Malcolm Bull
Training and Consultancy Publications

MB-Guide to the Basic Debugger

Malcolm Bull

MB-Guide
to
Basic symbolic debugger

MB-Guide

to

Basic symbolic debugger .

by

Malcolm Bull

Introduction

This *MB-Guide to the Basic debugger* is produced for those who need a quick introduction to the features of the debugger which is available for use with Basic programs on the Pick operating system.

The guide is primarily aimed at the experienced programmer, although there are parts which might be of interest to the less technical end-user.

This *MB-Guide* contains:

1) A general introduction to the Basic debugger

2) A description of the individual commands which are available for use with the Basic debugger

3) A *how-to* section describing how to achieve specific effects with the Basic debugger.

You may find the following titles in the *MB-Guide* series useful in conjunction with the present volume:

Program design
Basic programming
The system debugger

This *MB-Guide* is not intended to present a complete description of the subject but merely to place it in context and give the reader enough information to use the facilities and to survive.

Best use can be made of this *MB-Guide* if it is read in conjunction with the reference literature which is provided for your system. You should amend your copy of this guide so that it accurately reflects the situation and the commands which are used on the implementation which you are using. By doing this, your *MB-Guide* will become a working document that you can use in your daily work.

I hope that you enjoy reading and using this *MB-Guide* and the others in the series.

Malcolm Bull

Malcolm Bull
Training and Consultancy Publications
19 Smith House Lane
BRIGHOUSE
HD6 2JY
West Yorkshire
0484-713577
ISBN: 1 873283 10 5

1 Introduction

The Basic debugger - also known as the **symbolic debugger** - is a standard part of the Pick operating system. Its functions are

* to signal the occurrence of a run-time error during the execution of a Basic program, and then (possibly)
* to allow the programmer to fix the error and/or restart the program.
* to trace the flow of processing through an executing Basic program in order to observe the contents of the variables.

1.1 Invoking the Basic debugger

Under normal circumstances, it is possible to interrupt any process by pressing the <BREAK> key on the terminal keyboard. If you interrupt a Basic program whilst it is executing, then you will get a display something like:

```
*40
*
```

showing that you are in the **Basic symbolic debugger** and you have interrupted the program at line 40.

If you interrupt any other process, such as an Access report or a Basic program compilation, you will get a display something like:

```
I 123.45
!
```

showing that the process has stopped at the instruction at byte 45 (hexadecimal 45 = decimal 69) of frame number 123, and you are in the **interactive debugger** or the **system debugger**. This is discussed further in the *MB-Guide to the system debugger*.

In addition to entry to the Basic debugger on the detection of an error condition within a Basic program, you may also use the Basic

```
DEBUG
```

statement within your program as a debugging aid to force an interrupt when a suspect piece of coding is about to be executed.

Advanced Pick has a TCL DEBUG statement to invoke such an interrupt. This is an alternative to the <BREAK> key as the <BREAK> key serves another function on Open Architecture and Advanced Pick implementations.

1.2 Fundamental responses

Whenever the Basic debugger displays the

*

prompt, you can enter any of the following responses:

<RETURN>
used on its own, the <RETURN> key has no effect, except to repeat the * prompt.

In general, <RETURN> after any of the debug commands shown below will process the command but stay in the debug mode.

<LINE FEED>
to continue the processing, if possible. This is identical to the G command described below.

P<RETURN>
to switch the terminal print output function ON or OFF, and suppress the display to the terminal screen.

END<RETURN>
to terminate the processing and return to TCL, unless your account definition prevents you from doing this.

END<LINE FEED>
to terminate the current activity and return the processing to the calling Proc, if any.

OFF<LINE FEED>
to terminate the current activity and log off the account.

OFF<RETURN>
to terminate the current activity and log off the account.

<CTRL> J
to continue the processing, if possible. This is identical to the <LINE FEED> command described above, and is valuable if you are using a terminal with no <LINE FEED> key.

G
to continue the processing, if possible. This is identical to the <LINE FEED> command described above, and is valuable if you are using a terminal with no <LINE FEED> key.

The above commands are available with both the Interactive debugger (the ! prompt) and the Basic debugger (the * prompt).

The most powerful use of the Basic debugger is in locating (and possibly fixing) errors in a Basic program.

2 Using the debugger

The Basic debugger will let you:

* abandon the execution of the program.
* go through the program instruction by instruction.
* redirect the processing to a specific statement or to the beginning of the program.
* inspect the contents of any of the variables in the program.
* change the contents of any of the variables in the program.
* establish a set of **break conditions**. When any one of these conditions occurs, the program execution will interrupt and the Basic debugger will be invoked.
* specify that the program execution is to proceeds in steps of **n** instructions. The program execution will then interrupt after every *n*th instruction and the Basic debugger will be invoked.
* specify a set of up to six variables which are to be displayed whenever an interruption occurs.
* display the source statements of any program (or indeed any item on any file).

2.1 Symbol table

Before we look at the Basic debugger, there are two entities which you must be aware of - the **symbol table** and the **trace table**.

