34-2

The Computer Journal

Volume 1

April 1958
to
January 1959

REPRINTED 1970 FOR
Wm. DAWSON & SONS LTD., LONDON
with the permission of the Publishers

Further Autocode Facilities for the Manchester (Mercury) Computer

by R. A. Brooker

Summary: This article is a sequel to an earlier paper,* part of which described the basic facilities of the simplified programming system (Autocode) developed for the Ferranti Mercury Computer at Manchester University. The present article describes further facilities of the system and, together with the earlier description, amounts to a compact programming manual for the Manchester Mercury Computer.

UNROUNDED ARITHMETIC

In those arithmetical instructions involving a *variable expression* on the right-hand side, each sum, difference and product is formed to a maximum precision of 29 binary digits and then rounded off by making the last digit odd. If required, the rounding operation can be omitted by using the $\approx$ sign instead of $=$. In this case the result will normally be biased, but if the quantities involved can be expressed in at most 29 significant binary digits, then the $\approx$ sign provides a means of performing exact arithmetical operations on variables.

Rounding errors will be significant when using the *int pt* and *fr pt* instructions. Thus for example:

if $x = 3 + 2^{-27}$, then int pt (x) gives 3
and fr pt (x) gives 2^{-27}
if $x = 3 - 2^{-27}$, then int pt (x) gives 2
and fr pt (x) gives $1 - 2^{-27}$
if $x = 3$ precisely, then int pt (x) gives 3
and fr pt (x) gives $0 \cdot 2^{-256}$.

However, precise whole numbers can only be obtained as a result of unrounded ($\approx$) arithmetical operations (which include simple transfers, *mod* and *int pt*, but do *not* include division in any form) on numbers originally read in (see Note 4) as integers.

If the "nearest integer to x" is intended, it is sufficient to use (say)

$$y = \text{int pt}\,(x + 0\cdot 5)$$

unless x itself is half an odd integer, in which case the result will again depend critically on rounding errors.

MISCELLANEOUS INSTRUCTIONS

The instructions

$$i = \max\,(x_0, n, m)$$
$$i = \min\,(x_0, n, m)$$

may be used to determine the index of the maximum (or minimum) element in the range

$$x_n, x_{n+1}, \ldots, x_m.$$

Here i denotes any index, x_0 any first member, and n, m are indices or whole numbers. If there is no unique maximum (or minimum) element, then the least index is taken. The instruction $y = \phi_3(n)$ is equivalent to $y = (-)^n$. Here y is any variable, and n stands for any *index expression.*

STEP-BY-STEP INTEGRATION OF DIFFERENTIAL EQUATIONS

Special facilities are provided for the integration of differential equations. The equations must be written in the form

$$f_i = \frac{dy_i}{dx} = f_i(x, y_1, y_2, \ldots y_n) \qquad (i = 1, 2, \ldots n)$$

involving the special variable x, the main variables y_i, f_i and the index n. In addition the special variable h is used for the step length, and the main variables g_i, h_i, $(i = 1, 2, \ldots n)$ are introduced for "working space." The programmer must write a subsequence for calculating the f_i in terms of x and the y_i; this must not alter y_i, g_i, h_i, n, h, or x. The entry should be labelled, and the sequence should terminate with the special† instruction *592, 0*. With these arrangements the effect of the instruction

int step (m)

(where m is the entry to the subsequence) is to advance the integration by one step so that the initial and final values of the independent and dependent variables are respectively

$$\begin{array}{cc} x & x + h \\ y_i(x) & y_i(x + h). \end{array}$$

The method employed is that of Runge–Kutta, with a truncation error of $0(h^5)$. However, the truncation error also depends on the higher derivatives of the function, and for this reason the step length may be adjusted between steps if desired. The time per step is $(10n + 4T)$ msec, where T is the time (in msec) of the subsequence.

* *The Computer Journal*, Vol. 1, p. 15.

† Mercury programmers will recognise this as an ordinary machine instruction. Subject to certain restrictions it is possible to include such instructions in an Autocode program, but this is a topic beyond the scope of the present article since it involves a detailed knowledge of conventional Mercury programming. The interested reader may refer to the paper by Fotheringham and Roberts on p. 128.

