17 Procedures

Using procedures allows you to split up virtually any program you are going to write into a main program, followed by a number of ‘mini-programs’ (procedures), which can be called from the main program by a single statement.

A procedure is simply a collection of numbered BASIC statements which you write in order to perform a particular task. This collection of statements looks just like part of an ordinary program, but the differences are that the first fine contains the name of the procedure (which is decided by you), and the last line contains a BASIC word to signify the end of the procedure. When the computer encounters the end, it then returns to the main program and carries on.

The rules for using procedures are very simple. A procedure is called from the main program by the BASIC word PROC followed immediately by the procedure’s name. The name can be anything you like, but there must be no spaces in it. For example:

```
PROCnewline
PROCwait_a_second
PROCDRAWPICTURE
```

Note the underline character in the second example which helps ‘space out’ the name.

A procedure name should reflect the function of the procedure to which it applies. If you merely name your procedures **PROCA**, **PROCB**, **PROCC**, for example, then no one will understand what they do without having to work through each one. So if you have a procedure which converts feet into metres, then call it **PR0Cfeet_to_metres**. It is best to use lower case names for procedures so that they distinguish themselves from the **PROC**.

To define a procedure, you simply type a line number, followed by **DEF**, followed by the procedure name. It is a good idea to start defining your procedures at a fairly high line number, say 1000.

```
1000 DEF PROCwait_a_second
1100 NOW = TIME
1200 REPEAT UNTIL TIME-NOW>=100
1300 ENDPROC
```

This procedure will do as its name suggests.

ALL PROCEDURE DEFINITIONS MUST END WITH **ENDPROC**.

When you want the computer to carry out the instructions in the procedure, you have to call it by name:

```
70 PROCwait_a_second
```

or

```
120 IF INKEY$10 = "W" THEN PROCwait_a_second
```

You may have as many procedures in your program as you like, and usually the more the better.

THERE MUST ALWAYS BE AN **END** INSTRUCTION BETWEEN THE END OF THE MAIN PROGRAM AND THE PROCEDURE DEFINITION.

For example

```
10 REM Sample program
20 FOR X = 0 TO 29
30 PRINT TAB(10,10);"COUNTING..."X" Seconds "
40 PROCwait_a_second
50 NEXT
60 PRINT TAB(10,10);"Half a minute up!  "
70 END
1000 DEF PROCwait_a_second
1100 NOW = TIME
1200 REPEAT UNTIL TIME-NOW>=100
1300 ENDPROC
```

There is a program on the Introductory Cassette to illustrate the use of procedures, and also give you some fun. Load this program which is called

‘BUGZAP’ into your Electron first. See chapter 4 for instructions on how to load a program.

When you **LIST** a long program obviously you cannot see all of the lines on the screen at the samee time. Using **LIST** with specified line numbers is one way around this, but another is to put the computer into paged mode. This is done by pressing **CTRL** N (No **RETURN** is required.) If you now use **LIST**, the program will be listed until the screen is full. When you want to look at the next part just press **SHIFT** and another screen full will appear. If you want to change a line number, you must press **ESCAPE** . To get the computer out of paged mode, type **CTRL** O.

Look at just one procedure from this program:

```
520DEF PROCinfo
530CLS
540PRINT'''''"Welcome to the game of Bugzap
!"'''
550PRINT"The object of the game is to use y
our"
560PRINT"laser gun to zap the descending bu
g"
570PRINT"before it lands or bombs you."
580PRINT'"Your score increases every time y
ou"
590PRINT"zap the bug, with more points bein
g"
600PRINT"given the lower the bug is; it wil
l be"
610PRINT"displayed when you are killed."
620PRINT'"The controls are:"'
630PRINT"Z     =  left"
640PRINT"X     =  right"
650PRINT"SPACE =  fire"
660PRINT''"Pressing the ESCAPE key will tak
e you"
670PRINT"to the end of the program."
680PRINTTAB(5,31)"Press SPACE to start the
game";
690REPEAT UNTIL GET$=" "
700ENDPROC
```

This procedure is called from fine 90 of the main program.

```
90PROCinfo
```

Line 520 is the start of the definition

Line 530 clears the screen.

Lines 540 to 680 print the introduction and instructions about the BUGZAP game which you see when you run the program.

Line 690 is an example of putting two separate BASIC statements after one line number by separating them with a colon. The purpose of line 698 is to wait until the space bar is pressed: **GET$= ""**. When the space bar is pressed, line 700 is executed.

Line 700 signifies the end of the procedure, and the computer goes back to the main program to the line immediately after the procedure call (**90PR0Cinfo**), which is fine 100.