The symbol table is a map used by the Basic run-time processor and identifies the location of each variable by name. A typical symbol table might look like this:

```
C000A00000000REFNO
C001400000000RECORD
C001E000A0001NAME
C001E00000000DOB
C002800000000FILE
L008200000000X
L008C00000000REC
L009600000000RECNO
L00A000000000
```

The names of the variables can be recognised in this table.

The symbol table is produced when the program is compiled. The following points are important:

* On most Pick implementations, the symbol table is generated automatically at compilation time and is held

immediately behind the object code. The production of the symbol table can be suppressed by the S option on the Basic verb, as described below.

* On McDonnell Douglas implementations, the symbol table is generated by the M option on the BASIC verb and is held as a separate item (called *programname) on the same file as the program.

The symbol table must be available in order to use the full features of the Basic debugger, and especially those which allow you to identify each variable by name.

2.2 Trace table

The **trace table** is a list which is held and maintained by the Basic debugger, and contains the names of

1) up to six variables which are to be displayed each time the program execution is interrupted.

2) a set of up to four break conditions, the satisfaction of any one of which will invoke an interruption in the processing.

The trace table is empty when a program begins to execute. The contents are retained throughout the execution of a program (and its subroutines) and they are lost when the execution ends. The appearance of the trace table (as displayed by the D command) will then look like that in diagram (a).

```
T1
T2
T3
T4
T5
T6
B1
B2
B3
B4
```

(a) empty

```
T1 SALES.FIGS
T2 NAME
T3
T4
T5
T6
B1 $>30
B2 CODE='YES'&NAME=""
B3
B4
```

(b) in use

By means of T and B commands which you give conversationally to the Basic debugger, you will specify the variables and the break conditions - if any - which are to go into the trace table. The appearance of the trace table (as displayed by the D command) may then look like that in diagram (b).

As you add each variable name and/or break condition to the trace table, the debugger will respond with a

```
+
```

to indicate that the entry has been accepted. If you attempt to add to many entries to the trace table, then the debugger will respond with the

```
TBL FULL
```

message

You may subsequently wish to remove variables break conditions from the trace table (by means of the U and K commands). As you remove each variable name and/or break condition from the trace table, the debugger will respond with a

```
-
```

to indicate that the entry has been removed.

3 BASIC verb

The options which you use when you compile your Basic program have some impact on the use of the Basic debugger with the compiled program.

The purpose of the BASIC verb is to compile your program (or programs), check for and report any syntax errors and - if there are no errors - produce an equivalent object program.

```
BASIC filename itemlist {(options)}
```

Examples:

```
BASIC SALES.PROGS ADD.RECORDS

BASIC STOCK.BP ADD.STOCK AMEND.STOCK DELETE-STOCK

BASIC RU.PROGS *

SSELECT PROGS = "[AMEND]"
BASIC PROGS
```

Some implementations also offer the forms:

```
COMPILE filename itemlist {(options)}
```

The following options on the BASIC (and the COMPILE) verb are important if you intend to use the Basic debugger:

C to produce an object program without end-of-line characters. This produces a smaller object program but any Basic debugger operations which refer to the line-numbers will work successfully since the run-time processor always thinks it is on line number 1.

You should not use this option if you wish to use the full facilities of the Basic debugger with the program.

S to suppress the production of the debug symbol table.

You should not use this option if you wish to use the full facilities of the Basic debugger with the program.

If you are using McDonnell Douglas implementations, the corresponding options are:

E to produce an object program without end-of-line characters. This produces a smaller object program but any Basic debugger operations which refer to the line-numbers will work successfully since the run-time processor always thinks it is on line number 1.

You should not use this option if you wish to use the full facilities of the Basic debugger with the program.

M to produce the symbol table (for use with the interactive debugger) and output a program map showing the names of the variables used, the statement labels used, and the distribution of the program statements across disk frames.

You **should** use this option if you wish to use the full facilities of the Basic debugger with the program.

Full details of the BASIC verb are given in the *MB-Guide to Basic programming.*

4 RUN verb

The options which you use on the RUN command when you execute your program have some impact on the use of the Basic debugger with the program.

The format of the RUN command is:

```
RUN filename programname {(options)}
```

or, if the program has been catalogued:

```
programname {(options)}
```

Examples:

```
RUN SALES.PROGS ADD.RECORDS (D)

ADD.RECORDS (D,E)
```

The following options on the RUN verb, and with catalogued programs, are important if you wish to use the Basic debugger:

D to invoke the Basic debugger at the first line of the program.

E to invoke the Basic debugger when any error condition - whether it is a **fatal error** or a **non-fatal error** - arises. We discuss these errors below.

4.1 Errors

Many types of errors - the **syntax errors** - will be highlighted by the Basic compiler. These include

* certain spelling and typing errors.

* grammatical errors in the syntax of the statements.

* missing statement labels.

All syntax errors must be removed before the program will compile successfully.

If the program has compiled successfully, there may still be **logical errors** detected during the execution of the program. Some of these errors interrupt the program execution and call up the Basic debugger, others do not.