Example

Integrate the two equations:

$$\frac{dy_1}{dx} = y_1^2 - 1{\cdot}23y_1y_2 + 2{\cdot}47y_2^2$$

$$\frac{dy_2}{dx} = 1{\cdot}01y_1^2 - 0{\cdot}84y_1y_2 + 1{\cdot}59y_2^2$$

for $x = 0(\cdot 02)1$,
with the initial conditions:

$$(a)\ x = 0;\quad y_1 = 0;\quad y_2 = 1$$
$$(b)\ x = 0;\quad y_1 = 1;\quad y_2 = 0.$$

```
        Program                       Notes
   chapter 1                 ⎫
    f → 2                    ⎪
    g → 2                    ⎬ main directives
    h → 2                    ⎪
    y → 2                    ⎭
1)  n = 2                    ⎫ sets the number of
    h = 0·02                 ⎪   equations, the step
    x = 0                    ⎬   length and the first
    y1 = 0                   ⎪   initial conditions
    y2 = 1                   ⎭
2) int step (3)                advances the step
   newline                   ⎫
   print (x, 1, 2)           ⎪
   space                     ⎬ prints with layout
   print (y1, 2, 4)          ⎪
   print (y2, 2, 4)          ⎭
   jump 2, 0·99 > x            repeat until x = 1
   restart                     prepare to read the
                                 second set of initial
                                 conditions
3) f1 = y1y1 − 1·23y1y2      ⎫
        + 2·47y2y2           ⎪
   f2 = 1·01y1y1 − 0·84y1y2  ⎬ subsequence to calcu-
        + 1·59y2y2           ⎪   late f1, f2
   592, 0                    ⎪
   close                     ⎭
.........................
   across 1/1                ⎫ first chapter 0
   close                     ⎭ starts the calculation
.........................
   x = 0                     ⎫
   y1 = 1                    ⎪ second chapter 0
   y2 = 0                    ⎬ sets second initial
   across 2/1                ⎪   conditions
   close                     ⎭
.........................
   stop                      ⎫ third chapter 0
   close                     ⎭ stops the machine.
```

This example introduces the instruction

restart

which is used to read in further instructions. These will usually be confined to chapter 0, as in this case, and used to re-enter an existing program. The *restart* instruction may also be used, however, to read in a fresh program.

MULTI-CHAPTER PROGRAMS

```
  chapter 1
┌──────────────┐
│  directives  │
└──────────────┘
┌──────────────┐
│ instructions │
└──────────────┘
    close
  chapter 2
┌──────────────┐
│  directives  │
└──────────────┘
┌──────────────┐
│ instructions │
└──────────────┘
    close
  chapter 3
┌──────────────┐
│  directives  │
└──────────────┘
┌──────────────┐
│ instructions │
└──────────────┘
    close
  across ?/?
    close
```

The diagram shows the general layout of a program involving three chapters. Each chapter has its own labelling system, and to jump from one chapter to another an *across* instruction may be used: for example, one of the form

across 2/3

meaning "jump to the instruction labelled 2 in chapter 3." Since this is a rather long instruction (100 msec) it is preferable not to include it within an inner loop and, as far as possible, partition into chapters should correspond to distinct parts of the calculation.

"chapter 0" has two special features:

(1) It is entered automatically, at the first instruction, when the *close* directive is read.
(2) The directives used are those of the chapter last read, *unless* chapter 0 is actually headed "chapter 0," in which case the directives must be formally set, either in the usual way or by means of the *variables* directive explained in the following section.

VARIABLE DIRECTIVES

The main variables must be defined at the beginning of each chapter either explicitly or, if it is a case of repetition, by means of a directive of the form

variables 2

meaning "use the directives set in chapter 2"—which of course may have been those used in chapter 1. It may be necessary to appreciate the significance of the variable directives. Thus

chapter 1

$$\left.\begin{array}{l} a \to 99 \\ b \to 99 \\ c \to 99 \end{array}\right\} \begin{array}{l} \text{allocates} \\ \text{memory} \\ \text{locations} \end{array} \left\{\begin{array}{rl} 0\text{–}\ 99 & \text{to } a_0\text{–}a_{99} \\ 100\text{–}199 & \text{to } b_0\text{–}b_{99} \\ 200\text{–}299 & \text{to } c_0\text{–}c_{99} \end{array}\right.$$

chapter 2

$$\left.\begin{array}{l} c \to 99 \\ x \to 49 \\ y \to 49 \end{array}\right\} \begin{array}{l} \text{allocates} \\ \text{memory} \\ \text{locations} \end{array} \left\{\begin{array}{rl} 0\text{–}\ 99 & \text{to } c_0\text{–}c_{99} \\ 100\text{–}149 & \text{to } x_0\text{–}x_{49} \\ 150\text{–}199 & \text{to } y_0\text{–}y_{49}. \end{array}\right.$$

Thus the "c's" of chapter 2 are not those of chapter 1; if it is intended that they should be, then the latter directives might be recast as follows:

$x \to 49$
$y \to 49$
$\pi \to 99$ (waste)
$c \to 99$

A further consequence of this scheme is that "x_{50}" is identical with y_0, "x_{51}" with y_1, and so on. Such "overlapping" references are sometimes useful.

