*Chapter 7*

# PROGRAMMING ERRORS

The errors that occur in programming divide broadly into four types

1. Syntax errors.
2. Semantic errors.
3. Execution or ‘run-time’ errors.
4. Logical errors.

The errors that are associated with the operating system and the disc errors are not discussed extensively. The operating system does not generate error messages since the operating system calls are not error trapped. This presents no problem to anyone who only uses the standard BASIC features, but the wide range of non-standard facilities are too valuable to omit from many programs. Once any of the operating system commands are used it is necessary to be exact and correct since no safety-net to trap errors is provided.

**Example 7.1**

The incorrectly written and poorly structured program fragment below does not give an error message when the incomplete VDU 23 statement that needs 9 arguments is executed. The arguments required are obtained from the following INPUT statement, which consequently gives an incorrect prompt!

```
10 MODE 4
20 PRINT "CHOOSE A LETTER"
30 VDU23,234,129,255,129,
40 INPUT "NEVER GIVE A VOWEL",A$
50 IF (A$="A") OR (A$="E") OR (A$="I") OR (A$="O") OR (A$="U")
   THEN 20
60 PRINT "CHARACTER 234 ";CHR$(234)
70 PRINT "THIS IS THE END!"
80 END
```

The only protection against this type of error is careful and precise programming followed by thorough tests of all possible running conditions.

## 7.1 SYNTAX ERRORS

This type of error occurs whenever the computer cannot make sense of the commands that you have issued. There may be a variety of reasons for this. A typing error may result in an incorrectly spelt keyword which the computer will interpret as a variable. For example

```
10 IMPUT X
```

will generate the error message

```
Mistake at line 10
```

when it is run. This line may be inadvertently corrected to

```
10 IMPUT-X
```

if the spelling error is not noticed. This will generate the error message

```
No such variable at line 10
```

when it is run. The line may next be corrected to

```
10 INPUT=X
```

which, when it is run, will generate the error message

```
Syntax error at line 10
```

The correct form of the line is

```
10 INPUT X
```

It is important to realize that the error message needs to be interpreted as it may not be sufficiently specific to identify directly the error in the line. Any line containing an error should be read thoroughly to check that each character is correct. It should not be thought that all error messages are obscure; for instance the line

```
20 PTR=0
```

will generate the error message

```
Missing # at line 20
```

The error message is helpful if a program involving the use of the file pointer PTR#<channel number> is under development, but confusing if you want to

have a variable PTR to which you are assigning the value zero. However, once it is remembered that PTR# is a BASIC keyword and that these can not be used as variable names, it will be clear that PTR is not available as a variable name. It is, of course, perfectly in order to use ptr or Ptr as BASIC variable names.

Another way in which a syntax error may be generated is by an incorrectly inserted or omitted bracket which may confuse the computer. Another common error to watch out for is confusion between the letter O and zero. Apart from errors of typing you may have got the form of instructions confused, as for instance

```
FOR J=10 STEP 2 TO 20
```

These errors are usually easy to correct, since the computer always reports the line number in which the error occurred, so that this line can be LISTed and examined for faulty construction. Certain syntax errors will generate specific error messages, such as

```
Missing ) at Line 30
```

which would arise if the following lines formed part of a program which was then run.

```
10 DIM D(6)
20 FOR I=1 TO 6:NEXT
30 PRINT D(5
```

Another helpful syntax error message is ‘Type mismatch’ which will be generated by statements such as

```
10 A$=1
```

or

```
20 A="HELLO"
```

when a program containing them is run.

In the first case an attempt has been made to assign a numeric value to a string variable. In the second case an attempt to assign the string "HELLO" to a numeric variable has been made. Clearly the correction of errors such as these will depend upon what was originally intended.

The error ‘No such variable’ will occur if a variable appears on the right hand side of an equation or in a ‘PRINT’ statement before it has been given a value in a program statement such as

```
10 LATEST=0
```

Remember, however, that a statement in which a variable appears on both left and right hand side of the equals assignment such as

```
10 NUM=NUM+1
```

is allowed and will set NUM=1 the first time it is executed. The second and subsequent times that it is executed the value of NUM will be incremented as expected. The resident integer variables @% and A% to Z% may be used in PRINT statements and on the right hand side of equate statements without restriction and will have the values left over from use in previous programs. They will, however, need to be initialized if their previous values are not required. An attempt to use an array element in a statement such as

```
10 Q(3)=7
```

without declaring the array by the statement

```
5 DIM Q(10)
```

will generate the error message

```
Array at line 10
```

But many syntax errors will just result in the message ‘Syntax error’ or even ‘Mistake’ if the computer cannot make head or tail of what you meant. It is then necessary to read the line of program concerned to check that intended statements have been correctly formed. If the cause of the error is not identified in the line at which it is reported, then the variables in the line should be traced in earlier lines of the program to check them.

