ROM BASIC INTERPRETER

Microsoft BASIC has evolved over the years to its present position as the industry standard. It was originally written for the 8080 Microprocessor and even the MSX version is held in 8080 Assembly Language form. This process of continuous development means that there are less Z80-specific instructions than would be expected in a more modern program. It also means that numerous changes have been made and the result is a rather convoluted program. The structure of the Interpreter makes it unlikely that an application program will be able to use its many facilities. However most programs will need to cooperate with it to some extent so this chapter gives a detailed description of its operation. There are four readily identifiable areas of importance within the Interpreter, the one most familiar to any user is the Mainloop (4134H). This collects numbered lines of text from the console and places them in order in the Program Text Area of memory until a direct statement is received. The Runloop (4601H) is responsible for the execution of a program. It examines the first token of each program line and calls the appropriate routine to process the remainder of the statement. This continues until no more program text remains, control then returns to the Mainloop. The analysis of numeric or string operands within a statement is performed by the Expression Evaluator (4C64H). Each expression is composed of factors, in turn analyzed by the Factor Evaluator (4DC7H), which are linked together by dyadic infix operators. As there are several types of operand, notably line numbers, which cannot form part of an expression in Microsoft BASIC the term “evaluated” is only used to refer to those that can. Otherwise a term such as “computed” will be used. One point to note when examining the Interpreter in detail is that it contains a lot of trick code. The writers seem particularly fond of jumping into the middle of instructions to provide multiple entry points to a routine. As an example take the instruction:

```
3E D1       Normal: LD   A,0D1H
```

When encountered in the usual way this will of course load the accumulator with the value D1H. However if it is entered at “Normal” then it will be executed as a POP DE instruction. The Interpreter has many similarly obscure sections.

Address... 268CH
This routine is used by the Expression Evaluator to subtract two double precision operands. The first operand is contained in DAC and the second in ARG, the result is returned in DAC. The second operand’s mantissa sign is inverted and control drops into the addition routine.

Address... 269AH
This routine is used by the Expression Evaluator to add two double precision operands. The first operand is contained in DAC and the second in ARG, the result is returned in DAC. If the second operand is zero the routine terminates with no action, if the first operand is zero the second operand is copied to DAC (2F05H) and the routine terminates. The two exponents are compared, if they differ by more than 10^15 the routine terminates with the larger operand as the result. Otherwise the difference between the two exponents is used to align the mantissae by shifting the smaller one rightwards (27A3H), for example:

```
19.2100 = .1921*10^2 = .192100
  .7436 = .7436*10^0 = .007436
```

If the two mantissa signs are equal the mantissae are then added (2759H), if they are different the mantissae are subtracted (276BH). The exponent of the result is simply the larger of the two original exponents. If an overflow was produced by addition the result mantissa is shifted right one digit (27DBH) and the exponent incremented. If leading zeroes were produced by subtraction the result mantissa is renormalized by shifting left (2797H). The guard byte is then examined and the result rounded up if the fifteenth digit is equal to or greater than five.

Address... 2759H
This routine adds the two double precision mantissae contained in DAC and ARG and returns the result in DAC. Addition commences at the least significant positions, DAC+7 and ARG+7, and proceeds two digits at a time for the seven bytes.

Address... 276BH
This routine subtracts the two double precision mantissae contained in DAC and ARG and returns the result in DAC. Subtraction commences at the guard bytes, DAC+8 and ARG+8, and proceeds two digits at a time for the eight bytes. If the result underflows it is corrected by subtracting it from zero and inverting the mantissa sign, for example:

```
0.17-0.85 = 0.32 = -0.68
```

Address... 2797H
This routine shifts the double precision mantissa contained in DAC one digit left.

Address... 27A3H
This routine shifts a double precision mantissa right. The number of digits to shift is supplied in register A, the address of the mantissa's most significant byte is supplied in register pair HL. The digit count is first divided by two to separate the byte and digit counts. The required number of whole bytes are then shifted right and the most significant bytes zeroed. If an odd number of digits was specified the mantissa is then shifted a further one digit right.

Address... 27E6H
This routine is used by the Expression Evaluator to multiply two double precision operands. The first operand is contained in DAC and the second in ARG, the result is returned in DAC. If either operand is zero the routine terminates with a zero result (2E7DH). Otherwise the two exponents are added to produce the result exponent. If this is smaller than 10^-63 the routine terminates with a zero result, if it is greater than 10^63 an "Overflow error" is generated (4067H). The two mantissa signs are then processed to yield the sign of the result, if they are the same the result is positive, if they differ it is negative. Even though the mantissae are in BCD format they are multiplied using the normal binary add and shift method. To accomplish this the first operand is successively multiplied by two (288AH) to produce the constants X*80, X*40, X*20, X*10, X*8, X*4, X*2, and X in the HOLD8 buffer. The second operand remains in ARG and DAC is zeroed to function as the product accumulator. Multiplication proceeds by taking successive pairs of digits from the second operand starting with the least significant pair. For each 1 bit in the digit pair the appropriate multiple of the first operand is added to the product. As an example the single multiplication 1823*96 would produce:

```
1823*10010110=(1823*80)+(1823*10)+(1823*4)+(1823*2)
```

As each digit pair is completed the product is shifted two digits right. When all seven digit pairs have been processed the routine terminates by renormalizing and rounding up the product (26FAH). The time required for a multiplication depends largely upon the number of 1 bits in the second operand. The worst case, when all the digits are sevens, can take up to 11 ms compared to the average of approximately 7 ms.

Address... 288AH
This routine doubles a double precision mantissa three successive times to produce the products X*2, X*4 and X*8. The address of the mantissa's least significant byte is supplied in register pair DE. The products are stored at successively lower addresses commencing immediately below the operand.

Address... 289FH
This routine is used by the Expression Evaluator to divide two double precision operands. The first operand is contained in DAC and the second in ARG, the result is returned in DAC. If the first operand is zero the routine terminates with a zero result if the second operand is zero a "Division by zero" error is generated (4058H). Otherwise the two exponents are subtracted to produce the result exponent and the two mantissa signs processed to yield the sign of the result. If they are the same the result is positive, if they differ it is negative. The mantissae are divided using the normal long division method. The second operand is repeatedly subtracted from the first until underflow to produce a single digit of the result. The second operand is then added back to restore the remainder (2761H), the digit is stored in HOLD and the first operand is shifted one digit left. When the first operand has been completely shifted out the result is copied from HOLD to DAC then renormalized and rounded up (2883H). The time required for a division reaches a maximum of approximately 25 ms when the first operand is composed largely of nines and the second operand of ones. This will require the greatest number of subtractions.

Address... 2993H
This routine is used by the Factor Evaluator to apply the "COS" function to a double precision operand contained in DAC. The operand is first multiplied (2C3BH) by 1/(2*PI) so that unity corresponds to a complete 360 degree cycle. The operand then has 0.25 (90 degrees) subtracted (2C32H), its mantissa sign is inverted (2E8DH) and control drops into the "SIN" routine.

Address... 29ACH
This routine is used by the Factor Evaluator to apply the "SIN" function to a double precision operand contained in DAC. The operand is first multiplied (2C3BH) by 1/(2*PI) so that unity corresponds to a complete 360 degree cycle. As the function is periodic only the fractional part of the operand is now required. This is extracted by pushing the operand (2CCCH) obtaining the integer part (30CFH) and copying it to ARG (2C4DH), popping the whole operand to DAC (2CE1H) and then subtracting the integer part (268CH). The first digit of the mantissa is then examined to determine the operand's quadrant. If it is in the first quadrant it is unchanged. If it is in the second quadrant it is subtracted from 0.5 (180 degrees) to reflect it about the Y axis. If it is in the third quadrant it is subtracted from 0.5 (180 degrees) to reflect it about the X axis. If it is in the fourth quadrant 1.0 (360 degrees) is subtracted to reflect it about

both axes. The function is then computed by polynomial approximation (2C88H) using the list of coefficients at 2DEFH. These are the first eight terms in the Taylor series X-(X^3/3!)+(X^5/5!)-(X^7/7!) ... with the coefficients multiplied by successive factors of 2*PI to compensate for the initial scaling.

Address... 29FBH
This routine is used by the Factor Evaluator to apply the "TAN" function to a double precision operand contained in DAC. The function is computed using the trigonometric identity TAN(X) = SIN(X)/COS(X).

Address... 2A14H
This routine is used by the Factor Evaluator to apply the "ATN" function to a double precision operand contained in DAC. The function is computed by polynomial approximation (2C88H) using the list of coefficients at 2E30H. These are the first eight terms in the Taylor series X-(x^3/3)+(X^5/5)-(X^7/7) ... with the coefficients modified slightly to telescope the series.

Address... 2A72H
This routine is used by the Factor Evaluator to apply the "LOG" function to a double precision operand contained in DAC. The function is computed by polynomial approximation using the list of coefficients at 2DA5H.

Address... 2AFFH
This routine is used by the Factor Evaluator to apply the "SQR" function to a double precision operand contained in DAC. The function is computed using the Newton-Raphson process, an equivalent BASIC program is:

```
10 INPUT"NUMBER";X
20 GUESS=10
30 FOR N=1 To 7
40 GUESS=(GUESS+X/GUESS)/2
50 NEXT N
60 PRINT GUESS
70 PRINT SQR(X)
```

The above program uses a fixed initial guess. While this is accurate over a limited range maximum accuracy will only be attained if the initial guess is near the root. The method used by the ROM is to halve the exponent, with rounding up, and then to divide the first two digits of the operand by four and increment the first digit.

Address... 2B4AH
This routine is used by the Factor Evaluator to apply the "EXP" function to a double precision operand contained in DAC. The operand is first multiplied by 0.4342944819, which is LOG(e) to Base 10, so that the problem becomes computing 10^X rather than e^X. This results in considerable simplification as the integer part can be dealt with easily. The function is then computed by polynomial approximation using the list of coefficients at 2D6BH.

Address... 2BDFH
This routine is used by the Factor Evaluator to apply the "RND" function to a double precision operand contained in DAC. If the operand is zero the current random number is copied to DAC from RNDX and the routine terminates. If the operand is negative it is copied to RNDX to set the current random number. The new random number is produced by copying RNDX to HOLD, the constant at 2CF9H to ARG, the constant at 2CF1H to DAC and then multiplying (282EH). The fourteen least significant digits of the double length product are copied to RNDX to form the mantissa of the new random number. The exponent byte in DAC is set to 10^0 to return a value in the range 0 to 1.

Address... 2C24H
This routine is used by the "NEW", "CLEAR" and "RUN" statement handlers to initialize RNDX with the constant at 2D01H.

Address... 2C2CH
This routine adds the constant whose address is supplied in register pair HL to the double precision operand contained in DAC.

Address... 2C32H
This routine subtracts the constant whose address is supplied in register pair HL from the double precision operand contained in DAC.

Address... 2C3BH
This routine multiplies the double precision operand contained in DAC by the constant whose address is supplied in register pair HL.

Address... 2C41H
This routine divides the double precision operand contained in DAC by the constant whose address is supplied in register pair HL.

Address... 2C47H
This routine performs the relation operation on the double precision operand contained in DAC and the constant whose address is supplied in register pair HL.

Address... 2C4DH
This routine copies an eight byte double precision operand from DAC to ARG.

Address... 2C59H
This routine copies an eight byte double precision operand from ARG to DAC.

Address... 2C6FH
This routine exchanges the eight bytes in DAC with the eight bytes currently on the bottom of the Z80 stack.

Address... 2C80H
This routine inverts the mantissa sign of the operand contained in DAC (2E8DH). The same address is then pushed onto the stack to restore the sign when the caller terminates.

Address... 2C88H
This routine generates an odd series based on the double precision operand contained in DAC. The series is of the form:

```
X^1*(Kn)+X^3*(Kn-1)+x^5*(Kn-2)+X^5*(Kn-3) ...
```

The address of the coefficient list is supplied in register pair HL. The first byte of the list contains the coefficient count, the double precision coefficients follow with K1 first and Kn last. The even series is generated (2C9AH) and multiplied (27E6H) by the original operand.

Address... 2C9AH
This routine generates an even series based on the double precision operand contained in DAC. The series is of the form:

```
X^0*(Kn)+x^2*(Kn-1)+x^4*(Kn-2)+x^6*(Kn-3) ...
```

The address of the coefficient list is supplied in register pair HL. The first byte of the list contains the coefficient count, the double precision coefficients follow with K1 first and Kn last. The method used to compute the polynomial is known as Horner's method. It only requires one multiplication and one addition per term, the BASIC equivalent is:

```
10 X=X*X
20 PRODUCT=0
30 RESTORE 100
40 READ COUNT
50 FOR N=1 TO COUNT
60 READ K
70 PRODUCT= ( PRODUCT*X ) +K
80 NEXT N
90 END
100 DATA 8
110 DATA Kn-7
120 DATA Kn-6
130 DATA Kn-5
140 DATA Kn-4
150 DATA Kn-3
160 DATA Kn-2
170 DATA Kn-1
180 DATA Kn
```

The polynomial is processed from the final coefficient through to the first coefficient so that the partial product can be used to save unnecessary operations.

Address... 2CC7H
This routine pushes an eight byte double precision operand from ARG onto the Z80 stack.

Address... 2CCCH
This routine pushes an eight byte double precision operand from DAC onto the Z80 stack.

Address... 2CDCH
This routine pops an eight byte double precision operand from the Z80 stack into ARG.

Address... 2CE1H
This routine pops an eight byte double precision operand from the Z80 stack into DAC.

Address... 2CF1H
This table contains the double precision constants used by the math routines. The first three constants have zero in the exponent position as they are in a special intermediate form used by the random number generator.

ADDR.	CONSTANT
2CF1H	.14389820420821 RND
2CF9H	.21132486540519 RND
2D01H	.40649651372358
2D09H	.43429448190324 LOG(e)
2D11H	.50000000000000
2D13H	.00000000000000
2D1BH	1.0000000000000
2D23H	.25000000000000
2D2BH	3.1622776601684 SQR(10)
2D33H	.86858896380650 2^LOG(e)
2D3BH	2.3025850929940 1/LOG(e)
2D43H	1.5707963267949 PI/2
2D4BH	.26794919243112 TAN(PI/12)
2D53H	1.7320508075689 TAN(PI/3)
2D5BH	.52359877559830 PI/6
2D63H	.15915494309190 1/(2^PI)
2D6BH	4 EXP
2D6CH	1.0000000000000
2D74H	159.37415236031
2D7CH	2709.3169408516
2D84H	4497.6335574058
2D8CH	3 EXP
2D8DH	18.312360159275
2D95H	831.40672129371
2D9DH	5178.0919915162
2DA5H	4 LOG
2DA6H	-.71433382153226

ADDR.	CONSTANT
2DAEH	6.2503651127908
2DB6H	-13.682370241503
2DBEH	8.5167319872389
2DC6H	5 LOG
2DC7H	1.0000000000000
2DCFH	-13.210478350156
2DD7H	47.925256043873
2DDFH	-64.906682740943
2DE7H	29.415750172323
2DEFH	8 SIN
2DF0H	-.69215692291809
2DF8H	3.8172886385771
2E00H	-15.094499474801
2E08H	42.058689667355
2E10H	-76.705859683291
2E18H	81.605249275513
2E20H	-41.341702240398
2E28H	6.2831853071796
2E30H	8 ATN
2E31H	-.05208693904000
2E39H	.07530714913480
2E41H	-.09081343224705
2E49H	.11110794184029
2E51H	-.14285708554884
2E59H	.19999999948967
2E61H	-.33333333333160
2E69H	1.0000000000000

Address... 2E71H
This routine returns the mantissa sign of a Floating Point operand contained in DAC. The exponent byte is tested and the result returned in register A and the flags:

```
Zero ....... A=00H, Flag Z,NC
Positive ... A=01H, Flag NZ,NC
Negative ... A=FFH, Flag NZ,C
```

Address... 2E7DH
This routine simply zeroes the exponent byte in DAC.

