Oxford University Computing Laboratory

Computer Manuals

3146

SCIENTIFIC LANGUAGES

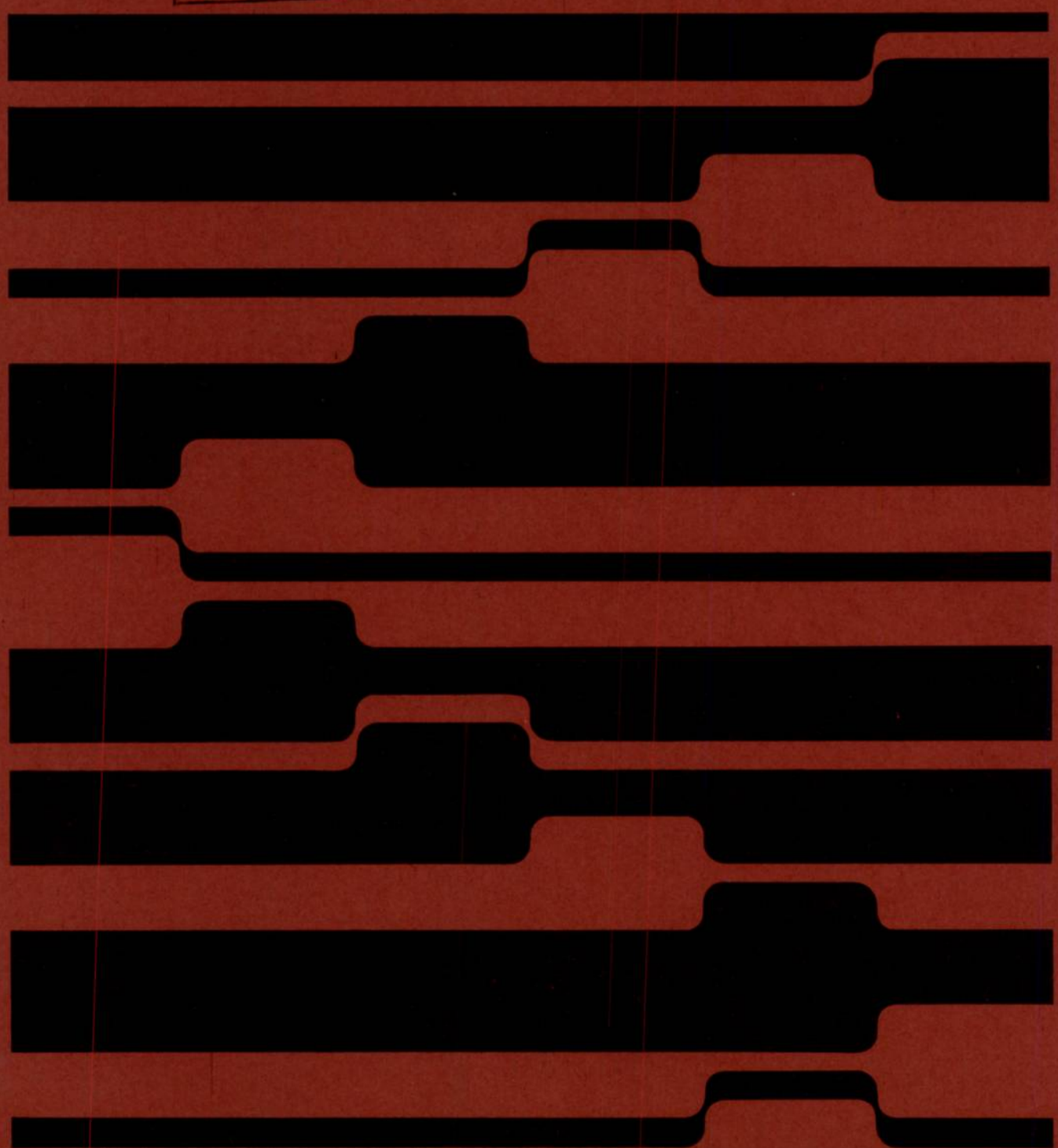
Extended Mercury Autocode

1900

ICL

EMA manual (Extended Mercury Autocode)

1900 Series



ICL International Computers Limited

User notice

RECEIVED 26 MAY 1972

9/12/70

MANUAL (NOTICE NO.)

3146 EMA (19)

File one copy of this notice with each of the manuals indicated.

DISC INPUT/OUTPUT AND FAULT 68

Run-time fault number 68 may occur when using disc input or output. This means that an attempt has been made to allocate a file which is already in use. Examples of typical situations are as follows:

1
```
OUTPUT 1 = ED3 (OUTFILE(2).SUB1(3))
CREATE 2 = ED3 (OUTFILE(2).SUB2(4))
```
Clearly it is not possible to create a file which is already an output file.

2
```
INPUT 1 = ED4 (INFILE(1).SUB1(2))
INPUT 2 = ED4 (DATAFILE(3).SUB2(4))
```
Two different input files cannot have the same unit number.

Although it is not clearly stated in the manual, disc file ED6 is used for storing auxiliary variables and disc file ED7 for ED or FD dumps (see *Backing Store directives*, section 3, page 4 and *Dump on directive*, section 4, page 17). Consequently, when these facilities are in use, unit numbers 6 and 7 should not be specified in peripheral description directives.

EMA COMPILER XMAE/4F

If consolidation is required after compilation, this compiler deletes itself and loads #XPCL. To make it load #XPCK instead, the following alteration to the compiler should be made using #XPEU (described in the 1900 Series manual *Library Specifications*, second edition, TP 4011).

```
        AOP   XMAE,4F,1,1,aa
        F
XMAE    2195,*70604353
XMAE    END
```

where *aa* should be CD or PT to specify that the following alterations are on cards or paper tape respectively.

Issued by Technical Publications Service, 30 - 31 Friar Street, Reading, Berkshire RG1 1JP

FORM 1/230/45(3.69)

ICL International Computers Limited

User notice

RECEIVED 20 MAY 1972

MANUAL (NOTICE NO.) 23/12/70

3146 EMA (20)

File one copy of this notice with each of the manuals indicated.

%MCA/5

This version of the complex arithmetic routine corrects the following errors:

1 The correct location of the fault is given if overflow occurs in a complex FIX instruction during evaluation of a complex expression.

2 Exponent overflow, if set on entry to %MCA will no longer cause incorrect results in expressions such as (A,B)/(CC,D) where CC has overflowed. Such overflow, and overflow occurring within %MCA is now correctly detected and reported.

3 The correct imaginary part is now given to some expressions which previously were given imaginary part zero, for example, $(2 \& 38, 5 \& -40)^2$

FORM 1/230/45(3.69)

Issued by Technical Publications Service, 30 - 31 Friar Street, Reading, Berkshire. RG1 1JP

ICL International Computers Limited

User notice

MANUAL (NOTICE NO.)

17/2/71

3146 EMA (21)

File one copy of this notice with each of the manuals indicated.

%MCA/6

This routine corrects the error reported in Software Notice 1900/SCIENTIFIC LANGUAGES/40, item 78.

Issued by Technical Publications Service, 30 - 31 Friar Street, Reading, Berkshire. RG1 1JP

ICL International Computers Limited

User notice

RECEIVED 20 MAY 1972

MANUAL (NOTICE NO.) 31/3/72

3146 EMA (22)

File one copy of this notice with each of the manuals indicated.

THE GEORGE 3 MACRO EMA

This user notice describes the system macro EMA which can be used to control the compilation and consolidation of EMA programs under the GEORGE 3 operating system. The macro is supplied to all installations using GEORGE 3.

The reader is assumed to be familiar with GEORGE 3.

FORM 1/230/45(3.69)

Format

The format of the macro is

EMA *parameter list*

Parameter types

The following parameter types are associated with the EMA macro:

SOURCE PROGRAM
LISTING
COMP
BIN
ERROR
SEMI

Each parameter type is discussed under the following four headings:

1 POSITION If the parameter has a fixed position, this is stated. If it is optional, the effect of omitting it is described; if it is mandatory, this is stated. Note that if an optional parameter is omitted, no null parameter is necessary in its place.

Issued by Technical Publications Service, 30 - 31 Friar Street, Reading, Berkshire. RG1 1JP

2 FORMAT The format (including all permissible alternatives) is described.

3 FUNCTION The user of the parameter and its effect on the execution of the macro is described.

4 NOTES Any additional information on the parameter types is given under this heading.

SOURCE PROGRAM

Position

If there is only one Source Program parameter, it must be the first parameter of the macro. If there is more than one, the first parameter of the macro must be a Source Program parameter, but additional parameters of this type may occupy any succeeding position(s) in the parameter string. There must be at least one Source Program parameter; otherwise an error will result.

COMP

Position

The Comp parameter is optional and has no fixed position. If this parameter is omitted, the macro will use and erase a workfile.

Formats

*CR
*TR
*CR *file description*
*TR *file description*
*MT *magnetic tape description*
*ED *file description*
*ED *exofile description*

Function

Each source program parameter specifies the location of a unit of source program to be read by the compiler. If a file description, exofile description or magnetic tape description is present, the source will be read from the specified file. The parameters *CR or *TR without a file description specify that the source program is to be read from the job source.

Notes

1 The first parameter of the macro must be a source program parameter.

2 There may be any number of parameters beginning *CR or *TR, provided that

(a) The total number of parameters of all types do not exceed 24.

(b) If the source program is to be read partly from the job source, this part must be read first, that is, the first parameter must be *CR or *TR.

2 There may only be one *ED parameter and one *MT parameter. The second and subsequent parameters of either type, if any, will be ignored.

3 The units of source will be read in the order of the parameters.

4 The last record of each unit of source read by the compiler, except the last, must be an appropriate switching statement (see section 4.3.6 of the manual).

LISTING

Position

The Listing parameter is optional and has no fixed position. If this parameter is omitted, the macro will use and erase a workfile.

Format

A Listing parameter consists of the characters *LP optionally followed by a file description. Thus the format is:

*LP *file description*

or

*LP

Function

A Listing parameter defines the file to which the compilation listing is to be written (if the file description is included) or specifies that the listing is to be sent to the monitoring file system (if the file description is omitted) in the OBJECT category. If the file specified in the file description already exists, the usual rules apply with regard to overwriting it.

When line printer output is sent to the monitoring file system, the Paper Feed Control Characters are ignored. Furthermore any lines longer than 120 characters (excluding the PFCC) will be split, the excess

characters being printed on the next line. It is not recommended that large amounts of output be sent to the monitoring file system.

Note

If more than 50 pages (245K) are to be output, a multifile must be specified in the file description; a work file cannot be used.

COMP

Position

The Comp parameter is optional and has no fixed position. If this parameter is omitted, the macro will use and erase a workfile.

Format

A Semicompiled parameter consists of the characters COMP followed by a file description. Thus the format is:

COMP *file description*

Function

A Semicompiled parameter defines a Unified Direct Access file to which the Semicompiled program is to be written. The file must previously have been created with a bucket size of 1 block, for example, by the command

CREATE *filename* (*DA,BUCK1,KWORDS1)

BINARY

Position

The Binary parameter is optional and has no fixed position. If this parameter is omitted, no binary program will be produced.

Format

A Binary parameter consists of the characters BIN optionally followed by a file description. Thus the format is:

BIN *file description*

or

BIN

Function

A Binary parameter specifies that a binary program is to be produced. It defines a UDAS file to which the binary program is to be written (if the file description is included) or specifies that a core image of the program is to be created (if the file description is omitted). If a file is specified it must previously have been CREATEd specifically for the macro with a bucket size of 1 block. The usual rules with regard to overwriting a file apply.

Note

In the case of an overlay program, a UDAS file must always be specified.

ERROR

Position

The Error parameter is optional and has no fixed position. If this parameter is omitted, the next command after the macro will be obeyed in the event of the run failing or an error being detected in the macro.

Format

An Error parameter consists of the characters ER followed by a label. Thus the format is:

ER *label*

Function

An Error parameter defines a label to which control is to be passed in the event of a failure during the run or an error in the macro.

Notes

1 Special action is taken if a command error occurs in a macro issued in a MOP job.

2 The label in the job description specified by the Error parameter must not coincide with any of the labels used within the macro, otherwise the command processor will go to the label within the macro when an error occurs. All labels in the EMA macro begin with the character 9. Hence the user must not specify in the Error parameter any label beginning with 9.

SEMI

Position

The Semi parameter is optional and has no fixed position. If this parameter is omitted, only the SRE1 subroutine group is used.

Format

SEMI *UDAS exofilename*

or

SEMI *UDAS file name*

Function

The SEMI parameter defines semicompiled input to the consolidator in addition to the semicompiled program and the SRE1 subroutine group. There may be up to eleven SEMI parameters. Each parameter should correspond to a LIBRARY or SEMICOMPILED program description statement and the parameters and program description statements should be in the same order.

Execution

The compiler #XMAE is used to compile the source program. If the BIN parameter is present in either of its formats, the consolidator #XPCK is then used to produce a binary object program.

Library entrants accessed

Entrant	*Modes of access*
:LIB.PROGRAM▽XMAE	Execute,read
:LIB.PROGRAM▽XPCK	Execute,read
:LIB.SUBGROUPSRE1	Read

Example

An EMA program is to be compiled and, if compilation is successful, loaded into core. The initial directives and program description directives are to be read from the job source (on cards) and the source program itself from a card-type file called EXSOURCE. The program description is terminated by

***T▽CR

The compiler listing is to be sent to the monitoring file system. In the event of a failure during the run or of an error in the macro, control is to be

transferred to the label 1FAIL in the job description. The following command would be used:

```
EMA *CR,*CR EXSOURCE,*LP,BIN,ER 1FAIL
```

ICL

EMA manual (Extended Mercury Autocode)

1900 Series

With effect from 9th July 1968 the name of
International Computers and Tabulators Limited
has been changed to
International Computers Limited.

Technical Publication 3146

First Edition November 1964

Second Edition December 1965

Reprinted October 1967

Reprinted July 1969
(Incorporating Amendment List 1)

Reprinted July 1971
(Incorporating Amendment Lists 1 and 2)

Issued by Technical Publications Service
International Computers Limited
Head Office: ICL House, Putney, London SW15
Produced by ICL Printing Services,
Letchworth, Hertfordshire

Contents

3146(5.68)

3146(5.68)

PREFACE

Mercury Autocode is a programming language devised by R.A.Brooker and others of Manchester University to facilitate the writing of mathematical and scientific programs for the Mercury Computer. Over the past few years, such programming has been simplified still further by a series of changes and extensions to the original autocode, and several variants have been introduced to provide versions of the language suitable for use with other computers.

Developments of the language specifically for Mercury have taken place at various Mercury centres, notably C.E.R.N. Geneva, A.E.R.E. Harwell, R.A.E. Farnborough and U.L.I.C.S. London. These establishments jointly introduced the first CHLF versions of the language and U.L.I.C.S. devised what is probably the final version for Mercury, CHLF 3.

A separate development of the language is MAC, available for use with the I.C.T. 1301 and 1101 computers. Although MAC differs in detail from the original versions of Mercury Autocode, it is similar in scope and power.

On the I.C.T. Atlas and Orion computers, however, a more powerful language, EMA (Extended Mercury Autocode), may be used. This includes as subsets both the original and the CHLF 3 versions of the autocode and contains a variety of extra facilities.

EMA has now been introduced for use with 1900 series computers. Like the Atlas and Orion versions, 1900 EMA is specifically designed to permit existing programs to be transferred with a minimum of amendment, and to permit programmers already familiar with a variant of Mercury Autocode to write programs for a new machine with a minimum of re -training.

Although this manual is designed to replace the provisional 1900 EMA manual, 3146 (11.64), almost all of the information given in this earlier manual is still valid. The present manual is provided to give more detailed information on the features described earlier, to supply more examples, and most important, to explain a variety of additional features that have been incorporated to make 1900 EMA compatible with Atlas and Orion EMA.

Section 1

BASIC FEATURES

INTRODUCTION 1.1

About this Manual 1.1.1

Newcomers to Extended Mercury Autocode should concentrate on Section 1. It covers the basic features of the language in sufficient detail to allow quite complex programs to be written.

Section 2 describes facilities that an experienced programmer may find useful but which are not essential for the newcomer to the language.

Section 3 covers facilities of more specialized interest. Each of these need be examined in detail only when a relevant problem, such as the integration of a set of differential equations, is to be solved.

Sections 1, 2 and 3 deal with the language in general terms, without reference to specific 1900 configurations which may impose minor restrictions on a program. Section 4, however, shows how an EMA program is prepared for input by particular 1900 peripherals, explains compilation requirements, and indicates how a program is tested before it is put into operation.

Section 5 describes some miscellaneous features. Much of the section will be of interest only to programmers accustomed to programming in a variant of Mercury Autocode devised for another computer.

When this manual is being used for reference, the index will normally be more helpful than the contents list. General forms of instructions, directives and functions are given in the table of summarized programming information.

Digital Computers 1.1.2

Digital computers operate upon numbers in some coded-digit form and make use of standard computational techniques to obtain direct numerical solutions to problems. A desk calculator is a simple form of digital computer. An automatic, stored-program digital computer is equivalent in some ways to a desk calculator and its operator regarded as a single unit. The main features of a digital computer are:

(a) **Input and Output** This is one of the roles of the operator of a desk calculator, who must set up numbers in the machine before they can be operated on and also read results from the machine, recording them elsewhere. For an electronic computer, information is transferred to and from the machine by automatic equipment.

(b) **Control** Here again, it is the operator who must control the sequence of operations on a desk calculator. An automatic computer is controlled by a program consisting of a sequence of detailed instructions in coded form. The whole program is stored within the machine before any of it is obeyed.

3146(10.65)

(c) **Arithmetic Unit** The mechanism of a desk calculator carries out the individual operations in accordance with the key depressed. Similarly, the arithmetic unit of an electronic computer performs the functions called for by the successive instructions in the program.

(d) **Storage** The keyboard and certain other registers of a desk calculator constitute a working store, insofar as the mechanism of the arithmetic unit is able to operate directly upon numbers contained in these registers. An electronic computer commonly has a working store capable of holding several thousand numbers.

Automatic Programming Languages 1.1.3

A program as stored in the computer consists chiefly of an ordered sequence of instructions in a code understood by the computer. Programs can be written directly in this machine code or they can be written in an Automatic Programming Language. Automatic Programming Languages or Autocodes are devices for speeding the writing of programs. They use some of the conventions of a human language such as English or Mathematics but follow more precise rules. This enables them to be unambiguously translated into a series of machine-code instructions to be obeyed by a computer. The translation is carried out by a computer under the control of a program called a *compiler*. This is a machine-code program, generally written by the manufacturer of the computer concerned.

A program written in an Autocode is called a *source program.* The program produced from it by the compiler is called an *object program.* One instruction in the source program may be translated or compiled into one object program instruction or into several, perhaps hundreds, of these instructions.

Autocodes have the following advantages:

(a) They are easier to learn than machine-code languages.

(b) Programs can be written and developed faster in an Autocode than in a machine-code language.

(c) A program is not necessarily restricted to one type of computer: it can run on any machine for which an appropriate compiler is available.

(d) An Autocode program is far easier to read and understand than the equivalent machine-code program. Since programs often have to be amended when the original programmer is no longer available this is not a trivial point.

The disadvantages of Autocodes are:

(a) Some computer time is taken up by compilation.

(b) An object program produced by a compiler is not usually as efficient as one written in machine code by a human programmer.

The balance of advantage and disadvantage will depend very much on individual circumstances. However, it is certainly true to say that in many scientific and technical problems the extra computing time needed by an Autocode program is a negligible factor compared with the tremendous gain in programming speed. This is especially so when, as often happens, the program will be run only a few times before the requirements change and it has to be altered.

The steps towards solving a problem with an Autocode program are:

(a) Precise formulation of problem.

(b) Decision made on general method of solution.

(c) Autocode program written.

(d) Object program produced by the compiler.

(e) Object program entered and obeyed.

Though the last step will often occur immediately after step (d) the two can take place at quite different times. It is essential that the distinction between the two processes be kept clearly in mind while this manual is being read.

Extended Mercury Autocode, or EMA, is intended mainly for scientific and technical work. It is a considerably extended version of Mercury Autocode, which was originally designed for the Mercury computer. EMA retains the simplicity of the earlier language while being much more powerful and having fewer restrictions. The remainder of Section 1.1 describes some basic concepts, starting with an informal introduction to the basic types of EMA instruction. The sequences below are useful only as examples and are worth only brief study.

The following is a short section of EMA program designed to obtain two numbers from a set of data and print out their sum and average:

```
Read (X)          | Read one number from data and call it X.
Read (Y)          | Read next number and call it Y.
A = X + Y         | Form sum and call it A.
Newline
Print (A) 3,2     | Print sum with three places before decimal point
                  | and two after.
Print (A/2) 3,2   | Print average in same style.
End               | Stop obeying instructions of this program.
```

When presented to the computer this program would be accompanied by a list of at least two numbers. Each number is made available to the program by being "read" from this list. If the list contained the numbers

781.212 102.352

then the program would assign the value 781.212 to the "variable" X and 102.352 to Y. The numbers are read and assigned in written order, so if the order of the two Read instructions had been changed the values of X and Y would have been reversed.

The third instruction shows a different method of assigning a value to a variable. Although it looks like an equation its actual meaning is: assign the numerical value of the expression on the right to the variable on the left. With the given program and data, A would be given the value 883.564 and would retain this value unless and until another instruction assigned a new one.

The results calculated by a program,, in this case the values of A and A/2, are printed in a style specified by the programmer. In the above case the results would appear as

883.56 441.78

The Newline instruction ensures that printing starts at the beginning of a line.

The End instruction prevents the computer from obeying any more instructions of the program and is the normal way of stopping.

Each of the above instructions is obeyed once and once only. A computer is used far more efficiently if sections of a program are designed to be obeyed repeatedly. To achieve this, the programmer must include instructions whose sole purpose is to alter the order in which instructions are obeyed. As an example, consider a similar program to the previous one, a program designed to read successive pairs of numbers from a list and print out the sum and average of each pair:

```
┌──►3) Read (X)
│     Read (Y)
│     A = X + Y
│     Newline
│     Print (A) 3,2
│     Print (A/2) 3,2
└──── Jump 3          | Carry on obeying instructions from the one labelled 3.
      - - -
```

Each time A/2 is printed, the instruction "Jump 3" causes the computer to start obeying the sequence again, starting at the instruction preceded by the label "3)".

The program would be accompanied by a list of numbers such as

131 97 21.9 731.02 10 291.4
- - -

The first time through the sequence, X and Y would be given the values 131 and 97 respectively. Next time through they would be assigned the next available numbers from the list, i.e. 21.9 and 731.02. The results would appear as:

```
228.00    114.00
752.92    376.46
301.40    150.70
- - -     - - -
```

The drawback of the above sequence is that there is no exit point. Once entered it will be obeyed repeatedly until the list of numbers is exhausted. This is not normally a satisfactory way of terminating a program: it is preferable that an End instruction be obeyed. This can be achieved by making the computer test a particular condition and jump back to the beginning of the sequence if the condition is true or stop if it is false. Suppose the programmer wishes to process a list of numbers he knows to be positive. Then a crude way of ending the program would be for the programmer to insist on two negative numbers being appended to the end of the data and for the program to test for a negative value of A; e.g.

```
3) Read (X)
   Read (Y)
   A = X + Y
   Newline
   Print (A) 3, 2
   Print (A/2) 3, 2
   Jump 3, A ≥ 0        | If A ≥ 0 carry on obeying instructions from the one
   End                  | labelled 3, otherwise obey the next instruction.
```

Here the Jump instruction is effective only as long as A is positive or zero. As soon as a negative value of A is computed the Jump instruction has no effect and the computer passes on to the next instruction, End.

The above program was described merely to introduce some types of EMA instruction. Of course it is unlikely that such a trivial program would be worth writing.

Directives 1.1.5

An EMA program contains *directives* as well as instructions. Whereas an EMA instruction is translated into one or more machine-code instructions a directive passes on information from the programmer to the compiler and is not itself translated. It is essential that the reader appreciates the difference. Consider the following sequence:

```
   C→10
3) Read (A)
   COMPILE QUERIES
   Y = Z-1
   X = 3.7
```

The two lines "C→10" and "COMPILE QUERIES" are both directives. The compiler takes note of them but produces an object program equivalent to:

```
3) Read (A)
   Y = Z-1
   X = 3.7
```

The meanings of these two directives will be discussed later.

Documents 1.1.6

Once a program has been written a copy of it is made on a Flexowriter or similar device. A Flexowriter is used in much the same way as a typewriter except that the carriage is never moved by hand. Even at the end of the line the carriage is returned by depressing the "newline" key. The main difference from a typewriter is that each key depressed not only prints a character or moves the carriage but also punches a pattern of holes across a length of paper tape. The pattern for each key is unique. There are thus two end products: a printed copy of the program and a length of perforated paper tape with patterns representing codes for the letters and other printed symbols and for such operations as space and newline.

3146(10.65)

The programmer uses the printed copy for reference and feeds the paper-tape version to the computer, which then has sufficient information available to build up internally an exact "image" of the printed copy.

The printed copy, the paper-tape version and the internal image are simply different representations of the same information and are referred to collectively as a *document*. Because the internal image is as close a copy of the printed document as possible, the programmer can largely forget about the paper-tape version and assume that if the printed document looks right then it is right. Throughout this manual the emphasis is on the printed representation of a document and the preparation of the paper-tape version is discussed only in Section 4.

A program document will usually be accompanied by one or more data documents from which data can be "read" by Read instructions. Again the emphasis is on the printed copy and the programmer can think of a Read instruction as scanning a printed document line-by-line until a number is found. Program and data documents are both *input documents*.

As results are generated they are "printed" on an *output document*, i.e. as Print instructions are obeyed an internal image is built up of the required printed document. The computer uses this image either to produce a printed document directly on a line-printer or else to output a paper-tape version from which the programmer can subsequently obtain a printed copy.

The paragraphs above describe documents largely in terms of paper tape, the most common medium for the input of program and data. However, the general principles apply regardless of the media. If, for example, the program is punched into cards, the card version, its internal image and any listing of the cards the programmer might choose to make are referred to collectively as an input document.

Program Layout 1.1.7

Each EMA instruction and directive must have a line to itself. It must not continue on to the next line. Blank lines between instructions are ignored by the compiler and each line can contain spaces in any position, though none are essential. Spelling must be correct but there is no distinction between upper- and lower-case letters. Thus, the following lines would be interpreted identically by the compiler:

```
Jumpdown 33
JUMP DOWN 33
Jum   PdOWn 3     3
```

The programmer is advised to make his program as clear as possible by the insertion of blank lines at intervals and the indentation of blocks of instructions.

The above refers only to a program document. In both data and output documents spaces are certainly significant, and while all input documents may contain upper- or lower-case letters, an output document will contain only upper-case letters. Even within a program there are some exceptions. An instruction such as

```
Print ('Case 5')
```

contains text. Here, spaces are not ignored and the instruction would print

```
CASE 5
```

Note that lower-case letters are converted to upper-case for the purpose of output.

Both program and data documents can contain *comments*. Anything preceded by a vertical bar is taken to be relevant to the programmer, not to the computer. This facility was used in the examples in Section 1.1.4 and allows a programmer to annotate his program in any way he chooses.

The remainder of Section 1 describes the basic features of EMA in more detail, starting with the arithmetic operations.

Constants 1.2.1

A *constant* is a number that is explicitly written in a program. It can appear either in conventional decimal notation, e.g.

3 +21 -33.0 -004 -.734567 00.00173400

or in a form such as:

-32.7345&5

which means -32.7345×10^5 (i.e. -3273450). This form is particularly useful for avoiding long strings of 0's in very large or very small numbers. Other examples are:

+3.24&-1 1&+9 2937&-20 .3&-9

The number after the ampersand (&) can be signed but must be a written integer, i.e. it must contain neither a decimal point nor an ampersand. Note that 10^{-7} is represented by 1&-7; 10&-7 means 10×10^{-7}, i.e. 10^{-6}. If the number before the ampersand is omitted, it is assumed to be 1; thus 1&9 and &9 represent the same number.

The Special Variables 1.2.2

An EMA *variable* has a numerical value that can be changed in the course of a program. There are several types of variables but for the moment only the set of *special variables* is considered. These 30 variables are identified by the symbols:

A B C D E F G H U V W X Y Z π

A' B' C' D' E' F' G' H' U' V' W' X' Y' Z' π'

There is no distinction in a program document between capital and non-capital letters, so these variables could equally well be written as a, b..., a', b'...; however, all variables will be written in capital letters throughout this manual.

At the start of a program the values of all variables (except for π) are undefined. They should not be assumed to be zero. Various kinds of instruction are available to assign values to variables (for instance the Read instruction already mentioned) and this section is concerned with the *arithmetic instructions* of which a very simple example is:

X = 7.2

This is not an equation in the normal mathematical sense; it means not 'X has the value 7.2' but 'give X the value 7.2'. If it were followed by

A = X + 4

B' = -3(2A-X)

then X, A and B' would be assigned the values 7.2, 11.2 and -45.6 respectively. Once a variable has been given a particular numerical value it retains this value until an instruction assigns a new one.

π is a special case. At the start of a program it is automatically set to

3.1415926536 (approximately)

Its value can be changed, but a programmer does so only at his own risk. In any case, π will be automatically re-set to its normal value at certain points in a program.†

The right-hand side of an arithmetic instruction must be an *arithmetic expression* constructed according to rules laid down in Section 1.2.6. These rules are fairly straightforward and it is sufficient to say here that + and - have their obvious meanings, / indicates division and * indicates multiplication (rather than the normal multiplication sign which would easily be confused with X in manuscript form). In practice, * is often omitted; Section 1.2.6 lists only two cases where it is absolutely necessary.

† It is reset each time one of the following instructions is obeyed: ACROSS, DOWN, UP, PRESERVE, RESTORE.

3146(10.65)

Examples

```
F  = -G'
X' = -172&7 A/(2π)                    |(i.e. (-172&7)*A/(2*π))
H  = 3 [(A-B)/(A+B) - (A'+B') / (A'-B')]
Z  = [1+A/(2+B/(3+C/D))]
A  = A+1
Z  =  ZZZ - 3Z
```

Brackets have their usual significance and can be nested to any depth. Both square and round brackets may be used as required as long as left and right members of a pair are matched.

The last two instructions above calculate new values of A and Z in terms of their old values. If A and Z had been 7.3 and 9 then after these instructions they would be 8.3 and 702 respectively. The expression in the last instruction could not be written as Z^3 - 3Z because all squares, cubes etc. must be written as explicit multiplications.

A special variable may take any value within a very large range, but this value will be held only to a precision of approximately 11 significant digits. More details are given in Section 1.2.10.

The Indices 1.2.3

There are 24 quantities called *indices* . They are identified by the symbols:

```
I  J  K  L  M  N  O  P  Q  R  S  T
I' J' K' L' M' N' O' P' Q' R' S' T'
```

These indices can be thought of as a set of integer variables since they can have only exact integer values. Although the range of index values is less than the range of variable values, numbers are held exactly. More details are given in Section 1.2.10.

Indices are given values and combined into arithmetic expressions in the same way as the special variables; e.g.

```
I  = 3
K' = 3M'(S+T-1)
J  = 3X+Y
M  = I'/7
```

Because there is no guarantee that X and Y are integers, or that I' is exactly divisible by 7, the expressions in the last two examples might not have integer values. Accordingly, each one is computed as it stands, then the value is rounded off to the nearest integer and assigned to the relevant index. Similarly,

```
I = 93.7
J = 175.3
```

are equivalent to (but take longer to execute than)

```
I = 94
J = 175
```

A convenient way of rounding a special variable X to the nearest integer is to write

```
I = X
X = I
```

Indices can be used to carry out exact integer arithmetic but their chief uses are to hold counts and to act as suffixes, as described below.

Note that the index T' has a special use which leads to its value being changed at various points in the program (see Section 1.4.1.). It is best avoided for normal use. The index O should also be used as little as possible as it is easily confused with zero.

However, the programmer may choose to establish some method of distinguishing an O from a zero by, for example, writing O as Ø.

The Main Variables 1.2.4

The special variables and the indices are available to every program. They will usually be supplemented by one or more sets of *main variables* , which are variables identified by a letter with a suffix, e.g.

A3 π91 U0 W500

Suffixes can be applied only to the unprimed letters

A B C D E F G H U V W X Y Z π

The main variables are held to the same precision as the special variables and are assigned values in the same way; e.g.

```
B3 = (X-Y)/π1+3X*B3+1
A197 = 981.5 K + (U2 + V2 + W2)/(U6 + V6 + W6)
```

There is no connection between a letter used as a special variable and the same letter used as a main variable with a suffix. In particular, A, B, C etc., are quite distinct from A0, B0, C0 etc.

Before he uses any main variables, the programmer must indicate to the compiler, by means of directives, the total number he intends to use and by what names he intends to refer to them. The directive:

```
MAIN 600
```

means: the following program uses at most 600 main variables. The directives:

```
B →99
π →199
A →149
```

mean: in the following piece of program the first 450 main variables are to have the names B0, B1..... ...B99, π0, π1,.....π199, A0, A1,.....A149.

The Main directive is one of a set of directives called *initial directives* which are written at the beginning of a program. The other directives mentioned above are called *set directives* or *variable -setting directives*. Their written position will be defined more precisely in Section 1.5 but for the moment they can be assumed to be written immediately ahead of the section of program that uses the variables they name.

The suffix of a main variable need not be written explicitly. It can be written as an index or as a particular type of arithmetic expression that combines indices and integers. For instance, if J and K are 5 and 7 respectively, the instruction

```
AJ = A(J-K+3) + B(JK)
```

is equivalent to

```
A5 = A1+B35
```

This method of writing suffixes has advantages when an instruction such as the above is made to be executed repeatedly; for if the values of J and K were changed between repetitions, the instruction would act on different members of the sets A0, A1, ... and B0, B1, ... each time.

There are obvious problems in introducing main variables into arithmetic expressions. For instance, it has not yet been made clear why A(J-K+3) refers to a main variable and not to the product of the special variable A and the expression (J-K+3). The reason will be given in Section 1.2.7, where the form that a suffix may take will be defined more precisely.

Functions 1.2.5

Many of the standard mathematical functions can be introduced into arithmetic expressions. For instance, the instruction

```
X = ϕsin(Y) + ϕcos[Z/3]
```

calculates the sum of sine Y and cosine Z/3. The symbol ϕ precedes all function names. Its only purpose is to indicate that the following letters are to be interpreted as a name, not a product (for instance, sin(Y) is an acceptable, if eccentric, way of writing S*I*N*Y). Function arguments must always be in brackets, even when the argument is a single constant or variable.

3146(10.65)

Functions can be nested to any depth, e.g.

A = ϕsq rt(X + ϕsq rt(Y))

sets A to $\sqrt{(X+\sqrt{Y})}$.

Some of the available functions are listed below. They are split into two classes, the functions which can have general values and the *integer functions* which can have only integer values.

General Functions

ϕ sq rt (x)	$\sqrt{x}$	$x \geqslant 0$
ϕ exp (x)	e^x	
ϕ log (x)	$\log_e x$	$x > 0$
ϕ mod (x)	modulus (absolute value) of x	
ϕ fr pt (x)	fractional part of x = x-ϕint pt(x) (see below)	
ϕ sin (x)	$\sin x$	x in radians
ϕ cos (x)	$\cos x$	x in radians
ϕ tan (x)	$\tan x$	x in radians
ϕ arcsin (x)	$-\pi/2 \leqslant \sin^{-1} x \leqslant \pi/2$	$-1 \leqslant x \leqslant 1$
ϕ arccos (x)	$0 \leqslant \cos^{-1} x \leqslant \pi$	$-1 \leqslant x \leqslant 1$
ϕ arctan (x)	$-\pi/2 < \tan^{-1} x < \pi/2$	
ϕ arctan (x,y)	$-\pi < \tan^{-1}(y/x) \leqslant \pi$ The quadrant is determined as if x and y are proportional to the cosine and sine of the angle respectively.	$x^2 + y^2 \neq 0$
ϕ radius (x,y)	$\sqrt{x^2+y^2}$	

Integer Functions

ϕ int pt (x)	the integral part of x, i.e. the largest integer $\leqslant x$	
ϕ sign (x)	the sign of x	i.e. 1 if $x \geqslant 0$ -1 if $x < 0$
ϕ parity (m)	$(-1)^m$	i.e. 1 if m even -1 if m odd
ϕ max (A0, m, n) ϕ min (A0, m, n)	The suffix of the largest (or smallest) element of the set Am, A(m+1), ...An. If there is no unique largest (or smallest) element, that with the maximum (or minimum) value and lowest suffix is taken.	

The arguments x, y, m and n in the above functions can be any arithmetic expressions, though m and n will normally be integers or expressions with integer values.

Other functions will be introduced later but the above are those chiefly used in arithmetic expressions. The table of summarized programming information contains the full list of functions.

Exponentiation of positive quantities can be obtained by using ϕlog and ϕexp in conjunction, e.g.

ϕexp(-3.9 ϕlog(X+A) gives $(X+A)^{-3.9}$

ϕexp(ϕlog(X)/3) gives $\sqrt[3]{X}$

The two functions ϕradius and ϕarctan, which have two arguments, provide a convenient means of obtaining the polar co-ordinates of a point with cartesian co-ordinates (X, Y).

Note that the definition of ϕint pt implies that ϕint pt (-5.73) and ϕint pt (-5.33) both have the value -6. Except within an arithmetic expression, ϕint pt can often be dispensed with, e.g.

I = X - 0.5

is equivalent to

I = ϕint pt (X)

To round off Y to the nearest 0.001, either

```
Y = 0.001 ϕ int pt (1000Y + 0.5)
```

or

```
I = 1000Y
Y = 0.001 I
```

can be written.

The instruction

```
J = ϕ max(X0, 3, N)
```

puts J equal to the suffix of the largest of X3, X4, ... XN. An integer function can be used as a suffix, so the effect of

```
Y = XϕmaX(X0, 3, N)
```

is to put Y equal to the largest member of the set, not just to its suffix.

There is a special instruction for evaluating polynomials whose coefficients have previously been calculated. It has the form

```
Y = ϕ poly(x)AS, n
```

where x and n can be expressions and AS is any main variable. The integral part of n is taken. The instruction sets Y equal to the value of

$$AS + A(S+1)x + ... + A(S+n)x^n$$

ϕ poly cannot be used as part of a larger expression.

Arithmetic Expressions 1.2.6

This section summarizes the rules for writing EMA expressions. Complex and double-precision expressions will be introduced in later sections. Unless the contrary is specifically stated, the term *arithmetic expression* is intended in this manual to mean a real, single-precision expression.

The elements of an arithmetic expression can be constants, special and main variables, indices, and functions (integer or general). They are combined by the operators +, -, * and /. A single element is regarded as a trivial form of arithmetic expression. The first element of an expression may be signed or unsigned.

Functions can be nested to any depth. Brackets have their usual meaning and can also be nested to any depth. Two types can be used:

```
( )  [ ]
```

They both have the same significance in an expression and, provided that opening and closing brackets of a pair are matched, the programmer can mix both types as he wishes. The pattern

```
[ ( ( ) ( ) ) [ ( ) ] ]
```

could equally well have appeared as:

```
( ( ( ) ( ) ) ( ( ) ) )
```

Throughout this manual, whenever brackets are mentioned it can be assumed that either type may be used unless the contrary is explicitly stated. This seldom occurs. A safe rule is that round brackets are never wrong. (One exception is described in Section 3.4 but it is unlikely to occur in practice.)

The operators * and / take effect before + and - unless brackets indicate otherwise; thus

```
7 + 2 * 12/6 -5
```

and `(7 + 2) * 12/(6-5)`

have the values 6 and 108 respectively. No two operators may be adjacent: A*-B should be re-written as A*(-B) or -A*B.

The operator * can be omitted as long as no change of meaning results. There are only two such cases:

(a) a product of two constants (73 does not, of course, mean 7 * 3)

(b) a product of one of the unprimed special variables and a quantity with the same form as a main variable suffix, with the variable coming first (this will be discussed in Section 1.2.7).

It is important to note that a divisor must be only a single element, i.e. a constant, an index or variable, a function, or a bracketed expression. The following are acceptable:

Z/I +1 I*Y/3&-9 +A Y/ϕsin(X) H/(1+A/(2+B/3))

The following are not acceptable and would be rejected during program translation:

X/A*B I/2π Y/2ϕsin(X) H/(A-1)(A'+1)

They should be rewritten as

X/(A*B) I/(2π) Y/(2ϕsin(X)) H/((A-1)(A'+1))

or

(X/A)*B (I/2)π (Y/2)ϕsin(X) (H/(A-1))(A'+1)

whichever meanings were intended.

Within all the above rules, an operator on the left takes effect before one on the right; for instance the expressions

A*B*C/D and A-B+C-D/E

are equivalent to

((A*B)*C)/D and ((A-B)+C)-(D/E)

The order in which an expression is evaluated is important only when it affects the accuracy of the result.

The *integer expressions* are a subset of the set of arithmetic expressions. They follow the same rules as above with the following extra restrictions. The only elements allowed are written integers, indices and integer functions. The operator / is not allowed. If these rules are broken the expression will not technically be an integer expression even if it gives an integer result. Section 1.2.7 discusses the only case in which this is important.

A *written integer* is a constant containing neither a decimal point nor an ampersand: 3 is a written integer, 3.0 and 3&0 are not. The integer functions can, and usually will, have non-integer arguments: thus

ϕint pt(X-Y/Z) + ϕsign(3.7 ϕsin(Z)-I/J)

is an integer expression even though it contains two real expressions as function arguments. It should be emphasised that it is the written form of an expression that determines whether or not it is an integer expression: (I+3) is an integer expression, whereas (I+9/3) is not.

Suffixes 1.2.7

There is only one context in which it is essential that an expression be an integer expression. That is where the expression is intended to be a main variable suffix. A suffix can be either a written integer, an index, an integer function, or an integer expression in brackets; thus

A3 CK' πϕint pt(X-Y) F[9+ϕsign(X/3)]

all represent main variables. Even a trivial departure from these rules leads to a product being obtained; thus

A3.0 CH' πϕmod(I) F[9/3+ϕsign(X)]

all represent products of a special variable with another quantity.

On the other hand, any quantity that can legally be a suffix is taken as such whenever it is immediately preceded by one of the letters A B C D E F G H U V W X Y Z π, even if the programmer had intended the letter to be a special variable and the whole sequence to be a product. The expressions

```
B4 + B4.0 + CL + CH + Y(I/J)
```

and

```
Dϕint pt(X) + Dϕsin(X) + XIJK + X(I-J)K
```

are interpreted as meaning

```
B4 + B*4.0 + CL + C*H + Y*(I/J)
```

and

```
Dϕint pt(X) + D*ϕsin(X) + (XI)*J*K + (X(I-J))*K
```

where * has been inserted wherever multiplication takes place. To avoid error the programmer should write all constants, integer functions and indices as near the beginning of a product as possible. If in doubt he should sprinkle liberally with asterisks and brackets.

Context of an Arithmetic Expression 1.2.8

Arithmetic expressions can appear in many types of EMA instruction besides the arithmetic instructions. For instance, the instruction

```
Print (X+Y)  P+Q,  3R-1
```

contains three separate expressions, one whose value is to be printed and two whose values indicate the number of integer and fractional digits to be printed. Now in some places it is essential that an integral value be obtained from an expression and such expressions are said to be in *integer contexts.* When non-integer expressions appear in such contexts, their values are computed as they stand, then modified to give an integer value.

This modification takes place in one of two ways. The value of an expression written on the right-hand side of an index arithmetic instruction (e.g. I = X/Y) or in one of the argument positions of the general loop instruction (to be described in Section 1.3.4) is rounded off to the nearest integer, so a slight deviation from standard integer form, though leading to a trivial increase in computing time, is unlikely to cause a wrong result, e.g.

```
I = 3.0
```

gives I the value 3 but causes a redundant rounding operation to take place that could have been avoided by writing

```
I = 3
```

In all other integer contexts the integral part is taken, i.e. the largest integer not greater than the actual value. It follows that if, say, X was supposed to be 3 and was actually 2.9999 the instruction

```
Print (Z) X, 1
```

would print two integer places instead of the expected three. The correct result could be obtained either by writing

```
Print (Z)  X+0.01, 1
```

where 0.01 is greater than the expected error in X, or by writing

```
J = X                          |Rounds off.
Print (Z) J, 1                 |Uses rounded-off value.
```

In general, any type of expression is evaluated as it stands and modified to suit its context; if possible; thus the integer expression I + J in the instruction

```
Print (I+J) 2, 4
```

is evaluated as an integer then given a zero fractional part.

Examples 1.2.9

(a) Read values of x and y from a data document then calculate and print the values of f and 3f given by

$$f = f(x,y) = \sin(xy+x/y^3+y/x^2) + 2\pi x$$

The following section of program is correct, though at this stage necessarily incomplete:

```
Read (X)                                  |Assign values to X and Y.
Read (Y)                                  |EMA variables X, Y, F
F = ϕsin(X(Y+1/(YYY))+Y/(XX)) + 2πX       |are used for x, y, f of the formula
Print (F) 2, 3                            |Print f
Print (3F) 2, 3                           |Print 3f
End
```

(b) Read values of a, b and c and find the roots of the equation

$$ax^2+bx+c = 0$$

from the formula

$$x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

The appropriate sequence could be

```
X→1

Read (A)
Read (B)
Read (C)
E = ϕsq rt(BB-4AC)
X0 = (-B+E)/(2A)                          |1st root
X1 = (-B-E)/(2A)                          |2nd root

Print (X0) 1, 4
Print (X1) 1, 4
End
```

The variable E was used solely to avoid computing the same square root twice.

(c) Evaluate

$$f(X+Y) = A5(X+Y)^5 + A6(X+Y)^6 \quad \ldots A20(X+Y)^{20}$$

The ϕpoly instruction can be used here:

```
B = X+Y
F = ϕpoly (B) A5, 15                      |A5+A6B+ ... A20 B^15
F = BBBBBF                                |Required value
```

If the first instruction had been omitted and B replaced by X+Y in the following two instructions, then the addition would have taken place six times, once in the polynomial instruction and five times in the next instruction.

Size and Precision of Numbers 1.2.10

Main and special variables, general constants and general functions can take values in the approximate range

$$-5.6 \times 10^{76} < x < 5.6 \times 10^{76}$$

with values in the approximate range

$$-10^{-77} < x < 10^{-77}$$

being treated as zero.

Such general values are stored in two words of 1900 storage in a form of binary floating-point representation. This resembles the mathematical notation by which a number is represented as a decimal fraction raised to a power of ten, and it provides a precision of approximately 11 significant digits. The accuracy is stated as an approximation because, in the same way that many fractions do not have an exact decimal equivalent, so many numbers do not have an exact binary floating-point equivalent.

The fact that not all numbers can be expressed exactly in floating-point form, and that the rounding of results takes place in binary rather than in decimal terms, means that some care is required in handling general values. Consider, for example, the three instructions

```
X = 0.1
Y = 10 (0.006 + 0.004)
Z = 10 * 0.006 + 10 * 0.004
```

Each of these assigns an approximation of 0.1 to a special variable. Though each is a very good approximation, X, Y and Z will be exactly identical only by chance: the difference would be of the order of 10^{-10}.

Although such inaccuracies are normally acceptable in results, they could cause problems in the program. If, for example, two of the above variables are compared in the program, an inequality is likely to result: an instruction such as "Jump 3, X = Y" is unlikely to cause a jump. In cases where exact results are essential - where, for example, a count is established to control the number of performances of a loop - indices (see below) and exact integer arithmetic should be employed.

In general, rounding errors are negligible, but there may be cases where they are important. Consider the instruction

```
Y = A - B
```

If A and B were almost equal, their difference could be comparable in size to the rounding error and it would then be accurate to far fewer than 11 significant digits. The best remedy is to re-arrange the calculation so that such circumstances do not arise. If this is impossible, the programmer must resort to the double-precision arithmetic described in Section 3.5.

There will be no rounding error if each intermediate result of a series of operations can be held exactly in binary form. In particular, integer values can be manipulated exactly so long as they do not exceed 11 digits in magnitude. Some care is required. Consider the following sequence:

```
Y =  2
Z = 10
A = (90Y)/Z
B = 90(Y/Z)
```

The variable A would be assigned an exact value of 18. Because the intermediate result (Y/Z) is not exact, B would be assigned a value slightly smaller or larger than 18. The difference would be of the order of 10^{-10}. Of course, exact integer arithmetic can also be carried out by using the indices (see below).

The general functions introduce only a little inaccuracy. However, with the trigonometric functions ϕcos, ϕsin and ϕtan, as few as possible of the 11 significant digits of the argument should be taken up by non-significant multiples of 2π: the larger the absolute value of the argument, the fewer digits are used to calculate the result. In fact, with an argument greater than approximately 10^{11}, an error is indicated because the above functions would yield a meaningless result. Also note that the accuracy of ϕtan is reduced for values of the argument close to odd multiples of $\pi/2$.

When a program works close to the limits of accuracy, it may well produce slightly different results on different types of computers. Such differences are more marked between Mercury or a 1300 series computer and the 1900 than between Atlas or Orion and the 1900.

Indices, written integers and integer functions may take values in the range

$$-8,388,608 \leqslant x \leqslant 8,388,607$$

and such values are held *exactly* in binary form in a single 1900 word.

Their main use is to act as suffixes or counts, but they can be used whenever exact integer arithmetic is required. Rounding errors can occur with the above integer quantities only when they are used in expressions which do not comply with the rules for integer expressions (see Section 1.2.6.).

A written integer outside the above range will be treated as a general constant, and any expression containing such an integer will be a non-integer expression.

3146(10.65)

Jump Instructions 1.3.1

A program consisting solely of a list of instructions to be obeyed in their written order does not make use of the essential ability of a computer to choose between different courses of action, depending on whether or not a given condition is true. This ability is provided in EMA by instructions such as the first one of the following sequence:

```
    Jump 3, X - Y ⩽ π /2
    X = X -π
3)Z = φsin (X)
    - - -
```

If X - Y $>\pi/2$ then the first instruction has no effect, the next one replaces X by X-π and the last one evaluates the sine of this new value of X. If X-Y $\leqslant\pi/2$ then the normal sequence of obeying instructions is suspended and resumed at the instruction labelled 3; thus the sine of the original value of X is evaluated. The departure from the normal sequence is called a *jump* and the instruction that caused it is called a *conditional jump* instruction.

An *unconditional jump* instruction has a form such as

```
Jump 5
```

which would always cause a jump to label 5). In the following sequence, the two types of jump are used in conjunction to evaluate f(x), where

$$f(x) = \begin{cases} e^{x} & \text{if } x < 0 \\ 1+\sin x & \text{if } x \geqslant 0 \end{cases}$$

```
    Jump 5, X ⩾ 0
    F = φexp(X)
    Jump 2
5)F = 1+φsin(X)
2)- - -
```

A *label* is written immediately ahead of the instruction to which it refers and must be an integer followed by a round, right bracket. Integers in the range 0 to 127 can be used, but labels 0) and 100) should be reserved for fault sequences - to be described in Section 4. The programmer need not use a consecutive block of label numbers and there is no significance in the order in which labels are written. An instruction can have any number of labels, e.g.

```
3) 97) 12)  X' = -90
```

Some typical conditional jumps are:

```
Jump 20, X > I                          Jump 99, J'+1 ≠ K'-L'
Jump 7, X φ sin(X/Y)⩽Z+1                Jump 8, 2+φsq rt(X) <φexp(Z)
```

The comma must appear and can be read as "if". Any two arithmetic expressions can be compared using any of the symbols:

$$= \quad \neq \quad > \quad \geqslant \quad < \quad \leqslant$$

It is better if = and $\neq$ are normally used only to compare two integer expressions, as other expressions may not be evaluated exactly and will be exactly equal only by chance.