The **fatal errors** which display an error message and then interrupt the program to invoke the Basic debugger include:

* an attempt to access an element of an array which is outside the size limits for that array.

  ```
  ARRAY SUBSCRIPT OUT-OF-RANGE
  ```

  For example, the sequence

  ```
  X=100
  SALES(X) = 0
  ```

  would generate an error if the array SALES had been declared with fewer than 100 elements. Interestingly, the statement

  ```
  SALES(100) = 0
  ```

  would be rejected by the compiler.

* an attempt to use a file-variable to which no file has yet been opened.

  ```
  FILE HAS NOT BEEN OPENED
  ```

  For example, the statement

  ```
  READ RECORD FROM STOCK.FV,ITEMID ELSE STOP
  ```

  would generate an error if there had been no prior statement such as

  ```
  OPEN 'STOCK' TO STOCK.FV ELSE STOP
  ```

  opening a file to the file-variable STOCK.FV

* an attempt to execute a RETURN statement without a prior GOSUB or CALL statement.

```
RETURN EXECUTED WITH NO GOSUB
```

* an attempt to use the contents of a file-variable in an expression.

```
FILE VARIABLE USED WHERE STRING EXPRESSION EXPECTED
```

For example, the statements

```
PRINT STOCK.FV
TOTAL = TOTAL + STOCK.FV
```

would generate an error if the variable STOCK.FV had been used as a file-variable in a prior statement such as

```
OPEN 'STOCK' TO STOCK.FV ELSE STOP
```

The **non-fatal errors** which simply display an error message and then continue with the execution include:

* reference to an unassigned variable in an expression.

```
VARIABLE HAS NOT BEEN ASSIGNED A VALUE
```

For example, either of the statements:

```
PRINT MESSAGE
TOTAL = TOTAL +1
```

would generate an error if there had been no prior statements such as

```
CLEAR  to clear all the program variables
MESSAGE = "Please enter the code number"
TOTAL = 0
```

If there are several unassigned variables in an expression, such as

```
TOTAL = A + B + C
```

where all of A and B and C are unassigned, then there will be one error message for each of the three variables.

* use of an alphanumeric string variable in an arithmetic statement.

```
NON-NUMERIC DATA WHEN NUMERIC REQUIRED
```

For example, the statement

```
TOTAL = TOTAL + COUNTER
```

would generate an error if either TOTAL or COUNTER contained an alphanumeric string value. An alphanumeric string is any value which includes non-numeric characters other than a decimal point and/or a leading plus or minus sign; spaces are also regarded as non-numeric characters in this context.

* use of the COL1() or COL2() functions without a FIELD function having been executed earlier.

```
COL1 OR COL2 USED PRIOR TO EXECUTING A FIELD STMT
```

* an attempt to divide by zero. A result of 0 will be returned.

```
DIVIDE BY ZERO ILLEGAL
```

In all cases, the Basic debugger will indicate the line number at which the error occurred.

5 Debugger commands - summary

We have already considered the fundamental command which may be issued at any interruption when the Basic debugger or the system debugger is active. The following commands are available for use whenever the Basic debugger displays the asterisk.

A more detailed discussion of their action is given in the following sections. Full details are given in the reference literature which is available for your implementation.

?
$

command for displaying the name of the program, the current line number and the status of the object code.

/

command for inspecting and changing the contents of a program variable.

[

command for specifying how much data is to be displayed for each variable.

B

command for setting a break point when a certain

condition is reached.

D
command for displaying the trace table.

DE
DEBUG
command for passing control to the system debugger.

E
command for setting the number of statements to be executed between break points.

END
command for terminating the execution and returning to TCL.

G
command for passing control to a specific line number.

K
command for removing a break point from the trace table.

L
command for displaying the program source code.

LP
command for switching program output to the printer.

N
command for skipping break points.

OFF
command for terminating the program execution and logging off the system.

P
command for cancelling display of program output.

PC
command for closing and printing the current spooler file.

R
command for cancelling a subroutine return address.

S
command for displaying the stack of subroutine return addresses.

T
command for adding a variable to the trace table.

U
command for removing a variable from the trace table.

Z
command for specifying the item which is to be displayed by the L command.

In general, if the command is followed by

<LINE FEED> then the command will be processed and the execution will continue to the next break point.

<RETURN> then the command will be processed but the execution will not continue allowing a further command to be issued at this break point.

5.1 Debug commands

Let us look at these commands a little more closely.

Full details are given in the reference literature which is available for your implementation.

5.1.1 ? or $ command

Either of the commands:

```
?
```

or

```
$
```

will display a message such as

```
WAGES.CALC.001 L 122 OBJECT VERIFIES
```

showing

* the name of the program currently being executed (this is WAGES.CALC.001 in this instance),
* the line-number at which the interrupt occurred (122) and
* whether or not the object code verifies correctly.

It is particularly helpful to use this command whenever the line-number, as displayed on the En message, takes a big jump or when the line number jumps back to 1. This suggests that you have entered an external subroutine, or that you have returned to a previous calling routine.

This command does not use the symbol table.

