SUPERBASIC

Contents

Fitting SUPERBASIC 2

Introduction to the facilities 3

LOCOS 4
HICOS 4
String Handling 4
LEFT$ 4
RIGHT$ 4
MID$ 5
INSTR 5
String concatenation 5

Multidimensional Arrays 6

READ..DATA..RESTORE 10
WHILE 10
ENDWHILE 11
FWHILE 11
XIF 11
XFIF 13
ON..GOTO 13
INKEY 13
BEEP 14
STOP 14
RERUN 14
ZERO 15
RESET 15
ERR 15
ERL 15
REPORT 15

Error Handling 16

ON..ERROR 16
FREE 16
PRINT TAB AND FPRINT TAB 17
b (Binary) 18
f (Base four) 18

User Defined Shapes 19

DEFSHP 20
INVSHP 20
MOVSHP 21

Points to note when using SUPERBASIC 23

Memory usage 24

Error messages 25

SUPERBASIC FITTING INSTRUCTIONS

Please read and understand these instructions before proceeding

SUPERBASIC is supplied in a 4K EPROM which must be inserted in socket 24 of your Acorn Atom. To fit the chip, turn the Atom upside down and remove the two large crosspoint screws which attach the base to the keyboard. Lift off the base and identify socket; this is the 24-pin socket wich lies between two 40-pin sockets, one holding the 6502 microprocessor , and the other the INS 8255 I/O chip. If a colour board is fitted this will have to be removed before proceeding, as it partly obscures socket 24. Insert the SUPERBASIC EPROM in socket 24 so that the end of the chip with the small semi-circular indent points towards the heat-sink at the edge of the printed circuit board. Make sure that none of the pins of the EPROM have become bent, and that they are all in the socket. When you are sure that SUPERBASIC is correctly installed, check once more to be certain, if necessary refit the colour board, then replace the base and screws.

SUPERBASIC

After installing the ROM (and selecting it if you have a selector board fitted) it should be enabled by typing LINK #AF00. This selects the 1200 baud cassette system, and prints a message on the screen to show that SUPERBASIC is ready for use. Alternatively, SUPERBASIC can be enabled by the command LINK #AF04 - this is identical, except that the 300 baud cassette system is selected, which is compatible with programs recorded in the ordinary ATOM format. In addition to the numerous facilities documented in more detail below, SUPERBASIC has the following advantages:-

(a) During cassette operations, each byte sent or received appears at the top right hand corner of the screen.

(b) The bug in ATOM BASIC, whereby an error message is given after use of a control character at the keyboard (e.g. CTRL-L to clear the screen), has been cured.

(c) In ordinary ATOM BASIC, there are a number of occasions when the line number given in an error message is wrong. Examples are when one tries to GOTO a non-existent line label; and when an error occurs in the statement immediately following a return from a subroutine. Attention has been paid to eliminate these problems in SUPERBASIC.

(d) When errors do occur, SUPERBASIC prints out not just an error number, but also the entire offending line. The position which the interpreter had got to when the error occurred will be highlighted in inverse video. In the case of a syntax error, the problem will normally be quite close to the highlighted position. Logical programming errors, of course, could be some distance away, but the printout should at least assist diagnosis.

(e) Practically all existing ATOM BASIC programs will run under SUPERBASIC without modification (or with only very minor modification in a few instances). Thus, you can add SUPERBASIC statements to your existing programs, perhaps to give better string handling, better graphics, or better sound. [In contrast to the BBC-Basic board, which though no doubt very powerful, will only run programs specifically written in BBC-Basic.]

(f) The command RUN can now be abbreviated R. - but see note 4 "Points to note when using SUPERBASIC".

SUPERBASIC FACILITIES

Where appropriate the smallest permissible abbreviation is given in curly brackets after each statement or function name

LOCOS {LOC.}

Switches to the 300 baud cassette system.

HICOS {H.}

Switches to the 1200 baud cassette system. Note that the cassette commands used in either system are exactly as they are without SUPERBASIC, e.g. LOAD, SAVE, *RUN, *SAVE etc.

STRING HANDLING

In the following definitions of the various string functions, the word string may be replaced by any of the following:

(a) a string in quotation marks, e.g. "HELLO"

(b) a string variable such as $A or $X, provided that A and X have previously been set to a suitable area of memory, e.g. by a DIM statement. The dollar sign may be followed by other things than simple variables, as in ordinary ATOM BASIC - e.g. $8200. But complex expressions such as $A + 3 require brackets, e.g. $(A+3)

(c) $X, where X is an integer from 0-255. This will produce a string containing one character, whose ASCII value is X. This is equivalent to CHR$(X) in other versions of BASIC. In ordinary ATOM BASIC, this facility was available only in PRINT statements - now it can be used elsewhere.

(d) A string array element such as MA$X(3) or MA$Q(1,2,3)
- see "multi-dimensional arrays" below.

(e) Any of the functions LEFT$, RIGHT$, MID$ defined below.

(f) Any string or expression enclosed within square brackets [ ]. This helps to make complicated expressions clearer, and is also used as discussed below for adding strings together.

The string functions available are:

LEFT$ {LEF.}

The form of this function is LEFT$ (string, X), where X is any integer or integer expression whose value does not exceed the length of the string. The function returns another string made up of the leftmost X characters of the original string.

E.g. $A = LEFT$ ("HELLO", 3) will place the string "HEL" in $A.

RIGHT$ (RI.)

This is similar to LEFT$, except that the right hand end of the original string is taken. E.g. $A = RIGHT$ ("HELLO", 2) puts the string "LO" in $A.

MID$ (M.)

The form of this function is MID$ (string,X,Y) where X and Y are integers or integer expressions. The function produces another string, Y characters long, starting at the Xth character of the original string. E.g. $A = MID$ ("HELLO", 2,3) puts the string "ELL" into $A. Taken together, the values of X and Y must not exceed the total length of the string.