Examples

(a) To put A equal to the largest of X, Y and Z:

```
    Jump 2, X ⩾ Y          | Find maximum of X and Y.
    A = Y                  |
    Jump 3                 |
2)A = X                    |

3)Jump 5, A ⩾ Z            |
    A = Z                  | Find final maximum
5)- - -                    |
```

(b) To evaluate F and G where

$$F = X^3 - Y^3 \text{ and } G = \frac{X-Y}{3} + 2 \text{ for } X > Y$$

or

$$F = Y^3 - X^3 \text{ and } G = \frac{X-Y}{3} \text{ for } X \leq Y$$

```
F = YYY - XXX                    | Assume X ≤ Y
G = (X - Y)/3
Jump 11, X ≤ Y
F = -F                           | Modify F, G if X > Y
G = G+2
11)- - -
```

(c) To form X^n such that $X^{n-1} < 1000 \leq X^n$, where $X > 1$.

```
    F = 1
┌──►3)F = XF
└─── Jump 3, F< 1&3
     - - -
```

Switches 1.3.2

This subsection may be omitted on a first reading of Section 1.

A *switch* is a particular type of jump whose destination is decided at some point before the instruction is obeyed. In EMA it can have the form

```
Jump (I)
```

where I can be replaced by any index. The label to be jumped to must have been previously specified by an instruction as

```
I) = 3)
```

This is called a *label-setting* instruction and has the effect of making the jump equivalent to

```
Jump 3
```

The numerical value assigned to I by "I) = 3)" is not in itself meaningful to the programmer and the index should be regarded as having been set equal to a label, not an integer. The important points to remember are that I does not simply have the value 3 and "I = I+1" will not set I equal to label 4).

The brackets in label-setting instructions must always be plain round brackets. The label can be specified by an expression with a value in the appropriate range (0 to 127). Thus, if K and L are 10 and 9 then

```
J) = 2(K+L)-1)
```

is equivalent to

```
J) = 37)
```

and causes a subsequent

```
Jump (J)
```

to have the same effect as

```
Jump 37
```

Jumps of this type can be conditional, e.g.

```
S') = 20)
- - - -
Jump (S'), L = M+1
- - - -
```

Brackets must not be added or omitted, so the following are both impermissible:

```
Jump (3)
Jump I
```


3146(10.65)

A *loop* is any sequence of instructions that may be obeyed repetitively, each repetition being called a *cycle*. Example (c) of Section 1.3.1 included a two-instruction loop that terminated when X^n became greater than 1000. A common requirement is for a loop to cycle a fixed number of times, with some quantity being steadily increased (or decreased) on each cycle. Suppose the sum 1 + 1/3 + 1/5 + ... + 1/15 is to be found. A suitable loop could be set up by using jump instructions but a more concise method is to set up an automatic loop by writing

```
A = 1
N = 3(2)15
A = A + 1/N
Repeat
- - - -
```

The effect is to repeat the instruction "A=A+1/N" seven times with N successively taking the values 3, 5, 7, 9, 11, 13 and 15. Another example is

```
X = 0
I = M(-1)0
A(M-I) = BI
X = X + BI
Repeat
- - -
```

This sequence sets A0, A1, ... AM equal to BM, B(M-1), ... B0 respectively, and at the same time sets X equal to the sum of these values.

This type of loop is a simple automatic loop and is defined by a pair of instructions: a *simple loop* instruction with the form

I = P (Q) R or I = P (-Q) R

and a *Repeat* instruction. The body of the loop appears between the two and may be any sequence of EMA instructions. These are repeatedly obeyed with I taking in turn the values

P, P ± Q, P ± 2Q, ... R

After the last time, the instruction written below the Repeat is obeyed with I still equal to R.

I could be replaced by any other index and is called the *controlled index*: P and R are the initial and terminal values and Q is the increment. P, Q and R can each be replaced by any index or by an unsigned written integer. Only plain round brackets are allowed in the instruction. Even a slight departure from this form is an error: contrast the acceptable loop instructions

S=1(1)10 I=1(3)61 M=10(-1)1 P'=L'(M)100

with the unacceptable loop instructions

X=1(1)10 I=1(3.0)61 M=-10(1)1 P'=L'[M]100

The initial and terminal values can be negative only if they are represented by indices; a loop to be executed for all values from -50 to -100 could be specified thus:

```
I = -50
J = -100
   K = I(-1)J
   - - - -
   - - - -
   Repeat
```

Only one Repeat is allowed for each loop instruction. The temptation to replace an unconditional jump to a Repeat by an extra Repeat must be resisted in sequences such as the following:

```
I = 1(1)100
Jump 1, AI>0
BI = φexp (AI)
Jump 2                          | Not 'Repeat' here.
1)BI = φlog(AI)
2)Repeat
 - - -
```

It is not essential for the full number of cycles to be executed. The following sequence simply finds the first negative element of the set F1, ... F100:

```
      S' = 1(1)100
      Jump 1, FS' < 0
      Repeat
      - - - -
      - - - -
    1)X = FS'
      - - - -
```

If there had been no negative element then the instruction after the Repeat would have been obeyed. After jumping out of an unfinished loop it is possible to jump back and resume cycling from the same point as long as the value of the controlled index is not changed.

When a simple loop instruction is obeyed there is always at least one cycle to the loop, but careless programming can lead to its having an indefinite number. Consider the instruction "I = 1(2)N". If N were 1 the loop would be obeyed once and the increment would be irrelevant. If N were 0 or 2 a closed loop would be entered with I taking the values 1, 3, 5, ...

To ensure that the terminal value is attained, and cycling not continued indefinitely, the following rules should be adhered to:

(a) the difference between the terminal and initial values should be a positive multiple of the increment

(b) the value of the controlled index, the increment and the terminal value should not be changed during a cycle

(c) the controlled index should not also appear as the increment or terminal value

(d) there should be no jump into the body of a loop, unless there has previously been a jump out of it.

Note that an index used to specify an initial value can be freely used once the loop instruction has been obeyed. These rules can be broken but inexperienced programmers are advised not to do so.

Loops within loops are allowed, but if the body of an automatic loop contains a loop instruction it must also contain its associated Repeat. The two are matched in like manner to left and right brackets, so there is never any doubt about which Repeat belongs to which loop instruction.

Example

To evaluate:

```
A1(B1+B2+. . . .+B19+C20+C21+. . . .+C30)
+A51(B51+B52+. . . .+B69+C70+C71+. . . .+C80)
. . . . . .
+A251(B251+B252+. . . .+B269+C270+C271+. . . .+C280)
```

The required sequence could be:

```
U = 0
      I = 1(50)251
      V = 0
            J = 0(1)18
            V = V+B(I+J)          | Obeyed 19 times for each I
            Repeat

            J = 19(1)29
            V = V+C(I+J)          | Obeyed 11 times for each I
            Repeat
      U = U+AI*V
      Repeat
- - - - -
```

This example has two loops within another loop. The indentations have been introduced to aid the eye; they have no effect on the computer's interpretation of the sequence.

There is no limit to the actual number of loops contained in any other loop but at no point may they be nested more than 24 deep, i.e. when reading down a written list of instructions there must never be more than 24 loop instructions awaiting their Repeat. This is an insignificant restriction.

General Loops 1.3.4

There is a more general form of loop instruction typified by:

```
Loop, V = -7.5 ϕinc [0.2] 3Z+0.7
      - - - -
      - - - -
      Repeat
```

V would here be given the values -7.5, -7.3, ... 3Z+0.7. Any variable or index can be controlled by such loops. The terminal and initial values and the increment (indicated by "ϕinc") can all be arithmetic expressions, and their values will be calculated before the first cycle. A more complicated example is:

```
Loop, A2 = ϕsin(X)+A2+A3    ϕinc(AI+BI)    ϕsq rt(A2)
      - - - -
      Repeat
```

No change to any variable or index within the loop can alter the values assigned to the controlled variable at the beginning of each cycle. Even if this variable is itself changed it will be brought back into step at the next Repeat, as in the following example:

```
Loop, X = 1.5 ϕinc(1)11.5
      I = 1(1)5
      X = XX
      Repeat
Print (X) 11, 0
Repeat
- - - -
```

This is a simple loop within a general one and would cause the values of $(1.5)^{32}$, $(2.5)^{32}$, ... $(11.5)^{32}$ to be printed.

With an index on the left of a Loop instruction the three expressions on the right are in integer contexts: each one is evaluated as it stands then, if necessary, rounded off to the nearest integer (not to the integer below the actual value as usually occurs in integer contexts - see Section 1.2.8). The instruction

```
Loop, I = -2.1 ϕinc(0.57) 10.71
```

is therefore equivalent to

```
Loop, I = -2 ϕinc(1) 11
```

Because the increment in the instruction

```
Loop, X = 0 ϕinc(π/180)π/2
```

is not an exact number, the values obtained by successive additions of this increment to X grow steadily less accurate and the 91st value could differ from $\pi/2$ by up to 10^{-9}. Such possible inaccuracies have no effect on the number of cycles of a loop, since all loops with this general form are arranged to terminate after being obeyed with any value of the controlled variable that differs from the terminal value by less than half of the increment. Immediately after the final cycle the controlled variable (or index) is set exactly equal to the terminal value, so that in the above example the programmer could rely on X having the best possible approximation to $\pi/2$ on exit from the loop. This step takes place even if the controlled variable has already exactly attained the terminal value.

It is not possible to set up a closed loop with this general type of Loop instruction. If the initial value is within $\frac{1}{2}$ ϕinc of the terminal value there will be one cycle with the controlled variable set equal to the initial value, then it will be re-set to the terminal value and the loop will be ended. If the initial value is not within $\frac{1}{2}$ ϕinc of the terminal value and successive additions will never bring it within that range then the body of the loop will not be executed. Consider the loop:

```
Loop, X = 1 ϕinc(1)A
      Print (X) 2, 0
      Repeat
```

If 0.5<A<1.5, the sequence is equivalent to

```
Print (1) 2, 0
X = A
```

If A<0.5 the sequence is equivalent to

```
X = A
```

These points need not be overstressed: given sensible limits a loop will be obeyed sensibly; given nonsensical limits it will not be obeyed at all, though the controlled variable will always be set equal to the terminal value. Of the four rules listed for simple loops, only the last one applies to general loops, i.e. there can be a jump into the body of a loop only if there has previously been a jump out of it. This rule must not be broken.

The simple form of loop should be regarded as the normal type and the general form used only when its convenience and built-in safeguards outweigh the conciseness of the former; otherwise the two types are mixed in any way that suits the programmer.

Examples

(a) To give A0 to A20 the values of the function

$$f(x) = \frac{\sqrt{x}}{x-1}$$

for x = 5.0, 5.1, ... 7.0:

```
X = 4.9
I = 0(1)20
X = X+0.1
AI = φsqrt(X)/(X-1)
Repeat
```

(b) To print the values of the above function for values of x running from (sin Y)/Z + 3 to tan (Y/Z) in steps of (A1+AJ):

```
Loop, X = φsin(Y)/Z + 3      φinc(A1+AJ)      φtan(Y/Z)
      Newline
      Print ( φsq rt(X)/(X-1)) 1, 6
      Repeat
```

Halt and End Instructions 1.3.5

The normal way to terminate a program is to use the instruction

```
End
```

which prevents the computer from obeying any more instructions.

Temporary halts can be achieved by means of the instruction

```
Halt
```

which suspends the program, normally for operator intervention. After such a suspension, the program can be re-started by the operator at the instruction after Halt.

The effect of these two instructions on 1900 series machines is as follows:

End causes the program to stop, and to be deleted from the store. The following message is sent to the console typewriter

#name: DELETED:- *ab*

where *name* is a four-character program identifier (described in Sections 4.4.4, 4.5.4, and 4.6.4) and *ab* is the last two digits of the last label before the *End* instruction, or the label of the *End* instruction.

Halt causes the program to be suspended, and the following message to be sent to the console typewriter

#name: HALTED:- *ab*

where *name* is a four-character program identifier and *ab* is the last two digits of the last label before the *Halt* instruction, or the label of the *Halt* instruction.

The program may be restarted at the next instruction after the *Halt* instruction by the operator message:

GO *#name*

3146(4.68)

Note: These messages will not be the same for all machines in the 1900 series. For example *#name* is omitted on some machines, and on a machine without a console typewriter equivalent indications are given on the control panel.

INPUT AND OUTPUT 1.4

Reading of Data 1.4.1

Data is made available to a program by being "read" from an input document. The document is scanned from left to right and down the page. No part of it can be read twice. There are several instructions for the reading of data but this section is concerned only with the reading of single numbers.

A data document will usually contain a set of numbers such as

```
317.5
+91.7        -1984          0.1073
3.71&-10        5'5'64
```

to be read one at a time by *Read* instructions such as

```
Read (Z)
Read (AI)
Read (M)
```

These particular instructions would give Z, AI and M the values 317.5, 91.7 and -1984 respectively. The remaining numbers would be read by further instructions of the same type.

On a data document each number must be followed by the start of a new line or by at least two space characters or by a prime ('). There is no significance in these different methods of terminating a number and the programmer uses whichever he finds most convenient. The set of numbers above could have been rewritten as

```
317.5        +91.7          -1984
0.1073         3.71&-10              5     5     64
```

with no change of meaning. Extra spaces and blank lines between numbers will be ignored by Read instructions.

There is no limit to the number of digits allowed in a number but only the 11 or 12 most significant of them contribute to its value. All the written forms that a number may take in a program document (see Section 1.2.1) are allowed in data, but spaces are now significant. Single spaces are allowed in a number but, except immediately after an ampersand, no number should contain two consecutive spaces. Thus

```
+10 000 761.617                    123&      -7
```

are two quite acceptable numbers, but

```
+  123
```

would be regarded as a fault.

On a data document, a comma can replace an ampersand, e.g. 123, -8 is equivalent to 123&-8. This notation is not allowed within a program because a comma has many other meanings.

An exponent on a data document can also be introduced by the character E.

A number may be read into any variable or index. When a number to be read into an index is not an integer the whole of the number is read but only its integral part is assigned to the index; e.g. if

```
123.78   -73.1
```

are read by

```
Read (J)
Read (K)
```

then J and K will have the values 123 and -74 respectively.

Every time a number is read, the value of T' is changed to provide information which can be used when a reading error occurs. Precise details are given in Section 5. Most programmers will not be interested in this information and need only ensure that no important value of T' is lost. The instruction, "Read (T')" has the normal effect of reading one integer to T'.

There are several techniques for reading large amounts of numerical data. Suppose it is required to read a list of salaries, such as

```
3000
981
1020
721
- -
- -
```

into the variables G0, G1 etc. If the programmer knows that this list always contains exactly 100 figures he could write

```
J = 0(1)99
Read (G J)
Repeat
```

If the length of the list varies from one run of the program to another, he could arrange for the salary list to be preceded by a number indicating how many entries it contains; then the appropriate instructions could be

```
Read (N)
N = N-1
      J = 0(1)N
      Read (GJ)
      Repeat
```

An alternative method that does not require the number of entries to be counted during data preparation is to end the list with a fictitious entry that is easily recognisable as such and to test for this entry in the program. On the assumption that no salary will be less than zero, the programmer could specify that the final entry on the list be negative. Then, with the further assumption that there will never be more than 1,000 real salaries, the sequence could be as follows:

```
J = 0(1)1000
Read (GJ)
Jump 1, GJ < 0
Repeat
- - -
- - -
1) J =J-1
- - -
```

As soon as the final entry is read there is a jump out of the loop and J is reduced by one, so that it indicates the suffix of the last useful value to be read. The sequence of instructions immediately after the Repeat instruction is obeyed only if the list contains more than the 1,000 positive values allowed for.

A very useful variant of the last method allows the list to be terminated by some character such as an asterisk or a letter rather than by a number. This will be described in Section 4.

Printing of Results 1.4.2

Of the several types of instruction concerned with the output and layout of results, the most common is the *Print* instruction. This instruction "prints" one number on to a document which will at some stage be output from the computer on such peripherals as a line printer, a paper tape punch or card punch. For example, the instruction

```
Print (X) 2, 3
```

means: print the value of X with 2 integral and 3 fractional digits, e.g.

```
47.771
```

A zero in either format position has a special effect. In the second position it causes an integer to be printed with no decimal point; thus

```
Print (A) 4, 0
```


3146(10.65)

could print

```
-1981
```

In the first position it causes a number to be printed in floating-decimal style with one place before the point; thus

```
Print (A) 0. 4
```

could print

```
-1.9814,    3
```

Whatever style of printing is chosen, the last digit printed is always correctly rounded off.

A Print instruction with no style specified, e.g.

```
Print (X)
```

prints a number with a fixed format. It is used mainly in program testing and is described in Section 4.

The number of character positions occupied by a number depends, in general, on the Print instruction rather than on the number itself. Though non-significant zeros are not printed before the decimal point they are replaced by spaces; an exception is the last figure before the point which is always printed. All zeros after the point are printed. The sign of a number is represented by a space or a minus sign immediately ahead of the first printed digit. Two space characters are added after the last visible character of the number.

The first results printed by a program start at the beginning of a line, so a series of numbers could be printed as follows:

```
98.132 -171  9123 -3.794,   4
```

Other layouts are obtained by inserting extra spaces and starting new lines. The instructions

```
Space
```

and

```
Newline
```

print one space character and start one new line respectively. Similarly

```
Space (I + J)
```

and

```
Newline 3
```

print I+J spaces and start 3 new lines. Brackets are necessary when anything other than a written integer specifies the number of spaces or new lines. Calling for 3 successive new lines is of course equivalent to generating 2 blank lines. "Space 0" and "Newline 0" have no effect.

Example

To print the numbers A1 to A100 in two columns:

```
A1     A2
A3     A4
- - - -
A99    A100
```

There are to be four spaces between columns and a blank line between each pair of numbers.

```
N = 1(2)99
Print (AN) 2, 3                  | Prints 1 number and 2 spaces
Space 2                          | 2 extra spaces
Print (A(N+1)) 2, 3
Newline 2                        | 1 blank line
Repeat
- - -
```

The printed results would appear in a form such as:

```
-98.123      -97.000
 27.001       -0.110
 -7.010        0.000
-12345.017     -98.123
 71.234      -71.374
 - -           - -
```

The fourth line above shows what happens if a number to be printed has more than the specified number of integer places: all are printed, but the number, and therefore the rest of the line, will be displaced to the right, causing any error to be made obvious. Instructions such as "Print (X) 1, 1" use this effect to obtain such left-justified layouts as,

```
21349173.5
10.3
-1.1
999.4
2.0
-1001.7
```

where each number is printed as far to the left as possible and the decimal points are therefore not aligned.

The general form of the Print instruction is

Print [x] m, n

where x, m and n may all be arithmetic expressions, though in practice m and n will usually be written integers. The effects of different combinations of m and n are tabulated.

Values of m, n	Effect	Number of character positions
$m \geq 1$, $n \geq 1$	m integral places, n fractional places.	m+n+4 (at least)
$m \geq 1$, $n = 0$	m integral places, no fractional places.	m+3 (at least)
$m = 0$, $n \geq 0$	Floating-decimal number with n+1 significant digits, 1 before the point and n after, and an exponent with up to 3 digits.	n+10 (exactly)

Any results printed by the Print instruction are in a suitable form for being read by Read instructions in other programs.

Numeric results should normally be accompanied by some explanatory text. The standard method of printing text is typified by the instruction

Print ('CASE 5, X =')

which would print

CASE 5, X =

i.e. all the characters between the quotation marks. All characters listed at the beginning of 2.3.4 plus space are permitted. In this and any other instruction which outputs text, any lower-case letters will be converted to upper-case. All spaces between the quotes are printed, but unlike Print, Print (' ') never inserts extra spaces.

Example

To read 10 values of a parameter a and print out for each of them the 5 values of

$$f(x) = \frac{x^2 - a}{1 - ax}$$

given by x = 0, -0.2, ... -0.8. The layout is to be as follows:

```
A= 100.00

  X        F(X)
 0.0      -1.0000,     2
-0.2        - - -
-0.4        - - -
-0.6        - - -
-0.8        - - -


A =  30.0

  X        F(X)
 0.0        - - -
-0.2        - - -
-0.4        - - -
-0.6        - - -
-0.8        - - -

etc.
```

The program could be:

```
L = 1(1)10                                        | 10 values of A
Read (A)
Newline 3                                         | 2 blank lines
Print ('A=')
Print (A) 1, 2
Newline 2                                         | 1 blank line
Print (' X       F(X)')
          M = 0(2)8                               | 5 values of X
          X = -0.1M
          Newline
          Print (X) 1, 1
          Space 2
          Print ((XX-A)/(1-AX)) 0, 4
          Repeat

Repeat
End
```

An alternative way of printing text is to write it on a line by itself after the instruction Printline, e.g.

```
Printline
WRONG DATA IN SECTION 10
```

and the text is then printed from the beginning of a new line:

```
WRONG DATA IN SECTION 10
```

It is thus precisely equivalent to

```
Newline
Print ('WRONG DATA IN SECTION 10')
```

There must be no blank lines between the instruction and the text. All space characters, including any printed after the last visible character, will be copied to the output document.

Example

```
Printline
THE INSTRUCTION IS PROBABLY MOST
Printline
USEFUL WHEN A LARGE BLOCK OF
Printline
TEXT EXTENDING OVER SEVERAL LINES
Printline
IS TO BE PRINTED.
```

This prints as:

```
THE INSTRUCTION IS PROBABLY MOST
USEFUL WHEN A LARGE BLOCK OF
TEXT EXTENDING OVER SEVERAL LINES
IS TO BE PRINTED.
```

Note that a Newline instruction is necessary after the last of a set of Printlines unless other results are to continue on the same line.

Data Titles 1.4.3

Text can be copied a line at a time from an input to an output document by the instruction

```
Read Data Title
```

This instruction searches the data document from wherever reading had previously finished until the heading

```
Data Title
```

is found. Then the next line is copied across to the output document and printed on a new line.

Example

An input document has the following layout:

```
Data Title
5 JAN 1964                          | Date

27                                  | Priority number

Data Title
LOW GRADE ORE.  BATCH LM/1A
1
91.7  91.3  72.4
-72.4  101.4

Data Title
MEDIUM GRADE ORE.  BATCH MM/1B
3
50.4  57.2  48.5
-57.2  109
- - -
- - -
Data Title
171 BATCHES EXAMINED

0                                   | Terminator.
```


3146(10.65)

This data is processed by the following section of program:

```
        Printline
        ORE ANALYSIS
        Read Data Title          | Date
        Read (P)                 | Priority number
    ►3) Newline                  | Blank line
        Read Data Title          | Batch number
        Read (Q)
        Jump 1, Q ≠ 0            | Test for end
        End
    ►1) Read (X)                 | Sequence to check
        - - -                    | "acceptability"
        Newline
        Jump 2, Z ⩾ Z'
        Print ('UNACCEPTABLE')
        Jump 3
    ►2) Print ('ACCEPTABLE')
        Jump 3
```

The printed results might appear as

```
ORE ANALYSIS
5 JAN 1964

LOW GRADE ORE.  BATCH LM/1A
ACCEPTABLE

MEDIUM GRADE ORE.  BATCH MM/1B
UNACCEPTABLE

- - -
- - -

171 BATCHES EXAMINED
```

Any mixture of upper- and lower-case letters can be used in the phrase "data title" on a data document. Spaces are completely ignored: they can occur anywhere but are not essential, even between the two words. There should be no other visible characters on the line. There must be no blank line between the heading and the actual text. Any items that precede the heading "Data Title" and have not been read prior to the instruction "Read Data Title" will be ignored. When the instruction has been obeyed, reading continues from the line after the title. As with the Printline instruction, no new line is automatically started after the text has been printed.

Runout and Paper Throw 1.4.4

Whenever it is required to output a short section of blank paper tape, the instruction

```
Runout
```

may be used. This outputs six inches of 'runout', i.e. 60 blank tape characters. If punched cards are being used, it will output two blank cards. Its effect on a line printer is to cause a throw to the head of the next page.

Chapters 1.5.1

A program is broken down into blocks of instructions and directives, each block carrying out some well-defined operation and having a degree of independence of other blocks. Such blocks are called *Chapters* and each one is given a number. Chapters are written one after the other, the extent of each being indicated by two directives such as

```
CHAPTER 1
- - -                          | Instructions of Chapter 1.
- - -                          |
CLOSE
```

The last Chapter of a program is always called Chapter 0 and in many programs this will be the only chapter.

Example

The EMA sequence below is a substantially complete program to print a table of factorials from 1! to 20!

```
CHAPTER 0
Printline
      n          factorial n
Newline
F = 1
N = 1(1)20
F = NF                         | Calculate factorial
Newline
Print (N) 2, 0                 | Print N
Space 6
Print (F) 1, 0                 | Print N!
Repeat

End

CLOSE
```

Note the difference between the last two lines of the above example. "CLOSE" is a directive marking the finish of the written form of the Chapter and is not compiled into machine-code instructions. "End" is compiled into an instruction which, when obeyed, terminates the current run of the program. It can appear anywhere in a Chapter.

When a program has been compiled, it is automatically entered at the first instruction of Chapter 0. The instructions of this Chapter are obeyed normally until an instruction to transfer to another Chapter is reached. A typical example is

```
Across 28/2
```

meaning: jump across to label 28 in Chapter 2 and continue obeying instructions from there. An instruction in Chapter 2 could then transfer across to another Chapter (which may or may not be Chapter 0). Chapters can be transferred to in any order. It is not possible to simply "run on" from one Chapter to the next; an Across or similar instruction is always necessary.

Each Chapter has its own set of labels (running from 0 to 127) which are quite independent of those in other Chapters. A Jump instruction can refer only to a labelled instruction in the same Chapter as itself.

Chapters have numbers in the range 0 to 4095. The numbers need not form a consecutive block and the Chapters can be written down in any order, provided that Chapter 0 is written last; thus, a program could have four chapters with the nonsensical numbering scheme of 327, 2075, 981, 0.

3146(10.65)

Example

The program below prints several tables of the function

$$f(x) = e^x \sin x$$

The initial value, increment and terminal value of x for each table are read from a data document such as:

```
  3      0.2      4
 7.1      3     10.1
  .       .       .
  .       .       .
  .       .       .
  7      0.5     12
-100
```

The last entry is negative and indicates that no more values are to be read. The tables will be printed as follows:

```
TABLES OF F(X) = EXP X * SIN X
TABLE 1
   X                F(X)
 3.00000,     0       .
    .                 .
    .                 .
    .                 .
 4.00000,     0       .
TABLE 2
   X                F(X)
 7.10000,     0       .
    .                 .
    .                 .
    .                 .
    .                 .

    .                 .
    .                 .
    .                 .
    .                 .

NO. OF TABLES = 10
TOTAL NO. OF ENTRIES = 201
```

The program has been split into three Chapters to illustrate the use of Across instructions, though in practice it would probably have been written as a single chapter.

```
CHAPTER 1          |Calculation Chapter
|This Chapter prints a complete set of values of x and f(x) for each table.

1) Loop, X = F ϕinc (G) H          |Across from Chapter 0.
        Newline
        Print (X) 0, 5
        Space 4
        Print [ϕexp(X)ϕsin(X)] 0, 5
        N = N+1                    |Count total entries
        Repeat
  Across 2/0                       |End of one table
  CLOSE
```

```
CHAPTER 2        |Wind-up Chapter
 |This Chapter prints the number of tables and the total number of entries in these tables.

1) Newline 2                              |Across from Chapter 0
   Print ('NO. OF TABLES =')
   Print (T-1) 1, 0
   Newline
   Print ('TOTAL NO. OF ENTRIES =')
   Print (N) 1, 0
   End
   CLOSE


   CHAPTER 0        |Control Chapter
   |This chapter prints some text, then sets up the initial conditions for each table in turn.

   |PROGRAM ENTERED HERE:
   Printline
   TABLES OF F(X) = EXP X * SIN X
   N = 0                                  |Total entries in all tables
       T = 1(1) 1000                      |No. of tables ≤ 1000
       Read (F)                           |Initial value of x.
                    Jump 1, F ≥ 0
                    Across 1/2            |End when F < 0
    1) Read (G)                           |Increment of x
       Read (H)                           |Terminal value of x
       Newline 2                          |Blank line
       Print ('TABLE')
       Print (T) 1, 0                     |Table no.
       Printline
          X              F(X)
       Across 1/1                         |To calculation Chapter
    2) Repeat                             |From Chapter 1
       Print ('TOO MANY TABLES')           |Error stop
       End
   CLOSE
```

Communication between Chapters 1.5.2

The special variables and the indices are available in all Chapters. Their values are not altered by Chapter changing, except that π is automatically re-set to 3.1415926536 (from which it should not normally be changed anyway). The status of the main variables during Chapter-changing is more complex. Their values do not change but their names may do so at the discretion of the programmer.

The initial directive Main specifies the total number of main variables used by the program and is written once, before the first chapter. If it is omitted, the directive

```
MAIN 480
```

is assumed and will give sufficient main variables for many programs. The names of the main variables are defined at the beginning of each Chapter. It is possible to avoid repeating Set directives when a Chapter uses the same sets of variables as a previous Chapter. Consider the following program:

```
MAIN  350

CHAPTER 1
  X → 99
  A → 99
  - - -
  - - -                                   |Instructions
  CLOSE
```


3146(10.65)

```
CHAPTER 2
VARIABLES 1
F → 149
- - -
- - -
CLOSE

CHAPTER 0
VARIABLES 1
- - -
- - -
CLOSE
```

This program uses 350 main variables. In Chapter 1, the first 100 are called X0 to X99, the second 100 are called A0 to A99 and the remainder have no names because, presumably, they are not required in that Chapter. In Chapter 2, the directive "VARIABLES 1" indicates that the Set directives "X → 99" and "A → 99" of Chapter 1 also apply to this chapter, while the directive "F → 149" gives the names F0 to F149 to the remaining 150 variables. Chapter 0 uses only the two sets defined in Chapter 1.

A Variables directive will normally appear immediately after the Chapter directive and, with one exception, it can refer to any other chapter of the program. Thus, Chapter 0 above could have contained

```
VARIABLES 2
```

and the effect would have been as though it were replaced by

```
X → 99
A → 99
F → 149
```

The exception is Chapter 0; VARIABLES 0 may not appear in 1900 EMA.

So far only a fairly straightforward case has been considered. In fact, the start of a new Chapter provides an opportunity for completely re-naming the variables reserved by the Main directive. Consider the following Chapters:

```
MAIN 400

CHAPTER 1
A → 399
- - -
CLOSE

CHAPTER 0
F → 99
B → 299
- - -
CLOSE
```

There are 400 main variables in use. In Chapter 1 they have the names A0 to A399. In Chapter 0 the same 400 variables have the names F0 to F99, B0 to B299. A Chapter change by, say, an Across instruction has no effect on the values of the variables, only the names are changed. Thus the sequence

```
B10 = 73
Across 12/1
```

if obeyed in Chapter 0 would cause Chapter 1 to be entered at label 12 with A110 equal to 73. This is because B10 and A110 are simply alternative names for the 111th main variable.

The names applied to the main variables can be re-distributed at the beginning of each Chapter in any way desired by the programmer, e.g.

```
MAIN 600

CHAPTER 3
π → 199
U → 10
A → 50
C → 337
- - -
CLOSE

CHAPTER 4
A → 150
D → 200
E → 150
F → 50
Y → 45
- - -
CLOSE
```

The main variables in Chapter 3 are called

π0, ... π199, U0, ... U10, A0, ... A50, C0, ... C337

The same main variables in Chapter 4 are called

A0, ... A150, D0, ... D200, E0, ... E150, F0, ... F50, Y0, ... Y45

Thus π0 and U0 in Chapter 3 correspond to A0 and D49 in Chapter 4. Notice that A0 to A50 in Chapter 3 bear no relationship to A0 to A150 in Chapter 4. If it had been intended that they should, then the Set directives ought to have been written in appropriate positions. The simplest change would be to write the directives in Chapter 3 as

```
A → 50
π → 199
U → 10
C → 337
```

but of course the relationship between all the other sets in each Chapter is then changed also.

This method of re-naming main variables allows a Chapter to be written without any regard being paid to the names used for the main variables in other Chapters, but it should be used with care since a change to the value of a main variable in one Chapter implies changes to apparently different variables in other Chapters. All problems of interpretation can be avoided by giving the main variables the same names in all Chapters.

Overlapping References to Main Variables 1.5.3

This subsection may be omitted on a first reading of Section 1.

It is sometimes convenient to regard the main variables as one continuous set, even when they have been split into sets by means of Set directives. This is achieved by allowing the suffixes of one set to overlap those of another set. Consider the chapter

```
MAIN 450

CHAPTER 1
D → 199
A → 149
- - -
CLOSE
```

The variables A0, ... A149 follow immediately after the variables D0, ... D199 and can be referred to within the chapter as D200, ... D349. Similarly, the 100 unnamed variables can be referred to as D350, ... D449 or A150, ... A249. Negative suffixes are also allowed: D0 to D199 could be referred to as A(-200) to A(-1). Clearly, any reference to D(-1) or A250 is an error, since there are no variables available for these names.

Overlapping references are seldom, if ever, essential.

Repeated Set and Variables Directives 1.5.4

This subsection should be omitted on a first reading of Section 1.

Normally, all Set directives are written at the head of a Chapter before any instructions but each one can appear anywhere in a Chapter as long as it comes before any reference to the variables it names. A further refinement, which is seldom necessary, is that directives within a Chapter can define different sets associated with the same letter, e.g.

```
MAIN 250

CHAPTER 1
B → 100
C → 50
- - -                          Instructions
- - -
B → 75
- - -
- - -
CLOSE
```

In the first part of the Chapter, the variables B0, ... B100 and C0, ... C50 are available. In the second part, the set of C's is still available but the original set of B's is not. Instead, a new set of variables B0, ... B75 is available and these variables have no connection with the earlier set. If B10 were put equal to, say, 19 by an instruction in the earlier part of the Chapter, then within the later part B10 were put equal to 99, this would not change the first B10 and a jump back to the earlier part of the Chapter would find it still having the value 19. The only way of referring to the first set B0, ... B100 in the second part of the Chapter is by using negative suffixes.

This sort of duplication is seldom necessary in practice and is best avoided.

When a Variables directive takes effect, all previous Set and Variables directives of the Chapter are cancelled. The directive names the main variables from the beginning of the set. The names given are those in force at the CLOSE of the specified Chapter. The following, unlikely, Chapter illustrates these points:

```
CHAPTER 0
A → 10                         | Defines A0 to A10
- - -
- - -
VARIABLES 5                    | "A→10" cancelled and variables
- - -                           | renamed as in Chapter 5.
- - -
VARIABLES 2                    | Variables again renamed.
- - -                           | This time as in Chapter 2.
CLOSE
```

Again, this sort of effect is seldom required.

Labels in a Chapter 1.5.5

This subsection may be omitted on a first reading of Section 1.

It has already been stated that each Chapter has its own set of labels. These are local to that Chapter and can be referred to outside it only in Across or similar instructions. The effect of a label-setting instruction is also localised. The instruction

```
J) = 5)
```

sets J equal to label 5 of the current Chapter. A subsequent

```
Jump (J)
```

should be obeyed only in the same Chapter.

Division into Chapters 1.5.6

Where possible, a program should be divided into Chapters so that each one carries out one logically distinct part of the calculation. There is no real restriction to Chapter length, but 100 EMA instructions is a reasonable working maximum. If the programmer keeps his chapters within this restriction he is unlikely to run out of labels; faults are fairly easily located; and, if the program is well designed, a change in requirements can normally be satisfied merely by replacing one Chapter by another.

One more factor to be taken into account is that an Across instruction takes longer to execute than a Jump instruction, so Chapter changing should be kept to a minimum.

SUBROUTINES 1.6

General 1.6.1

When the same operation needs to be carried out at different points in the program, it is often convenient to write only one sequence of instructions to perform this operation and to enter or "call in" this sequence whenever necessary. Such a sequence is called a *subroutine.* When, for instance, values of a particular function are required in a program, it is usually more efficient to call in a subroutine to calculate each value as needed than to store a table of values. As another example, a subroutine could be written that would be used whenever a block of numbers was to be printed with a particular, complicated layout.

One advantage of subroutines is that they can often be made sufficiently general for them to be useful in many programs. EMA functions such as ϕsin are actually disguised subroutines.

A simple-minded way for the programmer to write a subroutine in a chapter is to label the first instruction of a sequence and to make the last one a Jump. Thus, the sequence between the instruction labelled 2 and the instruction "Jump 5" below is a subroutine, which is entered from two places (at least)

This subroutine can be called in from anywhere in the Chapter but always passes control back to the instruction labelled 5 eventually.

In practice, it is more usual to write subroutines so that the return point depends on the instruction that called in the subroutine. The remainder of Section 1.6 is concerned with some techniques for programming such subroutines in EMA. Two further methods will be introduced in Section 2.1.

Jumpdown Sequences 1.6.2

The simplest way of incorporating a subroutine in a Chapter is to write it as a normal sequence of instructions with a label against the first one. Then entry is by an instruction such as

Jumpdown 3

Initially this causes a jump to the instruction labelled 3; then the instructions of the sequence are obeyed as usual until an instruction

Return

is encountered and this returns control to the instruction after the Jumpdown.

Example

The sequence of instructions to calculate F and G for given values of X and Y that appeared in example (b) of Section 1.3.1 is written below as a subroutine and called in with particular values of X and Y.

```
    X = 91
    Y = 70
    Jumpdown 10                          | Find F, G
    Print (F) 0,4
    Print (G) 0,4
    - - -
    - - -
10) F = YYY - XXX
    G = (X-Y)/3
    Jump 11, X ≤ Y
                F = -F
                G = G+2
11) Return                               | Jump back to instruction after Jumpdown.
```

In this example, once the subroutine had been written the instruction "Jumpdown 10" could have been regarded as an instruction to calculate values of F and G. The subroutine could be called in by such instructions any number of times in the Chapter.

A Return instruction is not permanently tied to a particular Jumpdown instruction and a subroutine can contain any number of Return instructions. Of course on any one entry only one of them will be obeyed.

A subroutine can call in another subroutine by means of a Jumpdown instruction. A Return in the second subroutine then passes control back to the instruction after the Jumpdown in the first subroutine. Chains of Jumpdown sequences will be described in more detail in Section 2.1.4.

A Return must be obeyed in the same Chapter as the Jumpdown to which it is intended to refer, though there may have been any number of Chapter changes between the two.

Example

The Chapter below contains a subroutine that calculates the function

$$f(n) = (n+1)/n!$$

for $n \geq 0$. The subroutine must be entered with N having the value n. The value of the function is then placed in F. Should n be outside the permissible range F is set equal to -1 (a value that it could not otherwise attain).

The chapter uses the subroutine to calculate and print the value of the expression

$$(2f(M) - f(2M))/3$$

It is assumed that M was assigned a value in a previous Chapter.

```
    CHAPTER 10
 7) N = M
    Jumpdown 20                          | Find f(M)
                Jump 2, F ≥ 0            | Found?
                Print ('M < 0')          | Error if M < 0
                Jump 4
 2) X = F
    N = 2M
    Jumpdown 20                          | Find f(2M)
    Print ((2X-F)/3) 0,6
 4) - - -
    - - -
    - - -

20) F = N+1                              | S/R ENTRY POINT
    Jump 21, N ≥ 2                       | End if
                Jump 22, N ≥ 0           | N = 0 or 1.
                F = -1                   | N < 0 is error
            22) Return
21) T = 2(1)N                            | Normal path:
    F = F/T                              | f(N) for N ≥ 2
    Repeat
    Return

    CLOSE
```


3146(10.65)

This subsection may be omitted on a first reading of Section 1.

An alternative form of the Jumpdown instruction is typified by

Jumpdown (I)

where I could be replaced by any index. This must have been set equal to a label (e.g. by "I) = 3)") before the Jumpdown was obeyed.

A common use of this form of the instruction is to enable a subroutine to call in another, auxiliary, subroutine that can be specified prior to entry. The label of the auxiliary subroutine can be put into an index, K say, before the initial entry to the major subroutine. Then the instruction "Jumpdown(K)" in this subroutine would call in the auxiliary subroutine. This technique is illustrated below.

Example

The following Chapter calculates and prints

$$\int_U^V \frac{x}{x+1}\,dx + \int_{U+1}^{V+1} x^2 \sin x\,dx$$

It includes a subroutine that uses Simpson's Rule to carry out the integration. This rule can be stated as

$$\int_{x_0}^{x_0+nh} f(x)dx = \frac{h}{3}(f_0 + 4f_1 + 2f_2 + 4f_3 + \ldots + 4f_{(n-1)} + fn)$$

where $f_t = f(x_0 + th)$ for $t = 0, 1, \ldots n$

The subroutine requires Y, Z, N to have been given the values x_0, $x_0 + nh$ and n respectively. N should be even; if it is not, the subroutine increases it by 1. The step length H is computed within the sequence. If this step length is less than 10^{-20} or if Z is less than Y, it is assumed that an error has occurred and suitable information is printed; otherwise the value of the integral is placed in A.

Before entering the integration subroutine it is also necessary to indicate what function is to be integrated by placing in the index S the label of a subroutine that calculates it. This subroutine must replace a given value of X by f(X).

CHAPTER 7

```
9) Y = U                    | Limits of
   Z = V                    | integration.
   N = 20                   | No. of steps
   S) = 10)                 | Function at 10)
   Jumpdown 30
   B = A                    | 1st integral in B

   Y = U+1                  | New
   Z = V+1                  | limits.
   N = 50                   | No. of steps
   S) = 11)                 | Function at 11)
   Jumpdown 30

   Print (A + B) 0, 4       | Sum of integrals

   Across 6/5
```

```
| INTEGRATION SUBROUTINE.  USES SIMPSON'S RULE.

30) Jump 32, N ≤ 0                          | Error?
    Jump 31, φparity(N) = 1
             N = N+1                        | Make N even.
31) H = (Z-Y)/N                             | Find step length
             Jump 32, H < 1& - 20           | Error?

    X = Y
    Jumpdown (S)                            | Find fo
    A = X

    X = Z
    Jumpdown (S)                            | Find fn
    A = A+X                                 | fo + fn

             W = Y
             I  = 2(1)N                     | N-1 values
             W = W+H                        | Add step length
             X = W
             Jumpdown (S)                   | Find fi
             A = A + (3+φparity(I))X        | 4fi or 2fi
             Repeat

    A = HA/3                                | Integral
    Return

32) Newline
    Print ('Error in integration sequence:  Y, Z, N=')
    Print (Y) 0, 10
    Print (Z) 0, 10
    Print (N) 1, 0
    Return

    | First function subroutine
10) X = X/(X+1)
    Return

    | Second function subroutine
11) X = XXφsin(X)
    Return


    CLOSE
```

An Alternative Technique 1.6.4

This subsection may be omitted on a first reading of Section 1.

There is an alternative technique that allows the return from the subroutine to be to any point in the Chapter, rather than always to the instruction after the one that calls in the subroutine. Prior to entering the subroutine the program must put an index, say I, equal to a label. Then the last instruction obeyed in the subroutine is not a Return but an instruction such as "Jump (I)" (see Section 1.3.2). Entry is by a Jump rather than by a Jumpdown.

Example

The first example of Section 1.6.2 is re-written using this technique:

```
    X = 91
    Y = 70

    K) = 15)                          | Set exit label
    Jump 10                           | Enter subroutine
    - - -
    - - -
15) - - -                             | Returns to here
    - - -
10) F = YYY - XXX
    G = (X-Y) /3
    Jump (K), X ≤ Y
                    F = -F
                    G = G+2
                    Jump (K)
```

The subroutine can be entered from any point in the Chapter by a jump to label 10. The return point is indicated by putting K equal to a label.

Chapters as Subroutines 1.6.5

A large subroutine can conveniently be written as one or more complete Chapters. The subroutine could then be more or less self-contained, with its own names for the main variables and its own sets of labels. Just as a sequence to be used as a subroutine within a Chapter is entered not by a Jump but by a Jumpdown instruction, so a Chapter used as a subroutine is entered not by an Across but by a Down instruction. For instance, if Chapter 2 of a program were to be regarded as a subroutine it could be entered at the instruction labelled 27) by

```
    Down 27/2
```

Exit from the subroutine would be by an instruction

```
    Up
```

which would return control to the original Chapter at the instruction after the Down instruction. Thus Down/Up pairs bear the same relation to Across as Jumpdown/Return pairs do to Jump.

When a Down has taken effect, the equivalent Up can be obeyed either in the Chapter called in or in a Chapter subsequently reached by an Across instruction, therefore with a multi-chapter subroutine control is normally transferred between Chapters of the subroutine by Across instructions.

π is re-set to 3.14159..... each time a Down or Up instruction is obeyed.

Example

Chapter 200 below is a subroutine to put X equal to an estimate of the limit of 101 given values. The subroutine requires the first 101 main variables to have been previously set equal to these values. Within Chapter 200 these variables are called A0 .... A100 but of course the programmer is at liberty to use other names in other Chapters.

Chapter 1 below reads two different sequences of numbers from a data document, uses Chapter 200 to calculate the two limits, prints their sum and transfers control to Chapter 2 (not present below).

```
    MAIN 600

    CHAPTER 200                       | Puts X = limit of first
    A → 100                           | 101 main variables.
 1) - - -
    - - -
    X = - -
    Up
    - - -
    CLOSE
```

```
    CHAPTER 1
    Z → 100                          | Equivalent to A's above
    Y → 100
    F → 200

10) K = 0                            | Use K as a marker
        12)N = 0(1)100
           Read (ZN)                 | Read 101 numbers
           Repeat
    Down 1/200                       | Find limit
    Jump 11, K ≠ 0                   | Jump 2nd time round
    Y = X                            | Y = 1st limit
    K = 1                            | Set marker non-zero
    Jump 12                          | Read 2nd set of numbers

11) Print (Y+X) 0, 7                 | Print sum.
    Across 9/2
31) - - -
    - - -
    CLOSE

    CHAPTER 0
    Across 10/1                      | Program entry point.
    CLOSE
```

Several Down instructions can be obeyed before an Up is reached. Each Down is said to lower the program logically, by one level. An Up causes an ascent of one level. A maximum depth can be specified by writing an initial directive such as

DEPTH 2

At the start of the run the program would be at depth, or level, 0. After the first Down had been obeyed it would be at level 1. An Up could now cause a return to level 0. Alternatively, a second Down could cause a descent to level 2; at this stage, because of the maximum depth imposed by the programmer, it would be an error to reach another Down before an Up is obeyed. The DEPTH directive is often omitted, in which case

DEPTH 3

is assumed.

The directive does not refer to the depth of Jumpdown/Return pairs, for which the appropriate directive is JUMPDOWN DEPTH (see Section 2.1.4).

Section 2

MORE ADVANCED FEATURES

ADDITIONAL SUBROUTINE FACILITIES 2.1

Routines 2.1.1

A subroutine to be entered by a Jumpdown instruction can either be labelled and written within a Chapter, as described in Section 1.6.2, or be written as a self-contained *Routine*.† A Routine is a block of instructions and directives headed by a directive of the form

ROUTINE 3

and followed by a directive consisting of two or more asterisks on a line to themselves:

**

Routines can have any numbers in the range 0 to 4095.

Entry is by an instruction of the form

Jumpdown (R5)

which would enter Routine 5 at its first instruction or by one of the form

Jumpdown (R5/27)

which would enter the same Routine at the instruction labelled 27. As with the simple type of Jumpdown sequence, exit is by the instruction

Return

A particular Routine will appear only once on a program document. It can then be called in by Jumpdown instructions in any Chapters or Routines. These instructions may be considered to cause a copy of the Routine to be inserted into each Chapter that requires it. If one Routine contains a Jumpdown instruction to another Routine, then a Jumpdown instruction to the first will cause both Routines to be available to that Chapter.

There are two restrictions on the written position of a Routine:

(a) it must not be written inside a Chapter or another Routine

(b) it must be written after the initial directives.

The normal practice is to group all the Routines of a program immediately after the initial directives. They need not be in numerical order.

The advantages of a Routine over the other type of Jumpdown sequence are that it need be written down only once, no matter how many different Chapters require it, and that it is so much more self-contained that it can easily be written to be useful in many different programs (see especially Sections 2.1.2 and 2.1.3 below).

† The term *Routine* for a particular type of subroutine is unfortunate but standard.

Examples

Each of the following two examples† includes a complex of subroutines to evaluate the hyperbolic functions:

$\sinh(x) = (e^{x} - e^{-x})/2$

$\cosh(x) = (e^{x} + e^{-x})/2$

$\tanh(x) = \sinh(x)/\cosh(x)$

(a) The following Routine has a separate entry point for each of the three operations.

```
ROUTINE 60                          |HYPERBOLIC FUNCTIONS.  Uses U, U', E, E'
|Sinh(X) => U.    Entry:  Jumpdown (R60) or (R60/1).
                  1) Jumpdown 4
                     U = 0.5(E-E')
                     Return

| Cosh(X) => U.   Entry:  Jumpdown (R60/2).
                  2) Jumpdown 4
                     U = 0.5(E+E')
                     Return

| Tanh(X) => U.   Entry:  Jumpdown (R60/3).
                  3) Jumpdown 1
                     U' = U
                     Jumpdown 2
                     U = U'/U
                     Return
                  4) E = φexp(X)
                     E' = 1/E
                     Return

                     **
```

(b) The four labelled sequences of the above Routine are re-written unrealistically below as four separate Routines:

```
                  ROUTINE 61

| Sinh(X) => U.   Entry:  Jumpdown (R61)
                  Jumpdown (R59)
                  U = 0.5(E-E')
                  Return

                  **

                  ROUTINE 62

| Cosh(X) => U.   Entry:  Jumpdown (R62)
                  Jumpdown (R59)
                  U = 0.5(E+E')
                  Return

                  **
```

† Both are based on examples in the CHLF3 Programmer's Notes issued by the University of London Institute of Computer Science.

3146(10.65)

```
                    ROUTINE 63

|Tanh(X) => U.      Entry:  Jumpdown (R63)
                    Jumpdown (R61)
                    U' = U
                    Jumpdown (R62)
                    U = U'/U
                    Return

                    **

                    ROUTINE 59

|Used by R61, R62, R63
                    E = ϕexp(X)
                    E' = 1/E
                    Return

                    **
```

Routine 59 is unlikely to be called from anywhere but Routines 61 and 62. Notice that Routine 63 uses Routines 61 and 62 as subroutines.

Suppose that the four Routines of example (b) appear in a two-chapter program as follows:

```
MAIN 900

ROUTINE 61
- - -
**

ROUTINE 62
- - -
**

ROUTINE 63
- - -
**

ROUTINE 59
- - -
**

CHAPTER 1
- - -
Jumpdown (R61)                  |Find sinh(X)
- - -
CLOSE

CHAPTER 0
- - -
Jumpdown (R63)                  |Find tanh (X)
- - -
CLOSE
```

Assuming that the Chapters contain no other calls for the Routines, the translated program can be thought of as having the following form:

CHAPTER 1
- Body of Chapter
- Copy of R61
- Copy of R59

CHAPTER 0
- Body of Chapter
- Copy of R63
- Copy of R61
- Copy of R62
- Copy of R59

The actual order in which Routines are inserted in a Chapter is undefined and irrelevant.