5.1.2 / command

The

```
/xxx
```

command will display the contents of a specific variable, and allow you to change them.

Thus, if you enter:

```
/EMPNAME
```

the Basic debugger might respond with:

```
SMITH=
```

showing that the string SMITH is currently held in the variable EMPNAME. If you wish to change the value held in EMPNAME, then you will enter the new value followed by <RETURN>. This new value will overwrite the previous contents of EMPNAME.

If you press <RETURN> without entering a new value, then the contents of EMPNAME will be left unchanged.

If you attempt to inspect a variable which is not in the symbol table (or if the program has been compiled without producing a symbol table), then the debugger will display

```
SYM NOT FND
```

If you attempt to display the contents of a variable which has not yet been assigned a value, then the debugger will display the message

```
0=
```

followed by

```
UNASGN VAR
```

It is possible to change the contents of a previously unassigned variable.

If you attempt to display the contents of a file variable, then the debugger will display

```
[B34] LINE n FILE VARIABLE USED WHERE STRING EXPRESSION EXPECTED
```

It is possible to display an individual element or all the elements of a dimensioned array. Let us imagine that the program contains an array by the name of SALES. If a single element of the array is to be inspected, then this should be entered as:

```
/SALES(36)
```

the contents of this element may then be changed, as required.

To display the contents of all the elements of the array, you will enter the command

```
/SALES
```

then the Basic debugger will display all the elements in the array in turn:

```
SALES(1) XXX=
```

If you press <RETURN> after displaying - without changing the contents - then the next element will be presented:

```
SALES(2) YYYYYY=
```

If you change any of the elements, then no further elements will be displayed.

If you enter the form:

```
/*
```

then the current contents of all variables will be displayed (a screen page full at a time) without any opportunity to change the contents.

5.1.3 [zone command

Normally, the entire contents of each variable will be displayed by means of either the / or the T command. The

```
[start,length]
```

command is used to restrict the amount of data which is displayed when variables are inspected. Both brackets and the start and the length parameters must be supplied.

For example:

`[1,30]`
will display only the first 30 characters of each variable.

`[20,55]`
will display the 55 characters starting at the 20th character; that is, characters 20 through to 74.

This zone will apply to all variables which are displayed.

The simple

```
[
```

command will cancel any existing zone.

5.1.4 B command

The

```
Bcondition
```

command will add a **break condition** to the trace table.

Typical examples are:

```
BCODE='YES'
BCODE="YES"
```

either of these commands will cause a break to occur when the variable CODE is set to the value YES

BTOTAL>100
will cause a break when the variable TOTAL reaches a value greater than 100.

BCODE=TEST
will case a break when the contents of the variable CODE and the variable TEST are equal.

The break condition may also include the number of the line which is about to be executed:

B$=30
will cause a break immediately *before* the execution of line-number 30 of the program.

B$>30
will cause a break whenever the line number is greater than 30, that is immediately *before* the execution of line-number 31 and beyond. If the processing passes back to a position before line number 30, then the break will not occur.

It is possible to specify that several conditions must all be satisfied:

BCOUNTER=5&CODE=YES
will cause a break when the variable COUNTER has a value of 5 *and* variable CODE has the value YES.

The effect of invoking a break at one condition *or* another is achieved by adding several break points to the trace table.

The operators used in the break condition are:

=	equal
£	not equal
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

Each new break condition will be added to the first available space in the table. The Basic debugger will acknowledge that each condition has been added to the trace table by displaying a

+

sign.

If you attempt to use a variable which is not in the symbol table (or if the program has been compiled without producing a symbol table), then the debugger will display

SYM NOT FND

If you attempt to add to many entries to the trace table, then the debugger will respond with the

```
TBL FULL
```

message

There may be up to four break conditions in the trace table at any one time.

Break conditions may be removed from the trace table by means of the K command.

This command requires the symbol table and end-of-line characters.

5.1.5 D command

The

```
D
```

command will display the current contents of the trace table, showing the variables which are currently being displayed at each break point and the break conditions which are active.

```
T1 SALES.FIGS
T2 NAME
T3
T4
T5
T6
B1 $>30
B2 CODE='YES'&NAME=""
B3
B4
```

You may need to use the D command to display the current trace table in order to known which break condition is to be killed (by the K command) and/or which variable is to be removed (by the U command).

5.1.6 DE or DEBUG command

The

```
DE
```

or

```
DEBUG
```

command will pass control to the *system debugger*. This will give you access to the facilities for frame inspection and patching.

If you subsequently enter just

```
<LINE FEED>
```

at any ! prompt from the system debugger, you will be returned back to the Basic debugger.

This is discussed in the *MB-Guide to the system debugger.*

5.1.7 E command

The

```
En
```

command will cause the Basic debugger to break after executing **n** statements.

For example:

E1
: will invoke a break before every statement is executed.

E10
: will invoke a break before every tenth statement is executed.

E
: will cancel any previous En specification.

