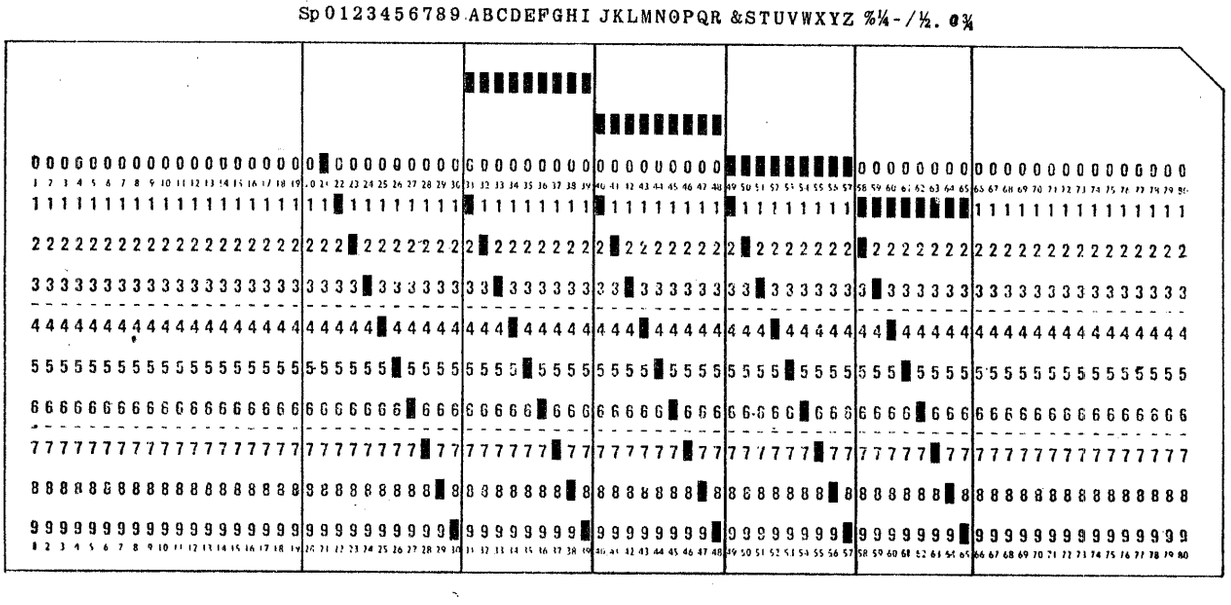


Fig. 11.3 *Specimen card - Hollerith New 5-Zone*

11.3 Handling Six-bit Characters

Before proceeding further with the description of the punched card facilities of Pegasus 2, three functions in the Pegasus 2 order-code, which are specifically designed to assist in the handling of six-bit characters, will be described.

To facilitate the monitoring of information arranged in this form, there is a switch above the left-hand monitor tube which splits the trace into six-bit characters.

11.3.1 *Function 27, Accumulative Multiplication*

(a) General form: $p' = (2X).p + n,$

$q' = 0.$

This form is primarily intended to allow fast assembly from paper tape, the radix being $2X$. Note that q (i.e. x_7), is destroyed.

(b) Special form when $N = 7$: $p' = (2X).p$ + (digits 5 - 8 of q shifted down 30 places).

$q' = 2^6.q$ (logical shift).

This special form automatically converts into binary a numerical quantity held as 6-bit characters in accumulator 7, the most-significant character being at the most-significant end of *X*7. The radix is again 2*X*. The order must be obeyed once for each character in the field being converted.
Note that only the last four bits of each character are used. This is sufficient as the maximum radix it is possible to use is 14. It also ensures that the character from punched cards 010000, which is commonly used for Space, is treated as zero because the least significant four bits are zero.

Time: 2 word-times.

There is no restriction on the sign of the quantities, and a test is made for overflow.
Modification is as for other Group 2 instructions.

Example: To convert a 5-character decimal field in 5.0 into binary. The characters occupy the first 5 positions in 5.0, and the sixth position is not used.

```
5.0  7 00      Characters 1 to 5 to X7
 7 (0)27       Ch.1 to X6 ; (Chs. 2 to 5).2^6 in X7
(4) 5 40       Count 4 more digits
 ┌→7 (5)27 }
 └   5 67  }   Build up binary integer in X6
```

Notes: (1) The *X* digit in the 27 order is ringed as it is a number (half the radix, 10) and not an address.
(2) It is sometimes better, especially with short fields, to write out the loop orders and avoid counting.
(3) The answer is in *X*6.

11.3.2 *Function 37, Character Generation*

The effect of this order is to convert an integer held in binary in accumulator 6 into 6-bit characters, and to pack these characters into accumulator 7.

$$p' = \text{Remainder of } \frac{(2X.p)}{n},$$

$$q' = 2^6.q + \text{Quotient of } \frac{(2X.p)}{n}.$$

Example: By a sequence of such orders, successive digits of p in radix $2X$ may be obtained as 6-bit characters in q, using the scale number n. $n = (2X)^m$ if an m-digit number is being formed in radix $2X$.

Limitations: $n > p \geqslant 0$.
There is an "Unassigned Order" stop if p or n is negative.
Overflow is set if $n \leqslant p$.
There is no detection of character overspill in accumulator 7.

Time: 5 word-times.

There is no modification for this order; the *M*-digits are used in connection with zero suppression and have the following effect:

(a) $M = 0, 1, 2, 3$.

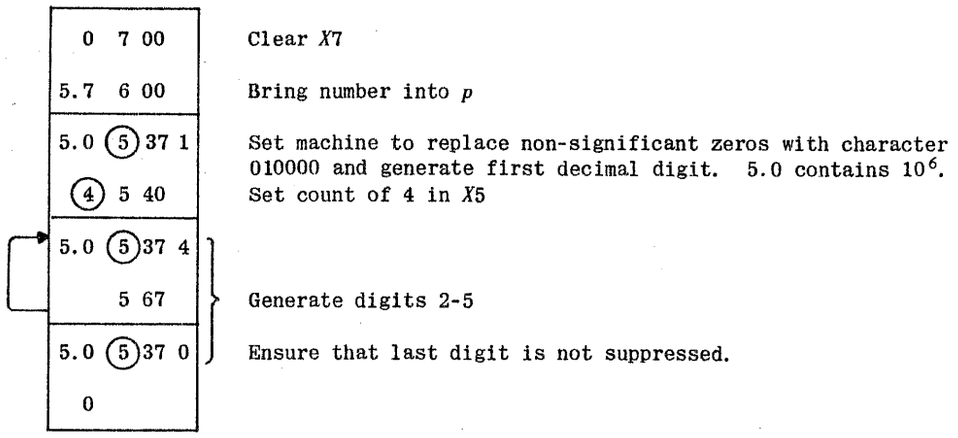
The first effect of a 37-order having $M < 4$ is to set certain circuits which cause zeros to be replaced by the six-bit character specified in the list below; it then generates a character as described above.

M	Coding of symbol
0	00 0000
1	01 0000
2	10 0000
3	11 0000

(b) $M = 4$ (and 5, 6, 7).

A character is generated as above. If zeros only have been generated since the last 37-order with $M < 4$ was obeyed, zero will be replaced by the character specified in that order; but this zero-suppression is inhibited as soon as a non-zero character is generated. The following example should make this clear.

Example: To convert a binary number in 5.7 smaller than 10^6 into 6 decimal characters. Non-significant zeros to be replaced by the six-bit character 010000.



Notes: (1) The six characters are left in *X*7.
(2) As with the 27-order, it may be better to write out the loop.

11.3.3 *Function 57, Character Shift*

This gives shifting by characters in each direction and also enables a selected field of 6-bit characters to be placed at the least-significant end of an accumulator with one order.

The order is written as:

$$(l.r)\ X\ 57\ M$$

where l = the number of 6-bit shifts up,
r = the number of 6-bit shifts down,
X = the accumulator shifted.

When both l and r are non-zero the shift up occurs first, followed by the shift down. If a null shift (0.0) is specified there is no change in the contents of the accumulator. Otherwise only digits 3-38 are used in the shifting and digits 0-2 are left zero.

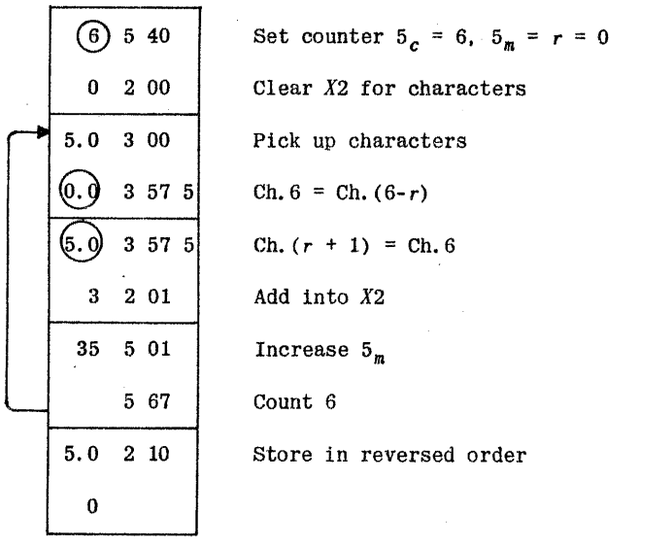
Modification is as for the rest of Group 5 orders, and modifiers must be constructed to treat l as block and r as a unit.

Time: $l + r + 2$ word-times.

Notes: The quantity N (i.e. $l.r$) is treated mod (64) e.g. (10) X 57 is treated as (1.2) X 57.

The following example using modified 57-orders should make their use with modifiers clear:-

Example: To reverse the order of six 6-bit characters in 5.0.



11.4 Transfers of Data to and from Buffer Stores - The 76-Order

With every major item of optional peripheral equipment for Pegasus 2 supplementary buffer stores are provided. The flow of data between the data buffer and the associated peripheral machine is at a rate determined by that machine. Once the computer has initiated the operation it is free to do other work; the transfer is therefore said to be "autonomous".

The 76-order should be regarded as a "buffer transfer" order. The N-address is used to specify not only the data buffer store and the block in that store, but also the direction of the transfer and certain other features of it. The X-address specifies the computing store block concerned (as in 72- and 73-orders). For example, the order

80 [4] 76

causes the content of *U*4 to be transferred to the first block of the output data buffer and the initiation of a punching or printing operation.

The following table gives all the permissible *N*-addresses for use with 76-orders.

N-address		
0 - 3	Tape Control 1	Interchange magnetic tape buffer and computing store.
4 - 7	Tape Control 2	
8 - 11	Tape Control 1	Transfer from magnetic tape buffer to computing store.
12 - 15	Tape Control 2	
16 - 19	Tape Control 1	Transfer from computing store to magnetic tape buffer.
20 - 23	Tape Control 2	
24 - 27	Card Control 1	Transfer from computing store to Code Conversion Table.
28 - 31	Card Control 2	
64,65	Card Reader 1	Transfer from input card data buffer to computing store and then read next card.
66,67	Card Reader 2	
72,73	Card Reader 1	Transfer from input card data buffer to computing store but do not read card.
74,75	Card Reader 2	
80,81	Output 1	Transfer from computing store to output data buffer and then punch card or print line.
82,83	Output 2	
88,89	Output 1	Transfer from computing store to output data buffer but do not punch card or print line.
90,91	Output 2	
96,97	Card Output Recovery 1	Transfer from Output Card Recovery Buffer to computing store.
98,99	Card Output Recovery 2	
104,105	Card Reader 1	After a fault, transfer the result of a not-equivalent ($\neq$) operation between main and check reading stations to computing store.
106,107	Card Reader 2	
112-115	Card Reader 1	Transfer from computing store to distribution table for punched card input.
116-119	Card Reader 2	
120-123	Output 1	Transfer from computing store to to distribution table for output.
124-127	Output 2	

32-63, 68-71, 76-79, 84-87, 92-95, 100-103, 108-111 unallocated.

Notes:- (1) In the above table *N*-addresses are given which cover all possibilities.
(2) The use of 76-orders with unallocated *N*-addresses will in most cases lead to a computer stoppage on an uncompleted 76-order. In a system where there is, for example, only one card reader the use of a 76-order with an *N*-address of 66 will have the same effect as a 76-order with *N*-address 64. However, use should not be made of this fact as it will preclude the use of the programme on a system with two readers.

Timing:- The transfer of data between a buffer store and the computing store for magnetic tape buffers takes 1.25 milliseconds; for the other buffers there is an additional average waiting time of 1 millisecond.

11.4.1 *Modification of 76-order*

76-orders can be modified in such a way as to cycle round the four addresses within one of the groups listed above, without the group or type of transfer being altered by modification. The block part of the modifier is added to the *N*-address modulo 4, always leaving a modified *N*-address within the same group of 4.

Suppose, for example, the modifier in *X*5 is 6.3 when the order 19 [5] 76 5 is obeyed.

N-address	= 19	= 4 × 4 + 3	
Block modifier		= 6	
Adding *N*		= 16 + 9	
Modulo 4		= 16 + 1(+ 2 × 4)	
		= 17	

therefore the modified order is 17 [5] 76.

11.5 Code Conversion and the Code Table

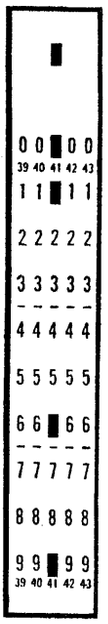
Each column of a card may be regarded as being composed of 12 bits of data. These 12-bits may be treated in three basic ways according to the mode of interpretation selected by the programmer.

(i) The 12 bits may be copied into any two character positions in the input data buffer.
(ii) The 12 bits can be automatically converted into one six-bit character selected by the programmer and placed in any character-position in the input data buffer.
(iii) The 12 bits can be divided into upper and lower curtates and each curtate can be automatically converted to a six-bit character. The two six-bit characters so obtained can be placed in any of the character positions in the input data buffer.

11.5.1 *Six-bit copy*

In this case the 12 binary digits obtained from a column are placed in two character positions in the input data buffer. For example, if it is specified that column 41 shall be copied into the first block of the buffer, word 4, character position 2, and into the second block of the buffer, word 5, character position 3, and if column 41 is punched as shown, then the two character positions above will contain respectively:-

1 0 1 1 0 0 and 0 0 1 0 0 1



This permits, for example:-

(i) Binary working.

(ii) Interpretation of irregularly coded columns by programme.

(iii) Searching for particular holes, e.g. copying the top half of the column and testing for a 1-bit in a particular position (acting as a designation or control hole).

11.5.2 *3-Zone Alpha-Numerical Conversion*

In this case the 12 binary digits obtained from a column are automatically converted to produce one 6-bit character in the computer. The column is assumed to be divided into two curtates by a division between the 11 and 0 rows. This will allow for cards punched in a 2 + 10 code and will deal with cards in Powers 32-character or Hollerith 3-zone alpha-numeric codes (and the 23-character alpha code).

At most one hole may be punched in each curtate. When there is no hole in the lower curtate, two alternative interpretations are permitted for each of the three possible single hole punchings (including no hole) in the upper curtate. There is automatic detection of columns punched with more than one hole in a curtate.

11.5.3 *4-Zone Alpha-Numerical Conversion plus Special Characters in a 5th Zone*

The column in this case is assumed to be divided into two curtates by a division between 0 and 1 rows. At most one hole is permitted in each curtate, except in the case of 5th zone working described below. Each column is converted into one character. This will allow for cards punched in a 3 + 9 code and so will deal with alpha-numeric information in British Standard, I.B.M., Powers-Samas 39-character, Hollerith 4-zone and Bull codes.

When there is no hole in the lower curtate four alternative sets of interpretations are permitted for the four punchings in the upper curtate with not more than one hole (including no hole). For example, one hole in row 10 could mean 10 pence (British Standard), or 11 pence (Hollerith), or minus, or end-of-pack designation. This deals automatically with most practices regarding months, pence, fractions and some special symbols punched in one column. It also permits alternative interpretation of blank columns, e.g. space and zero.

In addition, automatic conversion may be obtained for special symbols, e.g. I.B.M. row-8 code.

11.5.4 *2 + 10 Split-Column working*

Assuming the column to be split by a division between the 11 and 0 rows, the punching in the upper curtate may be converted to one character, and that in the lower curtate converted into another character. Two alternative interpretations are permitted of all the four possible punchings in the upper curtate (including no hole).

Numerical conversion (with not more than one hole) is permitted from the lower curtate (i.e. the same as would be obtained if the whole column were interpreted with the upper curtate blank).

11.5.5 *3 + 9 Split-Column working*

Assuming the column to be split by a division between the 0 and 1 rows, the punching in the upper curtate may be converted to one character and that in the lower curtate converted into another character.

Two alternative sets of interpretations are permitted of all the 8 possible punching combinations in the upper curtate (including no hole).

This conversion is used in general for overpunching to extend a field, for shillings, or for control holes.

The facility to ignore punching in row 11 ("B-ignore" or "X-ignore") is provided by giving the same meaning in the code conversion table to appropriate pairs of punchings, e.g. a pair of holes in rows 10 and 11 would be given the same meaning as one hole in row 10.

Normal numerical interpretation is given to a single hole in the lower curtate (i.e. the same as would be obtained if the whole column were interpreted with the upper curtate blank). There is provision for giving any required meaning inside the computer to no hole in the lower curtate, e.g. zero.

11.5.6 *Code Table*

The conversions from the holes in the columns to the characters in the data buffer are carried out by electronic circuits controlled by a Code Table. (Input and output use the same table.) This can be visualised as a table with many double-entries, each specifying the relation between the holes in a card-column and the corresponding 6-bit computer-character. The table can be loaded (by programme) in any way required, subject to certain small restrictions.

The Code Table is constructed so that two complete codes are loaded at one time which means that different fields in a card may be interpreted using different codes. These two codes are referred to as the Main and Second Code.

There are three ways in which the code tables may be loaded according to the extent of the Main or Second Code:

(i) Extensive Main Code including I.B.M. 5th Zone.

(ii) Limited Main Code, which includes some of I.B.M. 5th Zone punchings but not all. The Second Code is more extensive than in (i) and includes one set of "Upper Curtate Split" interpretations.

(iii) Extensive Second Code, including two sets of "Upper Curtate Split" interpretations.

The code tables are stored in four blocks of buffer store and comprise three blocks of tables X, Y, Z and one block of intermediate code. In installations with one card control the intermediate code

has a buffer block address of 24 and tables *X*, *Y* and *Z* have addresses 25, 26 and 27 respectively. These provide a main code and a second code. The second code was originally designed for use with an on-line printer with an error detecting code. This second code has been extended to make it more generally useful with punched cards, however, this has meant the introduction of certain restrictions.

The intermediate code is a code which bears a constant relationship to the punching in a column for a particular mode of conversion. The method by which conversion on input takes place is as follows. As the card is read, a character is formed from each column, the exact character formed depends on both the punching in that column and the mode of interpretation used. The characters are put into positions, determined by the distribution table, in buffer *A* (see diagram on page 243) and are transferred to *C* as soon as the card has been completely read and *C* is free to receive them. Conversion then takes place, the method being to test for coincidence between characters from the card buffer *C* and characters from the intermediate code buffer. When coincidence occurs the corresponding character from one of the code table buffers *X*, *Y* or *Z* is allowed into the card buffer *D'*. The choice of code tables is determined by the contents of the interpretation buffer. When a card is punched conversion takes place in the opposite direction, the method being exactly similar.

The same code tables are used by both the reader and the punch and it is arranged that punch conversion has priority over reader conversion and can also interrupt it. If interruption occurs, the reader conversion begins again when the punch conversion is completed.

11.5.7 *Code Conversion Numbers*

In the Conversion & Distribution Buffer the mode of interpretation used for any particular character is determined by the conversion number. Because the code conversion numbers can refer to different parts of the code conversion table, they make it possible to cope with the situation where the same punching in a column is to have different meanings within the computer, depending on the card field in which it occurs, and conversely where one code in the computer has to be obtained from different punching in the card.

In the figures 11.4 and 11.5, the punchings which can be recognised in each mode of conversion are shown against the conversion number. As an example, if the user specifies that a particular card column has to be converted with code conversion number 0, the equipment will treat this as a "basic" 4-Zone alpha-numeric column; it will refer to the following parts of the code conversion table:

(i) the first of the interpretations for only a single hole punched in the upper curtate (for example for 10, 11 and zero) and for blank column.
(ii) The decimal numbers 1 to 9.
(iii) The three alphabetic zones.

The various code conversion numbers give the following types of conversion:

For the first code:

0, 2, 4, 6	Decimal numbers 1 - 9 and letters of any 4-zone alpha-numeric code, with one of four alternative interpretations of a single hole punched in the upper curtate and of blank column.
16, 18, 20, 22	As 0, 2, 4, 6 but, in addition, interpretation of the special symbols of I.B.M. type; alternatively, the equipment can be supplied to interpret the 5th Zone of the ICT type, as shown in the second card on page 248.
1, 5	Two alternative interpretations of any punching in the upper curtate with 3 + 9 split column working, including no-hole.
3	Decimal numbers 1 - 9, or no hole, in the lower curtate with 3 + 9 split column working.

For the second code:

If loaded for a 4-zone code, with a 3 + 9 split:

8, 10	Decimal number 1 - 9 and letters of any 4-zone alpha-numeric code, with one of two alternative interpretations of a single-hole punched in the upper curtate and of a blank column.
9, 13	Two alternative interpretations of any punching in the upper curtate with 3 + 9 split column working, including no-hole.
11	Decimal numbers 1 - 9, or no-hole, in the lower curtate with 3 + 9 split column working.

Alternatively, the table may be loaded with the second code of a 3-zone type, with 2 + 10 split column working. In this case the code conversion numbers give:

8, 10	Decimal numbers 0 - 9 and letters of any 3-zone alpha-numeric code, with choice of two alternative interpretations of a single-hole punched in the upper curtate and of blank column.
9, 13	Two alternative interpretations of any punching in the upper curtate with 2 + 10 split column working including no-hole.
11	Decimal numbers 0 - 9, and no-hole, in the lower curtate with 2 + 10 split column working.

