EDINBURGH UNIVERSITY COMPUTER UNIT

ATLAS AUTOCODE COMPILER FOR KDF 9

COMPUTER UNIT REPORT No. 4 1st October 1965

by Paul Bratley
David Rees
Peter Schofield
Harry Whitfield

Atlas Autocode Compiler for KDF 9 Manual

Amendment Notice (1-6-66)

Version I of the compiler is now in general use and is free from 'bugs' as far as is known. The two main changes from version H concern:

(i) Output
(ii) Fault Trapping

The ISO tape code version (K) does not yet contain these new facilities but will shortly be bought into line with Compiler I, described here.

(i) Output

The output device, line printer or paper tape punch, may now be specified by the programmer in his Job Heading, assuming the device of his choice is available at the time of running the job. The additional page 9 - 1 - 1 specifies how this may be done. See also the revised page 11 - 2 - 0. Note also that the amount of output is now checked!

A further modification, for both devices, ensures that a newline or newline character is inserted every 120 printing characters, should the programmer inadvertently omit to insert any. For this reason also, a newline is no longer inserted for every query printing operation.

(ii) Fault Trapping

This facility is now available, subject to the limitation of textual level, mentioned on the new page 3 - 10 - 1 under a)ii). See the miscellaneous modifications below for the phrase structure associated with this kind of statement.

E.g.

<u>fault</u> 1 -> 23, 5, 6, 22 -> 31

means that if fault 1 occurs, then jump to label 23, or if faults 5, 6, or 22 occur then jump to label 31. The list of fault numbers is given on page 8 - 2 - 0 of the manual. For the example, therefore, if overflow (fault 1) occurred or a square root with a negative argument (fault 5) was attempted, etc., the program would be restarted at the appropriate label.

Miscellaneous Modifications to the Manual

Page 0 - 3 - 0

Add: Section 16 USING TELETYPES

Section 17 DOUBLE LENGTH ROUTINES

Page 2 - 2 - 0

Line 18: change [NAME]{(}[±'][N]{)}{:}

to [NAME]{(}[±'][CONST]{)}{:}

Insert between lines 23 and 24:

{fault}[FAULT LIST][S],

Page 2 - 3 - 0

Lines 19 and 20: change

```
P[SWITCH LIST]=[NAME LIST]{(}[±'][N]{:}[±'][N]{)}{,}[SWITCH LIST],
               [NAME LIST]{(}[±'][N]{:}[±'][N]{)};
```

to

```
P[SWITCH LIST]=[NAME LIST]{(}[±'][CONST]{:}[±'][CONST]{)}[SWITCH LIST],
               [NAME LIST]{(}[±'][CONST]{:}[±'][CONST]{)};
```

Line 42 onwards:

Delete
```
P[S]= ..........
P[TEXT]= ........
P[CHARACTER]= .....
P[END OF LINE]
```

and replace by:

```
P[S]={;},[NEWLINE CHARACTER];
P[TEXT]=[ANY CHARACTER EXCEPT ; OR NEWLINE][TEXT],∅;
P[FAULT LIST]=[N-LIST]{->}[N]{,}[FAULT LIST],[N-LIST]{->}[N];
P[N-LIST]=[N]{,}[N-LIST],[N];
```

Page 3 - 8 - 0

Line 11: replace ....[N].......

by ....[CONST]........

Line 17: replace ....[±'][N]..........

by .... -valued quantity [±'][CONST]......

Page 3 - 9 - 0

Lines 10 and 11:

Replace:	in the form	newline print (value,1,0)
by:	in the form	print (value,9,0)

Line 14:

Replace:	in the form	newline
by:	in the form	spaces (3)

Insert new page 3 - 10 - 1 (attached)

Page 8 - 1 - 0

Line 28:

Replace: 26. by:

26. fault STATEMENT NOT AT BASIC TEXTUAL LEVEL

Page 8 - 2 - 0

Line 10:

Delete asterisk in 9.* INPUT ENDED

i.e. now trappable.

Page 8 - 2 - 0 (continued)

Line 30:

Replace: 29. by:

29. * OUTPUT EXCEEDED

Insert new page 9 - 1 - 1 (attached)

Insert revised page 11 - 2 - 0 (attached)

Insert new Section 17 (attached)

PREFACE

Atlas Autocode is an automatic programming language of Algol type, originally developed by R.A. Brooker and J.S. Rohl for the Atlas Computer at Manchester University.

An Atlas Autocode compiler has now been written for the KDF9 computer. This document defines the version of Atlas Autocode acceptable to the KDF9 compiler. The authors wish to thank Mr.D. Kershaw for his help in writing and testing the mathematical function routines.

For details of other versions of the language, see references (1), (2) and (3). Where the differences are likely to affect the programmer, a footnote appears in this document.

References

(1) Programming in Atlas Autocode, Edinburgh University Computer Unit Report No. 1, P.D. Schofield and M.R. Osborne, (Revised Edition) 28th June 1965.

(2) Atlas Autocode Reference Manual, University of Manchester Computer Science Department, R.A. Brooker and J.S. Rohl, 1st March 1965.

(3) The Atlas Autocode Mini-Manual, Manchester University Computer Science Department, W.F. Lunnon and G. Riding, March 1965.

CONTENTS

<u>Section 1</u> <u>INTRODUCTION</u>

An Atlas Autocode program consists of a sequence of <u>statements</u>.

We describe these statements by giving first their syntax, and secondly their semantics.

Section 2 SYNTAX

The syntax describes in phrase-structure notation the structure of the various forms of statement which are allowed in the language. In this notation square brackets denote a class name, and slashed brackets denote actual text in the program. A class name must be replaced by one of the members of this class. For example, the phrase-structure of one class of statement is

[N] {:} .

[N] is the class of unsigned integers. 23 is a member of the class [N]. It follows that

23 :

is a syntactically correct statement in an Atlas Autocode program.

In this example, [N] is a class-name, or phrase, which is defined in phrase-structure form, in terms of class names or actual text. [N], the class of unsigned integers is strictly defined by

```
P[N]       =   [DIGIT] [N], [DIGIT] ;
P[DIGIT]   =   {0}, {1}, {2}, ....{9} ;  .
```

This means that an unsigned integer is either (note the comma between alternatives) a [DIGIT] followed by another unsigned integer, or simply a [DIGIT]: a [DIGIT] is one of the digits 0-9. A definition is started by P (for PHRASE), and ended by a semi-colon. ø denotes the null text { }, as spaces are ignored everywhere in a program.

All statements of an Atlas Autocode program belong to the class P[SS] defined below:

```
P[SS]              =    [UI][S],
                        [iu][COND]{then}[UI][S],
                        [UI][iu][COND][S],
                        {cycle}[NAME][APP]{=}[±'][EXPR]{,}[±'][EXPR]{,}[±'][EXPR][S],
                        {repeat}[S],
                        [TYPE][NAME LIST][S],
                        [TYPE']{array}[ARRAY LIST][S],
                        {switch}[SWITCH LIST][S],
                        [RT]{spec}[NAME][FPP][S],
                        {spec}[NAME][FPP][S],
                        [RT][NAME][FPP][S],
                        {begin}[S],
                        {end}[S],
                        {end of program},
                        [N]{:},
                        [NAME]{(}[±'][N]{)}{:},
                        {compile queries}[S],
                        {ignore queries}[S],
                        {comment}[TEXT][S],{|}[TEXT][S],
                        {upper case delimiters}[S],
                        {normal delimiters}[S],
                        [S];
```

where

```
P[UI]              =    [NAME][APP]{=}[±'][EXPR][QUERY'],
                        [NAME][APP],
                        {->}[N],
                        {->}[NAME]{(}[±'][EXPR]{)},
                        {caption}[TEXT],
                        {return},
                        {result}{=}[±'][EXPR],
                        {stop};
```