INSTR (I.)

This function does not return a string, but an integer. The form is:

INSTR (string 1, string 2)

The integer returned represents the position of string 2 within the (larger) string 1 e.g: $A="HELLO" X=INSTR($A,"ELL") would put the value 2 into variable X. If string 2 is not found in string 1, then INSTR gives the value zero. If string 2 is null, then the value 1 is returned.

Concatenation (addition) of strings

Within square brackets, any number of strings may be added together end-to-end.

The form is:

[string 1 + string 2....]

For example:

$A = "BAN" $A = [$A + RIGHT$ ($A, 2) + $65] puts "BANANA into $A.

The maximum length of string which can be built up is 255. As mentioned previously, any of the items (a) to (f) above can be used as an argument in the above string functions. Thus, complicated string expressions, with string functions nested inside each other can be built up.

It should also be noted that any of items (a) to (f) except item (c) ($X, where X is 0 to 255) can be used in any of the following places:

Round brackets cannot be used around these string arguments, but square brackets can.

(i) In a string assignment, LET $A = string or $A = string

(ii) On the right hand side of an equals sign in a relational expression, e.g.

```
IF $A = string  THEN....
```

(iii) In a PRINT statement.

(iv) In an FPRINT statement (if the floating point ROM is fitted).

(v) As the argument to LEN

(vi) As the argument to CH

(vii) As the argument to VAL (provided the floating point ROM is fitted).

The string $X, where X is between 0 and 255, can be used in PRINT, FPRINT and LEN, but elsewhere it is necessary to enclose it within square brackets. Otherwise, it will be interpreted as meaning a string beginning at the appropriate address in page zero, and the result will be garbage.

MULTI-DIMENSIONAL ARRAYS

Ordinary Atom Basic supports one-dimensional arrays, of the form AA(X). Here, AA is the name of the array, and X is the subscript referring to a particular element. Each element can hold an integer. If you have the floating point ROM, a further type of one-dimensional array can hold a number of floating point elements, and each element takes the form %AA(X). There is no explicit form of string array.

SUPERBASIC supports multi-dimensional arrays, where each array name takes two or more subscripts in order to identify the required element. For example, a two-dimensional integer array takes the form MAI(X,Y). Here MA is a prefix which introduces all SUPERBASIC integer multi-dimensional arrays. I is a letter used to identify the array - arrays can be identified by any letter from A to Z or by the symbol @. X and Y are the subscripts for a particular element of the array, and as with one-dimensional arrays, these subscripts can in use take the form of integer variables, integer expressions, or simple numbers in order to address the required element of the array.

A two-dimensional integer array, or "Matrix", can be thought of as a table having lots of boxes, each of which can hold a number. Suppose the table has six columns, numbered from 0 to 5, and five rows, numbered from 0 to 4. You could call such an array MAQ, for example. The individual elements of the array can be accessed in the form MAQ(X,Y) where X can take any desired value between 0 and 5, and where Y can take any desired value between 0 and 4. One way of getting data into such an array would be as follows:

```
10  DIM  MAQ(5,4)
20  FOR Y = 0 TO 4
```

```
30  PRINT"INPUT TO ROW NO.",Y
40  FOR X = 0 TO 5
50  PRINT"COLUMN NO.",X
60  INPUT A
70  MAQ(X,Y) = A
80  NEXT X
90  NEXT Y
```

This would ask you for 30 numbers and would store each of them in the array.

The above program illustrates two points. First, the array has to be DIMensioned, like any other. If you want you can mix the dimensioning of several SUPERBASIC arrays, together with ordinary one-dimensional arrays and strings, in a single DIM statement - all the items to be dimensioned should be separated by commas as normal. Second, to put a number in a desired element of the array, you use an assignment statement of the usual form:

```
MAQ(X,Y) =  expression
```

LET may not be used.

Of course, elements of SUPERBASIC arrays may be used in any kind of expression. For example, you can do arithmetic, e.g.

```
A = 10*(MAQ(0,0) + 4)/MAQ(3,2)
```

Or you could print out all of row no. 3 of the matrix, say, with:

```
100  FOR N = 0 TO 5
110  PRINT MAQ(N,3)
120  NEXT N
```

SUPERBASIC's multi-dimensional arrays are not limited to two dimensions. You can DIMension an array to have as many dimensions as you wish, For example,

```
10 DIM  MAJ(2,2,2),  MAK(3,4,2,6)
```

will dimension a three-dimensional array MAJ and a four-dimensional array MAK. Two points should be noted. First, whenever you refer to an array, it must have the same number of subscripts, i.e. the same number of dimensions, as in the corresponding DIM statement. Second, multi-dimensional arrays eat memory space at an alarming rate.

The statement

```
10 DIM  MAX (10,10,10)
```

looks innocent enough, but it has $11^3=1331$ elements (remember that you have to start counting from zero) and therefore allocates over 5K of storage space.....if you have that much to spare!

SUPERBASIC doesn't just provide multi-dimensional arrays which hold integers. You can dimension and use a string array in just the same fashion. The form of a two-dimensional string array, for example, might be

```
MA$T(A,B)
```

Note that this is similar to an integer array except that the array name has the prefix MA$ instead of MA.

As with integer arrays, you can have string arrays of as many dimensions as you wish, provided you always have the same number of subscripts wherever the array is used as you have in the corresponding DIM statement. A one-dimensional string array of the form MA$A(X), for example, is perfectly possible. (One-dimensional integer SUPERBASIC arrays are also possible, but not really needed because ordinary Atom Basic already provides this facility.)

You should note two things about string arrays. Firstly, SUPERBASIC only allows a maximum of 27 multi-dimensional arrays, corresponding to the 27 identifying letters A to Z and @, whatever the type of array. So you cannot have both an integer array MAP(X,Y) and a string array MA$P(X), for example.