## 7.2 SEMANTIC ERRORS

It is not always possible to make a hard and fast distinction between semantic and execution errors, but generally speaking semantic errors arise when commands which have correct syntax are incompatible in some way. For instance, if you accidentally cross a pair of FOR loops, or jump into a FOR loop, you will get the message

```
No FOR
```

or

```
Can't match FOR
```

**Example 7.2**

The program below contains a deliberate semantic error

```
10 DIM A(9)
20 I=0
30 GOTO 50
40 FOR I=1 TO 9
50 PRINT A(I)
60 NEXT
```

When it is run it will generate the message

```
No FOR at line 60
```

In this case line 60 is a valid line. The mistake has been made at line 30, which causes the program to jump into a FOR...NEXT loop. This mistake is clearly indicated by the error message. If line 20 is omitted the error message that occurs when the program is run is

```
No such variable at Line 50
```

which is not so helpful. This is an example of an error message for which it is necessary to follow the program through to track down the cause.

The error message may not occur the first time a section of program is executed. The Example 7 .3 contains crossed FOR...NEXT loops.

**Example 7.3**

```
10 FOR I=1 TO 10
20 FOR J=1 TO 10
30 PRINT I*J
40 NEXT
50 NEXT J
```

When this example is run on the computer it will generate the error message

```
Can't match FOR at line 50
```

after the first 10 numbers have been printed. This occurs because the BASIC interpreter has assumed that the NEXT at line 40 belongs with the FOR at line 20. The program can be corrected in two ways, either- by changing line 50 to

```
50 NEXT
```

or by changing line 40 to

```
40 NEXT J,I
```

and omitting line 50. The latter saves a line of program.

In fact, all the messages of the form

```
No...
```

indicate semantic errors. Another example of a semantic error is

```
Arguments
```

which occurs when too many or too few arguments are given in a function or procedure call.

Most semantic error messages are self-explanatory. A list of all the possible error messages is given in Appendix J and the precise meaning of each is given in Section 46 of the BBC User Guide. The combination of an explicit error message with a line number makes semantic errors simple to track down in most cases.

## 7.3 EXECUTION ERRORS

These errors usually occur when the program itself makes complete sense to the computer, but, at least in certain circumstances, something goes wrong when the program is RUN.

In some cases these execution errors may be due to insufficient protection for unlikely data values which are not worth catering for, but very often they arise from a fault in the logic used to draw up the program.

Error messages usually associated with execution are relatively few, and would include

```
Accuracy Lost
Bad DIM (this could also be due to a semantic error)
Division by zero
Eof
Exp range
Log range
-ve root
ON range
Out of DATA
String too long
Subscript
Too big
```

Errors such as ‘No such variable’ may also be errors of execution caused by the program taking an unplanned route and reaching a statement which uses the variable before it has been assigned.

By their very nature, execution errors are much the most difficult to locate, and may be due to the values which variables happen to have on one occasion. For instance

```
S=SQR(B*B-4*A*C)
```

is fine unless it happens that 4AC>B2, or B is greater than about $10^{20}$, and the quadratic equation

$Ax^2+Bx+C=0$

has the perfectly good solution x=-C/B when A=0, but use of the formula

(-B±S)/(2A)

will cause a ‘Division by zero’ error.

In association with the line number reported along with the error, an nvaluable aid in tracking down these errors is the fact that all variables and may elements (including the control variables of FOR loops) retain their values after the execution of a program is halted due to an error. They are also retained if a program locked in an infinite loop is stopped by <ESCAPE> (but not <BREAK>).

When the source of an execution error is not obvious, therefore, the first thing to try, which will usually solve the problem, is to PRINT, in immediate mode, the variables on the offending line (and possibly some of the other program variables also, particularly the control variable of any FOR loop if the program stopped inside it). The values obtained should then be compared with those expected. Any discrepancy should be investigated as a possible source of error in the program. Alternatively an extra PRINT statement may be inserted in the program, inside the loop. This can be removed later when the error has been located and corrected.

## 7.4 RERUNNING THE PROGRAM

It may be that the values of the variables obtained, after the execution of a program has been stopped by an error, are themselves inexplicable. For instance, in the earlier example of a ‘Division by zero’ error, it might well be that the variable A should not have had the value zero because the program contained steps to cater for this eventuality.

The TRACE ON command causes the line numbers to be displayed as the program is executed each time a different line is reached. This enables you to trace the route taken through the program, and may for instance reveal that the A=0 trap was accidentally bypassed. TRACE ON can be inserted at any point in a program, to avoid an excessive listing of line numbers, and can also be switched off again with TRACE OFF, so you can just trace sect-ions of your program if you wish. The TRACE ON facility is also turned off by the use of the <ESCAPE> and <BREAK> keys. There is a third form of the TRACE command, TRACE <line number>, which traces only lines with numbers greater than <line number>. This facility is provided in case you wish to avoid tracing through tried and tested procedures called earlier in the program.

