AN AUTOCODE FOR THE PDP-8

by

B.S.Read, B.Sc.(Edin)

presented as part of the requirements for the

Diploma in Computer Science

University of Edinburgh

August 1968

CONTENTS.

INTRODUCTION.

PDP-8 Autocode is an algebraic language developed on the basic Digital Equipment Corporation PDP-8 with 4096 words of core store, a Model 33 ASR Teletype, a 300 character per second paper tape reader, and 140 character per second FACIT paper tape punch.

The original language was designed by D.J.Rees and writing a compiler for it was set as an exercise to illustrate his course of letures for this Diploma. It was developed further, taking ideas from Atlas Autocode, to try and make it a usable language for a small machine.

The language, its Operating System, two compilers, and a loader are described in detail. The program listings are affixed in a way that allows them to be inspected easily while the text is being read. (The chapters describing the Run-time system and the compilers are intended as commentary on the programs.) The development of the language is outlined, and further possible developments indicated together with a general discussion of the language, its implementation, and use.

The Appendices include brief descriptions of the PDP-8, its Assembler, and the instruction mnemonics that are used. There is also an outline of the method used to describe syntax. Full descriptions of these may be found in the references, also given as an Appendix.

```
PDP-8 AUTOCODE /R6

/ CALCULATE E TO N DECIMAL PLACES.
/ SALE, A.H.G. (1968). COMPUTER JOURNAL VOL.11, NO.2, P.230

TEXT( ~N␣=␣ ); READ (X0)
TEXT( ~M␣=␣ ); READ (X1)
/ M AND N FROM "FOCAL".
/ N->X0, M->X1,COEF[2:M] -> X2 TO XX1

X->X1
CALL ≠1
TEXT( ~! )


SUBROUTINE ≠1
/ I->Y0, J->Y1, CARRY->Y2, TEMP->Y3
/ Y4 COUNTS OUTPUT
Y->4
Y1 = 2
1: XY1 = 1
   Y1 = Y1+1
   ->1 IF Y1 <= X1
TEXT( ~~E=2. )
Y0 = 1; Y4 = 0
2: Y2 = 0
   Y1 = X1
   3: Y3 = XY1*10 + Y2
      Y2 = Y3/Y1
      XY1= Y3 - Y2*Y1
      Y1 = Y1-1
      ->3 IF Y1 >= 2
   PRINT SYMBOL (Y2+'0')
   Y4 = Y4+1
   ->4 IF Y4 < 50
   TEXT( ~␣␣␣␣ ); Y4 = 0
4: Y0 = Y0+1
   ->2 IF Y0 <= X0
END

R



N = 200
M = 130

E=2.71828182845904523536028747135266249775724709369995
     95749669676277240766303535475945713821785251664274
     27466391932003059921817413596629043572900334295260
     59563073813232862794349076323382988075319525101901

!
```

The example program opposite divides into two sections: the 'main coding', followed by the subroutines and functions. It is terminated textually by a $ (or π). Statements are terminated by a separator, which is either a newline or a semi-colon. Spaces and redundant separators are ignored by the compiler.

```
FORMAT[SS]  =  [SEP]
PHRASE[SEP] =  [newline], ;
```

The compiler first typed out its name and version number, and then received and compiled the source text line by line, from the teletype in this instance. The program ran, typing approximately two digits a second.

The input data, M, was previously calculated using 'FOCAL', the PDP-8's on-line Fortran-type language, as real arithmetic was needed for the calculation.

VARIABLES.

Each 12-bit word of the PDP-8 is referenced by an array named X of global variables or, in each subroutine or function, a local array Y. These are indexed from zero, with upper limits set by a declaration, or dimension statement:

FORMAT[SS] = [LETTER]->[EXPRESSION][SEP]
PHRASE[LETTER] = X, Y, Z

E.g. X -> 2

This sets aside X0, X1, X2. X0 has a fixed position and the declaration only alters the upper bound of the X-array. Thus the [EXPRESSION] must not be negative. A similar situation applies to the Y's during the execution of a subroutine or function.

X-declarations may only appear in the main-coding and Y-declarations only in subroutines and functions. The use of Z is explained under REFERENCE VARIABLES.

If no X-declaration is given then X->127 is assumed.

Variables hold integers in the range -2048 to 2047.
They are defined as:

PHRASE [VARIABLE] = [LETTER][OPERAND], <[EXPRESSION]>

E.g. X3, XY0, Y(X0+2), <X3>, <24>

The value of the [OPERAND] following a [LETTER] is the index to the array named by that [LETTER]. Thus if Y0 has the value 3 then the following legal [VARIABLE]s all reference the same location:

X3, X(3), XY0, X(Y0), XY(0), X(Y(0)), <@X3>

The use of <[EXPRESSION]> is explained under REFERENCE VARIABLES.

ARITHMETICAL EXPRESSIONS.

These may be defined, showing the precedence of the operators, as:

```
PHRASE[EXPRESSION]    =  [+-~ OPERAND][REST-OF-TERM*?][REST-OF-EXPR*?]
PHRASE[REST-OF-EXPR]  =  [+-][TERM]
PHRASE[TERM]          =  [OPERAND][REST-OF-TERM*?]
PHRASE[REST-OF-TERM]  =  [*/&][OPERAND]
PHRASE[+-]            =  +, -
PHRASE[*/&]           =  *, /, &
PHRASE[+-~ OPERAND]   =  [+-~][OPERAND]
PHRASE[+-~]           =  +,  -,  ~,  NIL
```

Expressions are evaluated from left to right taking account of operator precedence. Thus the unary operators [+-~] have the highest precedence, followed by [*/&], followed by [+-]. The operators +, *, - have their usual meaning, as in ALGOL or in Atlas Autocode.

/ is an integer division operator giving the largest integer not greater than the result of the exact division.

~ is logical NOT.

& is logical AND.

```
PHRASE[OPERAND]  = [N],
                   [VARIABLE],
                   '[symbol]',
                   S,
                   @[VARIABLE], BUT NOT @<[EXPRESSION]>,
                   F#[N][PARAMETERS?],
                   ([EXPRESSION]),
                   ![EXPRESSION]!
PHRASE[N]        = 0,1,2,3,4,5,6,7,8,9,10,11,12, ...
```

E.g. 302, XX2, 'A', S, @XY0, F#2(3,4), (Y0+1), !X0-1!

These alternatives represent, in order, the value of:

an unsigned integer,
a variable,
the ISO value of the symbol between the quotes,
the current setting of the PDP-8 Switch Register,
the machine address of the [VARIABLE],
the (possibly parameterless) function,
the [EXPRESSION],
the modulus of the [EXPRESSION].

Functions are discussed later. The value of a symbol is an integer between 0 and 127 given by the 7-bit stripped ISO code.
Space and newline are represented by _ and ~ respectively. The remaining operands are either sufficiently explained or else understood from a background of ALGOL or Atlas Autocode.

The value of an expression may be assigned to a variable by

FORMAT[SS] = [VARIABLE]=[EXPRESSION][SEP]

E.g. X3 = X4*X2 - 32/|X0+1|

The address of the [VARIABLE] is calculated before the [EXPRESSION] is evaluated and then the assignment is made last of all.

LABELS, JUMPS, AND CONDITIONS.

```
FORMAT[SS]          =  [LABEL]:[SS]

PHRASE[LABEL]       =  [N]

FORMAT[SS]          =  ->[LABEL][IF-COND?][SEP]

PHRASE[IF-COND]     =  IF [CONDITION][OR-COND*?]

PHRASE[CONDITION]   =  [EXPRESSION][COMPARATOR][EXPRESSION]

PHRASE[OR-COND]     =  OR [CONDITION]

PHRASE[COMPARATOR]  =  =, #, >, >=, <, <=
```

E.g.

```
1: ->2 IF Y0<Y1 OR S=0

   . . .

   ->1

2: PRINT (Y0)
```

A source statement is labelled by an unsigned integer, [LABEL], and a jump is indicated by ->[LABEL]. A [LABEL] may only be jumped to from within the main coding if it is set there, or from within the same subroutine or function if it is not. ->[LABEL] may not refer to a [LABEL] that is not set.

A jump is one of a small set of instructions that can be made conditional. (The other instructions are Switches, HALT, and RETURN.) A conditional instruction is executed if one of the [CONDITION]s following the IF, evaluated in order from left to right, is true. The [COMPARATOR]s represent the following relations between the [EXPRESSION]s:

=	equal
#	not equal
>	greater than
>=	greater than or equal
<	less than
<=	less than or equal.

SWITCHES.

Labelled statements may also be jumped to by setting the labels up in a switch list and using the value of an expression to index the list.

```
FORMAT[SS]          =  SWITCH #[N]([LABEL][SWITCH-LIST*?])[SEP]
PHRASE[SWITCH-LIST] =  [,][LABEL]
FORMAT[SS]          =  ->SW#[N]([EXPRESSION])[IF-COND?][SEP]
```

E.g.

```
SWITCH #3(1,3,2,999); ->SW#3(X0) IF X0<=3
6: . . .
1: . . .
   ->999
2: . . .
3: . . .
999: . . .
```

The same restrictions apply to referencing switches as apply to labels; of course these restrictions also apply to the labels in the switch list. A switch may be made conditional.

The [EXPRESSION] is evaluated and if it is 0 then the first [LABEL] is is jumped to, if it is 1 then the second, and so on. If there are n [LABEL]s then the [EXPRESSION] may only take on values between 0 and n-1, inclusive.

INPUT AND OUTPUT.

The following routines are available in the language's Operating System to input and output (signed) integers, ISO symbols, and paper tape binary characters:

```
FORMAT[SS]      =  [INPUT] ([VARIABLE])[SEP]
FORMAT[SS]      =  [OUTPUT] ([EXPRESSION])[SEP]
FORMAT[SS]      =  SKIP SYMBOL [SEP]
PHRASE[INPUT]   =  READ,READ SYMBOL,READ BINARY,NEXT SYMBOL
PHRASE[OUTPUT]  =  PRINT,PRINT SYMBOL,PUNCH BINARY,WRITE
```

READ and PRINT input and output integers in the range -2048 to +2047.

READ only skips spaces or newlines before reading either + or - and the number, or just the number. If anything else is read, or if the number read is outside the above range, then the program is terminated. If, however, input and output are both on the Teletype then READ types a question mark and tries again. The number read is stored in the [VARIABLE]. Any non-digit may be used to terminate a number and this symbol is still available to be read.

PRINT outputs the [EXPRESSION] as an integer right justified in a fixed field of 5 positions with any leading zeros or + sign suppressed.

WRITE outputs an integer in the range 0 to 4095 with no leading zeros, spaces, or + sign: -2048 is written as 2048 and -1 as 4095.