Address... 2E82H
This routine is used by the Factor Evaluator to apply the “ABS” function to an operand contained in DAC. The operand’s sign is first checked (2EA1H), if it is positive the routine simply terminates. The operand’s type is then checked via the GETYPR standard routine. If it is a string a “Type mismatch” error is generated (406DH). If it is an integer it is negated (322BH). If it is a double precision or single precision operand the mantissa sign bit in DAC is inverted.

Address... 2E97H
This routine is used by the Factor Evaluator to apply the “SGN” function to an operand contained in DAC. The operand’s sign is checked (2EA1H), extended into register pair HL and then placed in DAC as an integer:

```
Zero ....... 0000H
Positive ... 0001H
Negative ... FFFFH
```

Address... 2EA1H
This routine returns the sign of an operand contained in DAC. The operands type is first checked via the GETYPR standard routine. If it is a string a “Type mismatch” error is generated (406DH). If it is a single precision or double precision operand the mantissa sign is examined (2E71H). If it is an integer its value is taken from DAC+2 and translated into the flags shown at 2E71H.

Address... 2EB1H
This routine pushes a four byte single precision operand from DAC onto the Z80 stack.

Address... 2EC1H
This routine copies the contents of registers C, B, E and D to DAC.

Address... 2ECCH
This routine copies the contents of DAC to registers C, B, E and D.

Address... 2ED6H
This routine loads registers C, B, E and D from upwardly sequential locations starting at the address supplied in register pair HL.

Address... 2EDFH
This routine loads registers E, D, C and B from upwardly sequential locations starting at the address supplied in register pair HL.

Address... 2EE8H
This routine copies a single precision operand from DAC to the address supplied in register pair HL.

Address... 2EEFH
This routine copies any operand from the address supplied in register pair HL to ARG. The length of the operand is contained in VALTYP: 2=Integer, 3=String, 4=Single Precision, 8=Double Precision.

Address... 2F05H
This routine copies any operand from ARG to DAC. The length of the operand is contained in VALTYP: 2=Integer, 3=String, 4=Single Precision, 8=Double Precision.

Address... 2F0DH
This routine copies any operand from DAC to ARG. The length of the operand is contained in VALTYP: 2=Integer, 3=String, 4=Single Precision, 8=Double Precision.

Address... 2F21H
This routine is used by the Expression Evaluator to find the relation (<>=) between two single precision operands. The first operand is contained in registers C, B, E and D and the second in DAC. The result is returned in register A and the flags:

```
Operand 1=Operand 2 ... A=00H, Flag Z,NC
Operand 1<Operand 2 ... A=01H, Flag NZ,NC
Operand 1>Operand 2 ... A=FFH, Flag NZ,C
```

It should be noted that for relational operators the Expression Evaluator regards maximally negative numbers as small and maximally positive numbers as large.

Address... 2F4DH
This routine is used by the Expression Evaluator to find the relation (<>=) between two integer operands. The first operand is contained in register pair DE and the second in register pair HL. The results are as for the single precision version (2F21H).

Address... 2F83H
This routine is used by the Expression Evaluator to find the relation (<>=) between two double precision operands. The first operand is contained in DAC and the second in ARG. The results are as for the single precision version (2F21H).
Address... 2F8AH

This routine is used by the Factor Evaluator to apply the “CINT” function to an operand contained in DAC. The operand type is first checked via the GETYPR standard routine, if it is already integer the routine simply terminates. If it is a string a “Type mismatch” error is generated (406DH). If it is a single precision or double precision operand it is converted to a signed binary integer in register pair DE (305DH) and then placed in DAC as an integer. Out of range values result in an “Overflow” error (4067H).

Address... 2FA2H
This routine checks whether DAC contains the single precision operand -32768, if so it replaces it with the integer equivalent 8000H. This step is required during numeric input conversion (3299H) because of the asymmetric integer number range.

Address... 2FB2H
This routine is used by the Factor Evaluator to apply the "CSNG" function to an operand contained in DAC. The operand's type is first checked via the GETYPR standard routine, if it is already single precision the routine simply terminates. If it is a string a "Type mismatch" error is generated (406DH). If it is double precision VALTYP is changed (3053H) and the mantissa rounded up from the seventh digit (2741H). If the operand is an integer it is converted from binary to a maximum of five BCD digits by successive divisions using the constants 10000, 1000, 100, 10, 1. These are placed in DAC to form the single precision mantissa. The exponent is equal to the number of significant digits in the mantissa. For example if there are five the exponent would be 10^5.

Address... 3030H
This table contains the five constants used by the "CSNG" routine: -10000, -1000, -100, -10, -1

Address... 303AH
This routine is used by the Factor Evaluator to apply the "CDBL" function to an operand contained in DAC. The operand's type is first checked via the GETYPR standard routine, if it is already double precision the routine simply terminates. If it is a string a "Type mismatch" error is generated (406DH). If it is an integer it is first converted to single precision (2FC8H), the eight least significant digits are then zeroed and VALTYP set to 8.

Address... 3058H
This routine checks that the current operand is a string type, if not a "Type mismatch" error is generated (406DH).

Address... 305DH
This routine is used by the "CINT" routine (2F8AH) to convert a BCD single precision or double precision operand into a signed binary integer in register pair DE, it returns Flag C if an overflow has occurred. Successive digits are taken from the mantissa and added to the product starting with the most significant one. After each addition the product is multiplied by ten. The number of digits to process is determined by the exponent, for example five digits would be taken with an exponent of 10^5. Finally the mantissa sign is checked and the product negated (3221H) if necessary.

Address... 30BEH
This routine is used by the Factor Evaluator to apply the "FIX" function to an operand contained in DAC. The operand's type is first checked via the GETYPR standard routine, if it is an integer the routine simply terminates. The mantissa sign is then checked (2E71H), if it is positive control transfers to the "INT" routine (30CFH). Otherwise the sign is inverted to positive, the "INT" function is performed (30CFH) and the sign restored to negative.

Address... 30CFH
This routine is used by the Factor Evaluator to apply the "INT" function to an operand contained in DAC. The operand's type is first checked via the GETYPR standard routine, if it is an integer the routine simply terminates. The number of fractional digits is determined by subtracting the exponent from the type's digit count, 6 for single precision, 14 for double precision. If the mantissa sign is positive these fractional digits are simply zeroed. If the mantissa sign is negative each fractional digit is examined before it is zeroed. If all the digits were previously zero the routine simply terminates. Otherwise -1.0 is added to the operand by the single precision addition routine (324EH) or the double precision addition routine (269AH). It should be noted that an operand's type is not normally changed by the "CINT" function.

Address... 314AH
This routine multiplies the unsigned binary integers in register pairs BC and DE, the result is returned in register pair DE. The standard shift and add method is used, the product is successively multiplied by two and register pair BC added to it for every 1 bit in register pair DE. The routine is used by the Variable search routine (5EA4H) to compute an element's position within an Array, a "Subscript out of range" error is generated (601DH) if an overflow occurs.

Address... 3167H
This routine is used by the Expression Evaluator to subtract two integer operands. The first operand is contained in register pair DE and the second in register pair HL, the result is returned in DAC. The second operand is negated (3221H) and control drops into the addition routine.

Address... 3172H
This routine is used by the Expression Evaluator to add two integer operands. The first operand is contained in register pair DE and the second in register pair HL, the result is returned in DAC. The signed binary operands are normally just added and placed in DAC. However, if an overflow has occurred both operands are converted to single precision (2FCBH) and control transfers to the single precision adder (324EH). An overflow has occurred when both operands are of the same sign and the result is of the opposite sign, for example: `30000+15000=-20536`

Address... 3193H
This routine is used by the Expression Evaluator to multiply two integer operands. The first operand is contained in register pair DE and the second in register pair HL, the result is returned in DAC. The two operand signs are saved temporarily and both operands made positive (3215H). Multiplication proceeds using the standard binary shift and add method with register pair HL as the product accumulator, register pair BC containing the first operand and register pair DE the second. If the product exceeds 7FFFH at any time during multiplication both operands are converted to single precision (2FCBH) and control transfers to the single precision multiplier (325CH). Otherwise the initial signs are restored and, if they differ, the product negated before being placed in DAC as an integer (321DH).

Address... 31E6H
This routine is used by the Expression Evaluator to integer divide (\) two integer operands. The first operand is contained in register pair DE and the second in register pair HL, the result is returned in DAC. If the second operand is zero a “Division by zero” error is generated (4058H), otherwise the two operand signs are saved and both operands made positive (3215H). Division proceeds using the standard binary shift and subtract method with register pair HL containing the remainder, register pair BC the second operand and register pair DE the first operand and the product. When division is complete the initial signs are restored and, if they differ, the product is negated before being placed in DAC as an integer (321DH).

Address... 3215H
This routine is used to make two signed binary integers, in register pairs HL and DE, positive. Both the initial operand signs are returned as a flag in bit 7 of register B: 0=Same, 1=Different. Each operand is then examined and, if it is negative, made positive by subtracting it from zero.

Address... 322BH
This routine is used by the “ABS” function to make a negative integer contained in DAC positive. The operand is taken from DAC, negated and then placed back in DAC (3221H). If the operand’s value is 8000H it is converted to single precision (2FCCH) as there is no integer of value +32768.

Address... 323AH
This routine is used by the Expression Evaluator to “MOD” two integer operands. The first operand is contained in register pair DE and the second in register pair HL, the result is returned in DAC. The sign of the first operand is saved and the two operands divided (31E6H). As the remainder is returned doubled by the division process register pair DE is shifted one place right to restore it. The sign of the first operand is then restored and, if it is negative, the remainder is negated before being placed in DAC as an integer (321DH).

Address... 324EH
This routine is used by the Expression Evaluator to add two single precision operands. The first operand is contained in registers C, B, E, D and the second in DAC, the result is returned in DAC. The first operand is copied to ARG (3280H), the second operand is converted to double precision (3042H) and control transfers to the double precision adder (269AH).

Address... 3257H
This routine is used by the Expression Evaluator to subtract two single precision operands. The first operand is contained in registers C, B, E, D and the second in DAC, the result is returned in DAC. The second operand is negated (2E8DH) and control transfers to the single precision adder (324EH).

Address... 325CH
This routine is used by the Expression Evaluator to multiply two single precision operands. The first operand is contained in registers C, B, E, D and the second in DAC, the result is returned in DAC. The first operand is copied to ARG (3280H), the second operand is converted to double precision (3042H) and control transfers to the double precision multiplier (27E6H).

Address... 3265H
This routine is used by the Expression Evaluator to divide two single precision operands. The first operand is contained in registers C, B, E, D and the second in DAC, the result is returned in DAC. The first and second operands are exchanged so that the first is in DAC and the second in the registers. The second operand is then copied to ARG (3280H), the first operand is converted to double precision (3042H) and control transfers to the double precision divider (289FH).

Address... 3280H
This routine copies the single precision operand contained in registers C, B, E and D to ARG and then zeroes the four least significant bytes.

Address... 3299H
This routine converts a number in textual form to one of the standard internal numeric types, it is used during tokenization and by the "VAL", "INPUT" and "READ" Statement handlers. On entry register pair HL points to the first character of the text string to be converted. On exit register pair HL points to the character following the string, the numeric operand is in DAC and the type code in VALTYP. Examples of the three types are:

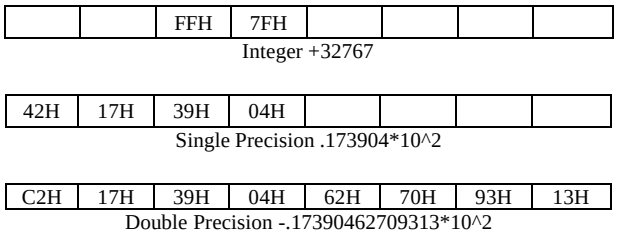

Figure 41: Numeric Types in DAC.

An integer is a sixteen bit binary number in two's complement form, it is stored LSB first, MSB second at DAC+2. An integer can range from 8000H (-32768) to 7FFFH (+32767). A floating point number consists of an exponent byte and a three or seven byte mantissa. The exponent is kept in signed binary form and can range from 01H (-63) through 40H (0) up to 7FH (+63), the special value of 00H is used for the number zero. These exponent values are for a normalized mantissa. The Interpreter presents exponent-form numbers to the user with a leading digit, this results in an asymmetric exponent range of E-64 to E+62. Bit 7 of the exponent byte holds the mantissa sign, 0 for positive and 1 for negative, the mantissa itself is kept in packed BCD form with two digits per byte. It should be noted that the Interpreter uses the contents of VALTYP to determine a number's type, not the format of the number itself. Conversion starts by examining the first text character. If this is an "&" control transfers to the special radix conversion routine (4EB8H), if it is a leading sign character it is temporarily saved. Successive numeric characters are then taken and added to the integer product with appropriate multiplications by ten as each new digit is found. If the value of the product exceeds 32767, or a decimal point is found, the product is converted to single precision and any further characters placed directly in DAC. If a seventh digit is found the product is changed to double precision, if more than fourteen digits are found the excess digits are read but ignored. Conversion ceases when a non-numeric character is found. If this a type definition character ("%", "#" or "!") the appropriate conversion routine is called and control transfers to the exit point (331EH). If it is an exponent prefix ("E", "e", "D" or "d") one of the conversion routines will also be used and then the following digits converted to a binary exponent in register E. At the exit point (331EH) the product's type is checked via the GETYPR standard routine. If it is single precision or double precision the exponent is calculated by first subtracting the fractional digit count, in register B, from the total digit count, in register D, to produce the leading digit count. This is then added to any explicitly stated exponent, in register E, and placed at DAC+0 as the exponent. The leading sign character is restored and the product negated if required (2E86H), if the product is integer the routine then terminates. If the product is single precision control terminates by checking for the special value of -32768 (2FA2H). If the product is double precision control terminates by rounding up from the fifteenth digit (273CH).