For 6-bit copy:

24, 26	Direct copy of any holes punched in the upper or lower halves of the column respectively.

The 96 6-bit characters in the data buffer may be derived in any combination from the columns in the card.

Examples are:

(i) Conversion of all 80 columns to 6-bit characters.

(ii) Conversion of all 80 columns to 6-bit characters, and direct 6-bit copying of the upper halves of 16 columns.

(iii) Conversion of 64 columns to single 6-bit characters and conversion of 16 columns to pairs of 6-bit characters (split column working).

(iv) Conversion of 64 columns to 6-bit characters and direct 12-bit copying of 16 columns.

(v) Direct 12-bit copying of 48 columns.

For cards all of whose columns have to be copied as 12 bits, only 48 columns can be read on one passage of the card through the reader. The card has to be passed twice to read all the columns.

MAIN CODE

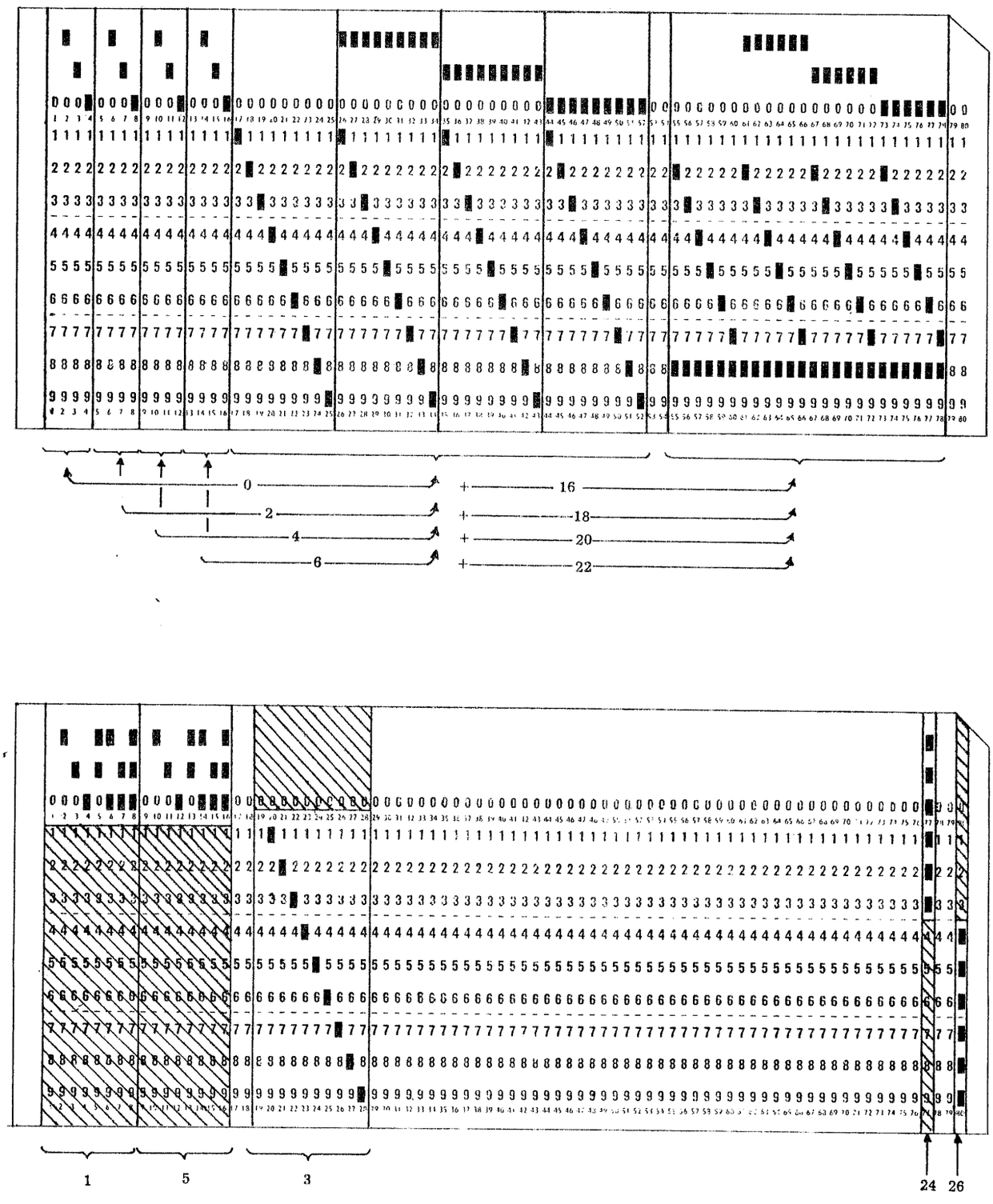
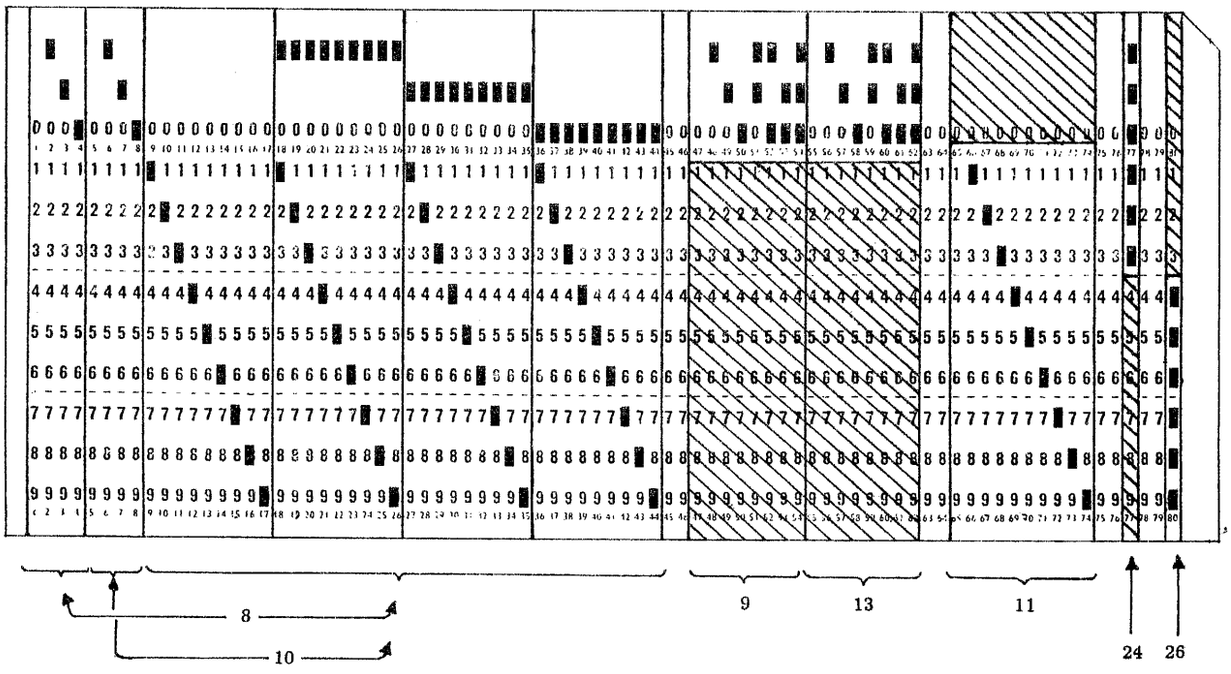


Fig. 11.4

SECOND CODE
4-ZONE CODE WITH 3+9 SPLIT



ALTERNATIVE LOADING
3-ZONE CODE WITH 2+10 SPLIT

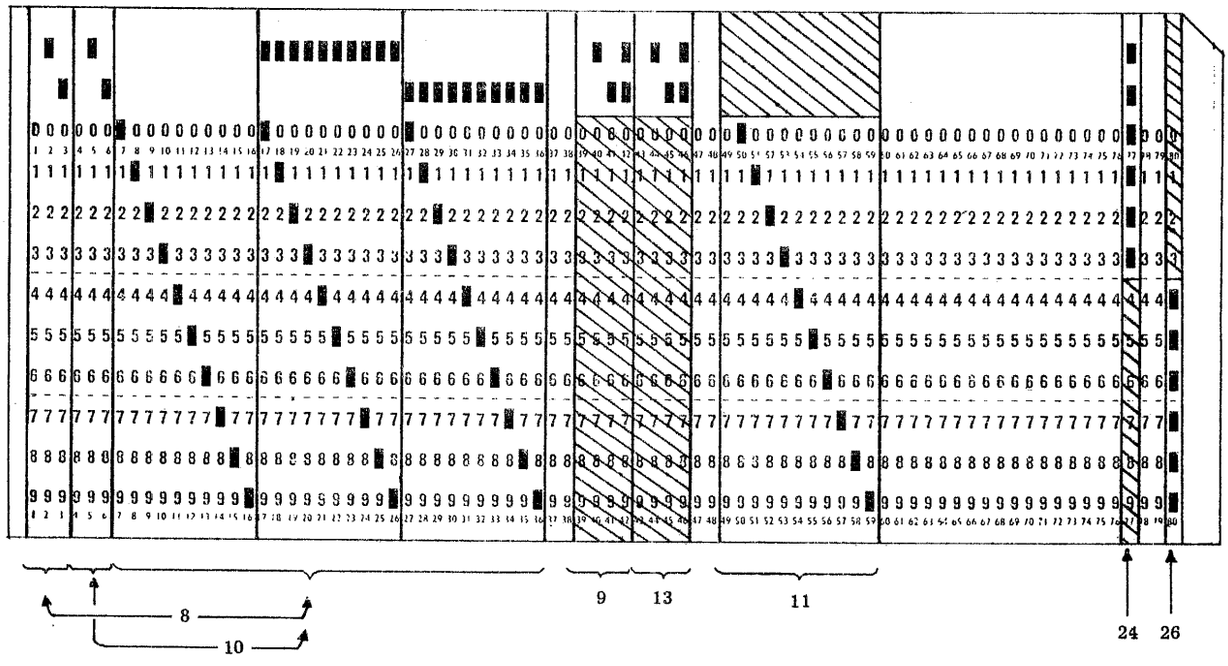


Fig. 11.5

11.6 Loading the Card Control Buffers

11.6.1 *The Distribution and Interpretation Buffer*

For each character position in the card input and output buffers, the column of the card to which it refers and the mode of interpretation to be used is specified in the Distribution and Interpretation Buffer. There is a separate buffer for input and output. These buffers each consist of four blocks of eight words; for engineering reasons the form in which the specifying information is stored is not straightforward. Because of this, a routine has been provided in the isolated part of the main store for loading these buffers. This routine reads the information from paper tape, packs it in the required form, and stores it in the main store in four blocks starting at the address specified by the Transfer Address when the routine was entered (this must be set at the beginning of a block). On exit, the Transfer Address will have been advanced by four blocks. There are three entries† to the routine:-

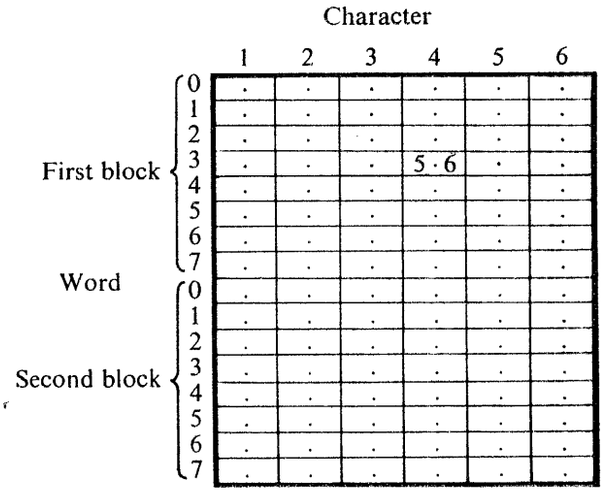
J	1001.0	Store the information in the Main Store and load it into the Input Buffer.
J	1001.1	Store the information in the Main Store and load it into the Output Buffer.
J	1001.2	Store the information in the Main Store only.

The information to be stored in the buffers is punched as 96 numbers of the form *c.m*, where *c* is the column number and *m* is the mode of interpretation. These are punched in order, starting with word 0 character position 1, then character position 2 and so on, to word 15 character position 6. Specifying "column 0" will give, on input, the same effect as reading a blank column; on output, the character is ignored, i.e. it is not punched and checking is inhibited. It should be remembered in this connection that, on input, where column 0 is specified, the character which arises will be exactly that which comes from a blank column using the interpretation given. If blank column has been set in the Code Table to give a six-bit character other than zero then this is the character which will arise from column zero. In this case, if zero is required in such character positions the six-bit copy mode of interpretation should be used and 0.24 punched. However, for output, 0.0 should be specified.

In punching these numbers the usual punching conventions for mixed numbers must be obeyed, each number should be terminated by Sp or CRLF and no signs should be punched. The set of numbers should be terminated by an asterisk (*). Failure to observe the punching conventions will lead to a loop stop in 2.7+.

This routine may also be used as a subroutine, entry being made by bringing *B*1001 to *U*0 and jumping to the appropriate entry. Before entry *U*5.7 should be set to *B*.0, where *B* is the first block in which the table is to be stored on the drum. The routine uses all the accumulators, *U*0, 1, 2, 3, 4 and 5.7. The link, which should be in *X*1 on entry, is obeyed in 1.3 and left in X1.

An example of a data tape for this routine is shown on the right below.




```
T20.0
J1001.1
 1.0   2.0   7.0   8.0   9.0  10.0
11.0  12.0  13.0  14.0  15.0  16.0
17.3  18.0  19.0  20.0  21.5  21.3
22.0  23.0  24.0  25.3  26.0  27.0
28.0  29.0  30.0  31.0  32.0  33.0
34.0  35.0  36.3  37.0  38.0  39.0
40.0  41.0   0.0   0.0  56.5  57.5
53.0  54.0  55.0  56.3  57.3  58.0
59.0  60.0  61.0  62.3  63.3  64.0
65.0  66.0  62.5  63.5   0.0   0.0
67.5  67.3  68.0  69.3  70.0  71.0
72.3  73.0  74.0  72.5  75.5   0.0
75.3  76.0  77.0  78.0  79.0  80.0
17.5   0.0   0.0   0.0   0.0   0.0
 0.0  48.8  49.8 50.10 51.10 52.10
42.24 42.26 43.24 43.26 44.24 44.26
*
```

Fig.11.6 *Layout of table and example of data tape for distribution and interpretation buffer*

11.6.2 *The Code Table*

This table is constructed so that, for each punching which can be recognized, the six-bit character into which it is to be converted is specified. For each intermediate code character there is one corresponding six-bit character in each of tables *X*, *Y* and *Z*. On pages 259 to 264 tables are given which show the punchings which correspond to the intermediate code characters for each of the three possible ways of loading the code tables. When a code conversion table is being designed for a particular programme the appropriate table should be completed by inserting the decimal values of the six-bit characters required in the spaces provided. (Copies of these tables may be obtained from Ferranti Ltd.) Where a punching does not apply Erase (63) should be inserted. A routine has been provided in the isolated part of the main store which will read a paper tape punched from these tables by rows, only the intermediate code characters and those inserted by the programmer should be punched.

† For machines with two card controls there are two further entries to this routine, J1001.3 to load the buffer associated with the second card reader and J 1001.4 to load the buffer associated with the second card punch. These entries lead to a loop stop in 0.3 on machines with only one card control.

On output the computer searches down an appropriate region of the code table until it first finds coincidence between a code table entry and the given six-bit character in the output data buffer. In some cases the region searched includes punchings not required by the specified conversion number, and if the code table is being used for output its rows may have to be permuted to satisfy the following rules:

1. Occasionally it will be found that in one mode of conversion the same six-bit character will arise from more than one punching on input. When the same table is also being used for output, only one punching can be produced for a given six-bit character in one mode of conversion - the one nearest the top of the table. If this is not the desired punching the complete row containing the desired punching should be moved up so that it appears above the unwanted punching, the remainder of the table being left unchanged.
2. Intermediate codes 8 to 15 inclusive (the top 8 rows in the specimen table) must always appear amongst the first 16 rows of the table.
3. When the main code is being used for output there is a special rule which applies to the six-bit computer character for the main code LC- punching, in table X opposite intermediate code 0. This must appear lower than other main code punchings which have the same six-bit computer character in table X or Y. This rule does not apply to the -/- punchings in table X opposite intermediate codes 1, 9, 17 and 25, or to the upper curtate split punchings in table Y opposite odd numbered intermediate codes.
4. If the main code is being used for output there must not be a second code character in table Y which is the same as a main code character in table X or Y and which appears before it in the table (or in the same row). In a type 3 table, for example, this means that the rows corresponding to intermediate codes 32 to 62 must be permuted if a six-bit character in table X is the same as an earlier one in table Y. Care must also be taken if these rows are moved up to obtain a preference punching.
5. If it were required to use the main code to input 5-zone codes and also to output 4-zone codes (using conversion numbers 0, 2, 4, 6) it would be necessary to ensure that each 4-zone code came higher in table X and Y than 5-zone codes with the same six-bit computer character.
6. In the second code there are two possible interpretations of a blank column or a single hole in the upper curtate. In using these for output, it is not permissible to have the same six-bit computer character corresponding to different punchings in the two interpretations. If one six-bit character is required to give the same punching in both interpretations it must be entered in the code table against the punching reserved by conversion number 8; the corresponding entry for conversion number 10 should be 63, or alternatively should be moved below conversion number 8 in the code table. Both methods permit the correct operation of conversion numbers 8 and 10 for output, but the first alternative prevents conversion number 10 being used for input.
7. If conversion number 3 is being used for output, each punching used by it must be above other main code punchings which have the same six-bit computer character in table X or Y opposite even numbered intermediate codes.
8. If conversion number 11 is being used for output each punching used by it must be above any other punching in table Z which has the same six-bit computer character.

The routine for loading the code table is similar to the routine for loading the distribution and interpretation buffer, described in 11.6.1, in that the information is packed and stored in the main store in four blocks, starting at the address specified by the Transfer Address on entry (which must be set at the beginning of a block). There are two† entries to the routine:-

J 1006.0	Store the information in the Main Store and load it into the Code Buffer.
J 1006.1	Store the information in the Main Store only.

In loading the code table there is a choice between a 3-zone and a 4-zone second code. In order to distinguish between these two cases the following indicator must be punched after the Transfer Address has been set and before the entry is made to the routine:-

-1.0	for 3-zone second code.
+0	for 4-zone second code (or 4-zone printer, section 11.9).

This indicator will be read in and placed in the first word of the tables in the main store by Initial Orders.

The routine may be used as a subroutine in the same way as the routine for loading the Distribution and Interpretation Buffer with the following differences:-

1. The indicator given above must be placed in main store location *B*.0, where *B* is the first block in which the table is to be stored in the main store.
2. The modifier of *U*5.7 must be set to *B*.1.

A failure to observe the punching conventions will lead to a loop stop in 2.7+.

† For machines with two card controls there is one further entry to this routine, J 1006.2 to load the code conversion table associated with the second card channel. This entry leads to a loop stop in 0.2 on machines with only one card control.

The code table enables the programmer to convert card punchings to and from any internal 6-bit code. If there is no special reason to prefer some other 6-bit code it is suggested that the code of directive K should be used. This is given in section 7.3 and in CS 265.

An example of a code table data tape is given below:

```
T 40.0
+0
J 1006.1

8 1 9 1
10 63 63 63
12 63 63 63
14 63 63 63
9 0 0 9
11 32 11 63
13 16 10 63
15 0 63 63
16 2 63 2
18 63 63 63
20 63 63 63
22 63 63 63
24 3 63 3
26 63 63 63
28 63 63 19
30 63 63 63
32 4 63 4
34 63 63 63
36 63 63 63
38 63 63 63
40 5 63 5
42 63 63 63
44 63 63 63
46 63 63 63
48 6 63 6
50 63 63 63
52 63 63 63
54 63 63 63
56 7 63 7
58 63 63 63
60 63 63 63
62 63 63 63
1 32 0 8
3 11 11 63
5 10 10 63
7 0 0 63
17 63 0 63
19 63 32 63
21 63 16 63
23 63 63 63
25 63 0 48
27 63 32 32
29 63 16 16
31 63 63 0
0 0 8 48
2 63 63 11
4 63 63 10
6 63 63 0
*

Z
```

CODE CONVERSION TABLES

Type 1

Extensive Main Code, including I.B.M. 5th Zone; simple Second Code with no split-column working.

* Denotes 3-zone second code only, write 63 for 4-zone.

** Denotes 4-zone second code only, write 63 for 3-zone.

Where there are alternative interpretations of a single hole punched in the upper curtate or of an upper curtate punching using split-column working, the conversion number is given in brackets after the punching. The conversion numbers which can be used with this loading are as follows:- 0,2,4,6; 16,18, 20,22; 1,5,3; 8,10; 24,26. Conversion numbers 9,11 and 13 are excluded.