READ SYMBOL reads an 8-bit ISO character, ignores the parity bit in the eighth track, and stores it as an integer in the range 0 to 127 in the [VARIABLE].

PRINT SYMBOL produces 8-bit even parity characters from the 7 least significant digits of [EXPRESSION] considered as a binary number. It also inserts a Carriage-Return character in front of each ISO Line-Feed.

NEXT SYMBOL is the same as READ SYMBOL but retains the symbol to be read in again. (It is not read by READ BINARY.) SKIP SYMBOL reads the next symbol and ignores it. Thus

```
NEXT SYMBOL (X0); SKIP SYMBOL
```

is equivalent to READ SYMBOL (X0).

All characters typed on the Teletype are automatically echoed, although READ, READ SYMBOL, NEXT SYMBOL,and SKIP SYMBOL all ignore Run-Out, Rubbout, and Carriage-Return characters, whatever the input device.

Strings of text may be output by means of:

```
FORMAT[SS] = TEXT( [text] )[SEP]
```

E.g.

```
TEXT( --NUMBER_OF_SHIPS_=_); WRITE (X3)
```

[text] may consist of any symbols except newline, or the two symbols); together. Space and newline are represented by _ and - respectively. PRINT SYMBOL is used to output the [text].

SELECTING INPUT-OUTPUT DEVICES.

FORMAT[$$] = DEVICE #[N][SEP]

This instruction causes a dynamic switch to device #[N] where [N] is interpreted as follows:

DEVICE #1 : Teletype keyboard/reader

DEVICE #2 : Teletype printer/punch

DEVICE #3 : High speed paper tape reader

DEVICE #4 : High speed paper tape punch

The Teletype keyboard and printer are assumed at the beginning of the program. Devices #1 and #3 are mutually exclusive, as are #2 and #4.

SUBROUTINES AND FUNCTIONS.

As with variables, labels, and switches, subroutines and functions are distinguished by numbers. They are declared as follows:

FORMAT[SS] = SUBROUTINE #[N][SEP]

FORMAT[SS] = FUNCTION #[N][SEP]

Then follows a 'body' of source statements that do not include the above two, since subroutines and functions may not be nested. They are terminated textually by

FORMAT[SS] = END[SEP]

This is also the dynamic end of a subroutine unless it executes a RETURN. The RETURN may be conditional.

FORMAT[SS] = RETURN [IF-COND?][SEP]

The dynamic end of a function is the statement that assigns its result:

FORMAT[SS] = VALUE=[EXPRESSION][SEP]

It is a fault to execute the END of a function, or to RETURN from one, or to assign a value to a subroutine.

Functions are called as operands in expressions, as seen earlier.

Subroutines are called by:

FORMAT[SS] = CALL #[N][PARAMETERS?][SEP]

Subroutines and functions may be called from anywhere, including from within their own 'bodies'. It is a fault to call a subroutine or function that is not declared in the same program.

PHRASE[PARAMETERS] = ([EXPRESSION][PARAM-LIST*?])

PHRASE[PARAM-LIST] = [,][EXPRESSION]

The parameters are evaluated at the call and their values are assigned in turn to Y0, Y1, Y2, etc. within the subroutine or function for as may Y's as there are parameters. The parameters are always of type 'value' in the ALGOL or Atlas Autocode sense.

On entry to the routine these Y's are already declared and no dimension statement is needed for them. If further local variables are required then they must be declared. (Since these will have to have numbers greater than that of the last parameter, any dimension statement declaring them will in fact also include the parameters, because the lower bound of a dimension statement is always 0). When a routine has no parameters then any local variables used must be declared.

E.g.

```
PRINT SYMBOL ('-')
PRINT (F#1(3))

FUNCTION #1
/ FACTORIAL OF Y0
   ->1 IF Y0 = 0
   VALUE = Y0*F#1(Y0-1)
1: VALUE = 1
END
$
```

```
READ (X0); READ (X1)
CALL #1(0,1)
PRINT SYMBOL ('-')
PRINT (X0); PRINT (X1)

SUBROUTINE  #1
/ SWOP NUMBERS IN XY0 AND XY1
Y->2
Y2=XY0; XY0=XY1; XY1=Y2
END
$
```

These are two complete and correct PDP-8 Autocode programs.

REFERENCE VARIABLES.

These appear in two forms: the first uses the [LETTER] Z, and the second uses <[EXPRESSION]>.

(i) The Z-array is the X-array in the main coding and the Y-array in subroutines and functions. It is always a local array. Z causes a location to be indirectly referenced, so that it effectively contains the address of another location and may be used in every way instead of it. Thus,

```
X0 = @X2;  Z0 = 0
```

will set X2 to zero.

The subroutine to swop the contents of two X-variables may be made more general as follows:

```
SUBROUTINE #1
Y->2
Y2=Z0; Z0=Z1; Z1=Y2
END
```

The call now reads

```
CALL #1(@X0,@X1)
```

This mechanism can thus be used to imitate Atlas Autocode's 'call by name'.

(ii) The second method is more general. The value of the [EXPRESSION] is treated as an absolute machine address. Thus,

```
X0 = @X2;  <X0> = 0
```

sets X2 to zero, while

```
X0 = @X2;  X2 = @X3; <<X0>> = 0;  <<X0>+1> = 2
```

sets X3 to zero, and X4 to two.

There are as many levels of indirect addressing as there are nested pairs of angle brackets.

STORAGE CONTROL.

As the Operating System only uses the run-time stack temporarily within a statement, or for the duration of a subroutine or function, dimension statements may be written anywhere any number of times in a program.

The stack may also be manipulated by these permanent routines:

FORMAT[SS] = STACK ([EXPRESSION])[SEP]

FORMAT[SS] = UNSTACK ([VARIABLE])[SEP]

STACK puts the value of the [EXPRESSION] onto the head of the stack, leaving the stack pointer pointing to it. UNSTACK transfers the last entry on the stack into the [VARIABLE], and moves the stack pointer back one to effectively erase it from the stack.

The user must keep track of what he stacks and unstacks so as not to interfere with the Operating System. The following statement sets [VARIABLE] to the absolute machine address of the last stack entry:

FORMAT[SS] = LOAD ([VARIABLE])[SEP]

OTHER FEATURES.

Comments may be written as source statements and consist of any text except [SEP].

FORMAT[SS] = / [comment][SEP]

A null source statement consisting simply of a [SEP is ignored:

FORMAT[SS] = [SEP]

The combination of a percent sign (%) followed by a newline will be ignored by the compiler to allow source statements to extend over a line.

Machine code, in a form dependent upon a particular version of the compiler, may be written into a program by:,

FORMAT[SS] = *[mcode][SEP]

The program may be stopped anywhere by

FORMAT[SS] = HALT [IF-COND?][SEP]

Execution continues after pressing the CONT button of the PDP-8.

The textual end of the program is indicated by:

FORMAT[SS] = $

OBJECT CODE AND THE RUN-TIME SYSTEM.

The organisation of the object program in the machine is as follows:

	OP-SYSTEM	PROGRAM	STACK	FREE	LOADERS
ADDRESS: 0	734	AX0	STP	7600	7777 (octal)

AX0 at the end of the object code is the address of X0 and is stored initially on page 0, in AX and AY. These two locations contain the addresses of X0 and Y0, respectively, at all times, but by initializing AY to AX0 it is possible in the main coding to reference X-variables relative to AY. This feature is used by the Z-variables.

The Operating System was written in PAL, the PDP-8's assembly language, and occupies most of the first four pages. It consists mainly of routines that are used by the program, and most of the object code produced by the compiler consists in calls on these routines. This was done to take advantage of the machine's speed, in an attempt to save space.

The routines may be divided into the following categories:

1. routines to set the accumulator to the value of an operand,
2. routines to set ASS to the address of a variable or a label,
3. routines to manipulate the stack, and subroutine entry and exit,
4. arithmetical routines,
5. input and output routines.

In the Operating System are also constants for its own and the user program's use, and the addresses on page 0 of routines not on page 0 so that they may be accessed from anywhere in store. The constants for user programs are 2, 3, ...10 and are labelled N1, N2, ...N9, respectively. The Operating System's work space consists of locations within itself only, unless it is directly manipulating the stack for the user program. The user program has no control over the accumulator and uses the stack for temporary work space.

The object code produced is described following the above categories.

ROUTINES TO SET THE ACCUMULATOR.

A convention is followed that ensures the accumulator is clear whenever it is to be set to a constant. Thus it is set to 0 immediately, and to 1 by using IAC. For 2 to 10, these constants on page 0 are simply added into the clear accumulator. For all other constants the routine SETAC is used which is called with the constant in the following core location. It is picked up from there and added into the accumulator.

PKUPX, PKUPY, PKUPZ, and IND are used to get the value of [VARIABLE]s. For <[EXPRESSION]>, IND is entered with the value of [EXPRESSION] in the accumulator. It leaves in the accumulator the contents of the address passed to it in the accumulator. [LETTER] [OPERAND] uses the first three, which are entered with the value of [OPERAND] in the accumulator. The 'base' AX or AY is added to this 'displacement' and IND provides the contents of this address. For Z, the neces-:sary address is its contents which are picked up using PKUPY before calling IND. As explained earlier, in the main coding Y is synonomous with X.

The convention of leaving the value of an operand (or expression) in the accumulator when it is evaluated allows these routines to be called any number of times to evaluate, for instance, XYXZ3.

The instruction OSR sets the clear accumulator to the value of the Switch Register.

ROUTINES TO SET 'ASS'.

ASS is location 21 (octal) and is used for indirectly addressing [LABEL]s or [VARIABLE]s for jumps or assignments.

ADDRX, ADDRY, ADDRZ, like PKUPX etc, are entered with the value of [OPERAND] in the accumulator and the [VARIABLE] address is calculated in the same way. It is then assigned to ASS, which is later used to assign the contents of the accumulator to [VARIABLE] with DCA I ASS.

SETAS is used for jumps; the call is followed by the address of the [LABEL] (occupying a whole word) which is picked up and put in ASS.

For a switch, the value of the [EXPRESSION] is in the accumulator on entry to SW. The location after the call on SW contains the base address of the switch-list and this is added into the accumulator for the address of the [LABEL] required. It is picked up by INT and put into ASS.

For convenience the routines SETAS and SW are used for both conditional and unconditional jumps, with the actual jump being JMP I ASS. Thus a simple unconditional jump occupies 3 core locations.

The switch-list is laid out in core where it is declared with the [LABEL]s in consecutive locations. A label for the list is set at the beginning, and another at the end which is used to jump around the whole list at run-time.
ASS is set up before evaluating the remainder of an assignment or a [CONDITION] because it is easier to compile that way and because it results in less object code. It means, however, that a conditional switch may fail during the evaluation of [EXPRESSION] despite a condition to prevent it; it was considered worthwhile to lose this complete generality.

