Queen Elizabeth College
COMPUTER UNIT

K300 Computer Science project 1976

A General Purpose Macrogenerator and its applications

H. W. Thimbleby

Queen Elizabeth College, University of London
Campden Hill Road, London W8 7AH

K300 Computer Science project 1976

A General Purpose Macrogenerator

and its applications

H. W. Thimbleby

K300 computer science project

Acknowledgements are due to Brian Meek my project supervisor, Peter Brown for sending me the LOWL program tapes and documentation, Craig Johnson for allowing me access to the Elliott 905 at the Royal College of Art and to Joan Boneham for typing the script.

H.W. Thimbleby.

May 1976.

C O N T E N T S

PART II

A low language application of GPM

APPENDIX

1.1 Introduction

The main purpose of the project was to devise and implement a new language for the Elliott 903 computer.

The original plan was to develop the language translator using a high level language, which would be done on a separate computer. In order to facilitate a bootstrap onto the 903 an extended version of the General Purpose Macrogenerator was written.[4,7]

It was soon realised that the macrogenerator provided a means for translating the language LOWL[1], in which several systems programs have been written. These programs include ML/I[2] and SCAN[3,6] which themselves provide, or can be tailored to provide, the facilities that the new language covered. Since these programs are standard (they have been implemented on other computers) it seemed to be worthwhile to change the course of the project to encompass their implementation.

The bulk of this report is documentation for the General Purpose Macrogenerator, in its extended form, and the LOWL translator. The machine independent documentation of the LOWL software is fully covered by other reference manuals.

1.2 Obsolescence

The main emphasis of this report is the documentation of several well tested programs that were implemented during the course of the project. The programs are of some practical use; as such they are subject to revision from

time to time. However they will remain available with the features described here unless under some circumstances they are found to deviate from the behaviour that might be expected from their informal description. This is stressed because there may be logical errors in the algorithms. Revisions will always attempt to maintain the external artefacts of the software even if this engenders radical internal changes.

1.3 Notation

The characters used in describing the General Purpose Macrogenerator, "$? < > , ! ; ", are arbitrary and may be replaced by any other characters (excluding digits and "+" or "-") without affecting semantics.

In the few places where syntactic descriptions are given, Vienna notation has been used. A symbol that is underlined stands for what it describes, all other symbols stand for themselves. Meta syntactic marks therefore are not underlined.

Braces (curly brackets) enclose an item which may be repeated but must occur at least as many times as indicated by the subscript on the closing brace. If the subscript is omitted the item does not appear in all possible productions.

PART I

2. General Purpose Macrogenerator

2.0 The General Purpose Macrogenerator

The general purpose macrogenerator (GPM) is a program for symbol manipulation; it provides a powerful shorthand notation which can be used to great effect to extend assembly languages.

The GPM, in its original form, is described elsewhere.[4] However the version for the 900 series computer has several extensions which considerably ease its use.

The GPM normally copies characters from the reader to punch directly, occasionally printing monitor information on the on-line teleprinter. The direct copying stops when the GPM recognises a macro call; then, rather than using the reader the GPM reads the definition text of the macro. The macro, then, is a shorthand representation of its definition. For instance there might be a macro PI with definition 3.14159. Then any occurrence of a macro call to PI is effectively replaced by 3.14159.

A macro call has the following syntax:

$$\$ \ \underline{\text{macro name}} \ \left\{ , \ \underline{\text{parameter}} \right\} \ ;$$

where the meta syntactic marks "{" and "}" mean 'any number of times - including zero'.

Examples are:

```
$PI;          $ABS,-6;
$GOSUB, INPUT, CHAR;
```

Each parameter, including the macro name, may include further macro calls. These are then evaluated before the call containing them. Thus $$PI;; is a call to a macro

called 3.14159. It is often necessary to inhibit evaluation of parameters: to do this, text is bracketed between string quotes. Evaluation of text enclosed between string quotes is simply a removal of the outer pair of string quotes and a direct copying of the remaining text. String quotes are represented by matched angle brackets "<" and ">".

```
$GOSUB, INPUT, CHAR;
```

is a macro call to the macro GOSUB, with two parameters INPUT and CHAR.

```
$GOSUB,<INPUT, CHAR>;
```

is a macro call to the macro GOSUB but with a single parameter INPUT, CHAR.

The definition text of a macro may contain references to its actual parameters by formal parameters ?0,?1..... ?9, ?A.....?Z corresponding to the macro name itself, the first parameter, the second ..... the thirty fifth.

Suppose GOSUB has a definition text:

```
11 ? 1
 8 ? 1+1
 5 ? 2
```

Then the macro call $GOSUB, INPUT, CHAR; is evaluated to:

```
11 INPUT
 8 INPUT + 1
 5 CHAR
```

Macro calls such as this may considerably shorten the hand written program, and help ensure its accuracy. Suppose that such a program failed to work correctly. It

would be useful to discover the location of the fault, and to do that the macro GOSUB might be redefined to produce some helpful information such as a call to MESS or ZOUTI (which are string and integer output routines). When the program is running correctly, simply redefine GOSUB as before, process the source tape using the GPM and, lo, a debugged program with no debugging routine calls! Sometimes it might be better to have calls to a completely separate macro DEBUG. When the program runs correctly DEBUG is redefined as nothing and the source processed again.

2.1 Macro DEF

Macros are defined using a special inbuilt macro DEF which takes two parameters: the name of the macro to be defined and its definition. The macro name is the first and its definition the second parameter.

The macros PI and GOSUB would have been defined thus:

```
$DEF, PI, 3.14159;
$DEF, GOSUB, < 11 ? 1
              8 ? 1 + 1
              5 ? 2 > ;
```

Notice how string quotes have been used in the second definition. Before the macro DEF is called and the definition performed the parameters are evaluated yielding GOSUB and

```
11 ? 1
8 ? 1 + 1
5 ? 2
```

If there had been no string quotes then ?1 and ?2 would have been evaluated. That would only make sense if this definition itself occurred within another definition text or possibly in a parameter list of another macro call. The definition of a macro lasts until it is superseded or until it passes out of scope. Usually a definition is global and remains in force until it is superseded. However a definition that occurs within a parameter list only remains in force during the evaluation of the macro call and its corresponding definition text.

An illustration of these scoping rules is an interesting method (but pointless because there are more readable and efficient ways) for making choices based on equality:

if a=b then c else d

It is done as follows:

```
define  a  as  d
        b  as  c
call the macro a
```

Now if a and b are the same, then when b is defined this is a redefinition of a. When the macro a is called the result is c. If a and b are distinct then the definition of b does not supersede the definition of a which is therefore still d. So when the macro a is called the result is d, as required.