Intermediate Code (Buffer Address 24)	Table X Main Code (Buffer Address 25)		Table Y Main Code Only (Buffer Address 26)		Table Z Second Code (Buffer Address 27)	
	Punching UC/LC	6-bit char.	Punching UC/LC	6-bit char.	Punching UC/LC	6-bit char.
8	-/1		-/9		-/1	
10	10/1		10/9		10/1	
12	11/1		11/9		11/1	
14	0/1		0/9		0/1 **	
9	-/- (2 or 18)		UC -/-/0 (1)		-/9	
11	10/- (2 or 18)		UC 10/-/0 (1)		10/9	
13	11/- (2 or 18)		UC -/11/0 (1)		11/9	
15	0/- (2 or 18)		UC 10/11/0 (1)		0/9 **	
16	-/2		-/2/8		-/2	
18	10/2		10/2/8		10/2	
20	11/2		11/2/8		11/2	
22	0/2		0/2/8		0/2 **	
24	-/3		-/3/8		-/3	
26	10/3		10/3/8		10/3	
28	11/3		11/3/8		11/3	
30	0/3		0/3/8		0/3 **	
32	-/4		-/4/8		-/4	
34	10/4		10/4/8		10/4	
36	11/4		11/4/8		11/4	
38	0/4		0/4/8		0/4 **	
40	-/5		-/5/8		-/5	
42	10/5		10/5/8		10/5	
44	11/5		11/5/8		11/5	
46	0/5		0/5/8		0/5 **	
48	-/6		-/6/8		-/6	
50	10/6		10/6/8		10/6	
52	11/6		11/6/8		11/6	
54	0/6		0/6/8		0/6 **	
56	-/7		-/7/8		-/7	
58	10/7		10/7/8		10/7	
60	11/7		11/7/8		11/7	
62	0/7		0/7/8		0/7 **	
1	-/- (0 or 16)		UC -/-/- (1)		-/8	
3	10/- (0 or 16)		UC 10/-/- (1)		10/8	
5	11/- (0 or 16)		UC -/11/- (1)		11/8	
7	0/- (0 or 16)		UC 10/11/- (1)		0/8 **	
17	-/- (4 or 20)		UC -/-/- (5)		-/0 *	
19	10/- (4 or 20)		UC 10/-/- (5)		10/0 *	
21	11/- (4 or 20)		UC -/11/- (5)		11/0 *	
23	0/- (4 or 20)		UC 10/11/- (5)		NOT USED	63
25	-/- (6 or 22)		UC -/-/0 (5)		-/- (10)	
27	10/- (6 or 22)		UC 10/-/0 (5)		10/- (10)	
29	11/- (6 or 22)		UC -/11/0 (5)		11/- (10)	
31	0/- (6 or 22)		UC 10/11/0 (5)		0/- ** (10)	
0	LC-		-/8		-/- (8)	
2	NOT USED	63	10/8		10/- (8)	
4	NOT USED	63	11/8		11/- (8)	
6	NOT USED	63	0/8		0/- ** (8)	

Table 11.7 *Code Conversion Table Type 1*

Type 2

Limited Main Code, which includes some of I.B.M. 5th Zone punchings, but not all. The second code is more extensive than in Type 1, and includes one set of "upper curtate split" interpretations.

* Denotes 3-zone second code only, write 63 for 4-zone.

** Denotes 4-zone second code only, write 63 for 3-zone.

In table Y where neither M nor S appears against a non-special punching, Main Code is understood.

Where there are alternative interpretations of a single hole punched in the upper curtate or of an upper curtate punching using split-column working, the conversion number is given in brackets after the punching. The conversion numbers which can be used with this loading are as follows:- 0,2,4,6; 16,18,20,22; 1,5,3; 8,10; 11,13; 24,26. Conversion number 9 is excluded.

Intermediate Code (Buffer Address 24)	Table X Main Code (Buffer Address 25)		Table Y Main/Second Code (Buffer Address 26)		Table Z Second Code (Buffer Address 27)	
	Punching UC/LC	6-bit char.	Punching UC/LC	6-bit char.	Punching UC/LC	6-bit char.
8	-/1		-/9		-/1	
10	10/1		10/9		10/1	
12	11/1		11/9		11/1	
14	0/1		0/9		0/1 **	
9	-/- (2 or 18)		UC -/-/0 M (1)		-/9	
11	10/- (2 or 18)		UC 10/-/0 M (1)		10/9	
13	11/- (2 or 18)		UC -/11/0 M (1)		11/9	
15	0/- (2 or 18)		UC 10/11/0 M (1)		0/9 **	
16	-/2		-/2/8		-/2	
18	10/2		10/2/8		10/2	
20	11/2		11/2/8		11/2	
22	0/2		0/2/8		0/2 **	
24	-/3		-/3/8		-/3	
26	10/3		10/3/8		10/3	
28	11/3		11/3/8		11/3	
30	0/3		0/3/8		0/3 **	
1	-/- (0 or 16)		UC -/-/- M (1)		-/8	
3	10/- (0 or 16)		UC 10/-/- M (1)		10/8	
5	11/- (0 or 16)		UC -/11/- M (1)		11/8	
7	0/- (0 or 16)		UC 10/11/- M (1)		0/8 **	
17	-/- (4 or 20)		UC -/-/- M (5)		-/0 *	
19	10/- (4 or 20)		UC 10/-/- M (5)		10/0 *	
21	11/- (4 or 20)		UC -/11/- M (5)		11/0 *	
23	0/- (4 or 20)		UC 10/11/- M (5)		NOT USED	63
25	-/- (6 or 22)		UC -/-/0 M (5)		-/- (10)	
27	10/- (6 or 22)		UC 10/-/0 M (5)		10/- (10)	
29	11/- (6 or 22)		UC -/11/0 M (5)		11/- (10)	
31	0/- (6 or 22)		UC 10/11/0 M (5)		0/- ** (10)	
0	LC-		-/8		-/- (8), LC-	
2	NOT USED	63	10/8		10/- (8)	
4	NOT USED	63	11/8		11/- (8)	
6	NOT USED	63	0/8		0/- ** (8)	
32	-/4		-/4/8		-/4	
34	10/4		10/4/8		10/4	
36	11/4		11/4/8		11/4	
38	0/4		0/4/8		0/4 **	
40	-/5		-/5/8		-/5	
42	10/5		10/5/8		10/5	
44	11/5		11/5/8		11/5	
46	0/5		0/5/8		0/5 **	
48	-/6		UC -/-/- S (13)		-/6	
50	10/6		UC 10/-/- S (13)		10/6	
52	11/6		UC -/11/- S (13)		11/6	
54	0/6		UC 10/11/- S (13)		0/6 **	
56	-/7		UC -/-/0 S** (13)		-/7	
58	10/7		UC 10/-/0 S**(13)		10/7	
60	11/7		UC -/11/0 S**(13)		11/7	
62	0/7		UC 10/11/0 S**(13)		0/7 **	

Table 11.8 *Code Conversion Table Type 2*

Type 3

Extensive second code, including two sets of "upper curtate split" interpretations in the Second code. No 5th Zone punchings are permitted in the Main Code.

* Denotes 3-zone second code only, write 63 for 4-zone.

** Denotes 4-zone second code only, write 63 for 3-zone.

In Table Y where neither M nor S appears against a non-special punching, Main Code is understood.

Where there are alternative interpretations of a single hole punched in the upper curtate or of an upper curtate punching using split-column working, the conversion number is given in brackets after the punching. The conversion numbers which can be used with this loading are as follows:- 0,2,4,6; 1,5,3; 9,11,13; 8,10; 24,26. Conversion numbers 16,18,20 and 22 are excluded.

Intermediate Code (Buffer Address 24)	Table X Main Code (Buffer Address 25)		Table Y Main/Second Code (Buffer Address 26)		Table Z Second Code (Buffer Address 27)	
	Punching UC/LC	6-bit char.	Punching UC/LC	6-bit char.	Punching UC/LC	6-bit char.
8	-/1		-/9		-/1	
10	10/1		10/9		10/1	
12	11/1		11/9		11/1	
14	0/1		0/9		0/1 **	
9	-/- (2)		UC -/-/0 M (1)		-/9	
11	10/- (2)		UC 10/-/0 M (1)		10/9	
13	11/- (2)		UC -/11/0 M (1)		11/9	
15	0/- (2)		UC 10/11/0 M (1)		0/9 **	
16	-/2		NOT USED	63	-/2	
18	10/2		NOT USED	63	10/2	
20	11/2		NOT USED	63	11/2	
22	0/2		NOT USED	63	0/2 **	
24	-/3		NOT USED	63	-/3	
26	10/3		NOT USED	63	10/3	
28	11/3		NOT USED	63	11/3	
30	0/3		NOT USED	63	0/3 **	
1	-/- (0)		UC -/-/- M (1)		-/8	
3	10/- (0)		UC 10/-/- M (1)		10/8	
5	11/- (0)		UC -/11/- M (1)		11/8	
7	0/- (0)		UC 10/11/- M (1)		0/8 **	
17	-/- (4)		UC -/-/- M (5)		-/0 *	
19	10/- (4)		UC 10/-/- M (5)		10/0 *	
21	11/- (4)		UC -/11/- M (5)		11/0 *	
23	0/- (4)		UC 10/11/- M (5)		NOT USED	63
25	-/- (6)		UC -/-/0 M (5)		-/- (10)	
27	10/- (6)		UC 10/-/0 M (5)		10/- (10)	
29	11/- (6)		UC -/11/0 M (5)		11/- (10)	
31	0/- (6)		UC 10/11/0 M (5)		0/- ** (10)	
0	LC-		-/8		-/- (8), LC-	
2	NOT USED	63	10/8		10/- (8)	
4	NOT USED	63	11/8		11/- (8)	
6	NOT USED	63	0/8		0/- ** (8)	
32	-/4		UC -/-/- S (9)		-/4	
34	10/4		UC 10/-/- S (9)		10/4	
36	11/4		UC -/11/- S (9)		11/4	
38	0/4		UC 10/11/- S (9)		0/4 **	
40	-/5		UC -/-/0 S** (9)		-/5	
42	10/5		UC 10/-/0 S**(9)		10/5	
44	11/5		UC -/11/0 S**(9)		11/5	
46	0/5		UC 10/11/0 S**(9)		0/5 **	
48	-/6		UC -/-/- S (13)		-/6	
50	10/6		UC 10/-/- S (13)		10/6	
52	11/6		UC -/11/- S (13)		11/6	
54	0/6		UC 10/11/- S(13)		0/6 **	
56	-/7		UC -/-/0 S**(13)		-/7	
58	10/7		UC 10/-/0 S**(13)		10/7	
60	11/7		UC -/11/0 S**(13)		11/7	
62	0/7		UC 10/11/0 S**(13)		0/7 **	

Table 11.9 *Code Conversion Table Type 3*

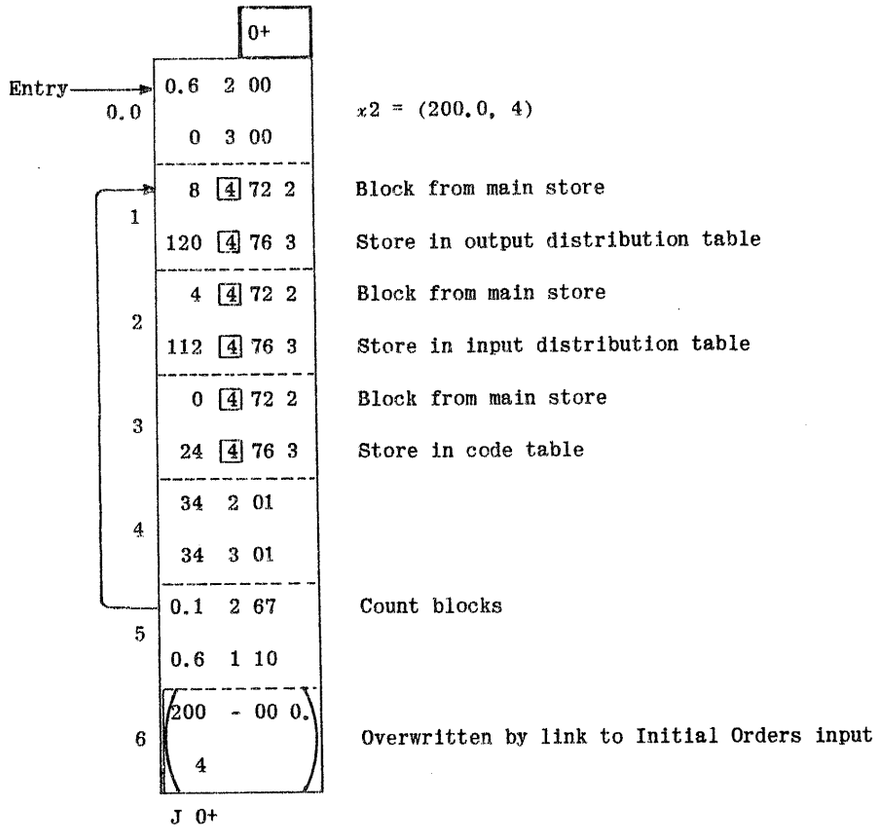
11.7 Programming of Punched Card Operations

In this paragraph a system with one card control is assumed. The information will apply equally to larger installations if appropriate changes are made to the 76-orders.

11.7.1 *Preliminary*

Before either the reader or the punch can be used, the code tables and the Distribution and Interpretation Buffers must be loaded. The routines which have been provided to load the buffers also leave the information in packed form in the main store. (See section 11.6.) This means that the programmer may punch out the contents of the buffers in binary. In this way there are two possible methods of loading the buffers:-

1. Using the standard routines to load the buffers.
2. Using these routines to pack the information once, and obtaining a binary punch of the contents of the buffers. This is then included with the programme tape, with an interlude which takes these blocks from the main store and loads the buffers. Suppose the code tables are stored in *B*200 - 203 inclusive, the reader distribution and interpretation tables in *B*204 - 207 and the punch distribution and interpretation tables in *B*208 - 211. The following interlude will load the buffers and return to initial orders input:-



11.7.2 *Card Input*

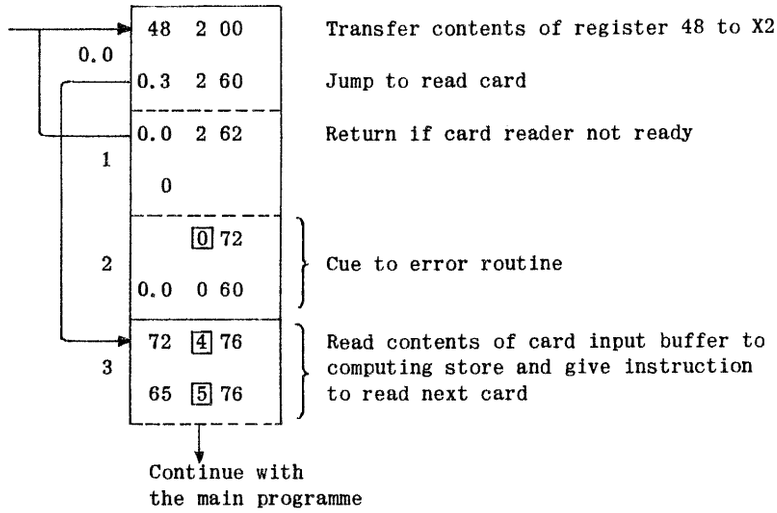
As was mentioned in Section 11.1, there is a fault register associated with the card reader. This register is addressed as *Special Register* 48†. This register is also used to give an indication, by means of the 2^{-1} bit, as to whether or not the contents of the card input buffer are ready to be read into the computing store. When the 2^{-1} bit is equal to 1, the card reader is busy and any attempt which is made to transfer information from the card input buffer to the computing store will cause the computer to be held up until the reading and conversion operation has been completed. This means that, in certain cases, some parallel programming is possible. Special register 48 is tested and, if it is found to contain 2^{-1} some other operation is carried out for a short time and register 48 is tested again later. If, by then, the 2^{-1} bit has been cleared, the programme which has been waiting for the card reader is resumed, otherwise the secondary operation is continued for a further period of time. Usually, however, such parallel programming is not worth the increased programming complications.

The sign bit of special register 48 is used to signal to the programme that a fault has been detected by the checking circuits in a card reading operation. The checking equipment will indicate a fault if the electronic conversion circuits are failing, if there is a difference between the card images obtained at the two reading stations or if double punching has been detected in a curtate where it is not permissible. The detection of double punching automatically makes allowance for cases in which double punching in a curtate is allowed, e.g. where a six-bit copy of half a column is being made. When a fault has been detected, the lockout which prevents the programme from having access to the card input buffer during reading and conversion is not removed. In order that the programme can differentiate between the case where a fault has been detected in the electronic equipment (and therefore immediate engineering action is necessary) and the other two cases where the card may be at fault, provision has been made so that, in the first case, it is impossible to clear special register 48 (that is the sign bit of register 48) by means of the order 48 0 10, whereas, in the other two cases, this is possible. The clearing of register 48 in this way removes the lockout and permits access to the card input buffer by programme.

---

† On machines with two card controls the fault register associated with the second reader is addressed as Special Register 50.

The routine for reading a card might be written as follows:-



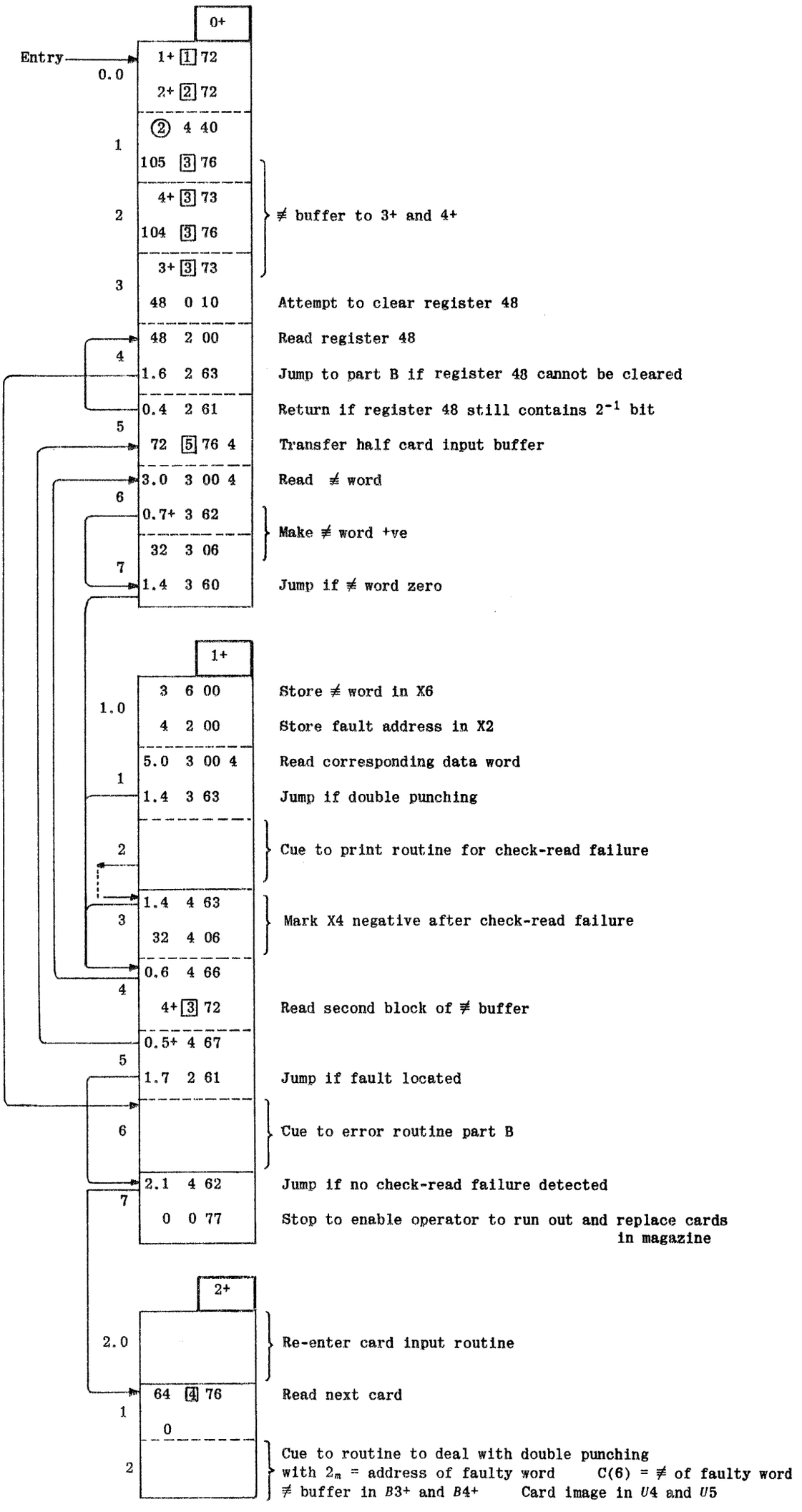
In the above example, no attempt at parallel programming has been made. The repeated reading of special register 48 until it is clear is done so that, should a fault be detected during the reading operation at some time after the contents of special register 48 have been examined the first time, the fault signal will be detected by the programme and the appropriate action will be taken. If this is not done and a fault is detected, the computer will be held up when trying to obey the order 72 4 76 (which it cannot obey because of the lockout) until some manual intervention takes place.

Whenever the card magazine becomes empty or the card stacker full the card reader will become busy, register 48 will always contain 2^{-1} and the appropriate light will be lit on the reader. The programme will therefore stay in a tight loop. The magazine should be replenished or the stacker emptied and the green button on the reader pressed. When this is done, further reading of cards will not be initiated until reference has been made to register 48, and for this reason the programme refers to register 48 before each card transfer; thus there is no need to have a special re-entry point for these occasions. The same thing applies when the reading process is being started. Loading the magazine and pressing the green button will not cause cards to be read until register 48 has been referred to. The reading programme should therefore be entered at 0.0 initially.

Before reading cards, the input data buffer must be clear. When there are no cards in the track and the magazine is empty the buffer may be cleared by pressing simultaneously the green button and the Run Out switch on the card reader. When there are cards in the track, the track and the input buffer may both be cleared by pressing the Run Out switch only.

Should a card wreck occur on the reader, register 48 will contain 2^{-1} permanently and the *card wreck light* on the reader will be lit. Since a card wreck can occur at any point in the card track, it is sometimes not possible to tell which card was last read into the computer. It is therefore advisable to store in the computing store, so that it can be monitored easily, a record of which card was last to be read. The wreck condition can be cleared by removing the damaged card and switching off momentarily the power to the card reader. The procedure is then to determine which card was last to be read, reload the magazine so that the next card is in a position to be read first (it may be necessary to repunch the card) and depress the green button to continue with the programme.

