Dynamic debugging in BASIC

G. M. Bull

Department of Computer Science, The Hatfield Polytechnic, Hatfield, Hertfordshire

One of the main advantages that on-line working provides is the ability to interact with a running program. Interaction at run-time is important from two standpoints. Firstly, it enables the programmer to control the action of the program by providing suitable data as the execution progresses. Secondly, and most importantly, given the right facilities, it enables a programmer dynamically to debug the program. One of the most popular languages designed explicitly for on-line working is BASIC. Although one is able to interact in the first sense, most implementations fail to give any run-time debugging facilities to the BASIC user. This paper describes the range of run-time debugging facilities provided on an implementation of BASIC at Hatfield Polytechnic on an ICL 803.

(Received January 1970, Revised July 1971)

The overall system

The essential features of BASIC are that each statement occupies a single line and is prefixed by a line number which serves both to order the lines and to act as a labelling system. The system described has an incremental compiler and a routine for sorting the lines into line-number order. The sort routine enables one to insert a line into the program, to replace a line by retyping with the same line number and to remove a line by typing the line number followed immediately by a new-line character.

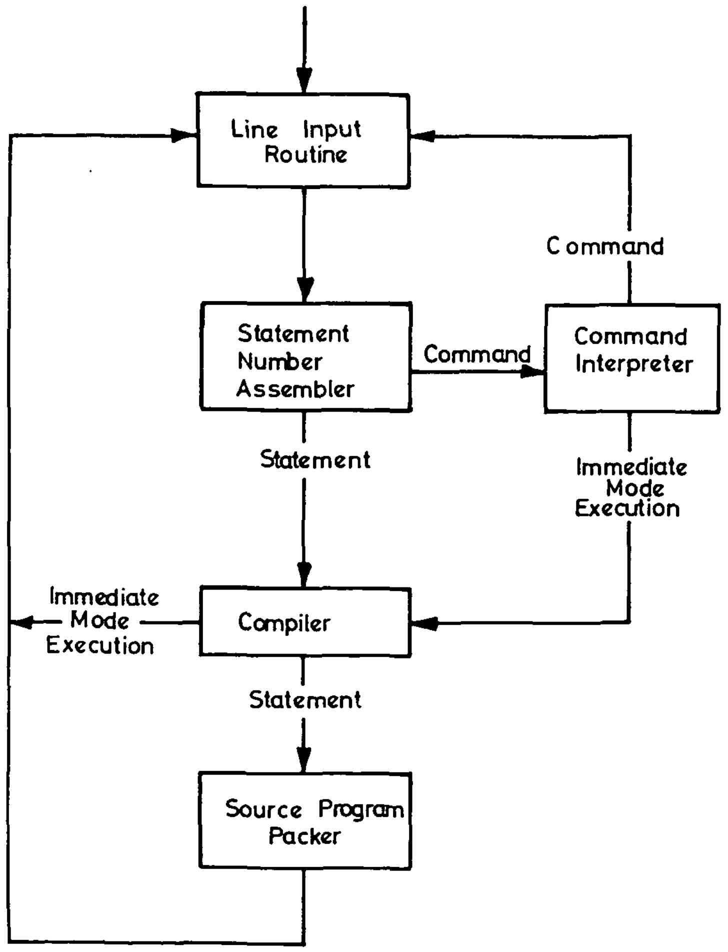


Fig. 1

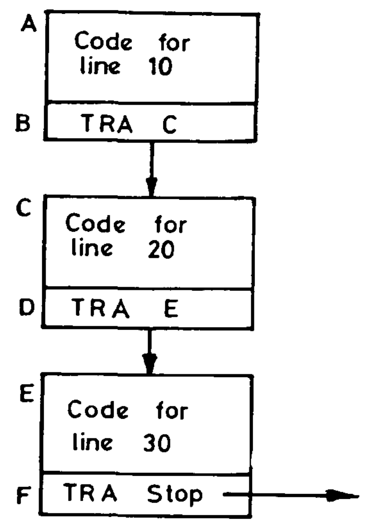
An overall picture of the software modules is given in **Fig. 1.** The line input routine assembles characters from the teletype, converts to an internal code and exits on meeting a new line character. If the assembled line does not start with a line number a command is assumed and a table of command names is scanned for a match. If it is a command (RUN, LIST etc.), the command is obeyed. If it is an immediate-mode statement (see later) the execute flag is set and the compiler entered. If the line starts with a line number the compiler is entered, and if no error is found, object code is generated. If the execute flag is set the compiled code is executed and control returns to the line input routine, otherwise the object code is 'patched' into the object program as it exists at that time. The source code is then packed, sorted into line number order and stored. The debugging facilities provided allow the programmer dynamically to:

1. Interrupt a running program.
2. Print the values of selected variables.
3. Change the program by submitting new lines of source.
4. Change the values of selected variables.
5. Set or unset break-points on one or more lines.
6. Set or unset trace on one or more lines.
7. Cause a subroutine to be executed.
8. Execute a single line of program.
9. Continue from the point of interruption or a breakpoint, or transfer control to any line in the program.