and

```
P[±']                =    {+},{-},ϕ;
P[EXPR]              =    [OPERAND][OP][EXPR],[OPERAND];
P[OPERAND]           =    [NAME][APP],[CONST],{(}[±'][EXPR]{)},{|}[±'][EXPR]{|};
P[APP]               =    {(}[EXPR-LIST]{)},ϕ;
P[EXPR-LIST]         =    [±'][EXPR]{,}[EXPR-LIST],[±'][EXPR];
P[OP]                =    {+},{-},{*},{/},{↑},ϕ;
P[QUERY']            =    {?},ϕ;
P[,']                =    {,},ϕ;
P[iu]                =    {if},{unless};
P[real']             =    {real},ϕ;
P[TYPE]              =    {integer},{real};
P[TYPE']             =    {integer},{real},ϕ;
P[NAME LIST]         =    [NAME]{,}[NAME LIST],[NAME];
P[ARRAY LIST]        =    [NAME LIST]{(}[BOUND PAIR LIST]{)}{,}[ARRAY LIST],
                          [NAME LIST]{(}[BOUND PAIR LIST]{)};
P[BOUND PAIR LIST]=       [±'][EXPR]{:}[±'][EXPR]{,}[BOUND PAIR LIST],
                          [±'][EXPR]{:}[±'][EXPR];
P[SWITCH LIST]       =    [NAME LIST]{(}[±'][N]{:}[±'][N]{)}{,}[SWITCH LIST],
                          [NAME LIST]{(}[±'][N]{:}[±'][N]{)};
P[RT]                =    {routine},{real fn},{integer fn},{real map},{integer map};
P[FPP]               =    {(}[FP-LIST]{)},ϕ;
P[FP-LIST]           =    [FP-DELIMITER][NAME][FP-LIST],[FP-DELIMITER][NAME];
P[FP-DELIMITER]      =    [,'][RT],[,']{integer array name},
                          [,']{integer name},[,']{integer},[,'][real']{array name},
                          [,']{real name},[,']{real},[,']{addr},{,};
P[COND]              =    [SC]{and}[AND-C],[SC]{or}[OR-C],[SC];
P[AND-C]             =    [SC]{and}[AND-C],[SC];
P[OR-C]              =    [SC]{or}[OR-C],[SC];
P[SC]                =    [±'][EXPR][COMP][±'][EXPR][COMP][±'][EXPR],
                          [±'][EXPR][COMP][±'][EXPR],{(}[COND]{)};
P[COMP]              =    {=},{≠},{>},{≤},{<},{≥};
P[NAME]              =    [LETTER STRING] [DIGIT STRING'] [PRIME STRING'];
P[LETTER STRING]     =    [LETTER] [LETTER STRING],[LETTER];
P[DIGIT STRING']     =    [N],ϕ;
P[PRIME STRING']     =    {'} [PRIME STRING'],ϕ;
P[LETTER]            =    {A},{B},........,{Y},{Z},{a},{b},........,{y},{z};
P[CONST]             =    [FP CONST] {α} [±'] [N],[FP CONST],{π},{½};
P[FP CONST]          =    [N] {.} [N],[N] {.},[N],{.} [N];
P[N]                 =    [DIGIT] [N],[DIGIT];
P[DIGIT]             =    {0},{1},{2},.......,{9};
P[S]                 =    {;},[END OF LINE];
P[TEXT]              =    [CHARACTER] [TEXT],[CHARACTER],ϕ;
P[CHARACTER]         =    [LETTER],[DIGIT],{!},{π},{&},{'},{(},{)},{*},{+},{,},{-},
                          {.},{/},{:},{;},{<},{=},{>},{?},{[},{]},{↑},{α},{|},{_};
P[END OF LINE]
```

Section 3 SEMANTICS (OF SOURCE STATEMENTS)

Not every statement which is syntactically correct is meaningful. For example

<u>cycle</u> i = 7,1,10.5

is a syntactically correct statement, and will be recognized as such by the compiler. However, its meaning is not clear, and that part of the compiler which checks semantics will signal a fault. We therefore give below, corresponding to each source statement, a description of the semantic checks made, and of the effect of the statement both at compile and run time.

Section 3.1 Source statement [UI] [S]

[UI] = unconditional instruction, i.e. one which may be made conditional.

[S] = separator, i.e. semi-colon or end of line.

a) Compile time

b) Run time

c) General

See separate section on Unconditional Instructions.

Section 3.2 Source statement [iu] [COND] {then} [UI] [S]

where [iu] = {if} or {unless}

a) Compile time

b) Run time

i) when [iu] = {if}. If the conditional expression is true, then the unconditional instruction is obeyed. Otherwise it is skipped.

ii) when [iu] = {unless}. Contrariwise.

c) General

i) Only instructions of the class [UI] may be made conditional.

Section 3.3 Source statement [UI] [iu] [COND] [S]

a) Compile time

b) Run time

c) General

Treated as [iu] [COND] {then} [UI] [S].

See section 3.2.

Section 3.4 Source statement {cycle} [NAME] [APP] {=} [±'] [EXPR] {,} [±'] ...

a) Compile time

i) [NAME] [APP] must be an integer variable. The three expressions must be of integer type.

ii) A record is made so that a repeat can be associated with this cycle.

b) Run time

i) The three expressions(p,q,r say) are evaluated upon first entering the cycle. The cycle is monitored if

$$\frac{r-p}{q} \neq n, \text{ where } n \text{ is an integer} \geq 0$$

or $$q = 0.$$

ii) The address of the variable [NAME] [APP] is recorded.

iii) The recorded variable [NAME] [APP] is set at the beginning of the first traverse of the cycle to the value p. When the associated repeat is encountered, the current value of [NAME] [APP] is tested against the final value r. If these two values are equal, control passes to the next statement after the repeat. Otherwise, the current value of [NAME] [APP] is incremented by an amount q, and another traverse of the cycle is begun.

iv) Transfers of control within the body of a cycle, or out from the body of a cycle, are allowed in the usual way. However, control should not be transferred into the body of the cycle from outside in such a way that the values of p, q and r are undefined when the repeat is encountered.

v) Assignments to the recorded variable should not be made within the body of the cycle without a full regard to the possible consequences. For instance, in view of iii) above it will be clear that on encountering a cycle such as

```
cycle i = 1,1,10
i = 0
repeat
```

the program control will loop indefinitely.

c) General

i) Cycles may be nested to any depth, but a cycle and its associated repeat must be in the same block.

Section 3.5 Source statement {repeat} [S]

a) Compile time

i) Each repeat is associated with the last unassociated cycle statement in the same block. If no such cycle statement exists, a fault is recorded.

b) Run time

c) General

i) See cycle section 3.4.

<u>Section 3.6</u> Source statement [TYPE] [NAME LIST] [S]

a) <u>Compile time</u>

i) If any name on [NAME LIST] has been declared before in this block then a fault is recorded. Otherwise a location in the store is assigned to this name for use within this block at run time.

ii) The type of the name in this block is recorded.

b) <u>Run time</u>

i) The values of the names are lost on leaving the block.

c) <u>General</u>

<u>Section 3.7</u> Source statement [TYPE'] {<u>array</u>} [ARRAY LIST] [S]

[TYPE'] = <u>real</u>, <u>integer</u> or ϕ (ϕ equivalent to <u>real</u>)

a) <u>Compile time</u>

i) If any name on [ARRAY LIST] has been declared before in the block then a fault is recorded. Otherwise the name and the type of the name in this block are recorded.

ii) Instructions for calculating bounds and allocating space at run time for each array are compiled into the program.

b) <u>Run time</u>

i) The bounds of each array are computed, and space is allocated.

ii) However, MONITOR is entered if any lower bound exceeds the corresponding upper bound.

iii) This space is made available for other use at the end of the block.

c) <u>General</u>

i) As array declarations have an effect at run time, they should normally be placed at the head of the block to prevent inadvertent repeated allocation of new space.

__Section 3.8__ Source statement ⟨__switch__⟩ [SWITCH LIST] [S]

a) __Compile time__

i) If any name in [SWITCH LIST] has been declared before in this block then a fault is recorded. Otherwise the name and type of name in this block are recorded.

ii) The bounds are recorded and space is allocated for the storage of the addresses corresponding to the __switch labels__ (see section 3.15.).

b) __Run time__

c) __General__

Section 3.9 Source statement [RT] {spec} [NAME] [FPP] [S]

a) Compile time

i) The [NAME] is declared as an [RT] type name of the current block. A fault is recorded if the name has been set before in the current block except as an [RT] type parameter in a routine, fn or map description.

ii) A record is made of the type of each formal parameter. The names are not recorded, and it is only the order of the parameter types which is made use of.

b) Run time

c) General

i) Any [RT] specified must be described somewhere in the same block, otherwise a fault is recorded; except as in (ii).

ii) Within an [RT] description which uses routine-type parameters, there must be, corresponding to each such parameter, a routine type spec which appears before the first reference to that parameter. However, in the case of routine-type formal parameters, there must be no description corresponding to the required spec.

iii) The reduced form {spec} [NAME] [FPP] [S] may only be used as an alternative to the above where the specification is of a [RT] type parameter.

Section 3.10 Source statement [RT] [NAME] [FPP] [S]

[RT]	=	routine type
[FPP]	=	formal parameter part

a) Compile time

i) If the [NAME] has not been specified (see section 3.9) in the current block, then this source statement is first treated exactly as [RT] {spec} [NAME] [FPP] [S].

ii) This statement marks the beginning of a new block.

iii) The formal parameter names are declared in the new block in the appropriate way. A fault is recorded if the type of the first formal parameter in the heading is not the same as the type of the first parameter given in the corresponding spec; similarly for the second parameter, and so on.

iv) An [RT] can only be entered by an [RT] call, and therefore the compiler inserts a jump around the description.

v) Compilation of the [RT] follows.

b) Run time

i) The complete block is skipped.

c) General

Section 3.11 Source statement {begin} [S]

a) Compile time

i) This statement marks the beginning of a new block.

ii) The first statement of a program will normally be this statement.

b) Run time

c) General

Section 3.12 Source statement {end} [S]

a) Compile time

i) Denotes the end of a block.

ii) May denote the end of a routine, function or map.

iii) Labels of the block which have been used but not set are faulted.

iv) A check is made to ensure that all cycle statements have been associated with a repeat.

v) All the names declared in this block are unset.

vi) If the block is a routine then end is treated as return; end.
If the block is a fn or map, end is treated as MONITOR; end.

vii) If the end corresponds to the first begin at the head of the program, a fault is recorded and compilation ceases.

b) Run time

c) General

Section 3.13 Source statement {end of program}

a) Compile time

i) Exactly as {end} [S] (see section 3.12) i), iii), iv), v)).

ii) The unconditional instruction stop is compiled.

iii) A fault is recorded if this statement is not the end corresponding to the first begin of the program.

iv) Compilation ceases.

v) If no faults have been recorded the program is entered.

b) Run time

i) Execution ceases.

c) General

<u>Section 3.14</u> Source statement [N] £:∃

a) <u>Compile time</u>

i) If the integer [N] is already on the label list associated with this block, then a fault is recorded.

ii) Otherwise, the integer [N] and the address of the next compiled instruction are added to the label list.

b) <u>Run time</u>

No action.

c) <u>General</u>

i) [N] must be in the range 1-32,767

<u>Section 3.15</u> Source statement [NAME] £(∃ [$\underline{+}$'] [N] £)∃ £:∃

a) <u>Compile time</u>

i) [NAME] must have been declared in the current block as a <u>switch</u> variable.

ii) This statement may not appear in a block within the block in which the name was declared.

iii) The signed integer [$\underline{+}$'] [N] must be within the bounds declared for this switch variable.

iv) This <u>switch label</u> must not have been set before in this block.

v) If a fault has not been recorded the address of the next instruction is recorded as the address of this <u>switch label</u>.

b) <u>Run time</u>

c) <u>General</u>

Section 3.16 Source Statement {compile queries}[S]

a) Compile time

i) A marker is set so that query printing instructions which occur will be compiled into the program.

ii) Cancels ignore queries.

b) Run time

i) See Section 4.1

ii) *If the expression after which the {?} appears is an integer expression then its value is printed in the form

```
newline
print (value,1,0)
```

iii) *If the expression after which the {?} appears is a real expression then its value is printed in the form

```
newline
print fl (value,10)
```

c) General

i) The query marker is initially set so that queries will be compiled unless instructions are given to the contrary.