THE AUXILIARY VARIABLES

To supplement the 495 working variables there are approximately 10,000 auxiliary variables to which indirect access is possible, the exact number depending on the requirements of the program.* Such access is a time-consuming operation, however, and should only be used when the working variables are insufficient for the problem in hand.

In order to refer to these variables, fourteen new working variables

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

are introduced. These are essentially similar to the special variables $a, b, \ldots, z$ and indeed may be used as such if desired. It is intended, however, that these variables will be used to designate groups of auxiliary variables. This is done by means of instructions in the program proper (unlike the main variables which are described by means of directives) which set these variables to give the address within the auxiliary stores (0, 1, . . ., 10,751) of the first member of an *auxiliary group*.

Thus, for example, the instructions

$$\begin{array}{lcl} a' \approx 0 & \text{or} & a' \approx 0 \\ b' \approx 100 & \text{alternatively} & b' \approx a' + 100 \\ x' \approx 200 & & x' \approx b' + 100 \\ y' \approx 1{,}200 & & y' \approx x' + 1{,}000 \end{array}$$

designate two groups a', b' of 100 variables, a group x' of 1,000 variables, and a group y' of unlimited extent. In contrast to the main variable directives the quantity on the right-hand side is the *starting-point and not the extent* of the group, although in both cases the extent is irrelevant if overlapping is allowed. Note that the $\approx$ form of the arithmetical instructions has been employed since the quantities in question are whole numbers.

To specify a particular member of a group (say x') the notation

x'_{750} or x'_u [one dimensionally: $x'750$, $x'u$]

is employed, where the suffix is essentially a *variable* (an index cannot be used because there is no limit to the size of a group).

The instruction $\phi_1(x'_{750}, a_1, 10)$ is an example of an *auxiliary transfer*. It means "transfer 10 consecutive variables from the auxiliary store to the working store, starting at x'_{750} in the former, and at a_1 in the latter." As a result $a_1, a_2, \ldots, a_{10}$ are replaced by $x'_{750}, x'_{751}, \ldots, x'_{759}$. The instruction $\phi_2(x'_{750}, a_1, 10)$ corresponds to the reverse operation, namely a transfer from the working store to the auxiliary store. In these instructions x'_{750} may be replaced by any particular auxiliary variable, e.g. x'_u, y'_0, and a_1 by any main variable, e.g. a_0 or $b_{(i-1)}$. The third parameter 10 is essentially an index quantity and may be replaced by an index letter.

* The number of auxiliary variables is given by 10,751 − 512n, where n is the highest chapter number. Up to 3,072 further variables can be obtained subject to certain restrictions on the program. (Details available on request.)

Examples of auxiliary transfers are

$$\phi_1(x'_0, a_0, 400)$$
$$\phi_2(x'_g, b_i, i)$$
$$\phi_2(x'_{h_i}, c_{(r-1)}, t)$$

An exception to the rule that the second parameter must be a main variable is provided by the instruction

$$\phi_1(x'_{750}, a, 1)$$

which replaces the special variable a by x'_{750}.

The execution time for a group transfer cannot be given very precisely but is less than

$$\{17[n/32] + 34 + 0.36n\} \text{ msec.}$$

where [ ] denotes "integral part of," and n is the number of variables transferred.

SUBCHAPTERS

At any point within a chapter it is possible to call in a "subchapter" and subsequently to return to the original chapter at the instruction following the point of departure. This is done by means of a *down* instruction in the main chapter and an *up* instruction in the subchapter. For example

down 2/3

calls in chapter 3 as a subchapter and enters it at the instruction labelled 2. When the subchapter has completed its task the single word instruction

up

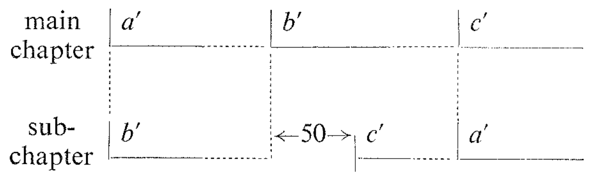
will return control to the main chapter at the instruction following the original *down* instruction. Alternatively the *up* instruction may be used in any chapter reached by means of *across* instructions from the original subchapter. A subchapter may have its own sub-subchapter but there the regression stops.