Secondly, when you DIMension a string array, SUPERBASIC will normally assume that you want each array element to be able to hold a string up to 10 characters long, and will reserve space accordingly. (Actually, it reserves 11 bytes per element, to allow for the carriage return at the end of each string.) Of course, you can put smaller strings in each element if you want - the extra space will just be wasted - but if you put a larger string into a particular element, it will overwrite another one. SUPERBASIC does not check for such overwriting - nor, for that matter, does it check that the values used for the subscripts when accessing array elements are within bounds. As with ordinary Atom Basic arrays, such checking is up to you.

If you wish to change the number of bytes which will be reserved for each element of a string array by the DIM statement, you should alter the contents of memory location ?#9F beforehand. SUPERBASIC sets up this location to hold the value 11, corresponding to 10 characters per string array element. But if you need a string array MA$G with elements 12 characters long, you can execute

```
?#9F = 13: DIM MAG(10)
```

This sets up space for eleven strings called MA$G(0) to MA$G(10), and each may hold up to 12 characters. Similarly, if you know that your string array will only ever have 5-character elements, you could save space by executing

```
?#9F = 6
```

before DIMensioning it. It follows from the above that putting

values of 0 or 1 into ?#9F is pointless; and that the maximum possible length for each element of a string array is 254,given by ?#9F = 255. If you have the floating point ROM, SUPERBASIC also supports multi-dimensional arrays of floating-point elements. The form is MA%F(X,Y,Z) (or however many dimensions you wish). In other words, the prefix MA is replaced by MA%. Otherwise, such arrays are dimensioned and used in a similar manner to integer multi-dimensional arrays. Note that the identifying letter - F in the above example - must not have been used to identify an integer or string array, because SUPERBASIC only supports a total of 27 multidimensional arrays, whatever their type.

You can dimension floating point SUPERBASIC arrays either in DIM or FDIM statements, whichever is most convenient. For that matter, integer and string arrays can also be dimensioned by FDIM instead of DIM.

A final word on SUPERBASIC's multi-dimensional arrays. After running a program which sets up arrays, you will still be able to refer to them in immediate mode, e.g. to print out their values for debugging purposes but they will be lost as soon as you edit your program and they could be corrupted if you do any string handling in immediate mode - string handling during programs is no problem of course.

READ..DATA..RESTORE

These statements provide a convenient way of incorporating data (numerical or character strings) in a program. A data pointer is initialised to the start of the current text space by the RESTORE command. Each READ statement searches though the program for the next DATA.

READ (REA. is possible, but doesn't save any space).

This command will take one or more of the following arguments (separated by commas if there are two or more):

- An integer variable e.g. X
- An integer array element, e.g. AA(3)
- An integer multi-dimensional array element e.g. MAB(1,2)
- A string variable, e.g. $X
- A multi-dimensional string array element, e.g. MA$B(1,2)
- A floating point variable, e.g. %X
- A floating array element, e.g. %XX(3)
- A multi-dimensional floating point. array element e.g. MA%B(1,2)
- A floating point indirection operator, e.g. %!#8200

Of course, the floating point ROM is needed to support floating point arguments.

DATA (DA.)

The Data statement cannot be on the same line as other statements, and a comma can be used straight after the keyword DATA to indicate that the spaces in front of the first string are significant. If an error occurs in a DATA statement, such as trying to read floating point or string data into an integer variable, then the error message (and any subsequent REPORT) will print out the DATA statement, not the READ statement. The line number of the READ statement can then be obtained by typing PRINT ERL.

Naturally, if a READ statement has a floating point argument, the DATA statement can have floating point data.

RESTORE (RES.)

This command resets the READ pointer to the first DATA statement.

WHILE {W.}

The statements following WHILE are executed repeatedly for as long as the condition specified is true. This loop differs from a DO...UNTIL in two respects:

(a) The condition is tested at the start of the loop, not at the end. Thus if the condition is false on entry, the entire loop will be skipped.

(b) The loop is repeated while the condition is true; the DO..UNTIL loop is executed while the condition is false.

WHILE can be followed by any testable integer expression (with the conjunctions AND and OR if desired). The loops can be nested to a maximum depth of 12.

ENDWHILE {ENDW.}

Used as a marker at the end of both WHILE and FWHILE loops. Please note WHILE..ENDWHILE is a structured loop, and the structure should be adhered to. Don't try jumping in and out of the loop.

FWHILE {FW.}

This works in the same way as WHILE, but will evaluate a floating point condition (the floating point ROM is needed). The relationship to WHILE is similar to the relationship between UNTIL and FUNTIL. WHILE and FWHILE loops can be nested inside each other, to a total maximum depth of 12 loops. Both sorts of loop require an ENDWHILE.

Example: FWHILE %X<1.234

Note that AND and OR cannot be used in FWHILE statements. When SUPERBASIC finds that the condition in a WHILE or FWHILE statement is false, it has to go scanning through the program to find the corresponding ENDWHILE. For this, it has to keep a count of any inner WHILE or FWHILE loops it finds, to be sure of getting the matching ENDWHILE. This is the reason why users must stick to the structure of the loops, and not jump in or out with GOTOs. It should be noted that the scanning routine will ignore occurrences of the keywords WHILE, FWHILE or ENDWHILE within quotation marks, so that you can still put what you like within quotes, but it will be fooled by the appearance of these keywords in REM statements if, (and only if) they follow immediately after a semi-colon and if an unmatched quotation mark occurs by accident, the scanning routine will ignore everything up to the end of the line - if in the process it ignores a keyword, you'll get loop nesting errors.