Note that this statement recognises executable statement lines.

```
001       PRINT 'THIS IS A TEST'
002       PRINT 'MESSAGE A'
003       PRINT 'MESSAGE B'
004       PRINT 'MESSAGE C'
005       PRINT 'MESSAGE D'
006       PRINT 'MESSAGE E'
007       FOR X=1 TO 10
008           Y=X*X
009           PRINT X; PRINT Y
010       NEXT X
011       END
```

Thus, if we were to use a command such as

```
RUN filename programname (D
```

to execute the program in the diagram and automatically invoke the Basic debugger (because of the D option) and then enter the command

```
E3
```

at the first break point, then breaks would subsequently occur as these lines were about to be executed:

```
4 7 10 9 8 7 10 9 8 7 10 9 8 7 10 11
```

Note especially that the line

```
PRINT X; PRINT Y
```

is considered as a single statement.

To avoid being inundated with debug output, it is suggested that, when you are trying to locate a fault, you use a form such as E10 (rather than E1) to monitor the execution flow in steps of 10 or so statements.

This command does not use the symbol table, although it will only work successfully if the Basic program has been compiled in the normal manner (that is, without the option to suppress the end-of-line characters)

5.1.8 END command

The

```
END
```

command is one of the fundamental commands used with the debugger, as described earlier in this *MB-Guide* and will terminate the program execution and return the user to TCL (or to the logon Proc, if this is a closed system).

5.1.9 G command

The

```
G
```

command will cause the execution to continue to the next break point.

Gn
: will transfer the execution to line-number n. Whenever a line-number is specified in a debug command, it is the program line-number and not the Basic statement number.

G1
: will go to the start of the program and not to the statement labelled 1.

When such a jump is made, the contents of all variables in the program (and the return address stack for internal subroutines) will remain as they were at the break point.

This command does not use the symbol table, although it will only work successfully if the Basic program has been compiled in the normal manner (that is, without the option to suppress the end-of-line characters)

5.1.10 K command

The B command is used the add break conditions to the trace table. The

Kn

command will *kill* break condition number n by removing it from the trace table.

For example:

K3
will remove the break condition which has been placed in the third of the break conditions in the trace table.

You may have to use the D command to display the current trace table in order to known which break condition is to be killed.

The simple form:

K

will kill all the break points in the trace table.

5.1.11 L command

The L command is used to display a section of the current source program. The identity of the *current program* is specified by the Z command (this is not necessarily the same as the program which is currently executing). If no Z command has yet been issued, then the debugger will display the

NO SOURCE

message

The simple form:

L

will display the current line of the source program, that is the line at which the break occurred and which is about to be executed, and the form:

L*

will display the entire program.

The extended form:

Lstart-number

command is used to display a section of the current source program. In this command, *start* is the number of the first line of the source code which is to be displayed, and *number* is the number of lines which are to be displayed. Note that a hyphen separates the starting position from the number of lines to be displayed.

For example:

L1-30
will display lines 1 to 30 of the current program.

L20-55
will display the 55 lines starting at line number 20; that is, lines 20 through to 74.

The L command (and the appropriate Z command) can be used to inspect any item on any file. This is a useful way of checking a data record which has been or is about to be processed.

5.1.12 LP command

The debugger output is always be displayed on the screen. this is so even if the

(P

option has been specified on the RUN command.

However, there are facilities for sending the *program output* to the printer. This makes it easier to read the debugger messages and is particularly useful if the PRINT @ statements for cursor control and screen formatting in your program make it difficult to follow the action.

The

LP

command directs the *program* output to the printer, or from the printer back to the screen.

Note that the LP command switches the program output from the screen to the printer, or from the printer to the screen; the P command switches the displayed program output on or off. Neither command affects the debugger output which is always
displayed.

The PC command, described below, will close the spooler output file and print the output before the end of the job.

5.1.13 N command

The

Ncount

command directs the debugger to ignore the next n break points.

For example, the command

N5

will ignore the next five break points, although the debugger messages will still be displayed. This is useful if you

simply want to watch the path of the processing through the program.

The simple form:

```
N
```

will cancel any previous Nn specification.

5.1.14 OFF command

The

```
OFF
```

command is one of the fundamental commands used with the debugger, as described earlier in this *MB-Guide* and will terminate the program execution and log the user off the system.

5.1.15 P command

The

```
P
```

command is one of the fundamental commands used with the debugger, as described earlier in this *MB-Guide* and switches off the display of the program output to the screen, cancelling the program output. If a previous P command has switched off the output, a second P command will switch on the display of the program output.

Note that the LP command switches the program output from the screen to the printer, or from the printer to the screen; the P command switches the displayed program output on or off. Neither command affects the debugger output which is always displayed.

5.1.16 PC command

The

```
PC
```

command is used when the program is directing output to the printer (under the action of the P option on the RUN command or the LP debugger command) and you need to look at the output before the end of the job. The action of the command is to close the spooler job. Otherwise, the spooler output would not be printed until the end of the job.

5.1.17 R command

The

```
R
```

command is used at a break point within a subroutine (whether

an external subroutine called via CALL statement or an internal subroutine called via a GOSUB statement), and removes the most recent return address from the top of the stack of such address - as discussed for the S command - and then causes the processing to return to the place immediately after the most recent CALL or GOSUB statement.