If at any time a fault is detected, register 48 will become negative, the jump in 0.1 will not take place and the error routine will be brought into the computing store. On entry to the error routine the ≠ buffers 104 and 105 are read and an attempt is made to clear register 48. There should be a delay of at least 16 beats between the detection of the fault indicator by programme and an attempt to clear register 48: this is to allow time for further errors to be detected, and for the operation of the device which offsets double-punched or mis-read cards. If register 48 cannot be cleared Part B of the error routine is entered. Otherwise the main error routine examines the contents of the ≠ buffer, which must have been read before register 48 was cleared. Any non-zero character in this ≠ buffer indicates a fault; this must be a check-reading failure unless the corresponding word in the input buffer is negative, indicating a double punching. The computer code character which arises in this case will be the character which corresponds to the punching nearest to the 9-row. The action to be taken after the error has been identified will depend very much on the work being done. A possible situation is that where double punching has been detected the card will be ignored and some indication of this error will be printed so that the card can be corrected later. Where there has been a check-reading failure, the error routine will come to a 77-stop to enable the failing card to be taken out of the stacker, the track emptied by pressing the run out key, and the failing card together with the cards which were in the track, to be replaced in the magazine so that the failing card may be re-read. Where a check-reading failure has occurred, the failing card should be examined for off-punching with a card gauge. An error routine will be available as a library subroutine but some users may wish to write their own, and to illustrate the method a possible way in which the error routine may be programmed is shown on the next page.


0+
Entry
0.0
1+ 1 72
2+ 2 72
1
② 4 40
105 3 76
2
4+ 3 73
104 3 76
≠ buffer to 3+ and 4+
3
3+ 3 73
48 0 10
Attempt to clear register 48
4
48 2 00
Read register 48
1.6 2 63
Jump to part B if register 48 cannot be cleared
0.4 2 61
Return if register 48 still contains 2-1 bit
5
72 5 76 4
Transfer half card input buffer
6
3.0 3 00 4
Read ≠ word
0.7+ 3 62
32 3 06
Make ≠ word +ve
7
1.4 3 60
Jump if ≠ word zero
1+
1.0
3 6 00
Store ≠ word in X6
4 2 00
Store fault address in X2
1
5.0 3 00 4
Read corresponding data word
1.4 3 63
Jump if double punching
2
Cue to print routine for check-read failure
3
1.4 4 63
32 4 06
Mark X4 negative after check-read failure
4
0.6 4 66
4+ 3 72
Read second block of ≠ buffer
0.5+ 4 67
5
1.7 2 61
Jump if fault located
6
Cue to error routine part B
7
2.1 4 62
Jump if no check-read failure detected
0 0 77
Stop to enable operator to run out and replace cards in magazine
2+
2.0
Re-enter card input routine
1
64 4 76
Read next card
0
2
Cue to routine to deal with double punching
with 2m = address of faulty word C(6) = ≠ of faulty word
≠ buffer in B3+ and B4+ Card image in U4 and U5

If the routine finds that every word in the not-equivalent buffer is zero after a clearable failure signal in register 48, this indicates a fault in the error detection equipment. In this case, or if register 48 cannot be cleared, the programme will enter part B of the error routine.

The purpose of part B of the error routine is to store any information in the main store which will be needed to restart the programme after the fault has been rectified. When this storing has been done, the routine will stop and give an indication to the operator that a fault has occurred. Before calling an engineer the operator should, if possible, check that the fault is not due to an error in the programme (see section 11.7.4).

The error print routine will be able to determine the column which has been double punched by examining the word in accumulator 6 on entry. Ths position of the non-zero character can then be related to the column from which it came, according to the distribution being used. This is probably best done by means of a table look-up operation using appropriate information stored in the programme. †

11.7.3 *Card Output*

The programming of card output is similar to that required for card input. The fault register associated with the card punch is addressed as *Special Register* 49††, the allocation of the signal bits in this register is exactly the same as in register 48. The punch will become permanently busy when the magazine becomes empty, the stacker full or a card wreck occurs. In each of these cases, the appropriate light on the card punch will be lit. If a stop occurs because of a card wreck, the wreck condition can be cleared by removing the damaged card and switching off momentarily the power to the card punch. Auto-repunching is then initiated when the green button on the punch is depressed.

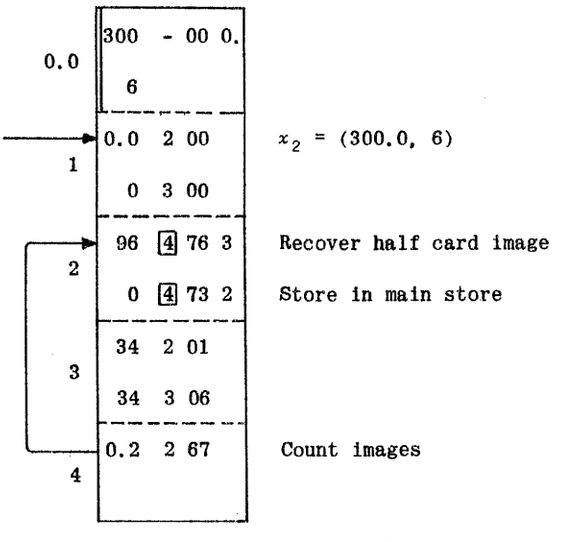
Before starting to punch cards, the output data buffer must be clear. The following procedure should be followed before loading the code tables and the distribution and interpretation tables:

(a) Switch on the punch.
(b) With cards in the magazine, press the punch "ready" button. (Any card images left in the card system will then be punched.)
(c) Lift cards from magazine and depress the "run out" switch.
(d) Empty the stacker.

If, however, a fault has been left on the card punch or the punch has not been used since the computer was switched on, it will be necessary to set up the order 96 [6] 76 on the handswitches, inhibiting parity failures, put the start key to MANUAL and go to RUN. This operation is then repeated with the order 97 [6] 76 set on the handswitches.

As with the card reader, there are two main classes of fault which will cause an error signal to be set in register 49. When register 49 cannot be cleared, this indicates an electronic fault or an error in the programme or the data. In this case it may not be possible to recover the card images from the punch recovery buffer as the fault may be a parity failure in the buffer. When the fault has been cleared the programme should be restarted at the last restart point. Sometimes, however, it may be useful to examine the contents of the punch recovery buffer to identify the point reached in the programme. Since an electronic fault is signalled immediately it occurs, there may not be three card images in the card recovery buffer: an attempt to recover more images than there are in the buffer will lead to a buffer parity failure.

When register 49 can be cleared, this indicates that there has been a failure to obtain an equivalence between the card image which has been punched and the card image obtained from the post-punch sensing station. This type of error is not signalled until the card buffer is full and the auto-repunching circuits have made one attempt at repunching the card. In this case a routine should be entered which will recover the card images from the card recovery buffer, 96, 97, and store them in the main store so that these cards can be repunched before continuing with the programme when the fault has been remedied. The order to recover the first block of the card recovery buffer must be obeyed within 16 beats of having cleared register 49. Since the fault is not indicated until the card output buffer is full there will be three card images in the output buffers. The following set of orders will recover these images and store them on the drum starting in *B*300.



† Note that if there are several double punched columns, the routine will only detect the last of these to appear in the input buffer. If a double punching is detected a check reading failure in the same word of the buffer would not be detected.

†† On machines with two card controls, the fault register associated with the second punch is addressed as Special Register 51.

On completion of this recovery process the programme should stop after indicating that a repeated punch check failure has occurred. Before calling an engineer the operator should, if possible, check that the fault is not due to an error in the programme or to using prepunched cards. After the fault has been dealt with the cards should be run out by pressing the green button with the rotary switch on the punch set to "Motor without Computer".

When auto-repunching takes place, both the faulty card and the card which follows it will be offset in the stacker, since by the time it has been discovered that a card has been wrongly punched the following card will already have been punched. Both cards are repunched.

11.7.4 *Location of Faults*

When a card fault is signalled the type of fault may be ascertained from the indicator on the card bay, (by setting the left hand rotary switch to 3 FAULTS and turning the right hand rotary switch until the fault lamp between the two switches is lit). The faults indicated are as follows:

*1. Check read fail (or double punching)	6. Timing
2. Read conversion	†7. General parity
3. Reader row marker	8. Punch row code
4. Reader row code	9. Punch row marker
†5. Code table parity	*10. Punch conversion
	*11. Punch check fail

Those faults marked with an asterisk can easily be caused by programme errors, such as incorrect code and distribution tables. Those marked with a dagger are normally electronic faults but can be caused by errors such as failure to load the code tables, punching on prepunched cards or changing tables during card operation. When running a new programme a check for programme errors should be made before calling the engineer. If the engineer is not called, the fault should be cleared by selecting the reader or punch and pressing the clear key in the card bay. In the case of the reader, this should be followed by pressing simultaneously the green button and the run out switch on the reader to clear the data buffer. In the case of the punch, the buffer must first be emptied: this may be done by using the orders 96 [6] 76 and 97 [6] 76 as described in Section *11.7.3*, lines 15-18. The cards should then be run out, by pressing the green button with the rotary switch set to "Motor without Computer", before pressing the clear key in the card bay.

11.7.5 *Timing*

When cards are being read as fast as possible, i.e. the programme is limited by the speed of the card reader, the data from successive cards may be read into the computer at intervals of 300 milliseconds. If, however, cards are being read in bursts so that, at some stage, the reader is waiting for the computer, the timing of reading cards will be as follows. The contents of the first card of the group are in D' (see diagram on page 243) and can be transferred to the computer immediately. The contents of the second card, as yet unconverted into computer code, are in C and will be converted as soon as the contents of D' have been transferred to the computer. The conversion process will take about 100 milliseconds so that the contents of the second card can be transferred to the computer about 100 milliseconds after the first. As soon as the contents of the second card have been converted a signal is given to read the next card. There will be an average waiting time of 150 milliseconds for the clutch to engage and the third card, which has already been read at the first reading station, can be read at the second reading station (at the same time the contents of A are transferred to C and the image of the fourth card read into A). The two card images are compared and conversion takes place. The contents of the third card will be available to be transferred to the computer 450 milliseconds (average) after the transfer of the contents of the second card from D' to the computer has taken place. This means that the contents of the first three cards of a group can be read into the computer in about 600 milliseconds, that is, in two card cycles. Thereafter cards may be read at intervals of 300 milliseconds.

When cards are being punched at full speed one card may be punched every 600 milliseconds. If cards are being punched in bursts where the gaps between the bursts are not sufficiently long for the run down switch on the motor to operate, the information for the first two cards can be sent to the punch buffers immediately and, taking an average of the waiting time for the clutch to engage, the information for the third card can be sent to the punch about 860 milliseconds later; thereafter cards can be punched at full punch speed.

11.8 Pseudo off-line working

11.8.1 *Introduction*

On certain installations of Pegasus 2 with punched card facilities, circuits have been provided which will permit automatic transfers of data between any pair of peripheral machines - magnetic tape, printer, card reader and card punch - while leaving the central computer free to do other work. This provides something akin to "time-sharing" in the whole computer installation.

In these transfers the distribution tables and the code conversion table are fully effective. It thus becomes possible to carry out any one of the following operations (for example) while the computer is doing something else:

1. Print out from a previously recorded magnetic tape, with automatic control of layout, and conversion from internal computer code to printer code. A "category search" facility is provided, whereby only selected items on the magnetic tape are printed; thus, for example, documents for customers can be obtained on the first run of the printing tape, documents for agents on the second run, and documents for management on the third run.
2. Print out from cards with automatic rearrangement of layout and, if required, conversion from any card code to the printer code.
3. Input data from cards and record on magnetic tape, as a preliminary operation, with automatic control of field distribution and conversion from any code.
4. Punch cards from a previously recorded magnetic tape with automatic control of field distribution and conversion from internal computer code to any card code.
5. Read cards prepared in any code and punch cards in one of several other codes, with rearrangement of fields if required.

6. As previous example, but also convert from 65-column cards to 80-column (round or rectangular holes), provided that the appropriate card reader and punch are connected.

In all cases the speed of operation is determined by the slower machine of the pair concerned, but this machine runs at its full natural speed. In each of the above cases the central computer can be used for normal computing work whilst the transfers are taking place. In addition, it is possible for the computer to obey programmes which use peripheral equipment (subject to certain restrictions) which is not being used for the automatic transfers.

This facility of automatic transfer of data is provided in the following manner. A supplementary "control" is built into the computer, providing two more special registers in the computing store. Pseudo order-pairs are sent to each of these, defining the peripheral machines involved, specifying the category-selection criteria and specifying the number of cards (or sections of magnetic tape) which have to be transferred. Transfers will then begin, being essentially automatic block-transfers between the relevant data buffers; the timing of these operations is controlled by automatic lockouts; wherever practicable, the computer has priority over the pseudo off-line working. When category search is used, each input information section contains a character which defines the category to which the information in that section belongs. This character is examined by electronic circuits and if the section is found to be in one of the required categories, the transfer to the output peripheral takes place. When the specified number of transfers has been performed the operation will stop and the indicator lamp, which has been lit during the operation, will be extinguished. The appropriate automatic stop will occur in the event of a card wreck, a magnetic tape failure, the card magazine becoming empty, etc..

When such transfers are in progress the appropriate data buffers will be locked out so that other programmes running in the computer at the same time cannot interfere with the transfers.

11.8.2 *Method of Use*

If magnetic tape is being used for the automatic transfers, the mechanism to be used is selected by means of a special socket on the selector panel of the tape control unit. There is also a lamp on this panel which is lit whenever an automatic transfer is taking place involving the tape unit. The magnetic tape may be used in either the 16- or 32-word mode depending on the position of the selector switch on the unit specified. However, since the information from a card is packed into two blocks, when the tape is being written in the 32-word mode only two of the four blocks in a section will contain information from the cards although the other two blocks will be overwritten. Similarly, when punching cards or printing from a 32-word tape by this means, only two of the four blocks in each section can be used during a particular run. Either the first or second pair of blocks in a section may be used, this being determined by the setting of the appropriate half of the magnetic tape buffer in the control word.

Before describing the method by which the automatic transfers are initiated, the category search facility must be described in greater detail. The categories to be transferred are specified by means of a defining character and a mask, both of which are contained in the controlling registers. Before a transfer is made, character 6 of word 0, block 0 of the information for transfer is taken and a not-equivalent operation is performed between this and the defining character. The result of this operation is collated with the mask. If the resulting character is zero, the transfer takes place. Therefore the character from the data to be transferred which is examined in this way defines the category to which the data belongs. Example (i) Suppose category 27 (011011) only is required, then a defining character of the same value and a mask of 63 (111111) will make this selection. Example (ii) Suppose, in addition to category 27, category 26 (011010) is also required, then the defining character may be 26 (or 27) and the mask should have value 62 (111110).

If the category search facility is not required, that is to say, every block of information is to be transferred, the mask should have value zero.

Before the programme which is to run concurrently with the automatic transfers is started, the pseudo off-line operation must be initiated. In order to do this, first the code tables and the appropriate distribution and interpretation buffers must be loaded. This is done in exactly the same way as was described in section 11.6.

The next step is to send a control word to one of the controlling registers, addressed as special register 54. The control word can be expressed as a pseudo order-pair as follows:-

$$\left.\begin{matrix} A_S & - - - \\ 0 & M_1M_2B\ T \end{matrix}\right\} \text{When using magnetic tape} \quad \text{or} \quad \left.\begin{matrix} 1 & - - - \\ 0 & M_1M_2B\ T \end{matrix}\right\} \text{When tape is not being used}$$

where A_S is the starting magnetic tape address of transfers.

M_1 and M_2 are two octal digits representing the mask used for category search. If category search is not being used both M_1 and M_2 should be zero.

B = 0 if the first half of the magnetic tape buffer is being used or magnetic tape is not being used;
= 2 if the second half of the magnetic tape buffer is being used.

T = 0 if transfers are being made which do not involve magnetic tape;
= 1 if transfers are being made *from* magnetic tape;
= 2 if transfers are being made *to* magnetic tape.

The final step in initiating the automatic transfers is to send a control word to the other controlling register, addressed as special register 55. This action causes the transfers to start. The control word can be expressed as a pseudo order-pair in the following way:-

$$\begin{matrix} A_f & - - - & 1 \\ 0 & K_1K_2 & 0 \end{matrix}$$

where, A_f is the last address on magnetic tape which is to be used for a transfer. When magnetic tape is not being used A_f is the number of successful transfers which are to take place. It should be noted that, after completion of the last transfer between the two peripheral machines, there will be one further transfer from the input device but not to the output device.

K_1 and K_2 are two octal digits representing the defining character used for category search; if category search is not being used both K_1 and K_2 may be zero.
For installations with two card controls channel selection digits will be allocated in the two least-significant octal digit positions.

As soon as automatic transfers have started, all data buffers associated with these transfers are locked out from a programme running concurrently on the computer; automatic transfers involving magnetic tape lock out only the half of the magnetic tape buffer specified in register 54. Any attempt to refer by programme to a locked out data buffer will cause an unassigned order stop but the pseudo off-line working will proceed without interruption. There are also lockouts which inhibit writing to register 54 and to the code table and the distribution buffer(s) concerned with the automatic transfers: any attempt to write to one of these will lead to an unassigned order stop without interrupting automatic transfers. However, if, for example, an automatic transfer from magnetic tape to punched cards is in progress, it would be possible for a programme which uses the card reader to run on the computer at the same time, provided that the cards read can be interpreted by the code table used for the automatic transfers. The distribution and interpretation buffer referring to the card reader could be changed without affecting the automatic transfers. Similarly, if the automatic working were using the second half of the magnetic tape buffer, the computer programme could use the first half. It should be noted that, if the computer programme is using magnetic tape whilst pseudo off-line transfers involving magnetic tape are in progress, *all* tape mechanisms plugged into the plugboard on the magnetic tape control unit must be switched to 16-word mode even including those which are not being used and which have their mains supply switched off.

If automatic transfers are being made where cards are the source of the data being transferred, then it is quite possible that the tape finishing address, A_f, will not be known, since the exact number of cards is unknown. In this case, A_f should be set to some high number known to be in excess of the number of cards. Before starting, the Veeder counter on the card reader should be set to zero. At the end of the operation this will show the number of cards read, the number of transfers made to magnetic tape may be found by reading register 54. If the current value of A_s is A'_s, then the number of transfers which have taken place is $A'_s - A_s$.

Automatic transfers will stop, and the lockouts lift, when the specified number of transfers has occurred or the specified tape address has been reached. Alternatively transfers may stop when there are no more cards to be read, but the lockout will remain. In either case, on completion of the current computer programme a short routine, such as R 7931, should be run. This will read registers 54 and 55, thus checking their parity, and print their contents so that there is a printed record of the correct completion of the off-line operation. It will also clear register 55 to ensure that the lockouts described above are lifted.

It is possible to stop the pseudo off-line operation before the number of transfers specified in register 55 has been completed, by a method depending on the type of operation. Where the card reader is being used as the input device, the magazine may be allowed to run out of cards. In cases where the input medium is magnetic tape, the "stop" button on the output device should be pressed. If it is desired to run the last few images out of the system the tape unit should then be switched to "local" and the output device switched on again. It is advisable when restarting a printing or card punching operation to repeat a few lines or cards as a check. A restart tape to continue the operation may be obtained, if desired, when R 7931 is run. Any pseudo off-line operation may be stopped by clearing register 55 manually or by running R 7931, but this is not always satisfactory because it is not then possible to run the last few images out of the system.

If there is a parity failure in register 54 or 55 during automatic transfers, the error may not be detected immediately but the incorrect parity will be preserved provided that there is not a further compensating error. If the computer programme then obeys any order (except the 23-order) with an N-address of 54 or 55, or obeys a shift instruction with an N-address of 54 or more, it will stop on a parity failure. When writing a programme which is designed to be run at the same time as automatic transfers, it is, therefore, advantageous to include occasional references to addresses 54 and 55 so that a parity failure in either of them will be detected at an early stage. Otherwise the fault may not be detected until the checking programme R 7931 is run.

The progress of automatic transfers may be monitored by examining registers 54 and 55. They may be displayed on the left-hand monitor tube by setting the upper left-hand monitor control to "TEST 2": the lower control must be set to "TAPE READER" to select 54 and to the next clockwise position to select 55.

11.9 Line Printer

It is possible to adapt the Pegasus 2 card control system to drive a line printer, which may be either an ICT 662 (150 lines/minute) or an ICT 902 (100 lines/minute). The printer is selected in place of the card punch by means of a rotary switch on the front of the card control bay.

The 662 can print 120 characters to the line and there are 120 "computer outlets" (addressed like card columns but numbered 1 to 120) which are usually connected to the corresponding printing positions. Certain extra facilities are available by means of a plugboard. A printing code has been chosen to give maximum protection against a fault changing one digit into another. The 902 is available with from 50 to 100 print wheels to the line and the same number of computer outlets, normally connected to the corresponding printing positions.

Either printer may be fitted with one or two independently controlled webs of continuous stationery, which may operate side by side or overlapped. A web consists of continuous folded stationery which may be single sheets or 2 or 3 part sets (top sheet and 1 or 2 copies), except that 3 part sets may not be used when the two webs are overlapped. Paper movement is controlled by the programme, assisted by form layout belts moving in step with the webs. The main web uses the ICT UPF (Universal Paper Feed) paper tape with two channels, A and B, each using 3 tracks on the tape. The second web uses the plastic belt of the Alacra dual feed mechanism.

11.9.1 *Method of Use*

Programming for the line printer is very similar to programming for the card punch, as described in section 11.7, but there is no equivalent of a punch check-read failure and therefore when register 49

indicates a fault it will not be possible to clear it. Each line of print is assembled in 96 six-bit characters: up to three of these characters may be needed for control purposes but the others may contain characters for printing. The two blocks containing these 96 characters are sent to the Output Data Buffer by 76-orders with the same N-addresses as for the card punch; for example, 88 [4] 76 followed by 81 [5] 76.

The distribution of the 96 characters in the Output Data Buffer to computer outlets (and thence to positions on the line) is controlled by the Output Distribution and Interpretation Table, as described in section 11.6.1. The correspondence between the six-bit computer code and the printed characters is defined by the second code of the Code Table, as shown in Table 11.10 for the standard character set. Erase (63) should be inserted against any unwanted characters in Tables X, Y and Z. The Code Table is loaded as described in section 11.6.2, but rules 3 to 8 of that section are not applicable unless the table is also to be used for card punching. Both the 662 and the 902 require a 4-zone code and the J1006.0 introducing the Code Table must be preceded by +0.