Section 3.17 Source Statement {ignore queries}[S]

a) Compile time

i) Cancels compile queries.

ii) Consequently queries are no longer compiled into the program.

b) Run time

i) See Section 4.1

c) General

i) See Section 3.16.

* On the Manchester compilers the format is different. See Manchester Reference Manual.

Section 3.18 Source statement {comment} [TEXT] [S]
or {|} [TEXT] [S]

a) Compile time

b) Run time

c) General

No action.

Section 3.19 Source statement {upper case delimiters} [S]

a) Compile time

Beginning with the line following the line in which this statement appears, every upper case letter in the program (but not in the data, if any) is replaced by the corresponding underlined lower case letter.

b) Run time

c) General

i) From the line following the line in which this statement appears, delimiters may be typed in upper case letters instead of being underlined.

ii) Upper case letters may not now appear in names, since a statement such as

```
        Fred = Jim + Albert
```

will be treated as

```
        fred = jim + albert
```

which is not a legal statement of the language.
Notice also that, for example,

```
        print symbol('A')
```

will be treated as

```
        print symbol('a')
```

iii) The effects of this statement are cancelled by

{normal delimiters} [S]

which statement also operates from the line following the line in which it appears.

Section 3.20 Source statement {normal delimiters} [S]

a) Compile time

Cancels {upper case delimiters} [S] (see section 3.19).

b) Run time

c) General

<u>Section 3.20.1</u>   Source statement <u>fault</u> [FAULT LIST] [S]

a) <u>Compile time</u>

i) The labels referred to within the jump instruction following each [N-LIST] must obey the normal rules for jump labels. See section 4.3.

ii) This statement may only appear at the <u>basic</u> textual level of the program, i.e. within <u>begin</u> ..... <u>end of program</u>, but <u>not</u> within any other blocks or routines, otherwise it will be faulted.

b) <u>Run time</u>

i) Certain information concerning the current state of the program is stored so that if a fault, which has been allowed for, occurs subsequently, this information may be used to restart the program from the relevant label. I.e. The statement must be executed at some time during the normal flow of control of the program in order for any faults to be trapped thereafter.

c) <u>General</u>

i) For the faults which may be trapped and their corresponding numbers, which appear in the [N-LIST]s, see section 8.2.

ii) The label jumped to when any particular fault is trapped may be changed dynamically by executing a further fault statement in which the new label appears.

<u>Section 3.21</u> Source statement [S]

a) <u>Compile time</u>

b) <u>Run time</u>

c) <u>General</u>

No action.

Section 4 SEMANTICS (UNCONDITIONAL INSTRUCTIONS)

Section 4.1 Unconditional instruction [NAME] [APP] {=} [±'] [EXPR] [QUERY']

a) Compile time

i) If the NAME is of integer type and the RHS is of real type then a fault is recorded.

ii) NAME must be capable of being a destination for a value, i.e. a function or a routine name or switch is not allowed.

iii) If the query marker is set (see section 3.16 and 3.17) and [QUERY']={?} then instructions are compiled to print the value of the RHS.

b) Run time

i) The RHS is evaluated and the value is assigned to the destination given by the LHS.

ii) If a query printing instruction has been compiled the value of the RHS is printed.

c) General

<u>Section</u> 4.2 Unconditional Instruction [NAME] [APP]

a) <u>Compile time</u>

i) A check is made to ensure that [NAME] has been declared as a <u>routine</u>.

ii) Checks are made to ensure that the actual parameters in [APP] are consistent with the formal parameters of the <u>routine spec</u> for [NAME], both in type and number.

iii) If no faults have been found the <u>routine</u> call is compiled.

b) <u>Run time</u>

i) The actual parameters are evaluated and passed on for use by the <u>routine</u> body.

ii) The <u>routine</u> body is entered.

iii) When the <u>routine</u> body is left, control is returned to the instruction following the one in which [NAME] [APP] appears.

c) <u>General</u>

i) For rules regarding formal-actual parameter correspondence see reference (1) or (2).

ii) Similar rules apply to the passing of parameters to other RT types.

<u>Section</u> 4.3 Unconditional Instruction {->} [N]

a) <u>Compile time</u>

i) As the label corresponding to this jump may not yet be set in this block a jump to address zero is compiled and a record is made so that this jump instruction can be adjusted at the end of the block when the address of the label should be known.

b) <u>Run time</u>

i) The normal sequence of instructions is broken and the next instruction to be obeyed is taken at label [N] of the block in which this jump instruction appears.

c) <u>General</u>

i) Because of a)i) above, labels which are not set are not faulted until the end of the block in which the jump appears.

<u>Section</u> 4.4 Unconditional Instruction {->}[NAME]{(}[$\underline{+}$'][EXPR]{)}

a) <u>Compile time</u>

i) Unless [NAME] is a switch variable declared <u>in</u> the current block a fault is recorded.

ii) Unless [EXPR] is an integer expression a fault is recorded.

b) <u>Run time</u>

i) The expression is evaluated and a check is made to see that the value is within the bounds of the switch vector as declared and that a switch label exists corresponding to this value of the argument. If the label exists control is transferred to that label, otherwise the program is monitored.

c) <u>General</u>

i) A jump to a switch label has the property of an ordinary jump in that control can only be transferred within the same block level. The scope of a switch declaration extends over the block <u>in</u> which it appears but NOT over blocks within this block, unlike the scope of other declarations. Consequently a jump may not be made to a switch label in an enclosing block.

Section 4.5 Unconditional Instruction {caption}[TEXT]

a) Compile time

i) Erases, spaces and underlined spaces in the [TEXT] are ignored in the usual way.

ii) The symbols s̸ s⃒ s̸̲ s⃒̲ n̸ n⃒ ;̸ ;⃒ have special significance and are replaced in the output text as shown below

s̸ s⃒	become	space
s̸̲ s⃒̲		underlined space
n̸ n⃒		newline
;̸ ;⃒		;

iii) [TEXT] is terminated by ; or newline only.

b) Run time

i) The [TEXT] modified as described above is output.

c) General

i) In view of a)iii) above the instruction

<u>caption</u> ANSWER = 3 <u>if</u> x = 3

will always result in the following output

ANSWER=3<u>if</u>x=3

If this was intended to be a conditional instruction it should have been written

<u>if</u> x=3 <u>then</u> <u>caption</u> ANSWER = 3

iii) Note that <u>c</u> at the end of a line has its usual effect.

Section 4.6 Unconditional Instruction {return}

a) Compile time

i) A fault is recorded unless this instruction appears in a block delimited by routine .... end.

ii) This instruction may appear more than once in such a block.

b) Run time

i) This instruction is the dynamic end of a routine block and control passes back to the instruction following the routine call which caused the entry.

ii) The routine is regarded as a block and consequently on exit from this block all of the local working space is deallocated in the usual way.

c) General

i) The end corresponding to the routine heading is regarded as return; end so that an exit from a routine can be made by running onto its end.

<u>Section</u> 4.8 Unconditional Instruction {<u>stop</u>}

a) <u>Compile time</u>

b) <u>Run time</u>

  i) Execution ceases.

c) <u>General</u>

SEMANTICS (ARITHMETIC EXPRESSIONS)

Phrase [±'] [EXPR]

This combination of phrases is treated as either an integer expression or a real expression, depending upon the context.

INTEGER EXPRESSION	REAL EXPRESSION
a) Compile time	
(i) A check is made that all operands are integer operands (i.e. integer variables or integer functions valid at the current level, or integer constants).	(i) Operands can be either real or integer, except that the operand immediately following the operator † must satisfy the conditions for an integer expression.

(ii) If an operator is recognised as ø, this is replaced by a multiplication.

INTEGER EXPRESSION	REAL EXPRESSION
b) Run time	
(i) Integers are held in 48-bit fixed point form, and must therefore lie in the range -2^{47} to $(2^{47}-1)$.	(i) Real numbers are held in floating point form (one sign bit, 8-bit binary exponent, 39-bit mantissa). Integer quantities occurring in real expressions are immediately converted to floating point form, except where a sub-expression in brackets or modulus signs consists wholly of integer operands and the operators +, - and *. In this case the sub-expression is evaluated fixed point, and then converted.
N.B. Programmers are recommended to restrict their integers further to the range -2^{36} to $(2^{36}-1)$ so that their programs will also run on Atlas if required.	