XIF(Extended IF) - THEN -ELSE - ENDIF
{X.} {omit} {EL.} {END.}

This is another feature "borrowed" from structured programming languages. It is used to select one out of two possible courses of action, depending on the result of a conmdition immediately following the word XIF. The condition can include AND and OR if desired. If the condition is true, then all the statements up to the word ELSE will be executed. Statements after ELSE will be skipped. Execution then resumes after the ENDIF statement. On the other hand, if the condition is false, all the statements up to ELSE are skipped, and those between ELSE and ENDIF are executed. The power of the structure stems from the fact that the keywords XIF, ELSE and ENDIF may be many lines apart - in fact as many as you like, as long as you keep them in the right order, XIF - ELSE - ENDIF. Note that you will have problems if you ignore the fact that this is a structured feature and use GOTO to jump into and out of XIF - ENDIF structures (or "across" the ELSE statements). The golden rule is that you should enter at the top, through XIF, and exit at the bottom, through ENDIF, at all times.

The keyword THEN is an optional extra, which simply acts to make the program clearer to read. As with ordinary IF statements, it is only essential if the statement following right after the XIF condition begins with T or ? or ! (In such situations, you could simply put in a semi-colon instead, if you want to save space.)

ELSE can also be omitted. In this case, all statements between XIF and ENDIF will be executed if the condition is true; otherwise they will all be skipped.

XIF - THEN - ELSE - ENDIF structures can be nested inside each other, as demonstrated by the next program. The maximum depth of nesting is 15. Note that inner loops must be completely within outer loops, and must not straddle the ELSE of an outer loop.

```
  5 REM A USELESS PROGRAM TO DEMONSTRATE XIF
 10 REM NOTE HOW INDENTING SHOWS UP THE STRUCTURE
 20 DO INPUT X
 30 XIF X<0 PRINT "NEGATIVE"'
 40 ELSE
 50   XIF X>500 THEN
 60         PRINT "THIS IS"
 70         XIF X<1000 PRINT"A BIT";ELSE PRINT"MUCH";ENDIF
 80         PRINT"TOO BIG"'
 90   ELSE XIF X%2=0
100             PRINT "EVEN"'
110         ELSE PRINT"ODD"'
120         ENDIF
130   ENDIF
140 ENDIF
150 UNTIL 0
```

This program also demonstrates various ways of setting out XIF - THEN - ELSE - ENDIF statements. Notice that even in the single-line structure of line 70, the ENDIF statement is essential. When XIF is skipping through a program searching for keywords, it uses the same routine as WHILE. So the comments on

p.10 are relevant.

XFIF {XF.} (Extended FIF)

This keyword is used in place of XIF, together with THEN - ELSE - ENDIF statements. It will evaluate a floating point condition instead of an integer one. AND and OR are not permitted. Otherwise, operation is the same as XIF. The maximum depth of nesting of XIF and XFIF structure is 15.

ON X GOTO {ONX G.} ON X GOSUB {ONX GOS.}

This statement allows you to select one of a number of possible destination for the GOTO or GOSUB statement, depending on the value of X. X may be replaced by any variable or factor (such as an array element) or by an expression in brackets.

The word GOTO or GOSUB should be followed by a list of possible line numbers, separated by commas. If X has the value 1, control will transfer to the line number specified in the first list item; if it has the value 2, control goes to the line specified in the second list item, and so forth. An error message is given if X is less than 1 or more than the number of items in the list. In the case of ON X GOSUB, the program is sent back to the next statement after the ON statement, when it gets to the RETURN in the subroutine.

Actually, the items in the list need not be explicit line numbers, but each can be anything which could follow GOTO or GOSUB in ordinary Atom Basic. (Each item must still be separated from the rest by commas.) In particular, it is possible to use one or more line labels instead of line numbers. Indeed, as in ordinary Atom Basic, if a line has been labelled, you must use the label. All line labels in the list are checked for validity, not just the one selected by the value of X, and error 157 is signalled if any label does not exist.

List items which are not labels are checked to see if they evaluate to a number, but the existence of a line having that number is only checked if that list item has been selected - if the line doesn't exist, you get error 127.

INKEY {INK.}

This command scans the keyboard to see if a key is currently being pressed. It does not wait if no key is pressed, making it useful for real-time control of animated graphics. Note that it is a statement, not a function which can be used in expressions. The keyword should be followed either by a variable from A to Z, or a string variable (of the usual form $ expression). In the case of a variable, the ASCII code of the key pressed is placed in that variable, or zero is placed in it if no key is being pressed at the time. In the case of a string variable, the character of the key pressed is placed in the string variable, or a null string is set up if no key is being pressed.

LOCK, COPY, REPT and the cursor control keys are ignored. SHIFT

and CTRL have their usual effects if pressed with another key. You can wait for a specific key to be pressed with a loop such as:

```
DO INKEY A; UNTIL A=CH"Y"
```

or:

```
DO INKEY $FREE; UNTIL $FREE = "Y"
```

BEEP (BE.)

BEEP is followed by two arguments controlling the pitch and duration respectively. Both arguments may be numbers, variables or expressions between 0 and 255 (if they are outside this range only the least significant byte will be used). The lower the value of x the higher the note will be; if x is less than 8 no note will be heard-this can be used to give a programmable delay. The duration can lie between about 20 milliseconds (y=1) and six seconds (y=255). Space invader sound effects can be produced with the BEEP command e.g.:

```
100 FOR J=1 TO 5
110 FOR K=40 TO 80 STEP 4
120 BEEP K,1
130 NEXT;NEXT;END
```

Music can be produced by first reading the pitches of the notes of the scale into an array (using READ and DATA statements, see below), and then specifying the desired array element as the pitch in a series of BEEP statements. Alternatively, the durations and the pitches of the notes of a tune can be placed in a series of DATA statements, and you can then READ them as you play the tune. The note values are given in the table below:

NOTE	VALUE	VALUE	VALUE	NOTE	VALUE	VALUE	VALUE
A	246	121	60	D#	173	86	42
Bb	231	114	57	E	163	81	39
B	217	108	54	F	154	76	37
C	205	102	50	F#	145	72	35
C#	194	96	47	G	137	68	33
D	183	91	45	G#	129	64	31

STOP (ST.) (no abbreviation possible if floating point ROM fitted.)