STACK MANIPULATION.

The pointer to the last entry of the stack is STP, at address 17 (octal). It is an auto-indexing register so that DCA I STP is sufficient code to stack the contents of the accumulator.

The routine UST, entered with the accumulator clear, moves STP back one location and leaves the entry it was pointing at in the accumulator. (This is not done via STP because it would increment itself.) UST preserves the link for the routine MULT.

These routines are available as the permanent routines STACK and UNSTACK; LOAD provides the current value of STP.

STP is used during subroutine or function entry and exit, and in evaluating [EXPRESSION] and [CONDITION]. Otherwise it marks the end of the X-array during the execution of the main coding and the end of the Y-array during a subroutine or function; this is always true between source statements. Thus to declare or re-declare variables a dimension statement has only to reset STP relative to AX, or AY, as appropriate.

ARITHMETIC ROUTINES.

After evaluating an [OPERAND] its result is left in the accumulator. The unary operators - and - can then be applied using CIA and CMA which respectively negate and complement the accumulator. The binary operators have the first operand stacked while the second is evaluated. All the arithmetic routines take one operand from the head of the stack and the second from the accumulator. They leave the result in the accumulator. From there it is either stacked temporarily for the next operation, or stacked more permanently as a parameter, or simply assigned to a variable.

The arithmetic routines are AD, SUB, ND, MULT, and DIV. The last two are modifications by D.J.Rees of PDP-8 Library routines.

SUBROUTINE ENTRY AND EXIT.

The diagram below represents the state of part of the stack after a possible subroutine entry sequence. The positions of AY and STP before the entry sequence are shown in parentheses. The system used essentially follows that used by the Atlas Autocode Compiler on KDF9.

The subroutine's return address, R/A, and the previous contents of AY, @Y0, are stacked before the parameters in Y0, Y1, and Y2 in position but not in time. Thus a second auto-indexing register, PPP, is set to the position of STP where @Y0 and R/A will be stored, and STP is moved on two (ready to assign the parameters as they are evaluated). This is done by routine INCC.

After the parameters are evaluated CALL is entered, which stacks @Y0 and R/A via PPP. PPP is left in the position shown and AY is placed one position on from it. The call on CALL is followed in core by the address of the subroutine to be entered; it is now entered.

If subroutines were textually nested then the address of the base pointer (here AY) would have to be passed to CALL and the END sequence as a second parameter, lengthening these and the actual entry and exit sequence in the program. The facility was not considered useful enough to warrant it.

Returning from a subroutine involves a jump to END where AY is restored, and the return address picked up and jumped to. This unstacking requires STP in the position of PPP shown above, one behind AY. The process automatically restores STP to its former position.

The return instruction JMP I END can be 'skipped' over as the result of a condition.

FUNCTION ENTRY AND EXIT.

The scheme for subroutines described above needs some extra facilities for functions. Since a function may reset PPP in parameter evaluation, or ASS in an assignment or condition, the contents of these two are stacked before the call on INCC. This is done by INCF. The rest of the entry is as for subroutines.

The function exits with its value in the accumulator, and this is saved in TMP while the END sequence is followed. Then a routine FN is called which res-:tores ASS and PPP and resets the accumulator to the value of the function.

Functions are evaluated in [EXPRESSION] as encountered and are not given a special precedence as they are in Atlas Autocode.

INPUT ROUTINES.

The permanent routines READ, READ SYMBOL, READ BINARY, NEXT SYMBOL, and SKIP SYMBOL, are abreviated to R, RS, RB, NS, SS, respectively, in the Operating System.

Before they are called the address of [VARIABLE] is calculated and stored in ASS. SS assigns ASS to zero itself. R, NS, and SS call RS, which in turn calls RB. This last reads a character from the input device, echoing it if it is from the teletype, and exits with it in the accumulator. The other routines operate as described earlier. The buffering of a symbol by READ and NEXT SYMBOL is done using the location SYM, which is otherwise zero. RS always tests SYM for zero before calling RB to read another character. SYM is set by R and NS and cleared by RS and SS. RB has no access to SYM because it contains a symbol and not a binary character.

When R has a digit it first checks that the number so far read in is less than 205, because the new digit makes this 2050 which is too large and will cause over-:flow in the multiply routine. However neither of these trap 2049 which is tested for separately afterwards.

OUTPUT ROUTINES.

The permanent routines PRINT, PRINT SYMBOL, PUNCH BINARY, and WRITE, are abreviated in the Operating System to P, PS, PB, and W, respectively. They are called with the value of [EXPRESSION] in the accumulator and output it in the manner described earlier.

TEXT calls the routine MSGE followed by the values of the symbols in the text string in separate consecutive core locations and a zero terminating the string. It would have been more sparing on store to make the final symbol negative instead of taking an extra location for the zero. The message output routine in the PDP-8 library was considered too long and too restrictive in its available symbols. (It takes two stripped symbols packed to a word, using @ as a terminator and excluding lower case letters.)

P and W use the same basic routine to decode and print numbers by setting flags. The arithmetic is done on positive numbers in 13 bits, using the link to extend the accumulator. This allows WRITE to treated a 12 bit number as unsigned. PRINT takes the modulus of the number and sets SIGN to a 'space' if it was positive, or 'minus' if not. (The difference between these two symbols is 13, stored in MCR.) The routine types leading spaces for zeros until a non-zero digit is going to come up, when it prints SIGN and sets it to zero. If SIGN is zero no further sign or spaces must be printed, so that for WRITE it is initially set to zero. When a digit has been printed then PRNT is made non-zero so that later zeros are not suppressed. It is also set when the last digit is about to be printed to ensure that 0 is printed.

DEVICE CHANGING.

The first location of RB and PB is either a NOP (null instruction) or else a jump to the input-output sequence for the high speed paper tape devices. The NOP is stored in the location ASR, and the jumps in HSR and HSP. Changing devices requires picking up one of these and assigning it to location RB+1 or PB+1, as appropriate.

RUN-TIME FAULTS.

Any fault checking takes up space and so is generally avoided. Often it would be against the 'spirit' of the language, which generally assumes you know what you are doing. It would have been better to include essential checks that could be left out if required, but as it stands there is hardly room for them in the compiler.

However, some faults are trapped and cause the program to halt at location 0001 after a JMS to location 0000, which will contain the address of the fault. The correspondence between addresses and faults is as follows:

0244	multiplication overflow
0260	division by zero
0470	attempt to READ large number or non-numeric data
>0734	END of function executed

The addresses are in octal.

CONDITIONS.

Because of the conditional instructions of the PDP-8 the address of a conditional jump must be set up before hand so the jump may be 'skipped' over if the required condition fails; further, the condition must be reduced to a comparison with zero. Thus in a [CONDITION] the right-hand [EXPRESSION] is subtracted from the left-hand one and a skip over the JMP I ASS, or JMP I END, or HLT (for HALT), for the reverse condition, is applied.

This does not always work correctly. There is no hardware trap for arithmetical overflow (or anything else) on the PDP-8, and no software one in the addition or subtraction routines of the Operating System. As things stand -2046 is greater than +100 because the subtraction involves overflow and a result of +1950. This can be got round if necessary by comparing with zero first, so that only like signed [EXPRESSION]s are being compared.

A correct scheme would put a check in AD (only, since SUB calls it) that would set a flag positive if both operands were positive, negative if both were negative, and zero if they were differently signed. Then if after the addition the non-zero flag had a different sign from the result the flag would not be cleared. In ordinary arithmetic a non-zero flag would halt the program, while in conditions its sign would be use to obtain correct comparisons.

This was tested by a program (in PDP-8 Autocode) running on the PDP-9, doing the comparison in 18 bits and simulating the above scheme in 12 bits, but it has not been incorporated into the present Operating System.

It is perhaps worth noting that overflow into the link in two's complement arithmetic is not connected with arithmetic overflow and cannot be used to test it: it is complemented by perfectly legitimate overflows like adding 1 to -1.

THE COMPILERS FOR VERSION R5.

The manual describes Version R5 and most of these facilities of PDP-8 Autocode are illustrated in its compilers, which are themselves written in PDP-8 Autocode. There exist Compilers R5, R5.5, and R5.6 for this version, which only differ in the way their programs are compiled and assembled. The listing of Compiler R5.6 is given.

OUTPUT.

The Compiler R5 (like all the earlier Compilers) produces a tape for the PAL Assembler, with symbolic names that refer to the Operating System or to the labels of the program. The Operating System, followed by this tape, forms the complete program which is passed twice through the assembler to obtain a binary tape. This is loaded by the Binary Loader normally resident in the last page of the core store. The process is long and tedious, and only made bearable by the existence of the high speed, and very reliable, reader and punch.

By fixing the Operating System all references to it may be made absolute, leaving the assembler to deal only with the program's own labels and with producing the binary tape. This has been done in Compilers R5.5 and R5.6; the compiler first copies its own Operating System out of store and then compiles the source program, producing all instructions as decimal numbers.

The same labelling system has been employed as R5 and there are still three passes involved, but now with a single shorter tape for the final two.

COMPILER-GENERATED LABELS.

Each compiler generated label corresponds to a user label and consists of

1. a letter,
2. two digits,
3. one, two, three digits (from the label).

The letters are

L for [LABEL],
S for SWITCH,
A for end of SWITCH,
C for SUBROUTINE or FUNCTION,

In the user program [LABEL]s may be 0 to 999, and the others 1 to 49. The compiler adds 50 to the FUNCTION number to distinguish it from a SUBROUTINE of the same number. These numbers form the final digits of the compiler-generated labels. The first two digits after the letter indicate where the label was set. They are thus 00 for the main coding, 01 to 49 for a SUBROUTINE, and 51 to 99 for a FUNCTION. SUBROUTINE and FUNCTION labels all begin C00 , since they are always set in the main coding.

Switch-lists are jumped around during program execution and, for this jump, the compiler puts out a label at the end of each of them.

MACHINE CODE.

Since the output of these compilers is passed through the PAL Assembler it is possible to write PAL instructions in the source program. Thus in an [SS beginning with * anything except spaces between * and [SEP] is copied through for the assembler replacing _ and - by space and newline respectively.

The problem of not knowing where page boundaries come may be overcome by machine code subroutines at the beginning of the program. Since a HLT instead of a jump is inserted before SUBROUTINE and the jump inserted by the user around the subroutine body occupies three locations, then the the first machine address after the Operating System available to such a subroutine is 0741 (octal). END or RETURN becomes the single instruction JMP I END; the numerical equivalent is 2847. The instructions to pick up the value of a constant or a parameter may be deduced from SUBROUTINE #3 and SUBROUTINE #13 respectively of the compiler. All numbers should be written in decimal.

INPUT.