When a macro is redefined the previous definition does not 'evaporate' in the sense that the storage required for the definition is not released for other purposes, and consequently many conditionals based on the method

above would gradually use up all the available definition storage, with virtually useless information.
The macro definitions for a and b must therefore have a restricted scope; this way the storage required for the now temporary definition may be reused later for other purposes.
All this is achieved by

```
$a, $DEF,a,d;  $DEF,b,c;;
```

Before the macro a is called, its parameter

```
$DEF,a,d;  $DEF,b,c;
```

is evaluated. This creates two definitions of local scope. The macro a is then called and yields either c or d. Note that it is permissible to supply too many parameters, more than those required. (In fact an untidy restriction of the current version of the GPM is that macro definitions may not occur in a parameter that has a corresponding formal parameter that needs to be evaluated. This is due to the fact that a local definition (only) does have a side effect - it yields a value which has no external representation and therefore cannot be used.)

2.2 Macro COND

The alternative method is to use an inbuilt macro COND. COND is a special macro that can take any number of parameters, possibly more than 35 if needed.
A call to COND has this form:

$$\$COND,a \left\{,bi,ci\right\}_1^{,\ d;}$$

With the following semantics

if a = bi then ci else d (least i)

This gives exactly the same effect as

$a, $DEF,a,d; { $DEF,bi,ci; } ;

This is emphasised because the chosen ci or d is evaluated twice; once as a parameter of COND, then when the definition text of macro a is evaluated which is then a copy of the once-evaluated parameter.

Thus $COND,A,,,<< X >> ; yields X, not < X > .

2.3 Macro UPDATE

Frequently there is a need to redefine an existing macro, and as this may happen any number of times during processing it would be wasteful to create totally new definitions with their associated overhead in store grabbing. The inbuilt macro UPDATE overcomes this problem to a large extent.

The parameters for UPDATE are equivalent to those of DEF, but the macro must already exist and its new definition text must be no longer than the original definition text. This is because the storage once allocated for a definition cannot be extended, though it need not all be used.

```
$DEF, X, YZ;
$X;              yields YZ
$UPDATE,X,A;
$X;              yields A
$UPDATE,X,YZ;
$X;              yields YZ
```

There is a trap which the unwary may fall into, that of attempting to update a definition which is currently

being evaluated. This mistake is a terminal error when detected.

Consider -

X has definition text

A $UPDATE,X,$Z;;B

Z has definition text

YYYYYYYYYYYYYYYY

Depending on one's interpretation this should mean that the call $X; should update X as YY....Y and either yield AB or AY. Because of this doubt both cases are illegal. It is left as a trivial exercise to see how the problem can be avoided and yield whichever of the alternatives is required.

The GPM is a very powerful tool in the hands of an experienced programmer, and its generality does not suffer for the restrictions the monitor system imposes in order to make life safer, but possibly less exciting.

2.4 Macro BAR

The GPM also provides rudimentary facilities for integer arithmetic operations and for making decisions on their results.

The macro BAR can add, subtract, multiply, divide or find the remainder of two numbers.

$BAR, op, n, m;

calculates and yields n op m

'op' may be either +, -, *, /, or R for remaindering which is defined

nRm = n-m* (n/m)

The range of integer values permitted is -131072 to

+131071, and the numerical effect of overflow is undefined.

The representation of numbers has been chosen to allow 'special effects' - or tricks.

A number has the following syntax:

{neither-sign-nor-digit} {sign} {digit} 1 {non-digit}

There must be at least one digit. The characters preceding the signs and digits are ignored. The sign may be either '+' or '-' which are treated as monadic operators:

- number gives the negative of the number
+ number gives the number directly

Thus '--1' has the same value as '+1' and '1'.

This representation is only significant where ultimately it leads to a call of an inbuilt macro that makes use of the numeric value. Elsewhere the interpretation given to numbers is external to the GPM.

The macro definition text containing

2 -?1

may well generate '2 -+1', which probably was not expected.

The result of BAR is a left justified number signed only if it has a negative value, with no trailing spaces.

2.5 Macro LEG

The macro COND is not sufficient for testing ranges of numeric values; for instance for a macro call to evaluate 'A' if its parameter is between 1 and 100 would require a call to COND with 202 parameters. The inbuilt macro LEG overcomes this problem:

$LEG, number,l,e,g;

results in l, e, or g evaluated depending on whether the number is less than zero, zero, or greater than zero respectively.

2.6 Macro VAL

A macro that sometimes simplifies programming, though it is rarely necessary, is the inbuilt macro VAL which takes a single parameter. VAL yields the unevaluated definition text of the macro corresponding to its only parameter.

2.7 Macros NOTE, TRACE, UNTRACE

The three remaining macros are to help in monitoring the progress of macro processing.

NOTE takes a single parameter which is immediately output on the on-line teleprinter in the form:

**MONITOR: (parameter)

TRACE takes a single parameter which must be a macro name. Subsequent calls to that particular macro, as opposed to calls to previous or subsequent distinct definitions with the same name, cause simple tracing information to be printed in the form

ENTERED (macro-name)
ARG: (parameter-1)
ARG: (parameter-2)
: :
: :

The macro UNTRACE returns a macro to its default state: no tracing information is given when the macro is entered.

2.8 Layout

In the practical use of the GPM, macro calls and macro definitions may straddle several lines, and this may be inconvenient.

A character "!" followed by consecutive new lines is evaluated to nothing.

```
$DEF,A,!ALL ON!
ONE LINE;
$A; yields ALL ON ONE LINE.
```

2.9 Diagnostics

When the GPM monitor system detects a mistake the monitor gives a fault message, a trace back giving details of the circumstances that lead to the fault and a list of all the current macro definitions.

An example would be

```
**MONITOR:          UPDATE TOO LONG
ENTERED:            (UPDATE)
ARG:                ( X )
ARG:                (YYYYYY)
NOT ENTERED:        (OOPS)
**MONITOR:          GPM ABORTED
CURRENT MACROS
---                 ( X )
VAL:                (SHORT)
---                 (OOPS)
   :                  :
   :                  :
```

2.10 Error messages

STACK OVERFLOW

There are too many definitions or some macros are deeply recursive - in which case the trace back information can use up a lot of paper.

UNMATCHED ;

The semicolon was not preceded by a dollar sign but occurs in a position where it ought to initiate a macro evaluation.

UNQUOTED ?

The question mark occurs where there are no possible actual parameters. It occurs in the source text in an evaluable position.

BAD ARGUMENT NUMBER

The character following a question mark is not alphameric.

NO ARGUMENT n

Actual parameter n was not supplied in a call to a macro requiring it.

INCOMPLETE CALL

The definition text has terminated before a macro call initiated within it was completed. This often occurs when a semicolon is missed out in a macro definition, e.g. $DEF,A, < $B> ; $A;; is illegal and is not a call to macro B.