If a program is misbehaving, you can insert STOP commands at several points throughout the program. When the STOP is reached the computer prints STOP AT, followed by the line number, and then waits for a key to be pressed before continuing. If ESCAPE is pressed control returns to the BASIC monitor. Once the program has been debugged, the STOP statements can be removed.

RERUN (RER.)

The command RUN (R.) can only be used in immediate mode, in a line of its own, in order to start a program off. In other situations, e.g. within a program to restart it, it is necessary

to use the statement RERUN to ensure that the system is set up properly. It can also be used in immediate mode in a multi-statement line, such as: X=0;RERUN

ZERO (Z.)

This statement zeroes all the integer variables A to Z. In addition, the three most significant bytes of variable @ are zeroed, but the least significant byte is left untouched, so that the effect of this variable on the PRINT format is unchanged. ZERO% (Z.%) will zero all the floating point variable %@ and %A to %Z

RESET (no abbreviation possible)

This statement makes SUPERBASIC "forget" all references to current subroutines and current FOR-NEXT, DO-UNTIL, DO-FUNTIL, WHILE, FWHILE, XIF and XFIF loops that it might be in the middle of. So you can then safely jump out of such subroutines and loops and back into the main program. But you must not jump into any other loop or subroutine, because that one will have been forgotten too. The chief use of this statement is when handling errors with ON ERROR (see below), if the possibility exists that an error might occur in the middle of a loop or subroutine. Note that RESET will also make SUPERBASIC "forget" about any ON ERROR statement which is active, so that subsequent errors will be reported as normal.

ERR

This is a function which can be used in expressions, conditions, etc. It returns the number of the last error to have occurred. If no error has occurred since you enabled SUPERBASIC, it returns an unpredictable number. See ON ERROR below for an example of its use.

ERL

This is another function, like ERR. It returns the number of the line in which the last error occurred. If the error occurred in immediate mode rather than in a program, the value zero is returned; as with ERR, an unpredictable number is returned if there has not yet been any error. Again, see ON ERROR below for an example.

REPORT (REP.)

This statement prints out the last error message, complete with the printout of the erroneous line and the usual beep. If used in a program, it stops it running first,and returns to immediate mode. It is illegal to use REPORT if the last error occurred in immediate mode, or if there has not yet been any error - in such a case, therefore, REPORT will appear to "report" itself. Note that if the last error was a SUPERBASIC error, the usual message to that effect will be missed out. The line printed out by REPORT will not be correct if you have edited your program since the error occured.

Error handling

ON ERROR (ONE.)

When this statement is encountered during the running of a program, SUPERBASIC saves details of its position. It then skips the rest of the line, including any multiple statements on the line. At the time, there is no noticeable effect of all this having happened. However, if an error occurs later on during the run of the program, the normal error messages will be suppressed.
Instead, SUPERBASIC will jump directly to the position it has saved, immediately after the words ON ERROR, and execute the statements which previously were skipped over. Normally, this will be a GOTO statement, which will point to a routine you have written to deal with the error condition in some suitable way. All references to loops and subroutines which were active when the error occurred will be retained intact. This means that your error handling routine can jump back to where the error occurred, using GOTO ERL. But you must ensure that the problem has been properly dealt with, and will not simply happen again - if it does, then control will immediately come back to your error handling routine and then go round and round in an endless loop. If your error handler is likely to pick up errors which occurred in loops or subroutines, and then send control back to the main program, outside all loops and subroutines, then you should first cancel the loops/subroutines with RESET. Don't forget that this will also cancel the ON ERROR so that future errors will not be picked up. If that's not what you want, your error handler can contain a further ON ERROR statement right after the RESET. The following example shows how to use ON ERROR in conjunction with ERR, ERL and REPORT.

```
  10 REM ERROR HANDLING EXAMPLE
  20 ON ERROR GOTO 1000
  30 INPUT"TYPE A NUMBER",A
  40 REM CONTINUE WITH YOUR PROGRAM
  50
 100 INPUT"TYPE ANOTHER NUMBER",B
 110 REM CONTINUE WITH YOUR PROGRAM
 120
 500 END
 600********************************
 999 REM TRAP ERRORS IN INPUT
1000 IF ERL<>30 AND ERL<>100 THEN REPORT
1010 IF ERR=109 PRINT"SORRY,THAT'S TOO  BIG"'; GOTO ERL
1020 IF ERR=29 PRINT"PLEASE USE ONLY DIGITS 0-9"';GOTO ERL
1030 REPORT
```

If desired, your program can contain several ON ERROR statements, to handle errors in different part of the program in different ways. When SUPERBASIC comes to a new ON ERROR statement, it saves its position and cancels the previous one. You can cancel ON ERROR entirely by executing ON ERROR REPORT

(or using RESET).

FREE (FR.)