Compiler R5.5 accepts the source program from the high speed reader and outputs the object code on the high speed punch. The line number of the beginning and end of each SUBROUTINE and FUNCTION, the end of the program, and each faulty line, are put out on the teletype.

In version R5.6 the switch register is used to indicate a number of alternatives:

bit 0	= 0	: syntax check and object code,
	= 1	: syntax check only, no object code,
bits 1-11	= 0	: teletype input, no program map,
	> 0	: high speed reader input and program map.

When a fault number is put out, the faulty [SS] is also put out. Except during a syntax check, the user is invited to retype the rest of the line, including [SS]s that may follow the faulty one on the same line. (This retyping of the whole of the rest of the line facilitates the control of storage for the compiler.) An origin reset instruction for the loader is put out followed by the code for the corrected line, so that it automatically overwrites the faulty code. It is not possible to reset a label in this process because the PAL Assembler takes the first address as correct when a symbol is defined twice.

Any lines typed in to the teletype may be edited before the line-feed using 'Control D', which causes the previous character in the input array to be echoed and deleted; it has no effect on an empty array or on input from the high speed reader.

A comment on the first line is copied through onto the program map by way of a title line, and after the end of the program the total length of the object program is printed as a decimal number. The address of AX0 is one greater than this number.

RECOGNITION AND COMPILATION.

Each line is read into a buffer that increases in length to hold it. The recognition is simple and ad hoc, and usually checks each character in turn. Initial symbols are most important in this and a list of necessary ones is kept at the end of the program to be matched against, so that a switch may be made on the value of a pointer to the matching symbol.

Since the last code to be put out is a HLT, this is stored in location X(-1), and the list is stored in locations X(-2) to X(-32). The initial symbols are stored in X(-2) to X(-23) for [SS], in X(-12) to X(-18) for the permanent routines, in X(-20) to X(-27) for [OPERAND], and in X(-28) to X(-32) for [COMPARATOR]. No jumps are inserted around subroutines so the list is not executed.

Code is 'planted' by calling SUBROUTINE #22 which increments the machine address counter and tests whether the run is a syntax check or not before outputting anything. The labels are similarly dealt with by SUBROUTINEs #1, #16, and #20.

Besides these four, the basic compiling subroutines are as follows:

SUBROUTINE #6 : put the next significant input symbol into X0,
SUBROUTINE #2 : compile [OPERAND],
SUBROUTINE #11 : compile [EXPRESSION],
SUBROUTINE #13 : compile [VARIABLE],
SUBROUTINE #12 : compile [IF-COND?].

These call others and each other as required by the syntax.

Short sequences of code that are repeated at all have been written as subroutines in an effort to gain space; this has little apparent effect on the speed of compilation with paper tape input and output. Such SUBROUTINEs are #1, #5, #7, #9, #14, #15, #18, and #19.

SUBROUTINE #8 deals with output to the teletype, formatting the program map and error messages. It temporarily switches output from the punch to the teletype. It only types one fault per [SS] as further ones are usually spurious; for the same reason the main coding skips the rest of an[SS] that it does not recognise completely. X11 flags a faulty [SS].

THE COMPILER R6 AND ITS LINKING BINARY LOADER.

This compiler accepts a subset of version R5 and produces a binary tape in one pass that is loaded using the standard PDP-8 Binary Loader. The subset excludes switches and PAL III instructions in [mcode].

The output from the compiler consists first of a Linking Binary Loader that overwrites the HLT of the ordinary loader with a jump to its beginning. On gaining control it first checks that it has been correctly loaded and then proceeds to read in the binary program following it on the same tape. It fills in forward references to labels using a method of R.W.Keeling's, which is explained below.

The compiler maintains two label tables, one for [LABEL]s and one for SUBROUTINE/FUNCTIONs. Each table entry consists of two words:

1. the label number,
2. an address.

If the label number is positive then the address is the address of the label, otherwise it is the address of the last forward reference to the label. As each new forward reference is found the address of the last one is put out into the word that will finally hold the label address and that of the new one is stored as the second entry. The first link of the chain has a zero to mark it for the loader. When finally the label is found its address is put out with special flag bits, followed by the address of the last forward reference to it. The loader will take this last link in the chain and go back up it filling in the label address till it reaches the zero link.

When a label is set its number is stored positive, with its address, for any references afterwards.

This simple method of handling the many-one mapping problem between jumps and labels is ideal for the PDP-8 because it requires the minimum of space; also, the machine could fill in at least one thousand forward references between reading successive characters from the paper tape reader. It is particularly simple in that it is only filling in whole words and not modifying address fields, though this is envisaged in Keeling's complete scheme.

The PDP-8 Binary Loader assembles pairs of 6-bit characters on the binary tape into complete words and then assigns them to consecutive locations in store.

A pair with the 7th track of the first character punched marks the word as a new origin, while both the 7th and 8th tracks punched marks the single character as the number of a new core bank and is ignored for our machine. Characters with the 8th but not the 7th track punched mark the beginning and end of the binary program. The last word before this 'trailer' code is the 'checksum', which is the sum (ignoring overflow) of all characters on the tape except the leader and trailer codes. The loader keeps a similar sum and the two are compared at the end for possible misreads.

I adapted this loader to recognise the 'core bank swop' code as the flag to fill in the address of a label that had been forward referenced.

The binary punch subroutine is #22 and it takes the flag bits as the second parameter. When this is zero it will occupy a location in store so that X6 is incremented. This is only false in the case of the address of a label that has been forward referenced because it occupies locations that have already been counted, and X6 must be reduced after it is put out.

SUBROUTINE #21 handles the label tables as outlined above, and calls #22 when it outputs an address or the end of a forward reference chain. SUBROUTINE #20 checks for unset [LABEL]s in the main coding on encountering the first SUBROUTINE/FN, or at the end of the program, and in each SUBROUTINE/FN on encountering its END. This means that no main coding can appear between SUBROUTINE/FNs, which can use the same space for their [LABEL] tables. Since a positive label number indicates that the label is set, 0 cannot be forward referenced and is best avoided.

The listing of the compiler R6 is given with comments against statements concerned with producing binary, being the only places where there is any difference from compiler R5.6 .

THE DEVELOPMENT OF PDP-8 AUTOCODE.

The syntax of the first version of PDP-8 Autocode is as follows:

```
PHRASE[SS]          =  [INDEX]=[EXPRESSION][SEP],
                       ->[N][CONDITION?][SEP],
                       [N]:,
                       CALL ([N])[SEP],
                       SUBROUTINE ([N])[SEP],
                       END[SEP],
                       TEXT ([text])[SEP],
                       PRINT SYMBOL ([OPERAND])[SEP],
                       READ SYMBOL ([INDEX])[SEP],
                       * [mcode][SEP],
                       / [comment][SEP],
                       HALT [SEP],
                       $
PHRASE[INDEX]       =  II[N], I[N]
PHRASE[OPERAND]     =  [N], [INDEX], '[symbol]'
PHRASE[EXPRESSION]  =  [OPERAND][OPERATOR][OPERAND], [OPERAND]
PHRASE[CONDITION]   =  IF [OPERAND][COMPARATOR][OPERAND]
PHRASE[COMPARATOR]  =  =, #, >=, >, <=, <
PHRASE[N]           =  [DIGIT*]
PHRASE[DIGIT]       =  0, 1, 2, 3, 4, 5, 6, 7, 8, 9
PHRASE[SEP]         =  [newline]
```

The first compiler for this language was written by D.J.Rees, and after I had sorted out a few ideas on what was involved, I took over most of his Operating System as it stood. Since this is a very important part in a compiler of the type being described, it was a significant help. There was a simple improvement to the language that I made immediately, which was to let READ SYMBOL and PRINT SYMBOL work the high speed devices as well as the teletype. The device

selection was introduced as,

FORMAT[SS] = DEVICE ([N])[SEP]

The compiler made up a table of labels and subroutines as soon as they were set or referred to. The CALL and -> instructions worked as switches on these tables, and the Operating System routines involved were entered with the accumulator set to the displacement of the label in the appropriate table. Labels were restricted to the range 0 to 99 and subroutines to 1 to 40, so that (subrtn.no. * 100 + label) gave a unique number for a label, that could be stored in a single word.

Since there were no local variables for subroutines a stack of 100 locations was set aside to store return addresses only, while the index array took up the remainder of the store. All arithmetic operations were carried out with fixed locations inside the Operating System.

In Rees' compiler the units 1 to 9, tens 10 to 90, hundreds 100 to 900, and 1000 and 2000, were stored and used to assemble a constant when required. I introduced the routine SETAC in place of this.

It should be pointed out that the paging of the PDP-8 makes it necessary to do all addressing indirectly, so that stacking of subroutine return addresses is a natural way to store them, giving in fact recursive subroutines. The same considerations make an array of indices a simple and obvious data structure to provide.

The first version of the compiler was written in a strict subset of Atlas Autocode that fitted very closely to PDP-8 Autocode. A simple routine called CALL was used with an integer parameter used to switch on a switch called SUBROUTINE. These switches labelled single blocks inside CALL.

A simple test program that would strip away the redundant syntax and convert the compiler into PDP-8 Autocode was compiled by the compiler running on the KDF 9. When this program ran successfully on the PDP-8 it was then possible to pass the compiler through itself and bootstrap off the KDF 9 onto the PDP-8 (if 'bootstrap' is what you would call it). This standard technique for moving a compiler from one machine to another was described by D.J.Rees in his lecture course for the Diploma.

Version 2 investigated the possibility of using the label tables more effectively to do the checking of labels and subroutines with all the other syntax checking. The entry for each label or subroutine took two words: the first being its unique number and the second the address where it was found; the second word was initially set to zero to provide a check. The address was given by a counter that was incremented each time a machine address was used. The tables were put out after the code, as in version 1, and the compiler could have been developed then to produce binary, but it wasn't yet worth it.

Not only was the language inconvenient to use with only a single global array, but the tables were unreasonably bulky considering that in general each jump also involved four words to access them. Since one was a call on SETAC it was reasonable to introduce SETAS along the same lines. It was a matter of substituting the actual address required, for the displacement in a table containing it. This was done in version 3.

There were also introduced local variables, called J, and parameters to make subroutines more usable. This brought about the change of syntax for CALL etc. to that presently used. Conditions were generalised to compare expressions, and RETURN was introduced as a conditional because it was more efficient than a jump to a label at END, and often needed. Expressions remained simple since it really requires local variables to compile general ones conveniently and that would have to await the next version. Again because of a saving in object code, and the ease of implementing it, conditions were allowed to be OR-ed together. AND was not introduced because it would involve the same amount of object code while increasing the complication of SUBROUTINE #12.