Paper movement for each web is controlled by a six-bit layout character in the Output Data Buffer: the corresponding entries in the Output Distribution and Interpretation Table must be 125.24 for the main web and 126.24 for the second web. Recommended values of these layout characters, to determine the paper movement after printing the line, are shown below; in the absence of a layout character there will be no paper movement and the line will be overprinted.

Character	Main Web (125)	Character	Second Web (126)
0	No Movement	0	No Movement
2	1 Line Feed	2	1 Line Feed
4	2 Line Feeds	4	2 Line Feeds
8	3 Line Feeds		
16	Throw Channel A	16	Throw
32	Throw Channel B	32	Short Throw

The length of a paper throw is determined by form layout belts which move in step with the paper. A "short throw" is one of less than three inches, for which time is available without inhibiting the printing mechanism. On the main web a short throw instruction is unnecessary because the inhibition of the printing mechanism is controlled by information on the layout belt. The two layout characters may appear anywhere in the Output Data Buffer, but by convention they are placed in characters 3 and 4 of word 0 if that is convenient.

11.9.2 *Additional Facilities*

Five characters are available via character emitters on the 662 plugboard for insertion in any specified printing position. These are

* (Asterisk or Plus)	- (Dash or Minus)	. (Full Stop)
, (Comma)	| (Vertical Bar)	

Note that the last two, comma and vertical bar, are not available from the computer. By the use of "symbol selectors" on the plugboard the printing of these emitted characters may be made conditional on the value of specified characters in the line. Zero suppression is also available from the plugboard but it is usually better to arrange this by programme, using the 37 order, and thus avoid special plugging.

Two further control facilities are available by sending a six-bit character to the special computer outlet 127. The character must appear in the first word of the Output Data Buffer, either in character position 6 or, if this clashes with the pseudo off-line category, in character position 5. The corresponding entry in the Output Distribution and Interpretation Table must be 127.26. The six-bit character then has the effects shown below; only two bits are used for control purposes and the others, denoted by δ, are available for category definition if character position 6 is used.

Character	Example	Effect
0$\delta\delta\delta\delta$0	0	Print line
0$\delta\delta\delta\delta$1	1	Print line with alternative layout
1$\delta\delta\delta\delta\delta$	32	Punch card

The facility to switch between the printer and the punch by programme is dependent on setting the rotary switch on the card bay to its neutral position, between "Print" and "Punch". If computer outlet 127 is not used the effect is the same as sending zero to it, so the printer is selected. The alternative layout facility is only available using the 662 plugboard: it enables the connections from up to 40 computer outlets to be switched to different printing positions. The relevant connections must be made via co-selectors on the plugboard and the pick up points of these co-selectors must be connected to computer outlet 127. A further ten connections may be switched by using pilot selectors as co-selectors.

If it is required to change the Output Distribution and Interpretation Table, as will usually occur when changing from the 662 to the punch and back, it is first necessary to allow sufficient time for all information under the old control to be cleared from the system. The necessary delay can best be measured from the instant at which register 49 becomes clear after sending the previous line or card image to the Output Data Buffer. A sufficient delay is 0.6 seconds for the 662, 1.3 seconds for the 902 and 3.0 seconds for the card punch. If the printer or punch has been kept running at full speed these times can be reduced to 0.5, 0.7 and 1.9 seconds respectively, and the punch delay times can be reduced by a further 1.2 seconds if automatic repunching is suppressed. When the programme is run the printer and punch must not be allowed to stop due to the paper running out, the red button being pressed, the magazine becoming empty or the stacker full, because such stoppages would make the above delays inadequate. The Distribution and Interpretation Table appropriate for the punch must be restored at the end of the programme before pressing the run-out switch on the punch.

When using the printer it may be required to change the layout of the line by changing the distribution part of the Output Distribution and Interpretation Table, leaving the interpretation part unchanged (i.e. retaining the same conversion number for each character in the Output Data Buffer). This can be done without the delays described above provided that the changes do not affect computer outlets 0, 125, 126 and 127. The first line to which the new distribution applies must be sent to the Output Data Buffer *before* changing the distribution. The programme should then enter a short loop testing register 49. As soon as register 49 is clear the Output Distribution and Interpretation Table may be changed, and the change must be completed within 100 milliseconds. This means that the Run key must not be brought to STOP during this part of the programme.

CODE CONVERSION TABLES WITH PRINTER

Type 1

Extensive Main Code, including I.B.M. 5th Zone;

Second code used with line printer

Where there are alternative interpretations of a single hole punched in the upper curtate or of an upper curtate punching using split-column working, the conversion number is given in brackets after the punching. The conversion numbers which can be used with this loading are 0, 2, 4, 6, 16, 18, 20, 22, 1, 5, 3 with the main code, 8 with the printer code and 24, 26. Conversion numbers 9, 10, 11 and 13 have meaning only when the second code is being used for card input or output.

The letter O is represented by Ø to distinguish it from zero. The character ⌗ can represent either plus or asterisk.

Intermediate Code (Buffer Address 24)	Table X Main Code (Buffer Address 25)		Table Y Main Code (Buffer Address 26)		Table Z Printer Code (Buffer Address 27)	
	Punching UC/LC	6-bit char.	Punching UC/LC	6-bit char.	Printing 902 662	6-bit char.
8	-/1		-/9		1 Ø	
10	10/1		10/9		A ⌗	
12	11/1		11/9		J .	
14	0/1		0/9		. -	
9	-/- (2 or 18)		UC -/-/0 (1)		9 X	
11	10/- (2 or 18)		UC 10/-/0 (1)		I 9	
13	11/- (2 or 18)		UC -/11/0 (1)		R Z	
15	0/- (2 or 18)		UC 10/11/0 (1)		Z I	
16	-/2		-/2/8		2 0	
18	10/2		10/2/8		B A	
20	11/2		11/2/8		K &	
22	0/2		0/2/8		S £	
24	-/3		-/3/8		3 B	
26	10/3		10/3/8		C 1	
28	11/3		11/3/8		L D	
30	0/3		0/3/8		T E	
32	-/4		-/4/8		4 F	
34	10/4		10/4/8		D G	
36	11/4		11/4/8		M H	
38	0/4		0/4/8		U 3	
40	-/5		-/5/8		5 10	
42	10/5		10/5/8		E K	
44	11/5		11/5/8		N 4	
46	0/5		0/5/8		V L	
48	-/6		-/6/8		6 M	
50	10/6		10/6/8		F 5	
52	11/6		11/6/8		Ø P	
54	0/6		0/6/8		W Q	
56	-/7		-/7/8		7 R	
58	10/7		10/7/8		G S	
60	11/7		11/7/8		P T	
62	0/7		0/7/8		X 7	
1	-/- (0 or 16)		UC -/-/- (1)		8 11	
3	10/- (0 or 16)		UC 10/-/- (1)		H V	
5	11/- (0 or 16)		UC -/11/- (1)		Q 8	
7	0/- (0 or 16)		UC 10/11/- (1)		Y W	
17	-/- (4 or 20)		UC -/-/- (5)		U	
19	10/- (4 or 20)		UC 10/-/- (5)		Y	
21	11/- (4 or 20)		UC -/11/- (5)		"	
23	0/- (4 or 20)		UC 10/11/- (5)		C	
25	-/- (6 or 22)		UC -/-/0 (5)		2	
27	10/- (6 or 22)		UC 10/-/0 (5)		J	
29	11/- (6 or 22)		UC -/11/0 (5)		N	
31	0/- (6 or 22)		UC 10/11/0 (5)		6	
0	LC-		-/8		Sp Sp	
2	NOT USED	63	10/8		+ <	
4	NOT USED	63	11/8		- >	
6	NOT USED	63	0/8		0 /	

Table 11.10 *Code Conversion Table Type 1 for Printer*

Appendix I

The Pegasus Order–Code

F	Effect	Description
Group 0		
00	$x' = n$	Replace content of accumulator by copy of content of register.
01	$x' = x + n$	Add number in register into accumulator.
02	$x' = -n$	Replace content of accumulator by minus the number in the register.
03	$x' = x - n$	Subtract number in register from accumulator.
04	$x' = n - x$	Subtract number in accumulator from number in register, leaving the difference in the accumulator.
05	$x' = x \,\&\, n$	In the binary digital positions where the word in the register has 1's, leave the digits in the accumulator unchanged; elsewhere replace them by 0's.
06	$x' = x \not\equiv n$	In the binary digital positions where the word in the register has 1's, reverse the digits in the accumulator; elsewhere leave them unchanged.
07	Unassigned.	
Group 1		
10	$n' = x$	Replace content of register by copy of content of accumulator.
11	$n' = n + x$	Add number in accumulator into register.
12	$n' = -x$	Replace content of register by minus the number in the accumulator.
13	$n' = n - x$	Subtract number in accumulator from register.
14	$n' = x - n$	Subtract number in register from number in accumulator, leaving the difference in the register.
15	$n' = n \,\&\, x$	In the binary digital positions where the word in the accumulator has 1's, leave the digits in the register unchanged; elsewhere replace them by 0's.
16	$n' = n \not\equiv x$	In the binary digital positions where the word in the accumulator has 1's, reverse the digits in the register; elsewhere leave them unchanged.
17	Unassigned.	

Group 2 Suffixes I and F indicate integers and fractions, respectively; suffix M indicates a mid-point number. $\epsilon = 2^{-38}$.

Multiply

20 $(pq)'_I = n_I . x_I$ or $(pq)'_F = n_F . x_F$

or $(pq)'_M = n_I . x_F = n_F . x_I$ $\quad (q' \geqslant 0)$

Multiply the number in the specified register by the number in the specified accumulator and place the product in accumulators 6 and 7.

21 $(pq)'_F = n_F . x_F + \tfrac{1}{2}\epsilon$

or $(pq)'_M = n_I . x_F + \tfrac{1}{2} = n_F . x_I + \tfrac{1}{2}$ $\quad (q' \geqslant 0)$

Multiply the fraction in the specified register by the fraction in the specified accumulator and place the rounded product in X6 (X7 receives the rest of the product).

22 $(pq)' = (pq) + n . x \quad (q' \geqslant 0)$

Multiply the number in the specified register by the number in the specified accumulator and add the resulting double-length product into the double-length accumulator (X6 and 7).

Justify

23 $(nq)'_F = n_F + \epsilon\, q_F +$ *contribution from OVR* $\quad (q' \geqslant 0)$

Put into standard form (or justify) the double-length number in the specified register and X7, on the assumption that an earlier addition or subtraction in X7 has determined the state of OVR. Leave $C(7)$ non-negative, and leave OVR clear unless the left half of the number (in the register) overflows. In the above equation the contribution from OVR is zero if OVR is clear, but is $\pm\, 2\epsilon$ if OVR is set (the sign being opposite to that of q).

Divide

24
$$\left.\begin{aligned} q'_I + p'_I/n_I &= (xq)_I/n_I \\ \text{or } q'_F + p'_F/n_F &= (xq)_F/n_F \\ \text{or } q'_I + p'_F/n_F &= (xq)_M/n_F \end{aligned}\right\} \quad (0 \leqslant p'/n < 1)$$

$$\text{or } q'_F + p'_F/n_I = (xq)_M/n_I \qquad (0 \leqslant p'_F/n_I < \epsilon)$$

Divide the double-length number in the specified accumulator and X7 by the number in the register; place the quotient in X7 and the remainder in X6. (Often $X = 0$ and the description can read: divide the integer in X7 by the integer in the register, placing the quotient in X7 and the remainder in X6.)

25 The equations given for the 24-order apply also to the 25-order, but the inequalities become

$$-\tfrac{1}{2} \leqslant p'/n < \tfrac{1}{2} \quad \text{or} \quad -\tfrac{1}{2} \leqslant p'_F/n_I < \tfrac{1}{2}$$

Divide as with the 24-order, but place the rounded quotient in X7 (and the corresponding remainder in X6).

26
$$\left.\begin{aligned} q'_F + p'_F/n_F &= x_F/n_F \\ \text{or } q'_I + p'_F/n_F &= x_I/n_F \end{aligned}\right\} \quad (-\tfrac{1}{2} \leqslant p'_F/n_F < \tfrac{1}{2})$$

$$\text{or } q'_F + p'_F/n_I = x_I/n_I \qquad (-\tfrac{1}{2} \epsilon \leqslant p'_F/n_I < \tfrac{1}{2}\, \epsilon)$$

Divide the fraction in the specified accumulator by the fraction in the register; place the rounded quotient in X7 (and the corresponding remainder in X6).

27
$$\left.\begin{aligned} p' &= 2X.p + n \\ q' &= 0 \end{aligned}\right\} \text{ General form}$$

This is primarily intended to allow fast assembly from paper tape, the radix being $2X$.

$$\left.\begin{aligned} p' &= 2X.p + (\text{digits 5 - 8 of } q \text{ shifted down 30 places}) \\ q' &= 2^6 . q \quad (\text{logical shift}) \end{aligned}\right\} \text{ Special form when } N = 7$$

This special form converts into binary a numerical quantity held as 6-bit characters in accumulator 7.

Group 3

30 - 36 These orders are unassigned.

37 $p' =$ Remainder of $\left(\frac{2X.p}{n}\right)$,

$q' = 2^6\, q +$ Quotient of $\left(\frac{2X.p}{n}\right)$.

Convert an integer held in binary in accumulator 6 into 6-bit characters and pack these characters into accumulator 7. The most significant character in radix $2X$ is obtained using the scale number n.

Group 4

40	$x'_I = N$	Replace content of accumulator by integer written in the order.
41	$x'_I = x_I + N$	Add integer written in the order to integer in accumulator.
42	$x'_I = -N$	Replace content of accumulator by minus the integer written in the order.
43	$x'_I = x_I - N$	Subtract integer written in the order from integer in accumulator.

Group 4

44	$x_I' = N - x_I$	Subtract integer in accumulator from integer written in order; result in accumulator.
45	$x' = x \,\&\, N$	In the binary digital positions where the integer written first in the order has 1's, leave the digits in the accumulator unchanged; elsewhere replace them by 0's (N.B. the result can have 1's only in the 7 right-hand digits).
46	$x' = x \not\equiv N$	In the binary digital positions where the integer written first in the order has 1's, reverse the digits in the accumulator; elsewhere leave them unchanged (N.B. at most the 7 right-hand digits will be changed).
47	Unassigned.	

Group 5 (N is the number written first in the order)

Single-length arithmetical shifts

50 $x' = 2^N x$

Multiply the number in the specified accumulator by 2^N.

51 $x' = 2^{-N}x = x/2^N$ (rounded)

Divide the number in the specified accumulator by 2^N, and place the rounded quotient in the accumulator.

Single-length logical shifts

52 Shift the binary digits of the word in the accumulator to the left (up) N places. Discard the N digits which are shifted beyond the sign-digit position, and make the last N digits of the result all zero. Do not affect OVR.

53 Shift the binary digits of the word in the accumulator to the right (down) N places. Discard the N digits which are shifted off the right-hand end of the accumulator, and make the first N digits of the result all zero.

Double-length arithmetical shifts

54 $(pq)' = 2^N(pq)$

Multiply the double-length number in X6 and 7 by 2^N (assuming that $q \geqslant 0$).

55 $(pq)' = 2^{-N}(pq)$ (unrounded)

Divide the double-length number in X6 and 7 by 2^N and place the unrounded double-length quotient in X6 and 7 (this definition assumes that $q \geqslant 0$).

Normalize

56 $(pq)_F' = 2^{\mu}(pq)_F$, $x_I' = x_I - \mu$,

where μ is an integer chosen so that

(a) $\frac{1}{4} \leqslant (pq)' < \frac{1}{2}$ and $-1 \leqslant \mu \leqslant N - 1$,

or (b) $-\frac{1}{2} \leqslant (pq)' < -\frac{1}{4}$ and $-1 \leqslant \mu \leqslant N - 1$,

or (c) $-\frac{1}{4} \leqslant (pq)' < \frac{1}{4}$ and $\mu = N - 1$.

Normalize the floating-point number whose argument is the fraction (pq) in X6 and 7, and whose exponent is the integer in the specified accumulator. But do not shift the argument up more than $N - 1$ places. This definition assumes that $q \geqslant 0$.

57 Shift the digits of the word in the accumulator up l 6-bit characters and then down r 6-bit characters, where the order is written as $(l.r)$ X 57.

Group 6 (N specifies where a jump is to go)

Accumulator Test

60 Jump to N if $x = 0$.

Jump if the number in the accumulator is zero.

61 Jump to N if $x \neq 0$.

Jump if the number in the accumulator is not zero.

62 Jump to N if $x \geqslant 0$.

Jump if the number in the accumulator is positive or zero.

63 Jump to N if $x < 0$.

Jump if the number in the accumulator is negative.

Overflow Test

64 Jump to *N* if OVR clear; clear OVR

If the overflow-indicator is clear, jump. If it is set, clear it and do not jump.

65 Jump to *N* if OVR set; clear OVR.

If the overflow indicator is clear, continue with the next order. If it is set, clear it and jump to the specified order.

Unit-modify

66 $x'_M = x_M + 1$, jump to *N* if $x'_P \neq 0$

Add one to the modifier in the specified accumulator, and then jump to the specified order if the new position-number in the modifier is not zero.

Unit-count

67 $x'_C = x_C - 1$ (leaving x_M unchanged), jump to *N* if $x'_C \neq 0$.

Reduce by one the counter x_C in the specified accumulator (without affecting the modifier x_M), and jump to the specified order if the new value of the counter is not zero.

Group 7

70 *Single-word read.* Place in X1 a copy of the word in the main store location whose block-number is in the *N*-part and position number in the *X*-part of the order.

71 *Single-word write.* Place a copy of the word in X1 in the main store location whose block-number is in the *N*-part and position-number in the *X*-part of the order; but stop instead if OVR is set.

72 *Block read.* Read the block in the main store whose address is in the *N*-part of the order and place a copy of its contents in the computing store block specified by the *X*-part of the order.

73 *Block write.* Take the computing store block specified by the *X*-part of the order and write a copy of its contents into the main store block whose address is in the *N*-part of the order; but stop instead if OVR is set.

74 *External conditioning.* Select the input and output equipment to be used, by setting the external conditioning relays to correspond to the binary digits of *N*, the number written first in the order. Usually only two values of *N* (0 and 1) are encountered; these select the main or the second tape reader.

75 Unassigned.

76 *Buffer transfer.* If magnetic tape or punched card equipment or a line printer is fitted, a transfer of information takes place between a block in the computing store and a buffer store block.

77 *Stop.* Continue with the next order when the Run key is operated.

Appendix 2

The Lesser Library

This collection of useful subroutiones is mainly used for teaching purposes; it occupies Blocks *B*50 to *B*85, which should not be overwritten.

	Cues		Description
A	**Integer Print**		Uses: U0, B0. Link: in X1. Number in X7. Number of digits required in X2. Precedes number by minus sign or space.
1	50 [0] 72 0.0 0 60	CR LF before number	
2	50 [0] 72 0.3 0 60	Sp before number	
B	**Fraction Print**		Uses: U0, B0. Link: in X1. Number in X7. Number of digits required in X2. Precedes unrounded fraction by minus sign or space.
1	51 [0] 72 0.0 0 60	CR LF before number	
2	51 [0] 72 0.3 0 60	Sp before number	
C	**Mixed Number Print**		Uses: U0, B0. Link: in X1. Number in X6 and 7; integral part in X6, fractional part in X7. Numbers of digits required before and after decimal point are to be set in X4 and 5 respectively.
1	53 [0] 72 0.0 0 60	CR LF before number	
2	53 [0] 72 0.2+ 0 60	Sp before number	
D	**Sterling Print**		Uses: U0, B0, X6. Link: in X1. Signed, whole number of pence in X7 is printed in £.s.d. Number of digits required in the £ figure to be set in X2.
1	60 [0] 72 0.0 0 60	CR LF before amount	
2	60 [0] 72 0.2 0 60	Sp before amount	
E	**Initial Orders Input**		Uses: All computing store; B0. Link: in X1, obeyed when L-directive is read. Normally used to read in numbers in ordinary Initial Orders notation (integers or fractions) with L after the last. Main store address for first number to be set in 5.7_M or else by T-directive on number-tape.
	37 X 00 10 [0] 72 X 0.0 0 60	X is any accumulator except X1	

F Mixed Number Input

```
64 [0] 72
0.0  0 60
```

Uses: U0, 1; X1, 6, 7. Link: in X1. Reads a signed or unsigned mixed number from tape, leaving integral part in X6, fractional part in X7. Number must be terminated by CR LF or Sp.

G Sterling Input

```
69 [0] 72
0.5+ 0 60
```

Uses: U0, 1; X1; B0. Link: in X6. Reads a signed or unsigned sterling amount from tape, converts to pence and leaves in X1. Amount must be punched as *a.b.c* and followed by CR LF or Sp.

H Handswitch Input

```
75 [0] 72
0.0  0 60
```

Uses: U0, 1; X7; B0. Link: in X1. Reads unsigned integer tapped out, digit by digit, on handswitches and leaves it in X7. Optional stop on entry; then clear handswitches, go to RUN and momentarily press keys of digits required (e.g. H7 for 7). Press H10 to show finish.

I Square Root

```
77 [0] 72
0.0  0 60
```

Uses: U0; X5, 6, 7. Link: in X1. Double-length fraction (pq) in X6, 7. Result is $\sqrt{(pq)}$ in X6.

J Sine or Cosine

1
```
78 [0] 72
0.0  0 60
```
sine

2
```
78 [0] 72
0.0+ 0 60
```
cosine

Uses: U0, 1; X5, 6, 7. Link: in X1. Angle set as fraction (p) of 180° (or π) in X6. Result is $\sin \pi p$ or $\cos \pi p$ in X6.