This is a "pseudo-variable" which gives you access to the "Free space pointer" which is used by DIM and FDIM when reserving space for strings, arrays and multi-dimensional arrays. See Atomic Theory and Practice, page 145, for an explanation of how DIM increments the free space pointer, now called FREE. "Pseudo-variable" means that you can use FREE both in expressions and in statements such as PRINT &FREE; and also you can change its value by having an assignment statement of the form FREE = <expression>. (LET is not permitted). For example, a statement like FREE = FREE + 256 is quite valid. A more useful statement might be FREE =#8200. This would mean that any arrays and strings dimensioned afterwards would be stored in the graphics text space - useful if space is short above your program in the lower text space and DIM statements are giving error 30. In SUPERBASIC, the commands RUN and RERUN have the effect of setting FREE to the value TOP + 54. This reserves 54 bytes which are used as workspace by the multi-dimensional arrays - over and above the storage space which is reserved by increasing the value of FREE when you actually DIMension such an array. The requirement for this 54 bytes above TOP is not affected by changing the value of FREE, though if you do not have the full 54 bytes your program will run normally until you try to DIMension a multi-dimensional array (when you will get SUPERBASIC error 255). Such errors are more likely with multi-dimensional arrays having a "high" identifying letter like X, Y or Z than with a "low" identifying letter like @ or A. SUPERBASIC string handling functions also make use of space above FREE. Since they only use it for temporary scratch-pad workspace, they do not change the value of FREE. You too can use the space above FREE temporarily, with expressions like ?FREE, !FREE and $FREE, as long as you bear in mind that string handling will destroy whatever you have stored. In fact, you should use FREE for such storage in preference to TOP, which would interfere with the multi-dimensional arrays. For another use of the pseudo-variable FREE, see point 6 of "Points to note when using SUPERBASIC".

During immediate mode, you cannot change the value of FREE, because it will immediately be set back to TOP + 54. This is done so that FREE gradually increases when you type in a program - otherwise you could corrupt your newly-typed program if you broke off to do some string handling.

PRINT TAB and FPRINT TAB

The keyword TAB can be used anywhere in a PRINT or FPRINT statement. It can take either one or two arguments, in the form TAB(X) or TAB(X,Y). Here, X and Y may each be any integer number, variable or expression. In the form TAB(X), X may be anything between zero and 255, and the next item to be printed will start at column X on the screen and/or on the printer. Note that the leftmost column is numbered zero. On the screen, if X is greater than 31, there will be a "wrap-around" effect

onto the next line. TAB(X) works by issuing spaces until the value of the Basic function COUNT is equal to or greater than X, and it will work on any printer which is connected up and turned on, since the spaces will also be sent to the printer. Note, however, that if COUNT already equals or exceeds X, then TAB(X) will have no effect whatsover, and the next PRINT item will start from where the previous one finished. Note also that COUNT religiously counts up any control codes which you print (e.g. PRINT $2 to turn on the printer) so unless you then start a new line with PRINT', the next print item could start one position earlier than you expected, for each control code printed.

TAB(X,Y) works rather differently. This sends the cursor on the screen directly to column X in row Y, and has no effect on your printer. X may be any number between 0 and 31, while Y may be any number between 0 and 15. Y = 0 gives the top row of the screen, and Y = 15 gives the bottom row. If X and/or Y are greater than 31 and 15 respectively, then only the lowest significant 5 or 4 bits respectively of the value will be used. For example, PRINT TAB(33,17) has the same effect as PRINT TAB (1,1).

Whichever form of TAB you use, numbers will still be printed in a field the size of which depends on variable @. As a result, there could be some leading spaces before the printing of the number starts. This can be very useful when formatting tables, for example, but if it is not what you want, set @ = 0 or @ = 1 before printing.

b (produced by typing SHIFT-B; appears as inverted B on the screen).

The letter b is used to introduce a number written in binary (base 2), containing only the digits 0 and 1. It can be compared to the symbol #, used to introduce a number written in hexadecimal(base 16). It must be followed by at least one zero or one, and can be followed by up to 32 binary digits (zero or one). b1111, for example, is the same as #F, or 15 decimal. The binary number after b will be read right up to the first character which is not 0 or 1. If an illegal character appears, the computer will usually try to interpret it as an arithmetic operator which is intended to act on the binary number - if it fails in this, you'll normally get an ordinary Atom Basic error message such as error 174, or error 94 (illegal character after valid statement). The chief use of b is when defining black and white graphics shapes - see below.

f (produced by typing SHIFT-F; appears as inverted F on the screen.)

f is used to introduce a number written in base 4, containing any of the digits 0,1,2 or 3. It can be followed by up to sixteen such digits. It is very similar to b in operation. The chief use of f is when defining colour graphics shapes - see below.

User defined graphics shapes

SUPERBASIC allows you to define up to 27 different shapes, identified by the letters A to Z and @ (quite separate from multi-dimensional arrays having such identifying letters). Such shapes may be defined for any of the black and white graphics modes 1 to 4, or colour modes 1a to 4a, but not for mode 0.

In black and white modes, user defined shapes are 8 pixels wide by 8 pixels high, and once defined can be used in any of modes 1 to 4 (though the proportions will be a bit different from mode to mode). In the colour modes 1a to 4a, shapes are 4 pixels wide by 8 pixels high, though in modes 1a and 4a at least, the pixels are twice the width than in the corresponding modes 1 and 4, so it does not follow that the overall shape will be any narrower. But in any case, if you require a larger shape, you can simply define it in terms of two or more component shapes, and insert the component shapes side-by-side on the screen and move them around altogether.

User defined shapes require workspace below #2900, but above the floating point variables and arrays. So they will only work properly if you have at least 1K of memory in the lower text space. In addition to this workspace requirement, details of the shapes you have defined are stored above your program, in exactly the same way that arrays and strings are stored by DIM (i.e. by increasing the value of FREE).

When you are defining a shape, think of it as split into 8 rows, each four pixels wide (colour) or 8 pixels wide (black and white). So defining the shape needs 8 numbers, one per row, starting at the top.
Each number can be in the range 0 to 255. You could write these numbers in decimal, or in hexadecimal, or even as variables or expressions. But it is most convenient, for shapes intended for use in black and white modes, to use binary numbers (see above).

Each binary number should have 8 digits, with ones corresponding to white pixels in the resulting row of the shape, and zeroes corresponding to dark pixels (background).