Here is one of the procedures from the ‘MARSLANDER’ program also on the Introductory Cassette.

```
860DEF PROCrocket(direction%)
870REM If there is any fuel then fire rocke
t-motor and make sound
880IF fuel% THEN fuel%=fuel%-1 ELSE ENDPROC
890IF fuel%=29 THEN SOUND 1,-10,60,10 ELSE
SOUND 0,-1,5,2
900ON direction% GOTO 910,920,930,940
910VY%=VY%- 5:ENDPROC
920VX%=VX%+10:ENDPROC
930VY%=VY%+15:ENDPROC
940VX%=VX%-10:ENDPROC
```

This procedure alters the speed of the spacecraft according to the direction in which it is pointing, which is given by the variable direction%.

This procedure is called from fine 250.

```
250IF INKEY(-99) THEN PROCrocket%(Z%)
```

Z% is an integer variable which is used by the program to give the attitude of the spaceship.

When the computer reaches line 250 it tests to see if the space bar is pressed. If it is, the computer then places the contents of Z% into direction%.

The variable Z%, and hence the parameter direction%, can be anything from 1 to 4, where 1 indicates the capsule pointing up, 2 to the fight, 3 down, and 4 to the left. These positions are represented by characters 224 to 227 which are user-definable.

Line 880 checks to see whether there is any fuel left. The variable fuel% will be FALSE when it is zero and the procedure will end.

If it is TRUE one unit of fuel is deducted by decrementing its contents by 1. Line 890 makes either a ‘beep’ (fuel is low), or a rocket motor sound (fuel is not low).

SOUND is explained in chapter 22.

Line 900 uses **ON . . . GOTO**, to act according to the direction of the spacecraft The parameter direction% now contains the value given to it by Z%. If the spacecraft is pointing up, direction% is 1 and execution continues at line 910.

Line 910 decreases the vertical speed of the capsule. (VY% is the vertical speed measured positive in a downward direction; VX% is the horizontal speed measured positive in a left-to-right direction.) If the capsule is pointing to the left fine 960 passes execution to fine 920 which increases the horizontal speed of the capsule.

Lines 930 and 940 increase the vertical speed and decrease the horizontal speed respectively. After any one of these lines (910 to 940) has been executed, the procedure ends.

Using parameters in procedures

Using the above example, Z% and direction% are termed parameters. The idea behind using parameters is that they are more efficient than global variables. A global variable is one which is accessible throughout the whole program, and may be altered or re-assigned at any line number.

Once a global variable such as Z% has been passed to the procedure as a parameter, the variable which takes its place, direction%, is only known to that procedure. Outside **PROCspaceship** you can ask the computer to

```
252 PRINT direction%
```

and it will give an error because the variable direction% does not exist in that part of the program. Global variables which are passed to the procedure are called the actual parameters, and the variables within the procedure are called formal parameters.

A procedure may be defined with only one parameter, or it may be defined with lots of parameters. But a procedure must always be called with the correct number of parameters. **PR0Cspaceship**, starting at line 750, has three parameters.

So you could not call **PR0Cspaceship(X%,Y)**.

Parameters may be integer, real, or string. If a string variable is used as a formal parameter then it must have either a string or a string variable passed to it. Real and integer parameters may be passed to one another and interchanged freely, but remember that the fraction part of a real variable will be lost when assigned to an integer variable.

The idea of a variable being defined only within a certain section of a program is commonplace in a lot of computer languages, but unusual in BASIC. Electron BASIC allows you to declare any variable as *local* to a procedure or function (functions are discussed in chapter 19). A local variable may even have the same name as a global variable in the same program, but will lead a separate existence.

For example:

```
10 FOR I = 1 TO 3
20 PROClocal(I)
30 PRINT “OUT OF PROCEDURE I = ”;I
40 NEXT I
50 END
60 DEF PROClocal(J)
70 LOCAL I
80 I = J
90 I = I*10
100 PRINT “IN PROCEDURE I = ”;I
110 ENDPROC
>RUN
IN PROCEDURE I = 10
OUT OF PROCEDURE I = 1
IN PROCEDURE I = 20
OUT OF PROCEDURE I = 2
```

```
IN PROCEDURE I = 30
OUT OF PROCEDURE I = 3
```

Notice line 50 which says **END**. Because procedures are usually defined at the end of a program. you sometimes need to stop the execution after all the calls have been made. The program will terminate when the computer reaches the instruction **END**.

There is still another way to use procedures, and that is *recursively*. A recursive procedure is one which calls itself from within its own definition

```
10 answer = 1
20 INPUT X
30 PROCfactorial(X)
40 PRINT answer
50 END
60 DEF PROCfactorial(N)
70 answer = answer*N
80 IF N>1 THEN PROCfactorial(N-1)
90 ENDPROC
```

This is a recursive procedure to find the factorial of a number. Check through the logic of it in your head to see that it works. Recursive procedures are very useful in certain circumstances, but they consume memory very quickly.