UNDEFINED MACRO

An attempt has been made to call an undefined macro, or refer to its definition text.

UNMATCHED >

There is no corresponding opening string quote.

UPDATE TOO LONG

The new definition for a macro is longer than its original. In the current macro listing the definition text of the macro is given as if the update was not attempted.

BAD NUMBER

An attempt has been made to use a non-numeric string where a number was expected.

INVALID CALL

This occurs when the effect of an inbuilt macro is not defined. This happens under these circumstances:

1. BAR not supplied with +, -, *, / or R.
2. VAL given an inbuilt macro.
3. an attempt has been made to substitute an evaluated call to DEF for a formal parameter.

UPDATING ENTERED MACRO

A macro has directly caused its own updating before completion of its call.

WARNING CHARACTERS EQUAL

It is possible to define a new set of the characters " $, < > ? ; ! ". This message is given when the same character has multiple meanings.

The method that characters may be changed is described below.

2.11 Running the GPM

The GPM is distributed as a single SCB tape which may be loaded under initial instructions in the normal way. On completion of a successful load the GPM identifies itself,

gives its version number, a list of entry points and all the inbuilt macro names.

There are three entry points:

32 - run the GPM

All user definitions of previous runs are destroyed unless the GPM was halted by reading a halt code character. Thus entry at 32 corresponds to the normal 8 and 9 entries for "enter" and "continue", but the choice is made by the GPM itself.

33 - give trace back

Entry at 33 stops the GPM and enters the monitor system to give a standard trace. The GPM itself is aborted.

34 - change warning character set

A list of the new character set should be punched on the tape and read by the reader. The on-line teleprinter outputs a table of the standard set against the last and new sets of characters. If any warning character is defined as carriage return then this is given in the table as NONE. There is no external representation of NONE; the character and its effects are now inaccessible.

The GPM accepts the full ISO character set, all characters are significant except -

runout, rubout, carriage return

which are all ignored.

The halt code character halts the GPM processing, but is otherwise handled as any other character. It may even be defined as a warning character.

The GPM does not check for even parity: this allows macros to generate binary data or legible titles on tape. Although hardware may ignore parity the GPM treats odd and even representations of the same character as distinct. This may give rise to curious messages such as "UNDEFINED MACRO (DEF)" because the characters D,E,F do not all have even parity in the call or reference.

The default stack size is 3K words, which is quite adequate for most purposes. The GPM may be configured to give a larger stack, but the advantage of this restriction is that the GPM never corrupts the contents of the high end of core. Consequently the GPM may be loaded together with other programs, the assembler in particular. The standard entry points are not corrupted, and these may still be used.

The GPM is also available as a symbolic SIR tape, for those who wish to configure their own systems. The next section describes the GPM's working in sufficient detail for it to be altered to suit new needs.

2.12 The GPM interior

This section cannot be understood clearly without reference to part 2 of C.Strachey's paper.[4] Nevertheless reading this and the source program should be sufficient

to make minor changes, such as changing the stack size, making the program operate on a different interrupt priority or changing the transput facilities, for instance.

Due to its bulk the source program does not contain (many) comments.

2.13 Fundamental differences

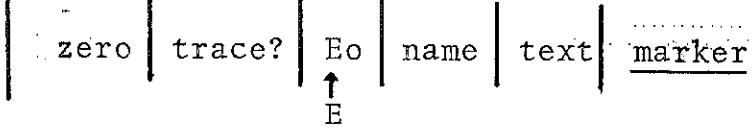
1. The string quote depth counter, Q, is always one less than in the CPL program. Thus text is evaluated when Q=0, not 1.

2. Warning characters have an internal representation, which is independent of any external representation.

$ is represented by value	-2
<	-3
!	-4
?	-5
;	-6
,	-7
>	-8
marker	-9

There is no significance in the actual values themselves, other than being negative.

3. An item on the environment chain has two additional words:



The zero serves as an 'illegal' character to trap attempts to use the definition as text and the trace flag is normally also zero but is set to -1 after a call to TRACE. The first zero will only occur if the definition is isolated and not preceded by another definition on the stack. The sequence marker/zero never occurs but appears as marker.

4. The vector ST is a label with value 10. All references to ST are therefore ten less than the actual machine address. Hence ST(P - n) is in SIR

```
O P
/4 ST-n
```

when n $\leqslant$ 10.

The storage allocated for the stack actually comes after the program constant pool, so the values of stack pointers are at least 2K odd.

5. The monitor system is completely different, and is entered by jumping to label Mn where Mn is a suitable diagnostic section. The monitor system never returns to the GPM proper - all entries cause the GPM to abort.

2.14 Program commentary

1. Defining new warning characters.

   Characters 0, and 255 are ignored.

   In writing the table on the teleprinter characters 10, 141 and 160 are represented (in full) by NEWLINE, NONE and SPACE.

2. Program entry at 32.

   A has the last character read; if it was halt (20)

control returns to the input routine. Otherwise the environment is reset:

H:=P:=F:=C:=Q:=0

E:=SETE

S:=SETS

and control transfers to the START loop.

The section labelled NEXTITEM comes before START as this eliminates a jump at the end of the section.

3. The conditional goto on warning characters in the CPL program is replaced by a direct case switch if A is negative. The last element of the switch table is not a jump but an entry to FN.

4. APPLY

   After looking up the macro using FIND, ST (A-1) is examined to see if the macro is being traced.

5. LOADARG

   Alphabetic formal parameters are also permitted.

6. ENDFN

   If F > P goto M5.

7. VAL

   VAL checks that the value of the macro is not negative; if it is it must be an inbuilt macro which, of course, has no external value.

8. TRACE

   sets ST (A-1) to -1.

9. UNTRACE

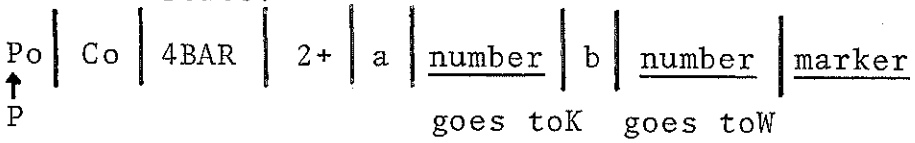
   sets ST (A-1) to +0.

10. UPDATE

    Uses W to enter FIND and runs down the P- chain to check that the macro has not been entered.

11.BAR

Initial state:



DEC is used to put the result on the stack.

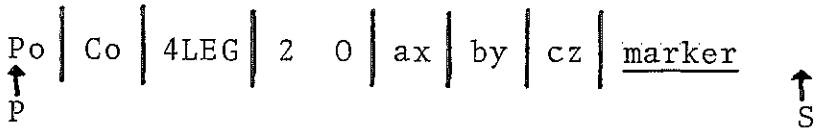
12.NOTE

Calls the monitor routine ITEM which prints a LID (length+identifier, e.g. 4/DEF) on the on-line teleprinter, between matched round brackets.