A Routine can contain all those instructions and directives that are legal in a Chapter, with the one exception of the Variables directive. Those listed below should be used only by programmers who fully understand their effects within a Routine. (Not all have yet been described.)

ACROSS
CLEAR LEVEL
DOWN
FAULT n
INDEX STORES
JUMP (I)
JUMPDOWN (I)
Label-setting instruction (e.g. I) = J))
Set directives (e.g. A→9)
I or V = ϕTABLE (J, n)
UP
UP LEVEL

A Repeat instruction must be in the same Routine as the corresponding loop instruction.

Variables and Routines 2.1.2

The values of the variables and indices are not changed on entry to a Routine. π is not re-set. A copy of a Routine usually refers to the main variables defined in the containing Chapter. Thus, if a Routine used, say, variables A0 to A10, each Chapter calling that Routine would normally have a Set directive "A→10" (at least).

As an alternative this directive could be written within the Routine. Any Set directives in a Routine act as though they followed on from those of the containing Chapter, regardless of any directives in other Routines. It follows that Set directives in different Routines of a Chapter define alternative names for the same group of variables. This group consists of any main variables reserved in the Main directive but not allotted names in the Chapter proper. Should a Set directive in a Routine refer to the same letter as a directive in the Chapter the situation is similar to that described in Section 1.5.4, i.e. within the Routine the earlier set can be referred to only indirectly.

Consider the following, extreme, example:

```
MAIN 300

ROUTINE 10
A→49
B→49
- - -
**
```


3146(10.65)

```
ROUTINE 11
C→49
- - -
**

ROUTINE 12
- - -
**

CHAPTER 1
C→199
Jumpdown (R10)
- - -
Jumpdown (R11)
- - -
CLOSE

CHAPTER 0
X→49
C→199
Jumpdown (R11)
Jumpdown (R12)
- - -
CLOSE
```

Without the aid of overlapping suffixes the variables available in different parts of this program are as follows:

Chapter 1	:	C0, ... C199
Routine 10 in Chapter 1:		C0, ... C199, A0, ... A49, B0, ... B49
Routine 11 in Chapter 1:		C0, ... C49
Chapter 0	:	X0, ... X49, C0, ... C199
Routine 11 in Chapter 0:		X0, ... X49 C0, ... C49
Routine 12 in Chapter 0:		As Chapter 0.

Where different names are applied to the same variable they have been written in corresponding positions. Notice that Routines 10 and 11 of Chapter 1 have a block of 50 variables in common but refer to them by different names.

The only advantage of writing Set directives within Routines is that the Routines of a Chapter can then share a block of main variables which can be renamed for each Routine. The technique is most useful when the variables will not have any useful values on exit from the Routine.

Complicated examples such as that above are likely to occur only when previously written Routines are incorporated in a new program. Even then it would not necessarily seem complicated to the programmer. He would probably need to know only the number of main variables defined in a Routine and not their names.

The number of main variables required by a Chapter is the number defined in the Chapter proper plus the maximum of the numbers defined in its associated Routines. The Main directive must reserve at least as many variables as are required by any Chapter of the program. It is conventional to add a safety margin of 20 to allow for any mis-counting.

VARIABLES directives are not allowed in Routines.

Labels and Routines 2.1.3

A Routine can have its own set of labels running from 0) to 127) and these are quite distinct from those in any other Routine or Chapter. Labels 0) and 100) have no special uses in a Routine, so they may be used as normal labels.

When label numbers are written explicitly in instructions such as

```
Jump 7, X > Y
Jumpdown 10
```

within a Routine, then the effect is to jump to a labelled instruction in the Routine. Referring to an index which has been set equal to a label has a less obvious effect and is best avoided in Routines by all but experienced programmers.

In fact, inexperienced programmers should now jump straight to Section 2.1.5.

Though the boundaries between a Chapter and its Routines and the boundaries between different Routines are normally crossed only by Jumpdown (Rn) and Return instructions, this is not invariably so. Most Label-setting instructions will be taken to refer to a label of the Chapter even if they appear in a Routine: for instance, the sequence

```
K = 10
J) = K+3)
Jump (J)
```

will always cause a jump to label 13 of the current Chapter, even if a non-standard exit from a Routine is caused.

The only form of label-setting instruction whose effect is localized is that typified by

```
I) = 3)
```

where only a single, unsigned, written integer appears on the right. If obeyed in a Routine this instruction would set I equal to label 3 of that Routine and a subsequent instruction

```
Jump (I)
```

or

```
Jumpdown (I)
```

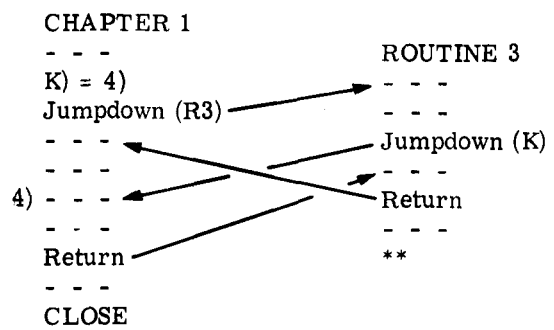
obeyed either in the same Routine or in another Routine of the Chapter or in the Chapter proper would cause a jump or jumpdown to label 3 of the Routine.

Non-standard entries and exits from Routines have their dangers and are only occasionally useful; Section 2.1.4 should be consulted before they are programmed.

Jumpdown Levels 2.1.4

The concept of the *level* of execution of a subroutine was introduced in connection with Down/Up instruction pairs. A related, though separate, system applies to the operation of Jumpdown/Return pairs. A program is said to be at Jumpdown level 0, or Chapter level, until a Jumpdown instruction is obeyed. Each such instruction, whether it calls in a Routine or simply a labelled sequence, causes the program to descend one level, while each Return instruction causes the program to ascend one level; if two Jumpdown instructions have been executed without any Return instructions having been reached, then the program is at level 2.

This is illustrated diagrammatically by a piece of program consisting of a Chapter that calls in (and therefore contains a copy of) one Routine, which in turn calls in an auxiliary sequence from the Chapter:



3146(10.65)

Assuming for the moment that no other Jumpdown instructions appear in the above section of program, the instruction "Jumpdown (R3)" causes the program to descend to level 1; the instruction "Jumpdown (K)" causes a descent to level 2 at label 4); the Return on the left causes a return to level 1; then the other Return causes an ascent to level 0 again.

The level at which a sequence or a Routine is obeyed is purely a function of the order in which instructions are executed, not of the way in which they are written down; for instance, if an instruction

Jumpdown 4

were obeyed in the left-hand sequence of the above example, then the instruction labelled 4) would be obeyed at level 1 this time instead of level 2.

There is a limit to the number of levels that a given program may descend. Level 8 is normally the lowest but an alternative can be specified by an initial directive such as

Jumpdown Depth 4

With this directive, if more than four Jumpdown instructions were obeyed without a Return being reached the program would be stopped.

The current Jumpdown level is independent of chapter changing caused by Across, Down or Up instructions. It is, however, essential that a Return instruction should be obeyed in the same Chapter as was the corresponding Jumpdown, even if Chapter changes have taken place meanwhile. "Chapter" is here taken to include all Routines inserted in the Chapter.

Though most programmers need to be conscious of Jumpdown levels only when they accidentally exceed the Jumpdown depth of their program, complicated patterns of level changing do sometimes arise and it can be desirable to change the Jumpdown level without obeying the relevant Return instructions. For instance, if the auxiliary sequence above was required to return control to the instruction after "Jumpdown (R3)" without first returning to the Routine, one way of doing this would be to obey the instruction

Up Level

before a Return was reached in the auxiliary sequence. Then the Jumpdown level would be reduced by 1, the record of the last Jumpdown instruction, i.e. "Jumpdown (K)", would be wiped out and the Return would have the required effect.

"Up Level" always causes the program to ascend one level and cancels the record of the last Jumpdown for which a Return has not yet been executed. The instruction

Clear Level

returns the program to Jumpdown level 0 and cancels the records of all outstanding Jumpdown instructions, if any. It would then be a fault to obey a Return instruction before the next Jumpdown. Neither Clear nor Up Level actually transfers control but both can affect the future flow of control of a program.

The value of the current Jumpdown level can be obtained in an index by an instruction such as

Test Level (K)

Programmes 2.1.5

Large subroutines can be conveniently packaged as numbered *Programmes.* A Programme consists of one or more chapters headed by a Programme directive, typified by

PROGRAMME - 2001

The end of a Programme is marked by the start of another one or the occurrence of a Chapter 0. A Programme would better have been called a subprogram; the pronunciation "program - ee" is often used to distinguish "Programme" from "program".†

† "Program" is the spelling recommended by the British Standards Institution.

Programme numbers can be in the range -0 to -4095. Chapters in each Programme are numbered independently of those in other Programmes. A possible layout for a complex program could be:

```
PROGRAMME -10
CHAPTER 1
- - -
CLOSE

CHAPTER 2
- - -
CLOSE

PROGRAMME -871
CHAPTER 1
- - -
CLOSE

CHAPTER 2
- - -
CLOSE

PROGRAMME -850
CHAPTER 1
- - -
CLOSE
- - -
- - -

CHAPTER 5
- - -
CLOSE

CHAPTER 0
- - -
CLOSE
```

The first Programme would have been specially written for the problem, while the other two would be drawn from a Library.

Entry to a Programme is by an instruction such as

```
Down    101/1-850
```

or

```
Across  101/1-850
```

either of which would enter Chapter 1 of Programme-850 at the instruction labelled 101. Exit is by an Up or Across instruction. Within a Programme the simpler forms of Down and Across instruction refer to Chapters of that Programme. It follows that instructions

```
Down 77/2
```

and

```
Down 77/2 - 871
```

obeyed in Programme -850 would not have the same effect: the first would transfer control to Chapter 2 of Programme -850, the second would transfer control to Chapter 2 of Programme -871.

Similarly, the Variables directive normally refers to a previous Chapter of the same Programme. However, it is possible for one Programme to refer to the variable settings of another by means of a directive such as

```
VARIABLES 3-850
```

which means 'transfer the variable settings of Chapter 3 of Programme 850 to the current Chapter of the current Programme'.

Any Chapters without a Programme heading are automatically taken to form Programme -0. Consider the following program:

```
CHAPTER 1
- - -
CLOSE

CHAPTER 2
- - -
CLOSE

PROGRAMME -862
CHAPTER 1
- - -
CLOSE

CHAPTER 2
VARIABLES 1
- - -
CLOSE

CHAPTER 0
VARIABLES 1
- - -
CLOSE
```

It consists of two Programmes: the explicit Programme -862 containing Chapters 1 and 2 and the implicit Programme -0 containing Chapters 1, 2 and 0. Notice that "VARIABLES 1" in Chapter 0 refers back to a Chapter in the same Programme, i.e. to the first Chapter 1. For an Across or Down instruction in Programme -862 to transfer control to the first Chapter 2 it would have to be written in a form such as

```
Across 70/2-0
```

However, since Chapter 0 is unique and always in Programme -0 it can be transferred to from any Programme without a Programme number being specified.

DUMPING TECHNIQUES 2.2

General 2.2.1

It is in the nature of things that a programmer often wants to use the same indices or variables for several different purposes. To take a trivial case, he might want to use T to represent a temperature and also to use a subroutine in which T represents a time. Such conflicts are often resolved by the introduction of *dumps*. The values of a set of indices or variables can be preserved in a dump while the set is temporarily used for another purpose. The original values will be restored when required. Several types of dumps will be described.

Index Stores 2.2.2

A program can have available as a dump a number of *index stores*, each one of which can hold four index values. The stores are identified by integers from 0 upwards. An instruction to dump the values of the four consecutive indices K, L, M and N in index store number 6 would have the form

```
Preserve Indices (K, 6)
```

The values could be returned by the instruction

```
Restore Indices (K, 6)
```

For the purpose of these instructions the full set of indices should be regarded as being two distinct, non-adjacent sets, I J K L M N O P Q R S T and I' J' K' L' M' N' O' P' Q' R' S' T'. Any four consecutive indices of a set can be preserved or restored but no overlapping between sets is allowed. Since a reference to Q or Q' in a Preserve or Restore Indices instruction preserves or restores RST or R' S' T', there is never any reason to refer directly to these six indices in such instructions, and it would be an error to do so.

A Preserve Indices instruction does not alter the values of the indices and a Restore Indices instruction does not alter the values held in the index store; the following sequence would copy the values of I J K L to both M N O P and Q R S T:

```
Preserve Indices(I, 0)
Restore Indices (M, 0)
Restore Indices (Q, 0)
```

(This is not necessarily the best method for the operation.)

The number of index stores available is controlled by a directive such as

```
INDEX STORES 10
```

which means that eleven index stores numbered 0 to 10 are to be available, i.e. there are to be sufficient stores to hold forty-four index values. The directive is best written immediately after the Set directives at the head of a Chapter, e.g.

```
CHAPTER 5
A ⟶ 499
U ⟶ 499
- - - - -
INDEX STORES 9
```

The stores are available only in the Chapter in which the directive appears. An Index Stores directive can be thought of as a special form of Set directive, so a directive

```
VARIABLES 5
```

written in, say, Chapter 6 would ensure that this Chapter had available the same index stores as Chapter 5.

Each Chapter or Routine will normally contain at most one Index Stores directive. The same rule applies as with ordinary Set directives: the index stores available to a Routine are those of the Chapter containing it unless the Routine itself contains an Index Stores directive.

Index stores are continuous with the main variables and the Main directive of a program must take account of them, one index store being equivalent to four main variables; thus, a program containing the above Chapter 5 would need a directive of at least

```
MAIN 1040
```

i.e. 500 + 500 + 10×4. The programmer must not refer to part of an index store by extending main variable suffixes beyond their specified limits.

Suppose that a Routine takes a value of S, carries out some calculation and puts S equal to the result, and that in the course of the calculation it destroys the values of J, K, L and M. A programmer using the Routine would have the choice either of ensuring that these indices never have any useful values immediately prior to entering the Routine or of preserving their values during its execution. The second alternative could appear as:

```
ROUTINE 3
| Puts S equal to f(S).   Uses J, K, L, M.
- - -
Return
**

CHAPTER 2
INDEX STORES 0
- - -
S = 10
Preserve Indices (J, 0)
Jumpdown (R3)
Restore Indices (J, 0)
Print (S) 1, 0
- - -
CLOSE
```

On the other hand, the Routine could have its own index stores and be almost completely self-contained:

```
ROUTINE 3
| Puts S equal to f(S).   Contributes 4 to MAIN.
INDEX STORES 0

Preserve Indices (J, 0)
- - -
- - -
Restore Indices (J, 0)
Return
**

CHAPTER 2
- - -
S = 10
Jumpdown (R3)
Print (S) 1, 0
- - -
CLOSE
```

Once the Routine has been written in this form, the programmer can forget that it uses J, K, L and M. The disadvantage of this method is that the preservation and restoration take place each time the Routine is entered, whether the four indices have useful values or not.

The Auxiliary Variables 2.2.3

The earlier versions of Mercury Autocode were designed for use with computers with relatively small working stores. These stores were capable of holding EMA indices, special variables and a limited number of main variables, but a backing store had to be used to accommodate really large amounts of data.

Data was held in this backing store in the form of *auxiliary variables* and it was necessary to copy such variables into the working store before performing calculations upon them. They were normally introduced for any of three reasons:

3146(10.65)

(a) to act as a dump for the main (and special) variables

(b) to receive results generated between the execution of Preserve and Restore instructions

(c) to hold matrix elements.

Although a 1900 EMA programmer is unlikely to use a set of auxiliary variables as a dump - he will normally specify extra main variables for this purpose - the instructions required for (a) above are available and are described in the remainder of this section.

These same instructions will be required for (b) above, as explained in Section 2.2.4.

The most likely use of auxiliary variables in 1900 EMA is given in (c) above. The instructions for matrix operations are all written in terms of auxiliary variables and they will be explained in Section 3.2 which contains a full list of the instructions affecting auxiliary variables.

Auxiliary variables are held to the same precision as main or special variables but are identified by positive or negative integers, not by letters. No confusion with integer constants arises as auxiliary variables can be referred to only in certain instructions and not in arithmetic expressions. The size of the set is defined by an initial directive of the form

```
Auxiliary (-1000, 7050)
```

which would make available 8051 variables numbered -1000 to 7050. If the directive is omitted, no storage is reserved for auxiliary variables.

Values can be copied to a block of auxiliary variables from a block of main variables by a ϕ7 instruction such as

```
φ7 (1021) X10, 1000
```

which would copy the values of the 1000 main variables X10, ... X1009 to the auxiliary variables 1021, ... 2020. The ϕ6 instruction has the reverse effect, e.g.

```
φ6 (850) U25, 30
```

would copy the variables 850, ... 879 to U25, ... U54. Values can also be transferred to or from the special variables, but only one at a time; e.g. the sequence

```
U = 92.73
φ7(17) U, 1
```

would assign the value 92.73 to auxiliary variable 17. ϕ6 and ϕ7 simply carry out copy operations and do not affect the variables whose values are being copied.

Note that ϕ6 and ϕ7 are not EMA functions but are instructions and as such must be written one to a line. Their arguments may be fully general, e.g. if I, J, K have the values 1, 7, 5 then the instruction

```
φ6 (52I + J - K) π(J + 3), I - J + 10
```

is equivalent to

```
φ6 (54) π10, 4
```

Here, the value of the expression 52I+J - K is taken to define a particular auxiliary variable; similarly the value of A' in the instruction

```
φ7(A') B', 1
```

defines a particular auxiliary variable and A' is not itself altered.

There are instructions to read to and print from the auxiliary variables. ϕ10 is the read instruction. To read 2000 numbers to variables -100, ... 1899 it would be written as

```
φ10 (-100, 2000)
```

The numbers would appear on a data document exactly as though they were to be read by a series of ordinary Read instructions.

The print instruction is ϕ8, e.g.

```
φ8 (100, 1, 6, 3, 3)
```

Ignoring the integer "1" for the moment, the instruction means: print the values of the 6 variables 100, ... 105; begin each number on a new line; print each number with 3 integer and 3 fractional places, exactly as with "Print(X)3, 3". The resultant printing could be

```
 123.456
 987.654
-330.033
 214.320
 -14.172
 693.400
```

The second and third arguments of the instruction control the insertion of clear lines after groups of numbers, e.g. the instruction

ϕ8(100, 3, 2, 3, 3)

would print the same numbers as before but in 3 groups of 2 with a clear line between each group:

```
 123.456
 987.654

-330.033
 214.320

 -14.172
 693.400
```

Thus the instruction

ϕ8(A, P, Q, M, N)

outputs PQ numbers and a total of P(Q+1)-1 lines.

ϕ9 is a variant that prints numbers in floating-decimal form:

ϕ9(A, P, Q, N)

is precisely equivalent to

ϕ8(A, P, Q, 0, N)

Example

Chapter 30 is a subroutine that destroys the values of the special variable X and of the first 100 main variables. If these values are useful then they must be dumped during execution of the subroutine:

```
   MAIN 300
   AUXILIARY (0, 100)

   CHAPTER 30                    | Uses X and 100 main variables
   A → 99
1) - - -
   UP
   CLOSE
```


3146(10.65)

```
CHAPTER 0
Y →199
Z →80
- - -
φ7(0) X, 1          | Dump X
φ7(1) Y0, 100       | Dump first 100 main variables
Down 1/30           | Enter subroutine
φ6(0) X, 1          | Restore X
φ6(1) Y0, 100       | Restore main variables.
- - -
CLOSE
```

It would have been better if Chapter 30 had been written to use only a continuous set of main variables, so that only one $\phi 6$ and $\phi 7$ would be needed for each call of the subroutine.

Preserve and Restore 2.2.4

Before entering a large subroutine such as a Programme it is often convenient to dump the bulk of the working store. The initial directive

DUMPS 3

would make available in the backing store dumps sufficient to hold copies of the working store three times over. These dumps are quite separate from any auxiliary variables that may also be in the backing store. If the directive is omitted, no dumps are made available.

To use a dump, the programmer would write the one-word instruction

Preserve

This would dump a copy of the values of the special variables (including π' but not π), the main variables (including the index stores) and the unprimed indices I to T. When the dumped values are again required, the instruction

Restore

must be obeyed. This copies values from a dump back to the working store, overwriting any values that have been generated there meanwhile.

π and the primed indices are not dumped. Preserve and Restore both re-set π to 3.14159... Neither instruction affects the values of I' to T'.

Preserve and Restore are usually used immediately before and after a Down instruction, so each dump is associated with a particular Down level (Section 1.6.5). At the start of a program a Preserve or Restore would use the first dump. If a Down instruction reduced the Down level by 1, a Preserve or Restore obeyed before the next Up would use the second dump. It follows that a dump can be used at Down level m only if $m+1$ dumps are available.

The only snag with using Preserve and Restore instructions before and after execution of a subroutine is that Restore would destroy any results left in the working store by the subroutine. (The primed indices could be used to hold the results, but only if they were all integers.) For this reason, large subroutines such as Programmes are usually designed to place results in the auxiliary variables. They can then be extracted after the restore operation. For maximum flexibility such subroutines are often written so that a parameter can be set to specify which auxiliary variables are to receive the results. A Programme that generates a single result in X might have as its last instruction before "Up" the instruction

φ 7(U) X, 1

where U is one of the parameters to be set before the Down instruction calling in the Programme;

thus, if the programmer wants the result placed in auxiliary variable 15, the calling-in sequence could be written as

Preserve	Dump copy of working store.
U = 15 M = 1 N = 3 X = 17.0	Parameters required by Programme.
Down 101/1 - 2055	Places result in X and auxiliary variable 15.
Restore	X overwritten.
ϕ6(15)Z, 1	Result copied to Z.

Preserve and Restore are relatively slow operations and should be avoided in inner loops.

Programme parameters

A, B	limits of integration
E	limit of accuracy
U'	specifies auxiliary variable for result: final accuracy placed in U' + 1

Variables and Indices Used

(a) The first 40 main variables

(b) A, B, E, X, U'

(c) I, J, K, L, M

(d) Two auxiliary variables specified by user

Entry "Down 1/1-502". An alternative entry, "Down 2/1-502", provides a printed record of the integration process.

Auxiliary Sequence

1 Must replace X by f(X).

2 Conventional entry to first instruction.

3 Exit to label 101)

4 Labels 3) to 99) available.

5 40 main variables already named.

6 None of the variables or indices listed as used by the Programme may be changed by the auxiliary sequence. All others can be used freely as working variables or "by-pass" parameters.

The above Programme is to be used to evaluate the m values of

$$\int_{a_{(n-1)}}^{a_n} f(x)dx \qquad n = 1, 2, \ldots . m$$

The values of m ($m \leqslant 60$) and a_0 to a_m are to be read from a data document. The program is required to choose between three possible functions f(x), the criterion being the size of z, the average value of a_0 to a_m:

$f(x) = e^{x \sin x} - 1$ if $0 \leqslant z \leqslant 10$

$f(x) = x \log (x+1)$ if $10 < z \leqslant 30$

$f(x) = (x+1)(x-10)^2(3x^3-1)^{16}$ if $30 < z$

The auxiliary sequence will thus be effectively three sequences in one. The value of z will be calculated before the Library Programme is called in and index T will be used as a "by-pass" parameter to indicate which function is required. The program below has 63 main variables available. Up to 61 of these will hold the values of a_0 to a_m. The first 40 will also be used by the Programme. Because of this conflicting use, the working store will be dumped before each entry to the Programme.

The full program could thus be:

```
MAIN 63
AUXILIARY (1, 2)                    | For Programme results
DUMPS 1

CHAPTER 1
A → 60
F → 1

1)Read (M)                          | Across from Chapter 0.
  Jump 2, M ≤ 60
          Print ('M too large:')    | Error-stop if M
          Print (M) 1, 0            | too large.
          End

2) Z = 0
  N = 0(1)M                         | Read M+1 numbers to AO-AM and form
  Read (AN)                         | sum in Z
  Z = Z + AN
  Repeat

  Z = Z / (M+1)                     | Average in Z

  Jump 3, Z ≥ 0
          Print ('Average < 0:')    | Error-stop if average negative.
          Print (Z) 0, 6
          End

3) Jump 4, Z >10                    |
          T = 1                     |
          Jump 6                    |
                                    |
4) Jump 5, Z > 30                   |
          T = 2                     | Set function parameter in T
          Jump 6                    |
                                    |
5) T = 3                            |

6) Printline
  FUNCTION NO.
  Print (T) 1, 0
  Printline
  ACCURACY (Unless otherwise stated): 0.0001
  Printline
  Lower Limit     Upper Limit     Result     Accuracy

  N = 1(1)M
  A = A(N-1)                        | Successive lower and upper limits
  B = AN
  Newline
  Print (A) 0, 6
  Space 4
  Print (B) 0, 6
  Space 4
  Jump 10, B ≥ A
           Print ('ERROR')          | Error stop if B < A
           End
```

```
10) Preserve                          | Dump working store
    E = 0.0001                        | Accuracy
    U' = 1                            | Result wanted in aux. variable 1
    Down 1/1-502                      | Call Programme
    Restore                           | Restore working store
    ϕ6 (1) F0, 2                      | Obtain result and accuracy in F0, F1

    Print (F0) 0, 6                   | Print result
    Jump 15, F1 = 0.0001              | Print accuracy
             Space 4                  | if not 0.0001.
             Print (F1) 0, 6

15) Repeat                            | Next pair of a(n-1), an

    End
    CLOSE

    PROGRAMME - 502                   |
    CHAPTER 1                         |
    - - -                             |
    - - -                             | Library Programme
    - - -                             |
    AUXILIARY SEQUENCE                |
    π → 1

    Jump 6, T ≠ 1
    X = ϕexp (X ϕsin (X))-1
    Jump 101

 6) Jump 7, T ≠ 2
    X = Xϕlog (X+1)
    Jump 101

 7) π 0 = X+1
    π 1 = X-10
      X = 3XXX-1

    P = 1(1)4
    X = XX                            | Raise to 16th power.
    Repeat

    X = π0 * π1 * π1 * X

    Jump 101

    CLOSE

    CHAPTER 0
    Across 1/1
    CLOSE
```

Notes on the above program:

(a) U' and E are set after the Preserve operation; A and B are set before it. If the programmer's Chapter 0 or 1 had used A and B for other purposes, they would necessarily have been set equal to the parameters after the Preserve operation had taken place.

(b) π0 and π1 were introduced to hold intermediate results in the auxiliary sequence. Because of the dumping that takes place, it is quite unimportant to the programmer that they are the same variables as A40 and A41 in the first Chapter 1.

(c) A pair of Preserve and Restore instructions are obeyed for each call of the Programme. Since relatively little computation takes place in the user-provided sections, a little care could lead to the elimination of these operations. The technique described in Section 2.2.3 could be used to dump the values of A0 and AM once-for-all in a set of auxiliary variables. These values could then be copied back to the working store. Alternatively, the program could start:

```
MAIN 104
AUXILIARY (1, 2)
DUMPS 0
CHAPTER 1
F → 41
A → 60
- - -
```

The Library Programme and auxiliary sequence use only the first 42 main variables, so A0 to A60 would be undisturbed. F0 and F1 would be used as before. F2 to F41 are "dummy" variables, unused in this chapter.

With either alternative the program would necessarily be re-written to eliminate conflicting uses of the special variables and indices. For instance, M in the programmer's Chapter 1 could be replaced by S.

When deciding whether or not to use Preserve and Restore the programmer must balance the extra backing store required for the dumps and the time required to perform the transfers of all the working store against the store and time required by alternative methods. However, many problems will be so complex that it will not be worth spending a lot of programming time trying to avoid the use of Preserve and Restore instructions.

MANIPULATION OF CHARACTERS 2.3

Reading and Printing Characters 2.3.1

Although most of the instructions introduced so far have been for the manipulation of numbers, certain of them have referred to non-numeric data. For instance, the line

```
Print ('JONES J.')
```

is an instruction to print non-numeric data consisting of the eight characters "J", "O", "N", "E", "S", space, "J" and ".". Further facilities for the reading, printing and manipulation of data in character form will now be introduced.

The instruction

```
Read Ch (J)
```

means: read to the index J the next printed character from the data document. When it has been obeyed, the value of J will, as always, be an integer but this will be a code for whatever printed character was read; e.g. if the line

```
QA*
```

had been read by the instructions

```
Read Ch (I)
Read Ch (J)
Read Ch (K)
```

then the indices I, J and K would contain codes for the characters "Q", "A" and "*" respectively.

There is a similar instruction for printing characters one at a time. Should the sequence of character -reading instructions above be immediately followed by

```
Print Ch (K)
Print Ch (I)
Print Ch (J)
```

then the output document would contain

```
*QA
```

EMA Character Codes 2.3.2

Although each of the character codes used in EMA corresponds to an integer value and can sometimes be treated as such (see 2.3.5), the programmer can refer to a particular character without knowing its code. This is achieved by use of the integer function, ϕ code, which has as its value the code of a single, specified character. For instance, ϕ code (*) provides the code of the character "*" and the instruction "I = ϕcode (*)" sets I equal to this code.

Example

```
Read Ch (M)                    | Read 1 character.
                               |
Jump  9, M = φcode(Y)          | To  9) if "Y" was read
Jump 10, M = φcode(N)          | To 10) if "N" was read
Jump 11, M = φcode(?)          | To 11) if "?" was read
Jump 101                       | To 101) otherwise
```

Here a character is read and the unknown integer value which corresponds to its character code is assigned to M. The next three instructions compare this integer value with the integer character codes for "Y", "N" and "?" and cause a jump if equality is found. Note that at no stage need the programmer know the codes for "Y", "N" and "?".

It should be emphasized that EMA codes are not the same as the 1900 internal codes nor are they any paper tape or punched card code. Each character is represented by an integer in the range 0 to 4095 (corresponding to 12 binary digits). The fact that this character may be represented by one or more entirely different codes on, say, paper tape is irrelevant.

The obvious advantage of using the ϕ code function rather than the explicit numerical code when referring to a character is that a program written in these terms can be run on computers employing different types of codes.

A full list of characters that may appear in the ϕ code function is given in Section 2.3.4.

Spaces and Newlines 2.3.3

Some characters must be represented in the ϕ code function by abbreviations. The two most common are ϕcode(SP) and ϕcode(NL). SP represents a single space character. NL represents the newline character, i.e. the character that is printed to terminate one line and start the next.

All space and newline characters in a data document can be obtained by character reading instructions. Thus,

```
J F

K
```

would probably be read as "J", space, space, "F", newline, newline, "K". The only ambiguity is that superfluous spaces might have been printed between "F" and the first newline character, or between the two newline characters (i.e. on the blank line). These would all have been obtained by Read Ch instructions. The following is a very common sequence of instructions to read and ignore spaces and newlines until until the first visible character.

```
1)Read Ch (K)
  Jump 1, K = φcode (SP)
  Jump 1, K = φcode (NL)
  - - -
```

The different treatment of spaces (and newlines) by the character and numeric Read instructions can be illustrated by considering the effect of different instructions on the line

```
92   3
```

where there are three spaces between "2" and "3". If this were read by

```
Read (I)
Read Ch (J)
Read Ch (K)
```

the indices I, J and K would have the values 92, ϕcode (SP) and ϕcode (3) respectively. Notice that the first two spaces after 92 have been taken to be part of the number by the numeric Read instruction. On the other hand, if the instructions had been

```
Read Ch (P)
Read Ch (Q)
Read (R)
```

then the indices P, Q and R would have been given the values ϕcode(9), ϕcode(2) and 3, respectively. The three spaces after 2 have been skipped by the numeric Read instruction, as would have been any newline characters that occurred there.

The instructions

```
Print Ch ( φcode(SP) )
Print Ch ( φcode(NL) )
```

are equivalent to the instructions

```
Space
Newline
```

This subsection requires some knowledge of the input and output media available for EMA data and results. It should be omitted until Section 4 has been read.

The characters with EMA codes are in two sets: those that can appear as a single character in the ϕcode function form the *main set*; those that can appear in the ϕcode function only as a two-letter abbreviation form the *special set*. All characters in the two sets can appear on eight-track tape, but other types of input/output media have various subsets (see Section 4).

Main Set

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
π ϕ + - * / > ≥ < ≤ = ≠
. , : ; ? & ' @ " ß ↑ ← →
( ) [ ]

This list includes all the characters required for writing an EMA program together with other characters from the standard 1900 input/output media. Note that several of these characters are actually represented on particular I/O media by quite different characters, e.g. π is £, → is —>.

In 1900 EMA, upper- and lower-case letters are identical as far as the ϕcode function is concerned. For example, if ϕcode (A) is compared with an "A" or an "a" read from an input medium, equality will result.

Special Set

Character	Abbreviation
Space	SP
Newline	NL
Tabulate	TB
Vertical Bar(!)	VB
Erase	ER
Paper Throw	PT
Stop Code (TC_4)	ST

Essentially, the special set of characters is provided to permit programmers to output paper tape control characters by means of ϕcode functions. They are seldom used in other circumstances because by the time a Read Ch instruction takes effect, most of the above characters will have been either edited out or replaced by one or more other characters.

Details of each of the characters in the special set are given below.

Space and Newline See Section 2.3.3.

Tabulate This character can be output to eight-track tape instead of a number of space characters. The appropriate instruction is

```
Print Ch (ϕcode (TB))
```

If a Tabulate character is read, it is treated as two separate space characters. Therefore it is not possible to use ϕcode (TB) to test whether or not a Tabulate character has been read.

Vertical Bar This character can be output ahead of a comment on a document to be subsequently used as an input document.
For example, the sequence

```
Print (X) 3,2
Print Ch (ϕcode (VB))
Print ('PRICE')
```

could print

```
684.32 | PRICE
```

If a Vertical Bar is read, it is treated as a newline character and the remainder of the line is ignored. (In practice, 1900 EMA prints Vertical Bar as !)

Erase This character may be output to paper tape when it is required to punch a complete row of holes to act as an indicator of some kind. If such a character occurs on input, it will be ignored by any EMA input instruction (except ϕTAPE when reading from five-track tape (see Section 5.2).

Paper Throw This character can be output to eight-track tape to cause form feed on a Flexowriter with suitable fittings, or it can be used to cause a paper throw to the top of the form of a line printer. In both cases, the appropriate instruction would be

Print Ch (ϕcode (PT))

If a Paper Throw character is read, it is treated as a newline character.

Stop Code This character may be output to eight-track paper tape where, on subsequent input, it will temporarily disengage the paper tape reader for operator intervention. An EMA Read instruction will ignore the character.

As stated above, not all output media can represent every character in the Main and Special sets. If a Print Ch instruction attempts to output a character not in the set of the relevant medium, standard EMA fault action is taken, i.e. a reasonable substitution is made or "?" is output.

Character Arithmetic 2.3.5

Although the programmer does not know the integer values of the EMA codes, he can assume the following relationships between them. The codes for the letters A to Z are 26 consecutive integers, i.e. ϕcode (Z) is equal to ϕcode (A) + 25; the codes for 0 to 9 are 10 consecutive integers; and ϕcode (SP) < ϕcode (A).

The ϕ code function most commonly appears in conditional jump instructions, but as it is an integer function, it can form part of an arithmetic expression in the normal way. Therefore the programmer can make use of a limited form of character arithmetic, based on the relationships stated above, to perform special purpose input/output routines or simple alphabetical sorting.

Examples

(a) Read one character. Jump to 1) if A, to 2) if B... to 26) if Z; otherwise jump to 51).

```
Read Ch (M)
Jump 51, M > φ code (Z)          | Test if
Jump 51, M < φ code (A)          | letter.
K) = M - φcode (A) + 1)          | Set label.
Jump (K)
```

(b) The value of index P is such that $0 \leqslant P \leqslant 99$. Print the value with no sign or space before it and one space after it.

```
Q = φ int pt (P/10)              | 0 ⩽ Q ⩽ 9
Print Ch (φ code (0) + Q)        | Print first digit
Print Ch (φ code (0) + P - 10 Q) | Print second digit
Space
```

Packing 2.3.6

There are only twenty-four indices, so when a large number of characters is to be handled it is convenient to hold some of them in the main variables. Though single characters can be transferred to and from the main (or special) variables by instructions such as "X0 = I" and "I = X0" there is a more efficient technique known as *Packing* which allows up to four characters to be held by one main or special variable.

3146(10.65) -

The programmer can regard a variable as having four character positions, numbered 0 to 3. A character held in J could be packed into position 2 of A3 by the instruction

```
Pack (A3, 2, J)
```

Similarly, the instruction

```
Unpack (C(I+1), 0, M)
```

copies the character in position 0 of C(I+1) to index M.

The main variables can be regarded as a continuous string of character positions. A main variable specified in a Pack or Unpack instruction is taken as an initial reference point and position numbers higher than 3 then specify character positions in succeeding main variables; e.g. positions 0 to 3 of A2 could equally well be thought of as positions 4 to 7 of A1 or 8 to 11 of A0. No position numbers higher than 3 should be used with the special variables. Negative position numbers are never allowed.

Example

The following Chapter reads and packs 800 characters.

```
CHAPTER 0
X → 150
F → 150

M = 0(1)799
   Read Ch (T)
   Pack (X0, M, T)
Repeat
- - -
CLOSE
```

The characters are packed in the 200 main variables X0, ... X150, F0, ... F48.

There is no reason why data other than characters should not be packed and unpacked. In binary terms, the Pack instruction copies the least-significant twelve bits of an index and ignores the others while Unpack first clears an index, then copies twelve bits into its less-significant end.

In decimal terms, Pack always selects from an index an integer in the range 0 to 4095. An index value outside this range is evaluated modulo 4096, i.e. the effect is as though a multiple of 4096 were added or subtracted to bring the packed value within range. Unpack always gives an index a value within the range.

A packed variable should not appear on the right hand side of an arithmetic instruction nor anywhere else in an arithmetic expression. Even a simple arithmetic instruction such as

```
X = AI
```

may lead to unexpected results if AI is a packed variable.

General Constants Tables 2.4.1

Fixed data can be written as part of a program instead of being prepared separately. A Chapter can include a table of general constants in a form such as

```
7) 197.5
   -30
3) 172&-7
   +0.0007
   917753.21475
```

Each line contains one constant with one of the formats described in Section 1.2.1. The first line must be labelled and other lines can be labelled if desired.

A value from the above table can be assigned to a variable by a ϕ Table instruction typified by

```
X = φTable (7, 2)
```

This must be in the same chapter as the table. The "7" refers to the label at the head of the table and the "2" to a particular position within the table. Position numbering starts at 0, so the instruction above assigns the value 172&-7 to X. That entry in the table is itself labelled and the same effect could have been obtained by the instruction

```
X = φTable (3, 0)
```

Any number of tables can be written anywhere within a chapter but the programmer must take care that no attempt is made to treat an entry in a table as an instruction; in particular, an instruction on the line before a table should be either "End" or an unconditional transfer of control (but not a Jumpdown or Down).

Only a special or main variable can appear on the left-hand side of a ϕTable instruction applied to a general constants table. ϕTable is not an EMA function, i.e. it cannot appear in an arithmetic expression. The label reference can be an integer or an index that has been set equal to a label. The position within the table can be specified by an expression.

Example

The following Chapter of a program contains a table of the prices in pounds of 250 products listed in order of their product numbers, which run from 10, 001 to 10, 250. The Chapter reads a product number and the size of an order for that product, finds the unit price from the table and prints the total value of the order

```
CHAPTER 1
- - -
Read (P)                          | Product number
Read (S)                          | Size of order
F = φTable (21, P-10001)          | Unit price
Newline
Print (S*F)5, 3                   | Value
- - -
21)713.5                          | Product 10001
   691.75                         | Product 10002
   - - -
   - - -
   17.3                           | Product 10250
   CLOSE
```

Integer Constants Tables 2.4.2

A table of integers can be written and stored more compactly than above by means of an integer constants table. Each line of such a table begins with the directive "INTEGERS" and should contain an

even number of signed or unsigned integer constants separated from each other by commas. A typical table could be written as

```
13)INTEGERS 1, -2, 3, +5, -7, 11
   INTEGERS 72, 73
23)INTEGERS 12, 1984, 6, -7
```

There is no limit on the number of entries per line, except that of page width, and no limit on the number of lines per table. A label is essential against the first line and optional against others.

Only indices can be assigned values from an integer constants table and the appropriate form of instruction is

```
J = ϕTable (13, 9)
```

which, if it applies to the above table, is equivalent to

```
J = 1984
```

i.e. it selects the value from position 9 of the table headed by label 13. Position numbering always starts at zero but can be relative to any labelled line of a table, e.g. the instruction above could be replaced by

```
J = ϕTable (23, 1)
```

A zero value is automatically added on the end of any line that contains an odd number of entries.

Examples

(a) The eight numbers A1 to A8 are to be printed across a line with formats specified by a table of integer constants.

```
K = 1(1)8
        P = ϕTable (3, K-1)
        Q = ϕTable (3, K+7)
        Print (AK) P, Q
Repeat
- - -
3)INTEGERS 3,4,3,1,5,4,2,6
  INTEGERS 2,2,1,4,2,1,4,1
```

The ϕcode function is really a disguised integer constant and can be written in an integer constants table (but not a general constants table). The ϕoctal function introduced in Section 3.6 can also appear in an integers table.

(b) One of MON, TUE, WED, THU or FRI is to be printed according as S has the value 0, 1, 2, 3 or 4.

```
   CHAPTER 1
10)INTEGERS ϕcode (M), ϕcode (O), ϕcode (N), ϕcode (T), ϕcode (U), ϕcode (E)
   INTEGERS ϕcode (W), ϕcode (E), ϕcode (D), ϕcode (T), ϕcode (H), ϕcode (U)
   INTEGERS ϕcode (F), ϕcode (R), ϕcode (I)
12)- - -
   L = 0(1)2
   P = ϕTable (10, 3S+L)
   Print Ch(P)
   Repeat
   - - -
   CLOSE
```

Labels Table 2.4.3

A table whose entries are labels can be written in a similar fashion to a table of integers, but with a LABELS rather than an INTEGERS directive, e.g.

```
15) LABELS 98) 99) 101) 102)
    LABELS 126) 127) 1) 2) 3) 4)
```


3146(10.65)

The same form of φTable instruction is used, e.g.

```
K = φ Table (15, 0)
```

but it is now equivalent to a label-setting instruction, in this case

```
K) = 98)
```

All the labels in a table must exist in the Chapter (or Routine) containing it.

As with a table of integers, each line should contain an even number of entries or else a zero (and therefore meaningless) value will be added on the end.

Mixed tables of integers and labels are permitted, e.g.

```
28)LABELS 68) 69) 38) 41)
   INTEGERS 4, 16, 3, -9, φ code (*),  φoctal (124)
```

Thus the instruction "M = φTable (28, L)" could mean "M) = 38)" if L were 2 or "M = φcode (*)" if L were 8. General constants tables cannot be mixed with other types in this way.

Example

There is to be a jump to label 20), 31), 27), 19) or 15) depending on whether the value of T' is 33, 34, 35, 36 or 37.

```
    T' = φTable (101, T'-33)
    Jump (T')
101)LABELS 20) 31) 27) 19) 15)
```

Tables and Routines 2.4.4

If the label reference in a φTable instruction written within a Routine is an explicit label number, not an index, then a value will be selected from a table within the Routine and there are then no special difficulties. If the label reference is an index, then the effect of the instruction depends on whether the index has been set equal to a Routine or Chapter label.

The remainder of this subsection assumes a knowledge of the effect of label-setting instructions within a Routine and of related topics (Section 2.1.3).

Suppose several Routines of a Chapter need the same table. It is convenient to write this in the Chapter proper but to refer to it by φTable instructions within each Routine. The techniques for setting an index equal to a Chapter label are then used, e.g. a table labelled 10) in a Chapter could be referred to in a Routine of that Chapter by the sequence

```
J = 10
J) = J)
K = φTable (J, 3)
```

Alternatively, before the Routine was entered J could have been set equal to 10) in the Chapter and the φTable instruction alone would have been sufficient.

The converse case is also possible. Suppose a Routine contained a table labelled 20). If an index were set equal to 20) within the Routine, then the table could be referred to in the Chapter proper (or even in another Routine of the Chapter) by an instruction such as "K = φTable (J, 6)".

A label selected from a labels table always refers to the Routine (or Chapter) in which the table stands, no matter where the relevant φTable instruction was obeyed.

Example

The following is not a useful sequence but merely illustrates some of the points above.

```
   ROUTINE 1
   K = ϕTable (L, 3)
   Return
 1)L) = 21)
   K = ϕTable (L, 3)
   Return
21)INTEGERS 1, 2, 3, 1982
   **

   CHAPTER 0
   L) = 21)
   K = ϕTable (L, 3)
   Print (K) 1, 0                          | Prints "-2891"
   Jumpdown (R1)
   Print (K) 1, 0                          | Prints "-2891"
   Jumpdown (R1/1)
   Print (K) 1, 0                          | Prints "1982"
   K = ϕTable (L, 3)
   Print (K) 1, 0                          | Prints "1982"
   End
   21) INTEGERS 1, 2, 3, -2891
   CLOSE
```

The Select Input Instruction 2.5.1

In many cases, an EMA program will read and process data from only one data document. However, there are often occasions when it is not convenient to interleave several sets of data in a single document: when, for example, two large sets of experimental data produced at separate times are to be compared. In such circumstances, as many different data documents may be prepared as there are input devices to read them up to a limit of 15.

Each data document is assigned a number in the range 1 to 15 (see Section 4), but the numbers need not be consecutive. When the program requires data from, say, document number 2, the instruction

```
Select Input 2
```

is obeyed. Such an instruction causes all succeeding input instructions (Read, Read Ch, etc.) to read from document number 2 until another Select Input instruction is obeyed.

Documents can be selected and re-selected in any order. Each time a document is re-selected, reading continues from wherever it previously left off, exactly as though no other document had meanwhile been read.

If it exists, Input 1 will be automatically selected on program entry. Therefore if a program is accompanied by a single data document, no Select Input instruction need appear in the program provided that the data document is defined as Input 1.

Example

Input 1 contains a large list of numbers. Input 2 is a "steering" document. It has a layout such as the following:

```
DATA TITLE
Processed Survey Data.  5 Jun 1964.          | Title to be output
B10     C980    B700
B2      D12     A71     C50                  | Defines operations to be carried out on
- - -                                        | data of document 1
- - -
C127    A2      E500
DATA TITLE                                   | Status of results.
Above results provisional. Input data unchecked.
```

Each item on the steering document indicates how many numbers are to be read and the operation to be carried out. There are five possible operations:

A Print the sum of the numbers.
B Print the reciprocal of the sum of the numbers.
C Print the product of the numbers.
D Read each number and ignore it.
E Read and print each number; carry out no more operations.

Not more than 1000 numbers will be required for each operation. There will be two data titles on the steering document. The results are to appear with a blank line after the output resulting from each operation, e.g.

```
PROCESSED SURVEY DATA.  5 JUN 1964.
817.345

219.721

37.129

- - -
917.321
214.973

ABOVE RESULTS PROVISIONAL.  INPUT DATA UNCHECKED.
```

A single Chapter program to carry out this job could be:

```
    MAIN 1000                          | No auxiliary variables or dumps needed.

    CHAPTER 0
    A →999
 1) Select Input 2
    Read Data Title                    | Copy first heading

13) Read Ch (J)                        | Jump to here if D read.
    Jump 13, J = φcode (SP)            | Find letter, ignoring spaces and
    Jump 13, J = φcode (NL)            | newlines.

    Read (N)                           | N numbers
    L = N - 1

    Select Input 1
    M = 0(1)L                          | Read A0, . . . . A(N-1) from document
    Read (AM)                          | number 1
    Repeat

    Select Input 2                     | Re-select document 2 in preparation for
                                       | next code letter.

    K) = J - φcode (A) +10)            | Jump to one of 10), 11), 12), 13), 14).
    Jump (K)

10) Jumpdown 2                         | If A, find sum
    Jump 3

11) Jumpdown 2
    X = 1/X                            | If B, find 1/sum
    Jump 3

12) X = 1
    M= 0 (1) L
    X = X * AM                         | If C, find product
    Repeat

 3) Newline
    Print (X) 3,3                      | Print value obtained for A, B or C, followed
    Newline                            | by blank line.
    Jump 13                            | Find next operation.

14) M = 0(1)L                          | If E, print all numbers.
    Newline                            | Print each on newline.
    Print (AM) 3,3
    Repeat
    Newline                            | Blank line

    Read Data Title                    | Copy last heading from document 2 and end.
    End

 2) X = 0
    M= 0(1)L
    X = X + AM                         | Subroutine to find sum of A0, .... A(N-1).
    Repeat
    Return

    CLOSE
```

The Select Output Instruction 2.5.2

Output documents are numbered in a similar way to input documents and are selected in the program by instructions such as

```
Select Output 3
```

which causes all succeeding output instructions (Print, Print Ch, etc.) to print on document number 3 until another Select Output instruction is obeyed.

Documents can be selected and re-selected in any order. Each time a document is selected, printing continues from where it previously left off, exactly as though no printing had meanwhile occurred on other documents.

Although output documents must have numbers within the range 1 to 15, the numbers need not be consecutive. If it exists, Output 1 will be automatically selected on program entry. Therefore, if a program is to provide a single output document, no Select Output instruction need appear in the program provided that the output document is defined as Output 1.

The Relinquish Instruction 2.5.3

On multi-programming machines, peripherals should be associated with a program for the minimum possible time. If the end of an input or output document is reached long before the end of a program, then the associated peripheral should be released from that program by an instruction such as

```
Relinquish Input 2
Relinquish Output 3
```

These instructions relinquish the devices on which input document 2 is being read and output document 3 is being printed, and if a charge is being made for computer time, no further charge will be made for these particular devices. However, at least one output device should be retained at all times.

It is important to note that once a document has been relinquished, it cannot be re-allocated, and any subsequent attempt to use it would be an error.

MAGNETIC TAPE 2.6

General 2.6.1

Magnetic tape is a form of storage medium that can hold large amounts of data relatively cheaply. It can be thought of as being an extension of the computer store, but it is not a fixed part of the computer and therefore has some of the characteristics of an input/output medium. As far as programming is concerned, its chief difference from ordinary computer store is that it is normally used sequentially. Before any data is transferred to or from a magnetic tape the program must ensure that the tape has been wound to the appropriate position. Since tape winding can take an appreciable time, depending on the length of tape to be wound, it is preferable that at the end of one transfer the tape is left in the right position for the next one.

Some of the uses of magnetic tape are:

(a) As a dump. When the normal computer store proves inadequate for all the data required by a program, some of it can be dumped on magnetic tape and recovered later in the program.

(b) As an input/output medium. A program that generates a large amount of data whose sole use is to act as input to another program would generally copy it to magnetic tape rather than to a slower output medium such as paper tape.

(c) As a file. A magnetic tape might hold, say, the results of a series of experiments carried out at regular intervals. There would be one or more programs to update the file by copying to it from input documents new or corrected information. There would also be one or more programs designed to read the file and print out selected parts of it as required.

The EMA Magnetic Tape Instructions 2.6.2

An EMA program can transfer data from a set of main variables to magnetic tape, and conversely. The tape is split into numbered blocks, each capable of holding 512 main variable values. These values may be the usual numeric values or characters packed by the EMA Pack instructions (Section 2.3.6). All transfers are of complete blocks, the first block available for data being block 1.

A program can use several magnetic tapes at a time. Though all tapes will have names it is more convenient within a program to use numbers to distinguish them. The programmer can choose any integers in the range 0 to 7 for these tape numbers. As with the document numbers described in Section 2.5, the choice is arbitrary and the programmer must list the actual names of the tapes he refers to in the description which precedes the program.