The stack pointer STP was not moved up during parameter evaluation but was used to stack the return address and AY. The auto-index register PPP was used to assign the parameters. This worked because the stack was not used in the evaluation of expressions, but was still unsatisfactory because a J-declaration inside a subroutine with parameters became obligatory if it was going to call another subroutine. However, it was sufficient to bootstrap to version 4.

Ideas for extending the language were usually drawn from Atlas Autocode, and in one sense it became an exercise in how to implement a simplified version of facilities of Atlas Autocode. Thus version 4 included all the permanent routines now available, except STACK and UNSTACK. (LOAD then gave the value of the Switch Register.) The READ routine was obviously useful and once it was written, buffering the terminating character, it became trivial to introduce NEXT SYMBOL and SKIP SYMBOL. WRITE was introduced as well as PRINT because the compiler wanted to treat some 12-bit words as unsigned for PAL symbols, and leading spaces were wrong in this context. PRINT uses a fixed format for simplicity.

FUNCTIONs and SWITCHes were also introduced.

A FORTRAN 'computed GO TO' would have been a suitable switch, as the switch-list would be automatically jumped over at run-time, but since this list would have to be repeated for further switches on it, the ALGOL type of switch was implemented. (The Atlas Autocode array type of switch is out of keeping with PDP-8 Autocode 'data structures'.) Although the FORTRAN feature would have been better for the way I have used switches in the compiler, these still represent a considerable saving in code over a list of ordinary jumps.

The present method of evaluating expressions was introduced in version 4, but the method of compiling them easily took a while to emerge. The first attempt dealt explicitly with the precedence of an operator, kept a local stack, and generally followed an algorithm for converting expressions into reverse polish that had been given in the lectures. However, not for nothing does the language have an automatic stack, and eventually the subroutines used followed the examples given by Brooker and Morris to illustrate the Compiler-Compiler. Operator precedence is now implicit in the subroutine calls, as it is implicit in the phrase definitions given earlier. The precedence of parentheses was assigned by making sub-expressions into operands. The same could be done for unary operators to tidy up SUBROUTINE #11, and allow expressions like -X0 & -X1 which must be written -X0 & (-X1) at present.

FUNCTIONs were organised along the present lines but did not preserve PPP at all and were therefore faulty. The RESULT was called VALUE because this begins with V, and R was already over used.

Version 5 remedied the defect in function calls and at the same time slightly reorganised the handling of conditions. Previously the value of the first expression was stored in a location ARG while the second was evaluated. This was negated and the contents of ARG added in before the test for the condition was applied. Thus ARG as well as ASS and PPP would need saving during a function call and so a subtraction routine was introduced for expressions and used here also. Now only ASS and PPP need saving at a function call as the first expression of a condition is stacked anyway.

Version 5 also introduced the superficial change from I and J to X and Y because I was often typed as 1. Z- variables and the address operator @ were also defined then.

The Z-array was going to be considered as a separate local array of fixed length that had to be declared before the Y-array and displaced the latter from AY by its length, so as to avoid a separate base pointer that would complicate the CALL sequence. This approach had a number of disadvantages:

1. it involved the compiler in a lot of checking,
2. it required extra code to displace the Y-variables,
3. it wasn't general that only the Z-array had a fixed length,
4. it firmly restricted reference variables to be parameters only.

The solution was simply to define Z-variables as indirectly addressed Y-variables, and extend the Y-array to be defined as the local array everywhere so that Z could be used in conjunction with X-variables in the main coding. By removing the restrictions all the listed objections disappear.

It was not felt that reference variables would be so greatly used as to merit having proper global ones with associated ADDR and PKUP Operating System routines.

The absolute address expression between angle brackets as a variable was something for nothing as far as the Operating System was concerned, as it uses a routine already available. This is useful as it is now difficult to alter the

Operating System. It was experimentally introduced at A.Freeman's suggestion.

Sometimes it duplicates the work of Z-variables, which are more compact in object code, and is not easy to use more generally. I attempted to write a simple list-processing scheme in terms of it and too easily lost track of what was going on. One use is to imitate store mapping functions.

The following program considers the X-array from X5 to X59 to be a two-dimensional array with bounds (0:10,1:5).

```
    X0 = 0
1:  X1 = 1
2:  <F#1(X0,X1)> = <F#1(X0,X1)> + 1
    X1 = X1+1
    ->2 IF X1<=5
    X0 = X0+1
    ->1 IF X0<=10

FUNTION #1
VALUE = @X(Y0*5 + Y1 + 4)
END
```

The incrementing statement in this case could be written better as

```
2:  X2 = F#1(X0,X1) ;  Z2 = Z2+1
```

STACK and UNSTACK have proved useful in highly recursive subroutines (like R.W.Keeling's syntax recogniser in his CCV3) by giving a complete control over what is remembered at any level. It is sensible that a language which depends heavily on a stack should provide these same facilities to the user, and the pity is that the small store makes it difficult to provide them more generally to control a private user stack with operations and tests that apply to it.

In version R5 a number of ways were tried to make the language and system more usable. Those retained are as follows: reading a whole line in at a time (instead of character by character) to stop hammering the reader and to allow the text of a faulty [SS] to be given, allowing a line continuation marker (%), and allowing a line to be corrected on-line.

The facility of single stepping [SS]s was tried and abandoned as being insufficient alone for effective fault correcting. When Switch Register bit 0 was up the compiler produced code to halt the program at the end of each [SS] if the same setting was made during the program's execution.

By this time the compiler was near to filling the store but I was keen to apply R.W.Keeling's method of dealing with forward references and compile to binary. To make room, then, switches were removed as being a single easily identifiable and less used unit of the compiler. This meant that this version, version R6, would not be able to compile itself, but it was not likely to be able to anyway because of the large label table it would require. This is in fact the case, although some over-laying of the table has been done by restricting subroutines to the end of the program and throwing away the main coding labels at the first subroutine.

Since the PAL Assembler is not used in this version a restriction has been placed on [mcode] which may now only consist of single decimal numbers. Thus R6 is completely a sub-set of R5.

CONSIDERATIONS OF THE COMPILERS.

It was possible in the early versions to determine an [SS] by its first symbol, and this approach has worked fairly well in later versions.

However it is apparent that a large amount of the compiler is concerned with checking for certain symbols by conditional jumps, which are the most expensive in object code. This could be done by a syntax recognition routine but the question arises as to whether the testing of the resulting analysis record, as well as the analysis record itself, would not be as expensive. After all, statements of PDP-8 Autocode are bare enough to be 'analysis records' themselves.

It seems that the Compiler-Compiler 'Built-in Phrase' is the most hopeful way of providing a compact compiler. This approach was adopted by R.W.Keeling in his on-line language CCV3 (Compiler-Compiler Version 3) which is described briefly in the next section with an example of an expression compiler written in it. A similar approach is used by J.M.Foster in his 'Syntax Improving Device', though he reports an expansion of the resulting compiler, when compiled, over a hand-written one: Computer Journal (May 1968), Vol.11, No.1, p.31.

Optimising the object code of PDP-8 Autocode is difficult, because of the instruction set, and because of the paging. The instruction set has been improved by using Operating System routines. The slowness of doing this however means that it is not practical for PDP-8 Autocode programs to drive, for instance, the cathode ray display tube which is also connected to the machine used. In-line functions could be introduced to add 1 to an expression or variable as being the easiest way to recognise when to use IAC and ISZ, though the actual saving in code is not very great. There could also be functions for the rotate instructions: the lack of these meant that SUBROUTINE #22 in Compiler R6 had to be written in machine code.

The question of paging is more difficult. It is certainly wasteful that each jump requires three words, particularly when there are so many of them, but I can see no simple way of dealing with the problem.

Finally, in writing a compiler for the PDP-8 it is worth knowing from the start what facilities the language will provide and designing some basic operating routines to carry them out easily. Also, some sort of on-line fault correction, the type of binary loader described here, and the method of copying it and the Operating System straight from store to produce a single binary tape are worth incorporating in the system to make it all usable.

THE EXPRESSION COMPILER IN 'CCV3'.

CCV3, or Compiler-Compiler Version 3, was written by R.W.Keeling for the PDP-8 as an on-line language. It was written in itself with the interpreter written in PDP-8 Autocode (though this has since been re-coded in PAL to give more room for syntax definitions).

Statements of CCV3 are active syntax definitions written in a notation like that of the Compiler-Compiler, but abbreviated in several instances: for example, PHRASE becomes P. It has different primitive built-in-phrases, but many more of them. Those used in the example are explained briefly below.

The interpreter operates in one of four modes on the phrase definitions:

[R] : read and match (normal recognition),
[W] : write,
[C] : read, match and copy,
[M] : read, match and move (to a buffer in the example).

The mode remains set for all sub-phrases (unless it is reset) until the end of the alternative in which it is set, when it reverts to the previous mode.

The buffer referred to by [M] may also have items planted in it:

[P] : plant the following phrase,

and this buffer may be read into the input or executed:

[RB] : read the buffer,
[EB] : execute the buffer.

When the buffer is executed it is necessary to begin the relevant section with [O], indicating there is no other alternative there, so that it follows the internal conventions of the interpreter.

The built-in-phrase [ST] matches the previous phrase until it fails, so it is not completely analogous to the * operator of Compiler-Compiler. The NIL alternative is written [] and cannot be indicated by ? after the phrase identifier.

A whole phrase may be failed explicitly from one of its alternatives with

[TF] : total fail.

This gives a controlled back-tracking. Back-tracking on the text may also be done by

[RI] : recover input,

which recovers the input so far matched by the current alternative for further matching by it.

Various litterals are represented by these built-in-phrases:

[_] : space,
[-] : newline,
[,] : comma,
[D] : any digit,
[A] : any character.

The phrase

[PR] : print the following syntax entry as a decimal number,

is used to output numbers in the absence of facilities to turn numbers into strings of digits. The reverse facility would be useful as well.

The example given is the expression compiler for PDP-8 Autocode written to produce equivalent PAL III instructions to Compiler R5. It is most obscure when attempting to deal with numbers, which PDP-8 Autocode can do easily. It does not signal any fault except that of the interpreter to say that the whole thing is not recognised. The example is given without further explanation as the reason for writing it was to compare the space it occupies with that occupied by the relevant subroutines of Compiler R5, namely #11, #10, #2, #3, #4, #5, #6, #13, #14, #15, #17, and #18. R5 is the version producing PAL text and not numbers.

These subroutines occupy 1360 locations, and the compiler occupies 3430. The sections of CCV3 (also in PDP-8 Autocode) required by the expression compiler occupy 1300 locations, and the syntax table requires another 360. The 477 locations of the PDP-8 Autocode Operating System are included in these figures.

The expression compiler is 300 words longer using this syntax recognition routine than it is using ad hoc recognition, although this is the least ad hoc part of the compiler. Also, using the method of CCV3, it would be possible to write a set of primitive built-in-phrases that better suited the compiler, or to re-organise the compiler and its output to better suit CCV3. There could be more repeated sequences written as separate phrases.