THE *reset* INSTRUCTION

If necessary the working variables (i.e. main and special) can be preserved during the operation of the subchapter and subsequently restored on return to the main chapter. This can be done by means of the single-word instruction *reset* used before the *down* instruction: the variables are "dumped" on the magnetic drum and subsequently restored by the corresponding *up* instruction. As a consequence, any results calculated by the subchapter would have to be stored as auxiliary variables if they were to be preserved. However, if subchapters are designed to select from and return to the drum all relevant material, they provide a convenient means of arranging a calculation for possible future use as a "subprogram." The subchapters can be written without defining the auxiliary variables. Instead the special variables a', b', c', etc., can be defined in terms of those used in the main chapter, by including appropriate instructions between the *reset* and *down* instructions. Since they follow the *reset* instruction the original values of a', b', c', etc., will be restored on returning to the main chapter. In this way each "level" of organisation may use its own frame of reference for the auxiliary variables. As an example suppose that both the main chapter and the subchapter employ three groups of auxiliary variables denoted in both cases by a', b', c', and starting relative to each other as follows:

$$\begin{aligned} a'_{(sub)} &= c'_{(main)} \\ b'_{(sub)} &= a'_{(main)} \\ c'_{(sub)} &= b'_{(main)} + 50. \end{aligned}$$

These relations may be represented diagrammatically thus:



The instructions for calling in the subchapter and redefining the auxiliary variables are then as follows:

$$\begin{aligned} &\text{reset} \\ \pi &\approx a' \\ a' &\approx c' \\ c' &\approx b' + 50 \\ b' &\approx \pi \\ &\text{down } ?/? \end{aligned}$$

The special variable π has been used as a "shunting station" since, in any case, it will be reset on entering the new chapter (π is automatically reset to $3 \cdot 14159 \ldots$ as a result of the instructions *across*, *down*, *up*, and *reset*).

OPERATIONS WITH COMPLEX NUMBERS

Operations with complex numbers written as number pairs are provided by instructions of the form

$$\begin{aligned} &(u, v) = (x, y) \\ &(u, v) = (x, y) + (a, b) \\ &(u, v) = (x, y) - (a, b) \\ &(u, v) = (x, y) * (a, b) \\ &(u, v) = (x, y) \ / \ (a, b) \\ &(u, v) = \text{sq rt}\,(x, y) \qquad (v > 0) \\ &(u, v) = \log\,(x, y) \qquad (\pi > v > -\pi) \\ &(u, v) = \exp\,(x, y) \end{aligned}$$

Further functions may be added. Here u, v; x, y; a, b denote any variables or (except in the case of u, v) signed constants.

Examples of instructions in this class are:

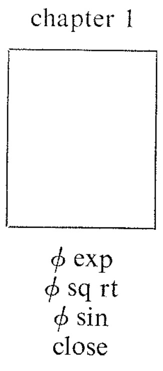
$$\begin{aligned} &(f_i, g_i) = (f_{(i-1)}, g_{(i-1)}) + (f_{(i+1)}, g_{(i+1)}) \\ &(x, y) = \text{sq rt}\,(1, 1) \\ &(a, b) = \log\,(-0 \cdot 5, h) \end{aligned}$$

QUICKIES

Every time one of the functions

sq rt	cos	log	tan
radius	sin	exp	arctan

is referred to, 17 msec are spent in transferring the necessary set of instructions (subroutine) from the magnetic drum to the instruction store. Provided there is room, however, they can be included in the chapter itself, and in this case the average execution time is reduced from about 23 msec to 6 msec. Functions treated in this way are known as *quickies*. All that is necessary is to list them (each preceded by ϕ) in order of preference at the end of the chapter in question, immediately before the *close* directive (as in the accompanying diagram). Any functions for which there is not room will be treated in the usual way.



As regards the complex functions, these involve the use of certain real functions, as follows:

$$\text{complex}\begin{cases} \text{sq rt} & \text{involves} \quad \text{sq rt} \\ \exp & \text{involves} \quad \exp, \sin, \cos \\ \log & \text{involves} \quad \log, \arctan. \end{cases}$$

Thus in order to treat the complex functions as quickies, it is sufficient to list the relevant real functions.

Finally, it should be mentioned that *sin* and *cos* involve the same set of instructions, so that if either one is treated as a quicky the other will be also. The same applies to *sq rt* and *radius*.