K Logarithm

```
81 [0] 72
0.0  0 60
```

Uses: U0 1; X5, 6, 7. Link: in X1. Double-length positive fraction (pq) in X6, 7. Result is $\frac{1}{32}\log_e(pq)$ is X6.

L Exponential

```
84 [0] 72
0.0  0 60
```

Uses: U0, 1; X6, 7. Link: in X1. Fraction p in X6. Result is ¼ exp p in X6.

Appendix 3

Special Register 53 — Creed 3000 Punch

Special Register 53 - Creed 3000 Punch

Special register 53 is used for sensing Creed 3000 punch faults, and for re-setting the punch fault circuits. It may be used in two ways dependent on whether it is desired to check individual characters or blocks of characters.

If a fault occurs, special register 53 will be set to -1.0 and the punch will stop (except when the automatic reset facility is used: see below). The sign bit of the register may be examined in the normal way, for example by obeying the order

53 1 00

and the register can be cleared (i.e. the sign bit can be cleared) by writing X0 into it. Clearing register 53 causes a "reset" signal to be sent to the punch, and this restores the punch fault circuits to their quiescent condition and allows the punching to proceed. This mode of operation thus allows for the checking of individual characters. It is possible that register 53 may not be clear (the fault bit may be present) on entry to a programme and therefore the programme should arrange to clear it before sending a character to the punch.

The check reading station in the Creed 3000 punch is spaced 3 characters from the punching station. Thus, assuming that a character *F* has been wrongly punched, *F* will be at the check reading station whilst character *F*+3 is being punched. On completion of punching cycle *F*+3, the computer can initiate punching of character *F*+4, but the fault logic in the punch will prevent *F*+4 from being punched, and a permanent busy signal will be sent back to the computer. Although the computer can continue to do any organisational work, as soon as it attempts to initiate the punching of character *F*+5, a tape busy stop will be produced. If at this stage the manual reset button on the punch is pressed, the punch will proceed with the punching of *F*+4, *F*+5 etc., but the fault bit will remain in register 53.

Once the punch has stopped on an error, a programmers' method of clearing the fault would be to enter manually a character recovery routine, which incorporates a 53 0 10 instruction to clear the fault bit in register 53. This routine would have to re-organise character *F* for re-punching, which entails going back 5 characters (since the computer will have had character *F*+5 ready before the tape busy stop occured).

If output is in the form of a series of blocks of characters, it may only be necessary to inspect register 53 at the end of each block, and arrange to re-punch the complete block with appropriate warning characters should a fault have been detected. In this case register 53 should be set initially to +½ by the programme. When an error is detected register 53 is set to -1.0, as before, but reset signals are then sent continuously to the punch until register 53 is cleared by writing X0 into it. The punch does not, therefore, stop when a fault occurs. Once the programme has written ½ into register 53, the resulting automatic reset facility can only be cleared by moving the Start key to START.

Note that "clearing" register 53 will cause the sign bit only to be cleared. Also copying the contents of register 53 to an ordinary register will cause the sign bit only to be copied. The automatic reset facility, which results when +½ is sent to register 53, can only be cleared by operating the Start key, and cannot be removed by programme.

Note also that at any point at which the punch has come to an error stop, pressing the manual reset button will allow the punch to continue, but will not clear the fault bit in register 53. The logic of the punch will cause the punch to stop again if another character fails the check reading. Thus, register 53 is not affected by the manual reset facility.

Appendix 4

A Guide to the Timing of Programmes

The unit of measurement of time in a programme is the *beat* or *word-time*. This is the time taken to transfer a word from one part of the computer to another. One beat is 126 microseconds, very nearly 1/8th millisecond.

The computer normally obeys an order-pair in 5 beats, allocated as follows:-

1	Order-pair flows into order register.
2 and 3	*a*-order obeyed
4 and 5	*b*-order obeyed

It is convenient to think of this as 3 beats for an *a*-order and 2 beats for a *b*-order. Some orders require extra beats, as follows:-

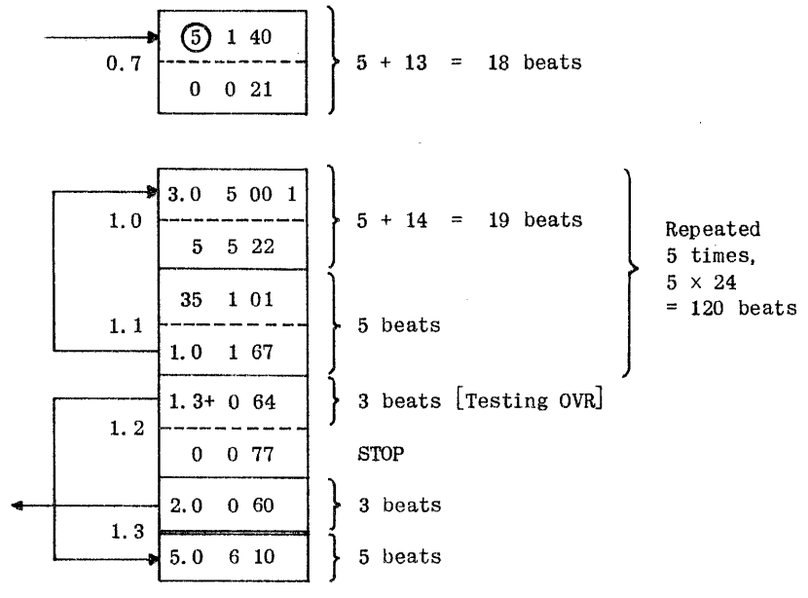
Order	Number of extra beats
Groups 0, 1, 4	None
20, 21	13
22	14
23	None
24, 25, 26	41
50, 51, 53-55	*N* (Number of places shifted)
52	See below
56	*m*+2 (*m* = Number of places shifted)
70 - 73	See below
74	None
76	8

Jump orders require no extra beats except where a jump to a *b*-order occurs. In this case 3 extra beats are needed. If the jump order does not cause a jump no extra beats are required.

52 order: Counter-to-Modifier (Logical) Shift

If $N = 25$ (after modification) in the order (N) *X* 52, no extra beats are required. In general, if $N \geqslant 25$ there will be a slow shift of $N-25$ places, each place taking one beat, followed by a fast shift of the remaining 25 places taking 2 beats; thus the order will take $N-25$ extra beats. If $N < 25$, the order takes N extra beats.

The following example illustrates the timing of a short piece of programme.



The total time (if overflow does not occur) is thus

18 + 120 + 3 + 5 = 146 beats (nearly 18 millisec).

Main Store Transfers

The main store comprises 8192 words and of these, 128 (or optionally 256) words are held on 8-word delay lines; the remaining 8064 (or 7936) words are stored on the drum.

Delay Line Transfers

The 8-word delay line storage is used for quick transfer of information between the main store and the computing store, and is referred to as the Intermediate Access Store (or I.A.S.). Each line contains a block of locations and the blocks are numbered 0-15 (or 0-31). There will be no waiting time for transfers involving these blocks, and an average of only 4 beats waiting time for single word transfers.

Extra beats taken by I.A.S. transfers:

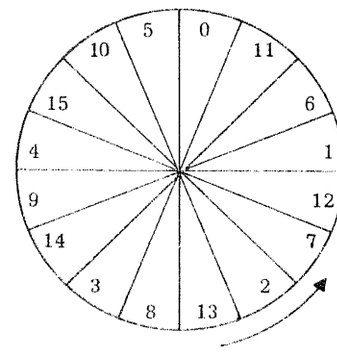
Order	Extra beats	
70	0	+ waiting time of up to 7 beats
71	1	
72	7	
73	8	

Drum Transfers

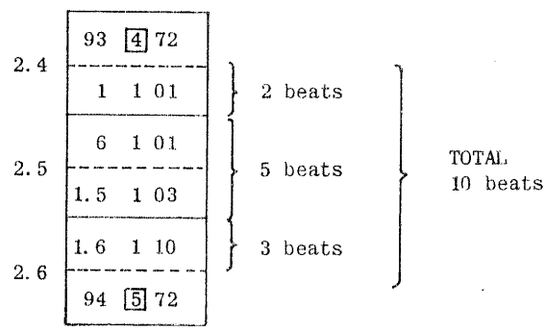
The drum of the computer has 63(or 62) tracks, each of 16 blocks (128 locations). The time for one revolution is 128 beats (16 millisec). There will thus be an average waiting time of 64 beats before having access to any particular drum location. This average waiting time will normally be sufficient for estimating the timing of drum transfers. It is, however, sometimes convenient to consider this timing in a little more detail.

The 16 blocks of any track of the drum are not arranged in numerical order, but in the order shown in the diagram.

It will be seen that any two consecutively numbered blocks are separated by two other blocks. There will consequently be at least 11 beats waiting time between any two consecutively numbered block transfers. This time may be used to obey 4 simple orders.



e.g. -



The two drum transfers will be obeyed on the same drum revolution. If more than 11 beats occur between two consecutively numbered transfers, there may be a delay of one drum revolution.

Extra beats taken by drum transfers are:

Order	Extra beats	
70	3	+ waiting time of up to 127 beats.
71	1	
72	10	
73	8	

Input and output

The tape reader functions at up to 300 tape characters per second, corresponding to 3.3 milliseconds or 26 beats per character. Input of programme by the Initial Orders takes just less than one second per block. Input of numbers varies very much with the form of punching, but 2000 numbers per minute will serve as a rough guide. Binary input takes about 0.23 seconds per block.

The output punch functions at up to 60 characters per second (i.e. 17 millisec or about 133 beats per character). Most output routines use very nearly the full speed of the punch.

Subroutines

The times of subroutines given in library specifications are measured from the first order of the subroutine to the first order of the link. On the average about 160 beats should be added to the specified times to allow for the cue and the link, if these include block-transfer orders.

Note on timing of a complete programme

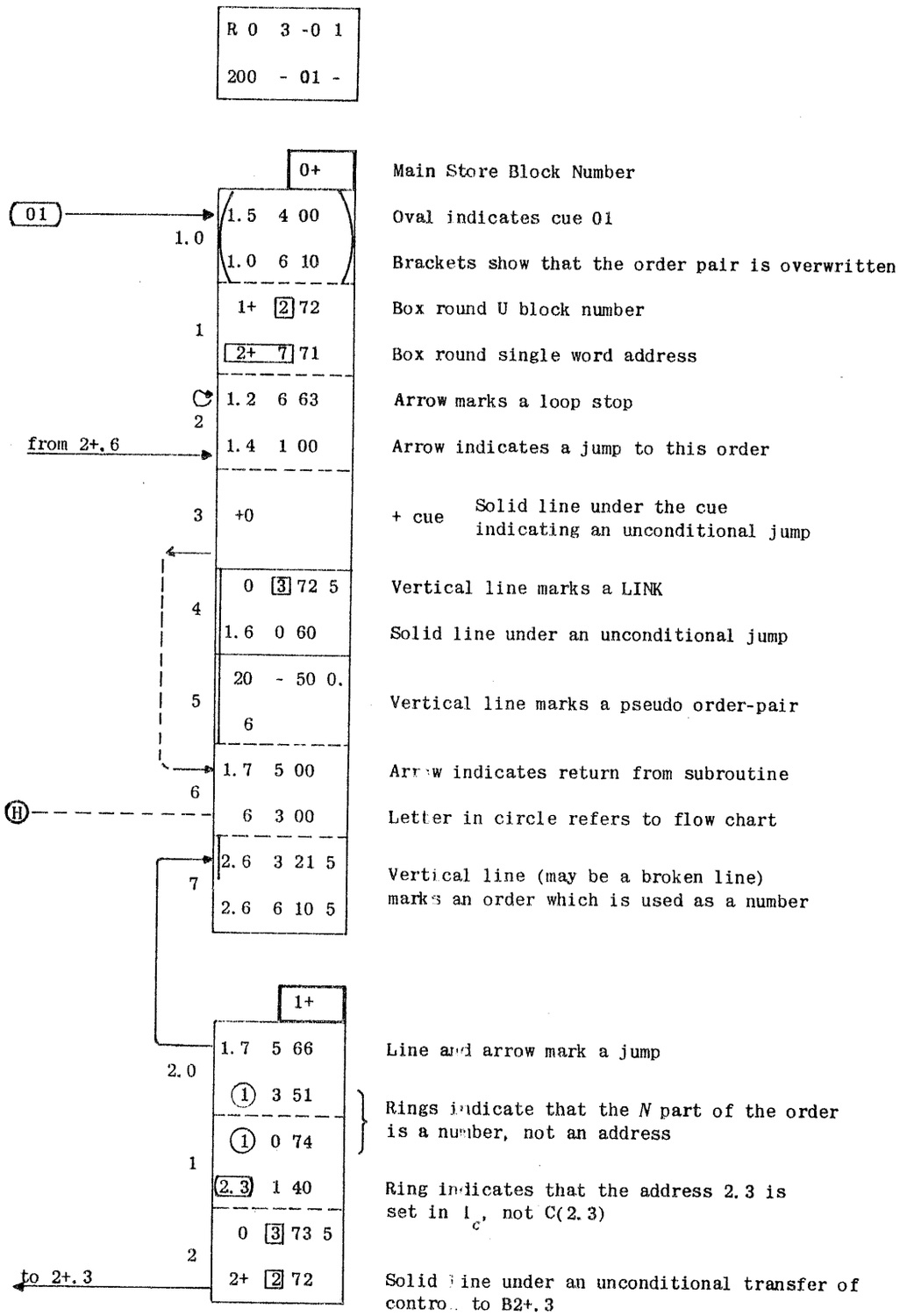
It is seldom necessary to time all the orders of a programme. In most cases the time taken by a programme is governed by the time taken in its inner loop, or by its output time, and an estimate of one or both of these times will generally give a good approximation to the total programme time.

Appendix 5

Abbreviations and Symbols

B 2+	Block 2+ in the main store
B.P	Block and Position Address
C(2)	The content of Accumulator 2
Er	Tape Character Erase
I.O.	The Initial Orders
l.s.	Least significant (or less significant)
m.s.	Most significant (or more significant)
OVR	The Overflow Indicator
P	Accumulator 6 (*P* for Product)
p	The content of Accumulator 6
P.P.	Preset Parameter
PQ	The double-length Accumulator, 6 and 7
(*pq*)	The double-length number in *PQ*
Q	Accumulator 7 (*Q* for Quotient)
q	The content of Accumulator 7
r.o.c.	Round-off constant
Sp sp	Tape Character Space
U 2.1	Register 2.1 in the Computing Store. The U is often omitted
W 2	Block 2 in the magnetic tape buffer
X 2	Accumulator 2. The X is usually omitted
x_2	The content of Accumulator 2
α	The α-search is the section (or setting) of an input routine which ignores blank tape
β	The β-search is the section (or setting) of an input routine which searches for the start of a number or a letter shift
γ	The γ-search is the section (or setting) of an input routine which searches for a warning character or directive
ϕ	Tape Character Figure Shift (≡ Blank Tape)
λ	Tape Character Letter Shift
⊙ ●	Tape Character Full Stop (≡ Decimal Point)
3_c or 3_C	The counter in Accumulator 3
3_m or 3_M	The modifier in Accumulator 3
3_B	The block part of the modifier
3_P	The position part of the modifier
(4.3, 20)	A modifier of 4.3 and a counter of 20

Symbols on Programme Sheets



Appendix 6

Entries to Initial Orders Routines

1. Initial Orders 'Start' entry

```
        37  X 00
         0 [0] 72 X      (B896)
       0.0  0 60
```

This sequence may be obeyed from programme, but re-entry to the programme may only be effected by reading an E- or J- directive; the whole computing store will have been used.

2. Initial Orders Input

```
          A  1 00
          100 [3] 72 1    (B903)  }
    A                             }  self-modified entry
          3.1  0 60               }
```

On exit (by reading an E- or J- sequence), the whole of the computing store will have been used.

3. Initial Orders subroutines

(a) *Initial Orders Subroutine*

Entry:

```
        37  X 00
        10 [0] 72 X      (B906)
       0.0  0 60
```

Exit: Link set in X1 is obeyed in 0.3 on reading an L- directive.

Uses: U0, 1, 2, 3, 4, 5; B0.

(b) *Number print*

Entries:

```
        93 [0] 72 4      (B925)  }
         N  0 60                  }  To print the integer in X7
                                  }  (except in B below)
    where 4m = 832.0              }

        92 [0] 72 4      (B925)  }
         N  0 60                  }  To print the unrounded fraction in
                                  }  X7 (except in B below)
    where 4m = 833.0              }
```

The entry address, *N*, may be one of the following:

(A) 0.6+ Print a full signed integer or fraction, (or a signed number to *n* digits, where $6_m = n\text{-}8$*).

(B) 0.2 Print 3_B.

(C) 0.3 Print 3-digit unsigned number: straight right-hand margin.

(D) 0.4 { As for (P) provided that $5_c = 11$, $5_m = 1$.
As for (Q) provided that $5_c = 11$, $5_m = 0$.

(P) 0.7 Print signed number to 2_c digits*. 2_m must be clear on entry.

(Q) 0.3+ Print unsigned number to 2_c digits*. 2_m must be clear on entry.

---

* On printing an integer $I \geqslant 10^{11}$ 12 digits will always be printed irrespective of the number of digits specified. On printing an integer, I, such that $10^{11} > I \geqslant 10^5$, a maximum of 11 digits will be printed even if the number of digits specified is greater than 11.

Exit: Link set in X1 is obeyed in 0.7.

Uses: U0; X2,3,(4),5,6,7. 4_B is always reset to 832 on exit but 4_c is not changed.

(c) *Order print*

Entry:

```
92 [0] 72 4      (B924)

0.4  0 60
```

where 4_m = 832.0

If C(3) $\geqslant$ 0, the *a*-order in X7 is printed,
C(3) $<$ 0, the *b*-order in X7 is printed.
A decimal point is printed after a "stop" *a*-order.

Exit: Link set in X1 is obeyed in 1.7.

Uses: U0,1,4,7; X1,(2),(3),5,6,7.
X2 is reset from U5.1; X3 is reset from U5.3.
On exit C(5) = (0.1,0); C(1) = 0; C(6) = 12.2^{-38}.

(d) *Date* (To print the date from B895.7, increase the serial number in B895.6 by 1 and print it.)

Entry:

```
37  X 00

44 [0] 72 X      (B940)

0.0+ 0 60
```

4_m must be set to 832.0

Exit: Link set in X1 is obeyed in 0.7.

Uses: U0; all the accumulators except X4.

(e) *E and J*

Entry:

```
37  X 00

47 [4] 72 X      (B943)

4.0  0 60
```

This routine causes entry to a programme at the address specified in 3_m (at the *a*-order if X3 is positive or the *b*-order if X3 is negative), after reading four consecutive blocks into U0 to U3 and setting the accumulators (except X1) from B0.

On exit from the E,J routine X1 contains the link which was set before entry.

If the E,J routine is entered at B943.0+ instead of at B943.0, on exit X1 will contain the self-modified link for returning to Initial Orders Input.

(f) *Address Input*

Entry:

```
81 [0] 72 4

0.0  0 60
```

where 4_m = 832.0

This routine reads one address, or two addresses separated by a minus sign, from paper tape.
If only one address is read it is left in X3 and U5.4.
If two addresses are read, the first is left in X3 and U5.3 and the second in U5.4.
In each case, a sign bit equal to 1 indicates a *b*-address.

Exit: Exit is made to U3.5, leaving the address input routine undisturbed in the computing store.

Uses: U0,1,2,5.3,5.4; all the accumulators except X4.

Appendix 7

Index to the Library

1. INTRODUCTION

This index gives brief details of all subroutines in the Pegasus Library, so that users may see what routines are available. The appropriate specifications should be studied before a routine is used.

2. LAYOUT OF THE INDEX

The first two columns of the index give the routine number and a brief description of each subroutine listed.

The third column indicates, by means of the following code, the state of each subroutine.

A = Fully tested
B = Partly tested
C = Wholly or partly written but not tested
D = Project

The fourth column, headed 'Spec', shows whether the specification and programme sheets have been printed.

S = Specification issued or in preparation.
P = Programme sheets and specification issued or in preparation.

Draft specifications are usually available for reference at the London Computer Centre.

The fifth column, headed Tapes, shows which tapes are generally issued. If there is no T in this column the relevant tape is sent when specially requested.

The sixth column, headed Store Blocks, shows the number of Main Store blocks occupied by the subroutine programme. An asterisk in this column indicates that the subroutine has an interlude obeyed during input. See section 3 below, 'Note on Interludes'.

The three columns headed 'Uses' show which Computing Store Blocks (U), Accumulators (X) and Main Store Blocks (B) are used as working space by the routine. The Main Store blocks occupied by the subroutine itself and listed in the sixth column are not included. In general the parts of the store stated to be 'used' by a subroutine are those which it may alter. For instance, X1 will *not* be listed as used if a link set before entering the routine remains unchanged in X1 on exit.

The approximate time of operation of the subroutine is given in the tenth column of the index. In order to save space a simpler but less accurate formula may be given here than in the subroutine specification. See also section 4.

The eleventh and last column of the index gives a brief explanation of the function of the subroutine. In this column the following abbreviations are used:

SP. indicates that the subroutine is self-preserving

CP. indicates that the routine is a complete programme

I.O. refers to the Initial Orders.

3. NOTE ON INTERLUDES

Certain subroutines have interludes which are obeyed immediately the subroutine has been read in. These interludes make use of storage space following the programme of the subroutine. When the work of the interlude is completed the Transfer Address is reset and the interlude can be overwritten by the next subroutine.

The length of the interlude, if any, of the last subroutine accepted must be included when the storage space used by Assembly is considered. This interlude can be overwritten after the reading of the Library Tape is completed.

Interludes may sometimes be stored on the drum beyond 127.7 if necessary, but only if this is stated in the relevant specification or in the list below.