5.1.18 S command

The

```
S
```

command is used to display the current list of statements which represent the return address from any internal subroutines, that is, those called by means of a GOSUB statement.

The stack is displayed in a form such as

```
= 300 = 176 = 40
```

with the return address for the most recent GOSUB to the left.

5.1.19 T command

The

```
Tvariable
```

will add a variable to the trace table, causing that variable name and its contents to be displayed at every break point.

Thus, the command

```
TSALES.FIGS
```

will add the variable SALES.FIGS to the symbol table and display the contents of the variable at every subsequent break point.

The Basic debugger will acknowledge that SALES.FIGS has been added to the trace table by displaying a

```
+
```

sign.

There may be up to six variables in the trace table at any one time.

If you attempt to use a variable which is not in the symbol table (or if the program has been compiled without producing a symbol table), then the debugger will display

```
SYM NOT FND
```

If the trace table is full, then the debugger will respond with the message

```
BRK TBL FULL
```

Variables may be removed from the trace table by means of the U command.

5.1.20 U command

The T command is used the add variables to the trace table. The

```
Uvariable
```

command will remove a variable which has been added to the trace table.

Thus, if you have previously added the variable GRAND.TOT to the trace table by means of the command

```
TGRAND.TOT
```

then the command

```
UGRAND.TOT
```

will remove GRAND.TOT from the trace table.

The Basic debugger will acknowledge that GRAND.TOT has been removed from the trace table by displaying a

```
-
```

sign.

You may have to use the D command to display the current trace table in order to known which variable entry is to be removed.

The simple form:

```
U
```

will remove all the variables from the trace table.

5.1.21 Z command

The Z command is used to specify the source program which is to be used by any subsequent L commands.

The simple form:

```
Z
```

will cause the Basic debugger to ask for

```
FILE/PROG NAME?
```

to which you will enter

```
filename programname
```

The file name and the program name may be entered in one step by means of a command of the form:

```
Z filename programname
```

If you issue an L command without having previously issued a Z command, then the Basic debugger will respond with the

```
NO SOURCE
```

message.

The L command (and the appropriate Z command) can be used to inspect *any item on any file*. This is a useful way of checking a data record which has been or is about to be processed.

6 Debugger messages

The Basic debugger displays the following messages:

*Bc n
: when a break occurs as a result of satisfying break condition number c.

*E1
: when a break occurs as a result of initiating execution and interrupt by means of the:

```
RUN fffff iiiii (D)
```

command.

*En
: when a break occurs as a result of an E command.

*In
: when a break occurs as a result of a program execution error or by use of the Basic DEBUG statement.

\+
: after a successful Tvariable or Bcondition command indicating that the entry has been added to the trace table.

\-
: after a successful Uvariable or Kn command indicating that the entry has been removed from the trace table.

£ > PROGRAM LENGTH
: if you issue a Gn command in which n is greater than the number of lines in the program.

CMND?
: if you enter an invalid debug command.

NO SOURCE
: if an L command is issued to display a source program and no Z command has previously been issued to identify the source program which is to be displayed.

STK EMP
: if you issue an S command to display the contents of the stack of subroutine return address and the stack is empty.

SYM NOT FND
: after a Tvariable command of which the variable cannot be found in the symbol table.

TBL FULL
: after a Tvariable or Bcondition command if the trace table is full.

UNASGN VAR
: if you attempt to use the /variable command to change the value of a variable which has not yet been assigned a value.

OBJECT VERIFIES
OBJECT DOES NOT VERIFY
: the ? or $ command displays the name of the program which is currently executing and the line at which the interrupt occurred. The Basic debugger will also display one these two error messages to indicate whether or not the object code verifies. If the message indicates that there is an error in the object code and the program should be recompiled.

In all cases, **n** is the number of the line at which the break occurred and which is *about to be executed*.

7 How to ...

In this section, we look at a number of specific questions which you might ask as you are trying to debug a Basic program, and we see how to answer these by use of the Basic symbolic debugger.

If you have any further questions of your own, send them to us. If we use them in a future edition of this *MB-Guide* will give you a credit and send you a free copy.

> My program has just stopped executing. It printed an error-message and an asterisk. What should I do?

If you did not write the program, then you should make a note of all the error message(s) which were displayed, and then call your System Manager, the programmer or someone else who is responsible for the program.

It might also be help if you were to type

```
?
```

followed by the <RETURN> key and make a note of the name of the program which will then be displayed.

If you want to abandon the program, you should type

```
OFF
```

followed by the <RETURN> key, or

```
END
```

followed by the <RETURN> key.

In some circumstances, it may be better to wait until the cavalry arrives in the form of the programmer who is responsible.

If you are the cavalry, then read on.

A Basic program is executing and it seems to be in a loop. How can I find out which program is executing?

Interrupt the processing by means of the <BREAK> key. This will invoke the debugger and tell you the line number at which you interrupted the processing. Then enter the debugger command

```
?<RETURN>
```

This will tell you the name of the program which is looping.