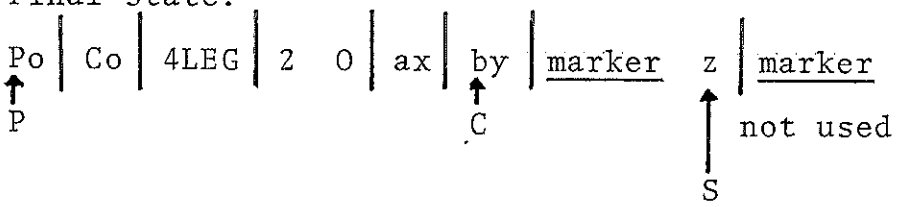
13.LEG

Calls BIN, and chooses parameter. C is then set to point to the first character of that parameter and the end of the parameter is set to marker. LEG then exits to START, not ENDFN. The START loop then uses as text the parameter of LEG as if it was a definition text. When marker is read ENDFN is used to delete LEG from the stack in the normal way.

Initial state:



Final state:



14.COND

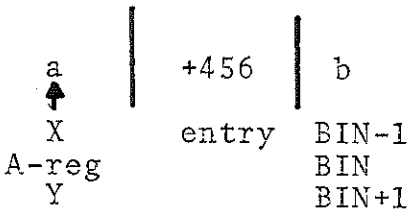
The action of COND is much the same as LEG, but the parameter is chosen in a different way.

15. Subroutine BIN uses LINK as its return address. If the A-register points to the start of a LID containing a number, entry to BIN will decipher the number and return with its value both in W and the A-register. X is set to point at the start of the next LID, or marker if there is none. If BIN is entered at BIN-1 then the A-register is loaded from X. If BIN is entered at BIN+1 then it is assumed that Y points to the LID.

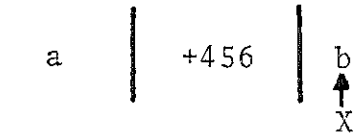
BIN starts like this:

```
        4  X
BIN     5  Y
        4  ...  (A-reg loaded)
```

Initial state:



Final state:



W=A-reg=456

16. DEC is used to load a number onto the stack, and return to ENDFN.

DEC starts like this:

```
        4    W
        2    +0
DEC BIN 5    W
        4    ...
```

17.CHECKARG

Uses a backward table ARGS to check that inbuilt macros are called with at least the correct number of parameters. Exit is either to M4 to give a fault message or a switch to JIM (jump if minus) which is a table of gotos to the various macros.

18.ITEMP

Uses the routine ITEM to print the macro name at the head of the P-chain.

19.MONITOR

Calls routine WRITE to print "newline, **MONITOR:" and then calls WRITE again to output its own parameter.

20.WRITE

Simply outputs its parameter to the on-line teleprinter.

Method of entry:

```
        11    WRITE
         8    WRITE +1
£TEX        £T↑"
```

Which would print "TEXT, newline." ' ↑ ' represents a newline and '"' marks the end of the parameter.

WRITE will automatically give a newline if the end of the current line is reached, the variable CHARS is incremented each time a character is printed. There is a little free space: the newline is given before the physical end of line, this allows output such as

NO ARGUMENT n

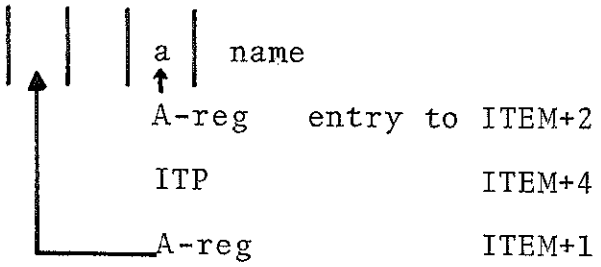
not to need to increment CHARS when n is printed, (n will be the character in ST(C)).

WRITE may be used to output single characters and take newlines as required. Before entering WRITE for the first time set MASK:=0.

To output a newline enter at NEWL, otherwise at OP-2, with the character in the A-register. Control will return to rv WRITE+2.

21.ITEM

Uses WRITE to print the characters of a LID, character by character.

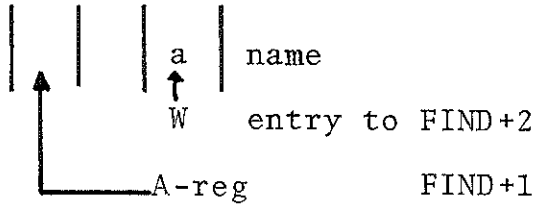


If a=0 then, when ITEM was called, the LID cannot have been completed. ITEM then gives all characters upto ST(S-1) and prints "...INCOMPLETE".

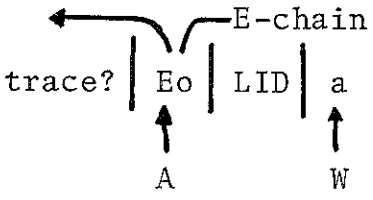
22.FIND

FIND looks up a LID in the E-chain. If it has not been defined the monitor system is entered.

Initial state:



Final state:



23.LOAD

Depending on H, loads a character onto the stack or punches it.

To load A-register enter at LOAD+1

To load A LOAD+2

On output, warning characters are converted to their external representation.

24.NEXTCH

Pops a character from the stack or reads from the tape reader. Character values 0, 141 and 255 are ignored. A halt code causes a dynamic stop inside the routine; if the GPM is re-entered, A=20 and control jumps to the instruction after the dynamic stop.

On exit A and the A-register both contain the value read. Warning characters are converted into their external representation.

25.HELP

Returns to rvLINK+1, entry is HELP.

HELP prints a single line of trace back information. Before entry, set Z:=P,Y:=F,K:= maximum number of parameters to be printed, including the macro name itself. On exit K=1 and either Z or Y will have advanced down their chains.

26.Monitor system entry points:

M0	Stack overflow
M1	Unmatched;
M2	Unless H=0, unquoted warning character
M3	Bad parameter number
M4	Parameter missing
M5	Incomplete call

M6 (not used)

M7 Undefined macro name

M8 Unmatched

M9 Update too long

M10 Bad number

M11 Give trace back

M12 Abort, list current macros

M13 Invalid call

M14 Multiple definition of warning character

M15 Updating entered macro

All entries give a message and goto M11 except M11 which goes to M12, and M12 which enters a dynamic stop.

2.15 Procedure for adding new macros

Four things must be done to add a new "inbuilt" macro.

1. The environment chain must be extended and the pointer SETE adjusted accordingly.
2. The number of parameters (minimum) must be added into the table ARGS, as a negative number.
3. A jump must be added into the table JIM.
4. The macro machine code must be added somewhere in dead code.