Apart from interrupting a running program, all other facilities may be used at any point in time either before, after, or during a run. The interrupt facility (the ESCAPE key on the teletype) causes a bit to be set which is examined at the beginning of the next statement to be obeyed.

The compiler

Most implementations of BASIC on executing the RUN command, sort and then compile the complete source program as it then exists and then, if possible, execute it. The approach adopted here is to compile each line as it is met and by means of a compiled program directory (CPD) and links, logically to insert the object code for this line into the correct position according to its line number. The CPD has a single word entry for each line of program and records the line number, the starting address of the compiled code corresponding to this line, and the address of the link for this line. When the next line of source has been compiled, an entry is made in the CPD so that the entries are maintained in line number order. The link



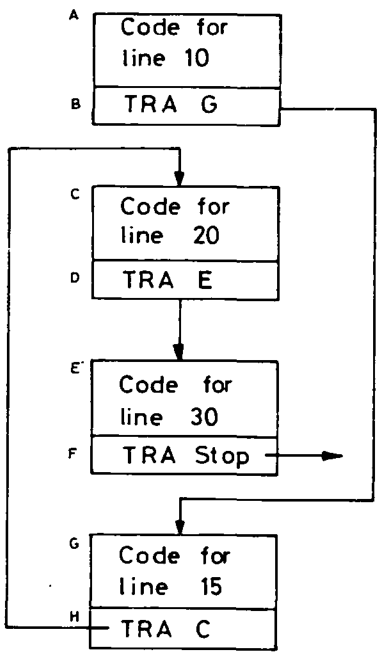


Fig. 2 above
Fig. 3 right

word of each code block is a jump to the code block of the next highest line. For example, suppose the current state is

```
10 READ X, Y
20 IF X < Y THEN 50
30 LET W = X*Y/2
```

the CPD would have three entries, say

A, B, 10
C, D, 20
E, F, 30

Where A is the address of the start of the code block for line 10 and B is the address of the link (which is always the last word of the code block) for this line, C, D, E and F are addresses in the same way for lines 20 and 30.

The code blocks would be as in **Fig. 2.** TRA is an unconditional transfer instruction.

If the next line entered is

```
15 LET P = 0
```

the CDP becomes:

A, B, 10
G, H, 15
C, D, 20
E, F, 30

and the code blocks would be as in **Fig. 3.**

The compilation of the code blocks is quite straightforward and will not be discussed; however, each code block carries two extra words over and above the compiled code and the link. The first word of each block is a transfer to a subroutine which performs four tasks:

1. Plants in a standard location the line number found in the second word. If the program is interrupted whilst running, the line currently being obeyed may be identified.
2. If an interrupt has occurred, the message BREAK AT LINE n is output (where n is the current line number) and control returns to the line input routine.
3. Tests the break bit in word two of the compiled code block to see if a break has been set; if so the message BREAK AT LINE n is output and control returns to the line input routine.
4. Tests the trace bit in word two of the compiled code block to see if a trace has been set on the line; if so *n is output (where n is the current line number). Control returns to the object code. The flow chart for the subroutine is shown in **Fig. 4.** The second word contains the statement number and two bits used to record break and trace.