Furthermore, the expression compiler represents just over two-fifths of the whole code of Compiler R5, excluding the Operating System: thus the syntax table required by the same sections of CCV3 for the whole compiler is likely to be about one thousand locations, leaving the ad hoc version still nearly one thousand longer.

I am grateful to Roger Keeling for the information concerning CCV3 and for advice over coding the example (given on the next page).

```
P[EXPR]    =  [+-~OPD][ROT*?][ROE*?]

P[+-~OPD]  =  +[OPD],
              -[OPD][W][~]CIA,
              ~[OPD][W][~]CMA,
              [OPD]

P[ROT]     =  *[STK][OPD][W][JMS]MULT,
              /[STK][OPD][W][JMS]DIV,
              &[STK][OPD][W][JMS]ND

P[ROE]     =  +[STK][TERM][W][JMS]AD,
              -[STK][TERM][W][JMS]SUB

P[STK]     =  [W][~]DCA[_]I[_]STP

P[JMS]     =  [~]JMS[_]

P[TERM]    =  [OPD][ROT*?]

P[OPD]     =  [N],
              [VAR],
              '[SYM]',
              S [W][~]OSR,
              @X[OPD][W][TAD]AX,
              @Y[OPD][W][TAD]AY,
              @Z[OPD][W][JMS]PKUPY,
              [FN],
              ([EXPR]),
              |[EXPR]| [W][~]SPA[~]CIA

P[TAD]     =  [~]TAD[_]

P[N]       =  0[~D], 1[~D][W][~]IAC,
              [2-10][~D][RI][W][TAD]N[C][2-10],
              [D*][RI][W][JMS]SETAC[~][C][D*]

P[~D]      =  [D][TF], []

P[2-10]    =  10, [D]

P[VAR]     =  [M][LETT][R][OPD][W][JMS]PKUP[RB][C][LETT],
              <[EXPR]> [W][JMS]IND

P[LETT]    =  X, Y, Z

P[SYM]     =  [O][P][PR][M][A][W][JMS]SETAC[~][EB]

P[FN]      =  F#[M][D*][W][JMS]INCF[R][PRMS?][W][JMS]CALL[~]FOO[RB][C][D*][W][JMS]FN

P[PRMS?]   =  ([EXPR][ROP*?]), []

P[ROP]     =  [STK][,][EXPR]

P[D*]      =  [D][D*?]

P[D*?]     =  [D][ST], []

P[ROT*?]   =  [ROT][ST], []

P[ROE*?]   =  [ROE][ST], []

P[ROP*?]   =  [ROP][ST], []
```

CONSIDERATIONS OF PDP-8 AUTOCODE.

PDP-8 Autocode has principally been used for programs involving the manipulation of characters, such as paper tape code conversion programs, a program to make Atlas Autocode programs acceptable to an IMP compiler, and for CCV3 which will be discussed later. This is because the word length of the PDP-8 makes it suitable for such programs.

Also the language provides no floating point arithmetic. This has been a design error in so far as the language is algebraically oriented but lacks the full computational facilities expected of algebraic languages. It was a onscious choice to leave out floating point arithmetic because the full PDP-8 Floating-point Package occupies just over three-eights of the store, and because floating point numbers require three words each: some sort of parameter specification would become necessary to know when a parameter requires one word or three and there is no room to store such specifications.

It would have been better to provide some way of handling strings, or simply vectors, in a way convenient for symbol manipulation, while still retaining the integer arithmetic facilities. As it is, programs to process strings are inadequate and clumsy.

Further, since most of this string processing involves reading and punching lots of paper tape it would be worth providing proper interrupt controlled and buffered input-output. The characteristics of the reader would need sorting out first, however, as I tried an Operating System in the first version that buffered 1 page of input, without interrupts, but found the reader dropped more characters than when it either bumped along reading one at a time or ran more smoothly reading a line at a time.

Writing programs in the language is made difficult by two things in particular: the lack of names and the superabundance of labels. Names were left out because of the room needed to store them, though a more compactly written compiler might find that room.

More serious is the lack of reasonable conditions and the resulting list of labels stored by the compiler. For the versions producing output for PAL it would have been simple to introduce many compiler generated labels using

different initial letters, but I always looked to when the compiler would go straight to binary and keeping a single label table would be much simpler, and more sparing on space.

This is in fact not true. Since there is no difference made at present between 'local' labels that are referred to once only and labels that are referred to several times, they are all stored to the end. It would be more sparing on space in the compiler, make no difference to the object program's length, and provide more readable source programs, if complete conditions and compound statements were introduced. The address of the location to be filled in could be remembered by an explicit STACK instruction, or by a recursive subroutine call, then put out with the forward reference, when it is known, and forgotten about.

The attraction of the PDP-8 is due to its availability, speed, and interactive on-line Teletype. It has been useful in letting undergraduates work their symbol manipulation programs on-line, since it was a simple matter to recode the Atlas Autocode programs in PDP-8 Autocode. To actually develop programs on-line to the PDP-8 it seems better to have some interpretative system that can later compile if necessary, because as soon as students have a program running they want to modify it. This particularly applies to the basic PDP-8 with only the Teletype for input and output. Multi-pass systems are a mistake, and a modification of Compiler R6 to input and output only on the Teletype would be useful.

APPENDICES

REFERENCES.

Bratley, P., Rees, D.J., Schofield, P.D., Whitfield, H. (1965). Atlas Autocode Compiler for KDF9, Computer Unit, University of Edinburgh.

Brooker, R.A., Morris, D. (1962). A General Translation Program for Phrase Structure Languages, J.A.C.M. 9, 1962.

Keeling, R.W. (1968). CCV3, Department of Computer Science, University of Edinburgh.

PAL III Symbolic Assembler Programming Manual, Digital Equipment Corporation, Maynard, Massachusetts, U.S.A.

PDP-8 Users Handbook, ditto.

SYNTAX DESRIPTION.

The notation for describing syntax follows that used by Brooker and Morris in the Compiler-Compiler, which may be considered a variant of Backus-Naur Form. The notation used is outlined below.

All source statements of the language are of the class [SS] and each new source statement is added to the class by writing FORMAT[SS] = ...

Phrase definitions are preceded by the name PHRASE and are named by an identifier written between square brackets. An alternative within the definition is terminated by a comma or by the end of the line. Spaces and End of Line characters after a comma are ignored, but otherwise all litterals stand for themselves. Comma and the End of Line character are represented by the phrases [,] and [newline] respectively. Phrases with indentifiers written in lower case are only defined in the text.

An Alternative beginning with BUT NOT is an illegal alternative, and one beginning with NIL indicates a possible absence of the phrase. The NIL alternative may also be be indicated by a query (?) written after the phrase name inside the square brackets. An asterisk (*) in the same position indicates that the phrase occurs one or more times, so that the combination *? means the phrase occurs zero or more times. A phrase name may be subscripted with a slash followed by a number to distinguish it from other occurrences of the same phrase.

In the text phrase identifiers are used to fix more precisely what is being referred to in a particular context. The identifier is used to represent both a particular piece of text that has matched the phrase definition, and the result of executing the code generated in the recognition process: which is meant is clear from the context.

THE PDP-8.

The basic PDP-8 has 4096 12-bit words of core, with a cycle time of 1.5 microseconds. It operates peripherals on a 'bus-line', each one having a selection code and a 'ready' flag. The setting of this flag can cause a program interrupt, or may simply be tested for by the program when it is ready to use the device. (This latter course has been taken in the language's Operating System.)

Programmable registers are the single arithmetic Accumulator and the Switch Register; this last may only be read by the program and is set from the console switches. The Accumulator is used for the input and output instructions, and for all conditional instructions except ISZ. It is extended for some instructions by the Link, which is a single bit and acts in these cases as the most significant bit of a 13-bit register. These are referred to sometimes as the AC and LINK.

The twelve Accumulator and Switch Register bits are considered to be numbered from 0 to 11 where bit 0 is the most significant. Eight-bit characters are entered into the Accumulator in a way that results in them being counted in the reverse direction. Bit 8 is the Accumulator bit 4, and bit 1 is the Accumulator bit 11.

The core is considered to consist of 32 'pages' of 128 words each. At any location an instruction may address any word on the same page or on page 0. To address any other word in store it is necessary to use indirect addressing.

Locations 10 to 17 (octal) on page 0 are 'Auto-index Registers'. These increment their contents by 1 when indirectly addressed and the result is used as the effective address of the instruction.

PAL III ASSEMBLY LANGUAGE.

This is a brief summary of the facilities and mnemonics of the PAL III Assembler of the PDP-8, that have been used in the Compiler, Operating System, and Loader of PDP-8 Autocode.

Symbols consist of a letter possibly followed by further letters or digits up to 6 charaters long. They may be assigned the value of simple expressions.

E.g.
```
FRED=JIM+1
```

Instructions are written, separated by carriage-returns (or semi-colons), as an operation followed by an address expression, where . is the 'current address'.

E.g.
```
TAD N1
JMP .+6
```

An indirect reference is indicated by an I written between the operation and the address.

E.g.
```
JMP I .+1
DCA I STP
```

Symbols may also be assigned values as labels. They are written either by themselves or before an instruction or constant and are terminated by a comma.

E.g.
```
JIM,JMP FRED
```

A symbol may not be assigned further values: it is defined once only in a program.

Numbers are assumed to be in octal unless DECIMAL is specified.

An origin address is specified by a * followed by a number.

E.g.
```
*128
```

Each of the Operate Instructions may be combined with others in the same Group if the combination is meaningful.

E.g.
```
SNA CLA     / skip if AC non-zero & clear it anyway,
CLL RAR     / clear LINK then rotate  AC and LINK right.
```

(Some of these instructions are used also as negative constants or masks.)

Comments are indicated by a / as above, and the program is terminated by $.

The source tape is completely assembled in three passes through the Assembler.

PASS 1 : set up the table of Symbols with their values,
PASS 2 : produce the Binary tape,
PASS 3 : source listing with each location and contents in octal.

The listings given of the Operating System and the Loader are from PASS 3.

Note : 'skip' is used to mean 'skip the following instruction'.

MEMORY REFERENCE INSTRUCTIONS.

AND Y	Logical AND the contents of Y into the AC. Result in the AC.
TAD Y	Two's complement add the contents of Y into the AC. Result in the AC. (Also used to load the AC.)
ISZ Y	Increment the contents of Y by 1 and skip if the result is zero. (Also used to simply add one.)
DCA Y	Deposit the contents of the AC in Y and clear the AC.
JMS Y	Subroutine jump. Store (current address + 1) in Y and execute the next instruction at (Y + 1).
JMP Y	Jump to Y.