2.16 Assembling and loading

The program has the following map:

first segment	entry points set warning characters GPM proper monitor system some routines initial inbuilt environment constant pool
second segment	start of stack identification of program

There is a global SETS, which on assembling the second segment is set to the word beyond the end of the constant pool of the first segment offset by 10.

The second segment also contains the program identification a call to WRITE, which is overwritten as soon as the GPM is entered because it is at the base of the stack.

To assemble the GPM, enter at 8 and note the first and next messages. Enter at 9 and note the first and next messages.

The sections of core that are dumped to obtain a single SCB tape are then from the first first to the first next, and from ENTRY to the second next. The program should finally trigger to ENTRY.

2.17 Examples

1. SIR integer division

```
$DEF,DIV,< 14 8176
           13 ?1
           14 8191 > ;
```

2. Expanding for statements

Problem to define a macro FOR which produces several copies of a piece of text,

```
$FOR,I,1,4,< 4 X$I;
             5 P+$I;
           >;
```

to generate

```
4 X1
5 P+1
4 X2
5 P+2
4 X3
5 P+3
4 X4
5 P+4
```

Solution:

```
$DEF,FOR,< $SUBFOR,?1;!
$BAR,+,0,?3;,?4,
$DEF,?1;XXXXXXX;
$UPDATE;?1,$BAR,+,0,?2;;> ;
$DEF,SUBFOR,< ?3
$COND,$?1;,?2, <> ,
< $UPDATE, ?1  ,$BAR,+,1,$?1;;
  $SUBFOR, ?1  ,  ?2  , ?3  ;
>;>;
```

3. Formal differentiation

Problem: given a prefix expression, print its derivative, e.g.

$D,$+,$X;,$Y;;;

is to yield

D/DX X+Y = 1+DY/DX

Solution:

```
$DEF,X,<X,1>;
$DEF,Y,<Y,DY/DX>;
$DEF,D,<D/DX?1=?2>;
$DEF,+,<(?1+?3),(?2+?4)>;
$DEF,*,<?1*?3,(?1*?4+?2*?3)>;
$DEF,SIN,<SIN(?1),?2*COS(?1)>;
```

How this works is left as an easy exercise for the reader. How could this idea be extended (i) to remove singular cases, e.g. 1*X or X+0, (ii) to eliminate redundant brackets e.g. X*(Y*X).

(Hint: only convert to infix notation when printing the result)

4. Delayed SIR code

Very often it is inconvenient to write out-of-line jumps as this decreases legibility.

Compare two versions of a routine to count the number of set bits in a word.

```
            0   +0      (clear B and Q-registers)
            4   WORD
LOOP        9   MSDSET
            7   CLEAR
SHIFT      14   1
            8   LOOP
```

```
MSDSET      10   1        (increment B-register)
             8   SHIFT
CLEAR        4   1
             0   LINK
            /8   1        (return)
```

with this:

```
             0   +0
             4   WORD
LOOP         9   $LOC, 10 1
                      8 SHIFT;
             7   $LOC, 4  1
                     0  LINK
                     /8 1;
SHIFT       14   1
             8   LOOP
             $POOL;
```

The macros LOC and POOL might be defined as follows:

```
$DEF,POOLT, a very long string;
$UPDATE, POOLT,      ;
$DEF, POOL, < $POOLT;
$UPDATE, POOLT, < > ; > ;
$DEF, NAME, 1       ;
$DEF, LOC, < L$NAME;
$UPDATE, POOLT, L$NAME; ?1
                    $POOLT;;
$UPDATE, NAME, $BAR,+,1, $NAME;;;>;
```

PART II

A low language application of GPM

3.1 The LOWL translator

LOWL is a low level machine independent language.[1] A LOWL program can be regarded as a sequence of macro-calls which are to be translated into a particular assembly language.

Machine independence is achieved by two devices. Most of the macros have supplementary arguments some of which may be redundant for a particular target machine and all structure manipulation such as calculating addresses is done from a basis of constants. The constants specify the machine and are set to represent values, the number of bytes per word for instance.

The LOWL translator for the 903 takes as input a program written in LOWL and compiles this into a complete self-contained program in SIR.[5] The LOWL translator has two passes; first the LOWL program is preprocessed into a different format, secondly the processed program is translated by an extended version of the GPM.

There are several programs written in LOWL, and all will run within the 8K words of the 903.

They are -

1) LOWLTEST — this program tests the translator, and gives suitable information if any errors are found.[1]

2) ALGEBRA — a demonstration program for students learning or researching into logics.[1]

3) UNRAVEL — produces 'intelligent' core dumps.[8]

4) ML/I — a powerful general purpose macrogenerator.[6,2]

5) SCAN — a simple textual analysis programming language.[3,6]

Of these programs only ML/I and UNRAVEL present problems because of their size. ML/I cannot be assembled in 8K because its length exceeds the core space that is available when the loader is also in core; and UNRAVEL, although smaller, is a bit pointless because it occupies so much core that very little else is left to be dumped!

3.2 Using LOWL

The preprocessing program is written in Algol 60; it is entered at 10 and simply reads the LOWL program tape and simultaneously punches a processed tape suitable for the next pass.

The reason for this first pass is that the first few programs to be translated were only available in the processed form.

The second pass does the real work of translation. It should be entered at 32 with the correct run-time system in the reader. The different LOWL programs have different run-time systems so the correct tape should be used at this stage.

After reading the run-time system the translator is entered at 32 again to read the processed LOWL program. The tape punched by the second pass is a complete SIR program which may be assembled in the normal way (unless it is ML/I).

3.3 Translator efficiency

Unfortunately translation from LOWL to SIR is not trivial, mainly because the 903 uses a reverse subtract (negate and add) instead of the LOWL forwards subtract. LOWL also uses three conceptual registers; the 903 has only two that can correspond to them.

The program ALGEBRA is the shortest piece of LOWL software excluding LOWLTEST, and was translated to see where the main inefficiencies arise.

When a subtract instruction is translated, a reverse subtract instruction is generated. Thereafter the contents of the SIR register will be minus the contents of the corresponding LOWL register. The LOWL translator will leave the SIR register in this state as long as possible; negating the register only when the LOWL logic necessitates the correct sign, as when the register is stored.

The first translation of ALGEBRA required 1590 words for the program, and of that 69 were redundant and might be removed by a single pass peephole optimiser. 29 of these were an unused part of the run-time system which is not required by ALGEBRA.

This gives an expected redundancy in translation of about $2\frac{1}{2}\%$.

The table below gives an indication of the frequencies and causes of the redundancies.