New commands

Most implementations of BASIC embed the language in a standard command set; six new commands have been added to give the user the required facilities:

1. BREAK, UNBREAK, and CONTINUE
   Break points may be set by: BREAK list of line numbers, where a list entry may either be a single line number or of the form n-m, meaning all lines in the range n to m inclusive. The effect is to set a bit in the compiled code block of each of the lines in the list. If a line does not exist, the message NO SUCH LINE m is displayed and the next entry in the list is taken. At run-time the effect is to halt the program just before obeying a given line (say line n) and to output the message BREAK AT LINE n. At this point appropriate action may be taken. To continue from line p, following a break, one types the command CONTINUE p. If no line number is given the next line in sequence is assumed. All breaks remain set until explicitly unset by either UNBREAK list of line numbers, or UNBREAK, which unsets all breaks previously set.
2. TRACE and UNTRACE
   The trace command is of the form: TRACE list of line numbers, and causes the trace bit to be set on all the listed lines. At run-time the effect is to output the message *n, without halting just before obeying line n. The trace remains set until explicitly unset by UNTRACE list of line numbers, or UNTRACE, which unsets all traces.
3. EXECUTE
   A single line of program may be executed, and control returned to the line input routine following the execution,

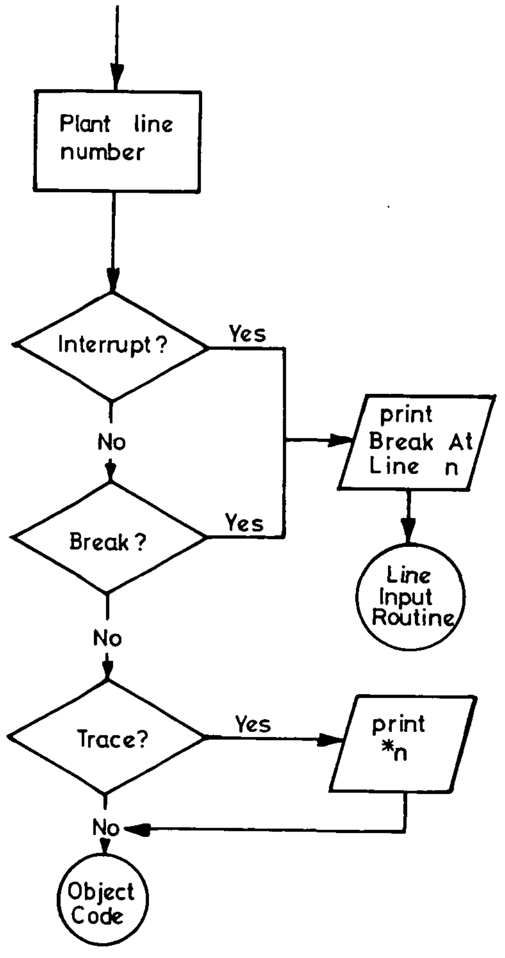


Fig. 4

by means of the command EXECUTE or EXECUTE n. Note if the statement executed is a GOTO or a GOSUB statement or a conditional statement which is true, control will not be lost. In all cases the message BREAK AT LINE n is displayed, where n is the line to which control has been transferred. If the executed statement is non-branching, n will be the next highest line number.

Immediate mode-statements

It is possible to write statements which are not to be included in the program, but are executed immediately and then discarded. This gives the user a 'desk calculator' mode which will not be discussed here, but more importantly it provides a powerful debugging tool. This facility is made available with the three statements PRINT, LET, and GOSUB.