If you want to observe the loop in action, issue a command such as

```
E5<LINE FEED>
```

then press <LINE FEED> each time the processing is interrupted. By noting the line numbers at which the processing is interrupted you can observe the piece of code around which it is looping.

I am testing a program and I want to know the contents of some of the variables. How can I do this?

Interrupt the processing by means of the <BREAK> key, then issue a series of commands of the form

```
/xxxx<RETURN>
```

where xxxxx is the name of the variable. This will display the contents of the variables.

An alternative way is to issue a series of commands of the form:

    Txxxx<RETURN>

where xxxxx is the name of the variable. This will display the contents of the variables whenever there is a break in the processing.

How can I change the contents of one of the variables whilst the program is executing?

Interrupt the processing by means of the <BREAK> key, then issue a command of the form

    /xxxx<RETURN>

where xxxxx is the name of the variable. This will display the contents of the variable. To change the contents of the variable simply enter the new value followed by the <RETURN> key.

For example, to change the contents of the variable COUNTER to 0, the sequence might look like this:

    /COUNTER 99=*0*

Your typing has been shown in italics.

To start the processing off again, just press the

    <LINE FEED>

key.

I am testing a program which calls a lot of different external subroutines. How can I tell when the program jumps to a subroutine? Can I intercept it just before it goes to the subroutine?

Look at a listing of the program and make a note of the line numbers of the CALL statements which send the processing off to the external subroutines.

Interrupt the processing by means of the <BREAK> key, then issue a series of (up to four) commands of the form:

    B$=n<RETURN>

where n is the line number of the CALL statement.

Resume the processing by means of the <LINE FEED> key. The processing will be interrupted immediately before the CALL statement is executed.

You can then issue a command such as

E1<RETURN>

followed by a series of <LINE FEED>s to observe the flow of the processing into and out of the subroutines.

I using the Basic debugger to follow my way through a program, but the cursor keeps jumping about all over the place because I use the PRINT @ statement to position the cursor. What can I do about this?

Issue the command

LP<RETURN>

to direct all the program output to the printer, then carry on using the debugger.

How can I direct all the debugger messages to the printer so that I can check it later?

You cannot. All the debugger output is directed to the screen of your terminal.

How can I change the program via the Basic debugger?

The only changes you can make via the Basic debugger are

* change the contents of the variables (using the / command),

* change the processing sequence (using the G command to jump to another statement),

You cannot change the source code or any other item without leaving the Basic debugger.

How can I restart the program from the beginning?

Interrupt the processing by means of the <BREAK> key, then issue the command

G1<LINE FEED>

How can I restart a program once I've interrupted it?

Press the <LINE FEED> key to resume from the place where the interruption was caused.

If this does not succeed, then it may be best to abandon the processing by typing

```
END<RETURN>
```

and this will return you to TCL from where you can start again.

> These debugger commands all seem very complicated. Is there a simpler way of finding and clearing up errors in this program I'm writing?

Like all useful pieces of software, the Basic debugger has many more features than you may need at any one time. For the absolute beginner, these may seem quite overpowering, but you should give them a go.

If you don't have the time or need to learn all the commands, you can lead a useful and happy life using just the

```
E
G
/
?
```

commands, plus the <RETURN> and <LINE FEED> keys.

Later, you may extend this to include the

```
L
Z
```

commands to look at the source program.

The really useful commands are the

```
B
T
```

commands. These allow you to create interrupts at regular places in the processing and to inspect the contents of a number of useful variables automatically at each interrupt.

Without even this partial use of the Basic debugger, you are condemning yourself to a life of

```
PRINT 'HERE AT STATEMENT 999':; INPUT FRED

PRINT 'THE VALUE OF GTOTAL = ':GTOTAL
```

and an unnecessary round of compiling and recompiling your program.

What's the difference between using <RETURN> after a debugger command and using <LINE FEED>?

In general, the difference is that

* <RETURN> sends the command to the debugger so that the debugger will process the command and then come back to you for another debugger command.

* <LINE FEED> sends the command to the debugger so that the debugger will process the command and then the debugger will attempt to continue the processing.

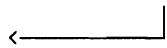
If you are in any doubt, use the <RETURN> key after all the commands and only use the <LINE FEED> key when you want the processing to carry on without you.

I don't have a <RETURN> key on my keyboard. What do I do about this?

Your keyboard should have a key marked

RETURN
or ENTER

or there may be a large key with a down-back pointing arrow, like this:



If you don't have any of these, then you could try holding down the <CTRL> key and pressing the M key at the same time.

I don't have a <LINE FEED> key on my keyboard. What do I do about this?

Your keyboard may have a key marked

LINE FEED
or LF

or there may be a small set of four keys with arrows pointing up, down, left and right: the down-pointing arrow may have the same effect as <LINE FEED>

If you don't have any of these, then you could try holding down the <CTRL> key and pressing the J key at the same time.

8 Glossary

The following terms are used in this *MB-Guide to the Basic debugger*:

Basic debugger a standard tool for use by programmers in identifying, locating and correcting errors in a Basic program.