A tape is wound to a particular position by an instruction such as

```
Wind Tape (3, 100)
```

This would wind tape 3 backwards or forwards so that it was correctly positioned ready for a transfer in the forward direction to or from block 100. Suppose it were now required to copy the 512 values A0 to A511 to this block. Then the next instruction would be

```
Write (3, A0)
```

After this instruction A0 to A511 would be undisturbed, block 100 would contain copies of their values, and the tape would have moved into position ready for data to be transferred *to* block 101.

Any consecutive block of 512 variables can be involved in a transfer.

It should not be assumed that at the beginning of a program, block 1 of a tape is automatically in position for a transfer. The first instruction of a program will always be a Wind Tape instruction, but thereafter such instructions will only occur as necessary.

Example

To copy A0 to A2047 to blocks 1 to 4 of Tape 2.

```
Wind Tape (2, 1)

J = 0(512) 1536
Write (2, AJ)
Repeat
```


3146(10.65)

Data can be copied back from magnetic tape in a similar way; e.g. block 50 of tape 1 could be copied to X201 up to X712 by the instructions.

```
Wind Tape (1, 50)
Read Fwd (1, X201)
```

As with the Write instruction, the tape would have been moved forwards, so that block 51 would now be in position for a forward read or write operation. However, if a 1974 Magnetic Tape System (96 kch/s) is in use, reading can also take place with the tape moving backwards; the instructions

```
Wind Tape (1, 51)
Read Bkd (1, X201)
```

would transfer block 50 to X201 to X712, i.e. would transfer the same values as before to the same variables as before. The only difference is that the tape would be left in a different position. If the program continued to read backwards block 49 would be read next. Alternatively it could reverse the direction of the tape and read or write block 50, while moving forwards.

Tape should be read backwards whenever this minimizes the amount of tape winding. Suppose a program is to dump data on blocks 1 to 500, in that order, and later recover that data. As long as it is acceptable for the blocks to be recovered in the reverse order, it will be more sensible to read backwards starting at block 500 than to wind the tape backwards to block 1 and read forwards from there.

Note that when a block is written, any higher-numbered blocks already recorded on that tape will be lost. Furthermore, it is an error (except in the special case of block 1) to apply WIND TAPE to a block that has not yet been written, or to a block that has been lost for the reason given above.

There are two other magnetic-tape instructions. A tape is rewound to block 1 if the tape number is specified in an instruction such as

```
Rewind (3)
```

This instruction has the same effect as

```
Wind Tape (3, 1)
```

but is somewhat faster.

If a program no longer requires a particular tape the tape deck on which it is held can be relinquished by an instruction such as

```
Relinquish Deck (4)
```

After this instruction tape 4 could not be referred to again by the program. A Rewind instruction is not essential before a deck is relinquished.

Example

An N x N matrix is to be generated and stored row-by-row on magnetic tape. Values of N and X are to be read from a data document. X is a parameter of the matrix. Given values of X, I and J, Routine 20 will set F equal to the element F_{IJ} of the matrix.

N is in the range $400 \leqslant N \leqslant 500$, so it is convenient for a row to occupy the major part of a block. Each block is to contain:

The row number
The value of N
The value of X
The elements of the row

The unused part of the block is to be filled with zero values .

Chapter 3 is to carry out the operation, then transfer control to label 40 in Chapter 7.

```
ROUTINE 20                          |Puts F equal to one element of the matrix
- - -
* *

CHAPTER 3

A —> 2                              |For block parameters
B —> 508                            |For matrix elements

1) Wind Tape (1, 1)                 |Start at block 1 of tape 1

Read (N)
Read (X)
A1 = N
A2 = X

I  = 1(1)N                          |I = row count
A0 = I                              |A0 = current row
J  = 1(1)N                          |J = column count
Jumpdown (R20)                      |Calculate one row in
B(J-1) = F                          |B0 to B(N-1)
Repeat

K = N (1)508                        |BN to B508 made zero
BK = 0
Repeat

Write (1, A0)                       |Copies A0-A2, B0-B508 to current block

Repeat                              |Next row

Across 40/7

CLOSE
```

SECTION 3

Section 3

SPECIALIZED FEATURES

DIFFERENTIAL EQUATIONS 3.1

EMA has a very useful facility for the step-by-step integration of a set of differential equations. Given a set of n first-order equations that can be written in the form

$$\frac{dy_t}{dx} = f_t\ (x, y_1, y_2, \ldots\ldots y_n) \qquad (t = 1, 2, \ldots\ldots n)$$

and a set of initial values for the variables, then the programmer may easily arrange to compute for each of a series of values of the independent variable x the appropriate values of the dependent variables y_1 to y_n.

The program must use X for the independent variable, Y1, Y2 etc. for the dependent variables, and F1, F2, etc. for $\frac{dy_1}{dx}, \frac{dy_2}{dx}$, etc.

The variables are set to their initial values, the number of equations is placed in the index N and a step length is placed in special variable H. Then to carry out one step of the integration an instruction such as

```
Int Step (3)
```

is obeyed. This increases X by H and calculates Y1, ... YN for this new value of X. The instruction refers to a label, either directly or by means of an index previously set equal to it. This label is against the first instruction of an auxiliary sequence provided by the programmer. The sequence calculates F1, ... FN in terms of X, Y1, ... YN and must be terminated by the instruction

```
End Aux Seq
```

Such a sequence is not entered directly by the programmer but will be used many times during the execution of an Int Step instruction that refers to it. Extra variables G1 to GN and H1 to HN must be available in the relevant chapter and are used by the Int Step instruction.

Example

The three equations

$$\frac{du}{dx} = 5v(\sin x)/w + u(\cos x)/v$$

$$\frac{dv}{dx} = 3w(\sin 4x)/u^2 + v^3 x$$

$$\frac{dw}{dx} = u^3/v^2 + w \cos 3x$$

are to be integrated for values of x running from 4 to 6 in steps of 0.2. The initial values of u, v and w are to be read from a data document.

The first step is to assign EMA variables for the given variables. There is little choice: x, u, v, w become X, Y1, Y2, Y3 and $\frac{du}{dx}, \frac{dv}{dx}, \frac{dw}{dx}$ become F1, F2, F3. The following Chapter sets the initial conditions, prints some text and column headings, prints the initial values of the variables, then integrates the equations step-by-step and prints the values obtained after each step.

```
MAIN 16


CHAPTER 0

F → 3
G → 3                                  | Reserve extra variables
H → 3                                  | for int step instruction.
Y → 3

N = 3                                  | Number of equations
H = 0.2                                | Step length
X = 4                                  | Initial value of X
      I = 1(1)3
      Read (YI)                        | Read initial u, v, w
      Repeat
Printline
Equation Set T/U/V/W/6
Newline                                | Blank line
Printline                              | Column headings
X    U    V    W

Jump 2                                 | Enter main loop, omitting
                                       | integration on first cycle

3) Int step (1)                        | Calculate X + H and new Y1 to YN
2) Newline
   Print (X) 1, 1
   Space 3
      I = 1(1)3
      Print (YI) 0, 6                  | Print u, v, w
      Repeat
   Jump 3, X < 5.9                     | Test for end of range
   End

1) F1 = 5Y2 φsin(X)/Y3       + Y1 φcos(X)/Y2
   F2 = 3Y3 φsin(4X)/(Y1 Y1) + Y2 Y2 Y2 X
   F3 = Y1 Y1 Y1/(Y2 Y2)     + Y3 φcos(3X)
   End Aux Seq

   CLOSE
```

Int Step is not a jump instruction and is followed as usual by the instruction written after it. For n equations at least $n+1$ main variables must be available in each of the sets F, G, H and Y. The Int Step instruction, the first instruction of the auxiliary sequence, and End Aux Seq must all be in the same Chapter or Routine, though chapter changing is allowed within the auxiliary sequence. In fact, the sequence may contain any type of instruction (except Int Step) as long as the values of N, H, X, Y1 to YN, G1 to GN, H1 to HN are left unchanged.

The Int Step instruction uses a Runge-Kutta method with a truncation error of $0(H^5)$. Because this error also depends on the higher derivatives of the function the programmer may adjust the step length between steps if required.

Second or higher-order equations can often be re-written as a set of first-order equations.

Example

Given the equation

$$3\frac{d^2y}{dx} - x^2\frac{dy}{dx} + 2y = 0$$

and the initial conditions

$$x = 0 \quad y = 0 \quad \frac{dy}{dx} = 4.5$$

it is required to calculate y at $x = 1$, using a step-length of 0.05 in x.

The first stage is to re-write the equation as a pair of first-order equations, with y_1 and y_2 replacing y and $\frac{dy}{dx}$; i.e.

$$\frac{dy_1}{dx} = y_2$$

$$\frac{dy_2}{dx} = (x^2y_2 - 2y_1)/3$$

Then the required program could be:

```
MAIN 20

CHAPTER 0
F→ 2
G→ 2
H→ 2
Y→ 2
N = 2
H = 0.05
X = 0
Y1 = 0
Y2 = 4.5

K = 1(1)20
Int Step (7)
Repeat

Print ('At x = 1, y = ')
Print (Y1) 0, 6
End

7) F1 = Y2
   F2 = (XXY2 - 2Y1)/3
   End Aux Seq

   CLOSE
```

MATRICES AND VECTORS 3.2

General 3.2.1

This Section requires a knowledge of auxiliary variables introduced in Section 2.2.3.

There is a group of EMA instructions to perform the more common matrix operations. For these instructions, each matrix must be held as a block of consecutive auxiliary variables with the matrix rows following on from one another. Thus, if the first element of a $p \times q$ matrix were auxiliary variable a the ij-th element would be:

$$a + (i-1)q + (j-1) \qquad (i=1, \ldots p,\ j=1, \ldots q)$$

A vector can be treated as a one-row or one-column matrix. The elements of a matrix can be copied to the auxiliary variables either by a $\phi 7$ instruction transferring from the working store or by a $\phi 10$ instruction transferring from a data document. Similarly, $\phi 6$ can copy a matrix to the working store or $\phi 8$ and $\phi 9$ can print it directly onto an output document.

A typical matrix instruction is:

3001 = ϕ11 (5001, 1001, 64)

This instruction adds the two 64-element matrices starting at auxiliary variables 5001 and 1001 and places the resulting matrix in variables 3001 to 3064. The programmer must ensure that the matrices to be added are the same shape. All four arguments of the above instruction could be expressions; e.g. the sequence

J = 1001

K = 64

J+2000 = ϕ11(J+4000, J, K)

would carry out the same operation as before. An expression on the left of an equals signs appears strange but is quite logical: its value is used only to specify the starting element of a matrix and none of the variables or indices included in it has a change of value.

Auxiliary Storage Directives 3.2.2

The directive

AUXILIARY (I,J)

where I and J are the numbers of the first and last auxiliary variables required, has already been described in Section 2.2

There is an optional addition to the AUXILIARY directive, giving the form

AUXILIARY (I,J,L)

where L is a parameter defining working space to be allocated for certain of the matrix instructions (see Section 3.2.4).

The following conditions should be satisfied:

1	Y = $\phi 25(a,p)$	$L \geqslant p$
2	a = $\phi 26(b,c,p,q,r)$	$L \geqslant p$ and $L \geqslant q$
3	a = $\phi 27(b,c,p,q,r)$	$L \geqslant p$ and $L \geqslant q$
4	a = $\phi 28(b,p,q)$	$L \geqslant p$

$2L + L/4$ words of store will be allocated as working space for these routines. If L is absent, a value of 128 is assumed; if L is not required, it must be set to 1. This directive applies if either core store or E.D.S. files hold auxiliary variables.

Backing Store Directives 3.2.3

Auxiliary variables will normally be held in core store. The programmer may, however, specify an E.D.S. file to hold auxiliary variables (either XMAM or XMAE must be used to compile the program) by writing the following directives before the first Chapter of the program (that is, in a program description, see Section 4.4.4 or Section 4.6.4)

USE AUX (X) = ED

USE AUX (X) = ED ($F(g)$)

CREATE AUX (X) = ED ($F(g)$)

where $F(g)$is the E.D.S. file name, see Section 4.6.2; X is the number of auxiliary variables the user requires to be held in core store at any one time. The compiler rounds X up to the next multiple of 128: the minimum value is 512. If X is omitted, a value of 768 is assumed. The file is allocated as unit ED6.

USE AUX (X) = ED

A scratch file will be opened of sufficient length to hold the auxiliary variables given in the AUXILIARY directive.

USE AUX (X) = ED ($F(g)$)

A named file is opened and it is assumed there is useful information on the file. The length of the file is checked against the length required by the AUXILIARY directive, and the file will be extended if necessary. The minimum and maximum variable numbers are held in the last bucket of the file. It is not possible to reduce the minimum variable number by using this directive.

CREATE AUX (X) = ED ($F(g)$)

A named file is opened but nothing on the file is assumed to be useful. An error will occur if an attempt is made to read from the file before any information has been written to it. The file will be extended if necessary.

The named files are previously set up using the File Allocator Program (see Library Specifications Manual). There are 2 words per auxiliary variable, and therefore the bucket size is 128 words.

128 auxiliary variables are held in a 256 word block on E.D.S. starting with 0 at the beginning of one block, 128 at the next, and so on. If a new block of variables is required, the last block used in store is normally written back to the E.D.S. file, and is replaced by the required block. However, if none of the variables held in the block had been altered since the block was brought into core store, it is not necessary to write it back. At the end of a run, or at an error stop, all the blocks are written back to the file which is then released. If the file is a scratch file it is released without writing blocks back.

There are two factors under the user's control that can effect the efficiency of the matrix routines when the auxiliary variables are held on E.D.S. The first is the position of the matrices. If each matrix is stored so that its first element coincides with the start of a block, that is, the block starts with a variable whose number is a multiple of 128, the matrix will span the least number of blocks, and in general fewer disc transfers will be required to process it.

The second factor is the length X, specified in the USE AUX and CREATE AUX directives. For the most efficient working of certain matrix routines, X should be as follows:

1	$a = \phi16(a,p,q)$	$X \geq p * q$
2	$Y = \phi25(a,p)$	$X \geq p * q$
3	$a = \phi26(b,c,p,q,r)$	$X \geq p * q + r * q$
4	$a = \phi27(b,c,p,q,r)$	$X \geq p * q + r * q$
5	$a = \phi28(b,p,q)$	$X \geq p * p + p * q$

Each product should be rounded up to the next multiple of 128 and then added. If the matrices do not start at the beginning of blocks, 128 should be added for each matrix.

The routines will work with smaller values of X, but less efficiently.

The notation used in the table below is as follows (note that A, B, C, D represent matrices, not special variables).

a,b,c,d,m,n,p,q,r,t	Expressions in integer contexts; the integral parts will be taken.
x	An expression. Its value will be used as a scalar multiplier.
Y	Any main or special variable.
A,B,C	Matrices held in the standard way with starting elements specified by the values of a, b, c respectively.
D	A diagonal matrix held in a special way. The diagonal elements are held as a vector with starting element specified by the value of d. The off-diagonal elements are not stored at all.
I	A unit matrix whose elements are not stored.
A^T	The transpose of matrix A.
A^{-1}	The inverse of matrix A.
det A	The determinant of matrix A.
diag A	The diagonal elements of matrix A.
L	Parameter in the Auxiliary directive defining working store required.
X	Parameter in the Backing Store directive defining maximum number of auxiliary variables held in core store.

Matrix dimensions are specified in one of two ways:

$A(t)$	A matrix A with t elements. Their arrangement by rows and columns is not defined.
$A(p,q)$	A matrix A with p rows and q columns.

Since only its diagonal elements would be stored, a matrix $D(p,p)$ held in special form would occupy only p auxiliary variables as opposed to the p^2 elements of $A(p,p)$.

INSTRUCTION	OPERATION	COMMENTS	
$\phi\,6(a)Y,t$ $\phi\,7(a)Y,t$	Copy A(t) to working store Copy from working store to A(t)	See Section 2.2.3	
$\phi\,8(a,p,q,m,n)$ $\phi\,9(a,p,q,n)$	Print A(p,q) in style 'm,n' Print A(p,q) in style '0,n'	See Note 2 and Section 2.2.3	
$\phi\,10(a,t)$	Read A(t)	See Section 2.2.3	
$a=\phi 11(b,c,t)$ $a=\phi 12(b,c,t)$ $a=\phi 13(b,x,c,t)$ $a=\phi 14(b,x,c,t)$	A(t)=B(t)+C(t) A(t)=B(t)-C(t) A(t)=B(t)+xC(t) A(t)=B(t)-xC(t)		
$a=\phi 15(b,t)$	A(t)=B(t)		
$a=\phi 16(b,p,q)$	A(q,p)=[B(p,q)]^T		
$a=\phi 17(b,p)$ $a=\phi 18(b,p)$ $a=\phi 19(b,x,p)$ $a=\phi 20(b,x,p)$	A(p,p)=B(p,p)+I(p,p) A(p,p)=B(p,p)-I(p,p) A(p,p)=B(p,p)+xI(p,p) A(p,p)=B(p,p)-xI(p,p)		
$a=\phi 21(b,d,p)$ $a=\phi 22(b,d,p)$ $a=\phi 23(b,x,d,p)$ $a=\phi 24(b,x,d,p)$	A(p,p)=B(p,p)+D(p,p) A(p,p)=B(p,p)-D(p,p) A(p,p)=B(p,p)+xD(p,p) A(p,p)=B(p,p)-xD(p,p)	A must not overlap D.	
$Y=\phi 25(a,p)$	Y=det A(p,p)	$p \leq L$, A destroyed. See Note 3	
$a=\phi 26(b,c,p,q,r)$	A(p,q)=B(p,r)C(r,q)	If A overlaps B then $q \leq r$, $p \leq L$, $q \leq L$	
$a=\phi 27(b,c,p,q,r)$	A(p,q)=B(p,r)[C(q,r)]^T	If A overlaps B then $q \leq r$, $p \leq L$, $q \leq L$ If A overlaps C then $p \leq r$,	
$a=\phi 28(b,p,q)$	A(p,q)=B(p,p)$^{-1}$ A(p,q)	$p \leq L$, B destroyed. See Note 4 A $\neq$ B	
$a=\phi 29(x,p)$	A(p,p)=xI(p,p)		"CHLF" versions. See Note 5
$a=\phi 30(b,p)$	D(p,p)=diag B(p,p)	See Note 6	
$a=\phi 29(p)$ $a=\phi 30(p)$	A(p,p)=I(p,p) A(p,p)=null matrix		"Manchester" versions. See Note 5
$d=\phi 31(b,p)$	D(p,p)=diag B(p,p)	See Note 6	

Notes

1 The maximum matrix size is determined by the amount of storage available.

2 $\phi 8$ prints the elements of a matrix in one column. The column is divided into blocks, corresponding to the rows of the matrix, by a blank line after every q numbers. If the blocks are to represent matrix columns, the matrix must be transposed before being printed.

3 Unlike other matrix instructions $\phi 25$ acts as an arithmetic instruction and assigns a value to the variable on the left of the equals sign. For accurate results all the elements should be of approximately the same size. Individual rows or columns should be scaled up or down if necessary. Later a correction can be applied to the value calculated by $\phi 25$.

4 For $\phi 28$ to give accurate results the elements of matrix B should be of approximately the same size. If necessary some elements should be scaled as in Note 3 above.

5 The "CHLF" $\phi 29$ and $\phi 30$ instructions and the "Manchester" $\phi 30$ and $\phi 31$ instructions provide similar facilities. They were introduced into Mercury Autocode at different installations and either or both of the sets can be used in EMA.

6 The "CHLF" $\phi 30$ and the "Manchester" $\phi 31$ extract the diagonal elements of a square matrix B and store them as a vector in auxiliary variables $d, d+1, \ldots d+p-1$. This vector is then suitable for use in the instructions $\phi 21$ to $\phi 24$.

7 If A overlaps C in $\phi 27$, then if C is larger than A the remnant of C is destroyed.

8 Overlapping is permitted only if first element addresses correspond. This does not apply to $\phi 15$.

9 A, B, C must not start at the same variable in $\phi 26, \phi 27$.

Examples

(a) Form the sum of five 10×10 matrices held consecutively as the auxiliary variables 1001 to 1500.

```
Z = 201                    | Put 201 .... 300
Z = φ30(10)                | equal to zero.
    K=1001(100)1401)
    Z=φ11(Z,K,100)         | Add each matrix in
    Repeat                 | turn.
```

Special variable Z was introduced merely to make the matrix instructions look more meaningful. It could be replaced by the integer constant 201 with no change of meaning.

(b) Multiply the 40×30 matrix defined by A by the 30-element column vector defined by V.

```
F = φ26(A,V,40,1,30)
```

(c) Read the matrices F(50,50) and G(50,50) from a data document. Print out by columns the inverse of the matrix F+3G.

```
F = 1
G = 2501
φ10(F,5000)                | Read F and G to 1, 2, ...5000
G = φ13(F,3,G,2500)        | Read F and G to 1, 2, ...50000
F = φ29(50)                | F = 1
F = φ28(G,50,50)           | F = G inverse
                           | F destroyed
G = 016 (F,50,50)          | Transpose result
φ8(G,50,50,2,5)            | Print by columns
```

The transposition was required so that the printing would be by columns, not rows.

3146(10.69)

PSEUDO-RANDOM NUMBERS 3.3

General 3.3.1

Programs written for such purposes as simulating the operating of an industrial plant, "dealing" a hand of cards, selecting numbers for bingo etc., often require a sequence of pseudo-random numbers to be available. There are two systems for generating such sequences in EMA. With neither system are the numbers truly random - a particular program with fixed data always produces the same sequence - but they both produce a string of numbers with a particular type of distribution.

The more general system uses various forms of the instruction

X = ϕRandom (Y, N)

N, or the integral part of an expression written in place of N, must be 0, 1 or 2. The significance of these values is described in the next three sections.

Starting a Sequence 3.3.2

Members of a sequence of pseudo-random numbers are generated one at a time. Each one is calculated from the previous one. The first term must be created by a ϕRandom instruction with N equal to zero, e.g.

X = ϕRandom (13, 0)

The first argument can be any expression whose value is an odd positive integer. Each such value creates a different first term and therefore a different sequence.

Once the initial value has been set up the sequence can be continued to give either a rectangular or normal distribution pattern as described below.

A Rectangular Distribution 3.3.3

Once a variable has been given an initial value by a ϕRandom instruction with N equal to 0, a series of such instructions with N equal to 1 can generate a sequence of numbers rectangularly distributed in the open interval (0, 1). Each number must be generated from the previous member of the sequence, so the instructions will generally take one of the forms typified by

X = ϕRandom (X, 1)

and

XI = ϕRandom (X(I-1), 1)

The first instruction replaces a number by its successor, the second instruction generates a sequence of numbers in a set of main variables (assuming that I is stepped on by 1 for each time the instruction is obeyed).

The instruction

Y = ϕRandom (X, 1)

would not be rejected but would always give Y the same value unless some other instruction changed X. But if X were changed by anything other than a ϕRandom instruction the sequence would be statistically meaningless anyway. For a similar reason X could not be replaced by an expression.

This form of the ϕRandom instruction generates pseudo-random numbers by the residue class method.

Although the numbers are not truly random and any given program will always produce the same numbers on one particular computer, it is important to note that Atlas, Orion and a 1900 series computer would all produce different sets of numbers from the same program.

Examples

(a) To give A1 to A100 values rectangularly distributed in the range (0, 1).

```
A0 = ϕRandom (7, 0)                    |7 or any odd integer > 0.

L = 1(1)100
AL = ϕRandom (A(L-1), 1)
Repeat
```

(b) To test Routine 10 by entering it 500 times with index P having values rectangularly distributed in the range 0 to 10000.

```
X = ϕRandom (5, 0)
I = 1(1)500
X = ϕRandom (X, 1)
P = 10000 X
Jumpdown (R10)
- - -
- - -
Repeat
```

Normal Distribution 3.3.4

Having set up an initial value by the method described in Section 3.3.2, it is possible to obtain a sequence of numbers which have an approximately normal distribution in the open interval (-6, 6) with a zero mean and unit standard deviation. First, an initial value is set up in, say, X; then each number is obtained by a ϕRandom instruction with N equal to 2; e.g.

```
Y = ϕRandom (X, 2)
```

This instruction places the required number in Y and also changes the value of X. It would be a fault for X to be anything but a variable. X should not be changed by the programmer or the sequence will be upset. It follows that, unlike ϕRandom with N equal to 1, the variable on the right of the equals sign should not contain a useful member of the sequence. This form of the instruction obtains a number by adding twelve numbers generated by a method equivalent to ϕRandom with N equal to 1.

Example

To give A1 to A100 values normally distributed in the range (-6, 6).

```
X = ϕRandom (3, 0)                     |Set up initial value
L = 1(1)100
AL = ϕRandom (X, 2)
Repeat
```

Contrast the above sequence with that in example (a) of Section 3.3.3.

Restarting a Sequence 3.3.5

Suppose that a sequence is to be suspended in one program and continued from the current value in a different program. Because of rounding errors it is not sufficient for the first program to print the value and the second one to read it. Instead, by the use of the Unpack instruction (Section 2.3.6.) the current value can be broken down into four integers that are effectively a coded form of the number and have in themselves no significance to the programmer. These integers can then be printed and read without introducing rounding errors. Once read they can be compounded by Pack instructions back into the original number.

Note that in the example in Section 3.3.4, X would be encoded, not AL.

Example

The following sequence prints a coded form of the number in $\pi 0$:

```
K = 0(1)3
Unpack (π0, K, J)
Print (J) 4, 0
Repeat
```

The sequence below could then appear in another program and would re-assemble the number:

```
K = 0(1)3
Read (J)
Pack (π0, K, J)
Repeat
```

The last set of instructions above would give a number suitable for continuing a pseudo-random sequence only if the integers were produced by unpacking a pseudo-random number on the same type of computer: it is not statistically meaningful to use just any integers or to use integers produced on a different type of computer.

Rectangular Distribution: a Different Method 3.3.6

There is an alternative method of producing a sequence of numbers rectangularly distributed in the open interval (0, 1). Each number is generated by an instruction typified by

```
Random No (X)
```

which places a pseudo-random value in X. Any main or special variable could be used in place of X. An initial value is created automatically and, because the same value is used for all programs, there is only one possible sequence for each type of computer. Although the instruction is roughly equivalent to

```
X = ϕRandom (X, 1)
```

there is an important difference in that a change in the value of X cannot change the value obtained by the next Random No instruction. The instructions of example (a) in Section 3.3.3 could be replaced by

```
L = 1(1)100
Random No (AL)
Repeat
```

The actual numbers obtained would differ but the distribution would be the same.

Suppose a sequence generated by the Random No instruction is to be broken off and continued from the same value in another program. The method described in Section 3.3.5 does not work because the value given by, say,

```
Random No (π0)
```

does not depend on the previous value of $\pi 0$. However, there are two instructions which have a similar effect. The instruction

```
From Random
```

places in the indices Q', R', S', T' four integers which are a coded form of the current random number. These numbers have in themselves no significance to the programmer, but they can be printed out and later read by a different program. They should be read back to the indices Q', R', S', T'. The instruction

```
To Random
```

can then be used to assemble from these indices a number which is used as the current random number.

It is meaningful to use To Random only when the numbers in Q', R', S', T' were originally obtained by a From Random instruction obeyed on the same type of computer.

COMPLEX ARITHMETIC 3.4

A complex quantity is written as a pair of real quantities such as (F, G) meaning F + Gi or (X-3, 5) meaning X-3 + 5i. Complex pairs can be combined into expressions using the operators +, -, * and / in exactly the same way as can real quantities. These expressions are then used in arithmetic instructions to assign values to complex variables.

Examples

(U, V) = (U, V) + (S, S') / [(U, U')(π1, π 2)]

(A1, A2) = (A0, A1) [A3+I , 4Y] + 1 / [X + Yϕsin (X), X - Yϕ sin (X)]

(FI, GI) = 2π(X, X') + (U, V) /3 - 2KX ϕtan (Z)

Any type of bracket is allowed round a complex pair. A pair used as an operand in an expression can have any real expressions as its real and imaginary parts. On the left of an equals sign a pair can be any two variables or any two indices, though not an index and a variable. Real quantities are allowed in a complex expression, e.g. 2 , 3 and 2KX ϕtan(Z) above. Complex pairs are not permanently associated, so after the above instructions it would be in order to refer to, say, (U, GI), (A2, A1) or (1, FI).

When a complex instruction with a pair of indices on its left-hand side is obeyed, the expression is calculated as usual but, if necessary, its real and imaginary parts are separately rounded off to the nearest integers before their values are assigned. Thus,

(P, Q) = (3.7, 4.1) - (2.4, -91.7)

is equivalent to

(P, Q) = (1, 96)

In addition to the real functions, which may be freely used in complex expressions, there are three complex functions: ϕexp, ϕsq rt and ϕlog. The EMA versions of the expressions

$\sqrt{u+v\mathbf{i}}$

$\log\ [(a + b\mathbf{i}) / (a \sin x + b\mathbf{i} \cos x)]$

$3\mathbf{i} + \sqrt{1+(f+g+h\mathbf{i})^{(3+2\mathbf{i})}}$

could be written using these functions as

ϕsq rt(U, V)

ϕ log [(A, BC) / (Aϕ sin (X), B ϕ cos (X))]

(0, 3) +ϕsq rt [1 + ϕexp((3, 2) ϕlog (F+G, H))]

Note that

(U, V) = ϕsq rt(X, Y)

gives U ⩾ 0, and

(U, V) = ϕlog (X, Y)

gives -π<V<π. An argument consisting of a single complex pair should have only one set of brackets.

These three functions should not be confused with the equivalent real functions: ϕsq rt(-4, 0) has the value (0, 2) but ϕsq rt(-4) has no value and would cause the program to be terminated.

Two real functions that are often useful in complex arithmetic are ϕradius (X, Y) and ϕarctan(X, Y). The first gives $\sqrt{X^2+Y^2}$, i.e. the modulus of X+Yi, while the second gives the complex argument of X+Yi. Because these are real functions, X and Y must be two separate real expressions and cannot be replaced by a single complex expression.

An instruction such as

(U, V) = 3π + ϕcos(X)

would be correctly interpreted as meaning

U = 3π + ϕcos (X)
V = 0

However, an instruction such as

X = (A, B) + Y/3

with a real variable on the left and a complex expression on the right would be rejected.

A complex pair in round brackets must never have a superfluous pair of round brackets, e.g. not ((X, Y)), though [(X, Y)] would be acceptable. This restriction is to prevent confusion with double -precision pairs (Section 3.5).

There are no special Read or Print instructions for complex numbers.

Example

To read the complex numbers α_1, α_2, ... α_{100} and the real number a and to tabulate the function

$$f(T) = \frac{(e^{i\theta T} + i\theta + T)}{\left| e^{iTa} + e^{-iT} \right|}$$

against the integer values of T from 1 to 50, where

$$\theta = \left|\alpha_m\right| \sum_{k=1}^{100} a_k^2$$

and $\left|\alpha_m\right|$ is the largest $\left|\alpha_k\right|$.

```
CHAPTER 0
X→100
Y→100
Z→100

Read (A)                                    | a
(U, V) = (0, 0)                             | Theta = 0
      K = 1(1)100
      Read (XK)                             | Re αk
      Read (YK)                             | Im αk
      (U, V) = (U, V)+(XK, YK)(XK, YK)      | Theta + αk*αk
      ZK = ϕradius (XK, YK)                 |  |αk|
      Repeat
(U, V) = Z ϕmax (Z0, 1, 100) * (U, V)       | Completed Theta

Print ('a=')
Print (A) 1, 3
Space 2
Print ('Re Theta =')
Print (U) 1, 4
Print ('Im Theta =')
Print (V) 1, 4
Printline
  T     Re f(T)     Im f(T)
```

```
T = 1(1)50
(F, G) = φexp (0, TA)+φexp (0, -T)
(F, G) = [(T-V, U) + φexp((0, T)(U, V))] / φradius (F, G)
Newline
Print (T) 2, 0
Print (F) 0, 6                                          |Re f(T)
Print (G) 0, 6                                          |Im f(T)
Repeat

End
CLOSE
```

DOUBLE - PRECISION ARITHMETIC 3.5

Values of variables and general constants are normally held to a precision of about 11 significant decimal digits. Arithmetic carried out on these quantities is called *single-precision* (*single-length*) arithmetic. If this introduces unacceptable rounding errors or if results are required to more than 11 places the programmer must resort to *double-precision* (*double-length*) arithmetic. With this, it is possible to manipulate numbers that are precise to the equivalent of about 20 significant decimal digits. Since these numbers cannot be stored in the machine directly, they are held as a pair of single-length numbers, one representing a single-length approximation to the 'true' value and the other representing a correction to make this approximation hold to about 20 significant digits. In a written program double-length pairs would be written in forms such as

((X, Y)) ((A0, B0)) ((19734578123, 0.12345)) ((8, 0))

Only round double brackets are allowed. The two components of a pair should be constants, variables or indices but not expressions or functions. As will be seen below the values of the two components should bear a sensible relationship to each other.

An instruction such as

Read ((X, Y))

reads one number from a data document and assigns single-length values to X and Y such that ((X, Y)) is the double-length value of the number read. The number would have been written in any of the usual forms. Any number of digits are allowed but only the most-significant 20 or 21 contribute to the stored values.

Double-precision arithmetic instructions have a similar form to single-precision instructions, e.g. the instruction

((X, Y)) = [((A, B))+((C, C'))]((AI, B(I-1)))/((U, V))

assigns the double-precision value of the expression to ((X, Y)), which may then be referred to in double-precision expressions. X may be referred to separately as a single-precision approximation to ((X, Y)) but a reference to Y by itself would be meaningless.

Double-precision output is by instructions such as

Print [((A0, B0))+((A1, B1))] 17, 2

The value to be printed can be specified by a double-length expression in brackets, as above, or by one double-length pair which may or may not have an extra pair of brackets, e.g.

Print ((X, Y)) 8, 12
Print [((A, B))] 7, 12

The style is specified exactly as for single-length numbers. The only difference in effect is that a single space is inserted after every five digits counting each way from the decimal point; e.g. the last instruction above might print a number such as

19 87654.13245 98736 54

Read instructions always ignore single spaces in a number, so such numbers can be re-input if necessary.

An index or a single-length variable or constant whose internal (binary) value is held exactly can be paired with zero to give a double-length pair. For instance the instruction

((X, Y))=((A, 0)) ((2, 0))/((5, 0))

assigns a double-length value of 0.4 A to ((X, Y)). As long as A is exact this value is accurate to double-length. If A is not exact the value assigned to ((X, Y)) is no more accurate than is that given by

X=0.4A
Y=0

Similarly,

((X, Y))=((A, 0)) ((0.4, 0))

does not assign an accurate double-length value to ((X, Y)) because 0.4 is not held exactly to single -length. Programmers unfamiliar with the binary representation of numbers should assume that the only numbers capable of being held exactly are integers in the range $-10^{11} \leqslant x \leqslant 10^{11}$.

3146(10.65)

A double-length version of a constant not in the above range can be included in a program as a combination of constants. The method of achieving this is described here. The first 11 significant digits should be written as an integer in the left-hand position of a double-length pair. The remaining digits should be written in the right-hand position as a fraction in the range $0.1 \leq |x| < 1$. A number with fewer than 11 digits should be written as an integer and paired with zero. The true value of the constant is obtained by multiplying or dividing the pair by the appropriate power of 10. Some examples will clarify this process.

Constant	Written Version
3	((3, 0))
.3	((3, 0))/((10, 0))
1293456781134.5678	((12934567811, .345678))((100, 0))
.123456789113456	((12345678911, .3456))/((1&11, 0))
.000123456789113456	((12345678911, .3456))/[((1&11, 0))((1&2, 0))]

As the last example shows, for very large or very small numbers the power of 10 itself may have to be written as a product to bring it within range.

It is not always necessary to couple a zero value with a single-length quantity in a double-length expression because some quantities are automatically coupled with zero. Care is needed. The instruction

((A, B))= ((2, 0))/((3, 0)) + ((7, 0))

could be abbreviated to

((A, B))=2/((3, 0)) + 7

without loss of accuracy but not to

((A, B))=2/3 + ((7, 0))

In the first case 2 is extended to ((2, 0)), the division takes place, 7 is extended to ((7, 0)) and the addition takes place. In the second case the single-length result of the division is meaninglessly paired with zero before being added to ((7, 0)). Any doubts as to the sequence of operations should be resolved by stating explicitly what is wanted. Functions should not occur in double-length pairs. One way to re-write the instruction

((X, Y))=ϕmax(A0, M, N)/I - ((F, G))

to make quite certain that the division is accurate to double-length is as a sequence such as

U=ϕmax(A0, M, N)
((X, Y))=((U, 0))/((I, 0)) - ((F, G))

The formal definition of a double-precision expression is similar to that of a single-precision expression given in Section 1.2.6. The only difference is the inclusion of double-precision pairs in the list of elements allowed. Any real expression that includes a double-precision pair is a double-precision expression and yields a double-precision value. There are only two double-precision contexts: the right-hand side of a double-precision arithmetic instruction and the main argument of the Print instruction. A double-precision expression is, however, allowed in a single-precision or integer context and the calculated value will be cut down as necessary. Thus, the instruction

J=((X, Y)) + ((F, G))

is equivalent to

((A, B))=((X, Y)) + ((F, G))
J=A

and the instruction

Jump 3, ((X, Y)) + ((F, G)) > ((U, V))/((3, 0))

is equivalent to

((A, B))=((X, Y)) + ((F, G))
((A', B'))=((U, V))/((3, 0))
Jump 3, A>A'

A more effective comparison than the above could be obtained by comparing the difference between the two expressions to zero, e.g.,

Jump 3, ((X, Y)) + ((F, G)) -((U, V))/((3, 0)) > 0

The components of complex pairs are in single-length contexts, so the instruction

```
[X, Y] = [((F, G)) + ((U, V)), D] + [((F', G'))/((U', V')), E]
```

is equivalent to

```
((A,  B ))=((F, G)) + ((U, V))
((A', B'))=((F', G'))/((U', V'))
[X, Y]  =[A, D] + [A', E]
```

Similarly the arguments of EMA functions are in integer or single-length contexts and the instruction

```
((X, Y))= φsin( ((F, G)) + ((U, V)) ) + [ φsqrt( ((U', V')) ) + ((F, G)) ]
```

is equivalent to

```
((A, B))=((F, G)) + ((U, V))
A=φsin(A)
A'=φsqrt(U')
((X, Y))=((A, 0)) + [ ((A', 0)) + ((F, G)) ]
```

Example

To read N numbers and print out the double-length root-mean-square of the reciprocals of these numbers. It is assumed that Routine 1 puts ((U, V)) equal to the double-length square root of ((X, Y)).

```
((X, Y)) = ((0, 0))
I=1(1)N
Read((F, G))
((X, Y))=((X, Y)) + 1/[ ((F, G))*((F, G)) ]
Repeat
((X, Y))=((X, Y))/N

Jumpdown (R1)
Print((U, V))0, 21
```

Binary Representation 3.6.1

Certain types of problem involve large amounts of data with only two possible values; for instance, a program might analyse the answers to a questionnaire, each answer being either "yes" or "no". Though it is possible to represent each answer by, say, the value of one main variable (perhaps with "1" for "yes" and "-1" for "no") much less computer store is needed if direct use is made of the binary representation in which numbers are held by the computer.

Each number is held internally as one or more binary *words*, i.e. as a pattern of *binary digits* or *bits*, each bit being either 0 or 1. The only words considered here are those that represent the values of indices or of expressions in integer contexts, and in 1900 EMA these consist of 24 bits. The integer 11, for example, would be stored as

000000000000000000001011

which would be written in this section of the manual in the abbreviated form

0 01011

This section does not demand a knowledge of binary arithmetic nor even a knowledge of how numbers are represented in binary form: it is concerned solely with bit patterns. Note that the left-hand and right-hand ends of a pattern are called the more-significant and less-significant ends respectively.

Logical Shifts 3.6.2

The shift functions move a bit pattern a given number of bit positions to the right or left, while keeping the same word length. Suppose the pattern of J were

101 1110011

Then the instruction

L = ϕshift right (J, 3)

would assign to L the same pattern as J, only moved to the right three places, i.e.

000101 1110

while the instruction

M = ϕshift left (J, 2)

would assign to M the pattern

1 111001100

Any bit that is shifted off the end of a word is lost and replaced by a zero at the other end. It follows that a shift of 24 or more places always gives a result consisting of 0 bits in every position (this corresponds to decimal zero). A shift of a negative number of places is allowed.

The arguments of the shift functions can be any expressions and are in integer contexts. The instruction

K = ϕshift right (ϕshift left (K, 5), 5)

has the net effect of clearing the top five bit positions of K, i.e. it makes than all zero. The instruction

L = ϕshift right (B1+I+J, 10)

evaluates the expression B1+I+J; takes the integral part of the result; moves the bit pattern representing this integer 10 places to the right; then assigns this pattern to the index L. A variable could have replaced L, in which case the integer value would have been automatically converted to the appropriate form before assignment. Thus, logical operations may be carried out in conjunction with variables, but only at the expense of some extra time (resulting from the conversions) and extra storage space (resulting from the use of 48-bit variables where 24-bit indices would be adequate).

Counting 1-bits 3.6.3

The function ϕside add is used to count the number of 1-bits in a pattern. If index I were

0 0111001011

the instructions

L = ϕside add (I)
M = ϕside add (ϕ shift right (I, 3))

would give L and M the values 6 and 4 respectively.

Logical Combination 3.6.4

Each of the functions $\phi$and, ϕor and ϕnoneq produces a bit pattern by combining two specified patterns one bit at a time. The rules obeyed by the three functions are as follows: for each bit position, the bits in this position in the original two patterns are compared, then

$\phi$and yields	1 if they are both 1
or	0 if either or both are 0
ϕor yields	1 if either or both are 1
or	0 if both are 0
ϕnoneq yields	1 if the two are different
or	0 if they are the same.

The result of applying these functions to possible values of the indices I and J is illustrated:

I:	0....1001
J:	0....1010
ϕ and (I, J):	0....1000
ϕ or (I, J):	0....1011
ϕ noneq (I, J):	0....0011

The three functions allow various modified copy or transfer operations to occur. In the following examples P is assumed to have the pattern

0....01111

The And operation can be used to copy particular parts of a pattern, e.g. the least-significant four bits of K can be copied to L by

L = $\phi$and (K, P)

The Or operation allows 1-bits to be superimposed on the pattern being copied, e.g. the instruction

L = ϕor (K, P)

copies K to L with the least-significant four bits forced into being 1's. The third operation is called non-equivalence and can be used to copy a pattern with certain bits reversed, e.g. the instruction

L = ϕnoneq (K, P)

copies K to L with the least-significant four bits reversed.

Logical Constants 3.6.5

For the various logical functions to be useful, it is necessary to be able to introduce into a program logical constants consisting of a particular bit pattern. There is no objection to writing the decimal equivalent of a particular pattern, e.g.

00....0111101

can always be specified by writing 61; however, it is usually more convenient to be able to write a constant in a form nearer to the required pattern.

The appropriate method is to use the function ϕoctal to introduce a number to the base 8. The pattern above could be obtained by writing

ϕoctal (75)

where 75 is treated as an octal, not a decimal number.

This function can be used by programmers unfamiliar with binary and octal representation by means of the following rule of thumb: the required pattern should be split into groups of three bits starting from the right, then for each group a single digit in the range 0 to 7 should be written. It will be selected according to the following table:

000	0	100	4
001	1	101	5
010	2	110	6
011	3	111	7

For example, J could be given the pattern

1 101001

by writing

J = ϕoctal (77777751)

The pattern is right-justified, i.e. if less than 8 octal digits are specified, then the appropriate number of zeros are added on the left, and if more than 8 octal digits are written, all but the 8 right-most digits are ignored. Negative octal numbers are permitted, so the instruction above could be written more concisely as

J = ϕoctal (-27)

and would still give J the pattern 1....101001.

The function is unnecessary for patterns consisting of 0-bits in all except the least-significant three positions, since the octal and decimal representations are then the same; for instance, "J = ϕoctal(1)" and "J = 1" both assign the pattern 0....001 to J. The simplest way of specifying a pattern consisting of 1-bits in every position is to write the decimal number -1 rather than to use a ϕoctal function.

Within expressions, octal constants are completely interchangeable with decimal integers, so the instruction

Print (X) ϕoctal(11)+1, ϕoctal(13)-9

is a valid (but useless) alternative to

Print (X) 10, 2

Octal constants are also allowed in integers tables (see Section 2.4.2).

Examples 3.6.6

(a) A large number of people have been questioned as to which newspapers they read. Each person's answers have been printed on a line of a data document in a form such as

Y N Y N N . . . .

meaning that the person does or does not read the *Guardian, Times, Daily Telegraph, Daily Mirror, Daily Express* etc., in that order. Twelve newspapers have been considered.

As a first step to processing the data, each line is to be compressed into a pattern of 12 bits and packed into a character position of the main variables. "Y" and "N" are to become 1 and 0 respectively; thus a typical line could be held as the 12 bits

101000000000

For simplicity it is assumed that there are exactly 500 lines to be packed away and no checks are made for wrongly prepared data.

```
MAIN 125

CHAPTER 0
G → 124

P = 0(1)499                          |Number of lines
R = 0                                |Each line to go in R
    Q = 1(1)12                       |12 answers per line
  1)Read Ch(S)                       |Read 1 character
      Jump 1, S = φcode (SP)         |Repeat until all spaces read
    R = φshift left (R, 1)           |Make least-significant bit zero
    Jump 2, S = φcode (N)            |Jump if "N"
    R = φor(R, 1)                    |Make least-significant bit 1 if "Y"
  2)Repeat

Pack (G0, P, R)                      |Pack one line
3)Read Ch (S)                        |Read to end
      Jump 3, S ≠ φcode(NL)          |of line

Repeat                               |Next line
- - -
- - -
CLOSE
```

(b) Given data packed as above, the following section of program computes and prints the average number of newspapers read by each person (to the nearest integer) and the number of people who read one or more of the "quality" papers, i.e. *Guardian, Times* or *Daily Telegraph* .

```
M = 0
N = 0
    P = 0(1)499
    Unpack (G0, P, R)                          |Data for 1 person in R
    M = M + φside add (R)                      |Total number of newspapers read in M
    Jump 1, φand (R, φoctal(7000)) = 0         |Does he read quality papers?
    N = N + 1                                  |Yes
    1)Repeat
Newline
Print (M/500) 2, 0                             |Average number of papers read
Print (N) 3, 0                                 |Number of people who read quality papers.
```

(c) On alternate cycles of a loop, it is required to jump to 10) with I equal to 3 or to 11) with I equal to 0.

```
I = 0
  T = 0(1)1000
  - - -
  I = φnoneq (I, 3)                            |I alternately
  Jump 10, I = 3                               |3 and 0
  Jump 11
  - - -
  Repeat
```

Section 4

PREPARING AND DEVELOPING A PROGRAM

ABOUT THIS SECTION 4.1

Most of the information given in the earlier sections of the manual applies to EMA programming for any computer. This section is largely concerned with preparing and developing an EMA program specifically for a 1900 series computer.

Section 4.2 is a brief general description of 1900 series computers. Section 4.3. covers the various types of input/output media available in 1900 EMA. Section 4.4. describes the magnetic tape compiler, XMAM, Section 4.5 describes the paper tape compiler, XMAP, and Section 4.6 describes the E.D.S. compiler, XMAE. Sections 4.7, 4.8, and 4.9 deal with testing and developing a program. Section 4.10 deals with writing overlay programs to be compiled by XMAM or XMAE. Section 4.11 deals with programs written in more than one language, and, finally, Section 4.12 gives examples of listings.

1900 SERIES COMPUTERS 4.2

Computers in the 1900 series differ greatly in size, power, and the number and type of peripheral units. The smaller machines run only one program at a time and are controlled by a human operator. The larger machines are multi-programming; that is, they can run several programs simultaneously, and may be controlled by one of the various operating systems available for different configurations.

Whether multi-programming or not, each machine has supplied with it a program called *Executive*. This program may be regarded as a permanent part of the computer, and works in conjunction with the operating system or human operator to run the machine effectively. It controls the running of other programs and the multi-programming activities of the larger machines; it also controls the peripherals and the central processor. It communicates with, and executes orders given by, the operator. This communication is achieved through the *console typewriter*, which gives a permanent typed record of all communications in the sequence in which they appear. Not only can the operator type instructions to Executive to load, activate, and delete programs for example, but Executive can also use the typewriter to inform the operator of unusual situations within the machine.

TYPES OF INPUT/OUTPUT MEDIA 4.3

Introduction 4.3.1

A *program document* may be prepared on eight-track paper tape, five-track paper tape, punched cards or occasionally a combination of any two or three of these media.

The *semi-compiled program* resulting from the compilation may be output to eight-track paper tape, magnetic tape, or an exchangeable disc store file. The choice is determined by the version of the compiler in use, which in turn is determined largely by the configuration of the computer in use. EMA compilers are described in Sections 4.4, 4.5 and 4.6.

A *data document* is prepared on eight-track paper tape, five-track paper tape or punched cards, and an *output document* can be obtained on eight-track paper tape, punched cards, magnetic tape, or, more usually, a line printer. An output document can also be recorded on five-track paper tape but this is not recommended.

Details of the various forms of input/output media are given below.

Punched Cards 4.3.2

Program, data and results may be punched on standard 80-column cards. The standard 1900 series card code (given in Appendix 1 of the PLAN Reference manual) is used. However, as this 64-character code does not include some of the EMA characters, the following equivalents (some of which are two standard characters) are used in a program document:

→	- >
ϕ	%
π	£
≥	> =
≤	< =
≠	#
¦	!

In *program,* each card may be regarded as a line containing 72 characters. The last 8 columns of a card, or any columns after a Vertical Bar, are ignored by the compiler.

In *data and results,* however, each card may be considered as a line containing 81 characters: one for each of the eighty columns plus a newline character representing the end of the card. If sequence numbers or comment are required, they must be preceded by a Vertical Bar so that the program ignores them; in such cases, the Vertical Bar character is treated as a newline.

If there is an attempt to output a character that is not available on punched cards – this could occur if paper tape were the input medium – then a reasonable substitution (such as two spaces for a Tab character) is made, or the character "?" is output as an error indication.

At the beginning of each pack there must be at least two blank cards. At the end of a complete program or data document, there must be a card blank except for ***Z or **** in columns 1 to 4, and at least two completely blank cards.

When EMA programs are run under a GEORGE operating system each document must have a document name, and must be terminated by ****.

Eight-track Paper Tape 4.3.3

Program, data and results may be punched into eight-track paper tape using the standard code given in Appendix 1 of the PLAN Reference manual. Apart from the graphic characters, this code contains a variety of additional characters, most of which will not normally be required in EMA. Certain EMA characters are not available in this medium, so the substitutions given for cards are also used on eight-track paper tape programs, that is:

→	- >
ϕ	%
π	£
≥	> =
≤	< =
≠	#
¦	!

The eight-track tape characters that can be handled by means of EMA output instructions are given in Section 2.3.4, but note that some of these can be output only by the Print Ch instruction.

The same characters can be handled by input instructions. In addition, other characters will be accepted but will be converted to one or more other characters in accordance with the list on the next page.