NOTES

1. Printing: all numerical values are rounded off by adding $\frac{1}{2} \cdot 10^{-m}$, m being the number of decimal places specified.
2. In the case of numbers $\geqslant 10^{16}$ an asterisk only is printed.
3. The ? print facility does not apply to the complex-number instructions.
4. Numbers may be read either directly, by means of a *read* instruction, or implicitly as constants on the right-hand side of an equation. In either case the number of digits after the decimal point is limited to 24 (no limit on integral part).
5. The description of the *arctan* function (given in the earlier article) is incomplete. The function calculates an angle in the range $(-\pi, \pi)$, the quadrant being determined as if x, y were proportional to the cosine and sine of the angle respectively.

ACKNOWLEDGEMENTS

The author is indebted to Dr. B. Richards for advice and assistance at various stages of the work, and has also benefited from discussions with various friends and colleagues, in particular Dr. J. Howlett of A.E.R.E., Harwell, and Dr. G. E. Thomas of the Central Instrument Laboratory, I.C.I., Ltd.

An Input Routine for the Ferranti Mercury Computer

by J. A. Fotheringham and M. de V. Roberts*

Summary: The principal features of an input routine for general use with the Ferranti Mercury Computer are described. The method of writing programs for input by means of this routine is given, and the reasons for including the various facilities are discussed. One of the main features is provision for the liberal use of "symbolic addresses."

1 DESCRIPTION OF THE MACHINE

The Ferranti Mercury Computer has been described elsewhere (Lonsdale and Warburton, 1956), but it seems worth while to outline the basic features of the machine here.

Mercury is a single-address, floating-binary-point computer with a computing store of 2,048 20-bit registers (using ferrite cores) and a backing store of up to 65,536 registers (on up to 4 magnetic drums). A register can hold a single instruction consisting of 7 function digits, 3 B digits and 10 address digits. However, instructions can only be obeyed from the first half of the high-speed store. Numbers occupy pairs of registers, 10 bits being reserved for the exponent and the remainder used for the mantissa; the first register of each pair must have an even address. Numbers can be held throughout the high-speed store. Thus the store can hold 1,024 instructions and 512 numbers, no instructions and 1,024 numbers, or any intermediate combination.

The B-registers hold 10-bit integers, and although their main purposes are counting and address modification, generous facilities are provided for arithmetic operations on their contents. An integer from a B-register can be stored in either half of a storage register without disturbing the other half, but such access to half-registers is restricted to the first half of the computing store.

The drums all rotate synchronously with a period of approximately 17 msec. Information is stored on "sectors," each capable of containing 32 numbers (or 64 instructions). There are 1,024 such sectors, two to each track on a drum. Information is transferred between the drum and the high-speed store in units of one sector. In the high-speed store, the transfer must be started at a register with an address which is a multiple of 64; it has therefore become general to think of the high-speed store as divided into "pages" numbered from 0 to 31. A transfer is therefore between a sector on the drum and a page in the high-speed store. This has led to an alternative method of addressing a register, namely in "page and line" form. Thus register 129 may be referred to as page 2, line 1, or briefly as register 2.1.

2 CHOICE OF PROGRAMMING PHILOSOPHY

Before writing an input routine, one has to decide which of several existing philosophies to adopt. There still exists some sympathy with primitive schemes, in which what appears on the program sheets resembles very closely what exists in the machine. This approach becomes desirable if it is intended to make considerable use of the cathode-ray tube monitors when de-bugging a program. However, the adoption of such a scheme usually makes the writing of a program somewhat tedious. In the case of Mercury, which has a ferrite core store, registers can be monitored only singly and while the computer is stopped; hence the primitive approach was rejected.

At the other extreme are the "automatic coding" and similar ambitious projects. It is often stated that such schemes make programming so easy that a beginner can be taught in a day, and that mistakes are seldom made. However, they have the disadvantage that if a mistake is made it may be difficult to find unless a comprehensive post-mortem routine is provided, which may be even more elaborate than the routine that performs the initial translation. Although automatic coding is highly suitable for a number of jobs, it was felt that the standard scheme for general use should not go too far in this direction. It should be mentioned, however, that an excellent "Autocode" scheme for Mercury has been prepared by R. A. Brooker of Manchester University, and is expected to be extensively used, particularly for one-off jobs (see Brooker, 1958).

The following compromise was finally adopted. Programs for Mercury will, for the most part, be written instruction by instruction. Thus the programmer can satisfy himself that he is obtaining the maximum efficiency where necessary. However, the assembly of the complete program, the insertion of appropriate addresses, and most of the other irritating clerical tasks are performed by the input routine. Satisfactory post-mortem routines accompanying this scheme are not unduly complicated.