It may be possible to correct a mistake in a program temporarily by reassigning variable values in immediate mode. It is then possible to continue execution by the command

```
GOTO <line number>
```

where <line number> is the faulty line which you have ‘corrected’. Unlike RUN, GOTO does not clear the values of variables, so these are retained and the program may continue successfully. (This tip may even be of value in other circumstances, say where you missed reading some answers in the program before they rolled off the screen, or if you want to reprocess some numbers which took a long while to calculate or to type in.)

The STOP command can be useful in tracking down the cause of errors in a program. If it is inserted before a section of program that is suspected as the source of the error the values of variables can be checked, by PRINT statements in immediate mode, to see that they are as expected before this section is executed. The GOTO <line number> statement can then be used to show conclusively whether or not the error is in the section of program that is suspect.

***Exercise 7.1***

The program below which is available on the supplementary disc, contains a variety of errors of different sorts. RUN the program to track them down and try to correct them, and use the technique of PRINTing in immediate mode to help with any execution errors.

```
 10 REM INITIALIZATION OF VARIABLES
 20 DIMV(2)
 30 V(1)=70: V(2)=99: V(3)=140
 40 REM G=9.81
 50 REM DISTANCE OF TARGET
 60 D=RND(1500)
 70 REPEAT
 80 REM DISPLAY
 90 CLS
100 PRINT TAB(10);"Target at ";D;" metres"
110 PRINT 'TAB(14);"MAXIMUM RANGE"
120 PRINT '"Missite 1      Missile 2      Missile3"
130 PRINT " 500m           1000m          2000m"
140 INPUT'"Select missile to fire (1,2,3) ",M
150 IF M<1 OR M>3 THEN GOTO 80
160 IF INT(M)<>M THEN GOTO 80
170 INPUT "Elevation (degrees) ",A
180 PRINT ''
190 REM SELECT SPEED OF MISSILE
200 V=V(-M)
210 REM CALCULATIONS
220 H=(V*SIN(RAD(A)))^2/(2*G:REM MAX.HEIGHT
230 T$=2*V*SIN(RAD(A))/G REM TIME OF FLIGHT
240 R=V^2*SIN(RAD(2*A))/G: REM RANGE
250 REM PRINTOUT
260 PRINT "Time of flight: ";INT(T);" sec"
270 PRINT"Maximum height:";INT(H);" metres"
280 PRINT"Range:";INT(R);" metres"'
290 REM RESULT OF FIRE
300 E=INT(R)-D
310 IFABS(E)<10 THEN GOTO 360
320 PRINT "Shot ";ABS(E);" metres ";
330 IF E>0 THEN PRINT "behind target" ELSE PRINT "short of
    target"
```

```
340 PRINT '"Press any key to continue:";
350 A$=GET$
360 UNTILABS(E)<10
370 PRINT "DIRECT HIT! TARGET DESTROYED"
380 INPUT'"Would you like another go (Y/N)? "YN$
390 IF LEFT$(YN$,1)="Y" THEN GOTO 20
```

## 7.5 ERROR TRAPPING

In some programs it may be convenient to suppress error messages and to effect recovery from situations which would otherwise halt execution of the program. The command ON ERROR will accomplish this. It has four forms

```
ON ERROR GOTO <line number>
ON ERROR GOSUB <line number>
ON ERROR <statements>
ON ERROR OFF
```

The first three commands cause any subsequent error messages that would otherwise occur during the running of a program to be suppressed, and in th.e first two cases the program jumps to <line number> rather than having its execution stopped. <line number> would normally be the start of some error handling section. If GOSUB is used with ON ERROR then a RETURN must be executed at the end of the error handling section to allow the program to continue from the ON ERROR statement. There may be several ON ERROR statements in a program; the line number to which execution jumps In the case of an error is the one indicated by the last ON ERROR passed in mrmal execution of the program.

The last form, ON ERROR OFF, is used to cancel the error trapping, when you want normal error reporting to resume.

In Example 7.4, ON ERROR GOTO is used to trap two ‘events’

1. The input of IMIN>IMAX which will lead to failure at line 110 in the square root function.
2. At line 180 a deliberate divide by zero is used to cause the request of the required YES or NO response.

In both cases the program will go to line 140. Try this example without line 10 first; this will enable you to find program mistakes rather than user mistakes.