To define shapes for use in colour modes, the numbers are most conveniently written in base four (see above). Each number should then have four digits, and the possible digits 0,1,2 and 3 correspond to the numbers used to set colours in the COLOUR statement in the floating point ROM. 0 is the background colour. Having defined your shapes as desired (and as mentioned above, there can be as many as 27 of them) you must then use an appropriate CLEAR statement to get the graphics mode required, followed by a COLOUR statement if you want a colour mode. A number from 0 to 3 is still needed after the COLOUR statement, but the precise value chosen does not affect the user defined shapes, but only PLOT, DRAW and MOVE. If you have a colour board

but no floating point ROM, you can still use the user-defined shapes (though not PLOT, DRAW and MOVE) as follows. First execute an appropriate CLEAR statement (CLEAR 1 to CLEAR 4). Then execute

```
?#B000=?#B000 & DF.
```

Finally, for mode 2a (but not 1a, 3a or 4a) execute

```
FOR N #8600 TO #87FF STEP 4; !N=0; NEXT N.
```

DEFSHP (DE.)

This is the statement used to define a shape (in any mode). In its basic form, it is followed by the identifying letter (@ to Z) for the shape, followed by a list of the eight numbers discussed above. The numbers should be separated from each other and from the identifying letter by commas.

However, if you are using binary or base 4 numbers as recommended, you will have difficulty fitting all eight numbers on one line. So you can break off in the middle of the definition (anywhere after the identifying letter, but do not finish with a comma), and SUPERBASIC will store details of what you have defined so far. You can then have one or more continuation DEFSHP statements to complete the definition.

The continuation statement is distinguished from the basic statement by the fact that it has an apostrophe(') after the keyword DEFSHP, and does not contain the identifying letter - that is remembered from the previous statement. The continuation DEFSHP' statement just contains one or more further numbers from the list, separated by commas. If the total number of numbers listed in the basic DEFSHP and any continuations exceeds 8, an error message is given. If you start to define a new shape before finishing the old one, the remainder of the old shape will be filled with garbage.

The following example defines a stick man shape for use in any black and white mode. Notice from the sketch of the man, on the right, how the ones in the definition are arranged.

```
10 DEFSHP  M, b00111100
20 DEFSHP'    b00011001
30 DEFSHP'    b11111111
40 DEFSHP'    b10011000
50 DEFSHP'    b00011000
60 DEFSHP'    b00111100
70 DEFSHP'    b01100110
80 DEFSHP'    b11000011
```

INVSHP (INV.)

This statement performs an inversion operation at a point on the screen, using a pre-defined shape. "Inversion" means that the shape is inserted if previously there was nothing there, and deleted if it had been previously inserted at that position.

Partial overwriting of an existing shape will look a little odd, but this will only last until one or other shape is deleted with an appropriate INVSHP.

The form is INVSHP A(X,Y) Here, A is the identifying letter for the desired shape, and X and Y represent any integer expressions. These are evaluated, and the shape is inserted or deleted with its top left-hand corner at pixel position (X,Y). That is, the values of X and Y correspond with the same coordinate system used by PLOT, DRAW and MOVE, in which position (0,0) is at the bottom left-hand corner, and (in mode 4) position (255, 191) is at the top right-hand corner of the screen.

Note that if necessary, the shape can be "inserted" or "deleted" partially or wholly off the screen.

It is possible to insert or delete two or more shapes at once with INVSHP. Suppose you had defined the left-hand side of a flying saucer as shape L, and the right-hand side as shape R. You could use:

```
INVSHP L(X,Y),R(X+D,Y)
```

In order to get the two components of the shape side-by-side, the value of D should be 8 in black and white modes, or 4 in colour modes.

MOVSHP (M.)

This statement is used to perform two INVSHP operations in quick succession, using the same predefined shape. Used properly, the effect is to move a shape from one position to another, by deleting it from the first position and inserting it at the second. The form is MOVSHP A(X,Y,V,W) This can be used to move shape A from position (X,Y) to position (V,W), or vice versa (depending what was already on the screen). As with INVSHP, the value of the coordinates refer to the top left-hand corner of the shape. Also as with INVSHP, a statement such as MOVSHP A(X,Y,V,W), B(X+D,Y,V+D,W) can be used to move two (or more) shapes as one body.

The following example program defines an invader with a body one colour, eyes a second and legs a third. It then moves it repeatedly across the screen, gradually moving down and getting faster.

```
 10 REM EXAMPLE OF USER DEFINED SHAPES
 20 REM DEFINE LEFT SIDE OF INVADER
 30 DEFSHP I,f0011,f0111,f1121,f1121
 40 DEFSHP' f0111,f0033,f0330,f3300
 50 REM DEFINE RIGHT SIDE
 60 DEFSHP J,f1100,f1110,f1211,f1211
 70 DEFSHP' f1110,f3300,f0330,f0033
200 CLEAR 4; COLOUR 0; REM COLOUR NO. UNIMPORTANT
220 S=0; REM S SETS SPEED
230 FOR Y=180 TO 20 STEP -10
240 S=S+1; REM GET FASTER
249 REM INSERT IT
250 INVSHP I(10,Y), J(14,Y)
299 REM MOVE IT
300 FOR X=10 TO 110 STEP S
310 MOVSHP I(X,Y,X+S,Y), J(X+4,Y,X+S+4,Y)
330 NEXT X
349 REM DELETE IT
350 INVSHP I(X,Y), J(X+4, Y)
400 NEXT Y
999 END
```

Both INVSHP and MOVSHP perform a WAIT before accessing the screen. This much reduces or avoids flicker on the screen, but obviously slows things down rather. If speed is important, and flicker less so, you can suppress the WAIT in both INVSHP and MOVSHP. This is done by inserting an apostrophe (') after the identifying letter and before the bracket,

e.g. INVSHP A' (X,Y)

If there is more than one shape being manipulated, then each needs an ' in this manner (unless you want one shape to WAIT and another not to).