Character	EMA Conversion
FE_3 (Line Feed)	newline
FE_4 (Form Feed)	
FE_5 (Carriage Return)	
a to z	A to Z
HT (Horizontal Tab)	In *data*, HT is converted to two spaces – this makes HT a useful number terminator. In *program*, HT is converted to one or more spaces depending upon the PLAN Tab settings.†
— (Underline)	@

The Stop characters, TC_4 and DC_4, and the Erase character will be accepted but ignored. Blank tape is also ignored.

In *program*, all eight-track tape characters absent from Section 2.3.4 and the list above will be converted to "↑" which acts as an error indication. In *data*, however, such characters are acceptable, but only when they are to be input by means of an instruction such as Read Data Title and then passed directly to eight-track paper-tape output without any processing.

In *program*, no line may contain more than 128 characters in the 1900 internal code. A few EMA characters correspond to two internal characters, but about 110 EMA characters is a reasonable maximum line length. In *data* and *results*, there is no limit to the length of line although there may be some restriction imposed by the tape editing equipment in use.

If a document is recorded on a single section of tape, this tape should start with at least one foot of runout (blank characters) which will be ignored. The tape must end with

newline

Z (or *)

newline

newline

TC_4

followed by at least six inches of runout.

If more than one section of tape is used, intermediate tapes must end as above but with ***Z replaced by ***T. The effect will be to disengage the tape reader ready for the operator to load the next length of tape.

† These are standard settings designed to correspond to the 80-column PLAN coding sheet. A tab in columns 1 to 6 causes advance to column 7; a tab in columns 7 to 12 causes advance to column 13; a tab in columns 13 to 15 causes advance to column 16, and so on. The complete series of tab stops is columns 7, 13, 16, 36, 70, 73, 76 and end of line.

Although they are designed for PLAN, a programmer may take advantage of this facility to lay out an EMA program.

Five-track paper tape is an acceptable input medium for 1900 EMA and a possible, though not recommended, output medium: it is provided largely for compatibility reasons. The five-track code is shown below.

5-TRACK TAPE CODES

Value	Tape	Printer FS	Printer LS
0	·	FS	FS
1	· ●	1	A
2	·●	2	B
3	·●●	*	C
4	●·	4	D
5	●· ●	(	E
6	●·●	)	F
7	●·●●	7	G
8	● ·	8	H
9	● · ●	≠	I
10	● ·●	=	J
11	● ·●●	-	K
12	●●·	≈(∪)	L
13	●●· ●	LF	M
14	●●·●	SP	N
15	●●·●●	,	O

Value	Tape	Printer FS	Printer LS
16	● ·	0	P
17	● · ●	>	Q
18	● ·●	⩾	R
19	● ·●●	3	S
20	● ●·	→	T
21	● ●· ●	5	U
22	● ●·●	6	V
23	● ●·●●	/	W
24	●● ·	φ(×)	X
25	●● · ●	9	Y
26	●● ·●	+	Z
27	●● ·●●	LS	LS
28	●●●·	.	.
29	●●●· ●	'(≠)	?
30	●●●·●	CR	£ (π)
31	●●●·●●	ER	ER

Symbols within parentheses appear on some sets of equipment.

Abbreviations used in this Table

CR = carriage return
ER = erase
FS = figure shift
LF = line feed
LS = letter shift
SP = space

Note that a system of Figure Shift and Letter Shift is used to expand the 32 basic combinations possible in five-track tape. Even so, the full 1900 EMA character set is not available, although the missing characters are not essential; X≤0, for example, can be written as 0 ⩾ X.

There is no individual newline character. Carriage Return (CR) moves the carriage of the tape equipment horizontally to the start of a line. Line Feed (LF) moves the paper vertically a distance of one line.

Thus, CR LF is equivalent to one newline, CR LF LF is equivalent to two newlines, and so on, and this is how EMA treats these codes. If, for example, CR LF is encountered on input, it is treated as newline and similarly, CR LF CR LF CR LF is treated as three newlines. The exact rule is that the number of newlines in a string containing one or more LFs is the same as the number of LFs; and a string of CRs with no LFs gives one newline. A string is terminated by any character except CR, LF or a shift character. On five-track output, each newline results in CR LF.

If superfluous shift characters or Erase characters are present, they are normally ignored but note that:

(a) LS is not permitted in numerical data even if it is immediately cancelled by an FS character

(b) the I = φ TAPE instruction (see Section 5.2) recognizes *all* characters.

On input, the character ≈ is treated exactly as the character = (the Mercury distinction between these two characters being unimportant in 1900 EMA) but note that, again, I = φ TAPE will distinguish between them.

If there is an attempt to output an illegal character, then a reasonable substitution will be made or the character "?" will be output as an error indication.

If a document is recorded on a single section of tape, this tape should start with at least one foot of runout (blank characters) which will be ignored. The tape must end with

CR LF ***Z CR LF

followed by at least six inches of runout.

If more than one section of tape is used, intermediate tapes must end as above, but with ***Z replaced by ***T.

Line Printer 4.3.5

The character set and substitutions used for punched cards are also used for line printer output.

Maximum line length is 96, 120 or 160 characters depending upon the model of line printer available.

If there is an attempt to output an illegal character then a reasonable substitution is made or the character "?" will be output as an error indication.

A throw to the top of a page can be achieved by the Runout instruction (see Section 1.4.4) or by the output of a PT character (see Section 2.3.4).

Mixed Documents 4.3.6

In program, but *not* data, a document may spread over more than one type of input medium. For example the main program may be recorded in eight-track tape but certain Routines may be available in the form of lengths of five-track tape prepared for another computer.

In such cases, it is possible to switch to another medium at any stage of the program by extending the ***T terminator as follows:

	***T TR	(switch to eight-track tape)
or	***T TR/FIVE	(switch to five-track tape)
or	*** T CR	(switch to cards)

Comments 4.3.7

In both program and data, the standard method of introducing comment is by means of a Vertical Bar (an exclamation mark in 1900 EMA) which is interpreted as a newline and causes the remainder of the line to be ignored. In program, however, an alternative method is provided, largely for use with five -track paper tape which has no code for exclamation mark.

A *complete* line of comment can be introduced and preceded by two 'greater than' signs. For example

>>END OF COMPUTATION

would be treated by the compiler as newline. Such comment cannot be used in data and cannot appear as *part* of a line.

Peripheral Description Directives 4.3.8

Within the body of an EMA program, documents are referred to by an integer in such instructions as:

Select Input 3

Select Output 14

The integers range from 1 to 15 for both input documents and output documents, though the numbers chosen need not be consecutive. These integers are quite arbitrary and do not refer to specific peripherals. The programmer must therefore associate each number used in the program with a particular peripheral type. This is done by writing peripheral description directives in the program description. Program descriptions are described in detail in Sections 4.4, 4.5 and 4.6, but basically a program description consists of a set of EMA directives written before the first Chapter or Routine of a program.

BASIC PERIPHERALS

The term *basic peripheral* refers to card readers and punches, paper tape readers and punches, and line printers.

Input document numbers are related to specific peripherals by directives of the form

INPUT *K* = *Z*

where *K* is the document number used in the program, and *Z* is a code describing the peripheral required. The simplest form of *Z* is two letters followed by a number. The two-letter code for a tape reader is TR, and for a card reader CR. The number is used to differentiate between units of the same type, for example:

INPUT 1 = TR0

INPUT 2 = TR1

INPUT 3 = CR0

In this program, input document numbers 1 and 2 will be on tape readers, and input document number 3 will be on a card reader.

If five-track tape is to be input, the qualifier /FIVE is used, for example:

INPUT 15 = TR2/FIVE

Input document 15 is punched on five track tape and is to be input on a tape reader allocated as TR2.

Output document numbers are defined in a similar way by the directive

OUTPUT *K* = *Z*

where *K* is as above. Codes for *Z* are:

card punch CP

tape punch TP

line printer LP

Five-track paper tape can be output by using the qualifier/FIVE, for example:

OUTPUT 2 = TP0/FIVE

A qualifier can be used with a line printer to specify the number of print positions required. The qualifier takes the form /*n*, where $n \leqslant 160$. The effect is to specify the maximum number of characters per line, and the compiler will arrange to print a line of greater length on two, or more, lines. If $n \leqslant 96$, any printer will be used. If $96 < n \leqslant 120$, only a 120 or a 160 position printer will be used. If no qualifier appears, /120 is implied. For example:

OUTPUT 1 = TP0

OUTPUT 2 = CP0

OUTPUT 3 = LP1/96

OUTPUT 4 = LP2/160

Document number 1 is output on a tape punch, number 2 on a card punch, number 3 on any line printer, printing up to 96 characters per line, and number 4 on a 160 position printer.

On some machines or if output is off-lined by GEORGE 2 or 3 it is impossible to control which printer is assigned. If a printer with less than *n* print positions is assigned, the superfluous characters may be lost, or the program may fail.

On entry to a program, document number 1 will be selected for input, and the first output specified in the program description will be selected for the output document, unless specified otherwise by a Select instruction. EMA programs should always have a basic output peripheral selected, otherwise error indications will be lost.

If for some reason it is desired to suppress all transfers to a particular peripheral, the qualifier /NONE can be used, for example:

OUTPUT *K* = /NONE

All Write instructions to channel *K* will have no effect, and no peripheral will be engaged.

3146/10.69

PAGING

When a line printer is being used for an output document, a check is made on the position of the line just printed, relative to the bottom of the page. When the bottom of a page is reached, the next line is automatically sent to the top of a new page. This facility may be suppressed by two methods.

1 The programmer may write in the program description the directive:

NO PAGING

All output to line printers during the running of the program will be on consecutive lines. Lines may be printed over the perforations.

NO PAGING is included to allow graphs to be drawn on the line printer.

2 The machine operator may suppress paging by the console message:

ON *#name* 23

Paging may be reinstated by the console message:

OFF *#name* 23

If the programmer requires paging suppression to be switched on and off during the running of the program, he should write an instruction such as:

17) Halt

When this instruction is obeyed, Executive will halt the program and type on the console typewriter:

0*#name*: HALTED:- 17

The operator must be instructed to type the required ON or OFF message when the above message is output, and to restart the program by the message:

GO *#name*

MAGNETIC TAPE

Instructions to program magnetic tape are available in EMA. These will be dealt with in the next section. The programmer can also specify magnetic tape when using the standard EMA input and output instructions. The channel number is associated with a magnetic tape deck by the following directives

INPUT K = $Z(F(g))$

OUTPUT K = $Z(F(g))$

where K is the channel number, Z is the peripheral code, and $F(g)$ is the file name. Z can be MT1 to MT15. Magnetic tape usage is described in Section 4.4.2

For example the following directives might be written in a program description:

INPUT 2 = MT1 (DATAFILE1(0))

INPUT 3 = MT2 (DATAFILE2(0))

OUTPUT 2 = MT3 (OUTPUTFILE20(1))

Input channel 2 is a magnetic tape file named DATAFILE1, this tape is on a unit allocated as MT1; channel 3 is a file DATAFILE2 on unit MT2. Output Channel 2 is a file named OUTPUTFILE20 with a generation number 1, on a unit allocated as MT3. The generation number of the output tape will be checked, but not the input tapes since both generation numbers are specified as zero.

The output statement refers to a tape that already has a file name. A further peripheral description directive is available to create a new file. This is as follows:

CREATE K = $Z(F(g,r))$

A scratch tape will be found, and given the specified file name. If the generation number is omitted, a generation number zero is implied. If the retention period is omitted a large retention period is implied.

This tape can be used only for output.

The effect of these directives is as follows. On the first instruction the file specified is searched for, or if the instruction is CREATE, a scratch tape is sought. When found, the deck containing the file is

allocated as the unit number specified, and the file is opened. For CREATE, the scratch tape is opened and the specified file name is written. The input or output instruction is then obeyed. Using these instructions, magnetic tape is treated like the basic peripherals, except that transfers are much faster. This means that tapes cannot be rewound or backspaced, and then re-read. Any previous information on a tape used for output will be overwritten. To terminate an output file, the instruction

Relinquish Output K

should be used. An End-of-File trailer label is written, the tape rewound, disengaged, and de-allocated. This occurs automatically at an error halt, and at the End instruction.

If an End-of-File trailer label is read on input, an error is reported (Fault Number 51).

Care should be taken when numbering the magnetic tape units. Peripheral description directives should not refer to the following numbers in the specified instances:

1 MT0, if the object program is to be overlaid.

2 MT7, if MT DUMPS are in use.

3 MT8 to MT15 if EMA magnetic tape instructions are being used.

Standard input and output should therefore use only MT1 to MT6, unless more than six tapes are needed at any one time.

If the same file is referred to by both an INPUT and an OUTPUT directive, it *must* be relinquished as an output document before it can be selected as an input document.

Tape reels that contain, or will contain, files defined by the OUTPUT and CREATE directives must have a write-permit-ring fitted.

No check is made on a write-permit-ring for tape reels that contain files defined by the INPUT directive.

SUBFILES

The peripheral description directives given above allow simple files on magnetic tape to be used for standard EMA input and output instructions. When this is done each input and output document will physically correspond to one reel of tape. The programmer may, however, find it more convenient to have more than one document on a reel of tape. Facilities are available, therefore, for composite files on magnetic tape to be used for standard EMA input and output instructions. For an input document, the peripheral description directive is

INPUT $K = Z(F(g).S(g_s))$

where $F(g)$ is the filename, and $S(g_s)$ is the subfile name.

The effect of this instruction is as follows. On the first input instruction referring to channel K, the file $F(g)$ is opened and scanned for subfile $S(g_s)$. When $S(g_s)$ is found, the input instruction is executed. If a generation number is either omitted or specified as zero, no check will be made on that generation number.

For an output document, the peripheral description directive is:

OUTPUT $K = Z(F(g).S(g_s))$

On the first output instruction referring to channel K, the file $F(g)$ is opened and scanned for subfile $S(g_s)$. When $S(g_s)$ is found the output instruction is executed. Any information in $S(g_s)$ or in any subsequent subfile will be lost. If $F(g)$ contains no $S(g_s)$, then a new subfile $S(g_s)$ will be created at the end of the file. Omission of g_s implies a g_s of zero.

To create a new composite file, the following peripheral description directive should be used:

CREATE $K = Z\ (F(g,r).S(g_s))$

The first output instruction referring to channel K will cause a scratch tape to be found and given the filename $F(g,r)$. The subfile $S(g_s)$ will be created on this tape as the first subfile.

The output instruction will then be executed. If r is omitted, the file will be given a large retention period. The omission of g_s implies a g_s of zero. A subfile does not have a retention period.

When a composite file is used, any number of input and output documents can refer to subfiles on the same file, providing that each channel number appears only in one peripheral description directive, but each channel number *must* be used in turn, and that input (or output) must be relinquished before another is selected. Channel numbers defined by an INPUT directive can be re-selected, but other channel numbers should only be used once.

Example

An EMA program is written which will create a composite file INFOFILE(0,4095) containing subfiles FDATA1, FDATA2, FDATA3, each with zero generation number. During the running of this program it is required to re-input data from these subfiles. The program description will include the following directives:

```
CREATE 1 = MT1 (INFOFILE(0,4095).FDATA1)
OUTPUT 2 = MT1 (INFOFILE.FDATA2)
OUTPUT 3 = MT1 (INFOFILE.FDATA3)
INPUT 13  = MT1 (INFOFILE.FDATA1)
INPUT 14  = MT1 (INFOFILE.FDATA2)
INPUT 15  = MT1 (INFOFILE.FDATA3)
```

The program could be written as follows:

```
    CHAPTER 1                 to output information for FDATA1.
    .................
    Up
    CLOSE
    CHAPTER 2                 to output information for FDATA2.
    . . . .
    Up
    CLOSE
    CHAPTER 3                 to output information for FDATA3.
    . . . .
    Up
    CLOSE
    CHAPTER 0
    . . . .
1)  Select Output 1           This creates the file INFOFILE(0,4095) subfile FDATA1
    Down 3/1
    . . . .
    Relinquish Output 1       Rewind and close file INFOFILE
    . . . .
    At this point Input 13 may be selected but not 14 or 15
2)  Select Output 2           This opens INFOFILE again, FDATA2 is not on this tape so is added
    Down 76/2                 at the end
    Relinquish Output 2       Rewind and close file INFOFILE
    . . . .
    At this point Inputs 13 and 14 may be selected, but not 15
3)  Select Output 3           This opens INFOFILE again, FDATA3 is not on this tape so is added
    Down 10/3                 at the end
```

Relinquish Output 3 **Rewind and close file INFOFILE**

Any of Inputs 13, 14 and 15, may now be selected

. . . .

End

CLOSE

It should be noted that the file must be rewound and re-assigned before another subfile can be opened. The instructions labelled 1) in Chapter 0 should be executed once only. Similarly, if the contents of the subfiles are to be preserved, the instructions labelled 2) and 3) must be executed once only.

BLOCK LENGTH

If magnetic tape is used for standard EMA input and output the block length is normally 128 words. If, for some reason, the programmer requires a different block length, a qualifier can be added to the peripheral description directive as follows:

INPUT K = MTn ($F(g)$)/m

Input document K is to be a simple magnetic tape file $F(g)$ with a block length of m, (where m should not be greater than 4095) and this tape will be assigned as MTn.

Example

CREATE 3 = MT6 (GROUPFILE(1,999). MYSUBFILE(0))/512

The first instruction using output document 3 will cause a scratch tape to be found and assigned as MT6. This tape will be renamed GROUPFILE, generation number 1, retention period 999 days, and a subfile MYSUBFILE, generation number 0, will be created. The block length of the subfile will be 512 words.

It should be noted that the block length must always be specified if it is not 128 words. For a composite file the block length specified applies only to the subfile, and not to the whole file.

EMA MAGNETIC TAPE INSTRUCTIONS

Instructions specifically intended for use with magnetic tape are described in Section 2.6. These instructions refer to each tape file by an integer in the range 0 to 7. These integers must be related to the required file names by peripheral description directives as follows:

INPUT MT K ($F(g)$)

The tape file $F(g)$, assigned as unit MT (K+8), can be used for all the EMA magnetic tape instructions except Write (R,AI).

OUTPUT MT K ($F(g)$)

The tape file $F(g)$, allocated as unit MT (K+8), which has not previously been written using EMA magnetic tape instructions, can be used for all the EMA magnetic tape instructions. This tape *must* have a write-permit-ring fitted to the reel. Writing to the tape must precede any attempt to read from it.

USE MT K ($F(g)$)

The tape file $F(g)$, allocated as unit MT (K+8), can be used for all EMA magnetic tape instructions. This file will have previously been written to using EMA instructions, and reading may precede writing. This tape *must* have a write-permit-ring fitted to the reel.

In these three directives, if the generation number is specified as zero, or is omitted, no check on generation number will be made.

CREATE MT K ($F(g,r)$)

A scratch tape is obtained, allocated as unit MT(K+8), and renamed $F(g,r)$. If the retention period is omitted, a large retention period is implied.

If a scratch tape is required, and is to be left scratch at the end of a program, the directive

USE MT K

can be used.

3146(10.69)

Example

A program is written which updates a data file held on magnetic tape, using a parameter file, to produce a new data file, and also produces an output file. The program description includes the following directives:

```
INPUT MT 0 (DATAFILE(36))
INPUT MT 1 (PARAFILE)
CREATE MT 3 (DATAFILE(37,14))
CREATE 2 = MT6 (OUTPUTFILE(0,999))
```

An input file DATAFILE(36) is required. This is allocated as MT8 and referred to in the program as tape 0. The file PARAFILE is allocated as MT9 and referred to in the program as tape 1. This tape is also used for reading only. The new file DATAFILE(37,14) is created and is referred to as tape 3 within the program, and allocated as MT11. A retention period of fourteen days is specified. A new file OUTPUTFILE(0,999) is also created, but is written using standard instructions. It is allocated as MT6, and referred to as output 2 within the program. Tapes MT6 and MT11 must have write-permit-rings fitted, and MT8 and MT9 must not.

All tapes can be closed before the program ends, by the EMA instructions:

```
Relinquish Deck (0)
Relinquish Deck (1)
Relinquish Deck (3)
Relinquish Output 2.
```

but this is not essential, as the operation is automatic at the end of the program.

DISC

The programmer can specify E.D.S. or F.D.S. when using the standard EMA input and output instructions. The channel number is associated with a disc file (E.D.S. or F.D.S.) by the following directives:

INPUT $K = Zx\ (F\ (g))/N$

CREATE $K = Zx\ (F\ (g))/N$

where K and $F\ (g)$ are as for magnetic tape, x is the unit number and N is the size of bucket in words (if omitted, a bucket size of 128 words is assumed). Z may be ED (for E.D.S.) or FD (for F.D.S.)

The INPUT directive will cause the compiler to associate the input channel number K and the unit number x with the simple file $F\ (g)$. The CREATE directive will cause the compiler to associate the output channel number K with the simple file $F\ (g)$. This file is created with data commencing in bucket number 1.

The generation number will be checked in both cases and if omitted a file of highest generation number will be opened.

Example

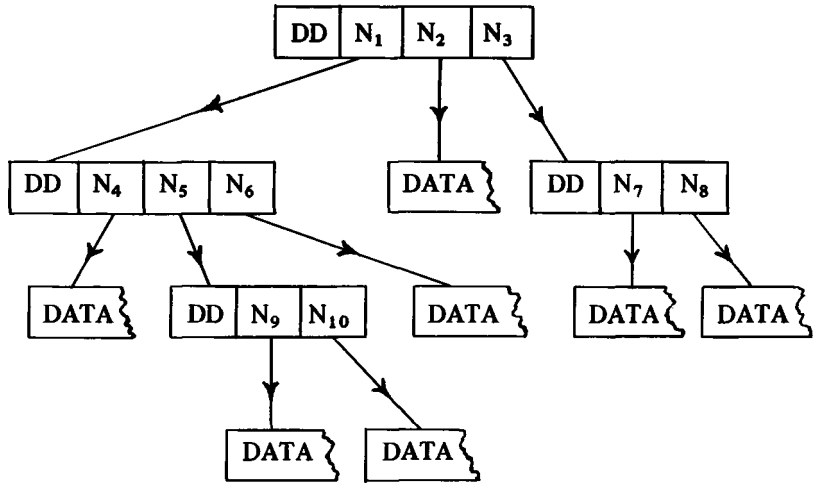
```
INPUT 2    = ED1 (INFILE1(0))
INPUT 3    = ED2 (INFILE2(0))
CREATE 2 = ED3 (OUTFILE5(2))
```

Input channel 2 is an E.D.S. file named INFILE1 on a unit allocated as ED1; similarly for input channel 3. Output channel 2 is a file named OUTFILE5 with a generation number of 2 on a unit allocated as ED3.

Subfiles

The peripheral description directives given above allow simple files on disc to be used for standard EMA input and output instructions. Facilities are also available for composite files on disc to be used. A typical subfile structure is shown in the following diagram.

DD represents the Directory Description record which contains information about the Subfile Description records (SFDs) contained in its structure. A subfile DD, similarly, contains information about the SFDs at the next level. In the diagram, N_1, N_2, etc. represents SFDs.



Note:The above diagram is schematic and is intended to show only the general form of a multi-level subfile structure.

No retention periods are allowed and generation numbers are not updated. No renaming facility is available. Both input and output composite files are opened in overlay mode. Simple files are opened in either input or overlay mode. The standard action is taken if the file generation number is omitted.

The INPUT directive for a composite file has the format:

INPUT K $= Zx\ (F\ (g).\ S_1\ (g_1).\ ---\ S_n\ (g_n).\ )/N$

where $K,Z,x,F\ (g)$ and N are as above. $S_1\ \ldots\ S_n$ are the subfile names and are optional; each subfile name can contain up to 12 characters. $g_1\ \ldots\ g_n$ are the subfile generation numbers; if the generation number g_n for a subfile S_n is omitted, a subfile with the name S_n and the highest generation number will be opened.

The string of subfiles in the directive represents the path to be followed on the file to the relevant data subfiles. If there are no subfiles mentioned, the file is assumed to be a simple file and reading commences from the first bucket of the file. If subfiles are mentioned in the directive, a check is made for the existence of the specified SFDs.

Fault numbers 65 and 66 may occur and are described on page 61.

The last subfile in the directive may be a data or directory type SFD; any preceding SFDs must be directory types.

The OUTPUT directive for a composite file has the format:

OUTPUT $K = Zx\ (F\ (g).\ S_1\ (g_1).\ ---\ S_n\ (g_n).\ )/N$

This directive assumes that a subfile structure exists on the file.

Fault number 65 may occur and is described on page 61.

New SFDs are created on the file, commencing with the first SFD in the directive which is found to be missing on the file. The last SFD in the directive is the data subfile; any preceding SFDs must be directory types.

The CREATE directive for a composite file has the format:

CREATE $K = Zx\ (F\ (g).\ S_1\ (g_1).\ ---\ S_n\ (g_n).\ )/N$

The file $F\ (g)$ is assumed to contain no useful information and a structure, defined by the directive, is created.

The last SFD in the directory is the data subfile and the preceding subfiles are created as directory SFDs. If the generation number (g_n), is omitted, the subfile will be given a generation number of zero.

Files will be extended onto the cartridge containing the last file area if necessary.

Examples

The following examples use the above diagram as reference.

1 The structure would have been created with the directive:

CREATE K = ED $x\ (F\ (g).\ N_1\ (g_1).\ N_4\ (g_4))$

2 Other subfiles would have been created with directives such as:

OUTPUT K = ED $x\ (F\ (g).\ N_1\ N_5\ N_9)$

3146(10.69)

and

OUTPUT K = ED x (F (g). N_2 (g_2))

3 To read the data contained in N_{10} the directive would be:

INPUT K = ED x (F (g). N_1 (g_1). N_5. N_{10})

to read all the data contained in N_7 and N_8 in that order the directive is:

INPUT K = ED x (F (g). N_3)

If a directive of the last form is used, the data subfiles read are in the order they appear in the directories, preference being given to directory SFDs.

For example,

INPUT K = ED x (F (g). N_1 (g_1))

would read the data from subfiles N_4, N_9, N_{10}, and N_6 in that order.

Use of Several Subfiles on the Same File

Due to the structure of disc files, it is possible to have open simultaneously several subfiles on the same file either for reading or writing.

If this facility is used, identical unit numbers should be used in those directives which refer to the same file name.

If an E.D.S. or F.D.S. channel is released, a check is made to see if any other subfiles are still open on that file; if there are, only that subfile is closed and the file remains open to the program. On opening, a similar check is made to see if the unit is already open.

Error Action

If an error originates from the I/O package then no further write operations are carried out on that channel. In general, this means that if a subfile or file is already open for output then the buffer in core is not written away to the file. However, the subfile structure will be corrupted.

If the error is from a direct access monitoring channel, the fault number is displayed on the console typewriter and no further output is made to this channel.

Reporting of Extensions

The extension routine keeps a list of all extensions made to named files. Up to ten extensions can be kept in this list and when the list is full then all ten are immediately reported on monitoring channel in the form:

ED UNIT x NOW *nnnn* BUCKETS

where *nnnn* is the number of buckets in file.

If the list does not exceed nine during the run of program, the extensions are reported at the end of program run on the monitoring channel.

Formats of Disc Files **4.3.9**

The following sections describe the file formats for programs and data on discs.

PROGRAM FORMATS

The compiler handles composite files consisting of a *directory* subfile at level 0 and any number of source subfiles at level 1; one or more of these subfiles may contain source and/or semicompiled program to be input to the compiler.

The directory subfile consists of an index enabling the data subfile to be accessed. The first record in the directory subfile is a description of the directory, known as the directory descriptions (DD). The remaining records describe the subfiles and spare space on the disc and are known as *subfile descriptions* (SFDs).

The formats of the directory and source subfiles are as follows:

DIRECTORY SUBFILES

Header

		Contents	*Meaning*
Word 0	bits 0 to 13	0	No overflow system
	bit 14	1	First bucket in file area
	bits 15 to 23	0	
Word 1	bits 0 to 11	128	126 words available for directory
	bits 12 to 23	*n*+2	*n* words used for directory

Note: Word 0 is not consulted by XMAE and, therefore, bit 14 can be on or off; for object programs compiled by XMAE, word 0 should be zero.

Directory Description

	Contents
Word 0	8
Words 1 to 3	COMPOFILE$_{\nabla\nabla\nabla}$
Word 4	0
Word 5	1
Word 6 and 7	0

Subfile Description for a Source Subfile

	Contents	*Meaning*
Word 0	10	
Words 1 to 3	*name*	name of subfile
Word 4	*b*	subfile starts at bucket *b*
Word 5	*m*	*m* is the number of buckets in the subfile
Word 6	0	
Word 7	1	
Word 8	B2E*x*	where *x* is in the range 0 to 9.
Word 9	*g*	subfile generation number (this is optional with input/output with XMAE)

Subfile Description for Spare Space

This will be omitted if there is no space.

	Contents
Word 0	9
Words 1 to 3	0
Word 4	number of first spare bucket
Word 5	number of spare buckets
Word 6	0
Word 7	1
Word 8	0

3146(10.69)

Notes

The directory subfile may vary from this form. The SFD for the source subfile need not precede that for the spare space and there may be any number of SFDs describing spare space and/or other subfiles. These SFDs may be any length, from nine words upwards.

SOURCE SUBFILE

Each bucket contains a standard bucket header and source records.

Bucket Header

	Contents	*Meaning*
Word 0	0	no overflow system
Word 1 bits 0 to 11	128	126 words available for insertion of source records
bits 12 to 23	$n + 2$	n words actually used to hold source records.

Each bucket contains one or more source records, each record corresponding to one punched card or line of paper tape.

Source Record Format

	Contents	*Meaning*
Word 0 bits 0 to 11	0	no overflow buckets
bits 12 to 23	n	n words in record (including word 0)
Words 1 to n - 1	source lines in character form (space-filled to the right if necessary)	

Notes

Newline is always removed from paper tape lines. Sequence numbers on cards (columns 73 to 80) are removed. Any empty record or a blank source card is held as one word of spaces.

DATA FORMATS

Formatted records

Bucket formats

Each bucket will have words 0 and 1 as a standard bucket header.

	Contents
Word 0	0
Word 1 bits 0 to 11	the minimum number of words moved in a single transfer (that is, the minimum of the bucket size and buffer size)
bits 12 to 23	number of words of information in bucket
Word 2	record 0
onwards	record 1

End of file bucket

	Contents
Word 0 bits 0 to 12	0
bit 13	end of file marker (only checked on input)

	Contents
bits 14 to 23	0
Word 1 bits 0 to 11	bucket size in words (a multiple of 128)
bits 12 to 23	2
Word 2	0
onwards s	

Note: This bucket is used only with simple files.

RECORD FORMAT

	Contents
Word 0 bits 0 to 11	0
bits 12 to 23	count of number of words in record

	Contents
Word 1	record information
onwards	

No specific allowance is made for print control characters. Hence, any file which may subsequently be off-lined should be created with the first character of each record as a print control character which can be interpreted by an off-lining routine.

Note: For formatted files, a buffer size can be specified in the relevant program description statements. If no buffer size is specified, a buffer size of 128 words is assumed.

3146(10.69)

EMA COMPILER XMAM 4.4

Introduction 4.4.1

The information in this section of the manual is concerned solely with the use of the EMA compiler XMAM. This compiler is intended for use on 1900 installations with magnetic tape.

XMAM COMPILATIONS

In common with other 1900 series compilers, XMAM does not translate programs directly into equivalent machine code programs, but first produces *semi-compiled* segments. These segments can be combined by a routine known as the *consolidator* to produce a consolidated semi-compiled program. XMAM outputs consolidated semi-compiled program to a magnetic tape file together with a *General Purpose Loader,* the function of which is to complete the translation into machine code and to load the program into core store. This program, as loaded into core store, is called an *object* program, to distinguish it from the program document, called the *source* program. For an overlay program, the compiler XMAM also produces a *binary* program, which can be loaded directly into core store by Executive. An overlay program is a program which is too large to be held in core store, and the program is held on a backing store, in this case magnetic tape. Parts of the program are loaded into store when required.

The compilation system is designed to permit segments compiled at different times, or segments compiled from different source languages, to be easily consolidated into a single program. In many cases, however, the programmer will write his program in the one language and it will be translated and consolidated in a single computer run; it will then be available for execution either immediately or at a later date.

A separate consolidation run is essential only if the program includes segments written in different languages, which will be produced in semi-compiled form by the compiler for that language. However, if the programmer has chosen to divide his program into different sections to be compiled at different times, he can either arrange a separate consolidation run or consolidate during the last compiling run.

XMAM ENVIRONMENT

XMAM requires a 1900 series central processor with at least 10,752 words of core store, a console typewriter (optional), and floating point instructions, performed either by hardware or by extracode.

Three or four magnetic tape decks are required.

One basic input peripheral is required, either a card reader or a paper tape reader (optionally set for five-track tape), and one basic peripheral for listing, either a line printer or tape punch.

COMPILING SYSTEM DIRECTIVES

When an EMA program has been written according to the rules given in Sections 1 to 3 of this manual, it is prepared for input to the compiler as described in the first part of this section of the manual. The compiler will, however, require more information about the program, and this is supplied in the form of compiling system directives. The use of some of these directives has been explained earlier in this manual. Other directives are described and their use explained in this part of the manual. It should be remembered that these directives control the compiler, and set the various modes of compilation of the source program; they are not EMA instructions.

In this manual all these directives are written in upper case letters only to distinguish them from EMA instructions; lower case letters could appear on a source document. Only round brackets are permitted in directives.

Table 1 lists the compiling system directives, and indicates where position is important, which directives can appear anywhere in a program, and which directives are optional.

Initial Directives	LIST Directive* SEND TO Directive* DUMP ON Directive* RUN Directive*
Program Description Directives	PROGRAM Directives*‡ PRIORITY Directive OVERLAY Directive QUERIES Directive+ TRACE Directive+ Peripheral Description Directives‡ Backing Store Peripheral Directives NO PAGING Directive DUMPS Directives AUXILIARY Directive MAIN Directive DEPTH Directive JUMPDOWN DEPTH Directive CONTINUE Directive Mode Directives LEADERS Directive OMIT COMMENTS Directive
Other Compiler Directives	READ FROM Directives+* COMPLETE Directive*

Table 1: EMA Directives (XMAM)

Notes: * Position important
+ Can appear elsewhere in program
‡ Not optional

Magnetic Tape Usage 4.4.2

The EMA compiler XMAM outputs semi-compiled segments to magnetic tape, and will also accept input from magnetic tape. Programs compiled by XMAM may also use magnetic tape as input and output media and as a backing store. This section gives general information that the programmer will require to use magnetic tape.

MAGNETIC TAPE FILES

Information is stored on magnetic tape in the form of a *file*, and for EMA purposes each file occupies one reel of magnetic tape. A file may be a *simple file* containing one category of information, or it may be a *composite file* holding many categories of information, possibly quite unrelated to each other. A simple file can contain information in one of the following categories:

1 Data

2 Binary Program

3 Consolidated Semi-compiled Program

A composite file contains one or more *subfiles*. In the simplest composite file structure one subfile follows another. A subfile may also contain one or more other subfiles, and this is called a *nested* subfile structure.

Though there is no limit on the depth of nesting, a composite file structure is usually kept fairly simple, with little or no use of nested subfiles.
In addition to the above three categories of information, a subfile may contain information in one of the following categories:

4 One or more Subfiles

5 Unconsolidated Semi-compiled Segments

6 EMA Source Program

Any subfile may contain only one category of information.

NAMED FILES AND SCRATCH FILES

In most computer installations the magnetic tapes in use will be divided into two groups, scratch tapes and named tapes. A scratch tape can be used by any user as a work tape during the running of a program, but once the program has finished, the scratch tape is considered to contain no useful information, and can be used by another program. Named tapes are used when information is to be retained. Each named tape contains a file with a particular name, and is assigned to a particular user or group of users, and this file can be opened only if it is requested by name.

In an installation using magnetic tape, a group of EMA users should have the following magnetic tapes available. A file containing the EMA compiler, libraries, and utility routines would be available for use by anyone in the installation. The group would have general files containing semi-compiled segments that are relevant to the programs being used or written by the group. There may also be other files containing complete programs which are of use to the group. These tapes would be physically identified with the appropriate file names. In addition, each member of the group could be assigned several files for programs and data, which would be for the use of that member only. Correct operating procedure would ensure that none of these files would be available to an 'alien' user without the permission of the EMA group. The leader of the EMA group would ensure that none of the EMA group's tapes would be altered without the group's consent. This system ensures that all files can be accounted for, and that the contents of each file are known. Any scratch tapes required may be taken from the installation's 'pool' as required, and may be returned after use.

To avoid confusion between tapes belonging to different users, the installation will normally introduce conventions for file names.

Some installations do not keep named and scratch tapes as distinct as the system described above, and a user may request a scratch tape and then give it a name. This makes it very difficult to keep track of the 'owner' of a tape and is not generally recommended.

FILE AND SUBFILE NAMES

Each file and subfile is identified by a name consisting of up to 12 characters chosen from the set:

A to Z

0 to 9

Space

Hyphen (-)

The first character must be a letter. All subfiles in a file must have different names. Further restrictions may be imposed by the installation. In this section of the manual, file names are abbreviated to F or F', and subfile names to S or S'.

GENERATION NUMBER

No two files in an installation should have the same name. Files which contain different versions of the same information may be distinguished by the addition of an integer generation number, written in parentheses after the file name, e.g. MYSOURCEFILE(10). File names with generation numbers are referred to as follows:

$$F(g),\ F'(g'),\ S(g_s),\ S'(g'_s)$$

When a file is being named, the specified generation number is written. When a named file is being opened, the generation number on the file will be checked against that specified. If a zero generation number is specified, or if the generation number is omitted, no check will be made.

3146(10.69)

RETENTION PERIOD

To prevent magnetic tapes being inadvertently overwritten, it is possible to specify the length of time (in days) that the tape should be retained, counting from the day it is named. When this *retention period* has expired, the 1900 system will treat the tape as a scratch tape. This tape could still be obtained by a program that specifies the correct name, but Executive might have allocated the tape to another program.

When a magnetic tape file is being created by the XMAM compiler to contain a program, a high retention period is automatically inserted.

When the programmer is setting up a tape file (by means of a CREATE directive, see Section 4.3.8), a retention period should be specified. The retention period is an integer in the range 0 to 4095, and is written after the file name and generation number as follows:

$F(g,r)$

WRITE PERMIT RING

The write permit ring is a small plastic ring that may be fitted to the tape spool as a further aid to file security. Some operations on magnetic tape require this ring to be present, others require it to be absent, while some make no check. Write permit ring requirements are given, where appropriate.

LISTING DIRECTIVE

During compilation, a complete or partial list of source program lines, and other information about the compilation, is output to a line printer or tape punch. The listing required is specified by one of the following directives.

LIST (LP) LIST	Full listing sent to line printer
LIST (TP)	Full listing sent to a tape punch
SHORTLIST (LP) SHORTLIST	Short listing sent to line printer
SHORTLIST (TP)	Short listing sent to a tape punch

In the absence of any listing directive, SHORTLIST (LP) is assumed, until a listing directive is read. If a listing on a tape punch is required, the appropriate directive must be the first statement of a program read by the compiler.

The LIST directive causes a full listing of all lines of the source program being compiled to be sent to the peripheral specified, together with an indication of each error found, and also some comments to clarify the action taken by the compiler in dealing with certain lines. Error and comment numbers are printed on a line following the listing of the source program line to which they refer.

The SHORTLIST directive gives a listing of the program description, Chapter, Programme, and Routine directives, and also copies the lines which contain errors, or give rise to comments, followed by the error or comment number.

Though the listing peripheral is fixed for the whole program by the initial directive, LIST and SHORTLIST may appear anywhere in the program to alter the mode of listing. This facility can be used when compiling a program where some Chapters are already developed, and the full listing is not required.

When a consolidated program is produced, that is, when the program description is introduced by PROGRAM(*name*), the listing is terminated by a line indicating the core store requirement of the program.

If a fault is found, semi-compiled output will cease. The listing, however, will continue so that any subsequent errors can be found. Comment numbers do not prevent consolidated output.

RUN DIRECTIVE

The directive

RUN

facilitates loading the compiled program by automatically deleting the compiler at the end of compilation and loading the object program into core store ready to be entered (by an operator message, see operating instructions, Section 4.4.8).

This directive must appear before the PROGRAM directive. In the absence of a SEND TO statement, semi-compiled output will be sent to a scratch tape, RUN is assumed, and the program will be loaded from this scratch tape. If RUN is inadvertently used with the SEGMENTS directive it will have no effect.

SEND TO DIRECTIVE

The SEND TO directive specifies the name of the magnetic tape that is to receive the final output from the compiler, and the new name, if the name is to be changed. In the following description of the various forms of the SEND TO directive, $F(g)$, $S(g)$, $F(g,r)$ etc. are as defined in Section 4.4.2; *name* is a 4-character program identifier, described in Section 4.4.4. A write permit ring must be fitted to every tape reel used for XMAM output.

1 SEND TO (MT)

The compiler XMAM outputs the consolidated program as a simple file to a tape with file name PROGRAM▽*name*. SEGMENTS must not be used with this directive. If no such tape is loaded, a scratch tape will be found and renamed PROGRAM▽*name*, with a high retention period.

2 SEND TO (MT, $F(g)$)

The compiler XMAM outputs the consolidated program to a tape with filename $F(g)$. If F is not PROGRAM▽*name*, or if the generation numbers differ, the file is renamed PROGRAM▽*name*, with a high retention period. The tape will contain a simple file. SEGMENTS must not be used with this directive.

3 SEND TO (MT, $F(g).S(g_s)$)

The compiler XMAM opens a file $F(g)$, which must be a composite file, and searches for a subfile $S(g_s)$, which must not be contained in another subfile. When the subfile is found, a new subfile $S(g_s)$ containing either consolidated or unconsolidated semi-compiled program, is written, overwriting the original subfile $S(g_s)$. Subfiles beyond $S(g_s)$ will be lost. If $S(g_s)$ cannot be found, or the g_s specified is incorrect, $S(g_s)$ is added as a subfile at the end of the file. A comment is sent to the listing peripheral.

4 SEND TO (MT, $F(g).S(g_s),. S'(g_s')$)

This is similar to the above directive, except that the subfile $S(g_s)$ is overwritten by the subfile $S'(g_s')$. If $S(g_s)$ is not found, $S'(g_s')$ is written at the end of the file, and a comment listed.

5 SEND TO (MT, $F(g)$, $F'.S(g_s)$)

The compiler opens a file $F(g)$, renames it F', with a high retention period and writes a subfile $S(g_s)$ to it. All original information on the file is lost. This directive allows a new composite file to be created.

6 SEND TO (MT, $F(g)$, F')

The compiler opens a file $F(g)$, and renames it F'. Consolidated program is output to this tape as a simple file. All original information on the file is lost. This directive allows a new simple file to be created.

When a composite file is used or created as in 3, 4 or 5, that file will be correctly terminated by an End of Composite File trailer label. When the directive is omitted, a scratch tape will be used, and will be left scratch.

When an error is found in the source program, or when the compilation is abandoned because there are segments missing, the action is as follows. Any scratch tapes used will be left scratch. If PROGRAM appeared in the program description, the final output tape will be unchanged, but if SEGMENTS was used with the fifth SEND TO above, the file name may have been changed, and the subfile specified in the SEND TO will have been overwritten by an End of Composite File trailer label, unless the error occurred in or before the SEND TO statement.

DUMP ON DIRECTIVE

Output from the XMAM compiler is normally in the form of consolidated semi-compiled segments when a complete program is being compiled, as described in Section 4.4.1. The completion of the translation into machine code is performed by the General Purpose Loader when the program is loaded into core store. The programmer may, however, obtain a final version of the program on magnetic tape. This is known as a *binary* program, and may be loaded directly into core store by Executive without a loader program. If an overlay program is being compiled (see Section 4.10), a binary version of the program will be required and will always be produced if the compilation is error free. A binary program, unlike a semi-compiled program, cannot be re-input to the compiler for recompilation. The directive

1 DUMP ON (MT)

specifies that a binary version of the program is to be produced. The compiler searches for a tape with the filename PROGRAM▽*name*, and outputs binary program to this tape as a simple file. If no such tape is loaded, a scratch tape is obtained and renamed PROGRAM▽*name*, with a high retention period.

An extended version of this directive is also available:

2 DUMP ON (MT, $F(g)$)

The compiler searches for a tape with the filename $F(g)$, renames it PROGRAM▽*name*, with a high retention period and outputs binary program to this file.

When an overlay program is being compiled (that is, a program with OVERLAY directives in the program description) a binary program is automatically output as described for DUMP ON (MT). The binary program can be output to a named file by using the DUMP ON (MT, $F(g)$) directive.

DUMP ON and SEND TO directives may appear in either order before the PROGRAM directive.

Program Description 4.4.4

PROGRAM DIRECTIVE

The directive

PROGRAM (*name*)

specifies the name of the program, which will be used to load and run the program. Up to 12 alphanumeric characters, starting with a letter, may be used for *name*. The first four characters are taken by XMAM to be the name of the program, and the remaining characters can be used as an accounting code by the installation. The directive indicates the start of the program description, and also indicates that a consolidated semi-compiled program is to be produced if no errors are found in the source program.

The directive

SEGMENTS (*name*)

where *name* is as described above, indicates that the source program that follows is incomplete, and unconsolidated semi-compiled segments are to be output. *name* is listed but otherwise ignored by the compiler.

In these directives, and in all program description directives, round brackets must be used.

PRIORITY DIRECTIVE

Some of the larger machines in the 1900 series are Multi-programming; that is, several programs can be in core store at any one time. When the execution of one program is suspended while a peripheral transfer is taking place, Executive will continue with another program. This program may in turn be suspended, and then a third program will be taken, and so on. This system permits the simultaneous use of a large number of peripherals at maximum efficiency, as well as permitting the most effective use of the central processor. To facilitate the organisation of time sharing between programs, each program is given a priority, which will indicate the ratio of peripheral activity to processing. A program with high peripheral activity requires a high priority, and vice versa. A priority is assigned to a program by the following directive in the program description

PRIORITY *n*

where *n* is a decimal integer in the range 1 to 99. If this directive is omitted, a priority of 50 is assumed. This priority is suitable for the majority of EMA programs. The directive will have no effect on a single programming machine

OVERLAY DIRECTIVES

OVERLAY directives are used to define the structure of an overlay program. They may appear anywhere in the program description. The general form is

OVERLAY (a,u) C_1,C_2,...C_n

where C_1,C_2,...C_n is a list of Chapters and Routines that are to be assigned to unit u of overlay area a, where $1 \leq u \leq 1023$, and $1 \leq a \leq 255$.

Chapters are indicated by the letter C followed by the Chapter and Programme number, and Routines are indicated by the letter R followed by the Routine number.

An OVERLAY directive must be contained within one line, but any number of OVERLAY directives may refer to the same area and unit.

Example

OVERLAY (2,3) C1, C2, C3, R7, R3, C1-1, C2-1

OVERLAY (2,3) C3-1, C4-1

OVERLAY (2,4) C1-2, C2-2

Chapters 1,2,3 of Programme 0, Chapters 1,2,3,4 of Programme -1 and Routines 3 and 7 are assigned to unit 3 of overlay area 2; Chapters 1 and 2 of Programme -2 are assigned to unit 4 of overlay area 2.

It is an error to assign Chapter 0 to an overlay area.

A detailed description of the overlay system is given in Section 4.10

QUERIES DIRECTIVES

During the development of a program it is possible to specify intermediate results to be printed by writing a query (?) after the required instructions. An instruction is available to suppress this printing when it is not required, and a more detailed description of this facility is given in Section 4.9.2.

Extra coding is incorporated in the object program at compilation to print queries. Though it is possible to suppress query printing, it is more sensible to omit this coding once a program has been developed. This is done using the QUERIES directives:

COMPILE QUERIES

IGNORE QUERIES

These directives may appear anywhere after the PROGRAM or SEGMENTS directives. If neither directive appears in the program description, COMPILE QUERIES is implied. Thus the programmer can specify where query printing coding is to be incorporated using these two directives.

TRACE DIRECTIVES

The directives

COMPILE TRACE

IGNORE TRACE

control the insertion of trace instructions in a program. These trace instructions output information that allows the path followed by a program to be traced. Their main function is to print out the label number when a labelled instruction is obeyed. As with QUERIES directives, the TRACE directives may appear anywhere after the PROGRAM or SEGMENTS directives. If neither directive appears in the program description, IGNORE TRACE is implied. Trace facilities are fully described in Section 4.9.3.

PERIPHERAL DESCRIPTION DIRECTIVES

Peripheral description directives are described in Section 4.3.8. All the directives described apply to XMAM.

DUMPING DIRECTIVES

The directive

DUMPS *n*

can be used to specify store required for the EMA instructions Preserve and Restore (see Section 2.2.). This directive will reserve enough core store to contain the dumps of the working area of a program. If the programmer requires the use of a backing store instead of core store the directive

MT DUMPS *n*

should be written in the program description. A scratch tape, allocated as unit MT7 will be obtained. This tape is left scratch at the end of the program.

If E.D.S. or F.D.S. is available, they may be used as the backing store instead of magnetic tape by writing the directive:

ED DUMPS $(F(g))n$

or

FD DUMPS

where *F* is the filename

g is the file generation number

n is the number of dumps required

If the file generation number is omitted, the file with the required name and the highest generation number is opened. All information on the named file is overwritten. A scratch file is opened if no file is specified.

AUXILIARY STORAGE DIRECTIVES

The program description directives

AUXILIARY (*I*,*J*)

AUXILIARY (*I*,*J*,*L*)

where *I* and *J* are the first and last numbers of auxiliary variables required, have already been described in Section 2.2.

BACKING STORE PERIPHERAL DIRECTIVES

Backing Store peripheral directives are described in Section 3.2.3.

MAIN DIRECTIVE

The directive

MAIN *n*

has been described in Section 1.2.4. It is used to reserve storage for *n* main variables. Omission implies:

MAIN 480

DEPTH DIRECTIVE

The directive

DEPTH *n*

has been described in Section 1.6.5. It is used to define the maximum depth of Down/Up pairs. Omission implies:

DEPTH 3

JUMPDOWN DEPTH DIRECTIVE

The directive

JUMPDOWN DEPTH *n*

has been described in Section 2.1.4. It is used to define the maximum depth of Jumpdown/Return pairs. Omission implies:

JUMPDOWN DEPTH 8

CONTINUE DIRECTIVE

When a complete program (that is, one introduced by PROGRAM (*name*)) is compiled, compilation normally ceases when Chapter 0 has been read, and consolidation commences.

If the directive

CONTINUE

is written in the program description, the compiler will continue to read source program and semi-compiled segments until the directive COMPLETE is read, but when Chapter 0 has been read, only Chapters and Routines that have been referred to earlier in the program will be consolidated.

MODE DIRECTIVES

To allow programs containing large areas of data, for example, auxiliary variables or main variables, to be compiled, the following directives are available:

EXTENDED DATA

COMPACT DATA

MIXED SEGMENTS

They are needed only in special cases described below.

1 EXTENDED DATA. This directive is required only if a program compiled on a medium processor and requiring more than 32,768 words of core store is to be run on a large processor.

2 COMPACT DATA. This directive is required only if a program compiled on a large processor is to be run on a medium processor or the segments that are required can only operate in compact data mode.

The compiler checks that each segment is suitable for the mode of compiling. All segments compiled by #XMAE and the associated library are marked for use in either mode. Though earlier versions of the compiler produced segments marked for only compact data mode, these segments will operate in both COMPACT and EXTENDED DATA mode programs. The FORTRAN, Algol and PLAN compilers produce segments marked 'either', #XMAE will reject segments that do not satisfy the test.