(ii) Sub-expressions in brackets (or modulus signs) are evaluated first. After this, the order of precedence between operators is (highest precedence first):-

(a)	†	(exponentiation)
(b)	* and /	(multiplication and division)
(c)	+ and -	(addition and subtraction)

Where two adjacent operators are of equal precedence, operations are carried out from left to right.

INTEGER EXPRESSION	REAL EXPRESSION

(iii) An initial minus sign has the same effect as '-1*'(although executed more rapidly) and so takes precedence over all operators except ↑.

(iv) The program is monitored ('OVERFLOW REGISTER SET') if at any stage in the calculation a number exceeding the capacity of a 48-bit word is produced.

INTEGER EXPRESSION	REAL EXPRESSION
(v) Exponentiation is carried out by repeated multiplication. The program is monitored immediately ('ILLEGAL EXPONENT') if the exponent n lies outside the range $0 \leq n \leq 63$. (Note that smaller exponents may also cause a monitor signal by setting the overflow register while attempting to execute, for example, 2↑50).	(v) Exponentiation is carried out by repeated multiplication. The program is monitored immediately ('ILLEGAL EXPONENT') if the exponent n lies outside the range $-255 \leq n \leq 255$. If n is negative, x↑n is evaluated as 1/x↑|n|.

(vi) 0↑0 gives the result 1 (as, of course, does any other quantity raised to the power zero).

INTEGER EXPRESSION	REAL EXPRESSION
(vii) The program is monitored whenever a division results in a non-integral quotient. Note the significance of the order in which operations are carried out in the following examples in which i is an integer:- i=(i+1)/2*i (fails if i is even) i=i*(i+1)/2 (valid for all values of i)	

(viii) An attempt to divide by zero causes the program to be monitored.

c) General

<u>Section 6</u> <u>SEMANTICS (CONDITIONAL PHRASES)</u>

Phrase [COND]

a) <u>Compile time</u>

Any constituent expression which contains

i) a real variable, real function or real constant

or ii) either of the operators / or ↑

is compiled as a real expression. All other expressions are compiled as integer expressions.

b) <u>Run time</u>

i) [±'] [EXPR] [COMP] [±'] [EXPR]

The left-hand expression is evaluated first, followed by the right-hand one. If one is an integer expression and the other real, the former is converted to floating point form before comparison.

ii) [±'] [EXPR] [COMP] [±'] [EXPR] [COMP] [±'] [EXPR]

The first two expressions are evaluated and compared. The third expression is only evaluated if the required condition between the first two is satisfied. Conversion to floating point form is carried out if necessary as in (i).

iii) [SC] <u>and</u> [AND-C]

[SC] <u>or</u> [OR-C]

The conditions are tested from left to right, stopping as soon as sufficient information is obtained to give the overall verdict true or false.

c) <u>General</u>

Section 7 PERMANENT ROUTINES AND FUNCTIONS

routine spec read ([TYPE] name x1, [TYPE] name x2,.....)

where [TYPE] is integer or real.
This takes the next number from the input data tape and stores it in x1, takes the next number and stores it in x2 and so on for each parameter. The numbers may be in either fixed or floating point form. A fault is signalled if the number assigned to an integer variable is not integral and also if any characters other than digits, +, -, α, . occur. This routine is unique in allowing a variable number of parameters.

routine spec print (real x, integer m,n)

This prints the value of x on the output medium in fixed point form with m digits before the decimal point and n digits after. Insignificant leading zeros are replaced by spaces and any minus sign right justified. If more than m significant figures occur before the decimal point, the point will be displaced to the right and the extra digits inserted.

routine spec print fl (real x, integer n)

Prints the value of x on the output medium in floating point form, standardised in the range $1 \leq x < 10$ with n digits after the decimal point.

routine spec read symbol (integer name n)

Reads the next alpha-numeric symbol (simple or compound) from the input data tape, stores it in the specified integer location, and moves onto the following symbol.

integer fn spec next symbol

Gives the value of the next symbol (simple or compound) read from the data tape, without moving on to the following symbol. In other words, the same symbol can be obtained again by either a 'read symbol' or a 'next symbol' instruction.

routine spec skip symbol

Passes over the next symbol from the data tape to the following one without reading.

routine spec print symbol (integer n)

Prints on the output medium the symbol specified by the integer value n.

<u>routine spec</u> read binary (<u>integer name</u> n)

Reads the next 7-bit row of holes from the data tape as a binary number in the range 0-127 (with the tape so oriented that the sprocket hole comes between the digits of value 4 and 8) and places it in the specified location.

<u>routine spec</u> punch binary (<u>integer</u> n)

Punches, as a row of holes on the output tape the seven least significant binary digits of the <u>integer</u> value specified by n.

<u>routine spec</u> newline

Either punches a newline character on the output paper tape, or resets the carriage of the line printer to a newline, if this device is selected.

<u>routine spec</u> newlines (<u>integer</u> n)

Punches the number of newline characters specified by the <u>integer</u> value n on the output paper tape, or resets the carriage and that number of 'line feed's when the line printer is selected.

<u>routine spec</u> space

Either punches a space character on the output paper tape or advances the carriage of the line printer one position.

<u>routine spec</u> spaces (<u>integer</u> n)

Punches the number of space characters specified by the <u>integer</u> value n on the output paper tape, or advances the carriage of the line printer that number of positions.

__routine spec__ tab

Punches a tab character on the output paper tape. If the line printer is selected as the output device, this is equivalent to a 'spaces (8)' instruction.

__routine spec__ tabs (__integer__ n)

Punches the number of tab characters specified by the __integer__ value n on the output paper tape. If the line printer is selected this is equivalent to 'spaces (8*n)'.

__routine spec__ run out (__integer__ n)

Punches n runout characters on the output paper tape. If the line printer is selected this is equivalent to a 'newpage' instruction.

__routine spec__ newpage

Causes the line printer to move to a new page of paper. If paper tape is selected, 30 newline characters are punched.

__routine spec__ stop code

Punches the 'stop code' character on the output paper tape. (This causes the Flexowriter to stop whenever it is read.)

__routine spec__ colour change

Punches the 'colour change' character on the output paper tape. (This causes any future printing on the Flexowriter to be in red if it was printing in black and vice versa.)

<u>integer fn spec</u> int pt (<u>real</u> x)

Gives the value of the integral part of the quantity specified by x.

<u>integer fn spec</u> int (<u>real</u> x)

Gives the value of the nearest integer to the quantity specified by x , i.e. int pt (x + .5).

<u>real fn spec</u> fracpt (<u>real</u> x)

Gives the value of the fractional part of the quantity specified by x.

<u>integer fn spec</u> parity (<u>integer</u> n)

Gives the value of (-1) raised to the power of the quantity specified by the <u>integer</u> value n.

<u>integer fn spec</u> addr ([TYPE] <u>name</u> x)

where [TYPE] is <u>integer</u> or <u>real</u>. It takes the value of the address of the named location.

<u>integer map spec</u> integer (<u>integer</u> s)

Specifies the <u>integer</u> location whose address is the quantity given by the value s.

<u>real map spec</u> real (<u>integer</u> s)

Specifies the <u>real</u> location whose address is the quantity given by the value s.

Section 8 DIAGNOSTIC MESSAGES

COMPILE TIME FAULTS

1. TOO MANY REPEATS
2. LABEL SET TWICE
3. spec FAULTY
4. SWITCH VECTOR NOT DECLARED
5. SWITCH LABEL ERROR
6. SWITCH LABEL SET TWICE
7. NAME SET TWICE
8. TOO MANY PARAMETERS IN RT DESCRIPTION
9. PARAMETER FAULT IN RT DESCRIPTION
10. TOO FEW PARAMETERS IN RT DESCRIPTION
11. LABEL NOT SET
12. NO TYPE DELIMITER IN RT DESCRIPTION
13. repeat MISSING
14. TOO MANY ends
15. TOO FEW ends
16. NAME NOT SET
17. NOT A RT NAME
18. SWITCH VECTOR ERROR
19. WRONG NUMBER OF PARAMETERS
20. SWITCH VECTOR IN EXPRESSION
21. RT TYPE NOT YET SPECIFIED
22. ACTUAL PARAMETER FAULT
23. routine NAME IN EXPRESSION
24. REAL QUANTITY IN INTEGER EXPRESSION
25. cycle VARIABLE NOT AN INTEGER VARIABLE
26.
27. SWITCH VECTOR INSIDE OUT
28. RT TYPE NOT DESCRIBED
29. LHS NOT A DESTINATION OR NAME IS NOT AN ADDRESS
30. return OUT OF CONTEXT
31. result OUT OF CONTEXT
32. MACHINE CODE NOT SWITCHED ON
33. PRIVATE LABEL SET TWICE
34. TOO MANY LEVELS: TEXTUAL LEVEL > 9
35. TOO MANY LEVELS: TEXTUAL LEVEL > 15
36.
37.
38. JOB HEADING OUT OF CONTEXT
39. REAL QUANTITY AS EXPONENT

RUN-TIME MONITOR MESSAGES

1. OVERFLOW SET
2. OVERFLOW SET
3. NON-INTEGRAL CYCLE
4. EXCESS BLOCKS
5. SQRT OF NEGATIVE ARGUMENT
6. LOG OF NEGATIVE ARGUMENT
7. SWITCH VARIABLE NOT SET
8.
9.* INPUT ENDED
10. NON-INTEGRAL QUOTIENT
11. RESULT NOT SPECIFIED
12.* EXECUTION TIME EXCEEDED
13.* PROGRAM TERMINATED BY SUPERVISOR
14. SYMBOL .... IN DATA
15. MORE THAN 3 SYMBOLS IN ONE PRINTED POSITION IN DATA
16. REAL INSTEAD OF INTEGER IN DATA
17.
18. PARITY FAULT IN DATA
19. UNASSIGNED CHARACTER IN DATA
20. MORE THAN 120 CHARACTERS ON A LINE IN DATA
21. ILLEGAL EXPONENT
22. TRIG FUNCTION INACCURATE
23. TAN TOO LARGE
24. EXP TOO LARGE
25. ARCTAN (0,0)
26. INT TOO LARGE
27. INTPT TOO LARGE
28. ARRAY INSIDE OUT
29.
30.
31.
32.