3 INSTRUCTION NOTATION

For engineering reasons, the function code of the machine is not arranged in a way that is easy for a programmer to learn. Hence it was decided to represent each function by means of a pair of decimal digits chosen for easy memorizing. The various functions may thus be conveniently tabulated in a matrix array. The size of the matrix is 10 × 10, and each function is specified by the pair of decimal digits defining the row and the column containing the function. The arrangement of the functions is such that, for example, all the operations on the floating-point accumulator that involve addition appear in the row which corresponds to the first function digit 4, etc. The relation between the programmer's function code and that of the machine itself is unfortunately not simple.

There is a similar complication in the case of addresses.

* Dr. Roberts is now with IBM Corp., New York.

The ten address digits are clearly not sufficient to specify a register anywhere in the high-speed store. However, as a number must occupy a pair of registers the first of which has an even address, ten digits can be made to serve the purpose in this case by storing the address divided by two. (The function part of the instruction, being of a type referring to numbers, will imply that the address is in this form.) In the case of jump instructions, addresses greater than 1,023 are forbidden in any case, since instructions cannot be obeyed from the second half of the store, so that the address in a jump order may be represented and need not be halved.

For operations involving half-registers (such as operations on B-registers), in order to accommodate the odd "half" in the address, the latter is stored multiplied by two. This, however, may cause an overflow, and in the case of such instructions, one of the function digits is borrowed to act as an extension of the address. Again, the function digits will indicate that this is an instruction of such a type that this situation obtains. There are therefore effectively three different address systems in Mercury, one for half-registers, one for registers, and one for register pairs.

While provision has been made for a programmer to work in terms of these three systems if he wishes, it was thought that normally a unified address system would be preferable. This system is that referring to single registers, and normally all addresses are written in this way.

The input routine examines the function part of each instruction to determine how the address part should be stored. In the case of half-registers, however, it is then also necessary to specify the half of the register to which the instruction is intended to refer. The programmer indicates this by inserting the symbol + following the address if the right-hand half is required, and omitting this symbol if the left-hand half is required. As has been mentioned earlier, addresses may be written in "page and line" form as an alternative to the normal consecutive decimal numbering of the registers. Negative addresses are also allowed, and are interpreted modulo 1,024. The reason for this will be seen below.

Between the function and address parts of each instruction, a B-digit is always inserted. The value 0 indicates that no B-register is involved; otherwise the value of the digit (from 1 to 7) specifies which B-register is involved.

As an example of a simple instruction, the following will cause the number in registers 2,046 and 2,047 to be copied into the accumulator.

	400	2,046
or	400	31.62
or	400	30.126, etc.

4 ROUTINES AND SYMBOLIC ADDRESSES

Regardless of the characteristics of the computer employed, the concept of a *routine* is basic to the whole art of programming. It is natural to break the problem up into self-contained sections and to concentrate attention on these individually. Thus one of the chief aims in designing the input routine has been to make the preparation of routines and their assembly into a complete program as easy as possible. The programmer is relieved as far as possible of the tedious clerical and organizational tasks. In general it is possible to set about writing a routine for Mercury without having to consider its ultimate position in the machine or, within wide limits, its length.

The most important step taken in this direction was the decision to adopt a flexible system of symbolic (or "floating") addresses. The basic philosophy behind symbolic addresses is that when a reference is made to some item a from an instruction b in another part of the program, then rather than referring in b either to the absolute address of a in the fast store or to the address of a relative to the start of its routine, a is labelled and the address in b refers to the label. This facility avoids the need for numbered program sheets; in fact, ordinary squared paper is recommended for Mercury programming. Furthermore, if a correction has later to be inserted, no address has to be changed.

This idea has been discussed elsewhere (Wilkes, 1953), but there are several points that are worth mentioning in the case of Mercury. For Mercury, items of information are labelled by terminating them with the symbol (followed by an integer. The address part of any instruction in the same routine referring to this item is then written as v followed by the same integer. Suppose, for example, that it is required to instruct the machine to copy into the accumulator a constant, say $-2\cdot 67$, held in another part of the routine. The necessary function for this is 40. Thus the instruction

400 $v1$

will have the effect of copying $-2\cdot 67$ into the accumulator, provided that somewhere in the routine

$-2\cdot 67$ (1

has been inserted.