INPUT-OUTPUT INSTRUCTIONS.

High Speed Reader:

RSF	Skip if the reader flag is set.
RRB	Read the contents of the reader buffer into accumulator bits 4 to 11. Clear the flag.
RFC	Clear the reader buffer and flag. Fetch the next character from tape into the reader buffer and set the flag.

High Speed Punch:

PSF	Skip if the punch flag is set.
PLS	Clear the punch flag and buffer. Load the accumulator bits 4 to 11 into the punch buffer and punch the character. Set the flag.

Teletype Keyboard:

KSF	Skip if the keyboard flag is set.
KCC	Clear the AC and the keyboard flag.
KRB	Clear the AC and the keyboard flag. Read the contents of the keyboard buffer into accumulator bits 4 to 11. Set the flag.

Teletype Printer:

TSF	Skip if the teleprinter flag is set.
TLS	Load the teleprinter buffer from the accumulator bits 4 to 11, clear the flag and print the character. Set the flag.

OPERATE MICRO-INSTRUCTIONS.

Group 1:

NOP	No operation.
RAL	Rotate the AC and LINK cyclicly left one place.
RAR	Rotate the AC and LINK cyclicly right one place.
RTL	Rotate the AC and LINK cyclicly left two places.
RTR	Rotate the AC and LINK cyclicly right two places.
CLA	Clear the AC.
CLL	Clear the LINK.
CMA	Complement the AC : logical NOT.
CML	Complement the LINK.
IAC	Increment the AC by one.
CIA = CMA IAC	Complement and increment the AC : two's complement negate.
STA = CLA CMA	Set the AC to -1.

Group 2:

HLT	Halt the program.
OSR	OR the AC with the Switch Register. Result in the AC.
SKP	Skip unconditionally.
SNL	Skip if the LINK is non-zero.
SZL	Skip if the LINK is zero.
SNA	Skip if the AC is non-zero.
SZA	Skip if the AC is zero.
SPA	Skip if AC >= 0.
SMA	Skip if AC < 0.
SPA SNA	Skip if AC > 0.
SMA SZA	Skip if AC <= 0.
CLA	Clear the AC.

FAULT LIST FOR COMPILERS R5 AND R6.

Compile-time:

FAULT 1	:	Instruction not recognised.
FAULT 2	:	Number out of range.
FAULT 3	:	Illegal declaration.
FAULT 4	:	Faulty [VARIABLE].
FAULT 5	:	Faulty [OPERAND].
FAULT 6	:	END out of context.
FAULT 7	:	Nested SUBROUTINE/FUNCTION.
FAULT 8	:	VALUE out of context.
FAULT 9	:	RETURN out of context.
FAULT 10	:	Label set twice.
FAULT 11	:	[LABEL] not set.
FAULT 12	:	SUBROUTINE/FUNCTION not declared.

The equivalents of FAULTs 10 to 12 for version R5 are given by PAL III.

Run-time:

0244	:	Multiplication overflow.
0260	:	Division by zero.
0470	:	Attempt to READ a large number or non-numeric data.
>0734	:	END of FUNCTION executed.

These faults halt the program at address 0001 with the octal number on the left in address 0000.

ABBREVIATIONS.

Because of the method of recognition these words may be abbreviated to:

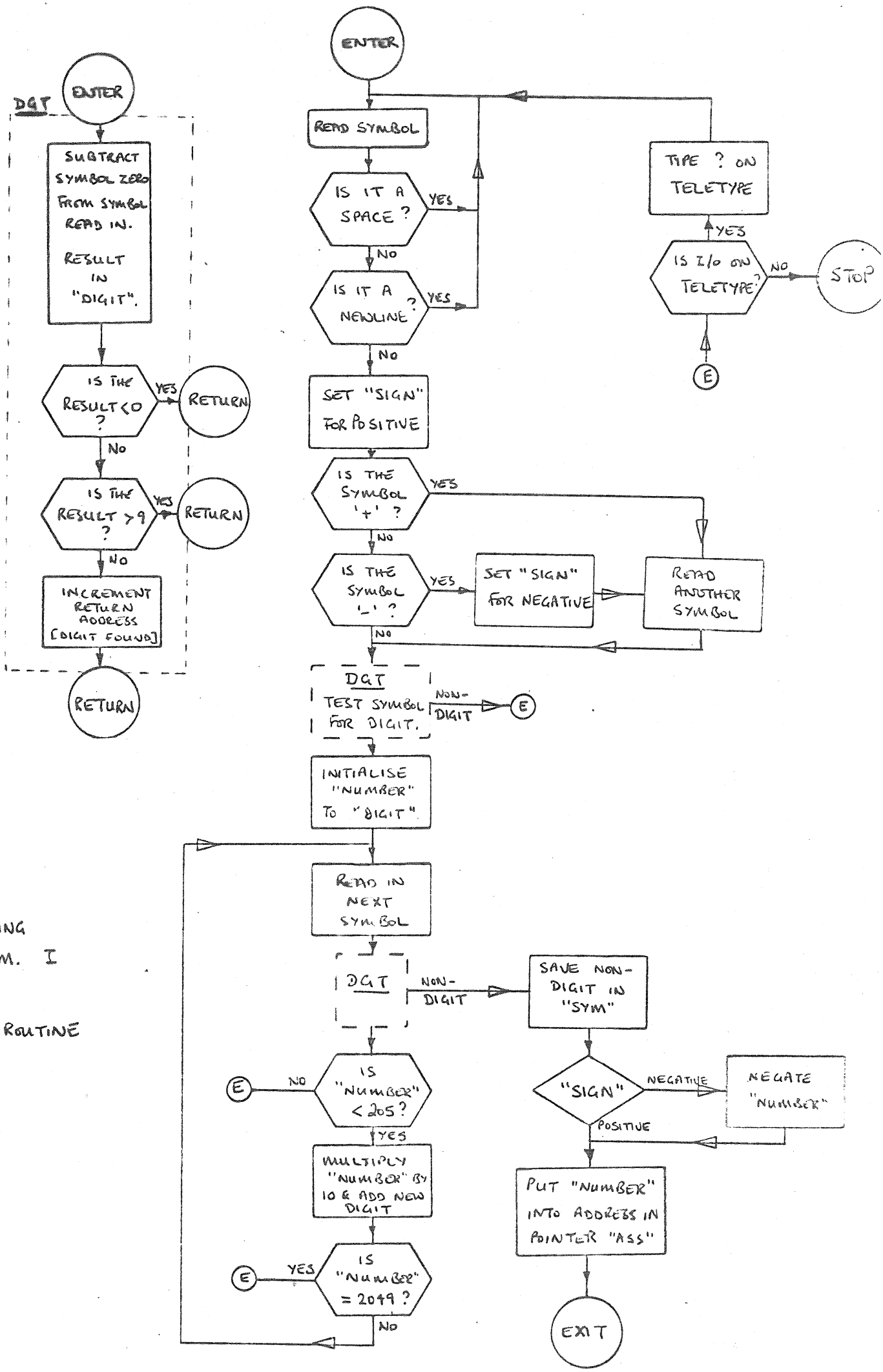
CALL	C
DEVICE	D
END	E
FUNCTION	F
HALT	H
LOAD	L
NEXT SYMBOL	NS
PRINT	P
PRINT SYMBOL	PS
PUNCH BINARY	PB
READ	R
READ BINARY	RB
READ SYMBOL	RS
RETURN	RN
SKIP SYMBOL	SS
STACK	ST
SUBROUTINE	SR
TEXT	T
UNSTACK	U
VALUE	V
WRITE	W

```
PDP-8 AUTOCODE /R5
/ EXAMPLES OF COMPILED SOURCE STATEMENTS

          X->24
JMS SETAC
24
TAD AX
DCA STP
          SWITCH #1(3,1,2)
JMS SETAS
A001
JMP I ASS
S001,
L003
L001
L002
A001,
          X0=0
JMS ADDRX
DCA I ASS
          4: ->SW#1(X0) IF XX0='*'
L004,
JMS PKUPX
JMS I SW
S001
JMS PKUPX
JMS PKUPX
DCA I STP
JMS SETAC
42
JMS SUB
SNA CLA
JMP I ASS
               X0 = X0+1
JMS ADDRX
JMS PKUPX
DCA I STP
IAC
JMS AD
DCA I ASS
               ->4
JMS SETAS
L004
JMP I ASS
          3: CALL #2(3,5)
L003,
JMS I INCC
TAD N2
DCA I STP
TAD N4
DCA I STP
JMS I CALL
C002
```