Thus typing:

PRINT C, D

causes the current values of C and D to be printed.

LET B = SQR(X)

causes B to be assigned the value of the square root of X

GOSUB n

causes control to be transferred to line n and control lost until a RETURN statement is obeyed, at which point control returns to the line input routine. In this way one is able to test a complete subroutine in isolation.

Example

Consider the following example of a program to tabulate sin(x) against x for x = 0(0·5) 6·5. The program contains a number of logical errors. The following conversation demonstrates ways of discovering the errors and correcting them. All computer typeouts are underscored, or enclosed in brackets for multiline sections.

```
NEW EXAMPLE
READY
10 READ E
20 LET T = 1
30 LET N = 1
40 FOR X = 0 TO 6.5 STEP 0.5
50 LET S = X
60 LET T = -T*X*X/(N*(N + 1))
70 LET N = N + 2
80 IF ABS(T) > E THEN 110
JUMP WARNING
90  PRINT X, S                  This warning is given since
100 GOTO 40                     line 110 has not yet been
110 LET S = S + T               met. It tells the programmer
120 GOTO 60                     that an entry has been made
130 NEXT X                      in the jumptable but a run-
140 DATA 0.00005                time failure will occur if
150 END                         line 110 is not entered and if
RUN                             an attempt is made to trans-
                                fer control to line 110.
[EXAMPLE
[0        0
[0        0
[0        0                     Interrupted during printing.
[BREAK AT LINE 100              The interrupt is detected in
                                the line following the print
                                statement.
BREAK 70                        Set up a break point on line
                                70.
READY
CONTINUE
BREAK AT LINE 70                Program breaks before
                                executing line 70
```

```
PRINT S; T                      Have a look at the values of
                                S and T
0        0                      Not correct!
TRACE 40-130                    Set up a trace to follow the
READY                           flow since we do not appear
                                to be incrementing X.
CONTINUE
[*70
[*80
[0        0
[*100
[*40
[*50
[*60
[BREAK AT LINE 70               The break point is met
                                again.
100 GOTO 103                    The trace points to line 100
                                being incorrect. The first
                                try for line 100 has a
                                typing error, (103 for 130).
JUMP WARNING                    The jump warning points
                                this out.
100 GOSO  130                   Another error made. The
[       ▲                       arrow points to the position
[  ILLEGAL INSTRUCTION          in the line where the error
100 GOTO 130                    was discovered and gives an
UNTRACE                         error message.
READY
UNBREAK
READY
CONTINUE 20
[0        0
[.5       .5
[1        1                     Interrupted during printing.
[BREAK AT LINE 100              Still not correct so try
TRACE 80-130                    tracing again.
READY
CONTINUE
[*100
[*130
[* 80
[* 90
[1.5      1.5
[*100                           Interrupted
[BREAK AT LINE 130              Since the program does not
20                              appear to goto line 110 we
30                              deduce that T is always too
42 LET T = 1                    small. We notice T and N
44 LET N = 1                    are initialised outside the
UNTRACE                         loop instead of inside.
READY
CONTINUE 40
[0        0
[.5       .377604
[1        .540278
[1.5      .570753               Interrupted—still not cor-
[BREAK AT LINE 100              rect.
BREAK 80                        Set up break point
READY
LET X = 0.5                     Force X to 0.5
CONTINUE 42                     Start at line 42
BREAK AT LINE 80                Program hits break point
PRINT T                         Print second term of sine
-.125                           series for X = 0.5
EXECUTE 60                      Execute line 60 to compute
BREAK AT LINE 70                third term of the series.
```

Program breaks at the following line after an execute.

```
PRINT T
2.60417E-03
```

Print value of third term. Both incorrect when checked by hand.

```
42 LET T = X
44 LET N = 2
UNBREAK
```

This leads us to question initial values of T and N. New lines 42 and 44 compiled and replace old versions.

```
READY
LIST
```