Table 3.1

Cause	Space
Unused run-time system	29
Negating before adding a constant	18
Re-loading modifier	7
Multiplying variable by one	4
Some run-time system could be open	3
Unused run-time system exits	2
Negating before labelled load	2
Redundant indirect load	2
Redundant load after label*	1
Subtracting a constant	1

* Not redundant until a preceding redundant negate was removed.

It should be noted that the other LOWL programs require extensions to the basic run-time system rather than not needing it all as ALGEBRA.
In ALGEBRA a phenomenal saving can be obtained by detecting special cases such as MESS'' which occur frequently, and could be translated into parameterless special subroutines, rather than the general subroutine MESS with one parameter.

3.4 Processed format: Pass one

Basically the second pass is a GPM using warning characters

colon and semicolon for the standard dollar sign and semicolon.

The full details of the transformation are as follows.

1) All instructions and subsidaries (OF and RL) must be written between a matching pair of colon and semicolon with their parameters separated by commas. Redundant spaces must be eliminated.

2) Labels must be written between ':LABEL,' and ';'
   Thus

   [BEGIN]

   becomes

   :LABEL, BEGIN;

3.5 Translation: Pass two

Machine dependent constants occur only in the macro OF in a form no more complex than

:OF,S*S ± S*S;

Where 'S' represents either an unsigned integer or one of the names -

LNM,LCH,LICH	taken as 1
LHV	taken as 32

The quote symbol is a new warning character and indicates to the LOWL translator the delimitation of a character or character string where, obviously, no other warning characters may be recognised.

Unpacked strings are translated by the macro STR, into an internal code designed to simplify some of the LOWL operations on characters.

A digit is represented directly by its value. A letter is represented by its ISO value but with bits 18 and 17 set, i.e. its value plus 600000 octal. Any other characters are represented by their ISO value (with even parity) plus 500000 octal, or bits 18 and 16 set. Thus bit 18 determines whether the character is a digit;if it is not a digit bits 16 and 17 determine whether it is a letter or not alphameric.
Strings for MESS are packed three characters per word using the macro MESSX. The format is that of SIR alphanumeric groups and is terminated by a quote '"'. Note that the dollar represents a newline in LOWL and is translated to 500215 by SIR and up-arrow by MESSX.

LOWL has a pseudo-instruction

IDENT variable-name, integer

to equate a name with a number. The name is then used freely where an integer is permitted. The problem this causes is overcome using the macro 'S', which is the inverse of IDENT, and converts named constants into integer form. 'S' is required as there is no equivalent facility for naming constants in SIR.

The LOWL translator must keep track of the sign of its SIR register. This is done by an internal flag which can take on values for 'register negative', 'register zero' and 'register positive'.

A special warning character '@' (at) or '\' (grave) is used as follows:

@N	set flag to negative
@Z	zero
@P	positive
@I	change sign
@B	set flag to positive and generate "2+0" if it was negative.

The macro 'A' chooses one of its three parameters according to the flag status.

Various LOWL instructions translate into especially simple SIR instructions. The table below lists all the cases that are recognised and treated specially.

Table 3.2

BUMP	with 0, 1 or 2
AAL	0
SUB	0
MULTL	with 0, 1 or 2
GOSUB	run-time routine system
LAL	0 following CLEAR
LAV var,R LAI var,R	after CAL 0

3.6 Transput

The final LOWL translator, as described, above was exceedingly slow. However its speed was increased to an acceptable level by changing the input-output routines. Originally the translator read a macro-call and then generated the output translation. This meant that the reader and punch were alternately standing idle. By buffering the input, and interleaving reader and punch use whenever possible, neither peripheral need stand idle for so long, and considerable savings result.

The general method is described below:

On the 903, too frequent peripheral transfer requests cause the processor to hold up.

If a transfer is attempted before the peripheral has finished handling the last transfer, there is no way that the program may continue and there is no way that this situation can be detected.

It is up to the program to time its transput so as not to attempt transfers too frequently.

In general it is not a simple matter for a program to generate requests at a suitable rate in the absence of a real time clock, which is not available on the 903. However the device response time does give a program a known delay; suitable alternation between peripherals will give a maximal data transfer rate, irrespective of the behaviour of the program between transfers.

The paper tape punch has a response time of 9ms, and the reader 4ms. A program that reads n characters and then

punches m characters in cycles will require

$$4(n-2) + 9(m-1) \text{ ms per cycle,}$$

assuming n-2 and m-1 are positive.

But if the transput is suitably organised the same number of characters can be transfered in

$$\max(4n, 9m) \text{ ms per cycle.}$$

To achieve this means that whenever possible the transput routines must read two characters for every one punched, though not necessarily in that order.

Since the program is unlikely to conform to this strategy the transput routines must queue excess characters until they can be handled by peripherals or program.

Suppose the transput routines have a common variable which by its value indicates the next actual peripheral transfer that should be attempted.

What action should the routines take if the program request is conflicting?

1. The output routine could:
   (a) queue the output character, to be output later
   or (b) queue some input, to be used by the input routine later
   then output its character
2. The input routine could:
   (a) output some of the output queue
   then input its character.

or (b) take a character off the input queue

These actions must be modified when the queues are either empty or full; this situation means relaxing the ideal alternation between input and output for a time.

Suppose the two queues are both half full, then it is readily apparent that both routines need only work on the same queue for the desired effect.

Indeed, with the input queue neither full nor empty there is no need for the output queue at all, and vice versa.

Thus there is only need for a single queue assuming that it is sufficiently large, and contains a sufficient reserve of characters at all times.

There is only need for one queue, either for input or output only, but it must be initialised with some reserve.

Without more knowledge of the program, say, that it starts by printing a title page which could be used to build a reserve in an output queue before ever the input routine is called, then it is only possible to build this reserve in the input queue.

Because building this reserve now involves peripheral transfers, it is actually a waste of time to deliberately set up a reserve.

If one queue becomes full, the action the program might take if it had two queues would be to queue characters

in the other queue (for the other peripheral) until some characters could be removed from the first. But if there is storage available to extend the second, why not use it for the first?
The routines only need one queue. Which: input or output? To decide the issue, there is additional information: if the paper tape punch is not used for several seconds, it powers down and will then require a significant time to return to running conditions. Characters for output should be punched as soon as possible. If there is an output queue, this simply increases the potential delays between punchings and therefore the probability that the punch powers down.

In the programs described in this report, input queueing has been used with considerable increases in run-time speeds.
The input queues have a maximal length of 128, the smallest power of two larger than the length of a line. The increase in speed is about 30% for programs with a unity expansion on data, the number of characters punched being the same as the number read.
Note that this algorithm is only relevant for systems that have no alternative to non-autonomous peripheral transfers.

3.7 Summary of LOWL

Statements in LOWL are written one to a line and possibly preceded by a label which is at most six characters in length and enclosed in square brackets.