Address... 340AH
This routine is used by the error handler to display the message " in " (6678H) followed by the line number supplied in register pair HL (3412H).

Address... 3412H
This routine displays the unsigned binary integer supplied in register pair HL. The operand is placed in DAC as an integer (2F99H), converted to text (3441H) and then displayed (6677H).

Address... 3425H
This routine converts the numeric operand contained in DAC to textual form in FBUFFR. The address of the first character of the resulting text is returned in register pair HL, the text is terminated by a zero byte. The operand is first converted to double precision (375FH). The BCD digits of the mantissa are then unpacked, converted to ASCII and placed in FBUFFR (36B3H). The position of the decimal point is determined by the exponent, for example:

```
.999*10 ^ +2 = 99.9
.999*10 ^ +1 = 9.99
.999*10 ^ +0 = .999
.999*10 ^ -1 = .0999
```

If the exponent is outside the range 10^-1 to 10^14 the number is presented in exponential form. In this case the decimal point is placed after the first digit and the exponent is converted from binary and follows the mantissa. An alternative entry point to the routine exists at 3426H for the "PRINT USING" statement handler. With this entry point the number of characters to prefix the decimal point is supplied in register B, the number of characters to point fix it in register C and a format byte in register A:

7	6	5	4	3	2	1	0
1	,	*	$	+	Sign	0	^^^^

Figure 42: Format Byte.

Operation in this mode is fairly similar to the normal mode but with the addition of extra facilities. Once the operand has been converted to double precision the exponential form will be assumed if bit 0 of the format byte is set. The mantissa is shifted to the right in DAC and rounded up to lose unwanted postfix digits (377BH). As the mantissa is converted to ASCII (36B3H) commas will be inserted at the appropriate points if bit 6 of the format byte is set. During post-conversion formatting (351CH) unused prefix positions will be filled with asterisks if bit 5 is set, a pound prefix may be added by setting bit 4. Bit 3 enables the "+" sign for positive numbers if set, otherwise a space is used. Bit 2 places any sign at the front if reset and at the back if set. The entry point to the routine at 3441H is used to convert unsigned integers, notably line numbers, to their textual form. For example 9000H, when treated as a normal integer, would be converted to -28672. By using this entry point 36864 would be produced instead. The operand is converted by successive division with the factors 10000, 1000, 100, 10 and 1 and the resulting digits placed in FBUFFR (36DBH).

Address... 3710H
This table contains the five constants used by the numeric output routine: 10000, 1000, 100, 10, 1.

Address... 371AH
This routine is used by the "BIN$" function to convert a numeric operand contained in DAC to textual form. Register B is loaded with the group size (1) and control transfers to the general conversion routine (3724H).

Address... 371EH
This routine is used by the "OCT$" function to convert a numeric operand contained in DAC to textual form. Register B is loaded with the group size (3) and control transfers to the general conversion routine (3724H).

Address... 3722H
This routine is used by the "HEX$" function to convert a numeric operand contained in DAC to textual form. Register B is loaded with the group size (4) and the operand converted to a binary integer in register pair HL (5439H). Successive groups of 1, 3 or 4 bits are shifted rightwards out of the operand, converted to ASCII digits and placed in FBUFFR. When the operand is all zeroes the routine terminates with the address of the first text character in register pair HL, the string is terminated with a zero byte.

Address... 3752H
This routine is used during numeric output to return an operand's digit count in register B and the address of its least significant byte in register pair HL. For single precision B=6 and HL=DAC+3, for double precision B=14 and HL=DAC+7.

Address... 375FH
This routine is used during numeric output to convert the numeric operand in DAC to double precision (303AH).

Address... 377BH
This routine is used during numeric output to shift the mantissa in DAC rightwards (27DBH), the inverse of the digit count is supplied in register A. The result is then rounded up from the fifteenth digit (2741H).

Address... 37A2H
This routine is used during numeric output to return the inverse of the fractional digit count in a floating point operand. This is computed by subtracting the exponent from the operand's digit count (6 or 14).

Address... 37B4H
This routine is used during numeric output to locate the last non-zero digit of the mantissa contained in DAC. Its address is returned in register pair HL.

Address... 37C8H
This routine is used by the Expression Evaluator to exponentiate (^) two single precision operands. The first operand is contained in registers C, B, E, D and the second in DAC, the result is returned in DAC. The first operand is copied to ARG (3280H), pushed onto the stack (2CC7H) and exchanged with DAC (2C6FH). The second operand is then popped into ARG and control drops into the double precision exponentiation routine.

Address... 37D7H
This routine is used by the Expression Evaluator to exponentiate (^) two double precision operands. The first operand is contained in DAC and the second in ARG, the result is returned in DAC. The result is usually computed using:

```
X^P=EXP(P*LOG(X))
```

An alternative, much faster, method is possible if the power operand is an integer. This is tested for by extracting the integer part of the operand and comparing for equality with the original value (391AH). A positive result to this test means that the faster method can be used, this is described below.

Address... 383FH
This routine is used by the Expression Evaluator to exponentiate (^) two integer operands. The first operand is contained in register pair DE and the second in register pair HL, the result is returned in DAC. The routine operates by breaking the problem down into simple multiplications: 6^13=6^1101=(6^8)*(6^4)*(6^1). As the power operand is in binary form a simple right shift is sufficient to determine whether a particular intermediate product needs to be included in the result. The intermediate products themselves are obtained by cumulative multiplication of the operand each time the computation loop is traversed. If the product overflows at any time it is converted to single precision. Upon completion the power operand is checked, if it is negative the product is reciprocated as X^-P=1/X^P.

Address... 390DH
This routine is used during exponentiation to multiply two integers (3193H), it returns Flag NZ if the result has overflowed to single precision.

Address... 391AH
This routine is used during exponentiation to check whether a double precision power operand consists only of an integer part, if so it returns Flag NC.

Address... 392EH
This table of addresses is used by the Interpreter Runloop to find the handler for a statement token. Although not part of the table the associated keywords are included below:

TO	STMT
63EAH	END
4524H	FOR
6527H	NEXT
485BH	DATA
4B6CH	INPUT
5E9FH	DIM
4B9FH	READ
4880H	LET
47E8H	GOTO
479EH	RUN
49E5H	IF
63C9H	RESTORE
47B2H	GOSUB
4821H	RETURN
485DH	REM
63E3H	STOP
4A24H	PRINT
64AFH	CLEAR
522EH	LIST
6286H	NEW
48E4H	ON
401CH	WAIT
501DH	DEF
5423H	POKE
6424H	CONT
6FB7H	CSAVE
703FH	CLOAD
4016H	OUT
4A1DH	LPRINT
5229H	LLIST

TO	STMT
00C3H	CLS
51C9H	WIDTH
485DH	ELSE
6438H	TRON
6439H	TROFF
643EH	SWAP
6477H	ERASE
49AAH	ERROR
495DH	RESUME
53E2H	DELETE
49B5H	AUTO
5468H	RENUM
4718H	DEFSTR
471BH	DEFINT
471EH	DEFSNG
4721H	DEFDBL
4B0EH	LINE
6AB7H	OPEN
7C52H	FIELD
775BH	GET
7758H	PUT
6C14H	CLOSE
6B5DH	LOAD
6B5EH	MERGE
6C2FH	FILES
7C48H	LSET
7C4DH	RSET
6BA3H	SAVE
6C2AH	LFILES

TO	STMT
5B11H	CIRCLE
7980H	COLOR
5D6EH	DRAW
59C5H	PAINT
00C0H	BEEP
73E5H	PLAY
57EAH	PSET
57E5H	PRESET
73CAH	SOUND
79CCH	SCREEN
7BE2H	VPOKE
7A48H	SPRITE
7B37H	VDP
7B5AH	BASE
55A8H	CALL
7911H	TIME
786CH	EY
7E4BH	MAX
73B7H	MOTOR
6EC6H	BLOAD
6E92H	BSAVE
7C16H	DSKO$
7C1BH	SET
7C20H	NAME
7C25H	KILL
7C2AH	IPL
7C2FH	COPY
7C34H	CMD
7766H	LOCATE

Address... 39DEH

This table of addresses is used by the Factor Evaluator to find the handler for a function token. Although not part of the table the associated keywords are included with the addresses shown below:

TO	FUNCTION
6861H	LEFT$
6891H	RIGHT$
689AH	MID$
2E97H	SGN
30CFH	INT
2E82H	ABS
2AFFH	SQR
2BDFH	RND
29ACH	SIN
2A72H	LOG
2B4AH	EXP
2993H	COS
29FBH	TAN
2A14H	ATN
69F2H	FRE
4001H	INP

TO	FUNCTION
4FCCH	POS
67FFH	LEN
6604H	TR$
68BBH	VAL
680BH	ASC
681BH	CHR$
541CH	PEEK
7BF5H	VPEEK
6848H	SPACE$
7C70H	OCT$
65FAH	HEX$
4FC7H	LPOS
6FFFH	BIN$
2F8AH	CINT
2FB2H	CSNG
303AH	CDBL

TO	FUNCTION
30BEH	FIX
7940H	STICK
794CH	TRIG
795AH	PDL
7969H	PAD
7C39H	DSKF
6D39H	FPOS
7C66H	CVI
7C6BH	CVS
7C70H	CVD
6D25H	EOF
6D03H	LOC
6D14H	LOF
7C57H	MKI$
7C5CH	MKS$
7C61H	MKD$

Address... 3A3EH

This table of addresses is used during program tokenization as an index into the BASIC keyword table (3A72H). Each of the twenty six entries defines the starting address of one of the keyword sub-blocks. The first entry points to the keywords beginning with the letter “A”, the second to those beginning with the letter “B” and so on.

```
3A72H ... A   3B9FH ... J    3C8EH ... S
3A88H ... B   3BA0H ... K    3CDBH ... T
3A9FH ... C   3BA8H ... L    3CF6H ... U
3AF3H ... D   3BE8H ... M    3CFFH ... V
3B2EH ... E   3C09H ... N    3D16H ... W
3B4FH ... F   3C18H ... O    3D20H ... X
3B69H ... G   3C2BH ... P    3D24H ... Y
3B7BH ... H   3C5DH ... Q    3D25H ... Z
3B80H ... I   3C5EH ... R
```

Address... 3A72H

This table contains the BASIC keywords and tokens. Each of the twenty-six blocks within the table contains all the keywords beginning with a particular letter, it is terminated with a zero byte. Each keyword is stored in plain text with bit 7 set to mark the last character, this is followed immediately by the associated token. The first character of the keyword need not be stored as this is implied by its position in the table’ The keywords and tokens are listed below in full, note that the “J”, “Q”, “Y” and “Z” blocks are empty:

AUTO	A9H
AND	F6H
ABS	06H
ATN	0EH
ASC	15H
ATTR$	E9H
BASE	C9H
BSAVE	D0H
BLOAD	CFH
BEEP	C0H
BIN$	1DH
CALL	CAH
CLOSE	B4H
COPY	D6H
CONT	99H
CLEAR	92H
CLOAD	9BH
CSAVE	9AH
CSRLIN	E8H
CINT	1EH
CSNG	1FH
CDBL	20H
CVI	28H
CVS	29H
CVD	2AH
COS	0CH
CHR$	16H
CIRCLE	BCH
COLOR	BDH
CLS	9FH
CMD	D7H
DELETE	A8H
DATA	84H
DIM	86H
DEFSTR	ABH
DEFINT	ACH
DEFSNG	ADH
DEFDBL	AEH
DSKO$	D1H
DEF	97H
DSKI$	EAH

DSKF	26H
DRAW	BEH
ELSE	A1H
END	81H
ERASE	A5H
ERROR	A6H
ERL	E1H
ERR	E2H
EXP	0BH
EOF	2BH
EQV	F9H
FOR	82H
FIELD	B1H
FILES	B7H
FN	DEH
FRE	0FH
FIX	21H
FPOS	27H
GOTO	89H
GO TO	89H
GOSUB	8DH
GET	B2H
HEX$	1BH
INPUT	85H
IF	8BH
INSTR	E5H
INT	05H
INP	10H
IMP	FAH
INKEY$	ECH
IPL	D5H
KILL	D4H
KEY	CCH
LPRINT	9DH
LLIST	9EH
LPOS	1CH
LET	88H
LOCATE	D8H
LINE	AFH
LOAD	B5H
LSET	B8H

LIST	93H
LFILES	BBH
LOG	0AH
LOC	2CH
LEN	12H
LEFT$	01H
LOF	2DH
MOTOR	CEH
MERGE	B6H
MOD	FBH
MKI$	2EH
MKS$	2FH
MKD$	30H
MID$	03H
MAX	CDH
NEXT	83H
NAME	D3H
NEW	94H
NOT	E0H
OPEN	B0H
OUT	9CH
ON	95H
OR	F7H
OCT$	1AH
OFF	EBH
PRINT	91H
PUT	B3H
POKE	98H
POS	11H
PEEK	17H
PSET	C2H
PRESET	C3H
POINT	EDH
PAINT	BFH
PDL	24H
PAD	25H
PLAY	C1H
RETURN	8EH
READ	87H
RUN	8AH
RESTORE	8CH

REM	8FH
RESUME	A7H
RSET	B9H
RIGHT$	02H
RND	08H
RENUM	AAH
SCREEN	C5H
SPRITE	C7H
STOP	90H
SWAP	A4H
SET	D2H
SAVE	BAH
SPC(	DFH
STEP	DCH
SGN	04H
SQR	07H
SIN	09H
STR$	13H
STRING$	E3H
SPACE$	19H
SOUND	C4H
STICK	22H
STRIG	23H
THEN	DAH
TRON	A2H
TROFF	A3H
TAB(	DBH
TO	D9H
TIME	CBH
TAN	0DH
USING	E4H
USR	DDH
VAL	14H
VARPTR	E7H
VDP	C8H
VPOKE	C6H
VPEEK	18H
WIDTH	A0H
WAIT	96H
XOR	F8H

Address... 3D26H
This twenty-one byte table is used by the Interpreter during program tokenization. It contains the ten single character keywords and their tokens:

```
+... F1H    * ... F3H   ^ ... F5H    ‘ ...  E6H = ... EFH
-... F2H    / ... F4H   \ ... FCH    > ...  EEH < ... F0H
```

Address... 3D3BH
This table is used by the Expression Evaluator to determine the precedence level for a given infix operator, the higher the table value the greater the operator’s precedence. Not included are the precedences for the relational operators (64H), the “NOT” operator (5AH) and the negation operator (7DH), these are defined directly by the Expression and Factor Evaluators.

```
79H ... +       46H ... OR
79H ... -       3CH ... XOR
7CH ... *       32H ... EQV
7CH ... /       28H ... IMP
7FH ... ^       7AH ... MOD
50H ... AND     7BH     \
```

Address... 3D47H
This table is used to convert the result of a user defined function to the same type as the Variable used in the function definition. It contains the addresses of the type conversion routines:

```
303AH ... CDBL
0000H ... Not used
2F8AH ... CINT
3058H ... Check string type
2FB2H ... CSNG
```

Address... 3D51H
This table of addresses is used by the Expression Evaluator to find the handler for a particular infix math operator when both operands are double precision:

```
269AH ... +
268CH ... -
27E6H ... *
289FH ... /
37D7H ... ^
2F83H ... Relation
```

Address... 3D5DH
This table of addresses is used by the Expression Evaluator to find the handler for a particular infix math operator when both operands are single precision:

```
324EH ... +
3257H ... -
325CH ... *
3267H ... /
37C8H ... ^
2F21H ... Relation
```

Address... 3D69H
This table of addresses is used by the Expression Evaluator to find the handler for a particular infix math operator when both operands are integer:

```
3172H ... +
3167H ... -
3193H ... *
4DB8H ... /
383FH ... ^
2F4DH ... Relation
```

Address... 3D75H
This table contains the Interpreter error messages, each one is stored in plain text with a zero byte terminator. The associated error codes are shown below for reference only, they do not form part of the table:

01	NEXT without FOR	19	Device I/O error
02	Syntax error	20	Verify error
03	RETURN without GOSUB	21	No RESUME
04	Out of DATA	22	RESUME without error
05	Illegal function call	23	Unprintable error
06	Overflow	24	Missing operand
07	Out of memory	25	Line buffer overflow
08	Undefined line number	50	FIELD overflow
09	Subscript out of range	51	Internal error
10	Redimensioned array	52	Bad file number
11	Division by zero	53	File not found
12	Illegal direct	54	File already open
13	Type mismatch	55	Input past end
14	Out of string space	56	Bad file name
15	String too long	57	Direct statement in file
16	String formula too complex	58	Sequential I/O only
17	Can’t CONTINUE	59	File not OPEN
18	Undefined user function		

Address... 3FD2H
This is the plain text message " in " terminated by a zero byte.