3 MIXED SEGMENTS. Provided the programmer is satisfied that segments marked for a mode other than that being compiled will operate in the compiled mode, he may override the checks by using the directive MIXED SEGMENTS. The compiler will then accept segments regardless of the mode. This enables, in particular, EMA segments compiled with earlier versions to be consolidated. If neither COMPACT DATA nor EXTENDED DATA appears in the program description, then on a medium processor COMPACT DATA and on a large processor EXTENDED DATA is assumed. MIXED SEGMENTS can be used regardless of whether the mode is defined in the program description.

For EXTENDED DATA compilations the SRE1 subroutine group version number 2 must be used.

EXTENDED DATA DIRECTIVE

The directive

EXTENDED DATA

is required only if a program compiled on a medium sized processor and requiring more than 32,768 words of core store is to be run on a large processor.

LEADERS DIRECTIVE

The directive

LEADERS

causes the leader cues to be output on the output peripheral during consolidation. Its absence will inhibit this output.

OMIT COMMENTS DIRECTIVE

The directive

OMIT COMMENTS

stops the listing of comment lines. In general, comment lines give useful information and at least one run of a program should list them.

Terminal Directive 4.4.5

COMPLETE DIRECTIVE

When the directive

COMPLETE

is read, compilation ceases, and the consolidation phase can begin. This directive is required when the program description is introduced by SEGMENTS or when the directive CONTINUE is set in the program description.

EMA Libraries 4.4.6

All EMA compilers, together with some other compilers, are held on the Scientific Master Library Tape which is provided by I.C.T., and updated periodically. It is recommended that an optimised library tape is used with all XMAM compilations. This reduces tape movement, and consequently reduces compilation time. Items required can be extracted from the Scientific Master Library Tape using the standard library program XPMC. Three items are required for XMAM compilations. These are, in the required order:

#yyyy	Search program similar to #TAPE
#XMAM	EMA compiler (magnetic tape version)
SRE1	EMA library group

The magnetic tape should be given the filename PROGRAMyyyyy.

Other programs and library groups may appear if required, e.g. XPMC. The user can also insert subroutines of his own into the EMA library group SRE1 using the program XPMU. Chapters and Routines compiled initially from EMA source should appear before any subroutines beginning with the % character. This is because implicit calls which the programmer will not know about are set up to these subroutines.

The SRE1 group is scanned during the consolidation phase of a program, and subroutines which have been called for, either implicitly or explicitly, are extracted. This search is usually made when the Close of Chapter 0 has been read. If, however, the programmer has set the CONTINUE directive in the program description, the compiler will continue to read input after the Close of Chapter 0, until the COMPLETE directive is read, but then only those Chapters, Routines, and subroutines that have been explicitly called will be accepted for consolidation. This allows the programmer to use other libraries, or to include his own. A library held in the source language will be completely compiled every time it is input, although only those routines called for are consolidated. This takes up unnecessary machine time and also causes unwanted output on the listing peripheral. It is recommended that libraries are held in semi-compiled form on magnetic tape.

Where several programmers are using the same compiler but different private libraries, all the libraries can be put on the same tape as XMAM and SRE1. This means the operator has only one tape reel to load when compiling any of this group of users' programs. The specifications of the relevant programs XPMU and XPMC are included in the Magnetic Tape section of the LIBRARY SPECIFICATIONS Manual.

It should be noted that each library group is only scanned in the forward direction so if there are two subroutines with the same name in one group, the second will always be ignored.

A library held on the same basic input medium as the source program will be read in after Chapter 0.

The order of input is significant in a library. For example, if a Chapter appears in a library and is referenced before Chapter 0, this Chapter will then be consolidated into the object program. If this Chapter references a Routine, also in the library, this Routine will be consolidated into the object program only if it appears after the Chapter. If it appears before, it will not be consolidated. In fact the compiler will not proceed to the consolidation phase because a segment is missing. Consolidation could be forced in this case by an operator message.

Compiler Input 4.4.7

Input to the EMA compiler XMAM may be from any of the three input media, punched paper tape, punched cards or magnetic tape. Source and semi-compiled program may be mixed and presented to the compiler in any order, except between Chapter 0 and the CONTINUE directive, when order is significant.

Most programs are presented to the compiler held on only one basic input medium, with possibly some input from magnetic tape. Occasionally, however, some segments of a program may be held on a different basic input medium. This usually occurs when segments were originally prepared for another program, or have been written by another user.

INITIAL INPUT MEDIUM

The first input medium is specified by the entry point to the compiler (see operating instructions). The three entry points correspond to:

1 Start the compilation, reading initially from eight-track paper tape

2 Start the compilation, reading initially from punched cards

3 Start the compilation, reading initially from five-track paper tape

It can be seen that some input from a basic peripheral is essential even if the complete program is held on magnetic tape.

COMPILING FROM BASIC PERIPHERALS

When a complete program is not held all on the same basic medium, a switch between peripherals should be made using the following extended terminator directives:

***T TR	Switch to eight-track tape reader
***T TR/FIVE	Switch to five-track tape reader
***T CR	Switch to card reader

A change in peripheral causes the previously used peripheral to be released by the compiler, and the requested one assigned. The compiler continues reading from this peripheral until the end of the program is reached, or a switch is made by another directive.

An I.C.T. eight-track paper tape reader can be set up by an engineer to read five-track tape, using the appropriate exchangeable wiring boards, but if both types of tape are to be used, the programmer should assume that two peripherals will be required.

Example

Consider a program in which Chapter 0 is held on eight-track tape. Chapter 1 and the program description are also held on eight-track tape, but Routine 3 was originally written for another machine and is held on five-track tape.

This is punched into eight-track paper tape	LIST (LP)	
	PROGRAM (ROZZ)	
	INPUT 1 = TR0	
	OUTPUT 1 = LP0	
	MAIN 500	
	CHAPTER 1	
	.....	
	CLOSE	
	***T TR/FIVE	The eight-track reader is released, and a five-track reader assigned.
This is punched into five-track paper tape	ROUTINE 3	
	.....	
	**	
	***T TR	The five-track reader is released, and an eight-track reader assigned.
This is punched into eight-track paper tape	CHAPTER 0	
	.....	
	CLOSE	

MAGNETIC TAPE INPUT

A program or segments, either in source language or semi-compiled form, may be input to the compiler from magnetic tape. Source segments, semi-compiled segments, and possibly consolidated semi-compiled segments may be read from a subfile, but only consolidated semi-compiled program may be input from a simple file. If a program under development requires one Chapter to be re-written, it is much quicker to copy the other Chapters, as semi-compiled segments, from the original semi-compiled file to a new file, than to recompile the whole program from source.

Some information must always be read from a basic peripheral. If a program, and the program description, is held entirely on magnetic tape this information may consist solely of a READ FROM magnetic tape directive that provides the necessary information about where the program can be found. The basic peripheral is retained during the magnetic tape input and if the end of the subfile is reached before the Close of Chapter 0, or the COMPLETE directive, more input will be read from this basic device. It should be noted that the listing directive (see Section 4.4.3), if one is required, must always be the first line read by the compiler.

All the program description directives may be input from magnetic tape, but it is recommended that these directives are always input from a basic device, since the versatility of the compiling system is thus retained.

COMPILING FROM SUBFILES ON MAGNETIC TAPE

For each subfile input from a composite file on magnetic tape there must be a corresponding READ FROM directive. If segments of a program are held in several subfiles, one segment per subfile, then the input from the basic device will consist of the program description, followed by a list of READ FROM directives.

In the two following forms of READ FROM directives, $F(g)$ and $S(g_s)$ are as defined in Section 4.4.2.

READ FROM (MT,$F(g)$.$S(g_s)$)

READ FROM (MT,.$S(g_s)$).

When a file name $F(g)$ is present, the compiler tests to see if that file is currently open. If it is open, the tape is rewound and a search for the subfile $S(g_s)$ is made from the beginning of the file. If it is not open, any input magnetic tape file that is currently opened is rewound and closed. Then the file with the specified name is opened and scanned for the subfile $S(g_s)$.

When the file name is omitted, the search for $S(g_s)$ will start from the current position of the file that is open. If $S(g_s)$ is not found, this file is rewound and scanned from the beginning.

The subfile $S(g_s)$ may contain other subfiles nested to any depth, in which case the whole nest of subfiles will be compiled. On reading the end of $S(g_s)$, the compiler will read from the previous basic input device.

If the programmer requires several subfiles, the list of READ FROM statements should specify the subfiles in the same order as they are held on the tape to minimise the amount of unnecessary tape rewinding.

If the specified subfile $S(g_s)$ cannot be found, the compiler will cease compiling.

Examples

```
1    LIST (LP)
     PROGRAM (CHAP15892)
     INPUT 1 = CR0
     OUTPUT 1 = MT2 (RESCHAP)
     MAIN 250
     READ FROM (MT, EVERYTHING(1).TWINS)
     CHAPTER 0
     .....
     CLOSE
```

A program is contained on cards except for two segments held in a subfile TWINS in a composite file EVERYTHING(1). In fact this subfile is contained in an outer subfile along with several others.

2 SHORTLIST (TP)

SEND TO (MT, CSCZ00-101)

READ FROM (MT, Z00-59.PROGDES)

READ FROM (MT, .SEG-1)

READ FROM (MT, .SEG-2)

READ FROM (MT, .SEG-3)

Each segment of a program is contained in a separate subfile. The names of the subfiles in the order in which they appear on the composite file Z00-59 are PROGDES, SEG-1, SEG-2, SEG-3. Chapter 0 is at the end of SEG-3. This tape will be scanned in the forward direction, and will not be rewound during the compilation.

3 Program MASS is contained in a subfile S03MAS in the composite file MATHSERV(2). Each segment of the program, and the program description which appears first, is contained in a separate inner subfile. A short listing is required on a line printer and the program is to be compiled and run automatically with no compiler output tapes retained. The only directive required to be input on a basic peripheral is:

READ FROM (MT,MATHSERV(2).S03MAS)

COMPILING FROM SIMPLE FILES ON MAGNETIC TAPE

Consolidated semi-compiled segments may be input for re-consolidation with a different program description, possibly with some segments replaced. A simple file can contain only consolidated semi-compiled program. When input by a READ FROM simple file directive, the contents are best thought of as a stream of unconsolidated semi-compiled segments.

The READ FROM simple file directive takes the form:

READ FROM (MT, *F*(*g*))

Example

```
LIST (LP)
SEND TO (MT)
PROGRAM (KANZ)
INPUT 1 = CR0
OUTPUT 1 = MT3 (RESKANZ(0,99))
MAIN 1000
DEPTH 7
JUMPDOWN DEPTH 23
ROUTINE 27
.....
**
READ FROM (MT, PROGRAM KANZ)
```

A consolidated semi-compiled program is contained in a simple file PROGRAM$_{\nabla}$KANZ. It is to be re-consolidated with a different program description and with the segment ROUTINE 27 replaced by a revised version. This revised version is provided in source form on a basic input medium, and is compiled first. The compiler will then ignore the old version.

Obviously, this is a very useful means of revising existing programs when programs tend to be large.

COMPILING FROM SUBROUTINE GROUPS ON MAGNETIC TAPE

A group of subroutines on the Library Tape may be input by a directive of the form

READ FROM (MT, -.*S*)

where *S* is the name of a subroutine group, such as S-RS (PLAN library) or SRF7 (FORTRAN library).

The following restrictions apply:

1 CONTINUE must be set in the program description.

2 The above directive must follow the close of Chapter 0.

3 The directive COMPLETE must terminate the program.

Example

The following program is held in a subfile RDAM in the composite file TREAS-15. Each segment is held in a separate inner subfile. One segment was originally written in PLAN, and uses some subroutines held in the PLAN Library group. This segment has been inserted at the beginning of the SRE1 Library group.

```
LIST (LP)
SEND TO (MT)
PROGRAM (RDAM)
CONTINUE
INPUT 1 = TR0
OUTPUT 1 = LP0
MAIN 230
READ FROM (MT, TREAS-15.RDAM)     | Read program
READ FROM (MT, -.SRE1)            | Scan EMA library
READ FROM (MT, -.S-RS)            | Scan PLAN library
COMPLETE
```

PRIVATE LIBRARIES HELD ON MAGNETIC TAPE

A library may be held in a subfile on magnetic tape. This subfile may be read using a READ FROM directive of the form:

READ FROM (MT, $F(g).S(g_s)$)

Example

A library is held in a subfile JOHNH, of a composite file KPAULL.

```
(Other program description directives)
CONTINUE
(Chapters and Routines)
CHAPTER 0
.....
CLOSE
READ FROM (MT,KPAULL.JOHNH)
COMPLETE
```

When the READ FROM directive is encountered, the file KPAULL will be opened and searched for subfile JOHNH, which will be scanned for required subroutines. If the file KPAULL is already opened, it will be rewound and searched for subfile JOHNH.

This section gives the operating instructions of the EMA compiler XMAM. Though written as a reference document for operators, it is also intended as a programmer's guide for writing the operator's instructions.

The operating instructions are divided into three sections: the first lists the compilation sequence, the second lists the possible messages that may occur during compilation, and the third gives the object program operating instructions.

When a GEORGE operating system is in use, the programmer should supply a Job Description for the compilation and the object program run.

A system macro is available to compile and run a non-overlay EMA program, or a program without a DUMP ON directive, using the XMAM compiler.

The macro command statement is:

EMARUN *%A*, *%B*, *%C*, *%D*, *%E*, *%F*

Parameters are set as follows:

Parameter	*Function*	
%A	input stream to compiler	/0 TR0 } (*document name*) CR0
%B	#XMAM entry point	0 = eight-track paper tape 1 = cards 2 = five-track paper tape
%C	*action*, if there are segments missing	END (abandon) RESUME (force consolidation) GOTO *label*
%D	name of object program	*name*
%E	input stream to object program	/0 TR0 } (*document name*) CR0
%F	time limit of the object program in seconds	for example, 10

The programmer can also write his own macros, for example, to cater for overlay programs, using operating instructions given in the following pages.

Each document should be introduced by a document name when a program is being run under a GEORGE operating system.

Full details will be found in the Operating Systems GEORGE 1 and 2 manual.

Hardware Requirement

Processor:	Any 1900 processor with floating point facilities performed either by hardware or extracode, console typewriter (optional), and enough core store to hold the compiler.
Core Store:	10752 words.
Basic Peripherals:	At least one reader, card and/or paper tape reader (optionally set for five-track tape). Optional listing device, line printer or paper tape punch.
Magnetic Tapes:	The library tape. One named tape and one scratch tape for compiler output. Other magnetic tapes used for input (optional).

Peripheral Usage	LP0	Listing peripheral.
	TP0	Alternative listing peripheral.
	TR0	Paper tape input.
	CR0	Card input.
	MT0	Library tape containing the compiler and the EMA subroutine group SRE1.
	MT1	Scratch tape, left scratch unless used to dump binary object program; or named file specified by DUMP ON directive.
	MT2	Final output tape, or intermediate tape containing consolidated semi-compiled overlay program.
	MT3	Input tape, either simple file containing consolidated semi-compiled program, or composite file containing unconsolidated subfiles. Alternative library tape, if library is not on MT0.

Priority The compiler as supplied has a priority of 50.

Operating Instructions

I COMPILING AND CONSOLIDATING

Narrative	*Console Message*
1 Load the library Tape and other magnetic tapes as requested by the programmer.	
Load the compiler into store.	
It will be extracted from the EMA Library Tape which remains allocated throughout the run as XMAM is an overlay program.	
2 Activate the compiler by one of the following:	
(a) Begin, reading initially from paper tape reader set for eight-track tape –	GO #XMAM 20
(b) Begin, reading initially from punched cards –	GO #XMAM 21
(c) Begin, reading initially from paper tape reader set for five-track tape –	GO #XMAM 22
3 (a) If DUMP ON (MT, $F(g)$) appeared in the program description a tape with the file name $F(g)$ will be opened. If this tape cannot be found Executive will output the message –	7#XMAM; LOAD *F*
Note: *F* must be of the form PROGRAM▽*name*	
The correct tape should be loaded, and Executive will restart the compiler.	
(b) If DUMP ON (MT) appeared in the program description a tape with file name PROGRAM▽*name* will be opened. If this tape cannot be found, XMAM will output the message –	0#XMAM; HALTED:- LOAD PROGRAM▽*name*
The correct tape should be loaded, and XMAM restarted by the message –	GO #XMAM

Narrative	*Console Message*
If the required tape is still not loaded the message will be output again –	0#XMAM; HALTED:- LOAD PROGRAM▽*name*
When the compiler is restarted by – a scratch tape will be opened and labelled PROGRAM▽*name*	GO #XMAM
(c) If no DUMP ON appears in the program description the compiler proceeds directly to step 4.	
4 The program will be read in, compiled, and in most cases consolidated. If PROGRAM is present in the program description, but not CONTINUE, the compiler will proceed to step 6. If CONTINUE is present, the compiler will continue to read source program until COMPLETE is read, and will then proceed to step 6. If however, any errors have been found when the compiler has read all the source program, the compiler will suspend itself, and output the message –	0#XMAM; HALTED:- ZZ
5 If SEGMENTS are being compiled, and there are no errors found, the compiler will close the output, suspend itself, and output the message –	0#XMAM; HALTED:- EC
If errors have been found, the compiler will halt as in step 4.	
6 The compiler searches the Library Tape for the subroutine group SRE1, and extracts the required subroutines. If this group cannot be found the compiler will output the message –	0#XMAM; HALTED:- NL
7 (a) If SEND TO (MT, *F*(*g*)) appeared in the program description a tape with file name *F*(*g*) will be opened. If this tape cannot be found, Executive will output the message –	7#XMAM; LOAD *F*
The correct tape should be loaded, and Executive will restart the compiler.	
(b) If SEND TO (MT) appeared in the program description a tape with file name PROGRAM▽*name* will be opened. If this tape cannot be found, XMAM will output the message –	0#XMAM; HALTED:- LOAD PROGRAM ▽*name*
The correct tape should be loaded, and XMAM restarted by the message –	GO #XMAM
If the required tape is still not loaded the message will be output again –	0#XMAM; HALTED:- LOAD PROGRAM▽*name*
When the compiler is restarted by – a scratch tape will be opened and labelled PROGRAM▽*name*	GO #XMAM
(c) If no SEND TO directive appears in the program description, the compiler proceeds to step 9.	
8 (a) If neither RUN nor DUMP ON was present in the program description, the compiler will consolidate the object program and halt, outputting the message –	0#XMAM; HALTED:- EC
The consolidated object program may be loaded into core store by the message –	FI#*name*

Narrative	*Console Message*
If segments are missing the compiler will halt –	0#XMAM; HALTED:- SM
Blank cues are listed, on the listing peripheral. Consolidation may be forced by –	GO #XMAM
If there are missing overlay segments, the first missing overlay segment is listed on the listing peripheral and the compiler will halt, outputting the message –	0#XMAM; HALTED:- ZZ
No continuation is possible.	
(b) If DUMP ON was present in the program description or an overlay program was being compiled, the compiler will consolidate the object program outputting the message –	0#XMAM; DELETED:- LO#*name ab*
A scratch tape or a named tape is opened on the tape deck with the unit number *cd* and the binary program is written to this tape.	
If RUN was present in the program description the creation of the binary program ends – where *cd* is the unit number of the tape deck holding the binary program.	0#*name*; DELETED:- LO#*name cd*
#*name* is loaded and halts –	0#*name* ; HALTED:- LD
The object program may then be entered by a GO message.	
If RUN was not present the creation of the binary program ends with the message –	0#*name*; DELETED:- AE
The binary object program may be loaded into core store by the message –	FI #*name*
If there are missing segments, the compiler will halt as in 8(a).	
9 If RUN appears in the program description or if RUN and DUMP ON do not appear in the program description, the compiler consolidates the object program.	
(a) If there are no missing segments and a non-overlay program is being consolidated the compiler deletes itself and loads the object program, producing the messages –	0#XMAM; DELETED:- LO#*name ab* 0#*name*; HALTED:- LD
where *ab* is the unit number of the tape deck holding the consolidated semi-compiled program. The object program may then be entered by a GO message.	
(b) If an overlay program is being consolidated the compiler deletes itself, outputting the message –	0#XMAM; DELETED:- LO#*name ab*
A scratch tape is opened and the binary program is written to this tape, ending with the message –	0#*name*; DELETED:- LO#*name cd*
where *cd* is the unit number of the tape deck holding the binary program.	
#*name* is loaded and halts –	0#*name*; HALTED:- LD

Narrative	*Console Message*
10 If only DUMP ON appears in the program description the compiler consolidates the object program, and deletes itself, outputting the message –	0#XMAM; DELETED:- LO#*name ab*
A named tape is used for the binary program, ending with the message –	0#*name*; DELETED:- AE
In both 9 and 10 above, if segments are missing the compiler halts as in 8(a).	
11 If the compiler has not deleted itself i.e. has not reached step 9 (a) or 8 (b) another program may be compiled by –	GO #XMAM 20, 21, or 22
If another compilation is not required immediately peripherals may be freed by typing –	GO #XMAM 27
and the message – will be output, or the compiler may be deleted.	0#XMAM; HALTED:- PF

II CONSOLE TYPEWRITER MESSAGES

Message	*Meaning*
0#XMAM; HALTED:- EC	Compilation complete and error free.
0#XMAM; HALTED:- ZZ	Compilation complete. Errors found.
0#XMAM; HALTED:- PF	Peripherals freed.
0#XMAM; DISPLAY COMPILED *n #name*	*n* = number of machine code instructions generated. *name* = program name (4 characters) plus accounting code (up to 8 more characters).
0#XMAM; HALTED:- CH	Error in the semi-compiled paper tape being input. Restart by pulling back two blocks and typing GO #XMAM.
FIX message	Reader has read off end of incorrectly terminated tape.
0#XMAM; HALTED:- SM	Not all required segments are present. GO #XMAM to force consolidation, if required.
0#XMAM; HALTED:- LP ST etc.	Peripheral or store not available. When available type GO #XMAM to continue.
0#XMAM; HALTED:- NL	Subroutine block not found on the Library Tape. GO #XMAM to search another Library Tape.
0#XMAM; DELETED LO #*name ab*	*ab* is the unit number of the tape deck holding the consolidated semi-compiled program.
0#*name*; DELETED:- AE	Binary program has been obtained.
0#*name*; HALTED:- LD	Object program has been loaded ready for entry.
0#XMAM; HALTED:- E1	Tape deck error. Compiler must be refound.
0#XMAM; HALTED:- LOAD PROGRAM *name*	Load a tape with file name PROGRAMγ*name*, where *name* is the program name as written in the program description (compiler message).
7#XMAM; LOAD *F*	Load a tape with file name F. The programmer has specified this file name in a SEND TO (MT, *F*(*g*)) or a DUMP ON (MT, *F*(*g*)) directive (Executive message).

ENTRIES TO THE EMA COMPILER XMAM

Entry	*Action*
GO #XMAM 20	Compile from eight-track tape.
GO #XMAM 21	Compile from cards.
GO #XMAM 22	Compile from five-track tape.
GO #XMAM 27	Free peripherals.
GO #XMAM 29	Force COMPLETE. Program will be consolidated if possible.

III OBJECT PROGRAM

Narrative	*Console Message*
1 Load object program, if not done automatically by the compiler –	FI *#name*
2 Activate by typing –	GO *#name* 20
3 If the program ends in good order with	
(a) An End instruction, the message will be output –	0*#name*; DELETED:- *ab*
(b) A Halt instruction, the message will be output –	0*#name*; HALTED:- *ab*
where *ab* are the last two digits of the label of the last labelled instruction executed. If there is no such label, the corresponding messages are –	0*#name*; DELETED:- OL 0*#name*; HALTED:- OL
4 If the program fails, there will be comprehensive printout on the users monitor peripheral, and the program will halt with the message –	0*#name*; HALTED:- ER
5 If the object program goes illegal, or loops due to an error in the program that was not checked by the compiler, type –	GO *#name* 28
Post-mortem printouts will be sent to the monitor peripheral, and the program will halt –	0*#name*; HALTED:- ER

EMA COMPILER XMAP 4.5

Introduction 4.5.1

The information in this section is concerned with the use of the EMA compiler XMAP. This compiler uses paper tape as the output medium and is primarily intended for installations without magnetic tape or E.D.S. The input medium is punched cards, or either eight-track or five-track punched paper tape. Since, however, this compiler can also be used on installations with magnetic tape, EMA magnetic tape instructions can be used, and magnetic tape can be used as a backing store, but E.D.S. facilities are not available.

XMAP COMPILATIONS

XMAP will translate EMA Chapters and Routines into semi-compiled segments. Each Chapter, and each Routine, will correspond to one semi-compiled segment. XMAP will also consolidate these segments, together with any segments called from a library tape or produced by another compiler, into a complete object program which can then be loaded into core store and executed. If errors are found in the source program, semi-compiled output will be terminated, but the remainder of the source program will be input to check for further errors. If the program is incomplete, or if only semi-compiled output is required, the programmer can specify that the program should not be automatically consolidated.

XMAP ENVIRONMENT

XMAP requires a 1900 series central processor with at least 13,056 words of core store, a console typewriter (optional), and floating point instructions performed either by hardware or extracode.

One basic input peripheral is required, either a card reader or a paper tape reader (optionally set for five-track tape). A paper tape reader will be required to load the object program.

One paper tape punch is required for output, and one other output device is required for listing, either a line printer or another paper tape punch.

The compiler allots and releases peripherals as required.

Initial Directive	LIST Directive*
Program Description Directives	PROGRAM Directives*‡
	PRIORITY Directive
	QUERIES Directive+
	TRACE Directive+
	Peripheral Description Directives‡
	NO PAGING Directive
	DUMPS Directives
	AUXILIARY Directive
	MAIN Directive
	DEPTH Directive
	JUMPDOWN DEPTH Directive
	CONTINUE Directive
	COPY Directive
Terminal Directive	COMPLETE Directive*

Table 2: EMA Directives (XMAP)

Notes: * Position important
+ Can appear elsewhere in program
‡ Non-Optional

3146(4.68)

Paper Tape Usage 4.5.2

The EMA compiler XMAP can use punched paper tape for both input and output. Input tapes should be prepared as described in Section 4.3.3. Output tapes are as follows.

If there are no errors and the program is complete, the output is a length of tape containing:

A: Semi-compiled segments corresponding to the original source program,

B: Pause block (when read, this halts the input).

C: Request slip. This is a message to Executive, giving the program name, priority, and store requirement, but not giving the peripherals required. A LOAD message is included.

D: General Purpose Loader. This is a short routine to load the program.

E: Consolidated leader information. This gives information on store layout for the General Purpose Loader.

F: Pause block.

After the first pause block (B) there is some blank tape, one delete character, and more blank tape. The tape can be torn off at the delete character.

This tape can then be loaded into core store by inputting items in the order CDE(F)AB. The second pause block (F) is not required if the two lengths of tape are spliced together. If an error is found, this output will cease and the tape will be terminated by four delete characters. Items B to F will not be output, and the tape will not be useful.

Initial Directives 4.5.3

LISTING DIRECTIVE

During compilation, a complete or partial list of source program lines, and other information about the compilation, is output to a line printer or tape punch.

The listing required as specified by one of the following directives:

LIST(LP) LIST	Full listing sent to line printer
LIST(TP)	Full listing sent to a tape punch
SHORTLIST(LP) SHORTLIST	Short listing sent to line printer
SHORTLIST(TP)	Short listing sent to a tape punch

In the absence of any directive, SHORTLIST(LP) is assumed, until a listing directive is read. If a tape punch is required, the listing directive must be the first read by the compiler.

The LIST directive causes a full listing of all lines of the source program being compiled to be sent to the peripheral specified, together with an indication of each error found, and also some comments to clarify the action taken by the compiler in dealing with certain lines. Error and comment numbers are printed on a line following the listing of the source program line to which they refer.

The SHORTLIST directive gives a listing of the program description, Chapter, Programme, and Routine directives, and also copies the lines which contain errors, or give rise to comments, followed by the error or comment number. Though the listing peripheral is fixed for the whole program by the initial directive, LIST and SHORTLIST may appear in the program to alter the mode of listing. This facility may be used when compiling a program where some Chapters are already developed, and the full listing is not required.

When a consolidated program is produced, that is, when the program description is introduced by PROGRAM (*name*), the listing is terminated by a line indicating the core store requirement of the program.

If a fault is found, semi-compiled output will cease. The listing, however, will continue so that any subsequent errors can also be found. Comment numbers do not prevent consolidated output.

PROGRAM DIRECTIVE

The directive

PROGRAM (*name*)

specifies the name of the program, which will be used to load and run the program. Up to 12 alphanumeric characters may be used for *name*, starting with a letter. The first four characters are taken by XMAP to be the name of the program, and the remaining characters may be used as an accounting code by the installation. The directive indicates the start of the program description, and also indicates that the program is to be consolidated if no errors are found in the source program.

The directive

SEGMENTS (*name*)

indicates that the source program that follows is incomplete, and unconsolidated semi-compiled segments are to be output; *name* is listed, but is otherwise ignored by the compiler.

PRIORITY DIRECTIVE

Some of the larger machines in the 1900 series are Multi-programming, that is, several programs can be in core store at any one time. When the execution of one program is suspended while a peripheral transfer is taking place, Executive will continue with another program. This program may in turn be suspended, and then a third program will be taken, and so on. This system permits the simultaneous use of a large number of peripherals at maximum efficiency, as well as permitting the most effective use of the central processor. To facilitate the organisation of time sharing between programs, each program is given a priority, which will indicate the ratio of peripheral activity to processing. A program with high peripheral activity requires a high priority, and vice versa. A priority is assigned to a program by the following directive in the program description

PRIORITY *n*

where *n* is a decimal integer in the range 1 to 99. If this directive is omitted, a priority of 50 is assumed. This priority is suitable for the majority of EMA programs. The directive will have no effect on a single programming machine.

QUERIES DIRECTIVES

During the development of a program it is possible to specify intermediate results to be printed by writing a query (?) after the required instructions. An instruction is available to suppress this printing when it is not required, and a more detailed description of this facility is given in Section 4.9.2.

Extra coding is incorporated in the object program at compilation to print queries. Though it is possible to suppress query printing, it is more sensible to omit this coding once a program has been developed. This is done using the QUERIES directives:

COMPILE QUERIES

IGNORE QUERIES

These directives may appear anywhere after the PROGRAM or SEGMENTS directive. If neither directive appears in the program description, COMPILE QUERIES is implied. Thus the programmer can specify where query printing coding is to be incorporated using these two directives.

TRACE DIRECTIVES

The directives

COMPILE TRACE

IGNORE TRACE

control the insertion of trace instructions in a program. These trace instructions output information that allows the path followed by a program to be traced. Their main function is to print out the label number when a labelled instruction is obeyed. As with QUERIES directives, the TRACE directives may appear anywhere after the PROGRAM or SEGMENTS directive. If neither directive appears in the program description, IGNORE TRACE is implied. Trace facilities are fully described in Section 4.9.3.

PERIPHERAL DESCRIPTION DIRECTIVES

Basic peripheral description directives, and magnetic tape description directives, applied to EMA magnetic tape instructions, are as described in Section 4.3.8. Magnetic tape description directives should not be applied to standard EMA input and output instructions.

DUMPING DIRECTIVES

The directive

DUMPS *n*

can be used to specify store required for the EMA instructions Preserve and Restore (see Section 2.2.). This directive will reserve enough core store to contain the dumps of the working area of a program. If the programmer requires the use of backing store instead of core store the directive

MT DUMPS *n*

should be written in the program description. A scratch magnetic tape, allocated as unit MT7 will be obtained. This tape is left scratch at the end of the program. (This facility assumes that magnetic tape is available.)

Only one of these directives can be used.

AUXILIARY STORAGE DIRECTIVES

The program description directives

AUXILIARY (*I*,*J*)

and AUXILIARY (*I*,*J*,*L*)

where *I* and *J* are the first and last numbers of auxiliary variables required, have already been described in Section 2.2. Auxiliary variables will be held in core store.

MAIN DIRECTIVE

The directive

MAIN *n*

has been described in Section 1.2.4. It is used to reserve storage for *n* main variables.

Omission implies:

MAIN 480

DEPTH DIRECTIVE

The directive

DEPTH *n*

has been described in Section 1.6.5. It is used to define the maximum depth of Down/Up pairs. Omission implies:

DEPTH 3

JUMPDOWN DEPTH DIRECTIVE

The directive

JUMPDOWN DEPTH

has been described in Section 2.1.4. It is used to define the maximum depth of Jumpdown/Return pairs. Omission implies:

JUMPDOWN DEPTH 8

CONTINUE DIRECTIVE

When a complete program (that is a program introduced by PROGRAM (*name*)) is compiled, semi-compiled output ceases when Chapter 0 has been read, and the program is then consolidated. If, however, the directive

CONTINUE

is written in the program description, the compiler will continue to read source and semi-compiled program, but only those segments previously referred to will be accepted for consolidation. The directive COMPLETE is required to terminate this mode of compilation.

COPY DIRECTIVE

The compiler will not normally copy semi-compiled input to the output tape because semi-compiled input can be loaded at run time.

If however, the directive

COPY

is written in the program description, semi-compiled input will be copied. This directive is intended mainly for use when the CONTINUE directive is used to input a library of subroutines in semi-compiled form, and it is inconvenient to load several short tapes at run time.

Note: XMAP does not require a special library of EMA subroutines to be scanned when compiling, as the leaders are part of the compiler, but the library must be present at load time.

Terminal Directive 4.5.5

COMPLETE DIRECTIVE

When CONTINUE or SEGMENTS is written in the program description the directive

COMPLETE

is required to indicate the end of the program.

Compiler Input 4.5.6

Input to the EMA compiler XMAP may be from either punched cards or punched paper tape. Source and semi-compiled program may be mixed and may be presented to the compiler in any order, except between Chapter 0 and the CONTINUE directive, when order is significant. Semi-compiled input will not be copied from input to output unless requested by using the COPY directive, since semi-compiled input is already in a form suitable for loading at object program run time. If semi-compiled input is on cards, produced by some other compiler, then this must be copied onto paper tape by using the COPY directive.

Most programs are presented to the compiler held on only one basic input medium. Occasionally, however, some segments of a program may be held on a different basic input medium. This usually occurs when segments were originally prepared for another program, or have been written by another user.

INITIAL INPUT MEDIUM

The first input medium is specified by the entry point to the compiler (see operating instructions). The three entry points correspond to:

1 Start the compilation, reading initially from eight-track paper tape.

2 Start the compilation, reading initially from punched cards.

3 Start the compilation, reading initially from five-track paper tape.

COMPILING FROM BASIC PERIPHERALS

When a complete program is not held on the same basic medium, a switch between peripherals should be made using the following extended terminator directives:

***T TR	Switch to eight-track tape reader
***T TR/FIVE	Switch to five-track tape reader
***T CR	Switch to card reader

A change in peripheral causes the previously used peripheral to be released by the compiler, and the requested one assigned. The compiler continues reading from this peripheral until the end of the program is reached, or until a switch is made by another directive.

An I.C.T. eight-track paper tape reader can be set up to read five-track tape by an engineer using the appropriate exchangeable wiring boards, but if both types of tape are to be used, the programmer should assume that two peripherals will be required.

Example

Consider a program in which Chapter 0 is held on eight-track tape. Chapter 1 and the program description are also held on eight-track, but Routine 3 was originally written for another machine and is held on five-track tape.

This is punched into eight-track paper tape	LIST(LP)	
	PROGRAM (ROZZ)	
	INPUT 1 = TR0	
	OUTPUT 1 = LP0	
	MAIN 500	
	CHAPTER 1	
	.....	
	CLOSE	
	***T TR/FIVE	The eight-track reader is released, and a five-track reader assigned.
This is punched into five-track paper tape	ROUTINE 3	
	.....	
	**	
	***T TR	The five-track reader is released, and an eight-track reader assigned.
This is punched into eight-track paper tape	CHAPTER 0	
	.....	
	CLOSE	

MIXED LANGUAGE PROGRAMS

Programs written in more than one language are described in Section 4.11. Semi-compiled segments from another compiler are presented to XMAP as for EMA program. These semi-compiled segments will normally require library subroutines. The programmer can arrange to input each of the required subroutines when the program is loaded; however, there will be considerably more operator action required to load several short tapes than to load one longer one.

The programmer may not know which subroutines will be required at run time. He can therefore write CONTINUE and COPY in the program description, and input a complete tape of library subroutines after Chapter 0, terminating this library with the COMPLETE directive. The required subroutines will then be copied onto the output tape

Operator's Guide to the XMAP Compiler 4.5.7

This section gives the operating instructions of the EMA compiler XMAP. Though written as a reference document for operators, it is also intended as a programmer's guide for writing the operator's instructions.

The operating instructions are divided into three sections: the first lists the compilation sequence, the second lists the possible messages that may occur during compilation, and the third gives the object program operating instructions.

Hardware Requirement	Processor:	Any 1900 processor with floating point facilities performed either by hardware or extracode, console typewriter (optional), and enough core store to hold the compiler.
	Core Store:	13056 words.
	Basic Peripherals:	At least one reader, card and/or paper tape reader (optionally set for five-track tape). One paper tape punch and listing device, either a line printer or second tape punch.
Peripheral Usage	TR0	Paper tape input.
	CR0	Optional card input.
	TP0	Paper tape output.
	LP0	Listing peripheral.
	TP1	Alternative listing peripheral.
Priority	The compiler as supplied has a priority of 50.	

Operating Instructions

I COMPILING AND CONSOLIDATING

	Narrative	*Console Message*
1	Load program #XMAP	LO #XMAP
2	Activate the compiler by one of the following:	
	(a) Begin, reading initially from paper tape reader set for eight-track tape –	GO #XMAP 20
	(b) Begin, reading initially from cards –	GO #XMAP 21
	(c) Begin, reading initially from paper tape reader set for five-track tape –	GO #XMAP 22
3	The program will be read in, compiled, and in most cases consolidated. If PROGRAM is present in the program description, but not CONTINUE, the compiler will proceed to step 5. If CONTINUE is present, the compiler will continue to read source program until COMPLETE is read, and will then proceed to step 5. If however, any errors have been found when the compiler has been read all the source program, the compiler will suspend itself, and output the message –	0#XMAP; HALTED:- ZZ
4	If SEGMENTS are being compiled, and there are no errors found, the compiler will close the output, suspend itself, and output the message –	0#XMAP; HALTED:- EC
5	The compiler will consolidate the program if possible, suspending itself, and output the message –	0#XMAP; HALTED:- EC
	or	
	if there are missing segments the compiler will suspend itself and output the message –	0#XMAP; HALTED:- SM
	Consolidation may be forced by –	GO #XMAP

Narrative		*Console Message*
6	Another program can be compiled by typing –	GO #XMAP 20,21 or 22
	or peripherals may be freed by typing –	GO #XMAP 27
	and the message – will be output, or the compiler can be deleted.	0#XMAP; HALTED:- PF

II CONSOLE TYPEWRITER MESSAGES

Message	*Meaning*
0#XMAP; HALTED:- EC	Compilation complete and error free.
0#XMAP; HALTED:- ZZ	Compilation complete. Errors found.
0#XMAP; HALTED:- PF	Peripherals freed.
0#XMAP; DISPLAY:- COMPILED *n #name*	*n* = number of machine code instructions generated. *name* = program name (4 characters) plus accounting code (up to 8 more characters).
0#XMAP; HALTED:- CH	Error in the semi-compiled tape being input. Restart by pulling back last two blocks and typing GO #XMAP.
FIX message	Reader has read off end of an incorrectly terminated tape.
0#XMAP; HALTED:- SM	Not all required segments are present. GO #XMAP to force consolidation.
0#XMAP; HALTED:- LP ST etc.	Peripheral or store not available. When available type GO #XMAP to continue.

ENTRIES TO THE EMA COMPILER XMAP

Entry	*Action*
GO #XMAP 20	Compile from eight-track tape
GO #XMAP 21	Compile from cards
GO #XMAP 22	Compile from five-track tape
GO #XMAP 27	Free peripherals
GO #XMAP 29	Force COMPLETE. Program will be consolidated if possible.

III OBJECT PROGRAM

Narrative		*Console Message*
1	Load the object program, and any extra semi-compiled and library segments required.	LO *#name*
2	Activate by typing –	GO *#name* 20
3	If the program ends in good order with	
	(a) an End instruction, the message will be output –	0*#name* DELETED:- *ab*
	(b) a Halt instruction, the message will be output –	0*#name* HALTED:- *ab*
	where *ab* are the last two digits of the label of the last labelled instruction executed. If there is no such label, the corresponding messages are –	0*#name* DELETED:- OL 0*#name* HALTED:- OL

	Narrative	*Console Message*
4	If the program fails, there will be comprehensive printout on the user's monitor peripheral, and the program will halt with the message –	0#*name* HALTED:- ER
5	If the object program goes illegal, or loops due to an error in the program that was not checked by the compiler, type –	GO #*name* 28
	Post mortem printout will be sent to the monitor peripheral, and the program will halt –	0#*name* HALTED:- ER

EMA COMPILER XMAE 4.6

Introduction 4.6.1

The information in this section of the manual is concerned solely with the use of the EMA compiler XMAE. This compiler is intended for use on 1900 installations with E.D.S. or F.D.S. In this section references to disc imply E.D.S. or F.D.S.

XMAE COMPILATIONS

XMAE will translate EMA Chapters and Routines into semi-compiled segments. Each Chapter and each Routine will correspond to one semi-compiled segment. XMAE will not consolidate these segments, but the programmer can specify that the disc consolidator program, XPCL should be automatically loaded and entered at the end of the compilation. Alternatively, the operator can load XPCL to consolidate a semi-compiled program held on disc. XPCL loads the consolidated object program into core store, and the operator can then enter the object program. XPCL can also output the object program in binary form to a disc file is specified. When the object program is to be overlayed the programmer should specify that a binary program is to be produced, otherwise XPCL will not be automatically loaded by XMAE.

If there are any errors in the source program, no usable semi-compiled output will be produced, but all the source program will be read and listed to check for further errors.

XMAE ENVIRONMENT

XMAE requires a 1900 series central processor with at least 11,520 words of core store, an optional console typewriter, and floating point instructions, performed either by hardware or extracode. A line printer or paper tape punch is required for listing. One basic input peripheral is required, either a card reader or a paper tape reader (optionally set for five-track tape). One or more magnetic tapes can be used for input is required. One or two E.D.S. transports or F.D.S. units are required. One transport or unit can be used, but two are recommended.

Initial Directives	LIST Directive* SEND TO Directive*‡ DUMP ON Directive*
Program Description Directives	PROGRAM Directive* PRIORITY Directive OVERLAY Directive QUERIES Directive+ TRACE Directive+ Peripheral Description Directives‡ Backing Store Peripheral Directives NO PAGING Directive DUMPS Directive AUXILIARY Directive MAIN Directive DEPTH Directive JUMPDOWN DEPTH Directive SEMICOMPILED Directive LIBRARY Directive RUN Directive Mode Directives OMIT COMMENTS Directive
Other Compiler Directive	READ FROM Directive‡*

Table 3: EMA Directives (XMAE)

Notes: * Position important

+ Can appear elsewhere in program

‡ Not Optional

Disc Usage

The EMA compiler XMAE outputs semi-compiled segments to disc and will also accept input from disc. Overlay programs compiled by XMAE will use disc as the program store. In addition, EMA programs can also use disc as a backing store for data.

DISC FILES

Information is stored on disc in the form of a *file*. It is also possible to store information in a file which itself is contained within another file. Such a file is called a *composite file*, and the contained file is called a *subfile*. For EMA purposes a composite file can only contain unconsolidated semi-compiled program which can be input only by a disc compiler.

Disc files may contain information in one of the following categories:

1 Data

2 Unconsolidated Semi-compiled Program

3 Binary Program

4 EMA Source Program

NAMED FILES AND SCRATCH FILES

A *named* file is a file which can only be obtained by a program by specifying the name of the file. A *scratch* file is a file that can be obtained by any program. When a scratch file is relinquished by one program no useful information is assumed to be on the file, and the file can then be used by another program.

DISC FILE ALLOCATION

All named disc files must be allocated by the File Allocator program, XJEC or XJFC. It is not, therefore, possible for the programmer to allocate disc files using an EMA program.

The manager of a 1900 installation using disc will normally allocate E.D.S. cartridges and F.D.S. units to various sections uring the installation. Some restrictions may be imposed on file names at this level. Normally the leader of each section will then allocate files on the available cartridges. For EMA users the file allocation would be as follows.

A program file containing the EMA compiler XMAE, and perhaps utility routines such as XPES and XPEU, should be available to any user. This file should have a name of the form PROGRAM$_{\nabla}$*yyyy*, where *yyyy* are four characters chosen by the manager. This file could also contain other scientific language compilers. XPCL should be held on a file called PROGRAM$_{\nabla}$XPCL. XPCL will also require a file containing the EMA subroutine group. This file should have the name SUBGROUPSRE1. The EMA section will also require one or more files to hold semi-compiled output, one or more files to hold binary programs, for example overlay programs, and files for use as backing store, if required. There may also be other files required to hold semi-compiled segments that are relevant to the programs being used or written by the section, and perhaps files holding complete programs which are of use to the section. In addition each programmer could be assigned files for private use. These files will be set up by the leader of the section using XJEC. EMA disc files require a bucket size of 128 words.

FILE AND SUBFILE NAMES

Each file and subfile is identified by a name consisting of up to 12 characters chosen from the set:

A to Z

0 to 9

Space

Hyphen (-)

3146(10.69)

The first character must be a letter. All subfiles in a file must have different names. There will, however be restrictions imposed on file names in any particular installation. In this section of the manual file names are abbreviated to *F*, and subfile names to *S*.

GENERATION NUMBER

No two files in an installation should have the same name. Files which contain different versions of the same information may be distinguished by the addition of an integer generation number, written in parentheses after the file name, for example, MYSOURCEFILE(10). File names with generation numbers are referred to as follows:

$F(g)$, $S(g_s)$

When a file is opened, the generation number on the file will be checked against that specified. If the generation number is omitted, the file with the highest generation number is opened.

File generation numbers etc. are inserted by XJEC and XJFC.

LISTING DIRECTIVE

During compilation, a complete or partial list of source program lines, and other information about the compilation, is output to a line printer or tape punch. The listing required is specified by one of the following directives.

LIST(LP) LIST	Full listing sent to line printer
LIST(TP)	Full listing sent to a tape punch
SHORTLIST(LP) SHORTLIST	Short listing sent to line printer
SHORTLIST(TP)	Short listing sent to a tape punch

In the absence of any listing directive, SHORTLIST (LP) is assumed, until a listing directive is read. If a listing on a tape punch is required, the appropriate directive must be the first statement of a program read by the compiler.

The LIST directive causes a full listing of all lines of the source program being compiled to be sent to the peripheral specified, together with an indication of each error found, and also some comments to clarify the action taken by the compiler in dealing with certain lines. Error and comment numbers are printed on a line following the listing of the source program line to which they refer.

The SHORTLIST directive gives a listing of the program description, Chapter, Programme, and Routine directives, and also copies the line which contain errors, or give rise to comments, followed by the error or comment number. Though the listing peripheral is fixed for the whole program by the initial directive, LIST and SHORTLIST may appear anywhere in the program to alter the mode of listing. This facility may be used when compiling a program where some Chapters are already developed, and the full listing is not required.

When a consolidated program is produced, that is, when the program description is introduced by PROGRAM (*name*), the listing is terminated by a line indicating the core store requirement of the program.

If a fault is found, semi-compiled output will cease. The listing, however, will continue so that any subsequent errors can be found. Comment numbers do not prevent consolidation.

SEND TO DIRECTIVE

The SEND TO directive specified the name of the disc file that is to receive the semi-compiled output from the compiler. The directive is written as follows

SEND TO (ED, $F(g).S(g_s)$)

Semi-compiled program is output to a subfile $S(g_s)$ which is created as the first subfile in file $F(g)$.

If $F(g)$ is of insufficient length to hold semi-compiled output, extensions of the file will be made, 80 buckets at a time. These extensions are made at present on the same cartridge that holds the File Description Area, so when files are being allocated, the file should always be on the same cartridge as the File Description Area.

DUMP ON DIRECTIVE

When a permanent copy of an object program is required, the binary versions of the consolidated program can be obtained by writing the directive:

DUMP ON (ED)

or

DUMP ON (ED, $F(g)$)

The consolidator XPCL will output binary program to the file $F(g)$ or if $F(g)$ is not specified a file called PROGRAM$_\nabla$NAME of highest generation number available. g is optional and if not specified the file with the highest generation number is taken. Binary program is required for an overlay program and this directive must always be used when an overlay program is to be run.

3146101-590

Maximum efficiency will be obtained from XPCL if the files specified in SEND TO and DUMP ON directives are on different cartridges, though the files can be on the same cartridge.

The file name specified in a DUMP ON directive, $F(g)$ must be of the form PROGRAM∇*name*(g), though g is optional.

DUMP ON and SEND TO directives may appear in either order before PROGRAM directive.

Program Description 4.6.4

PROGRAM DIRECTIVE

The directive

PROGRAM (*name*)

specifies the name of the program, which is used to load and run the program.

Up to 12 alphanumeric characters, starting with a letter, may be used for *name*. The first four characters are taken by XMAE to be the name of the program, and the remaining characters may be used as an accounting code by the installation. The directive indicates the start of the program description, and it also indicates that XPCL is to be automatically loaded and entered to consolidate the semi-compiled program if no errors are found in the source program.

The directive

SEGMENTS (*name*)

where *name* is as described above, indicates that XPCL is not to be used, because either the following program is incomplete, or a consolidated program is not required. *name* is listed but otherwise ignored.

PRIORITY DIRECTIVE

Some of the larger machines in the 1900 series are Multi-programming; that is, several programs can be in core store at any one time. When the execution of one program is suspended while a peripheral transfer is taking place, Executive will continue with another program. This program may in turn be suspended, and then a third program will be taken, and so on. This system permits the simultaneous use of a large number of peripherals at maximum efficiency, as well as permitting the most effective use of the central processor. To facilitate the organization of time sharing between programs, each program is given a priority, which indicates the ratio of peripheral activity to processing. A program with high peripheral activity requires a high priority, and vice versa. A priority is assigned to a program by the following directive in the program description

PRIORITY *n*

where *n* is a decimal integer in the range 1 to 99. If this directive is omitted, a priority of 50 is assumed. This priority is suitable for the majority of EMA programs. The directive will have no effect on a single programming machine.

OVERLAY DIRECTIVE

OVERLAY directives are used to define the structure of an overlay program. They may appear anywhere in the program description. The general form is

OVERLAY (a, u) $C_1, C_2, \ldots C_n$

where $C_1, C_2, \ldots C_n$ is a list of Chapters and Routines that are to be assigned to unit u of overlay area a, where $1 \leq u \leq 1023$, and $1 \leq a \leq 255$.

Chapters are indicated by the letter C followed by the Chapter and Programme number, and Routines are indicated by the letter R followed by the Routine number.

An OVERLAY directive must be contained within one line, but any number of OVERLAY directives may refer to the same area and unit.

Example

OVERLAY (2, 3) C1, C2, C3, R7, R3, C1-1, C2-1

OVERLAY (2, 3) C3-1, C4-1

OVERLAY (2, 4) C1-2, C2-2

Chapters 1, 2, 3 of Program 0, Chapters 1, 2, 3, 4 of Programme -1 and Routines 3 and 7 are assigned to unit 3 of overlay area 2;

Chapters 1 and 2 of Programme -2 are assigned to unit 4 of overlay area 2.

It is an error to assign Chapter 0 to an overlay area.