Existing subroutines which use such interludes are:-

Subroutine	Length of Interlude (in Blocks)	Notes
R 1	4	Interlude may be stored after 127.7 Includes optional parameter list
R 3	Uses R 1 and the above interlude	
R 52	4.7	No part of interlude may be stored after 127.7
R 106	1	Interlude may be stored after 127.7
R 270	1	Interlude may be stored after 127.7
R 300	6	The first block of the interlude must be stored in an address $\leqslant$ B 125
R 400	3	Interlude may be stored after 127.7
R 402	2	Interlude may be stored after 127.7
R 600	3	Interlude always stored in B 126.0 to 128.7
R 630	4	Interlude may be stored after 127.7
R 650	4	Interlude may be stored after 127.7
R 700	3	Interlude may be stored after 127.7
R 740	9	Interlude may be stored after 127.7

4. SPECIFICATIONS

The following details are assumed in individual library specifications unless information to the contrary is given.

The overflow indicator (OVR) must be clear on entry and is left clear on exit.

The link should be set as an order-pair in X1.

The subroutine is not self-preserving; it must be brought in from the Main Store each time it is used.

Nothing should be assumed about the contents of any register stated to be 'used' by the subroutine.

The time of operation is approximate and is the time from the first order of the subroutine (inclusive) to the first order of the link (exclusive).

5. ROUTINE NUMBERS

The isolated routines, most of which are, in fact, parts of the Initial Orders, have been given numbers between 1025 and 1099. These numbers are allocated for ease of reference only and are not used by the Assembly Routine.

Standard programmes forming part of the Library but not of a type suitable for use with Assembly are given routine numbers greater than 2000. These standard programmes have been adapted for use with the 7168-word Store and have been given corresponding numbers above 7000 (i.e. the first digit of their numbers has been changed from 2 to 7). No further reference has been made in this index to the 7168-word Store routines, as most of the details are the same as those for the routines on the 4096-word Store.

Routine numbers 850-899 inclusive have been set aside to be at the disposal of individual Pegasus users. The intention is that any subroutine which a user wishes to incorporate permanently in his Library Tape but which, for any reason, is not to be included in the standard Library as distributed by Ferranti Ltd should be allocated a number in this range. These numbers differ from the range 1000-1023 in that the latter are meant to be allocated to subroutines which the user requires temporarily rather than permanently. No programmes or specifications with numbers in either of these ranges will be generally distributed.

6. ACKNOWLEDGEMENTS

FERRANTI LTD gratefully acknowledge contributions from the following organisations:-

Sir W.G. Armstrong Whitworth Aircraft Ltd.; Babcock and Wilcox Ltd.; British Iron and Steel Research Association; The De Havilland Aircraft Company; National Research Development Corporation; Queen Mary College (London); Mr. F.E. Radcliffe; Robson, Morrow and Company; Royal Aircraft Establishment; United Steel Companies Ltd.; University of Durham; University of Southampton; University College of South Wales; Westland Aircraft Ltd. (Saunders-Roe Division).

Allocation of Routine Numbers

0 - 99 Output

0 - 49 Output of numbers
50 - 99 Non-numerical output

100 - 199 Input

100 - 149 Input of numbers
150 - 199 Non-numerical input

200 - 299 Functions

200 - 219 Roots and powers
220 - 239 Exponentials, logarithms, etc.
240 - 259 Circular and hyberbolic functions and inverses
260 - 279 Other functions of one variable
280 - 299 Other functions of more than one variable

300 - 399 Operations

300 - 319 Quadrature and differentiation
320 - 339 Interpolation and curve fitting
340 - 359 Inverse interpolation, zeros of polynomials
360 - 379 Power series

400 - 499 Differential Equations

400 - 419 Ordinary differential equations, first order
420 - 439 Ordinary differential equations, linear (not first order)
440 - 459 Ordinary differential equations, other
460 - 479 Partial differential equations

500 - 599 Linear Algebra

500 - 509 General purpose schemes
510 - 529 Linear equations
530 - 549 Eigenvalues and eigenvectors
550 - 589 Other special-purpose matrix operations
590 - 599 Linear programming

600 - 699 Programmed Arithmetic

600 - 609 General-purpose aids to coding
610 - 629 Floating-point, single-length
630 - 649 Complex numbers (including floating-point)
650 - 679 Multiple precision (including floating-point and complex numbers)
680 - 699 Subroutines used by programmed arithmetic routines

700 - 799 Data Processing

700 - 719 Conversion
720 - 739 Sorting and merging
740 - 759 Standard processes
760 - 779 File organisation
780 - 799 Subroutines used by data processing routines

800 - 849 Applications

850 - 899 Individual Pegasus users' private routines

900 - 999 Miscellaneous

900 - 919 Programme Checking
920 - 939 Organisational
940 - 949 Non-numerical
950 - 969 Number Tapes
970 - 999 Miscellaneous routines

1000 - 1023 These numbers are available for temporary use by programmers
1025 - 1049 Isolated subroutines (in Initial Orders etc.)
1050 - 1099 Isolated test routines
1100 - 1199 Other test routines

R	Brief Description	State	Spec.	Tapes	Store Blocks	Uses			Time (milliseconds)	Notes
						U	X	B		
	Output									
1	Print number, general purpose	A	P	T	11*	0	-	0		SP. Prints scaled number from X1 with digit-layout. Parameter-list specifies styles of printing.
3	Page layout, using R1	A	P	T	4*	0	1,6,7	0		SP. Uses R1 to print a scaled number and supplies page-layout characters as required (parameter-list).
4	Number Print (short)	A	P	T	4	0	-	0		Prints contents of X7 as integer or fraction.
5	Double-length number print	A	P	T	7	0	-	0		Prints $(\not p q).2^{38}$ or $\not p.2^{38}$ or q.
	Fast double-length number print (7168 store)	A	P	T	7	0	-	-		Fast version of above routine to print at full speed of fast punches (7168 store only).
9	Print double-length integer	A	S	T	5	0	-	0		SP. Prints $(\not p q)$ as an integer.
10	Print double-length fraction	A	S	T	5	0,1	-	0	43c-120	SP. Prints $(\not p+2^{-38}q)$ as a fraction to up to 23 places. c is the number of characters punched.
11	Print floating-point numbers	A	P	T	11	0	7	0		SP. Prints single length binary floating-point numbers from X1 in fixed or decimal floating point.
26	Shift and Print	A	S	T	10	0,1,2,3	-	0	30(c+1)	Takes a series of numbers from the Main Store, multiplies by a scale factor and prints to a given number of significant figures. c = number of characters punched.

R	Brief Description	State	Spec.	Tapes	Store Blocks	Uses			Time (milliseconds)	Notes
						U	X	B		
35	Double-length print for Flexowriter	A	S	T	7	0	-	0		Similar to R 5 and R 42 respectively, but printing is in the code used by the five-channel Flexowriter at the London Computer Centre.
36	£.s.d. print for Flexowriter	A	S	T	4	0	6	0		
40	£.s.d. print from pence	A	S	T	3	0	-	0		Prints an amount in pence as £.s.d.
42	Signed £.s.d. print from pence	A	P	T	4	0	6	0	30*c*+19	SP. Prints an amount in pence as £.s.d. Negative amounts preceded by minus sign. *c* is the number of characters punched.
43	Mixed Radix Output	A	S	T	4	0	-	0		Prints the contents of X7 as a mixed radix number.
45	Character Print	A	S	T	4	0,1, 2,3	3,4,5, 6,7	-	210	Prints 6 6-bit characters on paper tape.
51	Steering routine for I.O. order output (See also R 1028)	A	S	T	3	0,1	-	0		Functions as warning character P with optional printing suppressed.
52	Text output and input	A	P	T	1.1*	0	-	0	30 per character	Facilitates the output of descriptive matter (e.g. English words) during the course of a programme; includes an interlude for the input of the descriptive matter.
53	Binary Punch. Steering routine for R 1033	A	S	T	1	0,1, 2,5	-	0	2 secs. per block	Punches out consecutive words from the Main Store in a form suitable for Binary Input, R 1031. Replaces R 50.
2054	Binary Punch without Transfer Address.	A	S	T	4	All	All	-	2 secs. per block	CP. Functions like R 1033 but does not put any binary T on the output tape.

R	Brief Description	State	Spec.	Tapes	Store Blocks	Uses			Time (milliseconds)	Notes
						U	X	B		
55	Printed Graph	A	S	T	1	0	-	0	30 per character	Plots y/c against x, where c is the interval.
2056	Binary Punch for Isolated Store	A			13.4	All	All	-	5½ mins.	CP. For use by Maintenance Engineers. Not a general issue.
2057	Binary Punch Programme	A	S		1	All	All	0	2 secs. per block punched	CP. Makes a binary punch of all non-zero locations in the Store.
58	Binary Punch without T.A.	A	S	T	1	0,1,2	-	0	2 secs/block	Similar to R 53 but does not punch T.A.
2059	Binary Punch Programme Mk.2	A	S	T	1	All	All	0	2 secs/block	CP. Improved version of R 2057.
	Input (See also R 1025, 1031, 1032)									
100	Input double-length integers or fractions	A	P	T	10	0,1	-	0		SP. Stores double-length numbers in consecutive pairs of Main Store locations.
101	Floating-point Input	A	P	T	11	0,1	All except 1	0.0		Floating-point numbers (each packed in one word) stored in Main Store.
102	Input Tables	A	P	T	2	All	5,6,7	0		Takes in sets of numbers, indexing first in each set. Uses Initial Orders.
103	Input Tables (Floating-point)	A	S	T	3	0,1,2	All except 1	0.0		Similar to R 102. Uses R 101.
105	Input mixed numbers	A	P	T	5	0,1	1,6,7	-		Reads double-length mixed number to (pq).
106	Modify R 105.	A	S	T	-*	-	-	-		Adapts R 105 to accept Sp as terminating character (consists solely of an interlude).

R	Brief Description	State	Spec.	Tapes	Store Blocks	Uses			Time (milliseconds)	Notes
						U	X	B		
107	Revised mixed number input	A	S	T	5	0,1	1,6,7	-		Improved version of R 105 and R 106.
112	Inner Loop Integer read	A	S	T	2	0,1	4,5,6,7	-	$5c$	SP. Reads an integer into X7. Not complete – requires steering routine.
113	Read and Scale Integers	A	S	T	1	0,1	1,4,5,6,7	-	$5c + 25$	SP. Uses R 112. Reads integer and divides by scale factor to form fraction.
116	Short Number read	A	P	T	3	0,1,2	1,5,6,7	-	$5c + 8$	SP. Reads single length mixed number $N = p'/q'$.
120	Slow Mid-point read	A	S	T	1	All	-	0	$5c + 46$	Uses Initial Orders Input. Reads number $N = 2^{-19}$ C(5.0).
121	General Single-Length Number Read	A	P	T	5	0,1	6,7	0	$5c + 54$	SP. Reads single length mixed number $N = p'/10^{q'}$
137	Read number from handswitches	A	S	T	5	0	6	0		SP. Reads to X6 signed fraction or integer tapped out on the handswitches.
140	£.s.d. input to pence	A	S	T	4	0,1	7	0	$5c + 50$	Read £.s.d. and convert to pence. c is the number of characters read.
142	Signed £.s.d. read to pence	A	P	T	6	0,1	1	0	$5c + 65$	SP. Read positive or negative £.s.d. and convert to pence. c is the number of characters read.
143	Mixed Radix Input	A	S	T	3	0,1,2	1,6,7	-		SP. Reads a mixed radix number and stores it in X7 as an integral number of units of the lowest denomination.

c = number of characters read

R	Brief Description	State	Spec.	Tapes	Store Blocks	Uses			Time (milliseconds)	Notes
						U	X	B		
148	Input £.s.d. tables to pence	A	S	T	3	0,1,2	1,4,5,6,7	0		SP. Similar to R 102. Uses R 142.
2150	Set Date	A	S	T	1	All	All	511.6 511.7		CP. Reads the date from tape and stores it in B 511.7. Clears the serial number in B 511.6.
2151	Quick replacement of Isolated Store	A			0	All	All	0 512 -639	50 secs.	CP. For use by Maintenance Engineers. Not a general issue.
195	Read Transactor Card	A	S		3	0,1,2,5	1,5,6,7	25 locations	4½ secs.	Reads a card in the transactor and stores the information in the Main Store.
	Functions									
200	Square root	A	P	T	1	0	5,6,7	-	40	Square root of $p + 2^{-38} q \geqslant 0$. Time given applies to numbers of about ¼ to ½. Time increases as $p + 2^{-38}q$ decreases.
201	Cube root	A	S	T	2	0,1	5,6,7	-	80	Cube root of $p + 2^{-38}q$. Times vary. See note to R 200.
202	Fast square root	A	P	T	2	0,1	4,5,6,7	-	30	SP. Uses more space but is faster than R 200. Time given applies to $¼ \leqslant (pq) < ½$. Time increases as $p + 2^{-38}q$ decreases.
211	Floating-point Cube root	A	S	T	3	0,1,2	1,5,6,7	-	103 - 149	$p' = \sqrt[3]{p}$. p' and p are floating-point numbers.

R	Brief Description	State	Spec.	Tapes	Store Blocks	Uses			Time (milliseconds)	Notes
						U	X	B		
215	Complex square root	A	S	T	6	0,1	All except 1	-	100 - 140	$(c + id) = \sqrt{(a + ib)}$; $(a + ib)$ double length.
220	Exponential	A	P	T	2	0	6,7	-	29	$p' = \frac{1}{4} \exp p$.
221	Logarithm, wide range	A	P	T	3	0,1	5,6,7	-	34 - 50	$p' = \frac{1}{32} \log_e (pq)$, $(pq) \geqslant 2^{-46}$. Superseded by R 224.
222	Floating-point Exponential	A	S	T	4	0,1,2	1,5,6,7	-	86	$p' = \exp p$. p' and p are floating-point numbers. Uses R 220. Superseded by R 225.
223	Floating-point Logarithm	A	P	T	2	0,1	1,4,5, 6,7	-	68	$p' = \log_e p$. p' and p are floating-point numbers. Uses R 221.
224	Logarithm, variable range	A	P	T	3	0,1	5,6,7	-	42 - 58	$p' = \frac{1}{2^n} \log_e (pq)$. n is specified by a preset parameter.
225	Shorter F.P. Exponential	A	P	T	3	0,1	1,6,7	-	71	Supersedes R 222. $p' = \exp p$. p and p' are floating-point numbers. Uses R 220.
240	Sine or cosine	A	P	T	2	0,1	1,5,6,7	-	24	$p' = \sin \pi p$ or $p' = \cos \pi p$.
241	Inverse tangent	A	P	T	4	0,1	All except 1	-	48	$p' = \frac{1}{\pi} \arctan p$ or $p' = \frac{1}{\pi} \arctan\left(\frac{p}{q}\right)$.
242	Inverse sine or cosine	A	P	T	2	0,1	All	-	130	$p' = \frac{1}{\pi} \arcsin p$ or $p' = \frac{1}{\pi} \arccos p$. (Uses R 200 and R 241).

R	Brief Description	State	Spec.	Tapes	Store Blocks	Uses			Time (milliseconds)	Notes
						U	X	B		
243	Cosine and Sine	A	P	T	3	3,4,5	1,4,5, 6,7	-	42	SP. $p' = \cos \pi p$ and $q' = \sin \pi p$.
244	Tan/Cot	A	S	T	3	0,1	1,6,7	-	29	SP. $p'/q' = \tan \pi p$.
250	F.P. Cosine and Sine	A	S	T	8	0	6,7	-	99	$p' = \cos p$ and $q' = \sin p$ where p', p, q' are floating-point numbers.
251	F.P. Arctan	A	P	T	8	0,1	4,5,6,7	-	88	$p' = \arctan p$ or $p' = \arctan \left(\frac{p}{q}\right)$ where p', p, q are floating-point numbers.
260	Complete Elliptic integrals	A	S	T	4	0,1	1,4,5 6,7	-	200	$p' = \frac{\pi}{2K}$; $q' = \frac{2E}{\pi}$.
261	Error Function	A	S	T	2	0,1	4,5,6,7	-	38	Given $p = \frac{1}{4} x$, $p' = \text{erf}(x)$.
262	Elliptic Functions	A	S	T	12	0,1,2	All	0,1	2 secs. (Average)	$x'_5 = \text{sn}(4Kq, p)$ $p' = \text{cn}(4Kq, p)$ $q' = \text{dn}(4Kq, p)$. (Uses R 200, 240, 241, 242).
270	F.P. Bessel Functions	A	S	T	34*	0	1,6	0,1	100 (Average)	SP. $p' = f_q(p)$ for $f_q = I_0, I_1, K_0, K_1, J_0, J_1, Y_0, Y_1$. p and p' are floating-point numbers. (Uses R 200, 220, 221, 225, 250).
	Operations									
300	Gaussian Quadrature	A	P	T	2* + roots & weights	0,1	5,6	0	See Specification.	Evaluates $\int_{X-h}^{X+h} f(x)dx$

R	Brief Description	State	Spec.	Tapes	Store Blocks	Uses			Time (milliseconds)	Notes
						U	X	B		
310	Differentiation of a Table	A	S	T	5	All	All	-	$30n$	n is the number of points in the table.
320	Linear Interpolation	A	P	T	3	0,1	5,6,7	-	58 - 72	$p' = f(p)$.
321	Polynomial Interpolation	A	P	T	8	0,1,2	6,7	0	About $5n^2-2n+100$	$p' = f(p)$. $(n-1)$ is the order of interpolation.
322	Two way Linear Interpolation	A	P	T	5	0,1,2	1,5,6,7	-	128 - 142	$p' = f(p,q)$.
323	Two way Polynomial Interpolation	A	P	T	10	0,1, 2,3	5,6	0	$5.2n^3+6n^2$ $+32n+170$	$p' = f(p,q)$. $(n-1)$ is the order of interpolation
324	Floating-point Interpolation	A	S	T	9	All	6,7	0	$48n^2-$ $9n+150$	$p' = f(p)$ [Floating-point] Uses R 610 $(n-1)$ is the order of interpolation
327	Polynomial interpolation (unequal intervals)	A	P	T	8	0,1, 2,3	6,7	0	$1.25m+6n^2$ $+4.5n+120$	$p' = f(p)$ m is the number of values in the table $n-1$ is the order of interpolation
328	Two way polynomial interpolation (unequal intervals)	A	S	T	15	0,1,2, 3,4	5,6,7	0,1	$5.5(u_x^2+u_y^2)$ $+18u_xu_y+75u_x$ $+45u_y+460$	$p' = f(p,q)$. u_x, u_y are the number of values of x and y used in interpolation.
332	Straight Line Fitting (asymmetrical)	A	S	T	4	0,1,4, 5	6,7	0	$11n+100$	Fits a straight line $y = a_0 + a_1x$ n = number of points given
333	Least Squares Parabola Fitting	A	S	T	9	0,1,3, 4,5	5,6,7	0	$20n+280$	Fits a curve $y = a_0 + a_1x + a_2x^2$
334	Straight Line Fitting (symmetrical)	A	S	T	4	0,1,3, 4,5	6,7	0	$13n+100$	Fits a straight line $ax + by = 2^{-v}$

R	Brief Description	State	Spec.	Tapes	Store Blocks	Uses			Time (milliseconds)	Notes
						U	X	B		
340	Zero of a Function	A	P	T	5	0, 1	6	1	See Specification	Finds a zero of $f(x)$ to within a tolerance specified as a preset parameter, starting from two approximations.
341	Linear inverse interpolation	A	P	T	4	0, 1	6, 7	0	$86 + 2.5\ r$	$f(p') = p$, (where p lies between the r^{th} and $(r+1)^{th}$ entries in a table).
342	Two way linear inverse interpolation	A	S	T	11	0, 1, 2, 3	5, 6	0		$p = f(p'q)$.
2350	Roots of Polynomials	A			95	All	All	-	$\frac{n}{2}$ sec. per iteration	CP. Double-length floating-point. Bairstow's Method. n is the degree of the polynomial.
360	Power Series Economisation	A	S	T	12	All	All	-		Used with R 650. Calculates coefficients of Chebyshev polynomials, economises a power series and prints its coefficients.
361	Evaluation of Polynomial. Floating-point	A	S	T	3	All	All	-	$121n + 31$	$F(z) = \sum_{j=0}^{n} a_j z^{n-j}$, where $z = p+iq$ and the coefficients a_j are real. Uses R 610.
2365	Fourier Series Mk. 2.	A	†			All	All	-	$\left(4 + \frac{n}{8} + \frac{8v}{3} + \frac{nv}{38}\right)$ seconds	CP. Calculates Fourier coefficients from n readings of values of the function at equally spaced points throughout the period. $n \leqslant 400$. v is the number of harmonics: $v \leqslant \left[\frac{n}{2}\right]$.

† Described in CS. 165, 166a.

R	Brief Description	State	Spec.	Tapes	Store Blocks	Uses			Time (milliseconds)	Notes
						U	X	B		
	Differential Equations									
400	Simultaneous first-order differential equations $n > 8$, $v \neq 0$	A	P	T	9*	All	3,4,5, 6,7	-	$38n+4t+100$	The solution of n first-order differential equations by a modified Runge-Kutta process. Scaling factor 2^v. Time of auxiliary = t. Times given are for one step of the integration.
401	ditto $n > 8$, $v = 0$	A	P	T	6	All	4,5,6,7	-	$22n+4t+100$	
402	Simultaneous first-order differential equations $n \leqslant 8$, $v \neq 0$	A	P	T	5*	All	5,6,7	-	$20n+4t$	
403	ditto $n \leqslant 8$, $v = 0$	A	P	T	4	All except 1	5,6,7	-	$11n+4t$	
405	As R 401, but floating-point	A	P	T	14	All	All	-	$490n+4t+300$	
407	As R 403, but floating-point	A	P	T	10	All	All	-	$480n+4t+60$	
408	Simultaneous first-order differential equations $n \leqslant 8$, $v = 0$	A	S	T	18	All	All		$19n+t+25$ ($n \leqslant 4$) $19n+t+41$ ($n > 4$)	The solution of n first-order differential equations using Simpson's formula. Time of auxiliary = t. Times given are for one step of the integration.
409	ditto $n \leqslant 8$, $v \neq 0$	A	S	T	20	All	All		18 - 34 extra to R 408	An addition to R 408 to prevent overflow of the variables. Scaling factor 2^{-v}.
411	Simultaneous first-order differential equations $n \leqslant 8$	A	S	T	11	All	1,4,5, 6,7	1	$19n+5t+44$	Similar to R 403 but more comprehensive, using Kutta-Merson process. Time of auxiliary = t. Times given are for one step of the integration.