In a long program, if all the symbolic addresses are to be kept distinct from one another, they tend to become long and unwieldy. A further difficulty that arises is that if two routines which have been written on different occasions are brought together in the same program, it may turn out that the same symbolic addresses have been used in both. These difficulties have been overcome in Mercury by associating the symbolic addresses with routines rather than with complete programs. Symbolic addresses, in the simple form described above, only refer to labels within the same routine. Up to 64 labels are allowed within a routine (and as many references to them as are necessary, which may well be more than 64). Each routine is given a number which serves to identify it. (A routine is numbered simply by writing at its head the word ROUTINE, or just R, followed by its number, which is a small integer.) If it is required to refer to a label which appears in another routine, the symbolic address is followed by the number of the routine containing the label, and is separated from it by the symbol /.

To illustrate this, suppose that in the last example the constant had been contained in routine No. 10. Then in any other routine the instruction

400 v1/10

would have served to copy the constant into the accumulator. This facility is far more often used for jumps between routines than for collecting constants, but nevertheless this example is valid.

Provision has been made for such routines to be numbered up to 999. This is, of course, a very much larger number than will ever be used in one program; however, it gives a useful degree of latitude in the numbering scheme. There is an overall limit of 1,024 on the total number of different symbolic addresses that may be used in one program.

5 THE ORGANIZATION OF A COMPLETE PROGRAM

When all the routines for a program have been written, they have to be assembled and arranged within the machine. The important consideration at this stage is the best possible use of the computing store. It was felt that if only one routine was held in the high-speed store at a time there would be certain programs for which this system would be very inefficient. A part of the computing store would have to be reserved throughout for numerical data, and it would be difficult to provide adequately for a short routine operating on a large set of data (such as pivotal condensation of a large matrix) and a long routine performing many operations on relatively little data within the same program. While the object of the input routine was to simplify the clerical work required from the programmer, this should not be at the expense of the speed at which the compiled program would run.

On the other hand, an input program which assessed the space and data required by each routine and planned the routines to fit into the computing store in the best possible way would be almost an autocode. The compiling would need to take account, among other things, of which routines used other routines and how often. It was felt that this part of the organization of a program was best left to the programmer himself.

The programmer, then, has to collect his routines into groups, and in general one of these groups will be held in the computing store at a time. The groups have been given the name of CHAPTERS. The length of a chapter will be decided principally by the amount of data it requires in the fast store, and it may contain one or several routines.

The start of a new chapter is indicated by the heading CHAPTER on the program tape. The assembly of routines within a chapter is done as economically as possible by the input routine, and the chapter is then stored on the drum on an integral number of sectors. Consecutive chapters from the input tape will normally be stored in consecutive sectors, unless the programmer indicates otherwise.

Each CHAPTER heading on the program tape is followed by a serial number by which that chapter can be identified. The input routine keeps a list of the sectors at which the chapters start, and a list of which chapter contains each routine of the program. The number of chapters in a program is limited to 63, but it is expected that many programs will only be one or two chapters long.

The input routine is designed on the assumption that one page of the computing store is reserved throughout the running of the program for working space and for a short sequence of instructions by which new chapters are brought automatically into the computing store when required. This set of instructions is known as the Chapter Changing Sequence, and it occupies the first half of page 0. The programmer indicates his desire to change chapters by punching ACROSS followed by a symbolic address which specifies a routine and the point at which it is entered. The input routine interprets this from the lists it has kept and inserts the correct instructions in the program; these instructions transfer control to the Chapter Changing Sequence, which brings into the computing store the whole chapter containing the required routine and enters it at the specified address.

A refinement of this facility has been included which enables the programmer to use a chapter (or part of a chapter) as a closed subroutine. A chapter used in this way is called a sub-chapter. If the word DOWN is used instead of ACROSS, the Chapter Changing Sequence stores a link to re-enter the original chapter at the instruction after the DOWN. At the end of the sub-chapter the word UP is punched on the tape; this is translated during input into instructions which cause the Chapter Changing Sequence to obey this link.

All the programmer has to do, therefore, is to arrange his program in chapters. The input routine will then store them and arrange for their transfer to the computing store at the appropriate times, also printing out, if required, where on the drum each chapter starts, and to which part of the computing store each chapter and routine will be brought. These locations will normally be arranged automatically by the input routine, but any or all of them can be specified on the program tape if required.

6 COMMON SUBROUTINES

Mercury has a fairly powerful order code, but basic functions such as exponential and cosine, and input and output subroutines, have to be programmed. A frequent practice is to keep a library of program tapes for these functions, and to copy the required subroutines on to the final input tape, adding to the labour involved in preparing tapes. In Mercury, subroutines for the most frequently used of these functions are permanently stored on the drum, and can be automatically inserted in the program during input. They have come to be called *quickies*, and provision has been made for thirty quickies, of which the first twenty will be standard and the remaining ten reserved for the particular requirements of individual users.