Address... 3FD7H
This is the plain text message "Ok", CR, LF terminated by a zero byte.

Address... 3FDCH
This is the plain text message "Break" terminated by a zero byte.

Address... 3FE2H
This routine searches the Z80 stack for the "FOR" loop parameter block whose loop Variable address is supplied in register pair DE. The search is started four bytes above the current Z80 SP to allow for the caller's return address and the Runloop return address. If no "FOR" token (82H) exists the routine terminates Flag NZ, if one is found the loop Variable address is taken from the parameter block and checked. The routine terminates Flag Z upon a successful match with register pair HL pointing to the type byte of the parameter block. Otherwise the search moves up twenty-two bytes to the start of the next parameter block.

Address... 4001H
This routine is used by the Factor Evaluator to apply the "INP" function to an operand contained in DAC. The port number is checked (5439H), the port read and the result placed in DAC as an integer (4FCFH).

Address... 400BH
This routine first evaluates an operand in the range -32768 to +65535 (542FH) and places it in register pair BC. After checking for a comma, via the SYNCHR standard routine, it evaluates a second operand in the range 0 to 255 (521CH) and places this in register A.

Address... 4016H
This is the "OUT" statement handler. The port number and data byte are evaluated (400BH) and the data byte written to the relevant Z80 port.

Address... 401CH
This is the "WAIT" statement handler. The port number and "AND" operands are first evaluated (400BH) followed by the optional "XOR" operand (521CH). The port is then repeatedly read and the operands applied, XOR then AND, until a non-zero result is obtained. Contrary to the information given in some MSX manuals the loop can be broken by the CTRL-STOP key as the CKCNTC standard routine is called from inside it.

Address... 4039H
This routine is used by the Runloop when it encounters the end of the program text while in program mode. ONEFLAG is checked to see whether it still contains an active error code. If so a "No RESUME" error is generated, otherwise program termination continues normally (6401H). The idea behind this routine is to catch any "ON ERROR" handlers without a "RESUME" statement at the end.

Address... 404FH
This routine is used by the "READ" statement handler when an error is found in a "DATA" statement. The line number contained in DATLIN is copied to CURLIN so the error handler will flag the "DATA" line as the illegal statement rather than the program line. Control then drops into the "Syntax error" generator.

Address... 4055H
This is a group of nine error generators, register E is loaded with the relevant error code and control drops into the error handler:

```
ADDR. ERROR
4055H Syntax error
4058H Division by zero
405BH NEXT without FOR
405EH Redimensioned array
4061H Undefined user function
4064H RESUME without error
4067H Overflow error
406AH Missing operand
406DH Type mismatch
```

Address... 406F
This is the Interpreter error handler, all error generators transfer to here with an error code in register E. VLZADR is first checked to see if the “VAL” statement handler has changed the program text, if so the original character is restored from VLZDAT. The current line number is then copied from CURLIN to ERRLIN and DOT and the Z80 stack is restored from SAVSTK (62F0H). The error code is placed in ERRFLG, for use by the “ERR” function, and the current program text position copied from SAVTXT to ERRTXT for use by the “RESUME” statement handler. The error line number and program text position are also copied to OLDLIN and OLDTXT for use by the “CONT” statement handler. ONELIN is then checked to see if a previous “ON ERROR” statement has been executed. If so, and providing no error code is already active, control transfers to the Runloop (4620H) to execute the BASIC error recovery statements. Otherwise the error code is used to count through the error message table at 3D75H until the required one is reached. A CR,LF is issued (7323H) and the screen forced back to text mode via the TOTEXT standard routine. A BELL code is then issued and the error message displayed (6678H). Assuming the Interpreter is in program mode, rather than direct mode, this is followed by the line number (340AH) and control drops into the “OK” point.

Address... 411FH
This is the re-entry point to the Interpreter Mainloop for a terminating program. The screen is forced to text mode via the TOTEXT standard routine, the printer is cleared (7304H) and I/O buffer 0 closed (6D7BH). A CR,LF is then issued to the screen (7323H), the message “OK” is displayed (6678H) and control drops into the Mainloop.

Address... 4134H
This is the Interpreter Mainloop. CURLIN is first set to FFFFH to indicate direct mode and AUTFLG checked to see if “AUTO” mode is on. If so the next line number is taken from AUTLIN and displayed (3412H). The Program Text Area is then searched to see if this line already exists (4295H) and either an asterisk or space displayed accordingly. The ISFLIO standard routine is then used to determine whether a “LOAD” statement is active. If so the program line is collected from the cassette (7374H), otherwise it is taken from the console via the PINLIN standard routine. If the line is empty or the CTRL-STOP key has been pressed control transfers back to the start of the Mainloop (4134H) with no further action. If the line commences with a line number this is converted to an unsigned integer in register pair DE (4769H). The line is then converted to tokenized form and placed in KBUF (42B2H). If no line number was found at the start of the line control then transfers to the Runloop (6D48H) to execute the statement. Assuming the line commences with a line number it is tested to see if it is otherwise empty and the result temporarily saved. The line number is copied to DOT and AUTLIN increased by the contents of AUTINC, if AUTLIN now exceeds 65530 the “AUTO” mode is turned off. The Program Text Area is then searched (4295H) to find a matching line number or, failing this, the position of the next highest line number. If no matching line number is found and the line is empty and “AUTO” mode is off an “Undefined line number” error is generated (481CH). If a matching line number is found and the line is empty and “AUTO” mode is on the Mainloop simply skips to the next statement (4237H). Otherwise any pointers in the Program Text Area are converted back to line numbers (54EAH) and any existing program line deleted (5405H). Assuming the new program line is non-empty the Program Text Area is opened up by the required amount (6250H) and the tokenized program line copied from KBUF. The Program Text Area links are then recalculated (4257H), the Variable Storage Areas are cleared (629AH) and control transfers back to the start of the Mainloop.

Address... 4253H
This routine recalculates the Program Text Area links after a program modification. The first two bytes of each program line contain the starting address of the following line, this is called the link. Although the link increases the amount of storage required per program line it greatly reduces the time required by the Interpreter to locate a given line. An example of a typical program line is shown below, in this case the line “10 PRINT 9” situated at the start of the Program Text Area (8001H):

09H	80H	0AH	00H	91H	20H	1A	00H

Figure 43: Program Line.

In the above example the link is stored in Z80 word order (LSB,MSB) and is immediately followed by the binary line number, also in word order. The statement itself is composed of a “PRINT” token (91H), a single space, the number nine and the end of line character (00H). Further details of the storage format can be found in the tokenizing routine (42B2H). Each link is recalculated simply by scanning through the line until the end of line character is found. The process is complete when the end of the Program Storage Area, marked by the special link of 0000H, is reached.

Address... 4279H
This routine is used by the “LIST” statement handler to collect up to two line number operands from the program text. If the first line number is present it is converted to an unsigned integer in register pair DE (475FH), if not a default value of 0000H is returned. If the second line number is present it must be preceded by a “-“ token (F2H) and is returned on the Z80 stack, if not a default value of 65530 is returned. Control then drops into the program text search routine to find the first referenced program line.

Address... 4295H
This routine searches the Program Text Area for the program line whose line number is supplied in register pair DE. Starting at the address contained in TXTTAB each program line is examined for a match. If an equal line number is found the routine terminates with Flag Z,C and register pair BC pointing to the start of the program line. If a higher line number is found the routine terminates Flag NZ,NC and if the end link is reached the routine terminates Flag Z,NC.

Address... 42B2H
This routine is used by the Interpreter Mainloop to tokenize a line of text. On entry register pair HL points to the first text character in BUF. On exit the tokenized line is in KBUF, register pair BC holds its length and register pair HL points to its start. Except after opening quotes or after the "REM", "CALL" or "DATA" keywords any string of characters matching a keyword is replaced by that keyword's token. Lower case alphabetics are changed to upper case for keyword comparison. The character "?" is replaced by the "PRINT" token (91H) and the character "'" by ":" (3AH), "REM" token (8FH), "'" token (E6H). The "ELSE" token (A1H) is preceded by a statement separator (3AH). Any other miscellaneous characters in the text are copied without alteration except that lower case alphabetics are converted to upper case. Those tokens smaller than 80H, the function tokens, cannot be stored directly in KBUF as they will conflict with ordinary text. Instead the sequence FFH, token+80H is used. Numeric constants are first converted into one of the standard types in DAC (3299H). They are then stored in one of several ways depending upon their type and magnitude, the general idea being to minimize memory usage:

```
0BH LSB MSB ................... Octal number
0CH LSB MSB ................... Hex number
11H to 1AH .................... Integer 0 to 9
0FH LSB ....................... Integer 10 to 255
1CH LSB MSB ................... Integer 256 to 32767
1DH EE DD DD DD ............... Single Precision
1FH EE DD DD DD DD DD DD DD ... Double Precision
```

There is no specific token for binary numbers, these are left as character strings. This would appear to be a legacy from earlier versions of Microsoft BASIC. Any sign prefixing a number is regarded as an operator and is stored as a separate token, negative numbers are not produced during tokenization. As double precision numbers occupy so much space a line containing too many, for example PRINT 1#,1#,1# etc. may cause KBUF to fill up. If this happens a "Line buffer overflow" error is generated. Any number following one of the keyword tokens in the table at 43B5H is considered to be a line number operand and is stored with a different token:

```
0DH LSB MSB ................... Pointer
0EH LSB MSB ................... Line number
```

During tokenization only the normal type (0EH) is generated, when a program actually runs these line number operands are converted to the address pointer type (0DH).

Address... 43B5H
This table of tokens is used during tokenization to check for the keywords which take line number operands. The keywords themselves are listed below:

```
RESTORE     RUN
AUTO        LIST
RENUM       LLIST
DELETE      GOTO
RESUME      RETURN
ERL         THEN
ELSE        GOSUB
```

Address... 4524H
This is the "FOR" statement handler. The loop Variable is first located and assigned its initial value by the "LET" handler (4880H), the address of the loop Variable is returned in register pair DE. The end of the statement is found (485BH) and its address placed in ENDFOR. The Z80 stack is then searched (3FE6H) for any parameter blocks using the same loop Variable. For each one found the current ENDFOR address is compared with that of the parameter block, if there is a match that section of the stack is discarded. This is done in case there are any incomplete loops as a result of a "GOTO" back to the "FOR" statement from inside the loop. The termination operand and optional "STEP" operand are then evaluated and converted to the same type as the loop Variable. After checking that stack space is available (625EH) a twenty-five byte parameter block is pushed onto the Z80 stack. This is made up of the following:

```
2 bytes ... ENDFOR address
2 bytes ... Current line number
8 bytes ... Loop termination value
8 bytes ... STEP value
1 byte  ... Loop type
1 byte  ... STEP direction
2 bytes ... Address of loop Variable
1 byte  ... FOR token (82H)
```

The parameter block remains on the stack for use by the “NEXT” statement handler until termination is reached, it is then discarded. The size of the block remains constant even though, for integer and single precision loop Variables, the full eight bytes are not required for the termination and STEP values. In these cases the least significant bytes are packed out with garbage. It should be noted that the type of arithmetic operation performed by the “NEXT” statement handler, and hence the loop execution speed, depends entirely upon the loop Variable type and not the operand types. For the fastest program execution integer type Variables, N% for example, should be used.

Address... 4601H
This is the Runloop, each statement handler returns here upon completion so the Interpreter can proceed to the next statement. The current Z80 SP is copied to SAVSTK for error recovery purposes and the CTRL-STOP key checked via the ISCNTC standard routine. Any pending interrupts are processed (6389H) and the current program text position, held in register pair HL throughout the Interpreter, is copied to SAVTXT. The current program character is then examined, if this is a statement separator (3AH) control transfers immediately to the execution point (4640H). If it is anything else but an end of line character (00H) a “Syntax error” is generated (4055H) as there is spurious text at the end of the statement. Register pair HL is advanced to the first character of the new program line and the link examined, if this is zero the program is terminated (4039H). Otherwise the line number is taken from the new line and placed in CURLIN. If TRCFLG is non-zero the line number is displayed (3412H) enclosed by square brackets, control then drops into the execution point.

Address... 4640H
This is the Runloop execution point. A return to the start of the Runloop (4601H) is pushed onto the Z80 stack and the first character taken from the new statement via the CHRGTR standard routine. If it is an underline character (5FH) control transfers to the “CALL” statement handler (55A7H). If it is smaller than 81H, the smallest statement token, control transfers to the “LET” handler (4880H). If it is larger than D8H, the largest statement token, it is checked to see if it is one of the function tokens allowed as a statement (51ADH). Otherwise the handler address is taken from the table at 392EH and pushed onto the stack. Control then drops into the CHRGTR standard routine to fetch the next program character before control transfers to the statement handler.