**Example 7.4**

```
 10 ON ERROR GOTO 140
 20 REM TEST OF ON ERROR GOTO XX
 30 CLS
 40 PRINT "Fringe visibility calculation"'
 60 INPUT "Imax and Imin ",I1,I2
```

```
 70 V=(I1-I2)/(I1+I2)
 80 IF V<=0 THEN PRINT '"Imin must be Less than Imax"
100 PRINT '"Fringe visibility = " ;V
110 ROOT=SQR(V)
130 PRINT '"Root of V = ";ROOT
140 INPUT '"Do you mt to do another calculation",A$
160 IF LEFT$(A$,1)="Y" THEN GOTO 60
170 IF LEFT$(A$,1)="N" THEN GOTO 190
180 V=V/0
l90 END
```

Try the data 2,1 then 1,2 and responses other than YES or NO to line 140.

***Exercise 7.2***

Sometimes it is desirable to correct an error in some way and then allow the program to continue. This can be done if the GOSUB form of the ON ERROR statement is used.

Modify Example 7 .4 as follows to illustrate the effects of ON ERROR GOSUB.

```
 10
 20 REM TEST OF ON ERROR GOSUB XX
 65 ON ERROR GOSUB 200: GOTO ERL
175 U=0
180 V=V/U
185 PRINT "RETURNed with U = ";U
188 GOTO 140
200 U=U+0.1: V=1: RETURN
```

Note that the RETURN is to the point immediately after the GOSUB on the ON ERROR line, rather than the point where the error occurs, which is what you would probably assume. This can be circumvented by the inclusion of GOTO ERL at the end of the ON ERROR line, which jumps back to the offending line. (Check that this is the case by leaving out GOTO ERL.) Actually this feature renders the use of GOSUB pointless - it would be simpler to use ON ERROR GOTO 200 and end line 200 with GOTO ERL instead of RETURN.

If the error is not corrected before RETURNing to the main program you can get into an infinite loop. Modify the program further by

```
200 U=U+0.1: PRINT "E";: RETURN
```

and use the data 1,2 in response to line 60. Press <BREAK> to get out of the infinite loop.

### 7.5.1 Error codes

When an error occurs in a program, the computer stores the code number of the error, and the line number on which it occurred, in the system variables

ERR and ERL respectively. In this way you can check in the error handling section which error occurred, and possibly deal with those you can anticipate and end the program with the rest. The error number of every error message is given in Appendix J, and you can look up the codes for the ones that you anticipate may occur.

For instance, the error numbers for '-ve root' and 'Division by zero' are 21 and 18 respectively, so we could further modify the program of Exercise 7.2 as follows

***Exercise 7.3***

```
200 U=U+0.1:V=1
210 IF ERR=18 OR ERR=21 THEN RETURN
220 PRINT "Error number ";ERR;" occurred at line number ";ERL:
    END
```

An interesting possiblility arises in that pressing the <ESCAPE> key is treated as generating an 'Escape' error with code number 17. Thus it is possibIe to protect your programs from users breaking out of them (whether by accident or on purpose) by trapping error number 17. To give further security you can also prevent users getting out of a program with <BREAK>, by issuing the command

```
*KEY 10 OLD|M RUN|M
```

This gives limited protection unfortunately, as one can always get out of any program with <CTRL-BREAK>, which does not allow the <BREAK> key to behave as a function key.

**Example 7.5**

This program illustrates the redefinition of the <BREAK> key (function key 10) to provide limited protection for a program.

```
 5 ON ERROR GOTO 60
 6 *KEY10 OLD|M RUN|M
10 INPUT "Type a number from 1 to 10",A
20 X=1/A
25 X=SQR(A)
30 X=SQR(10.1-A)
40 PRINT "good, ";A;" is between 1 and 10"
50 GOTO 10
60 IF ERR=17 THEN PRINT: PRINT "Ha Ha, you can't ESCAPE that
   easily": GOTO 10
70 PRINT "Error number ";ERR;" occurred at line number ";ERL
80 GOTO 10
```

Try this program out with correct and incorrect responses, including letters, and try and get out of the program with <ESCAPE> and <BREAK>.

Finally, when reporting the error on line 70, the error number is not very illuminating. There is a command, REPORT, which will report what the last error to occur was.

Try adding to the above program the extra line

```
75 REPORT: PRINT
```

### 7.5.2 ON ERROR limitations

It is not possible to resume a program part of the way through a FOR...NEXT loop, a REPEAT...UNTIL loop, a procedure, function or subroutine. Instead, you will have to restart the loop, procedure or whatever. This is necessary since system pointers are changed by the operation of the ON ERROR GOSUB statement when an error occurs. If after an error an attempt is made to GOTO ERL to within a FOR loop, for instance, then a 'No FOR' error will occur.

Since this error will again be trapped with another jump into the error handling subroutine, the computer will get stuck in an infinite loop and the system will be effectively 'hung'. You will need to press <BREAK> or <CTRL-BREAK> to regain control.