Mnemonic operation codes are preceded by a tab, and if there are any arguments they are followed by a tab and a list of arguments separated by commas. Each operation code has a fixed number of arguments, some have none.

Arguments that are literal strings are enclosed in quotes ('). No argument is ever null; instead the letter X is used to indicate that an argument is not applicable.

Blank lines are used to improve program layout.

Some LOWL statements have supplementary arguments which are used to convey extra information about the statement, for example a jump statement might be written

```
GO    L,200,E,T
```

This means jump to label L which is 200 statements later. The E means that the jump leaves a subroutine and the T that this jump is in a switch sequence.

Almost all statements in LOWL involve a storage address and may use one of three notional registers as follows.

A is the numerical register

B is the index register

C is the character register

LOWL statements may have constants for arguments which may either be numerical, character, mnemonic or a call to the macro OF.

The macro OF is used to evaluate expressions using mnemonic constants that help describe the LOWL mapping. The argument of OF is an expression of the form

S*S ± S*S

or simpler, for instance

S*S-S

S + S

The 'S' represents either an unsigned integer or one of the mnemonics:

LCH	the number of storage units occupied by a character
LNM	the number of storage units occupied by numerical data
LICH	=1/LCH

These mnemonics each have a value of one for the 903 LOWL translator.

There are additional mnemonics for some LOWL programs, LHV represents the size of ML/I hash tables (32) for instance.

3.8 LOWL instructions

1) Data types	Character
	Number (integer or pointer)
2) Variables	Represented by an indentifier
3) Constants	Numerical: integer or a call to OF.
	Character: a single character in quotes

4) Registers Three: A,B and C

5) Labels represented by identifiers

6) Subroutines at most one argument

The basic form of LOWL has 60 different instructions which are described in the supplementary material to this report.

In addition to the basis of 60 statements, each LOWL program (excepting LOWLTEST) have a few additional statements.

In the basic LOWL there is a statement

MESS string

which should output string to the monitor stream. In the program ALGEBRA this idea is extended to include

RMESS string

QMESS string

where the string should be printed on the result and question streams respectively.

The other LOWL programs have many more extensions.

3.9 The run-time systems

The run-time system for each LOWL program consists of code written directly in SIR and occupies between 700 and 1K words for the different programs.

The run-time system can be divided into six sections:

1) Initialisation

2) Transput routines

3) LOWL statement routines

4) Machine dependent program routines

5) Interrupt handling

6) Dumping routine

Initialisation

The initialisation code has all the entry points to the program, and at the start of a session identifies the program and its version number.

The input/output buffers are set up and pointers are set to delimit the LOWL program stacks.

Transput routines

These routines perform all input/output and character code conversions. The routines may use buffers.

All formatting is performed by these routines, coping with tab characters and lines that are too long to be printed in one go.

LOWL statement routines

Many LOWL statements require operations that are too complex to be mapped into just a few SIR instructions. These cases are dealt with by subroutines in the run-time system.

Examples are the stacking operations, block moves and logical operations.

Machine Dependent program routines

These routines are in SIR either because they involve data conversions or machine dependent operations.

Examples are converting an integer into a numeric string or calculating hashing codes.

Interrupt handling

All the LOWL programs can handle interrupts, though these are usually ignored by the LOWL program as peripheral transfers on the 903 do not use the interrupt system. However all the programs should be able to run under the RADOS operating system without clashing with the system requirements.

Note that the random access disc operating system (RADOS) does not run on the Q.E.C. 903.

Dumping routine

For those programs that can be restarted once they have been initialised, the dump routine will dump the entire program onto paper tape in its state at the time of the dump.

A dumped program should be entered at its break-in entry point, otherwise it will reset itself and lose the information for which it was dumped.

4.1 ALGEBRA

The ALGEBRA system allows the user to define a set of objects and then to define a set of operators that can be applied to those objects. Once the objects and operators have been defined the user can investigate their properties in a conversational manner by evaluating expressions involving variables, constants and operators.

The use of ALGEBRA is fully described elsewhere (Brown, 1974). The program is fully described in the project supplementary material.

4.2 Using ALGEBRA

The 903 ALGEBRA system is distributed as a single SCB tape which may be loaded under initial instructions. The system is entered at 8 and may be broken into at any time by entering at 9, ALGEBRA then types,

```
** BREAK IN
```

on a newline and the user can continue where he left off.

It may be dumped by entry at 10, and the dumped program should be entered at 9 if the objects and operators are to be preserved.

ML/I

ML/I is fully described elsewhere[2]; what follows is an appendix for the ML/I User's Manual for the ML/I version AIG for the 903. The section numbers conform to the references to machine dependent appendices in the user's manual.

5.1 Restrictions and additions

This version of ML/I supports all the features described in the ML/I User's manual (4th edition 1970) plus additional features one to five as described in that manual.

The version number is AIG.

5.2 Operating instructions and input/output

ML/I is distributed as a single SCB tape which may be loaded under initial instructions in the normal way. Entry is at 8; there is no entry point for continuation. The program takes source text from the reader and sends generated results to the punch. Monitor reports are printed on the on-line teleprinter.

The punch output should not normally be redirected to the teletype as this will disturb the monitor formatting.

5.3 Character set

The full ISO set is recognised, in particular lower case letters may be used as in section 2.2 of the manual. Runout, rubout and carriage return are all ignored

unless defined as the source stream terminator, see appendix section 7.

Input is checked for even parity, illegal characters are replaced by the error character which is a backarrow.

5.4 Error messages

The number 2 of section 6.2(f) is 60.

The message of section 6.3.12 is given at the end of every monitor message.

5.5 Integer calculations

All integers should be kept within the range - 131072 to +131071. Overflow is not detected except in the case of division by zero. The numeric effect of overflow is not defined.

5.6 Layout keywords

The 903 inplementation supports the following keywords:

SPACE	meaning a space
NL	newline
SL	start of line
TAB	tab
SPACES	one or more spaces

5.7 Macro variables

There are ten system variables. S1 to S9 have their meanings as assigned in the user's manual and S10 is the value of the input stream terminator, set to 20 (for halt) by default. S10 is compared with the value of characters input before verifying correct parity, and may be 0, 255 or 141 if required for runout, rubout or carriage return.

APPENDIX

A. The 903 and its assembly language

Addendum

References

A.1 Description of the 903

The 903 is a binary machine with an 18-bit word length and an internal memory of 8192 words. There is a simple priority interrupt system but no use can be made of it on a basic 903 without fast peripherals.

An instruction is stored and interpreted in the following manner

B	F	N
bit 18	bits 17-14	bits 13-1

Bits 13-1 (Address field)

Address bits which specify any of 8K locations or a more complete specification of the instruction action if there are no store accesses involved, as in an output instruction, where the address field will specify a particular peripheral.