Address... 4666H
Name..... CHRGTR
Entry..... HL points to current program character
Exit..... A=Next program character
Modifies.. AF, HL

Standard routine to fetch the next character from the program text. Register pair HL is incremented and the character placed in register A. If it is a space, TAB code (09H) or LF code (0AH) it is skipped over. If it is a statement separator (3AH) or end of line character (00H) the routine terminates with Flag Z,NC. If it is a digit from “0” to “9” the routine terminates with Flag NZ,C. If it is any other character apart from the numeric prefix tokens the routine terminates Flag NZ,NC. If the character is one of the numeric prefix tokens then it is placed in CONSAV and the operand copied to CONLO. The type code is placed in CONTYP and the address of the trailing program character in CONTXT.

Address... 46E8H
This routine is used by the Factor Evaluator and during detokenization to recover a numeric operand when one of the prefix tokens is returned by the CHRGTR standard routine. The prefix token is first taken from CONSAV, if it is anything but a line number or pointer token the operand is copied from CONLO to DAC and the type code copied from CONTYP to VALTYP. If it is a line number it is converted to single precision and placed in DAC (3236H). If it is a pointer the original line number is recovered from the referenced program line, converted to single precision and placed in DAC (3236H).

Address... 4718H
This is the “DEFSTR” statement handler. Register E is loaded with the string type code (03H) and control drops into the general type definition routine.

Address... 471BH
This is the “DEFINT” statement handler. Register E is loaded with the integer type code (02H) and control drops into the general type definition routine.

Address... 471EH
This is the “DEFSNG” statement handler. Register E is loaded with the single precision type code (04H) and control drops into the general type definition routine.

Address... 4721H
This is the “DEFDBL” statement handler. Register E is loaded with the double precision type code (08H) and the first range definition character checked (64A7H). If this is not upper case alphabetic a “Syntax error” is generated (4055H). If a “-“ token (F2H) follows the second range definition character is taken and checked (64A7H), the difference between the two determines the number of entries in DEFTBL that are filled with the type code.

Address... 4755H
This routine evaluates an operand and converts it to an integer in register pair DE (520FH). If the operand is negative an “Illegal function call” error is generated.

Address... 475FH
This routine is used by the statement handlers shown in the table at 43B5H to collect a single line number operand from the program text and convert it to an unsigned integer in register pair DE. If the first character in the text is a “.” (2EH) the routine terminates with the contents of DOT. If it is one of the line number tokens (0DH or 0EH) the routine terminates with the contents of CONLO. Otherwise successive digits are taken and added to the product, with appropriate multiplications by ten, until a non-numeric character is found.

Address... 479EH
This is the “RUN” statement handler. If no line number operand is present in the program text the system is cleared (629AH) and control returns to the Runloop with register pair HL pointing to the start of the Program Storage Area. If a line number operand is present the system is cleared (62A1H) and control transfers to the “GOTO” statement handler (47E7H). Otherwise a following filename is assumed, for example RUN “CAS:FILE”, and control transfers to the “LOAD” statement handler (6B5BH).

Address... 47B2H
This is the “GOSUB” statement handler. After checking that stack space is available (625EH) the line number operand is collected and placed in register pair DE (4769H). The seven byte parameter block is then pushed onto the stack and control transfers to the “GOTO” handler (47EBH). The parameter block is made up of the following:

```
2 bytes ... End of statement address
2 bytes ... Current line number
2 bytes ... 0000H
1 byte  ... GOSUB token (8DH)
```

The parameter block remains on the stack until a “RETURN” statement is executed. It is then used to determine the original program text position after which it is discarded.

Address... 47CFH
This routine is used by the Runloop interrupt processor (6389H) to create a “GOSUB” type parameter block on the Z80 stack. An interrupt block is identical to a normal block except that the two zero bytes shown above are replaced by the address of the device’s entry in TRPTBL. This address will be used by the “RETURN” statement handler to update the device’s interrupt status once a subroutine has terminated. After pushing the parameter block control transfers to the Runloop to execute the program line whose address is supplied in register pair DE.

Address... 47E8H
This is the “GOTO” statement handler. The line number operand is collected (4769H) and placed in register pair HL. If it is a pointer control transfers immediately to the Runloop to begin execution at the new program text position. Otherwise the line number is compared with the current line number to determine the starting position for the program text search. If it is greater the search starts from the end of this line (4298H), if it is smaller it starts from the beginning of the Program Text Area (4295H). If the referenced line cannot be found an “Undefined line number” error is generated (481CH). Otherwise the line number operand is replaced by the referenced program line’s address and its token changed to the pointer type (5583H). Control then transfers to the Runloop to execute the referenced program line.

Address... 481CH
This is the “Undefined line number” error generator.

Address... 4821H
This is the “RETURN” statement handler. A dummy loop Variable address is placed in register pair DE and the Z80 stack searched (3FE2H) to find the first parameter block not belonging to a “FOR” loop, this section of stack is then discarded. If no “GOSUB” token (8DH) is found at this point a “RETURN without GOSUB” error is generated. The next two bytes are then taken from the block, if they are non-zero the block was generated by an interrupt and the

temporary “STOP” condition is removed (633EH). The program text is then examined, if anything follows the “RETURN” token itself it is assumed to be a line number operand and control transfers to the “GOTO” handler (47E8H). Otherwise the old line number and program text address are taken from the block and control returns to the Runloop.

Address... 485BH
This is the “DATA” statement handler. The program text is skipped over until a statement separator (3AH) or end of line character (00H) is found. This routine is also the “REM” and “ELSE” statement handler via the entry point at 485DH, in this case only the end of line character acts as a terminator.

Address... 4880H
This is the “LET” statement handler. The Variable is first located (5EA4H), its address saved in TEMP and the operand evaluated (4C64H). If necessary the operand’s type is then changed to match that of the Variable (517AH). Assuming the operand is one of the three numeric types it is simply copied from DAC to the Variable in the Variable Storage Area (2EF3H). If the operand is a string type the address of the string body is taken from the descriptor and checked. If it is in KBUF, as would be the case for an explicit string in a direct statement, the body is first copied to the String Storage Area and a new descriptor created (6611H). The descriptor is then freed from TEMPST (67EEH) and copied to the Variable in the Variable Storage Area (2EF3H).

Address... 48E4H
This is the “ON ERROR”, “ON DEVICE GOSUB” and “ON EXPRESSION” statement handler. If the next program text character is not an “ERROR” token (A6H) control transfers to the “ON DEVICE GOSUB” and “ON EXPRESSION” handler (490DH). The program text is checked to ensure that a “GOTO” token (89H) follows and then the line number operand collected (4769H). The program text is searched to obtain the address of the referenced line (4293H) and this is placed in ONELIN. If the line number operand is non-zero the routine then terminates. If the line number operand is zero ONEFLG is checked to see if an error situation already exists (implying that the statement is inside a BASIC error recovery routine). If so control transfers to the error handler (4096H) to force an immediate error, otherwise the routine terminates normally.

Address... 490DH
This is the “ON DEVICE GOSUB” and “ON EXPRESSION” statement handler. If the next program text character is not a device token (7810H) control transfers to the “ON EXPRESSION” handler (4943H). After checking the program text for a “GOSUB” token (8DH) each of the line number operands required for a particular device is collected in turn (4769H). Assuming a given line number operand is non-zero the program text is searched to find the address of the referenced line (4293H) and this is placed in the device’s entry in TRPTBL (785CH). The routine terminates when no more line number operands are found.

Address... 4943H
This is the “ON EXPRESSION” statement handler. The operand is evaluated (521CH) and the following “GOSUB” token (8DH) or “GOTO” token (89H) placed in register A. The operand is then used to count along the program text until register pair HL points to the required line number operand. Control then transfers back to the Runloop execution point (4646H) to decode the “GOSUB” or “GOTO” token.

Address... 495DH
This is the “RESUME” statement handler. ONEFLG is first checked to make sure that an error condition already exists, if not a “RESUME without error” is generated (4064H). If a non-zero line number operand follows control transfers to the “GOTO” handler (47EBH). If a “NEXT” token (83H) follows the position of the error is restored from ERRTXT and ERRLIN, the start of the next statement is found (485BH) and the routine terminates. If there is no line number operand or if it is zero the position of the error is found from ERRTXT and ERRLIN and the routine terminates.

Address... 49AAH
This is the “ERROR” statement handler. The operand is evaluated and placed in register E (521CH). If it is zero an “Illegal function call” error is generated (475AH), otherwise control transfers to the error handler (406FH).

Address... 49B5H
This is the “AUTO” Statement handler. The optional start and increment line number operands, both with a default value of ten, are collected (475FH) and placed in AUTLIN and AUTINC. After making AUTFLG non-zero the Runloop return is destroyed and control transfers directly to the Mainloop (4134H).

Address... 49E5H
This is the “IF” statement handler. The operand is evaluated (4C64H) and, after checking for a “GOTO” token (89H) or “THEN” token (DAH), its sign is tested (2EA1H). If the operand is non-zero (true) the following text is executed either

by an immediate transfer to the Runloop (4646H) or, for a line number operand, the “GOTO” handler (47E8H). If the operand is zero (false) the statement text is scanned (485BH) until an “ELSE” token (A1H) is found not balanced by an “IF” token (8BH) and execution re-commences.

Address... 4A1DH
This is the “LPRINT” statement handler. PRTFLG is set to 01H, to direct output to the printer, and control transfers to the “PRINT” handler (4A29H).

Address... 4A24H
This is the “PRINT” statement handler. The program text is first checked for a trailing buffer number and, if necessary, PTRFIL set to direct output to the required I/O buffer (6D57H). If no more program text exists a CR,LF is issued (7328H) and the routine terminates (4AFFH). Otherwise successive characters are taken from the program text and analyzed. If a “USING” token (E4H) is found control transfers to the “PRINT USING” handler (60B1H). If a “;” character is found control just transfers back to the start to fetch the next item (4A2EH). If a comma is found sufficient spaces are issued to bring the current print position, from TTYPOS, LPTPOS or an I/O buffer FCB, to an integral multiple of fourteen. If output is directed to the screen and the print position is equal to or greater than the contents of CLMLST or if output is directed to the printer and it is equal to or greater than 238 then a CR,LF is issued instead (7328H). If a “SPC(“ token (DFH) is found the operand is evaluated (521BH) and the required number of spaces are output. If a “TAB(“ token (DBH) is found the operand is evaluated (521BH) and sufficient spaces issued to bring the current print position, from TTYPOS, LPTPOS or an I/O buffer FCB, to the required point. If none of these characters is found the program text contains a data item which is then evaluated (4C64H). If the operand is a string it is simply displayed (667BH). If it is numeric it is first converted to text in FBUFFR (3425H) and a string descriptor created (6635H). If output is directed to an I/O buffer the resulting string is then displayed (667BH). If output is directed to the screen or printer the current print position, from TTYPOS or LPTPOS, is compared with the line length and a CR,LF issued (7328H) if the output will not fit on the line. The maximum line length is 255 for the printer and is taken from LINLEN for the screen. Once the string has been displayed control transfers back to the start of the handler.

Address... 4AFFH
This routine zeroes PRTFLG and PTRFIL to return the Interpreter’s output to the screen.

Address... 4B0EH
This is the “LINE INPUT”, “LINE INPUT#” and “LINE” statement handler. If the following program text character is anything other than an “INPUT” token (85H) control transfers to the “LINE” statement handler (58A7H). If the following program text character is a “#” (23H) control transfers to the “LINE INPUT#” statement handler (6D8FH). Any following prompt string is evaluated and displayed (4B7BH) and the Variable located (5EA4H) and checked to ensure that it is a string type (3058H). The line of text is collected from the console via the INLIN standard routine, if Flag C (CTRL-STOP) is returned control transfers to the “STOP” statement handler (63FEH). Otherwise the input string is analyzed and a descriptor created (6638H), control then transfers to the “LET” statement handler for assignment (4892H). It should be noted that the screen is not forced to text mode before the input is collected.

Address... 4B3AH
This is the plain text message “?Redo from start”, CR, LF terminated by a zero byte.

Address... 4B4DH
This routine is used by the “READ/INPUT” statement handler if it has failed to convert a data item to numeric form. If in “READ” mode (FLGINP is non-zero) a “Syntax error” is generated (404FH). Otherwise the message “?Redo from start” is displayed (6678H) and control returns to the statement handler.

Address... 4B62H
This is the “INPUT#” Statement handler. The buffer number is evaluated and PTRFIL set to direct input from the required I/O buffer (6D55H), control then transfers to the combined “READ/INPUT” statement handler (4B9BH).

Address... 4B6CH
This is the “INPUT” statement handler. If the next program text character is a “#” control transfers to the “INPUT#” statement handler (4B62H). Otherwise the screen is forced to text mode, via the TOTXT standard routine, and any prompt string analyzed (6636H) and displayed (667BH). A question mark is then displayed and a line of text collected from the console via the QINLIN standard routine. If this returns Flag C (CTRL-STOP) control transfers to the “STOP” handler (63FEH). If the first character in BUF is zero (null input) the handler terminates by skipping to the end of the statement (485AH), otherwise control drops into the combined “READ/INPUT” handler.

Address... 4B9FH
This is the “READ” statement handler, a large section is also used by the “INPUT” and “INPUT#” statements so the structure is rather awkward. Each Variable found in the program text is located in turn (5EA4H), for each one the corresponding data item is obtained and assigned to the Variable by the “LET” handler (4893H). When in “READ” mode the data items are taken from the program text using the initial contents of DATPTR (4C40H). When in “INPUT” or “INPUT#” mode the data items are taken from the text buffer BUF. If the data items are exhausted in “READ” mode an “Out of DATA” error is generated. If they are exhausted in “INPUT” mode two question marks are displayed and another line fetched from the console via the QINLIN standard routine. If they are exhausted in “INPUT#” mode another line of text is copied to BUF from the relevant I/O buffer (6D83H). If the Variable list is exhausted while in “INPUT” mode the message “Extra ignored” is displayed (6678H) and the handler terminates (4AFFH). In “INPUT#” mode no message is displayed while in “READ” mode control terminates by updating DATPTR (63DEH). If a data item cannot be converted to numeric form (3299H) to match a numeric Variable control transfers to the “?Redo from start” routine (4B4DH).

Address... 4C2FH
The is the plain text message “?Extra ignored”, CR, LF terminated by a zero byte.

Address... 4C40H
This routine is used by the “READ” handler to locate the next “DATA” statement in the program text, the address to start from is supplied in register pair HL. Each program statement is examined until a “DATA” token (84H) is found whereupon the routine terminates (4BD1H). If the end link is reached an “Out of DATA” error is generated. As the search proceeds the line number of each program line is placed in DATLIN for use by the error handler.

Address... 4C5FH
This routine checks that the next character in the program text is the “=” token (EFH) and then drops into the Expression Evaluator. When entered at 4C62H it checks for “(“.