* THESE FAULTS MAY NOT BE TRAPPED.

Section 9

JOB HEADINGS

Control statement {*} {*} {*} {A} [S]

a) Compile time

All lines from ***A to COMPILER AA or AB are regarded as forming the Job Heading, and recognised independently of the main phrase structure definitions. The aim is to make tapes prepared for Atlas and KDF9 interchangeable wherever possible, although rather more freedom is allowed in the case of KDF9.

All lines of the Job Heading are disregarded unless they start with one of the key words given below or are on the lines following JOB or TAPE :-

i) JOB

The line immediately following JOB gives the title of the program, of which the first 60 characters are printed out at the head of the program map and also stored away for further use at run time.

ii) EXECUTION

This must be followed by the overall time limit for the job, in minutes or seconds.

e.g. EXECUTION 15 SECONDS

or EXECUTION 3.25 MINUTES

This time covers input of data, computing and output. If this item is not included the time allowed is 2 minutes.

iii) TAPE

The integer following on the same line, or the start of the next, must be an integer, n, in the range $1 \leq n \leq 8$, giving the magnetic tape channel number followed by the 8 characters of the corresponding tape identifier. If writing to this tape is intended, the tape identifier must be followed by * WITH WRITE PERMIT.

```
e.g.        TAPE
            1 DG720006
            TAPE
            5 DG720007* WITH WRITE PERMIT
```

On KDF9, but not yet on Atlas, the above can be contracted to

```
            TAPE
            1 DG720006
            5 DG720007* WITH WRITE PERMIT
```

iiia) OUTPUT

On the same line, or the start of the next, must be the integer 0, indicating channel 0 (unlike Atlas, where multi-channel output is possible). Following this on the same line is the output device required, which may be

either	LINE PRINTER
or	SEVEN-HOLE PUNCH

In the case of the line printer being selected, this should be followed by a number indicating the number of lines of of output to be allowed.

```
e.g.    OUTPUT
        0 LINE PRINTER 400 LINES
```

For the paper tape punch, the number of blocks (one block consisting of 4096 characters) of output to be allowed should follow.

```
e.g.    OUTPUT
        0 SEVEN-HOLE PUNCH 5 BLOCKS
```

If the OUTPUT section is omitted, the output will automatically be sent to the line printer, when 200 lines of output are allowed, or, if a printer is not available, to the paper tape punch, when 1 block is allowed.
If the limit of output is exceeded then the program is terminated.

iv) PARAMETER

This must be followed by * and an eight digit constant (distinguishing the individual programmer) which will be used as a security check to prevent overwriting one anothers' sections on tape.

e.g. PARAMETER *23127643

v) COMPILER AA or COMPILER AB

This terminates the job heading and the next line becomes line 0 in the program map.

b) Run time

The title line is reproduced at the end of the output.

c) General

Spaces are disregarded throughout the Job Heading, except in the title line and the constant giving the Execution time.

The program will be monitored if:-

i) ***A appears anywhere other than the beginning of the program.

or ii) COMPILER AA or AB has not been found within the first 20 lines after ***A.

or iii) The symbols following EXECUTION do not form a legal constant.

Any other fault in the job heading simply causes the line concerned to be disregarded.

EDINGBURGH UNIVERSITY ATLAS AUTOCODE : VERSION I

NORMAL OPERATING INSTRUCTIONS (dated 15/2/66)

(1) Fit 7-hole plug if using 7 track width tape.
Load 1184 word non-time sharing director program if system is being run on a 4 module machine.

(2) Load magnetic tapes

(a)	DG72I	without write ring
(b)	ZERO tape	with write ring.

(3) Run binary call tape DE72CPR/AA/I which asks you to type in:-

```
[m]  START OF AA RUN
[q]  COMPILER TAPE;DG72I       .->
[q]  CPR WORK TAPE;            .->  (8 spaces, means use ZERO tape)
[q]  READER;Y.->                    (means Yes, a tape reader is
                                                  available)
[q]  PUNCH;Y.->                     (means Yes, a paper tape punch
                                                is available)
[q]  LINE PRINTER;Y.->              (means Yes, a line printer is
                                                  available)
```

The particular output device is then chosen automatically by each individual program. If any of these peripheral devices are not available, reply N.-> to the question, meaning No, that device is not available. Details of these cases may be found in the 'Special Actions' section, (f)-(i). If the reply is neither Y.-> nor N.->, the question will be repeated.

(4) Feed in programs, followed by their data (if any) one after the other. This compiler automatically skips data if not required (but see special warning about binary data). The monitor will print:-

```
[m]   PROGRM hh.mm.ss    (when looking for the next program)
[m]   ENTERED            (when compilation is successful).
```

Where two or more programs are supplied on one tape, the compiler automatically runs on from one program to the next.

(5) At the end of run TINT; A.->

SPECIAL ACTIONS : VERSION I

(a) Reader Disengages (other than at end of tape)

Look at monitor

If it says	RELOAD
	PROGRM hh.mm.ss

pull the tape back about 6ins to 12 ins (to 'runout' between programs) and re-engage. Otherwise re-engage without moving the tape.

(b) Failure OON, OOT, OOL, LIV, etc.

```
REACT;IO.->
```

(c) Job Appears Overdue (See EXECUTION TIME given on box)

```
TINT;I1.->
```

Monitor will either print

TIME OK	(if within Execution time) or
PROGRM hh.mm.ss	(automatically killing that job and entering the next).

(d) To Kill a Job Which Appears Out of Control

```
TINT;IO.->
```

(e) Other On-Line Messages

AA PAR) AA SUM)	Faulty Mag. Tape Station with Compiler Tape. Try another deck.
AA FAIL)	Faulty Mag. Tape Station with ZERO Tape. Try another deck.

(f) Paper Tape Reader Not Available

To operate the system, the input must previously have been recorded on magnetic tape in the standard Atlas Autocode format (i.e. 30 word blocks suitable for character reading). The response to the query in this case will be

```
[q]  READER;N.->          (means No, reader is not available)
```

whereupon the identifier of the mag. tape on which the input is recorded must be typed in, in response to:-

```
[q]  INPUT MAG TAPE;????????.->
```

Then proceed as normal.

(g) Paper Tape Punch Not Available

In response to the punch query type in:-

```
[q]  PUNCH;N.->           (means No, punch is not available).
```

All the output will now go to the line printer.

(h) Line Printer Not Available

In response to the line printer query, type in

```
[q]  LINE PRINTER;N.->  (means No, line printer is not available).
```

All the output will now go to the paper tape punch.

(i) Neither Paper Tape Punch nor Line Printer Available

In response to the two queries, type in:-

```
[q]  PUNCH;N.->
[q]  LINE PRINTER;N.->
```

The output may now be dumped onto magnetic tape, the identifier of which must be typed in, in response to:-

```
[q]  OUTPUT MAG TAPE;????????.->
```

Then proceed as normal.

SPECIAL JOBS : VERSION I

(a) Jobs with Binary Data

These jobs will be specially marked on the box, and the data tape will be separate from the program.

(i) Feed in program. At end, look at monitor.

(ii) (a) If reply is ENTERED, feed in data tape.

(b) If reply is PROGRM, do not feed in data tape, but go on to next program.

(b) Jobs Using Extra Magnetic Tapes

If extra mag. tapes (in addition to the standard Compiler Tape and ZERO tape) are required, this will be marked in the job heading stuck on the outside of the box, in the form

TAPE

1 DG720004*WITH WRITE PERMIT

Tape Identifier   Write ring required.

If, '*WITH WRITE PERMIT' is omitted, do NOT fit the write ring. This extra tape(s) can be dismounted when the program ends and another PROGRM is called.

$\underline{\text{Section 11}}$ $\underline{\text{USING KDF9 FLEXOWRITERS}}$

As the keyboard is different and no backspacing is possible on KDF9-coded flexowriters, certain symbols required in Atlas Autocode are punched as follows:-

$\underline{\text{Atlas}}$	$\underline{\text{KDF 9}}$
'	$\underline{[}$ or $\underline{]}$
|	$\underline{(}$ or $\underline{)}$
?	$\underline{\uparrow}$
π	£
s⃒ or s̸	$\underline{*}$
$\underline{\text{s⃒}}$ or $\underline{\text{s̸}}$	$\underline{*}$
n⃒ or n̸	$\underline{£}$
;⃒ or ;̸	$\underline{;}$
^{2} (superscript)	$\uparrow$ 2
$\frac{1}{2}$	.5
α	$_{10}$(subscript)

$\underline{\text{For example}}$

$\underline{\text{if}}$ i = 'a' $\underline{\text{then}}$ $\underline{\text{caption}}$ n⃒ n⃒ FAULT s⃒ in s̸ DATA n̸
x = $\frac{1}{2}$ π * |x + y| ?
| rubbishy bit of program

$\underline{\text{can be punched}}$

$\underline{\text{if}}$ i = $\underline{[}$a$\underline{]}$ $\underline{\text{then}}$ $\underline{\text{caption}}$ $\underline{£}$ $\underline{£}$ FAULT $\underline{*}$ in $\underline{*}$ DATA $\underline{£}$
x = .5 £ * $\underline{(}$x + y$\underline{)}$ $\underline{\uparrow}$
$\underline{(}$ rubbishy bit of program

$\underline{\text{Notes}}$ 1. The jump instruction e.g. -> 7 is punched with - (minus) followed by > (greater than) and $\underline{\text{not}}$ the 'end message' symbol.