R	Brief Description	State	Spec.	Tapes	Store Blocks	Uses			Time (milliseconds)	Notes
						U	X	B		
	Linear Algebra									
2500	Matrix Interpretive Scheme	A	S	T	107	All	All	-		A floating-point scheme to facilitate matrix manipulation.
2501	Error Tracer for Matrix Programmes	A	S	T	3.3	3,4,5	2,3	0	2 secs. per instruction	Used with R 2500. Prints matrix instructions immediately before they are obeyed.
2502	Magnetic Tape Matrix Interpretive Scheme	A	S	T	114	All	All	-		An extension of R 2500, using magnetic tape as a backing store.
2503	Double-length Matrix Interpretive Scheme	A			107	All	All	-		Similar to R 2500 but numbers are held in three words as in R 650.
2510	Solution of simultaneous linear equations	A	†	T	21	All	All	-		CP. Solves n equations with r right-hand sides. If $r=1$ then $n \leqslant 85$. If $r=n$ then $n \leqslant 50$. (4096) If $r=1$ then $n \leqslant 115$. If $r=n$ then $n \leqslant 67$. (7168)
511	Simultaneous linear equations subroutine	A	S	T	15	All	All	Depends on r	$(3n+8r) \times (n^2+7n+5)$	Solves n equations with r right-hand sides.
2530	Latent roots and vectors. $n \leqslant 54$.	A	S	T	119	All	All	-		CP. Determination, by iteration, of up to 16 roots and vectors of a matrix with real roots.
2532	Latent roots and vectors. $n \leqslant 33$.	A	S	T	128	All	All	-		CP. Similar to 2530 but rather easier to use.
7534	7168 Latent roots and vectors. $n \leqslant 54$.	A	S	T	252	All	All			CP. 7168 version of R 2532.

† Described in CS. 132, 133, 134

R	Brief Description	State	Spec.	Tapes	Store Blocks	Uses			Time (milliseconds)	Notes
						U	X	P		
2535	Latent roots and vectors. $n \leqslant 400$	A			112	All	All	-		CP. Similar to R 2530, but requires 2 magnetic tape mechanisms. Evaluates up to 24 real roots and vectors.
2536	Complex latent roots. $n \leqslant 48$	A			123	All	All	-		CP. Evaluates all the latent roots (real or complex) of a real matrix.

R	Brief Description	State	Spec.	Tapes	Store Blocks	Uses			Time (milliseconds)	Notes
						U	X	B		
	Programmed Arithmetic									
600	Autocode	A	S	T	111*	All	All		About 70 per instruction	A floating-point conversion scheme to simplify the programming of special jobs.
610	Floating-point Arithmetic	A	P	T	4	3,4,5	1,4,5, 6,7	-	18	SP. Multiple-entry routine for addition, subtraction, multiplication and division of packed floating-point numbers.
611	Floating-point Square Root	A	P	T	3	0,1	1,5,6,7	-	15 - 50	$p' = \sqrt{p}$, where p and p' are floating-point numbers.
612	Shorter F.P. Arithmetic	A	P	T	3	4,5	1,4,5, 6,7	-	14½ (+, −) 11½ (×) 29 (÷)	SP. Similar to R 610. Uses only U4 and 5. Will give results in non-standard form in rare special cases.
630	Complex Arithmetic Interpretive Scheme	A	S	T	40*	1,2,3, 4,5	1,6,7	0		Three ~~words~~ per number (1 for real argument; 1 for imaginary argument; 1 for exponent).
650	Double-length floating-point Arithmetic	A	S	T	35*	1,2,3, 4,5	1,6,7	0		Three words per number (2 for argument; 1 for exponent).
670	Arithmetic with Rational Fractions	A	S	T	5	0,1	1,4,5, 6,7	-		
	Data Processing									
700	Output Conversion	A	S	T	1*	0	3,4,5, 6,7	-		SP. Converts an integer $< 10^6$ into 6-bit characters.

R	Brief Description	State	Spec.	Tapes	Store Blocks	Uses			Time (milliseconds)	Notes
						U	B	X		
710	Input Conversion	A	S	T	2	0,1,5	1,5,6, 7	-		SP. Converts words of n 6-bit characters into n-digit integers. $n \leqslant 6$.
720	Merging Sort (Maximum String)	A	S	T	7	All	All except 1	0	See Specification	Sorts positive numbers into ascending order in the Main Store.
721	Data Sort	A	S	T	8	All	All except 1	0	"	Sorts keywords, each followed by one data word.
722	Double-length keyword Sort	A	S	T	9	All	All	0	"	Sorts double-length numbers.
723	N-length Sort	A	S	T	9	All	All except 1	0	"	Sorts N-length numbers.
730	Sort Main File (Variable length file items)	A	S		About 65	All	All	0,511		Merging process using 4 magnetic tape units.
736	Statistical Sort (Merging)	A	S	T	10	All	All except 1	0	See Specification	Sorts keywords and adds data words when keywords are equal, using a merging process.
737	Statistical Sort (Indexing)	A	S	T	4	0,1,2	1,3,5, 6,7	-	"	As R 736 but indexes keywords one by one as they are presented and sorts them later using the index.
738	Scramble Store	A	S	T	4	0,1,2, 3	1,2,6,7	-		SP. Looks up random keyword, or indexes new keyword.
740	PAYE Mk.2	A	S	T	23*	All	All	-	See Specification	Comprehensive Tax Evaluation.
741	PAYE Code conversion	A	S	T	12	0,1,2	2,7	0		Reads tax code, looks up table A and prepares special code for R 740.

R	Brief Description	State	Spec.	Tapes	Store Blocks	Uses			Time (milliseconds)	Notes
						U	X	B		
761	File Updating Mk.2	A	S		53	All	All			Updates a Main File using programmers' sub-routines for the actual amendments and insertions. Permits change in length of items during amendment. Requires at least 3 magnetic tape units.
	Applications									
2800	Pipe Stressing Mk.2	A	†		About 100	All	All			Cp. Computes stresses for a 3-dimensional pipe system with up to 8 anchors.
2801	Frame Stressing (Livesley Method) Mk.1	A	‡			All	All			CP. Performs elastic analysis of two-dimensional rigid frame of up to 18 joints and 56 members.
2802	Frame Stressing (Livesley Method) Mk.2	A				All	All			CP. Performs elastic analysis of large two-dimensional rigid frames by partitioning into sub-frames.
2810	Multiple Regression Mk.1B	A	‡			All	All			CP. ⩽ 26 variables. Correlation, regression, significance tests.
2811	Multiple Regression Mk.4	A				All	All			CP. ⩽ 38 variables. Correlation and regression.
850 -899	**Individual Pegasus Users' Private Routines**									

† Described in CS.230

‡ Described in CS.194

‡ Described in CS.273

R	Brief Description	State	Spec.	Tapes	Store Blocks	Uses			Time (milliseconds)	Notes
						U	X	B		
	Miscellaneous									
2900	Compare Tapes	A	S	T	2	All	All	-	85 characters per second	CP. Compares two tapes at high speed, stopping if they are not identical.
2901	Clear Store	A	S	T	1	All	All	All	1.6 secs.	CP. Clears whole Main Store except Date and Serial Number.
2902	Identification	A	S	T	3	All	All	All	3 secs.	CP. Writes the integers 0 to 4093 into Main Store locations 0 to 4093.
2903	Floating-point print (Non-Assembly)	A	S	T	10	All	All	-		CP. Prints floating-point numbers from the Main Store in a manner analogous to the warning character F.
2904	Clear Magnetic Tape	A	S	T	3	All	All	-	41 or 53 per section	CP. Clears sections of magnetic tape as specified by a parameter tape.
2905	Testaid Break-point	A	S	T	10	All	All	0		Aids programme development by facilitating the printing of intermediate results.
2906	Fast block transfer translation	A	S	T	3	All	All	-		CP. Translates a punch on block transfer tape more rapidly than the Initial Orders.
2907	Store Use	A	S	T	1	All	All	0	20 secs.	CP. Prints one character indicating the contents of each block in the Main Store.
2915	Independent Double-Length Fraction Print	A	S	T	7	All	All	-		CP. Prints double length fractions in a manner analogous to the warning character F.

R	Brief Description	State	Spec.	Tapes	Store Blocks	Uses			Time (milliseconds)	Notes
						U	X	B		
2921	Binary Translation Routine (See also R 1030, 1031).	A	S	T	17	All	All	-		CP. Translation routine for converting tape containing relative addresses and warning characters from I.O. notation to BINARY INPUT notation.
2922	End of Library	A	S	T	4	All	All	-		CP. Appears at end of Library Tape. Prints out missing tag functions if Assembly has failed to find everything it needs.
2923	Binary to Initial Orders Translation	A	S	T	18	All	All			CP.
7924	Convert programme for 7168 Store	A	S	T	54	All	All	All	3 secs. per block	CP. Converts a binary punched programme for use with the 7168 Store.
7925	Convert programme for 4096 Store	A	S	T	63	All	All	All	3 secs. per block	CP. Converse of R 7924.
7927	Binary to Maglib translation	A	S	T	11	All	All	See Spec.	About 0.5 secs. per block	CP. Records subroutines on magnetic tape in a form suitable for use with MAGLIB.
930	Check Magnetic Tape	A	S	T	6	0,1,2	1,6,7	0	130 - 1000	Checks write and 16/32 switches and optionally prints information about the tape.
951	Random Normal Deviates	A	S							A tape containing random normal deviates punched correct to two decimal places.
970	Standard atmosphere	A	S	T	6	0,1	1,5,6,7	-	53 or 109	Uses R 200 and R 220.
971	Triple Exponential Standard Atmosphere	A	S	T	4	0,1	1,4,5, 6,7		92 - 129	Uses R 220.

R	Brief Description	State	Spec.	Tapes	Store Blocks	Uses			Time (milliseconds)	Notes
						U	X	B		
980	Pseudo-Random Number Generator	A	S	T	1	5	1,2,6,7	-	5½	SP. Computes sequences of pseudo-random numbers. One number is produced and left in X2 each time subroutine is entered.
981	Random Numbers in Main Store	A	S	T	5	0,1,2,3,4	-	0	120 per block	Fills a chosen sequence of locations in the Main Store with pseudo-random digits.
990	Read/Write Magnetic Tape	A	S	T	4	All		0	48 or 64 per section	Transfers information from tape to the Main Store or from the Main Store to tape. Faster than R 1032 or R 1044.
2991	Copy Magnetic Tape	A	S	T	8	All	All	All	96 or 133 per section	CP. Copies information from one magnetic tape to another under control of a steering tape. Faster than R 1045.
2992	Compare Magnetic Tapes	A	S	T	7	All	All		160 or 245 per section	CP. Compares information on two magnetic tapes and prints a record of the sections which disagree.

R	Brief Description	State	Spec.	Notes
	Isolated Routines			
1025	Initial Orders (I.O.)	A		
1026	I.O. Integer Output	A		
1027	I.O. Fraction Output	A		
1028	I.O. Order Output	A		
1029	I.O. Date and Serial Number	A		
1030	Assembly	A		
1031	Binary Input	A		
1032	Read Magnetic Tape	A	†	Transfers information from tape to the Main Store. Requires steering tape.
1033	Binary Punch (See also R 53 to R 2059)	A	S	Replaces R 2050. Entered with A4 warning character. Punches out from the Main Store in a form suitable for Binary Input, R 1031. Time approximately 2 seconds per block punched.
	Isolated Routines on 7168 Store			
1040	Check Magnetic Tape	A	‡	As R 930. Checks write and 16/32 switches and optionally prints information about the tape.
1041	Magnetic Tape Assembly	A	‡	As R 1030 but with the library stored on magnetic tape.

† Described in CS.127, Addendum 1

‡ Described in CS.265

R	Brief Description	State	Spec.	Notes
1042	Read Standard Programme from Magnetic Tape	A	†	Programme called for by directive Q and routine number.
1043	7168 Read Magnetic Tape	A	†	Similar to R 1032, but different steering tape.
1044	Write to Magnetic Tape	A	†	Similar to R 1043, but transfers from the Main Store to magnetic tape.
1045	Copy Magnetic Tape	A	†	Copies sections, specified on a steering tape, from one magnetic tape to another.
1046	Print word as 6-bit characters	A	†	Prints information packed in 6-bit characters under the control of directive K.
1047	Load card Distribution and Interpretation buffers	A	‡	
1048	Load card Code Table	A	‡	

† Described in CS.265

‡ Described in CS.303

Index

00	$x' = n$	40	$x' = c$
01	$x' = x + n$	41	$x' = x + c$
02	$x' = -n$	42	$x' = -c$
03	$x' = x - n$	43	$x' = x - c$
04	$x' = n - x$	44	$x' = c - x$
05	$x' = x \,\&\, n$	45	$x' = x \,\&\, c$
06	$x' = x \not\equiv n$	46	$x' = x \not\equiv c$
07		47	

$c = N.2^{-38}$

10	$n' = x$	50	$x' = 2^{N}x$ — SINGLE-LENGTH ARITHMETICAL SHIFTS
11	$n' = n + x$	51	$x' = 2^{-N}x$ (ROUNDED) — SINGLE-LENGTH ARITHMETICAL SHIFTS
12	$n' = -x$	52	SHIFT x UP N PLACES * — SINGLE-LENGTH LOGICAL SHIFTS
13	$n' = n - x$	53	SHIFT x DOWN N PLACES — SINGLE-LENGTH LOGICAL SHIFTS
14	$n' = x - n$	54	$(pq)' = 2^{N}(pq)$ — DOUBLE-LENGTH ARITHMETICAL SHIFTS
15	$n' = n \,\&\, x$	55	$(pq)' = 2^{-N}(pq)$ (UNROUNDED) — DOUBLE-LENGTH ARITHMETICAL SHIFTS
16	$n' = n \not\equiv x$	56	$(pq)' = 2^{\mu}(pq)$; $x' = x - 2^{-38}\mu$ (NORMALIZE)*
17		57	SHIFT x UP l, THEN DOWN r 6-BIT CHARACTERS. $N = l.r$

$x' = x$ if $N = 0$

$p' = p$, $q' = q$ if $N = 0$

20	$(pq)' = n.x$	60	JUMP TO N IF $x = 0$
21	$(pq)' = n.x + 2^{-39}$	61	JUMP TO N IF $x \neq 0$
22	$(pq)' = p + 2^{-38}q + n.x$	62	JUMP TO N IF $x \geqslant 0$
23	$(nq)' = n + 2^{-38}q$ (Justify)*	63	JUMP TO N IF $x < 0$
24, 25	$q' + 2^{-38}\frac{p'}{n} = \frac{x+2^{-38}q}{n}$	64	JUMP TO N IF OVR CLEAR — AND CLEAR OVR
26	$q' + 2^{-38}p'/n = x/n$	65	JUMP TO N IF OVR SET — AND CLEAR OVR
27	$p' = 2^{X}.p + n$, $q' = 0$*	66	$x'_m = x_m + 1$. JUMP TO N IF $x'_m \not\equiv 0 \pmod 8$. (UNIT MODIFY)
		67	$x'_C = x_C - 1$. JUMP TO N IF $x'_C \neq 0$. (UNIT COUNT)*

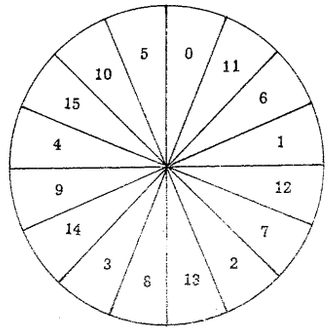
$0 \leqslant p'/n < \frac{1}{2}$; $-\frac{1}{2} \leqslant p'/n < \frac{1}{2}$; $-\frac{1}{2} \leqslant p'/n < \frac{1}{2}$

30		70	SINGLE-WORD READ TO ACCUMULATOR 1. $x'_1 = s$
31		71	SINGLE-WORD WRITE FROM ACCUMULATOR 1. $s' = x_1$
32		72	BLOCK READ FROM MAIN STORE $u' = b$
33		73	BLOCK WRITE TO MAIN STORE $b' = u$
34		74	EXTERNAL CONDITIONING*
35		75	
36		76	BUFFER ↔ COMPUTING STORE (See List) ⟶
37	$q' + 2^{-38}\frac{p'}{n} = 2^{6}q + 2^{-38}\left[\frac{2^{X}.p}{n}\right]$	77	STOP (WAIT)

NOTATION

N	First Address in Order (Register Address)
X	Accumulator Specified in Order
n	Word in N — Before Obeying Order
x	Word in X — Before Obeying Order
p, q	Words in 6 & 7 — Before Obeying Order
n', x', p', q'	Values after Obeying Order
(pq)	$= p + 2^{-38}q$, with $q \geqslant 0$
B	Block in Main Store
U	Block in Computing Store
W	Block in Buffer Store
P	Position - Number of Word in Block
OVR	Overflow - Indicator
x_m	Modifier in X, i.e. Integer Represented by Digits 1 to 13 of x
x_C	Counter in X, i.e. Integer Represented by Digits 14 to 38 of x
$(B.P,c)$	Modifier and Counter in One Word
A ---0.	Tape Order (Register 20)
0 $0fgh$	Tape Order (Register 20)
A	Section Address on Magnetic Tape
f	Number of Tape Mechanism
g	Section of the Buffer (0 or 2)
h	Type of Operation: $h = 0$ Search; $h = 1$ Read; $h = 2$ Write; $h = 3$ Rewind

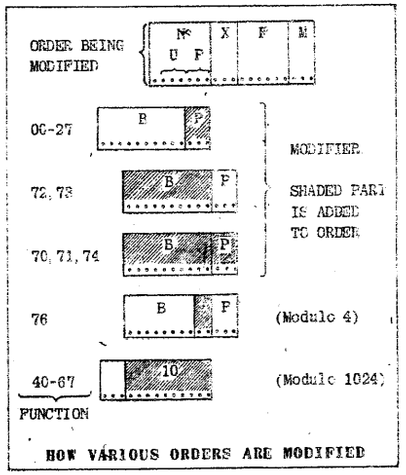
ARRANGEMENT OF BLOCKS ON EACH TRACK OF THE DRUM



TAPE	N		PRINTER	
	17	16	FIGS.	LETS.
◦	0	16	FIG.	SHIFT
◦ ●	1	1	1	A
●◦	2	2	2	B
●●	3	19	"	C
●◦◦	4	4	4	D
◦◦●	5	21	(	E
◦●◦	6	22	)	F
◦●●	7	7	7	G
●◦◦	8	8	£	H
●◦●	9	25	½	I
●●◦	10	26	=	J
●●●	11	11	-	K
◦◦◦	12	28	¼	L
◦◦◦ ●	13	13	L.F.	M
◦◦◦ ◦	14	14	Sp	N
◦◦◦ ●●	15	31	,	O
● ◦ ◦	16	0	0	P
● ◦ ◦ ●	17	17	>	Q
● ◦ ● ◦	18	18	≥	R
● ◦ ● ●	19	3	3	S
● ● ◦ ◦	20	20	→	T
● ● ◦ ●	21	5	5	U
● ● ● ◦	22	6	6	V
● ● ● ●	23	23	/	W
●◦◦	24	24	×	X
●◦◦ ●	25	9	9	Y
●◦● ◦	26	10	+	Z
●◦● ●	27	27	LET. SHIFT	
●●◦	28	12	.	.
●●◦ ●	29	29	π	?
●●● ◦	30	30	C.R.	&
●●● ●●	31	15	*	±

SPECIAL REGISTERS

15	Handswitches $H_0, H_1, \ldots, H_{19}$
16	Checked Input/Output Tape Reader → 16_C → X_C x_C → 16_C → Output Punch
17	Direct Input/Output Tape Reader → 17_m → X_m x_C → 17_C → Output Punch
20	Tape Control Word
24	External Conditioning Setting in 24_C
32	-1.0
33	$\frac{1}{2}$ = (512.0,0)
34	2^{-10} = (1.0,0)
35	2^{-19} = (0.1,0)
36	2^{-16} = (1 in Tape Address)
37	$\frac{7}{8}$ = (896.0,0)
48	Card reader fault register
49	Card punch fault register
52	Tape reader parity failure (7 track)
53	Fast punch check register
54, 55	Control registers for pseudo off-line working



HOW VARIOUS ORDERS ARE MODIFIED

* NOTES ON THE ORDER-CODE

23 Assumes that any overflow is due to operations in 7. Clears OVR unless n' overflows. $q' \geqslant 0$.

27 If $N = 7$, $p' = 2^{X}.p + q.2^{-30}$, $q' = 2^{6}.q$ (logical shift)

52 Fast shift for $N \geqslant 25$.

56 Either (i) $\frac{1}{2} \leqslant (pq)' < \frac{1}{2}$ and $-1 \leqslant \mu \leqslant N-1$,
or (ii) $-\frac{1}{2} \leqslant (pq)' < -\frac{1}{4}$ and $-1 \leqslant \mu \leqslant N-1$,
or (iii) $-\frac{1}{4} \leqslant (pq)' \not< \frac{1}{2}$ and $\mu = N-1$.

67 There is no carry from x_C to x_m.

74 If N even, main tape reader selected.
If N odd, second tape reader selected.

N-Address	76-Order
0-3	Interchange
8-11	Read Buffer
16-19	Write Buffer
24-27	Write code table
64-65	Read card
72-73	Read card buffer
80-81	Punch card
88-89	Write card buffer
96-97	Card recovery
104-105	Read card input not equivalent buffer
112-115	Write to input distribution buffer
120-123	Write to output distribution buffer

SUMMARIZED PROGRAMMING INFORMATION