Address... 4C64H
This is the Expression Evaluator. On entry register pair HL points to the first character of the expression to be evaluated. On exit register pair HL points to the character following the expression, the result is in DAC and the type code in VALTYP. For a string result the address of the string descriptor is returned at DAC+2. The descriptor itself comprising a single byte for the string length and two bytes for its address, will be in TEMPST or inside a string Variable. An expression is a list of factors (4DC7H) linked together by operators with differing precedence levels. To process such an expression correctly the Expression Evaluator must be able to temporarily stack an intermediate result, if the next operator has a higher precedence than the current operator, and start afresh on a new calculation. It therefore has two basic operations, STACK and APPLY. For example: `3+250\2^2*3^3+1,`

```
STACK:    3+         (\ follows)
STACK:    250\       ( follows)
APPLY:    2^2=4      (* follows)
STACK:    4*         ( follows)
APPLY:    3^3=27     (+ follows)
APPLY:    4*27=108   (+ follows)
APPLY:    250\108=2  (+ follows)
APPLY:    3+2=5      (+ follows)
APPLY:    5+1=6      (, follows)
```

Evaluation terminates when the next operator has a precedenceequal to or lower than the initial precedence and the stack is empty. The expression delimiter, shown as a comma in the example, is regarded as having a precedence of zero and so will always halt evaluation. Normally the Expression Evaluator starts off with an initial precedence of zero but the entry point at 4C67H may be used to supply an alternative value in register D. This facility is used by the Factor Evaluator to restrict the range of evaluation when applying the monadic negation and “NOT” operators.

Address... 4D22H
This routine is used by the Expression Evaluator to apply an infix math operator (+-*/) to a pair of numeric operands. There are separate routines for the relational operators (4F57H) and the logical operators (4F78H). The first operand, its type code, and the operator token are supplied on the Z80 stack, the second operand and its type code are supplied in DAC and VALTYP. The types of both operands are first compared, if they differ the lowest precision operand is converted to match the higher. The operands are then moved to the positions required by the math routines. For integers the first operand is placed in register pair DE and the second in register pair HL. For single precision the first operand is placed in registers C, B, E, D and the second in DAC. For double precision the first operand is placed in DAC and the second in ARG. The operator token is then used to obtain the required address from the table at 3D51H, 3D5DH or 3D69H, depending upon the operand type, and control transfers to the relevant math routine.

Address... 4DB8H
This routine is used by the Expression Evaluator to divide two integer operands. The first operand is contained in register pair DE and the second in register pair HL, the result is returned in DAC. Both operands are converted to single precision (2FCBH) and control transfers to the single precision division routine (3265H).

Address... 4DC7H
This is the Factor Evaluator. On entry register pair HL points to the character before the factor to be evaluated. On exit register pair HL points to the character following the factor, the result is in DAC and the type code in VALTYP. A factor may be one of the following:
(1) A numeric or string constant
(2) A numeric or string Variable
(3) A function
(4) A monadic operator (+-NOT)
(5) A parenthesized expression
The first character is taken from the program text via the CHRGTR standard routine and examined. If it is an end of Statement character a "Missing operand" error is generated (406AH). If it is an ASCII digit it is converted from textual form to one of the standard numeric types in DAC (3299H).If it is upper case alphabetic (64A8H) it is a Variable and its current value is returned (4E9BH). If it is a numeric token the number is copied from CONLO to DAC (46B8H). If it is one of the FFH prefixed function tokens shown in the table at 39DEH it is decoded to transfer control to the relevant function handler (4ECFH). If it is the monadic "+" operator it is simply skipped over, only the monadic "-" operator (4E8DH) and monadic "NOT" operator (4F63H) require any action.If it is an opening quote the following explicit string is analyzed and a descriptor created (6636H). If it is an "&" it is a non-decimal numeric constant and it is converted to one of the standard numeric types in DAC (4EB8H). If it is not one of the functions shown below then it must be a parenthesized expression (4E87H), otherwise a "Syntax error" is generated. The following function tokens are tested for directly and control transferred to the address shown:

ERR	4DFDH
ERL	4E0BH
POINT	5803H
TIME	7900H
SPRITE	7A84H
VDP	7B47H
BASE	7BCBH
PLAY	791BH
DSKI$	7C3EH

ATTR$	7C43H
VARPTR	4E41H
USR	4FD5H
INSTR	68EBH
INKEY$	7347H
STRING$	6829H
INPUT$	6C87H
CSRLIN	790AH
FN	5040H

Address... 4DFDH
This routine is used by the Factor Evaluator to apply the "ERR" function. The contents of ERRFLG are placed in DAC as an integer (4FCFH).

Address... 4E0BH
This routine is used by the Factor Evaluator to apply the "ERL" function. The contents of ERRLIN are copied to DAC as a single precision number (3236H).

Address... 4E41H
This routine is used by the Factor Evaluator to apply the "VARPTR" function. If the function token is followed by a "#" the buffer number is evaluated (521BH), the I/O buffer FCB located (6A6DH) and its address placed in DAC as an integer (2F99H). Otherwise the Variable is located (5F5DH) and its address placed in DAC as an integer (2F99H).

Address... 4E8DH
This routine is used by the Factor Evaluator to apply the monadic "-" operator. Register D is set to a precedence value of 7DH, the factor evaluated (4C67H) and then negated (2E86H).

Address... 4E9BH
This routine is used by the Factor Evaluator to return the current value of a Variable. The Variable is first located (5EA4H). If it is a string Variable its address is placed in DAC to point to the descriptor. Otherwise the contents of the Variable are copied to DAC (2F08).

Address... 4EA9H
This routine returns the single character pointed to by register pair HL in register A, if it is a lower case alphabetic it converts it to upper case.

Address... 4EB8H
This routine is used by the Factor Evaluator and the numeric input routine (3299H) to convert an ampersand ("&") Prefixed number from textual form to an integer in DAC. As each legal character is found the product is multiplied by

2, 8 or 16, depending upon the character which initially followed the ampersand, and the new digit added to it. If the product overflows an "Overflow" error is generated (4067H). The routine terminates when an unacceptable character is found.

Address... 4EFCH
This routine is used by the Factor Evaluator to process the FFH prefixed function tokens. If the token is either "LEFT$", "RIGHT$" or "MID$" the string operand is evaluated (4C62H), the address of its descriptor pushed onto the Z80 stack and the following numeric operand also evaluated (521CH) and stacked. Otherwise the function's parenthesized operand is evaluated (4E87H) and, for "SQR", "RND", "SIN", "LOG", "EXP", "COS", "TAN" or "ATN" only, converted to double precision (303AH). The function token is then used to obtain the required address from the table at 39DEH and control transfers to the function handler.

Address... 4F47H
This routine is used by the numeric input conversion routine (3299H) to test for a "+" or "-" character or token. It returns register D=0 for positive and register D=FFH for negative.

Address... 4F57H
This routine is used by the Expression Evaluator to apply arelational operator (<>= or combinations) to a pair of operands. If the operands are numeric the Expression Evaluator first uses the math operator routine (4D22H) to apply the general relation operation to the operands. If the operands are strings the string comparison routine (65C8H) is used first. When control arrives here the relation result is in register A and the Z80 Flags:

```
Operand 1=Operand 2 ... A=00H, Flag Z,NC
Operand 1<Operand 2 ... A=01H, Flag NZ,NC
Operand 1>Operand 2 ... A=FFH, Flag NZ,C
```

The Expression Evaluator also supplies a bit mask defining the original operators on the Z80 stack. This has a 1 in each position if the associated operation is required: 00000<=>. The mask is applied to the relation result producing zero if none of the conditions is satisfied. This is then placed in DAC as a true (-1) or false (0) integer (2E9AH).

Address... 4F63H
This routine is used by the Factor Evaluator to apply the monadic "NOT" operator. Register D is set to an initial precedence level of 5AH and the expression evaluated (4C67H) and converted to an integer (2F8AH). It is then inverted and restored to DAC.

Address... 4F78H
This routine is used by the Expression Evaluator to apply a logical operator ("OR", "AND", "XOR", "EQV" and "IMP") or the "MOD" and "\" operators to a pair of numeric operands. The first operand, which has already been converted to an integer, is supplied on the Z80 stack and the second is supplied in DAC. The operator token (actually its precedence level) is supplied in register B. After converting the second operand to an integer (2F8AH) the operator is examined. There are separate routines for "MOD" (323AH) and "\" (31E6H) but the logical operators are processed locally using the corresponding Z80 logical instructions on register pairs DE and HL. The result is stored in DAC as an integer (2F99H).

Address... 4FC7H
This routine is used by the Factor Evaluator to apply the "LPOS" function to an operand contained in DAC. The contents of LPTPOS are placed in DAC as an integer (4FCFH).

Address... 4FCCH
This routine is used by the Factor Evaluator to apply the "POS" function to an operand contained in DAC. The contents of TTYPOS are placed in DAC as an integer (2F99).

Address... 4FD5H
This routine is used by the Factor Evaluator to apply the "USR" function. The user number is collected directly from the program text, it cannot be an expression, and the associated address taken from USRTAB (4FF4H). The following parenthesized operand is then evaluated (4E87H) and left in DAC as the passed parameter. If it is a string type its storage is freed (67D3H). The current program text position is pushed onto the Z80 stack followed by a return to 3297H, the routine at this address will restore the program text position after the user function has terminated. Control then transfers to the user address with register pair HL pointing to the first byte of DAC and the type code, from VALTYP, in register A. Additionally, for a string parameter, the descriptor address is taken from DAC and placed in register pair DE. The user routine may modify any register except the Z80 SP and should terminate with a RET instruction, interrupts may be left disabled if necessary as the Runloop will re-enable them. Any numeric parameter to

be returned to the Interpreter should be placed in DAC. Strictly speaking this should be the same numeric type as the passed parameter, however if VALTYP is modified the Interpreter will always accept it. Returning a string type is more difficult. Using the same method as the Factor Evaluator string functions, which involves copying the string to the String Storage Area and pushing a new descriptor onto TEMPST, is complicated and vulnerable to changes in the MSX system. A simpler and more reliable method is to use the passed parameter to create the space for the result. This should not be an explicitly stated string as the program text will have to be modified, instead an implicit parameter should be used. This must be done with care however, it is very easy to gain the impression that the Interpreter has accepted the string when in fact it has not. Take the following example which does nothing but return the passed parameter:

```
10 POKE &H9000,&HC9
20 DEFUSR=&H9000
30 A$=USR(STRING$(12,"!"))
40 PRINT A$
50 B$=STRING$(9,"X")
60 PRINT A$
```

At first it seems that the passed string has been correctly assigned to A$. When line 60 is reached however it becomes apparent that A$ has been corrupted by the subsequent assignment of a string to B$. What has happened is that the temporary storage allocated to the passed parameter was reclaimed from the String Storage Area before control transferred to the user routine. This region was then used to store the string belonging to B$ thus modifying A$. This situation can be avoided by assigning the parameter to a Variable beforehand and then passing the Variable, for example:

```
10 A$=STRING$(12,"!")
20 A$=USR(A$)
```

Line 10 results in twelve bytes of the String Storage Area being permanently allocated to A$. When the user function is entered the descriptor, which is pointed to by register pair DE, will contain the starting address of the twelve byte region where the result should be placed. If the returned string is shorter than the passed one the length byte of the descriptor may be changed without any side effects. For further details on string storage see the garbage collector (66B6H). A point worth noting is that a "CLEAR" operation is not strictly necessary before a machine language program is loaded. The region between the top of the Array Storage Area and the base of the Z80 stack is never used by the Interpreter. A program can exist in this region provided that the two enclosing areas do not overlap it.

Address... 500EH
This is the "DEFUSR" statement handler. The user number is collected directly from the program text, it cannot be an expression, and the associated entry in USRTAB located (4FF4H). The address operand is then evaluated (542FH) and placed in USRTAB.

Address... 501DH
This is the "DEF FN" and "DEFUSR" statement handler. If the following character is a "USR" token (DDH) control transfers to the "DEFUSR" statement handler (500EH), otherwise the program text is checked for a trailing "FN" token (DEH). The function name Variable is located (51A1H) and, after checking that the Interpreter is in program mode (5193H), the current program text position is placed there. Each of the Variables in the formal parameter list is then located in succession (5EA4H), this is simply to ensure that they are created. The routine terminates by skipping over the remainder of the statement (485BH) as the function body is not required at this time.

Address... 5040H
This routine is used by the Factor Evaluator to apply the "FN" function. The function name Variable is first located (51A1H) to obtain the address of the function definition in the program text. Each formal Variable from the function definition is located in turn (5EA4H) and its address pushed onto the Z80 stack. As each one is found the corresponding actual parameter is evaluated (4C64H) and pushed onto the stack with it. If necessary the type of the actual parameter is converted to match that of the formal parameter (517AH)' When both lists are exhausted each formal Variable address and actual parameter are popped from the stack in turn. Each Variable is then copied from the Variable Storage Area to PARM2 with its value replaced by the actual parameter. It should be noted that, because PARM2 is only a hundred bytes long, a maximum of nine double precision parameters is allowed. When all the actual parameters have been copied to PRM2 the entire contents of PARM1 (the current parameter area) are pushed onto the Z80 stack and PARM2 is copied to PARM1 (518EH). Register pair HL is then set to the start of the function body in the program text and the expression is evaluated (4C5FH). The old contents of PARM1 are popped from the stack and restored. Finally the result of the evaluation is type converted if necessary to match the function name type (517AH). A user defined function differs from a normal expression in only one respect, it has its own set of local Variables. These Variables are created in PARM1 when the function is invoked and disappear when it terminates. When a normal Variable search is initiated by the Expression Evaluator the region examined is the Variable Storage Area. However, if NOFUNS is non-zero, indicating at least one active user function, PARM1 will be searched instead, only if this fails will the search move

on to the global Variables in the Variable Storage Area. Using a local Variable area specific to each invocation of a function means that the same Variable names can be used throughout without the Variables overwriting each other or the global Variables. It is worth noting that a user defined function is slower than an inline expression or even a subroutine. The search carried out to find the function name Variable, plus the large amount of stacking and destacking, are significant overheads.

Address... 5189H
This routine moves a block of memory from the address pointed to by register pair DE to that pointed to by register pair HL, register pair BC defines the length.

Address... 5193H
This routine generate an "Illegal direct" error if CURLIN shows the Interpreter to be in direct mode.

Address... 51A1H
This routine checks the program text for an "FN" token (DEH) and then creates the function name Variable (5EA9H). These are distinguished from ordinary Variables by having bit 7 set in the first character of the Variable's name.