A detailed description of the overlay system is given in Section 4.10.

QUERIES DIRECTIVES

During the development of a program it is possible to specify intermediate results to be printed by writing a query (?) after the required instructions. An instruction is available to suppress this printing when it is not required, and a more detailed description of this facility is given in Section 4.9.2. Extra coding is incorporated in the object program at compilation to print queries. Though it is possible to suppress query printing, it is more sensible to omit this coding once a program has been developed. This is done using the QUERIES directives:

COMPILE QUERIES

IGNORE QUERIES

These directives may appear anywhere after the PROGRAM or SEGMENTS directive. If neither directive appears in the program description, COMPILE QUERIES is implied. Thus the programmer can specify where query printing coding is to be incorporated using these two directives.

TRACE DIRECTIVES

The directives

COMPILE TRACE

IGNORE TRACE

control the insertion of trace instructions in a program. These trace instructions output information that allows the path followed by a program to be traced. Their main function is to print out the label number when a labelled instruction is obeyed. As with QUERIES directives, the TRACE directives may appear anywhere after the PROGRAM or SEGMENTS directive. If neither directive appears in the program description, IGNORE TRACE is implied. Trace facilities are fully described in Section 4.9.3.

PERIPHERAL DESCRIPTION DIRECTIVES

Peripheral description directives are described in Section 4.3.8. All the directives described apply to XMAE.

BACKING STORE PERIPHERAL DIRECTIVES

Backing Store peripherals are described in Section 3.2.3.

MAIN DIRECTIVE

The directive

MAIN *n*

has been described in Section 1.2.4. It is used to reserve storage for *n* main variables. Omission implies:

MAIN 480

DEPTH DIRECTIVE

The directive

DEPTH *n*

has been described in Section 1.6.5. It is used to define the maximum depth of Down/Up pairs. Omission implies:

DEPTH 3

JUMPDOWN DEPTH DIRECTIVE

The directive

JUMPDOWN DEPTH *n*

has been described in Section 2.1.4. It is used to define the maximum depth of Jumpdown/Return pairs. Omission implies:

JUMPDOWN DEPTH 8

MODE DIRECTIVES

To allow programs containing large areas of data, for example, auxiliary variables, to be compiled, the following directives are available:

EXTENDED DATA

COMPACT DATA

MIXED SEGMENTS

They are needed only in special cases described below.

1 EXTENDED DATA. This directive is required only if a program compiled on a medium sized processor and requiring more than 32,768 words of core store is to be run on a large processor.

2 COMPACT DATA. This directive is required only if a program compiled on a large processor is to be run on a medium processor or the segments that are required can only operate in compact data mode.

The compiler checks that each segment is suitable for the mode of compiling. All segments compiled by #XMAE and the associated library are marked for use in either mode. Though earlier versions of the compiler produced segments marked for only compact data mode, these segments will operate in both COMPACT and EXTENDED DATA mode programs. The FORTRAN, Algol and PLAN compilers produce segments marked 'either'. #XMAE will reject segments that do not satisfy the test.

3 MIXED SEGMENTS. Provided the programmer is satisfied that segments marked for a mode other than that being compiled will operate in the compiled mode, he may override the checks by using the directive MIXED SEGMENTS. The compiler will then accept segments regardless of the mode. This enables, in particular, EMA segments compiled with earlier versions to be consolidated. If neither COMPACT DATA nor EXTENDED DATA appears in the program description, then on a medium processor COMPACT DATA and on a large processor EXTENDED DATA is assumed. MIXED SEGMENTS can be used regardless of whether the mode is defined in the program description.

For EXTENDED DATA compilations the SRE1 subroutine group version number 2 must be used.

OMIT COMMENTS DIRECTIVE

The directive

OMIT COMMENTS

stops the listing of comment lines. In general, comment lines give useful information and at least one run of a program should list them.

EMA Libraries 4.6.5

For XMAE compilations a disc program library file containing the compiler, XMAE, and the consolidator, XPCL, should be on line. This file should be set up using the program library create utility program XPEU. This file is given a name of the form PROGRAM▽yyyy.

In addition XPCL will require a subroutine library file containing the EMA subroutine library group to be on line. This file can be set up using the subroutine library create utility program XPES, and the file should be given the file name SUBGROUPSRE1. XPCL will scan only this library file so any user library must be in this file. Only semi-compiled subroutines should be inserted, and if these subroutines were initially compiled from EMA source program they should be inserted at the beginning of the file. This is because implicit calls which the programmer will not know about may be set up to these subroutines.

Compiler Input 4.6.6

Input to the EMA compiler XMAE may be from any of the four input media: punched paper tape, punched cards, magnetic tape, or disc. Source and semi-compiled program may be mixed and presented to the compiler in any order.

Most programs are presented to the compiler held on only one basic input medium, with possibly some input from magnetic tape or disc. Occasionally, however, some segments of a program may be held on a different basic input medium. This usually occurs when segments were originally prepared for another program, or have been written by another user.

INITIAL INPUT MEDIUM

The first input medium is specified by the entry point to the compiler (see operating instructions). The three entry points correspond to:

1 Start the compilation, reading initially from eight-track paper tape

2 Start the compilation, reading initially from punched cards

3 Start the compilation, reading initially from five-track paper tape.

It can be seen that some input from a basic peripheral is essential even if the complete program is held on magnetic tape or disc.

COMPILING FROM BASIC PERIPHERALS

When a complete program is not held all on the same basic medium, a switch between peripherals should be made using the following extended terminator directives:

***T TR	Switch to eight-track tape reader
***T TR/FIVE	Switch to five-track tape reader
***T CR	Switch to card reader

A change in peripheral causes the previously used peripheral to be released by the compiler, and the requested one assigned. The compiler continues reading from this peripheral until the end of the program is reached, or a switch made by another directive.

An I.C.T. eight-track paper tape reader can be set up by an engineer to read five-track tape using the appropriate exchangeable wiring boards, but if both types of tape are to be used, the programmer should assume that two peripherals will be required.

Example

Consider a program in which Chapter 0 is held on eight-track tape. Chapter 1 and the program description are also held on eight-track tape, but Routine 3 was originally written for another machine and is held on five-track tape.

This is punched into eight-track paper tape

```
LIST (LP)
PROGRAM (ROZZ)
INPUT 1 = TR0
OUTPUT 1 = LP0
```


3146110.691

	MAIN 500	
	CHAPTER 1	
	.....	
	CLOSE	
	***T TR/FIVE	The eight-track reader is released, and a five-track reader assigned.
This is punched into five-track paper tape	ROUTINE 3	
	.....	
	**	
	***T TR	The five-track reader is released, and an eight-track reader assigned.
This is punched into eight-track paper tape	CHAPTER 0	
	.....	
	CLOSE	

DISC AND MAGNETIC TAPE INPUT

A program, or segments, either in source language or semi-compiled form, may be input to the compiler from disc or magnetic tape. Source segments and semi-compiled segments may be read from disc files and subfiles, and magnetic tape subfiles. If a program under development requires one Chapter to be rewritten, it is much quicker to copy the other Chapters, as semi-compiled segments, from the original semi-compiled file to a new file than to recompile the whole program from source.

Some information must always be read from a basic peripheral. If a program, and the program description, is held entirely on magnetic tape or disc this information may consist solely of a READ FROM directive which provides the necessary information about where the program can be found. The basic peripheral is retained during the magnetic tape or disc input and if the end of the file or subfile is reached before the Close of Chapter 0 more input is read from this basic device. It should be noted that the listing directive (see Section 4.6.3), if one is required, must always be the first line read by the compiler. All program description directives may be input from magnetic tape or disc but it is recommended that these directives are always input from a basic device.

COMPILING FROM SUBFILES ON MAGNETIC TAPE

For each subfile input from a composite file on magnetic tape there must be a corresponding READ FROM directive. If segments of a program are held in several subfiles, one segment per subfile, then the input from the basic device will consist of the program description, followed by a list of READ FROM directives.

In the two following forms of READ FROM directives, $F(g)$ and $S(g_s)$, are as defined in Section 4.4.2.

READ FROM (MT, $F(g).S(g_s)$)

READ FROM (MT,.$S(g_s)$)

When a file name $F(g)$ is present, the compiler tests to see if that file is currently open. If it is open, the tape is rewound and a search for the subfile $S(g_s)$ is made from the beginning of the file. If it is not open, any input magnetic tape file than is currently opened is rewound and closed. Then the file with the specified name is opened and scanned for the subfile $S(g_s)$. When the file name is omitted, the search for $S(g_s)$ will start from the current position of the file that is open. If $S(g_s)$ is not found, this file is rewound and scanned from the beginning. The subfile $S(g_s)$ may contain other subfiles nested to any depth, in which case the whole nest of subfiles will be compiled. On reading the end of $S(g_s)$, the compiler will read from the previous basic input device. If the programmer requires several subfiles, the list of READ FROM statements should specify the subfiles in the same order as on the tape to minimize the amount of unnecessary tape rewinding.

If the specified subfile cannot be found, the compiler will cease compiling.

COMPILING FROM DISC

A subfile on disc file may be input to the compiler by writing the following directive:

READ FROM (ED,$F(g).S(g_s)$)

The contents of $S(g)$ will be compiled if source, or copied if semi-compiled.

When #XMAE is used in conjunction with the new version of the disc consolidator program #XPCL, it is possible to consolidate selectively from other subroutine groups or private libraries held on disc files. A program description directive is available in the following forms:

LIBRARY ($Z,F(g)$)

LIBRARY ($Z,F(g).S(g_s)$)

where Z is ED or FD

F is the file name

S is the subfile name of the library of semicompiled segments which are to be selectively incorporated into the object program

g and g_s are the generation numbers of the file and subfile respectively.

If g is not specified, the file of name F is used with the highest generation number within the file. If no subfile name is specified, the standard subfile name SUBROUTINES$_\nabla$ is assumed (as set up #XPES).

The EMA library group SRE1 in the file SUBGROUPSRE1 will be automatically scanned at the end of consolidation unless it is previously scanned using a LIBRARY directive and no more subroutines are required.

It is also possible to consolidate subfiles containing semi-compiled segments using #XPCL without the compiler actually scanning the semicompiled subfile. A program description directive is available in the following form:

SEMICOMPILED ($Z,F(g).S(g_s)$)

where Z,F,S,g and g_s are as for the LIBRARY directive.

The semicompiled subroutines contained in a subfile S of a file F are all consolidated into the object program regardless of the presence or absence of a force-in bit, that is any such subfile is treated by the consolidator in exactly the same way as the semicompiled output subfile produced as a result of the SEND TO directive. This directive has the same effect on the object program as the inter-segment directive

READ FROM (ED, ($ED,F(g).S(g_s)$))

where S contains semicompiled segments. However, when the SEMICOMPILED directive is used, the segments are incorporated into the object program during consolidation, that is, they are not copied across to the SEND TO subfile.

RUN directive

It is possible to load automatically a binary program using the RUN directive. If RUN appears in the program description in conjunction with a DUMP ON ED ($F(g)$) directive, the object program will be automatically loaded into core from the disc file $F(g)$. If the object program is overlaid, the overlays are held on the disc ready for use. The program will stop HALTED LD after loading. If RUN is omitted, DELETED HH occurs and the object program must be found.

Operator's Guide to the XMAE Compiler 4.6.7

This section gives the operating instructions of the EMA compiler XMAE. Though written as a reference document for operators, it is also intended as a programmer's guide for writing the operator's instructions.

The operating instructions are divided into three sections : the first lists the compilation sequence, the second lists the possible messages that may occur during compilation, and the third gives the object program operating instructions.

Hardware Requirement	Processor:	Any 1900 processor with floating point facilities performed either by hardware or extracode, console typewriter (optional), and enough core store to hold the compiler.
	Core Store:	11520 words
	Basic Peripherals:	At least one card reader and/or paper tape reader (optionally set for five-track tape). Optional listing device, line printer or paper tape punch.
	Magnetic Tapes:	One or more magnetic tapes for compiler input (optional).
	Disc:	One or two E.D.S. transports or F.D.S. units. Programs can be compiled and run with one transport, but two are recommended.
Peripheral Usage	LP0	Listing peripheral.
	TP0	Alternative listing peripheral.
	TR0	Paper tape input.
	CR0	Card input.
	MT3	Input tape (optional).
	ED0	File holding compiler used as 0.
	ED1	File to contain semi-compiled output used as 1.
Priority	The compiler as supplied has a priority of 50.	

Operating Instructions

I COMPILATION AND CONSOLIDATION

Narrative	*Console Message*
1 Load E.D.S. cartridges or F.D.S. units for compilation.	
2 Load XMAE –	FI #XMAE#yyyy
where the name of the file containing the compiler is PROGRAM▽yyyy.	
3 Activate the compiler by one of the following:	
(a) Begin, reading initially from paper tape reader set for eight-track tape –	GO #XMAE 20
(b) Begin, reading initially from cards –	GO #XMAE 21
(c) Begin, reading initially from paper tape reader set for five-track tape –	GO #XMAE 22

Narrative	*Console Message*
4 The program will be read in, and compiled.	
(a) If errors are found during the compilation, XMAE will suspend itself and output the message –	0#XMAE HALTED:- ZZ
(b) If SEGMENTS introduced the program description, or an overlay program has been compiled without a DUMP ON directive, and no errors have been found, XMAE will suspend itself and output the message –	0#XMAE HALTED:- EC
XPCL can be used to consolidate the output, if required.	

Narrative	*Console Message*
(c) If PROGRAM introduced the program description, and no errors have been found, XPCL will be loaded automatically by –	0#XMAE DELETED:- FIND #XPCL
and the semi-compiled program will be consolidated.	
If no DUMP ON directive was used, the message output will be – The object program is in core store ready to be run.	0#*name* HALTED:- LD
If a DUMP ON directive is used, the message output will be – The object program must be loaded from disc.	0#*name* DELETED:- HH
5 If the compiler has not been overwritten (i.e. step 4 (c)), another program can be compiled by returning to step 3, or peripherals can be freed by the message – and the message – will be output, or the compiler may be deleted.	GO #XMAE 27 0#XMAE HALTED:- PF
6 If the RUN directive appears in the program description in conjunction with a DUMP ON ED ($F(g)$) directive, the object program will be loaded automatically into core from the E.D.S. file $F(g)$. If the program is overlaid the overlays are held on disc ready for use. After loading the program will stop with the message	HALTED LD
If RUN is omitted, the message will be output and the object program must be found.	DELETED HH

II CONSOLE TYPEWRITER MESSAGES

Message	*Meaning*
0#XMAE HALTED:- EC	Compilation complete and error free.
0#XMAE HALTED:- ZZ	Compilation complete. Errors found
0#XMAE HALTED:- PF	Peripherals freed
0#XMAE DISPLAY:- COMPILED *n #name*	*n* = number of machine code instructions *name* = program name (4 characters) plus accounting code (up to 8 more characters).
0#*name* HALTED:- LD	Semi-compiled output has been consolidated by XPCL, and the object program has been loaded into core store.
0#*name* DELETED:- HH	Semi-compiled output has been consolidated by XPCL, and a binary dump of the object program has been output to an disc file.
0#XMAE HALTED:- CH	Error in the semi-compiled tape being input. Restart by pulling back last two blocks and typing GO #XMAE.
FIX message	Reader has read off end of an incorrectly terminated tape.
0#XMAE HALTED:- ST LP TP etc.	Peripheral or store not available. When available type GO #XMAE to continue.

3146110.68

DISC HALTS

The following halts may occur after an operation involving disc has been attempted.

HALTED G1	File not in system or incorrect generation number.
HALTED G2	Failure of integrity code check.
HALTED C2	Insufficient space in core store for further file descriptions.
HALTED D1	Purge date not exceeded.
HALTED L1	Other file opened when trying to open System File for writing.
HALTED L2	System file opened for writing when trying to open another file.
HALTED C1	System control area full. Unable to extend file.
HALTED M1	No space for file extension.
HALTED M3	File to be extended is not open as a write file.
HALTED M4	Insufficient space for file extension.
HALTED I1	Logical bucket number out of range, or zero.
HALTED K1	Unclearable parity error.
HALTED C3	Auxiliary control area full.
HALTED Z1	Should not occur.

The first six of these error halts may be recoverable. They result from an unsuccessful attempt to open a file, and the attempt can be repeated, possibly with a different cartridge on line, by typing GO #XMAE.

ENTRIES TO THE EMA COMPILER XMAE

Entry	*Action*
GO #XMAE 20	Compile from eight-track tape
GO #XMAE 21	Compile from cards
GO #XMAE 22	Compile from five-track tape
GO #XMAE 27	Free peripherals.

III OBJECT PROGRAM OPERATING INSTRUCTIONS

	Narrative	*Console Message*
1	If the consolidation halted 0#*name* DELETED:- HH, load the object program from disc by the message -	FI #*name*#*xxxx*
	where PROGRAM▽*xxxx* is the name of the file containing the object program in binary form.	
2	When the message –	0#*name* HALTED:- LD
	is output, activate the object program by typing –	GO #*name* 20
3	If the object program ends in good order, the message –	0#*name* DELETED:- *ab*
	output, where *ab* is the last two digits of the last label prior to the END instruction, or if there is no such label –	0#*name* DELETED:- OL
4	If the program fails, there will be a comprehensive printout on the user's monitoring peripheral followed by the message –	0#*name* HALTED:- ER
5	If the program goes illegal, or loops due to an error not capable of being checked by the compiler, type –	GO #*name* 28

Narrative	*Console Message*
This will produce monitor printout on the monitor peripheral, and the program will end with the message –	0#*name* HALTED:- ER

Disc Consolidation 4.6.8

Programs that are compiled by a disc compiler such as XMAE are all consolidated by the consolidator program XPCL. For XMAE, however, XPCL is loaded, entered, and deleted automatically. The user can use XPCL separately, if required, and full details will be found in the XPCL specification in the LIBRARY SPECIFICATIONS Manual.

3146(10.69)

MONITORING 4.7

General 4.7.1

If there are faults in a program, some will be detected during compilation but others will not be discernible until run time when the object program is performed. Fault monitoring at compilation is considered below, and run time fault monitoring is considered in Section 4.8.

Compiler Monitoring 4.7.2

If a fault is found, a fault number will be printed. These numbers indicate faults such as

Too Few Repeats	i.e. more Loop than Repeat instructions in a Chapter or Routine.
Unset Label	e.g. 'Jump 3' when there is no label 3) in the Chapter or Routine

Faults such as Unset Label are not monitored until the end of a Chapter or Routine. A faulty loop instruction may be accepted as an arithmetic instruction but the Repeat would be faulted as not having an associated loop instruction.

After a fault is found, the compiler searches for other faults, but the program cannot be successfully compiled. Some care is needed in interpreting fault monitoring, since one fault may induce the compiler to fault a later, correct, line. For instance, a misprint such as A-9 for A→9 may cause all reference to AI within the chapter to be faulted.

These fault numbers are listed in Section 4.7.3. The fault messages are sent to the output device specified in the LIST directive.

Missing labels are listed at the end of a chapter or routine.

Compilation Fault and Comment Numbers 4.7.3

Fault Number	*Fault*
1	Label set twice in Chapter or Routine.
2	Label referred to but not set; indicated at close of Chapter or Routine.
3	Label out of range.
4	Incorrect set directive, e.g."A→X".
5	No set directive for main variable, or meaningless character string.
6	Incorrect OVERLAY directive, e.g. C0 included.
7	Unknown line in program description.
8	Unrecognizable word at beginning of a line.
9	Incorrect use of VARIABLES directive, e.g. "VARIABLES 0".
10	Fault after a recognized word, e.g. "CHAPTER (2)".
11	Unknown function or incorrect function arguments.
12	Incorrect left-hand side of an assignment instruction. e.g. "A(J/K) = Y + Z".
13	Incorrect conditional expression in a Jump instruction.
14	More than five arguments separated by commas.
15	Unmated brackets in an expression.
16	Faulty division format, e.g. "Y = 1/2F".
17	Meaningless character string.
18	Incorrectly constructed arithmetic expression.
19	Program description settings too large; indicated at end of program description.
20	Inadmissible Mercury machine code instruction.
21	Subfile cannot be found in file specified in a READ FROM directive.
22	Incorrect character after ***T.

3146(10.69)

Fault Number	Fault
*23	***Z before close of program.
24	Line too long.
25	Badly formed number (including lone decimal point).
26	Occurrence of overflow, e.g. caused by "X=10&200"
27	Loop and Repeat instructions not mated.
28	Line out of position, e.g. "ROUTINE 3" inside a Chapter.
29	Incorrect E.D.S. directive e.g. "USE AUX" without a corresponding AUXILIARY directive, or the use of the auxiliary variable instructions ϕ 6 to ϕ 31 without an AUXILIARY directive.
30	Working space exhausted, i.e. instruction too complicated.
31 (*XMAP)	Magnetic tape block too long on input (XMAM or XMAE), or program requires more than 4095 words of lower storage; e.g. caused by large tables of constants or too many Chapters with high numbered labels.
*32	No PROGRAM or SEGMENTS directive.
*33	No subfile name has been specified in a SEND TO directive, and the SEGMENTS directive has been read; or, no file name has been specified in the first READ FROM directive.
*34	An attempt has been made to add a subfile to a magnetic tape that does not already hold a composite file in standard form, or to read a non-existent or non-standard subfile from a specified file.
35 to 73	(Not allocated)
*80	Compiler fault, should never occur
*81	***Z before close of program
*82	No PROGRAM or SEGMENTS directive
*83	No file name specified in the first READ FROM directive
*84	Too many files referred to
*85	SEND TO or DUMP ON file used by any other directive.
*86	No file name or subfile name in SEND TO directive, or no SEND TO directive

* These faults cause the compilation to cease completely, and no more faults will be found.

Comment Number	Comment
101	Single-length expression rounded-off to nearest integer.
102	Double-length expression rounded-off to nearest integer.
103	Double-length expression rounded-off to single length.
104	Integral part of single-length expression taken.
105	Integral part of double-length expression taken.
106	Subfile added at end of composite file because specified subfile has not been found.
107	Expression in double-length brackets not taken as double-length.
108	Additional zero word added to integer or labels table with odd number of entries.
109	****T is not followed by a NL character on paper tape or a blank card.

The appearance of a comment number does not prevent a loadable program from being produced.

RUN TIME MONITORING 4.8

General 4.8.1

Many faults cannot be found until the program has been entered. Typical run-time faults are:

Division by zero

Jumpdown Depth exceeded

When a fault is detected at run-time there is an automatic post-mortem printout and the program is suspended. The post mortem will indicate the type of fault and its position, and will list the values of the indices and special variables when the fault occurred.

Not all faults are detected immediately, though an undetected fault will generally lead to a detected fault. Consider the faulty loop:

```
I = 1 (2) 0
Read (AI)
Repeat
```

The fault in the loop instruction would not be detected as such but the program would eventually be suspended for one of several reasons: eventually there would be no more numbers on the data document, or "AI" would refer to part of the computer not reserved for the program, or I would become larger than the allowed index value.

Overflow faults, that is those faults in which a number becomes too large, are not necessarily detected immediately they occur, but will always be found before an input or output instruction, a Chapter change or Routine change, or entry to a function.

These faults are listed in Section 4.8.5. The post-mortem printout is sent to the *first* basic output device specified in the program description. If there is no basic output device allocated, the message ER will be sent to the console typewriter, but no other information will be output. The message NO will be sent to the console typewriter if an output instruction occurs and there is no output device allocated. This situation could occur if an EMA segment was consolidated with a PLAN program as master. The programmer should ensure, therefore, that each Chapter in a program will have a basic output device allocated.

Label 100 4.8.2

Some faults can be dealt with without suspension of the program. If one of these faults occurs in a Chapter that has a label 100) set, there is an automatic jump to the instruction with this label and T' is set equal to a fault number. The programmer might have at label 100) a sequence to take appropriate action and continue the program, or he might simply include a post mortem sequence to print out, say, the main variables and end the program.

There are five such faults:

Fault Number	*Fault*
33	Square root of negative argument.
34	Number too large.
35	Logarithm of negative argument.
36	Division fault, usually division by zero.
37	Zero pivot in ϕ28, i.e. division by a singular matrix.

There is no automatic post mortem or other external indication of the fault when a jump to label 100) is caused.

As noted in Section 4.10.4 fault 34 may not be detected immediately it occurs.

Example

A loop is executed 10000 times. One of the instructions in the loop puts Y equal to $(\sqrt{X} + \sqrt{X - 1})$. If either X or X - 1 is negative, Y is to be put equal to 1. This is not expected to occur more than two or three times. If any other detectable fault occurs in the Chapter an error indication is to be printed and the program ended.

3146(4.68)

```
      CHAPTER 2
      - - -
      N = 1 (1)10000
      - - -
      Y = φsqrt (X) + φsqrt (X - 1)          | To 100) if faulted.
      - - -
    4) - - -                                 | To here after fault 33.
      Repeat
      - - -
  100) Jump 101, T' ≠ 33                     | To here if fault.
      Y = 1                                  | Continue if
      Jump 4                                 | fault 33.
  101) Printline                             | To here if
      Wrong data. Fault no.                  | not fault 33.
      Print (T') 1,0
      Print ('N =')
      Print (N) 1,0
      End
      - - -
      CLOSE
```

A fault within a Routine can cause a jump to label 100) of the containing Chapter but not to label 100) in the Routine. In this case, or if the fault is in the ordinary type of Jumpdown sequence, it is up to the programmer to ensure that the program continues at the right Jumpdown level. The Test Level instruction can be used to establish the correct level and the instructions Up Level and Clear Level used to correct it if necessary (Section 2.1.4).

Example

In the following sequence, it is assumed that a fault may occur either in the main body of the loop or in the Routine. In either case the program is to continue with the next cycle after an error indication has been printed. The jump to label 100) may take place at Jumpdown level 1 or 2 and corrective action may therefore be needed to ensure that the Repeat is obeyed at level 1.

```
      CHAPTER 3
    6) - - -
      Jumpdown 13                            | Lowers to JD level 1.
      - - -
   13) Loop, X = FZ φinc(φsin(Z)) Q/Y
      - - -
      Jumpdown (R 50)                        | Lowers to JD level 2.
      - - -
   14) Repeat                                | To here if fault found.
      - - -
      - - -
  100) Printline
      FAULT IN THIRD PHASE. FAULT
      Print (T') 1,0
      Print ('X =')
      Print (X) 0,7
      Test Level (T')                        | Finds JD level.
      Jump 14, T' = 1                        | Continue if 1.
      Up Level                               | Reduce JD level
      Jump 14                                | to 1 if currently 2
      CLOSE
```

The Fault Instruction 4.8.3

Programmers can define private fault numbers and specify them in instructions such as:

```
Fault 1300
Fault (N + 1)
```

A Fault instruction puts T' equal to the specified number and transfers control to label 100) of the current Chapter, if it exists. If there is no label 100) an automatic post-mortem occurs and the program is suspended.

Example

In a particular program it is an error for both X and Y to be negative at the start of Chapter 3. The program tests for this error and gives it the number 1201 when it occurs:

```
   CHAPTER 3
5) Jump 3, X ⩾ 0
   Jump 3, Y ⩾ 0
   Fault 1201                    | To label 100) if error found.
3) - - -
   - - -
   CLOSE
```

It is recommended that private fault numbers be greater than 1000 to avoid any chance of confusion with compiler fault numbers.

Label 0 4.8.4

Label 0 in a Chapter has a special use similar to that of label 100 (Section 4.8.2). If either of the instructions Read or ϕ10 finds a *spurious character* in or before a number there is an automatic jump to label 0 of the current Chapter. If there is no label 0 the fault is monitored in the usual way. After a jump to label 0 the first input instruction to read from the document containing the fault should be a Read Ch instruction. This will obtain the spurious character and reading can then continue normally. (A Read Ch instruction is not needed if the ϕTape character input system is being used; see Section 5.2.)

Example

A program contains a loop that is executed 1000 times. One number is read on each cycle. If this number contains a spurious character the rest of the line is omitted and the loop is continued from the next cycle. A count of the number of faulty lines is kept in T.

```
   T = 0
       N = 1 (1) 1000
       Read (X)
       - - -
     7) Repeat
   - - -
0) Read Ch (I)                   | Read and ignore spurious
   Jump 0,I ≠ φcode (NL)         | character and rest of line.
   T = T + 1                     | Add 1 to faulty-line count.
   Jump 7                        | Repeat loop.
```

The normal action of the Read instruction is a character-by-character scan of the selected input document. Before each line is scanned it is reconstructed and edited, so that tabs will have been replaced by two or more spaces and erases will have been eliminated. There are then two stages to the reading of a number. Firstly the document is scanned for the start of a number. Secondly the number is scanned for a terminator. Certain characters are ignored in the first stage. If a character is found that is not one of these, and is not a legal starting character for a number, then it is regarded as a spurious character and scanning stops. In the second stage, if a character is found that is not allowed in a number, or is wrongly positioned, then it is regarded as a spurious character and scanning stops.

Characters that are ignored in the first stage are:

Space Newline ' →

(Arrow, which may appear only on five-track paper tape, is ignored for compatibility with Mercury Autocode, which treated it as an end-of-tape marker.) Legal starting characters for a number are:

\+ - . Digit & , E

"&" and "," are allowed because numbers such as "&3" and ",2" are taken to be abbreviated forms of "1&3" and "1,2" respectively. "E" is a legal starting character for all media except 5-track input. Characters allowed in a number are:

3146(10.69)

+	-	.	Digit	&	,	Single space

In addition any number of spaces may occur immediately after "&" or ",". Legal terminators are:

Double space	Newline	'

A spurious character will usually be a printable character but there are some exceptions. On eight-track paper tape there are various non-printable characters; these should not occur but if they do, they will be treated as spurious characters.

On five-track tape, there can by any number of superfluous Figure Shift characters but a Letter Shift will cause a spurious character to be found. Normally this will be a character in letter shift, such as "B" or "?". When the Letter Shift is non-significant and no character in letter shift is printed, for instance if

FS 1 2 LS LS FS 3

appears on the tape and

123

on the printout, then the spurious character obtained by Read Ch would be the next EMA character, in this case "3".

If a number is to be terminated by a newline consisting of a string of five-track Carriage Return and Line Feed characters, the string must start with a Carriage Return. If a Line Feed not preceded by a Carriage Return is found after the start of a number it is faulted; after a jump to label 0 the normal contraction scheme for Carriage Return/Line Feed strings asserts itself and a Read Ch instruction would obtain newline as the spurious character.

If a legal terminator is registered as spurious the reason is probably that an isolated starting character has been read, e.g. the sequence

```
+   987
```

would be read as far as the second space after "+"; then there would be a jump to label 0 and Read Ch would obtain this space. If a character that is legal in a number registers as spurious the probable reason is that it is wrongly positioned. Consider, for instance

```
98 + 7
17 & 3 & 4
```

The "+" and the second "&" would both register as spurious.

A spurious character found by a Read instruction in a Routine will cause a jump to label 0 of the Chapter, not of the Routine. Similar precautions to those mentioned in Section 4.5.4 should be taken.

A ϕ10 instruction is equivalent to a series of Read instructions.

When a Read or ϕ10 instruction is unsuccessful the value(s) of the appropriate index or variable(s) may be changed, but not to anything useful.

One common use of a label 0 sequence is to detect a warning character at the end of a list of numbers of unknown length.

Example

A list of numbers is to be read to G0, G1... There will be at most one thousand numbers in the list and the last one will be followed by "F".

```
    CHAPTER 0
    G → 1000                          | 1001 variables
    - - -
    - - -
    J = 0 (1) 1000
    Read (GJ)
    Repeat
    - - -                             | Position reached only
    - - -                             | if > 1000 numbers.
3)  - - -
    - - -
0)  Read Ch (I)                       | Read spurious character.
```

```
      Jump 5, I = φ code (F)        | Check that it is F.
      Printline                     | If not, it is a genuine
      DATA ERROR. WRONG CHAR =      | error and program is
      Print Ch (I)                  | stopped.
      Print (J) 3,0
      End
   5) J = J - 1                     | J to indicate suffix of
      Jump 3                        | last number read
      - - -
      CLOSE
```

The last number must be properly terminated before the warning character is printed.

This technique should be used only with visible warning characters. Notice that the character "↦" in five-track paper tape cannot be used as a warning character because it is ignored before a number.

Run-time Fault Numbers 4.8.5

Fault Number	*Fault*
1 to 6	(Not allocated).
7	Int Step initial conditions incorrect, includes step length = 0.
8	Negative dimensions of matrix.
9	Illegal overlapping of matrices.
10	Limits of auxiliary variables exceeded by matrix instructions, includes check on L in ϕ25 to ϕ28.
11	(Not allocated)
12	Jump out of control.
13	Across or down to unset label.
14	Looped VARIABLES directives, e.g. VARIABLES 1 in Chapter 2 and VARIABLES 2 in Chapter 1.
15 to 31	(Not allocated)
32	Spurious character found by a Read or ϕ10 instruction. (error number only output if label 0 not set in current Chapter)
*33	Square root of negative number.
*34	Overflow, i.e. number too large (includes exponential or large argument)
*35	Logarith of zero or negative number.
*36	Division by zero.
*37	Zero pivot in ϕ28, i.e. division by a singular matrix.
38 to 39	(Not allocated)
40	Incorrect ϕpoly parameters.
41	Incorrect ϕmin or ϕmax parameters.
42	Sin or Cos of large argument.
43	Arcsin or arccos of x outside range $-1 \leqslant x \leqslant 1$.
44	Arctan with $x = y = 0$.
45	Incorrect operand for ϕRandom.
46	Invalid use of magnetic tape instructions.
47	Undefined tape number, i.e. magnetic tape instruction uses a tape number that has not been defined by a peripheral description directive.
48	(Not allocated)

Fault Number	Fault
49	Magnetic tape subfile is missing on Input.
50	Record too big for block on Output.
51	***Z at end of data document read, or a tape mark read from magnetic tape.
52	A block longer than 128 words, or that specified in a peripheral directive, has been read from a magnetic tape.
53	DEPTH, JUMPDOWN DEPTH or DUMPS limit exceeded.
54	Attempt to Restore without having first Preserved (checked only when MT Dumps are specified).
55	Too many Up or Return instructions.
56 to 58	(Not allocated)
59	Non-existent input document number.
60	Across or Down to non-existent Chapter.
61	When using a file with the USE AUX = ED ($F(g)$) directive, the new minimum auxiliary variable is less than the current one on the file.
62	If using a file with the CREATE directive and the first auxiliary variable instruction is other than 7 or 10.
63	The System Control Area is full.
64	The size of the bucket being read is less than 20 or the bucket is too big for the buffer.
65	With the INPUT directive, a subfile description mentioned in the statement is missing; with the OUTPUT directive, one of the following conditions has occurred: 1 No subfile mentioned in the directive (for simple files the CREATE directive should be used). 2 The file specified does not have a subfile structure. 3 A subfile, other than the last subfile in the directive, is not a directory subfile. 4 The last subfile in the directive is not a data subfile.
66	Logical bucket number of file out of range or zero, that is program corrupted or, in the case of input, trying to read data that does not exist.
67	Purge date of file not exceeded or failure of integrity code check.
68	(Not allocated)
69	Program looped or went illegal and the operator initiated fault action.

* These fault numbers are only output if label 100 is not set in current Chapter.

PROGRAM DEVELOPMENT 4.9

General 4.9.1

A program is likely to require several runs before it is free of faults. On the first run it may well not be entered at all because of format errors. When these are corrected there are likely to be logical errors. Some of these will be monitored. Others will merely produce wrong results.

Accordingly there should be a clear distinction between development and production runs. Development runs should be arranged to produce results that can be cross-checked by another method and the programmer should establish that all branches of the program are working satisfactorily. Ideally all combinations of branches should be tested but this is seldom possible in practice. It is advisable to print out intermediate results that will not be required once the program is working properly, as these will help locate errors. For large programs an effort should be made to keep all Chapters and Routines as self-contained as possible, so that they can be tested separately from the rest of the program.

During the development stage an estimate is made of how much computer store and time will be required by the developed program.

This Section introduces facilities particularly useful in the development of a program.

Query Printing 4.9.2

The easiest way of obtaining intermediate results is to insert a question mark before or after an arithmetic instruction; e.g.

```
X = Y - 1 ?
? N = N + 2
```

Then the new value of the index or variable is automatically printed out. This is known as *query printing*. It can also be applied to loop instructions. Consider the sequence

```
X = 0
N = 11 (1) 100?
X = X + N ?
Repeat
```

This could print

```
11
1.1000000000&1
12
2.3000000000&1
13
3.6000000000&1
-
-
-
```

Note that if output is to 5 track paper tape, the ampersand sign is replaced by a comma.

Values are printed as if by

```
Print (N) 1, 0
Print (X) 0, 10
Print ((X, Y)) 0, 21
```

depending on the form of the left-hand side of the instruction. Each number will be printed on a new line.

The great advantage of query printing is that it can easily be suppressed in whole or in part once a program is working. The directive

```
IGNORE  QUERIES
```

causes queries on following lines to be ignored. The directive

```
COMPILE  QUERIES
```

indicates that any following queries are not to be ignored. The two directives can appear any number of times in a program and are used when some parts of a program are fully tested and others are not.

3146(10.69)

They can appear anywhere, both inside and outside a Chapter. Since they are directives, they take effect during translation and affect only sections of program that appear after them in written order. In the absence of either, it is as though

```
COMPILE QUERIES
```

appeared at the head of a program.

A typical program could be:

```
CHAPTER 1
- - -
CLOSE

IGNORE QUERIES
CHAPTER 2
- - -
CLOSE

COMPILE QUERIES
CHAPTER 0
- - -
CLOSE
```

Chapter 2 is fully developed but query printing is still required for Chapters 1 and 0.

A Routine is translated immediately it is encountered by the compiler. It follows that compilation of query printing in a Routine is controlled by the last COMPILE or IGNORE QUERIES directive in written order before the Routine, or by any such directives within the Routine.

Query printing is effected by special Print instructions inserted in the compiled program under the control of the COMPILE/IGNORE QUERIES directives. Once a program has been compiled the number of these instructions is fixed. However, it is possible to inhibit them, i.e. to switch off query printing, by the instruction

```
Stop Queries
```

This causes a jump round all query print instructions that would otherwise be obeyed. The reverse effect can be obtained by

```
Start Queries
```

On initial entry to a program query printing is automatically switched on.

The effect of these two instructions can be illustrated by the following sequence:

```
X = Y?
N = 1 (1) 1000?
X = X + AN?
Repeat
Z = 3XY - 2FG ?
```

Assuming that all queries were compiled this would cause 2002 numbers to be printed. The programmer is unlikely to need so much information. The amount could be reduced by inserting directives as follows:

```
X = Y?
IGNORE QUERIES
N = 1 (1) 1000?
X = X + AN?
Repeat
COMPILE QUERIES
Z = 3XY - 2FG?
```

Only two numbers would then be printed and there would be no record of values generated in the loop. A satisfactory compromise might be to compile all queries but to allow query printing to take place only during the first and last five cycles of the loop:

```
X = Y?
N = 1 (1) 1000?
X = X + AN?
Jump 3, N ≠ 5          | Switch off query
Stop Queries           | printing after 5 cycles.
Jump 4
3) Jump 4, N ≠ 995     | Switch on query
   Start Queries       | printing for last 5 cycles.
4) Repeat
   Z = 3XY - 2FG?
```

This would cause a total of twenty-two numbers to be printed out.

Once the above sequence was working properly the simplest way to eliminate the printing would be to write IGNORE QUERIES before the sequence. It would be neater if the instructions that switch the printing on and off were also eliminated but this is not essential.

It is easy to become confused over the different methods of controlling query printing. IGNORE and COMPILE QUERIES are directives. They control what query printing is actually compiled but are not themselves translated. They affect parts of the program appearing later in written order. Stop and Start Queries are instructions. They are translated into machine-code instructions which are obeyed when the program has been entered. They switch on and off any query printing that has been compiled into the program.

Queries should be eliminated from a developed program by IGNORE QUERIES directives since they increase the store and time required, even if query printing is switched off.

The Start/Stop Queries instructions may also be performed by the machine operator. The instruction

```
Stop Queries
```

is equivalent to the console message

```
ON #name 19
```

and

```
Start Queries
```

to:

```
OFF #name19
```

Trace 4.9.3

There is a similar system to query printing called *trace*. The directives

```
COMPILE TRACE
```

and

IGNORE TRACE

control the insertion of special trace instructions in a program. These instructions output information that allows the path followed by a program to be traced. Their main function is to print out the label number when a labelled instruction is obeyed. Consider the sequence

```
COMPILE TRACE
- - -
18) I = 3
- - -
 9) Jump 7, I ≠ 3
 8) J = 17
    Jump 7
    - - -
 7) I = 1
    Jump 9
    - - -
```

This would print

L18 L9 L8 L7 L9 L7 L9 L7.....

Chapter changes are also traced.

Trace directives can appear anywhere in a program. Default action at the head of a program is

IGNORE TRACE

Compiled trace printing can be controlled in a similar way to query printing by the instructions

Stop Trace

and

Start Trace

On initial entry to a program, trace printing is automatically switched on but printing occurs only if COMPILE TRACE has appeared. The remarks on the difference between query directives and instructions that appear at the end of Section 4.9.2 apply also to trace directives and instructions.

The Start/Stop Trace instructions may also be performed by the machine operator. The instruction

Stop Trace

is equivalent to the console message

ON *#name* 0

and

Start Trace

to:

OFF *#name* 0

The Fixed Format Print Instruction 4.9.4

When a Chapter or Routine is tested separately it will probably be accompanied by a steering Chapter. This will contain little but Down (or Jumpdown) instructions and Print instructions. The ordinary Print instruction can be used but since the format of the test results is unlikely to be important the most convenient instruction may be that typified by

Print (X)

where no printing style is specified. The number would be printed out as though by

```
Print ( ) 1, 0
Print ( ) 0, 10
Print ( ) 0, 21
```

depending on whether the expression is an integer, single-length or double-length expression. Each number will be printed on a new line.

Though it is chiefly intended for program testing, there is no reason why this instruction should not be used as a normal part of a program.

OVERLAYS 4.10

This section describes the overlay system which may be used in 1900 EMA programs, using magnetic tape as a backing store with XMAM, or E.D.S. with XMAE.

Programs may be written which are too large to be held in the core store of the computer. Main and auxiliary variables may be held on a backing store such as E.D.S. and instructions to do this have already been described. Program instructions may also be held on magnetic tape or E.D.S. and such a program is run by overlaying parts of the program from that backing store. That is, only part of the program is in the core store at a time, the remainder is held on the backing store, and copied into the store whenever it is required.

The Overlay Directive 4.10.1

The programmer will divide the program into groups of segments, called units, one unit of which is not to be overlaid, and a number of *overlay units*. The section of program which is not to be overlaid is called the permanent unit, and is kept in the permanent area of core store. It contains Chapter 0 and any other Chapters and Routines that the programmer chooses. The permanent unit will also contain various subroutines that are inserted into the object program by the compiler and not referred to explicitly by the programmer (for example, the input and output subroutines). Each overlay unit of the program must be assigned to an *overlay area* of core store. There may be several areas and each one will have two or more overlay units assigned to it. At any one time the overlay area can contain one only of the units assigned to it. The storage assigned to an overlay area will be that required by the largest unit within each area. The organization of Chapters and Routines into units and areas is specified by OVERLAY directives in the program description. The form of the directive is as follows:

OVERLAY (a,u) $C_1,C_2,C_3,\ldots C_n$

where: a is the overlay area number, $1 \leqslant a \leqslant 255$

u is the overlay unit number, $1 \leqslant u \leqslant 1023$

$C_1,C_2,\ldots C_n$ is a list of Chapters and Routines to be included in unit u of area a

The numbers used for a and u need not be consecutive. Each OVERLAY Directive will occupy one line, but several directives may refer to the same unit and area.

Example

```
OVERLAY (3, 7) C1, C2, C3, R1, R2, R9
OVERLAY (3, 7) R3
OVERLAY (3, 8) C1-1, C2-1, C3-1
```

Chapters 1, 2, 3, Routines 1, 2, 3, 9 are assigned to unit 7 of area 3, and Chapters 1, 2, 3 of Programme -1 are assigned to unit 8 of area 3.

All other Chapters and Routines that have not been assigned to an overlay unit will be assigned to the permanent unit. This unit is not specified explicitly by the programmer.

Example

The following is a program description of an overlay program BUZZ

```
PROGRAM (BUZZ)
OVERLAY (1, 1) C1, C2, C3, R1, R2
OVERLAY (1, 2) C4, R7, R8
OVERLAY (1, 3) C1-26, C2-26
OUTPUT 1 = LP0
OUTPUT 2 = MT4 (BUZZWORKTAPE)
```

```
INPUT 1 = TR0
COMPILE TRACE
MAIN 4000
```

The Overlay Program 4.10.2

1900 EMA does not require the user to call any special overlay routines. The source program is written in the normal way, and Chapters are called by Down instructions, exactly as if the whole program was in core store. If the Chapter called is in an overlay unit, the system checks to see whether that unit is already in the core store, and if it is not, it will be copied from the backing store, then entered normally. An Up instruction in this Chapter will have a similar effect, checking to see whether the segment containing the Down instruction is in the core store, and copying it in if it is not. Decisions on overlay organization are not, therefore, required at the time of writing a source program. This organization need not be defined until the program has been tested and is to be consolidated for the last time.

Restrictions on Overlay Programs 4.10.3

There are two restrictions that must be observed when writing a source program that is to be overlaid:

1 The VARIABLES directive must always refer to a Chapter in store, either in the permanent area, or in the same overlay unit.

2 Routines can be overlaid provided that all Chapters that call them directly or Routines that call them directly or indirectly are in the same overlay unit.

Restriction 2 means that only Down or Across instructions can be used to pass control from one unit to another.

Compilation and Consolidation 4.10.4

The structure of an overlay program must be defined each time the program is compiled and consolidated, whether or not the structure has changed. If the structure is to be different from the previous compilation it is possible to reconsolidate by providing a new program description and re-inputting the consolidated semi-compiled program from the file produced by the previous run. If amendments to the source program are made, then those segments affected must obviously be re-compiled, but the remaining segments may still be obtained by re-inputting the old semi-compiled file.

MAGNETIC TAPE COMPILATION AND CONSOLIDATION

There is no need for the programmer to present EMA source Chapters to the compiler XMAM in any particular order. The compiler writes consolidated semi-compiled output either to a tape allocated as MT2 or to the file named in a SEND TO directive, using either a scratch tape allocated as MT1 or the file named in a DUMP ON directive for intermediate working. The compiler then makes a *binary dump* of the overlay program on MT1 from MT2, sorting the Chapters into the required order.

A program in the form of a binary dump is loaded directly into core store without requiring a loader program.

It is not possible to re-consolidate a binary program, so the semi-compiled program must be used for this. The MT2 tape will be released immediately the binary dump has been made, because this tape could be a simple file, named PROGRAM$_{\nabla}$*name*. The binary dump tape will be given this file name also. The MT2 tape could contain the semi-compiled program in a subfile, and this will have been specified by the SEND TO directive. The MT1 tape will still be called PROGRAM$_{\nabla}$*name* . If there is any likelihood that the program will require amending, both tapes should be preserved.

The time taken by the sort can be eliminated if Chapters are presented in the correct order. This order is as follows:

Areas are in order of increasing area number, with the permanent area last. Within each area, overlay units are in order of increasing unit number. The order of Chapters and Routines within a unit is immaterial.

E.D.S. COMPILATION AND CONSOLIDATION

There is no need for the programmer to present EMA source Chapters to the compiler XMAE in any particular order. The compiler writes consolidated semi-compiled output to the E.D.S. file specified in the SEND TO directive.

The consolidator program, XPCL will be loaded by XMAE, and XPCL will consolidate the semi-compiled output, making the binary dump to the E.D.S. file specified in the DUMP ON directive. The DUMP ON directive must always be used when compiling an overlay program on E.D.S.

Chapters will be sorted into the correct order, and maximum efficiency will be obtained from XPCL if the files specified in the SEND TO and DUMP ON directives are on different cartridges, though the files can be on the same cartridge.

Run-time Efficiency 4.10.5

During the running of program overlaid from magnetic tape, an overlay unit is obtained from the tape, and the tape remains in position after that unit. The magnetic tape overlay system will work most efficiently if the next Chapter required is in the overlay unit that follows immediately on the tape.

The most efficient utilization of available core store will be obtained if each overlay unit allocated to a particular overlay area is approximately the same length.

MIXED LANGUAGE PROGRAMS 4.11

This section deals with programs written in more than one source language. It is assumed that the programmer will be conversant with the other languages and no explanation will be given of parts of programs written in other languages. For detailed descriptions, the relevant manuals should be consulted.

Program Consolidation 4.11.1

Section 4.4.1 gave an outline of the steps involved in the compilation and consolidation of an EMA program. When the source program has been translated into semi-compiled segments, the segments are consolidated and can then be loaded into the machine core store. The consolidator and loader are either part of the compiler or are separate programs. This system allows the segments to be compiled initially from any source language. However the segments must be consolidated by the compiler appropriate to the master segment, or more precisely, the segment containing the entry point 0. This entry point is the first instruction of the control segment of an EMA program, the first executable statement of the MASTER segment of a FORTRAN program, etc. The only other restriction is that the segments must be compatible, and this section deals with writing segments compatible with EMA in other languages.

Communication Between Segments 4.11.2

Main variables in EMA are defined in each Chapter as explained in Section 1.5.2. This area of core store can be regarded as a *common* area. The Main directive reserves a section of core store for variables, and each Chapter must first define a set of names which that Chapter uses to refer to the store locations. These names will eventually be replaced by the addresses of the locations when the program is finally entered into core store. Thus this area of core store is available to all segments of a program, even though the actual names of the variables are not. Therefore, segments written in other languages requiring numerical values from EMA segments or passing numerical values on to segments written in EMA must specify this common area. If the special variables and indices and auxiliary variables are required their appropriate common areas must also be specified. The names given to EMA common areas, and which must be specified by segments in other languages, are as follows:

1 EIVS - A LOWER area containing the indices and special variables. The indices occupy one word each and the variables two words. They are stored in the following order – I′ to T′, I to T, A′ to H′, U′ to Z′, A to H, U to Z, π, π'. (Total 84 words)

2 EMAI - An UPPER area containing the main variables and index stores.

3 EAUX - An UPPER area containing the auxiliary variables, provided they have been specified in core store and not on a backing store.

The address of auxiliary variable 0 is held in the 85th word of the EIVS area.

Examples

1 An EMA program contains the directive:

A→99

A PLAN segment requires the following coding to refer to these variables.

#UPPER COMMON/EMAI/

AA(200)

Since EMA variables are stored as floating point numbers, 200 words are required for the 100 EMA variables. Therefore A0 is equivalent to AA and AA+1, A1 to AA+2 and AA+3, A73 to AA+146 and AA+147, etc.

2 An EMA program contains the directive:

C→199

A FORTRAN segment requires the following coding to refer to these variables:

COMMON/EMAI/CC(200)

Therefore C0 is equivalent to CC(1), C1 to CC(2), C139 to CC(140), etc. Variable quantities in FORTRAN are stored in exactly the same way as in EMA.

3 A PLAN program requires to address the special variables and indices. When defining the PLAN COMMON area, the different method of storage of EMA indices and variables is taken into account. The COMMON area would be defined as follows:

#LOWER COMMON/EIVS/

I1(12),I(12),AA(16),UU(12),A(16),U(12),PI(2), PPI(2)

Therefore T' is equivalent to II+11, K to I+2, U to U and U+1, π' to PPI and PPI+1.