Basic symbolic debugger another name for the Basic debugger.

Break point a situation in which the normal processing action of a program is interrupted - either by accidental error or by design - and the Basic debugger is called into action by the operating system.

Breakpoint table is that part of the trace table which holds details of up to four break conditions set by means of the B command.

Interactive debugger another name for the system debugger.

Symbol table a list of the names of the variables used in a program and their location with the run-time storage area. This table is generated by the compiler and used by certain Basic debugger commands.

Symbolic debugger another name for the Basic debugger.

System debugger a standard tool for use by systems programmers in identifying, locating and correcting errors in an assembly language process. Its main application by end-users is in inspecting and changing parts of the virtual memory system.

Trace table a work table used by the Basic debugger which consists of a list of the names of up to six variables whose contents are to be displayed automatically at each break point, and a list of up to four break conditions which are to cause an interrupt.

In this and the other *MB-Guides*, the keyboard control keys have been represented by their name enclosed in angle brackets:

<BREAK> identifies the break key or the equivalent sequence which interrupts the current process.

<CTRL> identifies the control key.

Certain characters are entered at the keyboard as a combination of one or more of the above keys together with other keyboard characters. For example, the subvalue-mark (character 252) may be entered as:

<CTRL> \

that is, by holding down the <CTRL> key and typing the normal \ character at the same time. Similarly, the value-mark can be keyed in as the sequence <CTRL>] and the attribute-mark as the sequence <CTRL> ^

<ENTER> identifies the ENTER key which is used to transmit each piece of data to the system.

This is generally represented by the <RETURN> key in the text.

<ESC> identifies the ESCAPE key.

<LINE FEED> identifies the line-feed key. . On some keyboards, this may be the down-pointing arrow. The sequence <CTRL> J is equivalent.

<RETURN> identifies the RETURN key which is used to transmit each piece of data to the system.

On some keyboards, this may be the <ENTER> key or the down-left-pointing arrow key. The sequence <CTRL> M is equivalent.

Index

MB-Guides

MB-Guides are designed to serve as introductory texts to a range of fundamental topics within the Pick operating system. They will be available for the following subjects:

MB-Guide to Access conversions and correlatives
MB-Guide to Access sentences
MB-Guide to Basic programming
MB-Guide to Creating and using Procs
MB-Guide to using the Editors
MB-Guide to File design
MB-Guide to File-save and file-restore
MB-Guide to Files: monitoring and sizing
MB-Guide to Group format errors
MB-Guide to Operations and systems management
MB-Guide to Pick on the PC
MB-Guide to Producing training courses
MB-Guide to Producing documentation
MB-Guide to Program design
MB-Guide to Security
MB-Guide to The Basic symbolic debugger
MB-Guide to The spooler
MB-Guide to The system debugger
MB-Guide to Using backing storage

The format of the *MB-Guides* is such that they may be easily updated and amended to reflect the current state of the operating system. In order that this and the other Mating MB-Guides continue to meet the needs of the users, we would appreciate your comments on this guide and your suggestions for further titles in this series.

MB-Master self tuition courses are also available on a wide range of topics related to the Pick operating system:

Moving to Basic - a conversion course
Pick systems management
Programming in Basic
Starting Access
Starting ACCU/PLOT
Starting CompuSheet+
Starting Jet
Starting Pick
Starting Runoff
Writing Procs

If you have any comments on this *MB-Guide* or any suggestions for further title in the series, then please send your suggestions to:

MALCOLM BULL
MALCOLM BULL TRAINING AND CONSULTANCY SERVICES
19 Smith House Lane
BRIGHOUSE
West Yorkshire
HD6 2JY

Telephone: 0484-713577 **Fax: 0484-714112**

We are particularly concerned about:

* any additional information which you would like.

* any topics which you feel are superfluous.

* any inaccuracies which you noticed.

* any areas in which the information given in the *MB-Guide* did not apply to your implementation. Please indicate what implementation you were using.

* any other suggestions, observations and comments.

If you do write to us, please give your name and address so that we can acknowledge your contribution in the next Edition of this *MB-Guide*.

MB-Guides

003
001

The booklets in the *MB-Guide* series cover a range of fundamental topics of interest to users and those responsible for running Pick systems.

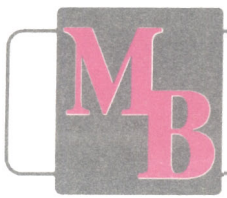
Each *MB-Guide* deals with a specific aspect of the operating system and the booklets represent an economical introduction to the various topics and the whole series forms an integrated presentation of the subject matter.

The booklets are intended to be a working document and, for this reason, space is provided for the user's notes, and the reader is encouraged to amend the booklet so that it applies to his/her own system.

It is anticipated that the series of *MB-Guides* will be of special interest to new users, and it should prove useful for training organisations, software houses and others who are responsible for the instruction of their clients and staff in the fundamental aspects of the Pick operating system.

B
a
s
i
c

d
e
b
u
g
g
e
r



Malcolm Bull

Training and Consultancy Publications