Address... 51ADH
Control transfers to this routine from the Runloop execution point (4640H) if a token greater than D8H is found at the start of a statement. If the token is not an FFH prefixed function token a "Syntax error" is generated (4055H). If the function token is one of those which double as statements control transfers to the relevant handler, otherwise a "Syntax error" is generated. The statements in question are "MID$" (696EH), "STRIG" (77BFH) and "INTERVAL" (77B1H). There is actually no separate token for "INTERVAL", the "INT" token (85H) suffices with the remaining characters being checked by the statement handler.

Address... 51C9H
This is the "WIDTH" statement handler. The operand is evaluated (521CH) and its magnitude checked. If it is zero or greater than thirty-two or forty, depending upon the screen mode held in OLDSCR an "Illegal function call" error is generated (475AH). If it is the same as the current contents of LINLEN the routine terminates with no further action. Otherwise the current screen is cleared with a FORMFEED control code (0CH) via the OUTDO standard routine in case the screen is to be made smaller. The operand is then placed in LINLEN and either LINL32 or LINL40, depending upon the screen mode held in OLDSCR, and the screen cleared again in case it has been made larger. Because the line length variable to be changed is selected by OLDSCR, rather than SCRMOD, the width can still be changed even if the screen is currently in Graphics Mode or Multicolour Mode. In this case the change is effective when a return is made to the Interpreter Mainloop or an "INPUT" statement is executed.

Address... 520EH
This routine evaluates the next expression in the program text (4C64H), converts it to an integer (2F8AH) and places the result in register pair DE. The magnitude and sign of the MSB are then tested and the routine terminates.

Address... 521BH
This routine evaluates the next operand in the program text (4C64H) and converts it to an integer (5212H). If the operand is greater than 255 an "Illegal function call" error is generated (475AH).

Address... 5229H
This is the "LLIST" statement handler. PRTFLG is set to 01H, to direct output to the printer, and control drops into the "LIST" statement handler.

Address... 522EH
This is the "LIST" statement handler. The optional start and termination line number operands are collected and the starting position found in the program text (4279H). Successive program lines are listed until the end link is found, the CTRL-STOP key is pressed or the termination line number is reached, control then transfers directly to the Mainloop "OK" point (411FH). Each program line is listed by displaying its line number (3412H), detokenizing (5284H) and displaying (527BH) the line itself and then issuing a CR,LF (7328H).

Address... 5284H
This routine is used by the "LIST" statement handler to convert a tokenized program line to textual form. On entry register pair HL points to the first character of the tokenized line. On exit the line of text is in BUF and is terminated by a zero byte. Any normal or FFH prefixed token is converted to the corresponding keyword by a simple linear search of the tokens in the table at 3A72H. Exceptions are made if either an opening quote character, a "REM" token, or a "DATA" token has previously been found. Normally these tokens will be followed by plain text anyway, the check is made to stop graphics characters being interpreted as tokens. The three byte sequence ":" (3AH), "REM" token (8FH), "

" token (E6H) is converted to the single " " character (27H) and the statement separator (3AH) preceding an "ELSE" token (A1H) is scrubbed out. If one of the numeric tokens is found its value and type are first copied from CONLO and CONTYP to DAC and VALTYP (46E8H). It is then converted to textual form in FBUFFR by the decimal (3425H), octal (371EH) or hex (3722H) conversion routines. For octal and hex types the number is prefixed by an ampersand and an "O" or "H" letter. A type suffix, "'" or "#", is added to single precision or double precision numbers only if there is no decimal part and no exponent part ("E" or "D").

Address... 53E2H
This is the "DELETE" statement handler. The optional start and termination line number operands are collected and the starting position found in the program text (4279H). If any pointers exist in the program text they are converted back to line numbers (54EAH). The terminating program line is found by a search of the program text (4295H), if this address is smaller than that of the starting program line an "Illegal function call" error is generated (475AH), otherwise the message "OK" is displayed (6678H). The block of memory from the end of the terminating line to the start of the Variable Storage Area is copied down to the beginning of the starting line and VARTAB, ARYTAB and STREND are reset to the new (lower) end of the program text. Control then transfers directly to the end of the Mainloop (4237H) to reset the remaining pointers and to relink the Program Text Area. Note that, because control does not return to the normal "OK" point, the screen will not be returned to text mode. If the screen is in Graphics Mode or Multicolour mode when a "DELETE" is executed, which is admittedly rather unlikely, the system will crash.

Address... 541CH
This routine is used by the Factor Evaluator to apply the "PEEK" function to an operand contained in DAC. The address operand is checked (5439H) then the byte read from memory and placed in DAC as an integer (4FCFH).

Address... 5423H
This is the "POKE" statement handler. The address operand is evaluated (542FH) then the data operand evaluated (521CH) and written to memory.

Address... 542FH
This routine evaluates the next operand in the program text (4C64H) and places it in register pair DE as an integer (5439H).

Address... 5439H
This routine converts the numeric operand contained in DAC into an integer in register pair HL. The operand must be in the range -32768 to +65535 and is normally an address as required by "POKE", "PEEK", "BLOAD", etc. The operand's type is first checked via the GETYPR standard routine, if it is already an integer it is simply placed in register pair HL (2F8AH). Assuming the operand is single precision or double precision its sign is checked, if it is negative it is converted to integer (2F8AH). Otherwise it is converted to single precision (2FB2H) and its magnitude checked (2F21H). If it is greater than 32767 and smaller than 65536 then -65536 is added (324EH) before it is converted to integer (2F8AH).

Address... 5468H
This is the "RENUM" statement handler. If a new initial line number operand exists it is collected (475FH), otherwise a default value of ten of taken. If an old initial line number operand exists it is collected (475FH), otherwise a default value of zero is taken. If an increment line number operand exists it is collected (4769H), otherwise a default value of ten is taken. The program text is then searched for existing line numbers equal to or greater than the new initial line number (4295H) and the old initial line number (4295H), an "Illegal function call" error is generated (475AH) if the new address is smaller than the old address. This is to catch any attempt to renumber high program lines down to existing low ones. A dummy renumbering run of the program text is first carried out to check than no new line number will be generated with a value greater than 65529. This must be done as an error midway through the conversion would leave the program text in a confused state. Assuming all is well any line number operands in the program text are converted to pointers (54F6H). This neatly solves the problem of line number references, GOTO 50 for example, as the program text is not moved during renumbering. Starting at the old initial program text position each existing program line number is replaced with its new value. When the end link is reached any program text pointers are converted back to line number operands (54F1H) and control transfers directly to the Mainloop "OK" point (411EH).

Address... 54F6H
When entered at 54F6H this routine converts every line number operand in the program text to a pointer. When entered at 54F7H it performs the reverse operation and converts every pointer in the program text back to a line number operand. Starting at the beginning of the Program Text Area each line is examined for a pointer token (0DH) or a line number operand token (0EH) depending upon the mode. In pointer to line number operand mode the pointer is replaced by the line number from the referenced program line and the token changed to 0EH. In line number operand to pointer mode the program text is searched (4295H) to find the relevant line, its address replaces the line number operand and

the token is changed to 0DH. If the search is unsuccessful a message of the form “Undefined line NNNN in NNNN” is displayed (6678H) and the conversion process continues. A special check is made for the “ON ERROR GOTO 0” statement, to prevent the generation of a spurious error message, but no check is made for the similar “RESUME 0” statement. In this case an error message will be displayed, this should be ignored.

Address... 555AH
This is the plain text message “Undefined line “ terminated by a zero byte.

Address... 558CH
Name..... SYNCHR
Entry..... HL points to character to check
Exit..... A=Next program character
Modifies.. AF, HL
Standard routine to check the current program text character, whose address is supplied in register pair HL, against a reference character. The reference character is supplied as a single byte immediately following the CALL or RST instruction, for example:

```
RST 08H
DEFB “,”
```

If the characters do not match a “Syntax error” is generated (4055H), otherwise control transfers to the CHRGTR standard routine to fetch the next program character (4666H).

Address... 5597H
Name..... GETYPR
Entry..... None
Exit..... AF=Type
Modifies.. AF
Standard routine to return the type of the current operand, determined by VALTYP, as follows:

```
Integer.............A=FFH, Flag M,NZ,C
String..............A=00H, Flag P,Z,C
Single Precision ... A=01H, Flag P,NZ,C
Double Precision ... A=05H, Flag P,NZ,NC
```

Address... 55A8H
This is the “CALL” statement handler. The extended statement name, which is an unquoted string up to fifteen characters long terminated by a “(“, “:” or end of line character (00H), is first copied from the program text to PROCNM, any unused bytes are zero filled. Bit 5 of each entry in SLTATR is then examined for an extension ROM with a statement handler. If a suitable ROM is found its position in SLTATR is converted to a Slot ID in register A and a ROM base address in register H (7E2AH). The statement handler address is read from ROM locations four and five (7E1AH) and placed in register pair IX. The Slot ID is placed in the high byte of register pair IY and the ROM statement handler called via the CALSLT standard routine. The ROM will examine the statement name and return Flag C if it does not recognize it, otherwise it performs the required operation. If the ROM call fails the search of SLTATR continues until the table is exhausted whereupon a “Syntax error” is generated (4055H). If the ROM call is successful the handler terminates.

Address... 55F8H
This routine is used by the device name parser (6F15H) when it cannot recognize a device name found in the program text. Upon entry register pair HL points to the first character of the name and register B holds its length. The name is first copied to PROCNM and terminated by a zero byte. Bit 6 of each entry in SLTATR is then examined for an extension ROM with a device handler. If a suitable ROM is found its position in SLTATR is converted to a Slot ID in register A and a ROM base address in register H (7E2AH). The device handler address is read from ROM locations six and seven (7E1AH) and placed in register pair IX. The Slot ID is placed in the high byte of register pair IY, the unknown device code (FFH) in register A and the ROM device handler called via the CALSLT standard routine. The ROM will examine the device name and return Flag C if it does not recognize it, otherwise it returns its own internal code from zero to three. If the ROM call fails the search of SLTATR continues until the table is exhausted whereupon a “Bad file name” error is generated (6E6BH). If the ROM call is successful the ROM’s internal code is added to its SLTATR position, multiplied by a factor of four, to produce a global device code’ The base code for each entry in SLTATR is shown below in hexadecimal. The “SS” and “PS” markers show the corresponding Secondary and Primary Slot numbers, each slot is composed of four pages:

SS0	SS1	SS2	SS3	
00 04 08 0C	10 14 18 1C	20 24 28 2C	30 34 38 3C	PS0
40 44 48 4C	50 54 58 5C	60 64 68 6C	70 74 78 7C	PS1
80 84 88 8C	90 94 98 9C	A0 A4 A8 AC	B0 B4 B8 BC	PS2
C0 C4 C8 CC	D0 D4 D8 DC	E0 E4 E8 EC	F0 F4 F8 FC	PS3

Figure 44: Device Codes.

The global device code is used by the Interpreter until the time comes for the ROM to perform an actual device operation. It is then converted back into the ROM's Slot ID, base address and internal device code to perform the ROM access. Note that the codes from 0 to 8 are reserved for disk drive identifiers and those from FCH to FFH for the standard devices GRP, CRT, LPT and CAS. With the current MSX hardware structure these codes correspond to physically improbable ROM configurations and are therefore safe to be used for specific purposes by the Interpreter.

Address... 564AH
This routine is used by the function dispatcher (6F8FH) when it encounters a device code not belonging to one of the standard devices. The device code is first converted to a SLTATR position and then to a Slot ID in register A and ROM base address in register H (7E2DH). The ROM device handler address is read from ROM locations six and seven (7E1AH) and placed in register pair IX. The Slot ID is placed in the high byte of register pair IY, the ROM's internal device code in DEVICE and the ROM device handler called via the CALSLT standard routine.

Address... 566CH
This entry point to the macro language parser is used by the "DRAW" statement handler, a later entry point (56A2H) is used by the "PLAY" statement handler. The command string is evaluated (4C64H) and its storage freed (67D0H). After pushing a zero termination block onto the Z80 stack the length and address of the string body are placed in MCLLEN and MCLPTR and control drops into the parser mainloop.

Address... 56A2H
This is the macro language parser mainloop, it is used to process the command string associated with a "DRAW" or "PLAY" statement' On entry the string length is in MCLLEN, the string address is in MCLPTR and the address of the relevant command table is in MCLTAB. The command tables contain the legal command letters, together with the associated command handler addresses, for each statement. The "DRAW" table is at 5D83H and the "PLAY" table at 752EH. The parser mainloop first fetches the next character from the command string (56EEH). If there are no more characters left the next string descriptor is popped from the stack (568CH). If this is zero the parser terminates (5709H) if MCLFLG shows a "DRAW" statement to be active, otherwise control transfers back to the "PLAY" statement handler (7494H). Assuming a command character exists the current command table is searched to check its legality, if no match is found an "Illegal function call" error is generated (475AH). The command table entry is then examined, if bit 7 is set the command takes an optional numeric parameter. If this is present it is collected and placed in register pair DE (571CH), otherwise a default value of one is taken. After pushing a return to the start of the parser mainloop onto the Z80 stack control transfers to the command handler at the address taken from the command table.

Address... 56EEH
This routine is used by the macro language parser to fetch the next character from the command string. If MCLLEN is zero the routine terminates with Flag Z, there are no characters left. Otherwise the next character is taken from the address contained in MCLPTR and returned in register A, if the character is lower case it is converted to upper case. MCLPTR is then incremented and MCLLEN decremented.

Address... 570BH
This routine is used by the macro language parser to return an unwanted character to the command string. MCLLEN is incremented and MCLPTR decremented.

Address... 5719H
This routine is used by the macro language parser to collect a numeric parameter from the command string. The result is a signed integer and is returned in register pair DE, it cannot be an expression. The first character is taken and examined, if it is a "+" it is ignored and the next character taken (5719H). If it is a "-" a return is set up to the negation routine (5795H) and the next character taken (5719H). If it is an "=" the value of the following Variable is returned (577AH). Otherwise successive characters are taken and a binary product accumulated until a non-numeric character is found.

Address... 575AH
This routine is used by the macro language parser "=" and "X" handlers. The Variable name is copied to BUF until the ";" delimiter is found, if this takes more than thirty-nine characters to find an "Illegal function call" error is generated (475AH). Otherwise control transfers to the Factor Evaluator Variable handler (4E9BH) and the Variable contents are returned in DAC.

Address... 577AH
This routine is used by the macro language parser to process the "=" character in a command parameter. The Variable's value is obtained (575AH), converted to an integer (2F8AH) and placed in register pair DE.

Address... 5782H
This routine is used by the macro language parser to process the "X" command. The Variable is processed (575AH) and, after checking that stack space is available (625EH), the current contents of MCLLEN and MCLPTR are stacked. Control then transfers to the parser entry point (5679H) to obtain the Variable's descriptor and process the new command string.