Points to note when using SUPERBASIC

1. If you have to press BREAK, it will be necessary to re-enable SUPERBASIC with LINK #AF00 or LINK #AF04. Don't forget to type OLD to retrieve your program. If you use any of SUPERBASIC's string handling facilities before you type OLD, your program could be corrupted.

2. Although existing ATOM BASIC programs normally give no problems to SUPERBASIC, this is not necessarily the case with other commercial software - particularly if such software uses the same memory locations as SUPERBASIC, or if it alters the operating system vectors (located above 200). Before running such software, you are advised to disable SUPERBASIC by pressing BREAK.

3. If you fit SUPERBASIC in a ROM selector board, disable it by pressing BREAK before switching to another ROM.

4. The command RUN should only be used in a line of its own in immediate mode - that is, the way in which it is normally used in order to run a program. It should not be used within programs, nor in immediate mode commands such as X=0; RUN.

   In these (relatively unusual) situations, the command RERUN should be used instead. (The command X=0; RUN in immediate mode can be broken up into two separate lines instead of using RERUN.) Similarly, non-standard ways of starting off programs, such as typing GOTO 10 in immediate mode, should be avoided - they may appear to work, but will not initialise the system properly, giving seemingly inexplicable problems later on.

5. The various SUPERBASIC floating point facilities will only work if you have the Acorn floating point ROM fitted. If it is not fitted, an appropriate error message will be given to show that the SUPERBASIC floating point statement or function has not been recognised.

6. When writing SUPERBASIC programs which use the ? and ! operators (or adapting existing programs) make sure that you are not using an area of memory which SUPERBASIC uses - see below. For similar reasons, it is wise not to use the variable TOP to store numbers or strings, e.g. by executing $TOP = "STRING". If you are not using SUPERBASIC'S string handling functions, you could use $FREE instead, but it is much safer to DIMension a suitable memory area in the proper manner.

   Similarly, the use of DIM P(-1) to allocate space for assembly of machine code is dangerous. (because it doesn't increment FREE). Use one of the following instead:

Either:

```
10  DIM  Q(100);  REM  Reserve  enough space
20  FOR N=1 TO 2; REM Two passes
30  P=Q
40 [Your code here;]
50  NEXT N
```

Or:

```
10 FOR N=1 TO 2; REM Two passes
20 P = FREE; REM Start address
30 [Your code here;]
40 NEXT N
50 FREE = P; REM End address plus one
```

SUPERBASIC memory usage

SUPERBASIC uses memory in the following areas. Users should not generally access these areas.

Area	Purpose
#57-#AF	General storage and workspace
#21C-#23F	WHILE/FWHILE stack
#28BD-#28F2	User defined shape workspace
TOP to TOP+54	Multi-dimensional array workspace (see FREE for more details).
above FREE	Temporary scratch pad for string handling. (See FREE for more details).

ERROR Messages

All the usual Atom error codes can occur with SUPERBASIC fitted. But in addition, the following codes may arise. They can be distinguished from the Acorn error codes by the fact that they are preceded by the message "SUPERBASIC" (except when produced by REPORT)

Error Code	Likely Causes
2,3 or 4	Search for ENDIF failed after XIF, XFIF or ELSE
6,7 or 8	Search for ENDWHILE failed after WHILE or FWHILE
10	No DATA left for a READ statement to read
19	Insufficient space above FREE for use by DEFSHP
35 to 41	Insufficient room above FREE for string handling
128	Bad syntax in INKEY
129	Bad syntax in BEEP
130	XIF statement takes number of nested XIF/XFIF loops over 15. Bad nesting of XIF loops.
131	XFIF statement error-similar to 130.
132	Bad nesting of ELSE. Jumping into an XIF/XFIF loop with GOTO ELSE without XIF or DFIF.
133	Bad nesting of ENDIF. Jumping into an XIF/XFIF loop with GOTO. ENDIF without XIF or XFIF.
134	Too many WHILE loops. Bad nesting of WHILE/ENDWHILE
135	FWHILE error - as 134
136	ENDWHILE with no corresponding WHILE or FWHILE
138	READ error - either bad syntax in READ or some problem in the corresponding DATA statement.
140	Attempt to use STOP in immediate mode.
141	MOVSHP error - either bad syntax or corresponding shape not yet defined.
142	Error in MA% assignment, e.g. bad syntax, array not yet dimensioned, or dimensional with a different number of subscripts.
143	Error in MA$ assignment - similar to 142.
144	Error in MA assignment - similar to 142.

145 INVSHP error - similar to 141

146 Bad syntax in ON ERROR or ON X GOTO/GOSUB. Value of X too large or small.

147 Bad syntax or too many arguments in DEFSHP or DEFSHP'

148 Illegal use or REPORT

149 Attempt to use RERUN with no program in memory - try typing OLD if necessary.

155 Error in MA% evaluation, e.g. bad syntax, array not yet dimensioned or dimensioned with a different number of subscripts.

158 Error in MA evaluation - similar to 155

159 No binary number after b; or binary number too large (Maximum 32 digits)

160 No base 4 number after f; or base four number too large (max. 16 digits)

161 Bad syntax in INSTR

163 MID$ syntax error, or number arguments longer than string.

165 Unmatched quotes in string expression.

166 LEFT$ syntax error or number argument longer than string

167 RIGHT$ syntax error, or number argument longer than string.

168 Error in MA$ evaluation - similar to 155

169 Unexpected character within square brackets during string handling. Attempt to add together strings which between them have over 255 characters.

170 Bad syntax in TAB

171 Attempt to use RUN with no program in memory - try typing OLD if necessary.

255 Error during DIM or FDIM of multi-dimensional array, e.g. bad syntax, or insufficient memory above TOP for the 54 bytes of workspace. Note that insufficient memory above FREE will cause the ordinary Atom error 30 (or its FDIM equivalent) rather than SUPERBASIC error 255.

(c) Micro Power 1983SUPERBASIC manual