2. $\underline{\div}$ (underlined divide) is an illegal symbol, even in captions and comments, as it is used to terminate the KDF9-tape input.

Output to KDF 9 Line Printers

Here, only upper case letters can be printed, and no underlining is possible.

a, A, $\underline{a}$ and $\underline{A}$	are all printed	A
;	is printed	;

but all other symbols produced by backspacing on the Atlas-coded flexowriter are printed with each component of the symbol occupying a separate character position.

E.g. $\neq$ will appear /=

Any single symbol not in the printer character set will be printed as %.

Section 13 LIBRARY PACKAGE SCHEME

13.1 Purpose of the Library Package Scheme

Because the storage available on KDF 9 for program and data is considerably less than that available on Atlas it is not considered proper to use a large part of the store for permanent routines and functions. Some of the more fundamental permanent routines and functions have been provided within the compiler system but others must be provided in other ways. As an immediate means of providing convenient use of library routines not included in the permanent material a library package scheme has been introduced. This may be replaced at some later date by some other library program scheme but conventions have been chosen which will involve no change of programs in the future should such a scheme be introduced.

13.2 Library Packages

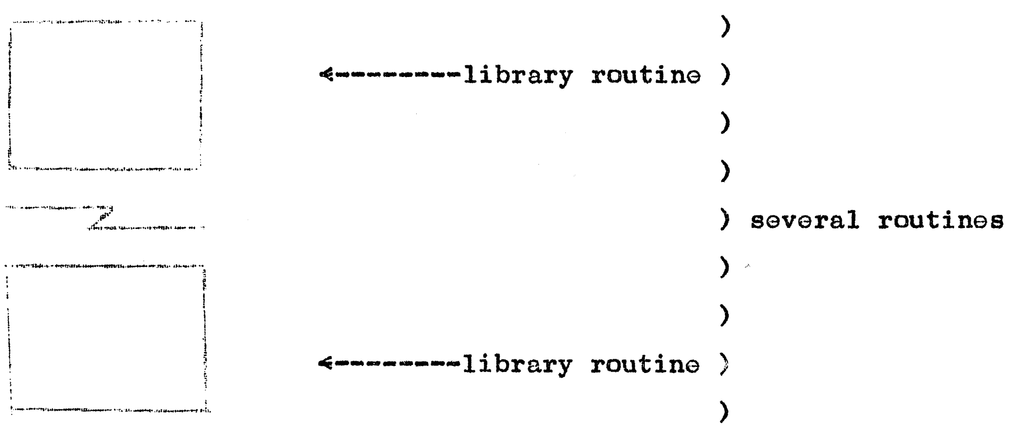
A library package is a small collection of routines or functions on paper tape. The routines and functions will usually be members of a set of similar or associated processes, e.g. packages for matrix operations and magnetic tape operations have already been provided. The library package is read into the computer immediately before the user's program and becomes part of the permanent material for the duration of this one user program only.

13.3 Structure of Library Packages

A library package is provided on paper tape and has the following form:-

```
***A                                    ) JOB HEADING
JOB                                     )     OF
TITLE OF LIBRARY PACKAGE AND DATE       ) LIBRARY PACKAGE
COMPILER AA                             )
```

<u>mcode</u>



<u>end of perm</u>

13.4 Use of Library Packages

13.4.1 The user must obtain a copy of the library package from his computer unit. This he should mark as 'tape 1'. His own program will then be 'tape 2' and he should clearly mark on the box in which he sends his program for running 'Two tapes marked tape 1 and tape 2 to be read in order'.

13.4.2 His main program (tape 2) will require a special job heading in the following form:-
(e.g. in the case of the magnetic tape package being tape 1)

```
|INSERT MAGNETIC TAPE PACK; ***A
JOB
etc. (in the usual way)
```

13.4.3 Apart from providing tape 1 (the library package) and using the special form of job heading described above the user has no restrictions to remember. (but see 13.5 and 13.6)

13.4.4 Within his program the user must treat the library routines as if they were part of the permanent material. He must therefore use them without specification or description in the same way that he uses sin, cos, print etc. without specification or description.

13.4.5 At compilation a program map is first produced for the library package and is followed by the user's program map in the usual way.

13.5 Use of Two or More Packages at the Same Time

If a user requires to use two or more packages with the same program then he must modify the job headings of the second and subsequent packages in the following way.
E.g. if the second package is the matrix pack its job heading must be changed from

```
***A
JOB
MATRIX PACK - DATE
COMPILER AA
```

to:

```
|SECOND PACK; ***A
JOB
MATRIX PACK - DATE
COMPILER AA
```

In fact only the first library package will start with the normal

```
***A
JOB
etc.
```

the subsequent packages and the main program starting with the special form

```
|SOME COMMENT; ***A
```

13.6 Cautions

13.6.1 The special form of job heading described above is required to force the ***A marker off the left margin. On KDF 9 this prevents it giving INPUT ENDED. LIBRARY PACKAGES CAN NOT BE USED ON ATLAS IN THIS FORM.

13.6.2 Anyone wishing to produce a library package should consult his computer unit for advice.

13.6.3 If a program fails in a library package and is MONITORED the line number given as the STOPPING LINE is the line number in the main program and this line will of course contain a call on the library routine or function in which failure was detected. Hence library packages have the same status as permanent routines or functions.

Section 14 Magnetic tape on KDF9

14.0 Introduction

Magnetic tape facilities are now available on the KDF9 at Glasgow and Newcastle Universities. A number of routines have been written for such standard tasks as filing data or programs on tape; copies of these routines, which are described below, may be obtained from the Computer Unit.

14.1 Security System

14.1.1 The magnetic tape routines use a tape which has been pre-addressed in sections of 512 words. Particular sections are allocated to particular users, and there is a security system in operation to ensure that no user has access to any other user's sections.

14.1.2 Any user wishing to have magnetic tape sections allocated to him should complete a Magnetic Tape Allocation Request Card, and return it to the Computer Unit. He will be required to state how many sections he needs, and whether he wants sections at Newcastle or Glasgow. The Unit will then take steps to allocate the sections required, and to inform the user of this allocation by a letter of the following form:

Dear ...,

You have been allocated 100 sections numbered 1 to 100 on a magnetic tape held at Glasgow University Computing Laboratory.

Tape Identifier	DG720002
Parameter	*01026561

Signed ...

14.1.3 The following points should be noted:

a) The sections allocated to a particular user will invariably be numbered from 1 upwards. It is the task of the security system to ensure that the correct physical sections on the magnetic tape are used.

b) Every reel of magnetic tape has a unique identifier consisting of eight letters or digits. In the example above the user has been allocated sections on tape DG720002. This identifier must be quoted in the job heading of any program requiring to use that tape (see 14.2 below).

A PROGRAM REQUIRING TO USE A TAPE HELD, FOR EXAMPLE, IN GLASGOW MUST BE CLEARLY LABELLED TO ENSURE THAT IT IS NOT SENT ELSEWHERE.

c) Every user will be assigned a security parameter when he first applies to have sections allocated. In the example above, the user has been assigned the parameter *01026561. A parameter will consist of an asterisk followed by eight digits, and must be quoted in the job heading of any of that user's programs which require to use magnetic tape (see 14.2 below).

If a user has been allocated sections on more than one magnetic tape, he will use the same security parameter to refer to all of them. However, two sets of sections on the same tape cannot be referred to by the same parameter.

14.1.4 The situation will occasionally arise where one user with one security parameter wishes to hand over data to another user with a different security parameter. As the sections allocated to two different users cannot normally be accessed by the same program, special arrangements have to be made in this case. The users concerned should consult the Computer Unit.

14.2 Job Headings for Magnetic Tape Jobs

14.2.1 General rules for job headings will be found in Section 9.

14.2.2 For a magnetic tape job, the first line of the job heading, namely

```
***A
```

must be replaced by

```
| PRECEDE BY MAGPACK ; ***A
```

14.2.3 The user's security parameter must be declared in the job heading by a line such as

```
PARAMETER *01026561
```

14.2.4 Each tape which the program may use must be declared in the job heading along with its logical channel number. For example, if the program needs to refer to the tape DG720002 as tape channel 4, say, then the following line must be added to the job heading:

```
TAPE 4 DG720002
```

Moreover, if the program will cause writing onto this particular tape, then the words *WITH WRITE PERMIT must be added, making the complete line

```
TAPE 4 DG720002* WITH WRITE PERMIT
```

If these words are not added, then the security system will not allow writing to this tape.

Exceptionally, a programmer using machine code may declare, for instance

```
TAPE 3 ZEROTAPE*WITH WRITE PERMIT
```

with the obvious meaning, but it should be noticed that the standard magnetic tape routines cannot now be used on this channel.

A logical channel number n, say, must be in the range $1 \leq n \leq 8$ (see also 14.4.1.1 below).