The quickies are open subroutines which (apart from those concerned with input and output) replace the content of the accumulator by some function of itself. To incorporate a quicky in his program the programmer simply writes QUICKY (or Q) followed by the number of the quicky required; the input routine then copies the quicky in at that point. For example, Q4 replaces the content of the accumulator by its exponential. The function codes for copying a number into and from the accumulator are 40 and 41 respectively, so that to replace the number in location N by its exponential the programmer would write

```
400  N
 Q4
410  N
```

Quickies are also included for input and output of numbers in fixed-point, floating-point, and integer form.

If the same quicky is used more than once in a program it may be uneconomical to insert it each time. It is, however, very easy to turn a quicky into a closed subroutine. The return address is stored in a B-register —conventionally B-register 1—and the quicky is made into a routine ending in a modified jump. The function code for an unconditional jump is 59, and the third digit of an instruction specifies a modifier, so that

```
 R  20
 Q4
591   0
```

makes routine 20 a closed subroutine which replaces the content of the accumulator by its exponential and returns control to the address specified in B1.

7 DIRECTIVES

A directive takes the form of a single English word on the program tape. Only the first letter of the word is in fact identified, and the rest may be omitted. For example, the word WAIT (or simply W) will, when read, cause the input routine to stop. Several directives have already been mentioned, such as R(OUTINE), C(HAPTER), U(P), Q(UICKY). One important directive is E(NTER), followed by either an absolute or a symbolic address. This should appear at the end of a program tape. It transfers control to the Chapter Changing Sequence to bring the appropriate chapter into the high-speed store and enter it at the specified address; the entry cue is also stored to allow subsequent re-entry to the program.

The directives have been chosen as far as possible to make their operations self-explanatory, and their number has been kept to a minimum. There are in all fifteen.

8 OTHER FACILITIES

No attempt was made to deal efficiently with the input of large quantities of data by the input routine itself. However, it is very useful to be able to store a few constants with a program, and attention has been paid to making these easy to insert. So long as it is preceded by a sign, a number written in most of the usual forms will be accepted by the input routine and stored. For example, $+2$, $+2\cdot0$, $+0\cdot2_{10}1$ and $+2\cdot0_{10}0$ will all be read in and stored as the number $+2$ in floating-point form. (The lowered 10 is a symbol incorporated in the tape code to indicate that a decimal exponent follows.) Like instructions, constants can be labelled and referred to by symbolic addresses.

Another facility which is of considerable use in the organization of data is the provision of 25 preset parameters. These are denoted on the tape by $x1$, $x2$, . . . $x25$. They are set by equations such as $x3 = 12$, and can be re-set to new values at any point on the program tape. Preset parameters, once set, can be either stored as integers or used in the address parts of instructions, being interpreted as integers or addresses depending on the function code used in the instruction. Thus if the equation $x1 = 1{,}846$ appears in the program, then the instruction 400 $x1$ copies into the accumulator the number in the register pair 1,846, 1,847. Preset parameters can be used while writing the program, and the equations setting them to their correct values inserted afterwards at the head of the tape.

Among other facilities provided by means of directives are printing titles, inserting corrections to a program in the store, printing out the computing store addresses of routines and the drum store addresses of chapters, and overriding the normal assembly performed by the input routine.

9 CONCLUSION

It has been found that programmers learn the essentials of the scheme very quickly. It is quite possible for the scheme to be used in a primitive fashion by a beginner, or for an experienced programmer to use the more elaborate facilities provided. The latter have not all been mentioned here, for it would then also be necessary to describe Mercury in detail.

The authors' acknowledgments are due to Dr. S. Gill, on whose ideas much of the Mercury Input Routine is based. They would also like to thank Messrs. Ferranti Ltd. for permission to publish this paper.

REFERENCES

BROOKER, R. A. (1958). "The Autocode Programs developed for the Manchester University Computers," *The Computer Journal*, Vol. 1, p. 15.

LONSDALE, K., and WARBURTON, E. T. (1956). "Mercury: A High Speed Digital Computer," *Proc. I.E.E.*, Vol. 103, Part B, Supplement No. 2, p. 175.

WILKES, M. V. (1953). "The use of a 'Floating Address' System for Orders in an Automatic Computer," *Proc. Camb. Phil. Soc.*, Vol. 49, p. 84.