Call for a listing of the program as it now stands.

```
EXAMPLE
10 READ E
40 FOR X = 0 TO 6.5 STEP 0.5
42 LET T = X
44 LET N = 2
50 LET S = X
60 LET T = -T*X*X/(N*(N + 1))
70 LET N = N + 2
80 IF ABS(T) > E THEN 110
90 PRINT X, S
100 GOTO 130
110 LET S = S + T
120 GOTO 60
130 NEXT X
140 DATA 0.00005
150 END
READY

CONTINUE 40
```

```
0       0
.5       .479427
1        .841468
1.5      .997497
2        .909296
2.5      .598449
3        .141131
3.5     -.350788
4       -.756849
4.5     -.97751
5       -.958933
5.5     -.705536
6       -.279387
6.5      .215107
END OF PROGRAM
```

Conclusions

In the above example, some, but by no means all of the potential of a dynamic debugging system is exhibited. Most arguments in favour of time-sharing place emphasis on quick turn-around for testing, access to a filing system, and interaction with the running program by supplying data dynamically to control its action. However, in the author's opinion, the greatest advantages will be realised when systems provide debugging aids similar to those described here for all languages available at the terminal.

Acknowledgements

The author is indebted to his colleague W. Freeman for his suggestions and comments on the draft of this paper.

References

BASIC Reference Manual. Dartmouth College, Hanover, NH, USA.
Bull, G. M., and Freeman, W. (1971). BASIC—A preliminary specification. Hatfield Polytechnic Comp. Sc. Tech. Memo No. 1.
Bull, G. M. (1971). BASIC—its growth and development, *J. Instn. Comp. Sc.*, Vol. 2, No. 3, p. 51.
Barron, D. W. (1969). A note on program debugging in an on-line environment, *The Computer Journal*, Vol. 12, No. 1, p. 104.
Barron, D. W. (1971). Approaches to conversational FORTRAN, *The Computer Journal*, Vol. 14, No. 2, p. 123.
Barron, D. W. (1971). Programming in Wonderland, *The Computer Bulletin*, Vol. 15, No. 4, p. 153.

Correspondence

To the Editor
The Computer Journal

Sir,

With reference to your article 'Step size adjustment at discontinuities for fourth order Runge-Kutta Methods', by P. G. O'Regan, published in this *Journal*, Volume 13, Number 4, November 1970, I should like to point out that equation (21) appears to be incorrect in the fifth term of the right-hand side which should, I think, be $A^5(14B^4 - 21B^2C + 3C^2)$ instead of $A^5(14B^4 - 21B^2C - 3C^2)$. The last column of Table 1 of the article was evidently computed using the erroneous equation.

Table 1 has been recomputed to a precision of 20 significant digits using the correct equation and the results are reproduced below. The answers for Newton's iteration formula are quoted to 9 decimal places, for which two iterations were necessary. The revised table shows even more clearly than the original that Newton's iteration formula (23) is more accurate than (21).

Yours faithfully,
E. Whiteley (Miss)

Table 1 $h = 0·1$

α_t	$10^6(\alpha_t - \alpha_n)$	$10^6(\alpha_t - \alpha_{iii})$	$10^6(\alpha_t - \alpha_{iv})$	$10^6(\alpha_t - \alpha_v)$
0	0	0	0	0
0·1	−0·202	− 0·301	−0·201	− 0·202
0·2	−0·655	− 2·249	−0·609	− 0·657
0·3	−1·170	− 9·336	−0·808	− 1·187
0·4	−1·606	− 27·730	−0·055	− 1·703
0·5	−1·871	− 66·426	2·948	− 2·252
0·6	−1·915	− 137·403	10·292	− 3·081
0·7	−1·729	− 255·782	25·130	− 4·741
0·8	−1·340	− 439·996	51·967	− 8·214
0·9	−0·809	− 711·955	96·976	−15·080
1·0	−0·228	−1097·231	168·339	−27·727

Structures Department
Royal Aircraft Establishment
Farnborough
Hampshire
13 August 1971