14.3 Running a Magnetic Tape Job

14.3.1 The magnetic tape routines are available on paper tape, and must currently be compiled immediately before the job heading (modified as described above) of the user's program. Clearly this may be achieved either by copying the routines onto the beginning of the program tape, or by labelling the magnetic tape routines TAPE 1 and the program TAPE 2 and instructing the operators to run the tapes in succession.

14.3.2 The program map produced by the compiler will have the following structure:

```
MAGNETIC TAPE ROUTINES 5/2/66
          2     ROUTINE claim tape       ) program map of the
                  .                      )
                  .                      )
        357     END OF ROUTINE           ) mag. tape routines
UEDIN, BLOGGS OGPU, TITLE OF USER'S PROGRAM
          0     BEGIN                    ) program map
                  .                      ) of the
        672     END OF PROGRAM           ) user's program
```

14.3.3 A copy of the program's job heading should be attached to the outside of the box in which it is packed, so that the operators can see which tapes need to be mounted. As a double check on the security system, the operators will not normally mount a reel of magnetic tape with a write-ring unless the tape is marked *WITH WRITE PERMIT in the job heading.
All magnetic tape jobs must be clearly marked with the name of the computing laboratory where they are to be run.

14.4 The Magnetic Tape Routines

The magnetic tape routines are supplied as a package which must be compiled immediately before the user's program as described in 14.3.1 above. When this is done, these routines become to all intents and purposes extra permanent routines of the compiler, which the program may call without either a specification or description.

The routines in the package are described below.

14.4.1 <u>routine</u> claim tape (<u>integer</u> channel)
Before any other operations can be performed on a magnetic tape, the tape must be claimed. The parameter 'channel' refers to the logical channel number assigned in the job heading. For example, if the line

TAPE 4 DG720002

appears in the job heading, then the instruction claim tape (4)

will make DG720002 available for other operations on channel 4.
The Director program will if necessary instruct the operator to mount the required tape, and the tape will then be positioned at the beginning of the sections allocated to the user with the declared security parameter.
The following monitors may occur, all of which are followed by termination of the program:

TAPE CLAIMED TWICE n

The tape on Channel n has been claimed before and has not yet been released.

TAPE CHANNEL NOT DEFINED n

Either n does not lie between 1 and 8, or else no line of the type

TAPE n DG720002

appeared in the job heading.

ILLEGAL TAPE CLAIM n

No sections on the tape on channel n have been allocated to the user with the declared security parameter.

14.4.1.1 Notice that it is legitimate (* not on Atlas) to declare the same tape on two channels in the job heading, for example by

TAPE 4 DG720002
TAPE 5 DG720002

provided that no attempt is made to claim channels 4 and 5 simultaneously. If such an attempt <u>is</u> made, the program is monitored

PROGRAM TERMINATED BY SUPERVISOR.

14.4.2 <u>routine</u> release tape (<u>integer</u> channel)

The tape on the logical channel designated by the parameter is released, that is to say, no more operations may be performed upon it by the program unless it is subsequently reclaimed (by a 'claim tape (n)' instruction) either on the same or on a different logical channel.
It is not good practice to alternate 'claim tape' and 'release tape' instructions unnecessarily, as every time a tape is released it is rewound to the beginning of the reel, with a consequent loss of time if it later has to be re-positioned at the user's sections.
Any tapes not specifically released by the program will be released automatically whenever the program stops or is monitored.
The following monitors may occur, both being followed by termination

of the program:

ILLEGAL TAPE RELEASE n

The tape on channel n was not claimed, and therefore may not be released.

TAPE CHANNEL NOT DEFINED n

n does not lie between 1 and 8.

14.4.3 routine write to file (integer channel, integer name section, c
addr start, finish)

'start' and 'finish' will normally be elements of the same array, so that, for example

```
   i = 1 ; write to file (4, i, A(1), A(800))
```

will write the 800 words of store A(1) to A(800) onto the tape on logical channel 4, starting at section i = 1. Words are written to magnetic tape only in complete 512 word blocks, so that, in this instance, A(1) to A(512) will be written to section 1 of the tape, and A(513) to A(800) and 224 words of rubbish will be written to section 2. This implies that all the information previously in section 2 has been lost, even though on this occasion we only really needed to use the first 288 words of the section. The integer name parameter 'section' is set on exit from the routine to the number of the last section written into, i.e. in the example we are considering, on exit from the routine i will be equal to 2.

Suppose an array was declared by

```
         array B(1:10, 1:20)
```

Then we may write the whole array to tape by

```
         write to file (4, i, B(1,1) B(10,20))
```

and similarly for arrays of more than two dimensions. However, if an instruction such as

```
         write to file (4, i, B(2,3), B(8,17))
```

were written, then the effect is not defined. Similarly instructions such as

```
         write to file (4, i, A(1), C(40))
```

should be avoided.

The following monitors may occur, all of which are followed by termination of the program:

TAPE CHANNEL NOT DEFINED n

n lies outside the range 1 to 8.

WRITE ON UNCLAIMED CHANNEL n

NO WRITE PERMIT n

The words *WITH WRITE PERMIT did not appear when the channel n was defined in the job heading.

TAPE WRITE INSIDE OUT n

addr(start) > addr(finish). For example,

```
         write to file (4, i, A(100), A(1))
```

would be monitored in this way.

TAPE WRITE OFF LIMITS n

The program is monitored if a user who has been allocated, for example, 50 sections on channel n attempts to address sections numbered 51 or above ; also any attempt to address a section with a negative or zero address is monitored.

EXCESS BLOCKS DURING WRITE TRANSFER n

Suppose a long program occupies so much space that the last array declared, Z(1:100), say, extends to within a few locations of the end of the core store. Then a routine call such as

```
        write to file (4, i, Z(1), Z(100))
```

will be monitored if there are less than 512 locations between Z(1) and the end of the store inclusive. This is because all write-transfers are made in blocks of 512 words, and it would not be possible in such a case to perform the transfer correctly. Clearly this problem will only arise with a program which uses almost all the capacity of the machine.

14.4.4 <u>routine</u> read from file (<u>integer</u> channel, <u>integer name</u> <u>c</u> section, <u>addr</u> start, finish)

Similar to 'write to file', section 14.4.3, with the following provisos:

Transfers are <u>not</u> necessarily made in 512 word blocks, so that, for instance,

```
        read from file (4, i, A(1), A(800))
```

will overwrite only the designated area of core-store, and will <u>not</u> spoil any information which may be held immediately above A(800).

The parameter 'section' is set on exit from the routine to the number of the last tape section read.

The following monitors may occur:

TAPE CHANNEL NOT DEFINED n

READ ON UNCLAIMED CHANNEL n

TAPE READ INSIDE-OUT n

TAPE READ OFF LIMITS n

Cf. section 14.4.3.

14.4.5 <u>routine</u> file this program (<u>integer</u> channel, section, marker)

A record of the current state of the program is dumped onto the magnetic tape on the indicated logical channel, beginning at the indicated section and using as many other sections in sequence as

are necessary (never more than 32).
The channel must previously have been claimed in the usual way, and moreover the channel must have been designated *WITH WRITE PERMIT in the job heading.
A message of the following form is output on completion of the transfer to tape:

```
PROGRAM STARTS SECTION 25 ON CHANNEL 4
PROGRAM ENDS SECTION 41
```

If the parameter 'marker' is zero, then the program will stop on completion of the transfer to tape and the associated message; otherwise execution of the program continues in the normal way.
A program recorded on magnetic tape in this way can be re-entered by use of the routine 'enter filed program' described below.
An attempt to file a program may be monitored for any of the reasons listed in section 14.4.3.

14.4.6 routine enter filed program (integer channel, section)

If a program has been previously filed starting at the designated section on the designated channel, then it is brought down into the store, and execution of it continues from a point immediately following the relevant call of 'file this program'.
For instance, in the simplest case, suppose we have a large program which will be used many times. Then the following arrangement will save continual recompilation of the complete program:
a) compile (not forgetting the magnetic tape package) and run

```
| PRECEDE BY MAGPACK ; ***A
JOB
UEDIN, BLOGGS OGPU, LONG PROGRAM
PARAMETER *01026561
TAPE 1 DG720002* WITH WRITE PERMIT
COMPILER AA
begin
claim tape (1)
file this program (1, 1, 0)
comment re-enter at this point
...                        ) rest of program
end of program
***Z
```

b) Subsequently, whenever this program is needed, compile and run

```
| PRECEDE BY MAGPACK ; ***A
JOB
UEDIN, BLOGGS OGPU, RECALL LONG PROBLEM
PARAMETER *01026561
TAPE 1 DG720002
COMPILER AA
begin
claim tape (1)
enter filed program (1, 1)
end of program
Data for this run if any
***Z
```

Notice that the tape on channel 1 must be claimed in the usual way before an attempt is made to enter a program kept on it.

The call of 'file this program' may be anywhere in the program, execution carrying on, when the program is re-entered by 'enter filed program', exactly where it left off. The contents of all variables and arrays will be exactly the same as they were when the program was suspended. Similarly, for the benefit of machine code programmers, the contents of the Q-stores, the nesting store, and the SJNS will all be unaltered.

In fact, for many purposes, a good rule-of-thumb is to regard the call of 'file this program' and the corresponding call of 'enter filed program' as having no effect on the program whatsoever except to allow an arbitrary time to elapse before execution is completed.

The exceptions to this statement are noted below.