The equivalent statement in FORTRAN is:

COMMON/EIVS/II(12),I(12),AA(8),UU(6),A(8),U(6),PI,PPI

In order to assign the required one word of storage to the FORTRAN INTEGER variables (which are declared implicitly in the above statement) the FORTRAN segment must be compiled in COMPRESS INTEGER AND LOGICAL mode. If however, the other language being used has no facilities for using named common areas, an intermediate segment must be written in a language which does have such facilities.

EMA Calls to Other Languages 4.11.3

A segment that is called by EMA using a Down or Jumpdown must appear to be an EMA Routine or Chapter, irrespective of the source language used for the segment.

Within an EMA program Chapters and Routines are referred to by numbers. Semi-compiled segments, however, are referred to by alphanumeric names, that is a string of letters and numbers starting with a letter and containing up to eleven characters. The names given to semi-compiled EMA segments are as follows:

For a Routine k, the corresponding name is EROUTk

For a Chapter m of Programme 0, the corresponding name is ECm

For a Chapter m of Programme $-n$, the corresponding name is ECmPn

EMA imposes the limits:

$0 \leq k \leq 4095$, $0 \leq m \leq 4095$, $1 \leq n \leq 4095$

Examples: Routine 732 of an EMA program is written by another programmer in a different language. This could be a PLAN segment introduced by the directive:

#PROGRAM /EROUT732

Routine 732 could also be a FORTRAN segment written as follows:

SUBROUTINE EROUT732(L)

3146(10.69)

```
      COMMON etc.
      .....
      .....
      RETURN
      END
```

This is entered at the first executable statement by the EMA instruction:

```
      Jumpdown (R732)
```

The RETURN statement passes control back to the next instruction after the Jumpdown. The reason for the argument (L) is explained later in this section.

In EMA it is possible to start executing a segment at a specified label, label numbers running from 0 to 127. If this facility is to be used to enter segments written in other languages, the labels must be programmed specifically. The reason for this is explained on page 70.

For FORTRAN and similar languages, the label *number* is passed over as an integer argument. This number will bear no relationship to the numbers used for FORTRAN labels, and therefore the programmer must write a label table so that control can be passed to the correct instruction. For FORTRAN, this table can take the form of a Computed GO TO statement.

Example

The segment called by the following instruction: Down 1/20 – 3 is written in FORTRAN as follows, where 1 could be 1,2, or 3.

```
      SUBROUTINE EC20P3(L)
      INTEGER L
      COMMON etc
      GO TO (1,2,3), L
1     .....
      .....
2     .....
      .....
3     .....
      .....
      RETURN
      END
```

The value of 1 is passed over to L.

Chapter 20 of Programme – 3 could also be written in PLAN as follows:

```
#PROGRAM    EC20P3
#UPPER      COMMON/EMAI/
            list of variable names etc.
#LOWER
BRANC▽SUSWT 2HRL          [suspend program, wrong label
      BRN   ONE
      BRN   TWO
      BRN   THREE
#PROGRAM
      STO 1 LINK          [store link
```

```
        OBEY   0(1)          [get address of label
        LDX 3  0(3)          [get label number
        OBEY   BRANC(3)      [branch to required instruction
ONE     .....
        .....                [program
        .....
TWO     .....
        .....
THREE   .....
        .....
UP      LDX 1  LINK          [restore link
        EXIT 1 1             [equivalent to Up
#END
```

In the label table under #LOWER, the label 0 is not used and has been given an address that would cause the program to be suspended. The programmer could also arrange an error print out and a fixed halt at this point. A label number greater than 3 would have an indeterminate result. In the FORTRAN program a label number out of range would also cause an indeterminate result. It is obvious that the easiest way or arranging this label table in PLAN is to start from label 0, and use consecutive label numbers. In FORTRAN, the use of the computed GO TO is the easiest solution, and the labels should be consecutive, starting from 1. Also in the FORTRAN program it is immaterial how many words are assigned to the INTEGER argument, since only the first will be used anyway.

Whether or not a Jumpdown or a Down instruction specifies a label, one argument *must* be allowed for, even though it is not used. In other words, the FORTRAN subroutine *must* be written with one argument, and the PLAN segment *must* specify one argument in the EXIT instruction.

This label could otherwise be used as an integer argument in the range 0 to 127, it it is not needed as a label.

The labels 0 and 100 are only significant within an EMA Chapter, and are not used as an error trap outside the Chapter.

Calls to EMA Chapters from Other Languages 4.11.4

The instruction Down 1/m-n in EMA is equivalent to

```
        CALL ECmPn(l)
```

in FORTRAN, and

```
        CALL   1  ECmPn
        LDN    3  address of word containing 1
```

in PLAN.

EMA Routines cannot be called directly by other languages. A short intermediate Chapter should be written to call the Routine.

Other languages such as Algol and COBOL do not use common areas of store, and an intermediate segment must be used. The relevant manuals should be consulted for details of communication.

Programming Peripherals 4.11.5

Peripheral transfers are arranged differently in each language, and the coding produced by a compiler for a high-level language can be quite complex. Serious interference could occur between the two sets of coding if the programmer tries to program the same peripheral unit in two different languages. Therefore as a general rule, a peripheral should be programmed exclusively in one language.

There is of course nothing to prevent the programmer releasing the unit in one language, calling a segment in another language which then re-engages the same peripheral. It should be remembered that an EMA program will normally require one slow output device allocated in each Chapter.

3146(4.68)

Overlay Programs 4.11.6

As with peripheral transfers different languages have different overlay systems. Once a program description has defined an overlay structure, implicit calls to the overlay package are set up at certain points in a program during compilation. Segments compiled from other languages will not have the correct calls set up, and the overlay system may fail to work when these segments are incorporated into a program. For mixed language overlay programs to work correctly, the following rules should be observed:

1 The overlay structure must be defined once in the master language, in this case EMA. All Chapters and Routines should be included, irrespective of the source language used.

2 The overlay structure must not be defined in terms of any other language used.

3 Segments not compiled from EMA can only call and exit to other segments that are present in core store at the time the call or exit is executed.

The last restriction means in effect that the next overlay unit can only be brought down by an EMA Down or Across instruction, and a Down or Across instruction is also required to enter another overlay area, if the unit required is not present in core store.

Error Diagnostics 4.11.7

The error diagnostic packages included in a program will be those corresponding to each source language used. Thus label 100 is meaningless outside an EMA chapter, the FORTRAN TRACE 2 package is similar to both EMA TRACE and EMA query printing but is not equivalent; error numbers have a different significance. For instance if overflow was set before a Down instruction to a FORTRAN segment was executed this fact would be reported in the EMA segment and appropriate action taken. Overflow is tested before every Chapter or Routine change and on entry to any library subroutine, e.g. SIN, READ, PRINTCH. If overflow was again set in the FORTRAN segment the error would be flagged when the offending statement has been executed (incorrectly) and not at the RETURN to the EMA chapter.

When a program is being developed a programmer should arrange for the highest level of monitoring possible. It would be best if the programmer could arrange to test separately each set of segments in each language, possibly by inserting 'null' segments to prevent the program going illegal when segments in other languages are called.

Example

An EMA program requires the beta and gamma functions evaluated at certain points. Subroutines are available in the FSCE group on the Scientific Master Library Tape to evaluate these functions.

These subroutines are written in FORTRAN and use parametric communication. Therefore a short segment must be written so that the EMA special variables can be used as function arguments. To make compilation of EMA programs simple the calling segment, the subroutines needed, and any FORTRAN library subroutines called by the subroutines, are compiled by a FORTRAN compiler, and the semi-compiled segments produced are inserted in an EMA library using XPMU or XPEU.

The following statements are input to the FORTRAN magnetic tape compiler XFAM:

```
      SEND TO (MT,EMAWORKTAPE(16).ADD▽LIB(1))
      PROGRAM (XXXX1026-JEP)
      TRACE 0
      END
      SUBROUTINE EC007(JAMES)
      COMMON/EIVS/XJUNK(35),V,W,X,Y
      GO TO (1,2), JAMES
1     CALL F2BETA(X,Y,V)
      RETURN
2     CALL F2GAMMA(V,X)
      RETURN
```

```
END
LIBRARY
READ FROM (MT,-.FSCE)
FINISH
```

The FSCE group is scanned in LIBRARY mode with a READ FROM statement. The output in subfile ADD$_{\nabla}$LIB(1) of file EMAWORKTAPE7(16) is inserted in either the EMA subroutine group SRE1 (for an XMAM compilation) using XPMU or the EMA library file SUBGROUPSRE1 (for an XMAE compilation) using XPEU. The programmer may then write in the EMA program

```
Down 1/007
```

to evaluate

$$V = \beta\ (x,y) = \int_0^1 t^{x-1}(1-t)^{y-1}dt$$

and `Down 2/007`

to evaluate

$$V = \Gamma\ (x) = \int_0^\infty t^{x-1}e^{-t}dt$$

EXAMPLE LISTINGS 4.12

EXAMPLE OF LISTING WITH ERROR

The following printout was obtained during compilation of a program. Line 52 has been flagged as in error, and fault 18 (incorrectly constructed arithmetic expression) is reported. This error was caused by the punch operator punching an apostrophe instead of a figure 4.

```
COMPILED BY   #XMAM/12     29/05/68    01/10/45

 1-         LIST(LP)                                                        1
 2-         PROGRAM (CMVS)                                                  2
 3-         INPUT 1=TR0                                                     3
 4-         OUTPUT1=LP0                                                     4
 5-         COMPILE TRACE                                                   5
 6-         MAIN 0                                                          6

 7-         ROUTINE 1 !TO PRINT OUT RESULTS                                 7
 8-   1)    JUMPDOWN (R2) !TO OBTAIN RESULTS                                8
 9-         PRINT ('SIZE OF SAMPLE   ')                                     9
10-         PRINT (N)4,0                                                   10
11-         PRINT ('MEAN    ')                                             11
12-         PRINT (Y)6,4                                                   12
13-         PRINT ('VARIANCE    ')                                         13
14-         PRINT (V)4,6                                                   14
15-         PRINT ('S.D.    ')                                             15
16-         PRINT (D)4,6                                                   16
17-         NEWLINE 2                                                      17
18-         RETURN !TO NEXT SET DATA                                       18
19-   2)    JUMPDOWN 1                                                     19
20-         PRINT LINE                                                     20
21-         END OF DATA TAPE                                               21
22-         RETURN                                                         22
23-         **                                                             23

24-         ROUTINE 2 !TO CALCULATE RESULTS                                24
25-         Y=X/N          !CALCULATE MEAN                                 25
26-         V=X'/N-Y*Y     !CALCULATE VARIANCE                             26
27-         D=XSQ RT (V)   !CALCULATE STANDARD DEVIATION                   27
28-         RETURN         !TO PRINT ROUTINE                               28
29-         **                                                             29

30-         CHAPTER 0                                                      30
31-         SELECT INPUT 1                                                 31
32-         SELECT OUTPUT 1                                                32
33-   1)    N=0                                                            33
34-         X=0                                                            34
35-         X'=0                                                           35
36-   2)    READ (A)            !READ ONE NUMBER FROM DATA TAPE            36
37-         N=N+1                                                          37
38-         X=X+A               !ACCUMULATE                                38
39-         X'=X'+A*A           !ACCUMULATE SQUARE                         39
40-         JUMP 2                                                         40
41-   3)    JUMPDOWN (R1/1)     !TO PRINT RESULTS                          41
42-         JUMP 1              !TO NEXT SET OF DATA                       42
43-   4)    JUMPDOWN (R1/2)     !TO PRINT LAST SET OF RESULTS              43
44-         END                                                            44
45-   0)    READ CH (J)         !READ SPURIOUS CHARACTER                   45
46-         JUMP 3,J=%CODE(#)   !END OF SET                                46
47-         JUMP 4,J=%CODE(*)   !END OF TAPE                               47
48-         JUMP 0              !SPURIOUS CHARACTER NOT # OR *             48
49-   100)  PRINT ('ERROR T = ')  !ERROR ROUTINE                           49
50-         PRINT (T')1,0                                                  50
51-         PRINT ('    AT N = ')                                          51
52-         PRINT (N)',0                                                   52
  FAULT     10

53-         JUMP 1              !SKIP TO NEXT SET                          53
54-         CLOSE                                                          54
```

EXAMPLE OF USE OF LABEL 100

The following program reads numbers from paper tape, and calculates mean, variance and standard deviation. Each set of numbers is terminated by #, and the list set is terminated by *. Label 100 is used to account for any error. Numbers out of range (i.e. numbers causing overflow) cause an error message to be sent to the line printer, and the next number is read. The first example of printout is with trace printing. The label of each labelled instruction is printed each time that instruction is executed. The first set of 13 numbers has a larger variance than expected, consequently the rest of the line is displaced to the right. After the second set of 18 numbers, four numbers were read that caused overflow (fault 34). From the Trace list, this fault occurred at label 2 (i.e. READ (A)), so the number that was too large occurred before this i.e. between READ (A) and JUMP2, and therefore at line 39 (X'=X'+A*A). If this number, and not just its square, had been too large, it would have been detected at READ (A) the first time, and only one L2 would have appeared in the trace list, and the error message would have indicated N=0.

3146(10.69)

```
L1  L2  L2  L2  L2  L2  L2  L2  L2  L2  L2  L2  L2  L2  L2  L0  L3
JUMPDOWN R1/1  L1
JUMPDOWN R2  RETURN

SIZE OF SAMPLE      13  MEAN      0.0000 VARIANCE    155400155401.366646  S.D.   394208.263991

RETURN
L1  L2  L2  L2  L2  L2  L2  L2  L2  L2  L2  L2  L2  L2  L2  L2  L2  L2  L2  L2  L0  L3

JUMPDOWN R1/1  L1
JUMPDOWN R2  RETURN

SIZE OF SAMPLE      18  MEAN      0.0000 VARIANCE      31.666667  S.D.        5.627314

RETURN
L1  L2  L2  L100

ERROR T=34       AT N=1
L1  L2  L2  L100

ERROR T=34       AT N=1
L1  L2  L2  L100

ERROR T=34       AT N=1
L1  L2  L2  L100

ERROR T=34       AT N=1
L1  L2  L2  L100
L1  L2  L2  L2  L2  L0  L4

JUMPDOWN R1/2 L2  L1
JUMPDOWN R2  RETURN

SIZE OF SAMPLE       4  MEAN      0.0000 VARIANCE       2.500000  S.D.    1.581139

RETURN
      END OF DATA TAPE
RETURN
```

This program was re-run with the same data tape, but with Trace printing switched off by the operator message:

ON #CMVS 0

The following output would be quite adequate for a production run of the program.

```
SIZE OF SAMPLE      13  MEAN          0.0000  VARIANCE    155400155401.566646  S.D.    394208.263991

SIZE OF SAMPLE      18  MEAN          0.0000  VARIANCE      31.666667  S.D.        5.627314

ERROR T =  34     AT N =  1  ERROR T =  34     AT N =  1  ERROR T =  34     AT N =  1  ERROR T =  34     AT N =  1
SIZE OF SAMPLE       4  MEAN          0.0000  VARIANCE       2.500000  S.D.        1.581139

      END OF DATA TAPE
```

The program was then recompiled without label 100 (i.e. without lines 49 to 53), and with the same data tape input. Fault 34 caused the full error halt printout to be made. The number that caused overflow, X', is shown as a line of asterisks. A was set to 1.0 & 39, at N=1, therefore X' should have been 1.0 & 78. This is outside the range of the machine. This printout is followed by a list of the last 50 Trace entries. These were also output during the printing of previous results.

```
L1   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L0   L3
JUMPDOWN R1/1   L1
JUMPDOWN R2   RETURN

SIZE OF SAMPLE      13  MEAN         0.0000  VARIANCE    155400155401.366646  S.D.    394208.263991


RETURN
L1   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L0   L3

JUMPDOWN R1/1   L1
JUMPDOWN R2   RETURN

SIZE OF SAMPLE      18  MEAN         0.0000  VARIANCE      31.666667  S.D.       5.627314


RETURN
L1   L2   L2


FAULT NO,  34
BETWEEN LABELS    2  AND   3  IN CHAPTER    0


I' =         0
J' =         0
K' =         0
L' =         0
M' =         0
N' =         0
O' =         0
P' =         0
Q' =         0
R' =         0
S' =         0
T' =        14


I  =         0
J  =        19
K  =         0
L  =         0
M  =         0
N  =         1
O  =         0
P  =         0
Q  =         0
R  =         0
S  =         0
T  =         0


A' =  0.00000000000&    0
B' =  0.00000000000&    0
C' =  0.00000000000&    0
D' =  0.00000000000&    0
E' =  0.00000000000&    0
F' =  0.00000000000&    0
G' =  0.00000000000&    0
H' =  0.00000000000&    0


U' =  0.00000000000&    0
V' =  0.00000000000&    0
W' =  0.00000000000&    0
X' =  **********
Y' =  0.00000000000&    0
Z' =  0.00000000000&    0


A  =  1.00000000000&   39
B  =  0.00000000000&    0
C  =  0.00000000000&    0
D  =  3.62731433388&    0
E  =  0.00000000000&    0
F  =  0.00000000000&    0
G  =  0.00000000000&    0
H  =  0.00000000000&    0


U  =  0.00000000000&    0


V  =  3.16666666678&    1
W  =  0.00000000000&    0
X  =  1.00000000000&   39
Y  =  0.00000000000&    0
Z  =  0.00000000000&    0


E  =  3.14159265358&    0
E' =  0.00000000000&    0


L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L0   L3
JUMPDOWN R1/1   L1
JUMPDOWN R2   RETURN
RETURN
L1   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L0   L3

JUMPDOWN R1/1   L1
JUMPDOWN R2   RETURN
RETURN
L1   L2   L2
```

This program was re-run with the same data tape but with Trace printing switched off by ON #CMVS 0. The only Trace printing output is at the end of the error printout. This is substantially the same as in the previous output.

```
SIZE OF SAMPLE      13  MEAN          0.0000  VARIANCE    155400155401.366646  S.D.    394208 263991

SIZE OF SAMPLE      18  MEAN          0.0000  VARIANCE      31.666667  S.D.       5.627314



FAULT NO.  34
BETWEEN LABELS    2  AND   3  IN CHAPTER    0

I' =         0
J' =         0
K' =         0
L' =         0
M' =         0
N' =         0
O' =         0
P' =         0
Q' =         0
R' =         0
S' =         0
T' =        14

I  =         0
J  =        19
K  =         0
L  =         0
M  =         0
N  =         1
O  =         0
P  =         0
Q  =         0
R  =         0
S  =         0
T  =         0

A' =  0.00000000000&    0
B' =  0.00000000000&    0
C' =  0.00000000000&    0
D' =  0.00000000000&    0
E' =  0.00000000000&    0
F' =  0.00000000000&    0
G' =  0.00000000000&    0
H' =  0.00000000000&    0

U' =  0.00000000000&    0
V' =  0.00000000000&    0
W' =  0.00000000000&    0
X' =  ********************
Y' =  0.00000000000&    0
Z' =  0.00000000000&    0

A  =  1.00000000000&   39
B  =  0.00000000000&    0
C  =  0.00000000000&    0
D  =  5.62731433888&    0
E  =  0.00000000000&    0
F  =  0.00000000000&    0
G  =  0.00000000000&    0
H  =  0.00000000000&    0

U  =  0.00000000000&    0



V  =  3.16666666678&    1
W  =  0.00000000000&    0
X  =  1.00000000000&   39
Y  =  0.00000000000&    0
Z  =  0.00000000000&    0

E  =  3.14159265368&    0
E' =  0.00000000000&    0



L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L0   L3
JUMPDOWN R1/1   L1
JUMPDOWN R2   RETURN
RETURN
L1   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L2   L0   L3

JUMPDOWN R1/1   L1
JUMPDOWN R2   RETURN
RETURN
L1   L2   L2
```

Section 5

MISCELLANEOUS FEATURES

ABOUT THIS SECTION 5.1

This section is provided largely for the benefit of programmers who are familiar with a version of Mercury Autocode, with Atlas or Orion EMA, or with 1300 series MAC, and for users who wish to transfer programs written in one of these languages to a 1900 series computer.

In Section 5.2 there is a list showing how the 1900 EMA compiler handles various instructions and directives from earlier versions of the language. Although some of these could be used when writing new 1900 EMA programs, in general they are provided for compatibility reasons.

Section 5.3 gives some general advice on transferring Atlas/ Orion EMA programs to a 1900 series computer. Section 5.4 gives similar advice for Mercury Autocode programs, and, finally, Section 5.5 considers the transfer of 1300 series MAC programs.

ALPHABETICAL CHECK LIST 5.2

The list below is not necessarily comprehensive but, when used in conjunction with Section 5.3, 5.4 or 5.5, it should answer many queries as to the changes that may be necessary to an existing program before it is compiled by the 1900 EMA compiler.

592, 0	This Mercury instruction is treated as equivalent to End Aux Seq as the last instruction of an Int Step auxiliary sequence.
620, *n*	This Mercury instruction is treated as equivalent to the Punch (*n*) instruction described below.
630(I)	This Mercury instruction is treated as equivalent to the Punch (*n*) instruction described below.
ALP or **ALP *n***	These directives and their associated instructions will cause an error indication; no Mercury Autocode List processing instructions are acceptable.
Arrow as Tape Terminator	An arrow on a line by itself is ignored by the 1900 EMA compiler and, in data, an arrow is ignored by the Read instructions if it is found ahead of a number. It is therefore unnecessary to remove the arrows used as tape terminators in Mercury Autocode (but see Section 5.4).

Arrow in Jump Instructions

An arrow is acceptable in place of the word "Jump" in Jump instructions but not in Jumpdown instructions. Thus

→ 4,X > 70

is an acceptable alternative to JUMP4, X>70.

BREAK OUTPUT

This Atlas instruction is ignored by the 1900 EMA compiler.

CAPTION

A line after the instruction CAPTION is printed as if by the Print (' ') instruction, e.g.

```
CAPTION
EXCEPTION CASES
```

would print

```
EXCEPTION CASES
```

All spaces on such a line, including those at the end of the line, are output, and no newlines are added.

In 1900 EMA, unlike some versions of Mercury Autocode, this instruction can appear in a Routine.

ϕ CHANGE EXPONENT (x, t)

This general function is accepted and has the value $x.2^t$.

CHANNELnP
CHANNEL [p]P

These Mercury Autocode instructions are treated as equivalent to SELECT OUTPUTn($1 \leqslant n \leqslant 15$).

CHANNELnR
CHANNEL [p]R

These Mercury Autocode instructions are treated as equivalent to SELECT INPUT n($1 \leqslant n \leqslant 15$).

CHAPLENGTH n

This Orion directive is ignored by the 1900 EMA compiler.

CHECK (X, Y, E, l)
CHECK (X, Y, E, I)

l is a label and I is an index that has been set equal to a label. The instructions are accepted as the exact equivalent of

JUMPl, E>ϕMOD(X-Y)

and

JUMP (I), E>ϕMOD(X-Y)

respectively.

CORRECTION
CORRECTIONS

These Atlas directives will cause an error indication during a 1900 EMA compilation.

"Curly" Equals

The character ≈, which may appear in data or programs punched in five-track paper tape for a Mercury Autocode compiler, is treated as = by the 1900 EMA compiler. This is acceptable in arithmetic instructions because 1900 series computers use a more sophisticated rounding system than Mercury.

DATA TABS

This initial directive in Atlas or Orion EMA is ignored by the 1900 EMA compiler.

DELETE m
DELETE m-n

These Orion directives will cause an error indication during a 1900 EMA compilation.

DELETE LINE n

This Atlas directive will cause an error indication during a 1900 EMA compilation.

ϕ DIVIDE (x, y)

This Mercury Autocode general function is accepted as equivalent to (x) / (y).

DOUBLE LENGTH

This Mercury Autocode directive is unnecessary in 1900 EMA and is ignored if it appears.

END OF CORRECTIONS

This Atlas directive will cause an error indication during a 1900 EMA compilation.

EVEN and ϕ EVEN

These Mercury autocode directives are ignored by the 1900 EMA compiler.

ϕ EXPONENT (x)

This integer function has the value f, where f is an integer defined by

$$x = a.2^f$$

a being in one of the ranges

$$-1 \leqslant a < -\tfrac{1}{2}$$

or

$$a = 0$$

or

$$\tfrac{1}{2} \leqslant a < 1$$

EXTRA CHAPTERS
EXTRA ROUTINES

These Orion directives will cause an error indication during a 1900 EMA compilation.

ϕ FLOATING POINT (x, t)

This general function has the value $a.2^t$, where a is defined by

$$x = a.2^f$$

and is in one of the ranges

$$-1 \leqslant a < -\tfrac{1}{2}$$

or

$$a = 0$$

or

$$\tfrac{1}{2} \leqslant a < 1$$

FREE TAPE n
FREE TAPE [p]

These Atlas EMA instructions are ignored by the 1900 EMA compiler.

HANDKEYS (I)

On the Orion and Mercury computers, this instruction copied the settings of the console handkeys to the specified index. 1900 series computers have no handkeys but word 30 of storage is set aside to simulate them. Initially word 30 will be zero but the operator may alter binary digits in the word.

The binary digits of a 1900 word are numbered 0 to 23 from the left-hand or more-significant end of the word. Bit 19 and Bit 0 of word 30 have a particular significance.

Bit 19, which corresponds in position and function to handkey 4 on Mercury, is associated with query printing. If it is 1, no query printing takes place even if queries have been compiled. If it is 0, then query printing takes place. Bit 19 may be set by the operator, or by START QUERIES (which sets it to 1) or STOP QUERIES (which sets it to 0).

Bit 0 performs å similar function for trace printing.

Bit 23 is used to suppress paging.

HOOT
HOOT n

These Mercury Autocode and 1300 MAC instructions, which are intended to provide an audible check that a program is proceeding correctly, are ignored by the 1900 EMA compiler.

I.C.I. variants

The I.C.I. Mercury Autocode phrases, φREADTAPECH, φNONEQUIV and φSIDEWAYSADD are accepted as equivalent to φTAPE, φNONEQ and φSIDEADD respectively.

INDICATE (I)

This instruction copies the value of I to a word known as the *indicator*. The contents of the indicator are then available to the TEST INDICATOR instruction (see below).

Word 31 is used as the indicator and, if required, the operator can examine its contents during the running of the program.

JUMP *n*, *a* SET

This 1300 MAC instruction will cause an error indication during a 1900 EMA compilation.

JUMP UP

This Mercury Autocode instruction is treated as equivalent to RETURN.

Machine Instructions

Except for the instructions

592, 0 620, *n* 630(I)

Mercury machine-code instructions are not accepted and should be replaced by EMA instructions or by a Chapter written in PLAN.

NEWCARD

See PUNCH (X) below.

NEWLINEN

See SPACEN below.

ORION ROUTINE

This directive and the associated Routine will cause an error indication during a 1900 EMA compilation.

PRINT 'ALPHA'

This 1300 MAC instruction will not be accepted and must be altered to the equivalent 1900 EMA instruction, PRINT ('ALPHA').

PRINT (*x*)

In 1900 EMA, the value of x is always output on a new line regardless of the output medium.

This statement only applies to PRINT (x) instructions, not to PRINT $(x)^m, n$ instructions.

PRINT (X) M, N

In 1300 MAC, if M=0 then X is printed in floating-point form with a mantissa x of N significant digits ($0.1 \leqslant x < 1$).

In 1900 EMA, however, if M = 0 then the mantissa x has N + 1 significant digits ($1 \leqslant x < 10$).

PROGRAMME TABS

This initial directive in Atlas or Orion EMA is ignored by the 1900 EMA compiler.

PROGRAM - *n*

This 1300 MAC directive is not acceptable and must be converted to the equivalent 1900 EMA directive, PROGRAMME - n.

PSA

This directive has no significance in 1900 EMA and is ignored.

PUNCH (*n*)

This Mercury Autocode instruction may be used to ouput characters to five-track paper tape. Normally n will be an integer in the range $0 \leqslant n \leqslant 31$ to specify one of the 32 characters shown in the table in Section 4; but n can be an expression, in which case the least -significant five bits of the integral part of this expression's value will be output. The instruction is therefore suitable for pseudo-binary output.

As in Mercury Autocode and Orion EMA, no extra shift characters are output. The instructions

3146(10.65)

PRINTLINE
PRINT (' ')
CAPTION
READ DATA TITLE

leave five-track paper tape in figure shift. A Print Ch instruction leaves the document in the same shift as the character it outputs. On media other than five-track paper tape an approximation is given for certain characters. It is provided solely for compatibility and is not recommended for normal use.

PUNCH (X)
PUNCH (I)
NEWCARD

These 1300 MAC instructions are not acceptable in 1900 EMA. They should normally be converted to PRINT and NEWLINE instructions which if necessary can be used in conjunction with a SELECT OUTPUT instruction. Note that there is a PUNCH instruction in 1900 EMA (see above) but this is totally unrelated to the MAC instruction.

Query Printing

In 1900 EMA, query printing is normally controlled by the directives COMPILE and IGNORE QUERIES and by the instructions START and STOP QUERIES; but operator control is also permitted during the running of the program (see HANDKEYS).

The simplest way of eliminating all queries from existing programs is to write IGNORE QUERIES ahead of the first Routine or Chapter.

Quicky Lists

These lists, used on Orion and Mercury, are ignored by the 1900 EMA compiler.

READ DATA TITLE

The instruction READ DATA TITLE has exactly the same effect in 1900, Atlas and Orion EMA (see Section 1.4.3). In I.C.I. Mercury Autocode, however, an automatic newline is obtained after the title has been printed and programs written in this version will require amendment.

REPLACE LINE *n*

This Atlas EMA directive will cause an error indication during a 1900 EMA compilation.

RMP

This instruction, Read More Program, is interpreted in different ways by earlier compilers and is treated as an error by the 1900 EMA compiler.

SET ((X, Y)) = Z

This instruction is treated as equivalent to

((X, Y)) = Z

where Z is any single-length expression.

SPACEN
NEWLINEN

In 1300 MAC, N could be an index; this is not permitted in 1900 EMA except in the forms SPACE (N) and NEWLINE (N). The brackets can be omitted only if N is an integer.

STOP

This Mercury Autocode instruction is ignored by the 1900 EMA compiler.

STOP 115 *n*

This 1300 MAC instruction is not allowed in 1900 EMA and will cause a compilation error.

SUNVIC LOGGER

This specialized Mercury Autocode instruction will cause an error indication during a 1900 EMA compilation.

I = ϕ TAPE (*n*)

This instruction may be used to read information from five-track paper tape. The effect is to read the next character, add its binary value to *n* and assign the sum to I. Erases and shift characters are treated like any other character. This instruction may not be

used for any form of input medium except five-track paper tape. It is provided solely for compatibility and is not recommended for normal use.

T' after READ or ϕ 10

After a READ or ϕ10 instruction is applied to five-track paper tape, T' will contain the five-track code for the last tape character to have been read. After a successful Read operation, T' will therefore contain 14 (the Space code), 30 (Carriage Return), or 29 ('). After an unsuccessful Read operation, T' will contain the spurious tape character and the program will have jumped to label 0 or will have been suspended.

The spurious character could be a printable character in figure shift or the character Letter Shift. In the latter case, the ϕTAPE instruction above might well be used to read the next tape character.

If the input medium is not five-track tape, T' will still be set equal to the five-track equivalent of the last character read. Thus, if a number is terminated by newline or end of card, T' is set to 30.

TEST (*l*)

This Mercury Autocode instruction is ignored.

TEST INDICATOR (I)

This instruction copies the value of the indicator to the specified index (see INDICATE (I) above).

TITLE

This Mercury directive and the accompanying text are ignored.

VARIABLES 0

This directive will cause an error indication during a 1900 EMA compilation. It can be replaced by an appropriate VARIABLES *m-n* directive (see Section 2.1.5).

In may cases the only changes necessary when an Atlas or Orion EMA program is transferred to a 1900 series computer will be those caused by the different operating systems and will be no more extensive that those required to transfer from Orion to Atlas or vice versa. Other programs which make extensive use of obsolescent features implemented differently by the three compilers may require more extensive changes. Some programs may be too large for the particular 1900 series configuration to be used and will require re-arrangment.

The notes and recommendations on the transfer of Atlas/Orion programs given below should be read in conjunction with Section 5.2.

1 Atlas or Orion Job Descriptions must be replaced by 1900 Program Descriptions (see Section 4).

2 All Atlas machine-instructions or ORION ROUTINES must be replaced by EMA instructions or Chapters written in PLAN.

3 Programs or data punched on seven-track paper tape must be re-punched on eight-track paper tape or on cards. As Atlas EMA uses a different card code from 1900 EMA, cards prepared for Atlas will also need to be re-punched, either as eight-track paper tape or as cards suitable for 1900 EMA. Note that all brackets except (,), [or] should be converted to [or] ; that certain characters used in text in Atlas and Orion EMA have no equivalent in 1900 EMA; that no underlining is permitted in 1900 EMA; and that lower-case letters are converted to upper-case during 1900 EMA input.

4 In Atlas or Orion EMA, the omission of AUXILIARY and DUMPS causes 10,752 auxiliary variables and 2 dumps to be reserved. In 1900 EMA, however, omission results in no auxiliary variables and no dumps, so it may be necessary to add either or both of these directives when a program is transferred.

5 The different word structure of Atlas, Orion and 1900 series computers may cause a transferred program to give slightly different results in cases where accuracy is marginal.

6 Again, it may be necessary to alter certain logical functions, such as ϕAND, because of the different word lengths employed on the three machines.

7 In 1900 EMA, the writing of any magnetic-tape block results in the loss of existing data in any following blocks. This is not true of Atlas or Orion EMA, and consequently some alteration to magnetic tape usage may be necessary when a program is transferred. Also note that the instruction

```
Read Bkd
```

is acceptable only if a 1974 Magnetic-tape System is in use.

8 VARIABLES 0 is not acceptable in 1900 EMA, and must be altered to an appropriate VARIABLES *m-n* directive (see Section 2.1.5).

9 The List Processing instructions available with the Atlas EMA/ALP compiler are not available in 1900 EMA and programs which use them cannot be run on a 1900 series computer.

10 The Atlas correction facility is not acceptable in 1900 EMA and a program containing a correction such as

```
REPLACE LINE 15 BY
3) A = B + 1
```

must be amended by removing this instruction and actually replacing line 15 by

```
3) A = B + 1
```

11 In Atlas/Orion EMA, certain instructions designed for five-track paper-tape input/output can be applied to seven-track paper tape or the line printer. This is not the case in 1900 EMA and such instructions must be replaced if they are to be applied to any media other than five-track paper tape.

Also note that in some cases Atlas EMA inserts extra shift characters on five-track paper-tape output and ignores shifts on input. This is not so in the case of 1900 EMA which behaves exactly as Orion EMA or Mercury Autocode.

12 In 1900 EMA, it is not possible to refer separately to the least-significant part of a double -precision number. Any existing program which does this will provide incorrect results on a 1900 series computer.

13 In 1900 EMA, a jump to label 100 with Fault Number 34 in T' occurs when *any* number outside the permitted ranges is detected. In Atlas and Orion EMA, it could never occur during calculations involving indices and integers alone.

14 In 1900 EMA, the original Mercury Autocode instruction Halt can be used; in Atlas and Orion EMA it is ignored.

15 ϕ Random and Random No produce a different sequence of numbers on Atlas, Orion and 1900 series computers.

16 Diagnostic aids such as error listing and trace printing are different for each machine, the 1900 EMA facilities being described in Section 4.

17 In 1900 EMA, unlike Atlas/Orion EMA, PRINT (x) always outputs the value of x on a new line, regardless of the output medium. (This does not apply to PRINT (x) m, n.)

18 Certain other features, which should normally occur only if the program was originally written for Mercury, will be treated differently on different machines. Details can be obtained from the check list in Section 5.2.

TRANSFERRING MERCURY AUTOCODE PROGRAMS 5.4

Many programs, even if they are written in a CHLF version of Mercury Autocode, will require only trivial changes made necessary by the 1900 series operating system; others which make extensive use of features that are implemented differently in Mercury Autocode and 1900 EMA may require more extensive changes; some Mercury Autocode programs may be too large for the particular 1900 series configuration to be used and will require re-arrangement.

The following notes and recommendations on the transfer of Mercury Autocode programs should be read in conjunction with Section 5.2.

1 A 1900 Program Description must be written (see Section 4) and added to the beginning of the program.

2 All machine instructions except

592, 0
620, *n*
630(I)

must be replaced by equivalent EMA instructions or Chapters written in PLAN.

3 Five-track paper tape is acceptable as input and available, though not recommended, for output. In general, it is advisable to direct output to eight-track paper tape or a line printer, but note that then all Punch (*n*) instructions must be replaced because they can be applied to five-track paper tape only.

4 If auxiliary variables or dumps are required, then AUXILIARY and DUMPS directives must be added to the program (see Sections 2.2.3 and 2.2.4). Again, if more than, or significantly fewer than, 480 main variables are used, a MAIN directive should be added (see Section 1.2.4).

5 The different word lengths of Mercury and 1900 series computers may cause a transferred program to give slightly different results in cases where accuracy is marginal. Note that, in general, 1900 EMA provides more accurate results than Mercury Autocode and that 1900 EMA indices are much larger than Mercury Autocode indices.

6 It may be necessary to alter certain logical functions such as ϕAND because of the different word lengths employed on the two machines.

7 VARIABLES 0 is not acceptable in 1900 EMA and must be altered to an appropriate VARIABLES *m-n* directive (see Section 2.1.5).

8 The List Processing instructions available with some versions of Mercury Autocode are not available in 1900 EMA, and programs which use them cannot be run on a 1900 series computer.

9 In 1900 EMA, it is not possible to refer separately to the least-significant part of a double -precision number. Any existing program which does this will provide incorrect results on a 1900 series computer.

10 In 1900 EMA, there is no fixed chapter length, although approximately 100 EMA instructions is a reasonable working maximum. It is therefore possible to re-arrange a Mercury Autocode program (restricted by fixed chapter length) to take advantage of the more flexible 1900 EMA facilities.

11 It is necessary to add the tape terminators ***T or ***Z to the end of each paper tape (see Section 4) but note that it is not necessary to delete the arrows used as tape terminators on Mercury, since the 1900 EMA compiler will merely ignore them.

12 The CHLF ϕ29 and ϕ30 instructions and the Manchester ϕ29, ϕ30 and ϕ31 matrix instructions are acceptable.

13 ϕRandom and Random No produce a different sequence of numbers on Mercury and 1900 series computers.

14 In 1900 EMA, a jump to label 100) with Fault Number 34 in T' occurs when *any* number outside the permitted range is detected. In Mercury Autocode, it occurs only when the argument of ϕexp is too large.

15 At compilation time, query printing is controlled by the directives COMPILE QUERIES and IGNORE QUERIES. At run time, it is preferable to control query printing by the instructions START QUERIES and STOP QUERIES, but it is also possible, as with Mercury, to put query printing under operator control (see HANDKEYS, Section 5.2).

16 The specialized SUNVIC LOGGER instruction is not acceptable in 1900 EMA.

17 The instruction RMP is not available in 1900 EMA and will be treated as an error.

18 Diagnostic aids, such as error listing, are different on the two machines, the 1900 EMA facilities being described in Section 4.

TRANSFERRING 1300 SERIES MAC PROGRAMS 5.5

All 1300 series MAC programs and data will need re-punching before they can be transferred to a 1900 series computer. The number of additional changes required will vary from program to program.

The following notes and recommendations on the transfer of such programs should be read in conjunction with Section 5.2.

1 All programs and data must be re-punched in eight-track tape or cards because the codes and formats used for 1300 MAC cards differ from those used in 1900 EMA (see Section 4). In MAC data, numbers may be terminated by one space but not by end of card. In 1900 EMA, however, *two* spaces, the end of a card (or newline on paper tape), or an apostrophe may be used to separate numbers.

2 A 1900 Program Description must be written (see Section 4) and added to the beginning of the program.

3 Any machine-code instructions must be replaced by equivalent EMA instructions or Chapters written in PLAN.

4 The different word structure of 1300 series and 1900 series computers may cause a transferred program to give slightly different results in cases where accuracy is marginal. In general, accuracy will be increased on a 1900 series computer.

5 VARIABLES 0 is not acceptable in 1900 EMA and must be altered to an appropriate VARIABLES *m-n* directive (see Section 2.1.5.).

6 An 'ambiguous' division such as B/DE means (B/D) E in 1300 MAC but is not permitted in 1900 EMA.

7 In 1900 EMA, there is no fixed chapter length although approximately 100 EMA instructions is a reasonable working maximum. It is possible to re-arrange a 1300 MAC program (restricted to a fixed chapter length depending upon I.A.S. size) to take advantage of the more flexible 1900 EMA facilities.

8 There is no system of absolute chapter numbering in EMA. Thus, if VARIABLES *n* appears in a master chapter, there must be a chapter numbered *n*.

9 There is no automatic chapter-changing in 1900 EMA as there is in 1300 MAC; an ACROSS, DOWN or UP instruction is always needed.

10 The MAC magnetic tape instructions φ1 to φ4 are not acceptable in 1900 EMA. Thus all magnetic tape instructions must be altered. All 1900 EMA magnetic tape transfers involve blocks of 512 main variables.

11 At compilation time, query (or Monitor) printing is controlled by the directives COMPILE QUERIES and IGNORE QUERIES. At run time it is preferable to control query printing by the instructions START QUERIES and STOP QUERIES, but it is also possible, as with MAC, to put it under operator control (see HANDKEYS, Section 5.2).

12 Labels 0) and 100) have a special significance in 1900 EMA (see Section 4) and should be avoided for normal use.

13 Diagnostic aids, such as error listing, are different on the two machines, the 1900 EMA facilities being described in Section 4.

14 Several instructions and directives in 1300 MAC are not available or have different effects in 1900 EMA. These are listed in Section 5.2.

15 The range of the ARCTAN function in 1900 EMA is different to that in 1300 MAC.

SUMMARIZED
PROGRAMMING
INFORMATION

SUMMARIZED PROGRAMMING INFORMATION

This section lists, with certain exceptions, the general forms of the instructions, directives and functions that may appear in an EMA program, including those that are allowed purely for compatibility with earlier versions of Mercury Autocode. The exceptions are:

(a) The matrix instructions. These are listed in Section 3.2.2.

(b) Program Description directives. These are covered in Section 4.

Except where otherwise stated, the notation used below is as follows:

A	One of A, B, C, D, E, F, G, H, U, V, W, X, Y, Z, π.
c	An expression in a complex context.
ch	An EMA character or character abbreviation.
d	An expression in a double-precision context.
I, J	Indices
K	An index that has been set equal to a label.
l, l_1, l_2 etc.	Label numbers.
m, n, n_1, n_2 etc.	Unsigned integers.
p, q, r	Expressions in integer contexts.
p_1, p_2 etc.	Integer constants or φCODE or φOCTAL functions.
S	A suffix.
t	An octal integer.
Text	A string of EMA characters.
U, V, W	Special or main variables.
$((u,v))$	A double-precision pair.
x, y, z	Expressions in single-precision contexts.
[ ]	Any type of matched bracket pair may be substituted.
( )	Only a round bracket pair is allowed.
)	Only a round bracket is allowed.
?	Indicates that the instruction may be query printed.

Spaces are not significant and, except in text, they can be added or omitted anywhere on a line. Upper -case and lower-case letters may be interchanged with no change of meaning.

INSTRUCTIONS AND DIRECTIVES

These are listed alphabetically under the key word, e.g. "TAPE" in "*I*=ϕTAPE [*p*]".

**	Close-of-Routine directive. Two or more asterisks.
592,0	
620,*n*	
630 [*p*]	
A→*n*	
I = *r* ?	*r* is expression in *rounded* integer context.
V = *x* ?	
[*U*,*V*] = c	
[*I*,*J*] = *c*	
((*U*,*V*)) = *d* ?	
I) = *p*)	0≤*p*≤127 Alternative Formats.
I) = *p*	
I = *p* (*q*) *r* ?	*p*, *q*, *r* are indices or unsigned integers.
I = *p* (-*q*) *r* ?	
ACROSS *//m*	Label/Chapter number.
ACROSS *//m-n*	Label/Chapter number - Programme number.
AUXILIARY [*m*,*n*]	*m*,*n* may be signed or unsigned.
AUXILIARY [*m*,*n*,*p*]	*m*,*n*,*p* may be signed or unsigned.
BREAK OUTPUT	Atlas instruction. Ignored on 1900.
CAPTION	On successive lines.
Text	
CHANNEL *n* P	1≤*n*, *p*≤15
CHANNEL [*p*] P	
CHANNEL *n* R	
CHANNEL [*p*] R	
CHAPLENGTH *n*	Orion directive. Ignored on 1900.
CHAPTER *m*	0≤*m*≤4095
CHECK [*x*, *y*, *z*, *I*]	
CHECK [*x*, *y*, *z*, *K*]	
CLEAR LEVEL	
CLOSE	
COMPILE QUERIES	
COMPILE TRACE	
DATA TABS [n_1, n_2, ...]	Ignored.
DEPTH *n*	
DOUBLE LENGTH	Ignored.
DOWN *//m*	Label/Chapter number.
DOWN *//m-n*	Label/Chapter number - Programme number.
DUMPS *n*	
END	
END AUX SEQ	
EVEN	Ignored.
ϕ EVEN	

FAULT n FAULT [p] FREE TAPE n FREE TAPE [p] FROM RANDOM	Atlas instruction. Ignored on 1900. $0 \leqslant n, p \leqslant 7$.
HALT HANDKEYS [I] HOOT HOOT n	Ignored.
IGNORE QUERIES IGNORE TRACE INDEX STORES n INDEX STORES [n] INDICATE [p] INTEGERS p_1 , p_2 , ... INT STEP [l] INT STEP [K]	Alternative formats.
JUMP l JUMP [K] JUMP l, $x \left\{ > \geqslant = \neq < \leqslant \right\} y$ JUMP [K], $x \left\{ > \geqslant = \neq < \leqslant \right\} y$ JUMP DOWN l JUMP DOWN [K] JUMP DOWN [Rn] JUMP DOWN [Rn/l] JUMP DOWN DEPTH n JUMP UP	"→" can replace the word "JUMP". Routine number. Routine number/label number. Alternative format for RETURN.
LABELS l_1) l_2) ... LOOP, $V = x$ ϕINC [y]z ? LOOP, $I = p$ ϕINC [q]r ?	 p, q, r are in *rounded* integer contexts.
MAIN n MAIN→n	Alternative formats.
NEWLINE NEWLINE n NEWLINE [p]	n, $p \geqslant 0$
PACK [V, p, I] V = ϕPOLY [x] AS, p ? I = ϕPOLY [x] AS, p ? PRESERVE PRESERVE INDICES [I, p] PRINT [p] PRINT [x] PRINT [d] PRINT ((u, v)) PRINT [p] q, r PRINT [x] q, r PRINT [d] q, r PRINT ((u, v)) q, r PRINT ['Text'] PRINT CH [p]	$P \geqslant 0$ I not R, S, T, R', S' or T' $0 \leqslant p \leqslant 4095$

PRINTLINE Text	On successive lines.
PROGRAMME-n	$0 \leqslant n \leqslant 4095$
PROGRAMME TABS [n_1 , n_2 , ...]	Ignored.
PSA	Ignored.
PUNCH [p]	
V=ϕRANDOM [x, p] ?	x must be a variable if $p \neq 0$.
RANDOM NO [V]	
READ [I]	
READ [V]	
READ ((U, V))	
READ [((U, V))]	
READ BKD [p, AS]	Alternative formats.
READ BKWD [p, AS]	$0 \leqslant p \leqslant 7$.
READ CH [I]	
READ DATA TITLE	
READ FWD [p, AS]	$0 \leqslant p \leqslant 7$
I=ϕREAD TAPE CH [p] ?	Alternative format for ϕ TAPE.
RELINQUISH DECK n	$0 \leqslant n, p \leqslant 7$
RELINQUISH DECK [p]	
RELINQUISH INPUT n	$1 \leqslant n, p \leqslant 15$
RELINQUISH INPUT [p]	
RELINQUISH OUTPUT n	
RELINQUISH OUTPUT [p]	
REPEAT	
RESTORE	
RESTORE INDICES [I, p]	I not R, S, T, R', S' or T'.
RETURN	
REWIND n	$0 \leqslant n, p \leqslant 7$
REWIND [p]	
ROUTINE n	$0 \leqslant n \leqslant 4095$
RUNOUT	
SELECT INPUT n	$1 \leqslant n, p \leqslant 15$
SELECT INPUT [p]	
SELECT OUTPUT n	
SELECT OUTPUT [p]	
SET ((U, V)) = z ?	
SPACE	
SPACE n	$n, p \geqslant 0$
SPACE [p]	
START QUERIES	
START TRACE	
STOP	Ignored.
STOP QUERIES	
STOP TRACE	
I= ϕTABLE [l, p] ?	$p \geqslant 0$
I= ϕTABLE [K, p] ?	
V= ϕTABLE [l, p] ?	
V= ϕTABLE [K, p] ?	
I= ϕTAPE [p] ?	
TEST [l]	Ignored.
TEST INDICATOR [I]	
TEST LEVEL [I]	
TITLE Text	On successive lines. Ignored.
TO RANDOM	
TRACE	Alternative format for COMPILE TRACE.

UNPACK [V, p, I]	$p \geq 0$
UP	
UP LEVEL	
VARIABLES m	m ($\neq 0$) is a Chapter number.
VARIABLES $m-n$	m is Chapter number; n is Programme number.
WIND TAPE [p, q]	$0 \leq p \leq 7$, $q \geq 1$.
WRITE [p, AS]	$0 \leq p \leq 7$

INTEGER FUNCTIONS

ϕAND [p, q]	
ϕCODE [ch]	
ϕEXPONENT [x]	
ϕINT PT [x]	
ϕMAX [AO, p, q]	
ϕMIN [AO, p, q]	
ϕNONEQ [p, q]	Alternative formats.
ϕNONEQUIV [p, q]	
ϕOCTAL [t]	
ϕOR [p, q]	
ϕPARITY [p]	
ϕSHIFT LEFT [p, q]	
ϕSHIFT RIGHT [p, q]	
ϕSIDE ADD [p]	Alternative formats
ϕSIDEWAYS ADD [p]	
ϕSIGN [x]	

GENERAL (Single-precision) FUNCTIONS

ϕARCCOS [x]	$-1 \leq x \leq 1$
ϕARCSIN [x]	$-1 \leq x \leq 1$
ϕARCTAN [x]	
ϕARCTAN [x, y]	$x^2 + y^2 \neq 0$
ϕCHANGE EXPONENT [x, p]	
ϕCOS [x]	
ϕDIVIDE [x, y]	$y \neq 0$
ϕEXP [x]	
ϕFLOATING POINT [x, p]	
ϕFR PT [x]	
ϕLOG [x]	$x > 0$
ϕMOD [x]	
ϕRADIUS [x, y]	
ϕSIN [x]	
ϕSQ RT [x]	$x \geq 0$
ϕTAN [x]	

COMPLEX FUNCTIONS

ϕEXP [x, y]
ϕEXP [c]
ϕLOG [x, y]
ϕLOG [c]
ϕSQ RT [x, y]
ϕSQ RT [c]

Index

Instructions, directives, functions etc. are in capital letters.

Amendment record

Should any additions or amendments to this Manual be necessary, they will be issued in the form of Amendment Lists.
When such changes have been incorporated, an appropriate entry should be made in the table below and the pink instruction sheet(s) should be filed behind this page

List number	Item number
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to