Bits 17-14 (Instruction field)

Function bits which specify the operation to be performed.

Bit 18 (Modifier bit)

If it is a zero, the instruction is obeyed as it is stored. If the modifier bit is a one the address field is modified by adding it to the modifier register. Any overflow does not affect the function (or modifier 1) bits, of course.

A program has access to four registers, two of which reside in core, and two in the core store itself.

The A-register is used for arithmetic and as data for peripheral transfers. It is 18 bits in length, augmented by an extension register, the Q-register, which is mainly to facilitate operations on a fixed point representation of the 18 bit words.
The B-register is the modifier and is identified with location 1 in core. It is 18 bits in length.
The S-register, or sequence control register determines the location in core of the next instruction to be obeyed. For a basic 900, then, only the low order 13 bits are of any significance. The S-register is indentified with location 0 in core, though attempts to manipulate it, while strictly unnecessary, cause unpredictable results.
The interrupt system provides for pairs of B and S registers in locations 0 to 7 inclusive while the A and Q registers are shared.
Core locations 8180 to 8191 inclusive are used for a hardware loader and this area is only available for reading or executing.

The instruction set of sixteen operations is as follows:
bits 17-14

0	load the B and Q-registers
1	add to the A-register
2	negate A-register and add
3	store Q-register

4	load A-register
5	store A-register
6	collate with A-register
7	jump to address if A-register is zero
8	unconditional jump
9	jump if A-register is negative
10	increment; count in store
11	store S-register
12	multiply A-register result into (AQ)-register
13	divide(AQ)-register result into A-register
14	shift the (AQ)-register, preserve sign bit
15	input/output

Notes

1. Modifications and some instructions affect the Q-register in an undefined way.
2. The instructions 7, 8, 9, 14 and 15 take as their operand the address field directly.
3. The Q-register is sometimes treated as a 17 bit register. Such occasions have not been indicated above.

A.2 The assembly language, SIR

The assembler, SIR, stores constants which may be described in a variety of formats, in consecutive locations of memory.

SIR can fill out references to undefined names when their definitions occur and at the end of the program set a table of all constants referred to by the program.

A name begins with a letter and is followed by letters or digits alone. Only the first six characters of a name serve to distinguish it from any other.

A name can represent a 13 bit integer constant, and it may be set explicitly as in L=6 or by using it as a label.

Values to be loaded as part of the program may be represented in decimal, octal, character, fixed-point or instruction form.

1) decimal representation

sign digits

The sign must be '+' or '-' and there must be at least one digit. The absolute value of the number must be less than 131072. Note that this specifically excludes $-2^{17} = -131072$ although this has a valid internal form.

2) octal representation

& digits

The digits must individually have a value between 0 and 7 inclusive. There must be at least one digit, and no more than six.

3) character representations

£ characters

The characters are converted into an internal 6 bit code (called SIR code) allowing an optimal packing density of three characters per word. This is termed an alphanumeric group.

4) fixed-point representation

sign . digits

Where the details are as in the decimal representation. Note that this format excludes a representation for -1, which has a valid internal form.

5) instruction representation

If the instruction is to be modified, it is preceded by an oblique.

The instruction code is represented by an unsigned decimal integer between 0 and 15 inclusive.

The instruction code is followed by the address field which may have any one of the following forms.

1) unsigned integer

2) name

3) name signed integer

4) semicolon signed integer

5) a constant

e.g. signed integer

sign . integer

& octal-integer

£ at most 3 characters

= an instruction with unsigned integer address

Examples

```
    4  256
   /7  SW
    6  &77
```

The various forms of the field have their obvious meaning : the sémicolon represents the current location in core. The constants are located entirely by the assembler, and are replaced by references to the value in the address field. Thus 4+0 is a concise way of loading zero somewhere and loading an instruction which uses it.

A.3 Program structure

The SIR program is sectioned into blocks to facilitate scoping of names and permit linkage loading.
A name that the assembler retains beyond the period of reading a program is a global name. A block begins with a specification of all global names that occur within it. Names not mentioned in this global heading are not accessible outside the block and consequently must be defined within it.
It is convenient to have a scoping which permits communication between separate blocks of the same program, but not between blocks of different programs. A name that has these scoping rules is called sub-global and is written in the global header of a block, but marked with a special character.

Obviously a subglobal must be defined by the program that uses it.

The end of a block is marked by the header for the next block.

The end of the program is marked by a percent symbol, and is an instruction to SIR to eliminate subglobal names from its dictionary and to locate all constants to which instructions within the program have referred.

A.4 Information given by SIR

The SIR assembler lists definitions of names as they occur with an indication of their scope.

Errors are indicated by a message of the form

En n n

printout of the current line

The significance of the various numbers (n) may be determined from a suitable manual. They mean: the error number, the current line number and the current assembly address, respectively.

On reading the end of the program marker, SIR gives a list of undefined globals and subglobals, the first address and the last of the program just assembled.

Addendum (12 May 1976)

Throughout this text descriptions that, since writing, have required modification are indicated by a vertical bar adjacent to them in the left hand margin. This addendum explains the modifications.

A halt code character is now treated as a warning character, and may be changed by entering the GPM at 34.

Monitor messages use string quotes (not round brackets) to enclose text.

In section 2.13.3, and subsequently, the zero word immediately lower down the stack than the trace flag is now eliminated. This is possible because the trace flag is either 0 or -1; both values cannot occur as external characters and are therefore quite simply illegal. The illegality is detected just before an attempt to output or during a loadarg copy process. The monitor message INVALID CALL may be caused by an improper call to macro COND.

ML/I, SCAN and UNRAVEL are at present being implemented using, and for, the 16K Elliott 905 at the Royal College of Art. Here the process has been to use the LOWL processor described earlier to translate LOWL into part code and part MASIR macros, thus enabling macro

generation for 8K or larger computers.

ML/I will not be re-issued for the 903, because there was a parity error in output of characters.

References

1. BROWN, P.J. (1974)

Macro Processors

John Wiley & Sons

2. BROWN, P.J. (1974)

ML/I User's Manual

University of Kent

3. BROWN, P.J. (1972)

SCAN: A simple conversational programming language for textual analysis

Computers and the Humanities (6.4)

4. STRACHEY, C. (1965)

A General Purpose Macrogenerator

Computer Journal (8.3)

5. FAIRTHORNE, S. & MEEK, B.L. (1969)

903 SIR programmer's guide

General Electric Company

6. BROWN, P.J. & THIMBLEBY, H.W. (1976)

900 SCAN reference manual

Royal College of Art

7. GIMPEL, J.F. (1976)

Algorithms in SNOBOL4

John Wiley & Sons

8. BROWN, P.J. (1972)

UNRAVEL - a programming language to put intelligence into dumps.

Computer Journal (16.1)