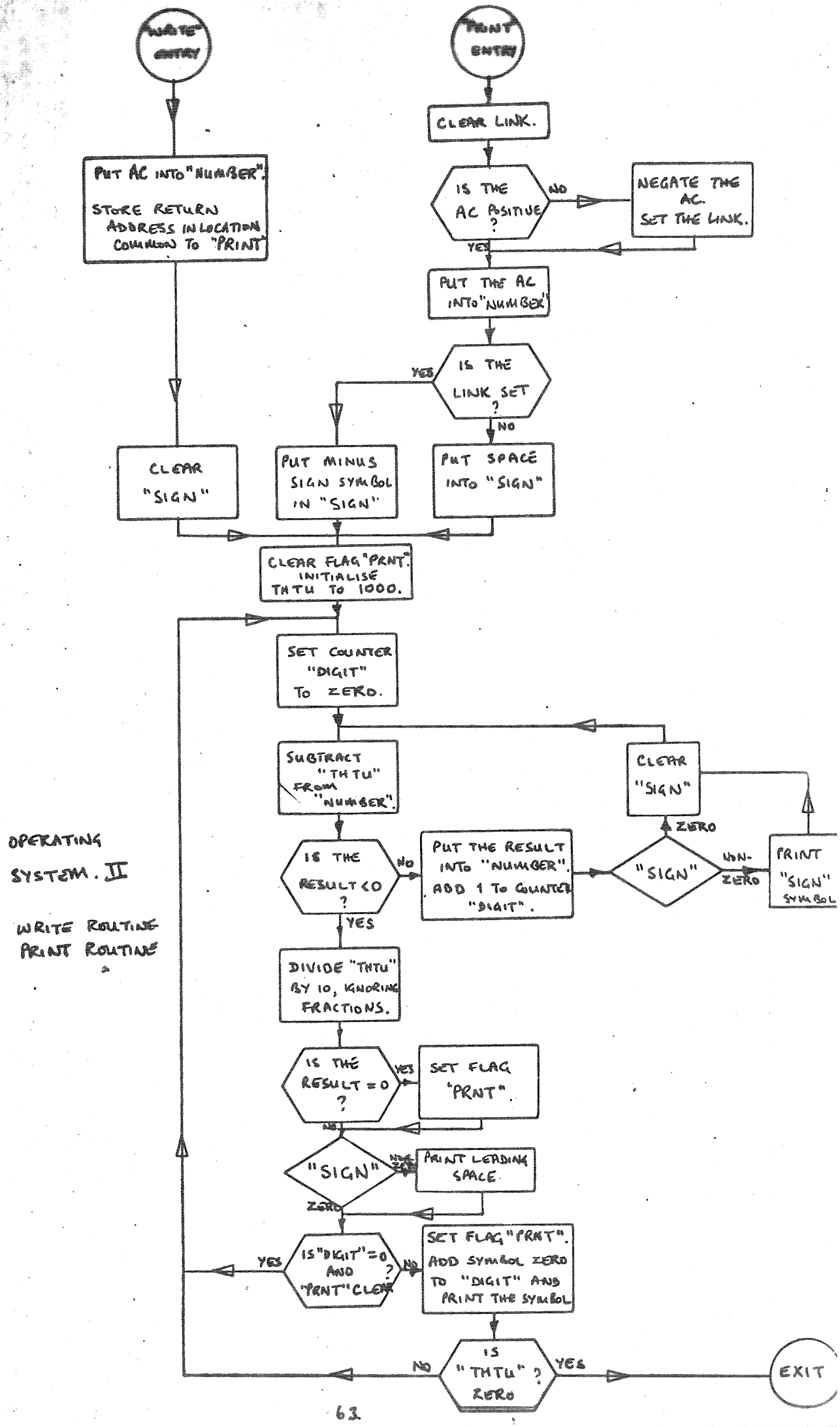
```
          1: X2 = F#2(F#2(3))
L001,
TAD N1
JMS ADDRX
JMS I INCF
JMS I INCF
TAD N2
DCA I STP
JMS I CALL
C0052
JMS I FN
DCA I STP
JMS I CALL
C0052
JMS I FN
DCA I ASS
          2: XX3 = X(X0-1) + X(Z3*3/X0) - !Z0-1!
L002,
TAD N2
JMS PKUPX
JMS ADDRX
JMS PKUPX
DCA I STP
IAC
JMS SUB
JMS PKUPX
DCA I STP
TAD N2
JMS PKUPZ
DCA I STP
TAD N2
JMS I MULT
DCA I STP
JMS PKUPX
JMS I DIV
JMS PKUPX
JMS AD
DCA I STP
JMS PKUPZ
DCA I STP
IAC
JMS SUB
SPA
CIA
JMS SUB
DCA I ASS
              TEXT( -! )
JMS I MSGE
10
33
0
```

```
          SUBROUTINE #2
HLT
C002,
          RETURN IF Y0<Y1
JMS PKUPY
DCA I STP
IAC
JMS PKUPY
JMS SUB
SPA CLA
JMP I END
          1: ->1
L021,
JMS SETAS
L021
JMP I ASS
          END
JMP I END
          FUNCTION #2
HLT
C0052,
          1: VALUE = Y0*Y0
L521,JMS PKUPY
DCA I STP
JMS I MULT
JMP I VAL
JMS PKUPY
DCA I STP
JMS PKUPY
JMS I MULT
JMP I VAL
          END
JMS WK
          $
HLT
AX0,0
$
```



OPERATING SYSTEM. I

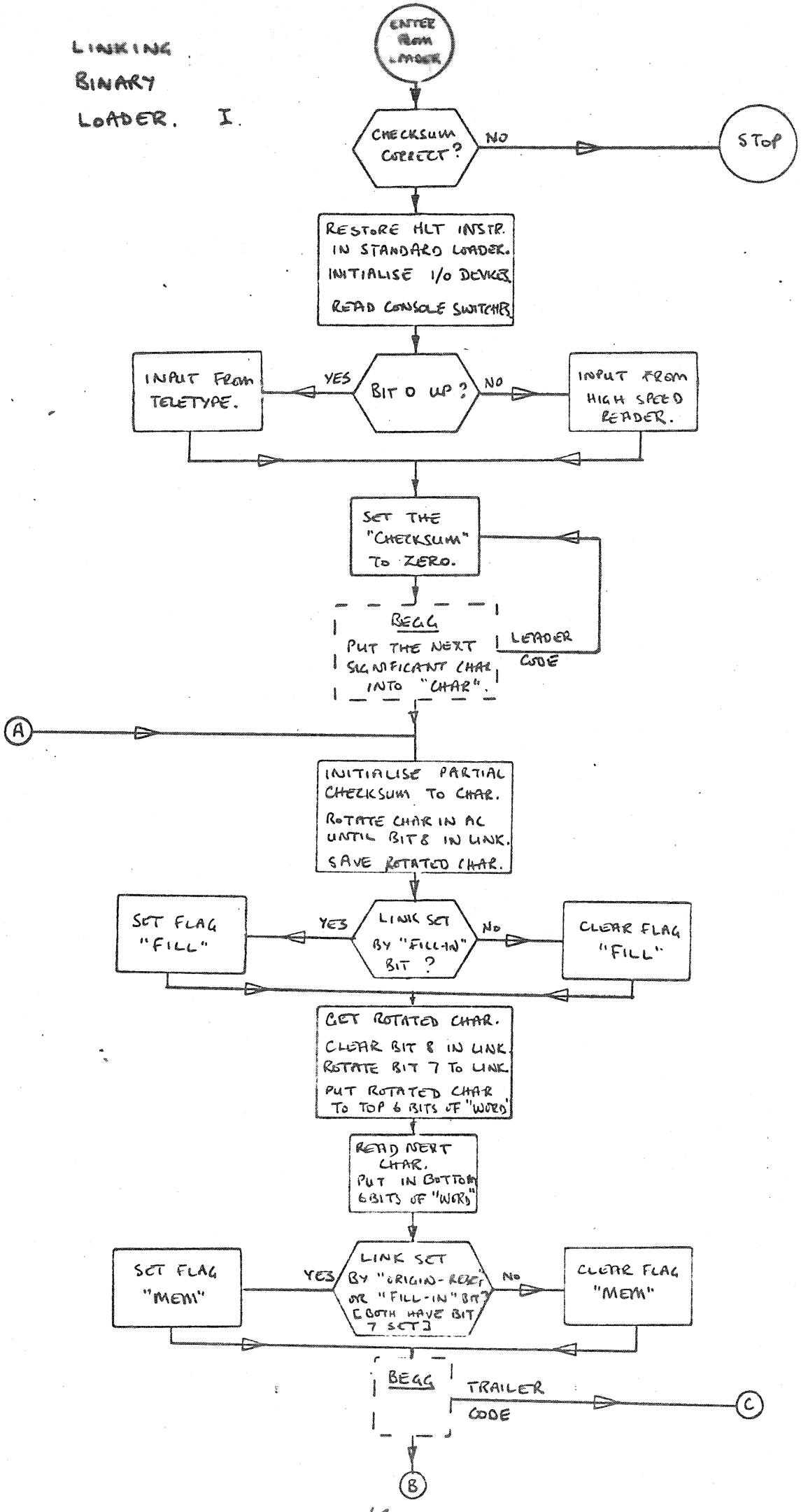
READ ROUTINE

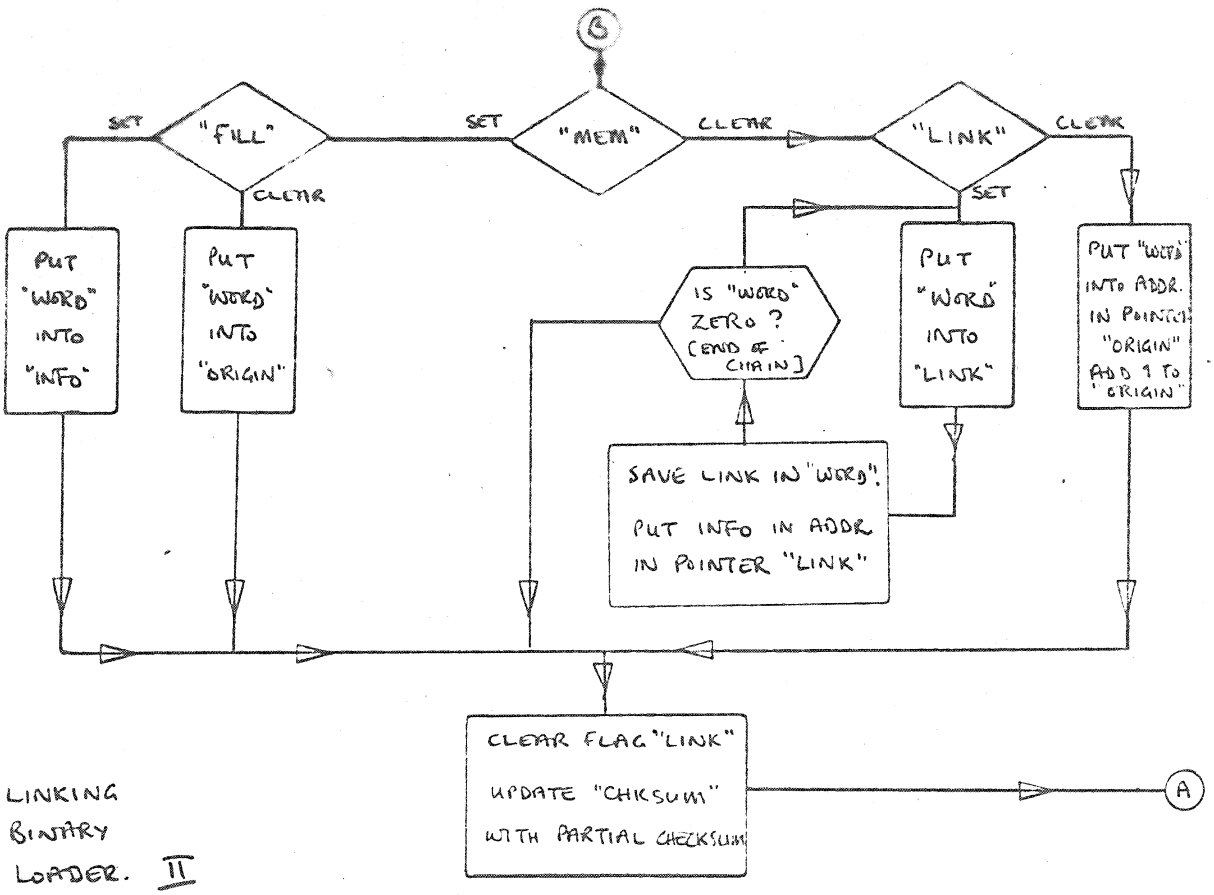


OPERATING SYSTEM. II

WRITE ROUTINE
PRINT ROUTINE

LINKING
BINARY
LOADER. I.





LINKING BINARY LOADER. II