14.4.6.1 Suppose the following program is run:

```
|PRECEDE BY MAGPACK ; ***A
JOB
UEDIN, BLOGGS OGPU, FILING PROGRAM
PARAMETER *01026561
TAPE 1 DG720002* WITH WRITE PERMIT
TAPE 2 DG720003
TAPE 3 DG720004
EXECUTION 5 MINUTES
OUTPUT 0 LINE PRINTER 1000 LINES
COMPILER AA
begin
integer n
claim tape (1)
 claim tape (2)
 file this program (1, 25, 0)
 read (n)
```

```
        print (n, 1, 0)
        comment rest of program
        end of program
        17
        ***Z
```

and suppose the filed program is subsequently recalled by

```
        |PRECEDE BY MAGPACK ; ***A
        JOB
        UEDIN, BLOGGS OGPU, RECALL PROGRAM
        PARAMETER *01026561
        TAPE 2 DG720003
        TAPE 8 DG720002
        EXECUTION 2 MINUTES
        OUTPUT 0 SEVEN-HOLE PUNCH 2 BLOCKS
        COMPILER AA
        begin
        claim tape (8)
        enter filed program (8, 25)
        end of program
        56
        ***Z
```

Then:

a) The title, date and other monitoring details used throughout this run will be the title etc. of the recalling program: for example, the output will end

```
        STOPPED AT LINE 123
            UEDIN, BLOGGS OGPU, RECALL PROGRAM
```

and so on.

b) The execution time for the complete run will not be allowed to exceed the time stated in the job heading of the recalling program; that is, regardless of the fact that the original program specified 5 minutes execution time in its job heading, execution will be terminated in this case 2 minutes after entry to the recalling program.

c) The input and output used throughout the run are those of the recalling program. For instance, in the example shown above, after the original program is re-entered by the instruction

```
        enter filed program (8, 25)
```

the value of n read will be 56, not 17, and the subsequent print instruction will cause output to the punch, not to the line-printer.

d) The states of all magnetic tape channels immediately after re-entry to the original program are <u>the same as their states in the recalling program immediately before entry to the routine 'enter filed program'</u>.
For instance, in the example above, the states of the 8 logical channels at the time when the read and print instructions are obeyed will be:

Channels 1, 3,4,5,6,7: undefined
Channel 2 : tape DG720003, as yet unclaimed, and with no write permit
Channel 8 : tape DG720002, claimed, no write permit.

It is the programmer's responsibility to ensure that the magnetic tape channels are in the state in which the original program expects to find them.

e) There is no reason why program A should not recall B which might subsequently recall C ... and so on. In this case, in paragraphs a), b) and c) above, the words 'recalling program' should be taken to mean 'the program which started the whole business off'; in paragraph d), however, the words 'recalling program' should be read literally.

14.4.6.2 Should an attempt be made to recall a program from a section where no program has previously been filed, the effect is undefined, but will most probably be the monitor

PROGRAM TERMINATED BY SUPERVISOR

An attempt to enter a filed program may also be monitored in the ways listed in section 14.4.4.

14.4.6.3 A shortened version of the magnetic tape package, containing just the routines 'claim tape' and 'enter filed program' is available from the Computer Unit. This tape, called SHORTPACK, is clearly all that is needed to enable a filed program to be re-entered, since if the original program had all the magnetic tape routines available to it when it was filed, then it will still have them <u>all</u> available when it is recalled.

14.5 <u>Hardware failure</u>

A monitor such as

```
TAPE MALFUNCTION
SEE COMP. UNIT
CHANNEL 4
TRUNK 7
```

should be reported immediately, as it is evidence of a failure in the hardware.

<u>Section</u> 15   <u>Routines and Functions for KDF9 Matrix Pack</u>

All arrays are regarded as having the lower bound for the suffix of each dimension equal to 1. The actions of these routines and functions are the same as those in section 11 of the green Atlas Autocode Reference Manual published by Manchester University. However, as the compiler on KDF9 differs in certain ways from the one on Atlas, these routines need more parameters than the corresponding ones on Atlas.

To use the matrix pack the job heading of your program should take the form:

```
| INSERT MATRIX PACK; ***A
  JOB
  etc.,
```

the rest of the job heading being normal.
The following is a brief description of each of the items in the pack.

<u>routine</u> eqn solve k(<u>array name</u> A,b, <u>integer</u> n, <u>real name</u> det)
A is n x n, b is n vector. Solution x of Ax = b is placed in b, A is destroyed.

<u>routine</u> matrix div k (<u>array name</u> A,B, <u>integer</u> n,m, <u>real name</u> det)
A is n x m, B is n x n, A = inv(B). A on exit, B is destroyed.

<u>routine</u> invert k(<u>array name</u> A,B, <u>integer</u> n, <u>real name</u> det)
A and B are n x n, A = inv(B) on exit, B is destroyed.
A and B must be distinct arrays.
<u>NOTE</u>
In the above three routines det contains on exit the value of the determinant of the matrix which is inverted. Should this value be zero then the matrix is not invertible and the process has failed.
In these three routines, and the function which follows, a programmed stop is encountered if $n \leq 0$.

<u>real fn</u> det k(<u>array name</u> A, <u>integer</u> n)
<u>result</u> = det(A), A is n x n and is destroyed. This function uses eqn solve k.

<u>routine</u> null k(<u>array name</u> A, <u>integer</u> n,m)
A is n x m. A is set to zero on exit.

<u>routine</u> unit k(<u>array name</u> A, <u>integer</u> n)
A is n x n. A = unit matrix on exit.

<u>routine</u> matrix add k(<u>array name</u> A,B,C, <u>integer</u> n,m)
A,B and C are n x m.
A = B + C on exit.

<u>routine</u> matrix sub k(<u>array name</u> A,B,C, <u>integer</u> n,m)
As previous routine, but A = B - C.

<u>routine</u> matrix copy k(<u>array name</u> A,B, <u>integer</u> n,m)
A,B are n x m, A = B on exit.

<u>routine</u> matrix mult k(<u>array name</u> A,B,C, <u>integer</u> n,p,m)
B is n x p, C is p x m, A is n x m and A = B * C on exit.

<u>routine</u> matrix mult k'(<u>array name</u> A,B,C, <u>integer</u> n,p,m)
B is n x p, C is m x p, A is n x m and A = B * C^T on exit.

<u>Note</u>
In the above two routines the arrays A,B and C must be distinct, as must A and B in the following routine.

<u>routine</u> matrix trans k(<u>array name</u> A,B, <u>integer</u> n,m)
A is n x m, B is m x n. A = B^T on exit.

Section 17 KDF 9 DOUBLE LENGTH PACK

This package contains certain routines and functions for use when double-word length precision arithmetic is required. In place of the single length word consisting of an exponent of 7 bits and a mantissa of 39 bits (plus 2 sign bits), there is, in the double length quantity, an exponent of 7 bits and a mantissa of 78 bits, approximately equivalent to 23 decimal digits.

Double length quantities, for the purpose of this package, occupy two, not necessarily contiguous, real locations. The location containing the more significant part will contain a standardised floating point number, and the lesser significant part will contain a positive floating point number with an exponent of 39 less than that of the more significant part. However, this less significant part may be non-standard and so should not be used as a separate real quantity, as operations on non-standard quantities are not guaranteed.

N.B. It is not sufficient to set the lesser significant part of a number to zero when converting from single to double length. The routine 'dcon' should be used.

e.g. real x1, x2

These two real locations may now be used to hold one double length quantity. In the routines and functions both parts are specified as parameters.

```
E.g.    dread (x1, x2)
        dprint (x1, x2, 4, 20)
```

will read the next number from the data into the double length location specified by the two real locations x1, x2 and then it will be printed out to 20 decimal places.

To use the pack, the job heading of the main program should take the form:

```
        |INSERT D.L. PACK; ***A
        JOB
        etc.
```

as described in section 13.

Copies of the package may be obtained from the Computer Unit.

routine spec dcon (real a, real name x1, x2)

Places the double-length equivalent of the single length quantity a in the locations x1,x2.

routine spec da (real name a1, a2, b1, b2, x1, x2)

Places the sum of the quantities specified by a1,a2 and b1,b2 in the locations x1,x2.

routine spec ds (real name a1, a2, b1, b2, x1, x2)

Subtracts the quantity b1,b2 from the quantity a1,a2 and places the result in x1,x2.

routine spec dm (real name a1, a2, b1, b2, x1, x2)

Multiplies the quantities a1,a2 and b1,b2 and places the result in x1,x2.

routine spec dd (real name a1, a2, b1, b2, x1, x2)

Divides the quantity a1,a2 by the quantity b1,b2 and places the result in x1,x2.

<u>routine spec</u> dread (<u>real name</u> x1, x2)

Performs the same function as the permanent routine 'read' except that in this case the number is stored to double length accuracy in x1,x2. A variable number of parameters is not allowed in this routine.

<u>routine spec</u> dprint (<u>real name</u> x1, x2, <u>integer</u> m, n)

Prints the quantity x1,x2 in fixed point form as permanent routine 'print'.

<u>routine spec</u> dprint fl (<u>real name</u> x1, x2, <u>integer</u> n)

Prints the quantity x1,x2 in floating point form as permanent routine 'print fl'.

<u>N.B.</u> The quantities printed x1,x2, are <u>name</u> type parameters in these routines.

<u>routine spec</u> dmod (<u>real name</u> a1, a2, x1, x2)

Takes the absolute value of the quantity a1,a2 and places it in x1,x2.

<u>routine spec</u> dsqrt (<u>real name</u> a1, a2, x1, x2)

Takes the square root of the quantity a1,a2 and places it in x1,x2.

<u>routine spec</u> dlog (<u>real name</u> a1, a2, x1, x2)

Takes the natural logarithm of the quantity a1,a2 and places it in x1,x2.