Address... 579CH
This routine is used by various graphics statements to evaluate a coordinate pair in the program text. The coordinates must be parenthesized with a comma separating the component operands. If the coordinate pair is preceded by a "STEP" token (DCH) each component value is added to the corresponding component of the current graphics coordinates in GRPACX and GRPACY, otherwise the absolute values are returned. The X coordinate is returned in GRPACX, GXPOS and register pair BC. The Y coordinate is returned in GRPACY, GYPOS and register pair DE.There are two entry points to the routine, the one which is used depends on whether the caller is expecting more than one coordinate pair. The "LINE" statement, for example, expects two coordinate pairs the first of which is the more flexible. The entry point at 579CH is used to collect the first coordinate pair and will accept the characters "-" or "@-" as representing the current graphics coordinates. The entry point at 57ABH is used for the second coordinate pair and requires an explicit operand.

Address... 57E5H
This is the "PRESET" statement handler. The current background colour is taken from BAKCLR and control drops into the "PSET" handler.

Address... 57EAH
This is the "PSET" statement handler. After the coordinate pair has been evaluated (57ABH) the current foreground colour is taken from FORCLR and used as the default when setting the ink colour (5850H). The current graphics coordinates are converted to a physical address, via the SCALXY and MAPXYC standard routines, and the colour of the current pixel set via the SETC standard routine.

Address... 5803H
This routine is used by the Factor Evaluator to apply the "POINT" function. The current contents of CLOC, CMASK, GYPOS, GXPOS, GRPACY and GRPACX are stacked and the coordinate pair operand evaluated (57ABH). The colour of the new pixel is read via the SCALXY, MAPXYC and READC standard routines and placed in DAC as an integer (2F99H), the old coordinate values are then popped and restored. Note that a value of -1 is returned if the point coordinates are outside the screen.

Address... 5850H
This graphics routine is used to evaluate an optional colour operand in the program text and to make it the current ink colour. After checking the screen mode (59BCH) the colour operand is evaluated (521CH) and placed in ATRBYT. If no operand exists the colour code supplied in register A is placed in ATRBYT instead.

Address... 5871H
This graphics routine returns the difference between the
contents of GXPOS and register pair BC in register pair HL. If the result is negative (GXPOS<BC) it is negated to produce the absolute magnitude and Flag C is returned.

Address... 5883H
This graphics routine returns the difference between the contents of GYPOS and register pair DE in register pair HL. If the result is negative (GYPOS<DE) it is negated to produce the absolute magnitude and Flag C is returned.

Address... 588EH
This graphics routine swaps the contents of GYPOS and register pair DE.

Address... 5898H
This graphics routine first swaps the contents of GYPOS and register pair DE (588EH) then swaps the contents of GXPOS and register pair BC. When entered at 589BH only the second operation is performed.

Address... 58A7H
This is the "LINE" statement handler. The first coordinate pair (X1,Y1) is evaluated (579CH) and placed in register pairs BC,DE. After checking for the "-" token (F2H) the second coordinate pair (X2,Y2) is evaluated (57ABH) and left in GRPACX, GRPACY and GXPOS, GYPOS. After setting the ink colour (584DH) the program text is checked for a following "B" or "BF" option and either the box (5912H), boxfill (58BFH) or linedraw (58FCH) operation performed. None of these operations affects the current graphics coordinates in GRPACX and GRPACY, these are left at X2,Y2.

Address... 58BFH
This routine performs the boxfill operation. Given that the supplied coordinate pairs define diagonally opposed points of the box two quantities must be derived from them. The horizontal size of the box is obtained from the difference between X1 and X2, this gives the number of pixels to set per row. The vertical size is obtained from the difference between Y1 and Y2 giving the number of rows required. Starting at the physical address of X1,Y1, and moving successively lower via the DOWNC standard routine, the required number of pixel rows are filled in by repeated use of the NSETCX standard routine.

Address... 58FCH
This routine performs the linedraw operation. After drawing the line (593CH) GXPOS and GYPOS are reset to X2,Y2 from GRPACX and GRPACY.

Address... 5912H
This routine performs the box operation. The box is produced by drawing a line (58FCH) between each of the four corner points. The coordinates of each corner are derived from the initial operands by interchanging the relevant component of the pair. The drawing sequence is:

```
(1) X1,Y2 to X2,Y2
(2) X1,Y1 to X2,Y1
(3) X2,Y1 to X2,Y2
(4) X1,Y1 to X1,Y2
```

Address... 593CH
This routine draws a line between the points X1,Y1, supplied in register pairs BC and DE and X2,Y2, supplied in GXPOS and GYPOS. The operation of the drawing mainloop (5993H) is best illustrated by an example, say LINE(0,0)-(10,4). To reach the end point of the line from its start ten horizontal steps (X2-X1) and four downward steps (Y2-Y1) must be taken altogether. The best approximation to a straight line therefore requires two and a half horizontal steps for every downward step (X2-X1/Y2-Y1). While this is impossible in practice, as only integral steps can be taken, the correct ratio can be achieved on average. The method employed is to add the Y difference to a counter each time a rightward step is taken. When the counter exceeds the value of the X difference it is reset and one downward step is taken, this is in effect an integer division of the two difference values. Sometimes downward steps will be produced every two rightward steps and sometimes every three rightward steps. The average, however, will be one downward step every two and a half rightward steps. An equivalent BASIC program is shown below with a slightly offset BASIC line for comparison:

```
10 SCREEN 0
20 INPUT"START X,Y";X1,Y1
30 INPUT"END X,Y",X2,Y2
40 SCREEN 2
50 X=X1:Y=Y1:L=X2-X1:S=Y2-Y1:CTR=L/2 60 PSET(X,Y)
70 CTR=CTR+S:IF CTR<L THEN 90
80 CTR=CTR-L:Y=Y+1
90 X=X+1:IF X<=X2 THEN 60
100 LINE(X1,Y1+5)-(X2,Y2+5)
110 GOTO 110
```

The above example suffers from three limitations. The line must slope downwards, it must slope to the right and the slope cannot exceed forty-five degrees from the horizontal (one downward step for one rightward step). The routine overcomes the first limitation by examining the Y1 and Y2 coordinates before drawing commences. If Y2 is greater than or equal to Y1, showing the line to slope upwards or to be horizontal, both coordinate pairs are exchanged. The line is now sloping downwards and will be drawn from the end point to the start. The second limitation is overcome by examining X1 and X2 beforehand to determine which way the line is sloping. If X2 is greater than or equal to X1 the line slopes to the right and a Z80 JP to the RIGHTC standard routine is placed in MINUPD/MAXUPD (see below) for use by the drawing mainloop, otherwise a JP to the LEFTC standard routine is placed there. The third limitation is overcome by comparing the X coordinate difference to the Y coordinate difference before drawing to determine the slope steepness. If X2-X1 is smaller than Y2-Y1 the slope of the line is less than forty-five degrees from the horizontal. The simple method shown above for LINE(0,0)-(10,4) will not work for slopes greater than forty-five degrees as the maximum rate of descent is achieved when one downward step is taken for every horizontal step. It will work however if the step directions are exchanged. Thus LINE(0,0)-(4,10) requires one rightward step for every two and a half downward steps. MINUPD holds a Z80 JP to the "normal" step direction standard routine for the drawing mainloop and

MAXUPD holds a JP to the “slope” step direction standard routine. For shallow angles MINUPD will vector to DOWNC and MAXUPD to LEFTC or RIGHTC. For steep angles MINUPD will vector to LEFTC or RIGHTC and MAXUPD to DOWNC. For steep angles the counter values must also be exchanged, the X difference must now be added to the counter and the Y difference used as the counter limit. The variables MINDEL and MAXDEL are used by the drawing mainloop to hold these counter values, MINDEL holds the smaller end point difference and MAXDEL the larger. An interesting point is that the reference counter, held in CTR in the above program and in register pair DE in the ROM, is preloaded with half the largest end point difference rather than being set to zero. This has the effect of splitting the first “stair” in the line into two sections, one at the start of the line and one at its end, and improving the line’s appearance.

Address... 59B4H
This graphics routine shifts the contents of register pair DE one bit to the right.

Address... 59BCH
This routine generates an “Illegal function call” error (475AH) if the screen is not in Graphics Mode or Multicolour Mode.

Address... 59C5H
This is the “PAINT” statement handler. The starting coordinate pair is evaluated (579CH), the ink colour set (584DH) and the optional boundary colour operand evaluated (521CH) and placed in BDRATR. The starting coordinate pair is checked to ensure that it is within the screen (5E91H) and is made the current pixel physical address by the MAPXYC standard routine. The distance to the right hand boundary is then measured (5ADCH) and, if it is zero, the handler terminates. Otherwise the distance to the left hand boundary is measured (5AEDH) and the sum of the two placed in register pair DE as the zone width. The current position is then stacked twice (5ACEH), first with a termination flag (00H) and then with a down direction flag (40H). Control then transfers to the paint mainloop (5A26H) with an up direction flag (C0H) in register B.

Address... 5A26H
This is the paint mainloop. The zone width is held in register pair DE, the paint direction, up or down, in register B and the current pixel physical address is that of the pixel adjacent to the left hand boundary. A vertical step is taken to the next line, via the TUPC or TDOWNC standard routines, and the distance to the right hand boundary measured (5ADCH). The distance to the left hand boundary is then measured and the line between the boundaries filled in (5AEDH). If no change is found in the position of either boundary control transfers to the start of the mainloop to continue painting in the same direction. If a change is found an inflection has occurred and the appropriate action must be taken. There are four types of inflection, LH or RH incursive, where the relevant boundary moves inward, and LH or RH excursive, where it moves outward. An example of each type is shown below with numbered zones indicating the order of painting during upward movement. A secondary zone is shown within each inflective region for completeness:

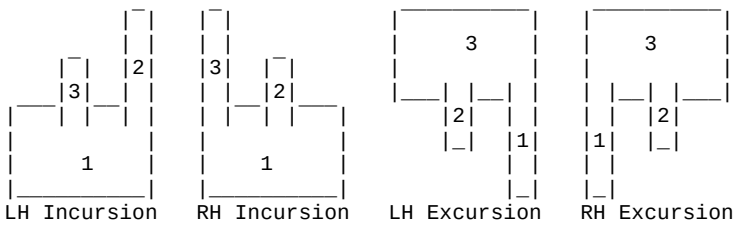

Figure 45: Boundary Inflections.

A LH excursion has occurred when the distance to the left hand boundary is non-zero, a RH excursion has occurred when the current zone width is greater than that of the previous line. Unless the excursion is less than two pixels, in which case it will be ignored, the current position (the bottom left of zone 3 in figure 45) is stacked (5AC2H), the paint direction reversed and painting restarts at the top left of the excursive region . A RH incursion has occurred when the current zone width is smaller than that of the previous line. If the incursion is total, that is the current zone width is zero, a dead end has been reached and the last position and direction are popped (5AIFH) and painting restarts at that point. Otherwise the current position and direction are stacked (5AC2H) and painting restarts at the bottom left of the incursive region. A LH incursion is dealt with automatically during the search for the right hand boundary and requires no explicit action by the paint mainloop.

Address... 5AC2H
This routine is used by the “PAINT” statement handler to save the current paint position and direction on the Z80 stack. The six byte parameter block is made up of the following:

```
2 bytes ... Current contents of CLOC
1 byte  ... Current direction
1 byte  ... Current contents of CMASK
2 bytes ... Current zone width
```

After the parameters have been stacked a check is made that sufficient stack space still exists (625EH).

Address... 5ADCH
This routine is used by the “PAINT” statement handler to locate the right hand boundary. The zone width of the previous line is passed to the SCANR standard routine in register pair DE, this determines the maximum number of boundary colour pixels that may initially be skipped over. The returned skip count remainder is placed in SKPCNT and the number of non-boundary colour pixels traversed in MOVCNT.

Address... 5AEDH
This routine is used by the “PAINT” statement handler to locate the left hand boundary. The end point of the right hand boundary search is temporarily saved and the starting point taken from CSAVEA and CSAVEM and made the current pixel physical address. The left hand boundary is then located via the SCANL standard routine, which also fills in the entire zone, and the right hand end point recovered and placed in CSAVEA and CSAVEM.

Address... 5B0BH
This routine is used by the “CIRCLE” statement handler to negate the contents of register pair DE.

Address... 5B11H
This is the “CIRCLE” statement handler. After evaluating the centre coordinate pair (579CH) the radius is evaluated (520FH), multiplied (325CH) by SIN(PI/4) and placed in CNPNTS. The ink colour is set (584DH), the start angle evaluated (5D17H) and placed in CSTCNT and the end angle evaluated (5D17H) and placed in CENCNT. If the end angle is smaller than the start angle the two values are swapped and CPLOTF is made non-zero. The aspect ratio is evaluated (4C64H) and, if it is greater than one, its reciprocal is taken (3267H) and CSCLXY is made non-zero to indicate an X axis squash. The aspect ratio is multiplied (325CH) by 256, converted to an integer (2F8AH) and placed in ASPECT as a single byte binary fraction. Register pairs HL and DE are set to the starting position on the circle perimeter (X=RADIUS,Y=0) and control drops into the circle mainloop.

Address... 5BBDH
This is the circle mainloop. Because of the high degree of symmetry in a circle it is only necessary to compute the coordinates of the arc from zero to forty-five degrees. The other seven segments are produced by rotation and reflection of these points. The parametric equation for a unit circle, with T the angle from zero to PI/4, is:

```
X=COS(T)
Y=SIN(T)
```

Direct computation using this equation, or the corresponding functional form X=SQR(1-Y^2), is too slow, instead the first derivative is used:

```
dx
-- = -Y/X
dy
```

Given that the starting position is known (X=RADIUS,Y=0), the X coordinate change for each unit Y coordinate change may be computed using the derivative. Furthermore, because graphics resolution is limited to one pixel, it is only necessary to know when the sum of the X coordinate changes reaches unity and then to decrement the X coordinate. Therefore:

```
Decrement X when (Y1/X)+(Y2/X)+(Y3/X)+... => 1
Therefore decrement when (Y1+Y2+Y3+...)/X => 1
Therefore decrement when     Y1+Y2+Y3+... => X
```

All that is required to identify an X coordinate change is to totalize the Y coordinate values from each step until the X coordinate value is exceeded. The circle mainloop holds the X coordinate in register pair HL, the Y coordinate in register pair DE and the running total in CRCSUM. An equivalent BASIC program for a circle of arbitrary radius 160 pixels is:

```
10 SCREEN 2
20 X=160:Y=0:CRCSUM=0
30 PSET(X,191-Y)
40 CRCSUM=CRCSUM+Y :Y=Y+1
50 IF CRCSUM<X THEN 30
60 CRCSUM=CRCSUM-X:X=X-1
70 IF X>Y THEN 30
80 CIRCLE(0,191),155
90 GOTO 90
```

The coordinate pairs generated by the mainloop are those of a “virtual” circle, such tasks as axial reflection, elliptic squash and centre translation are handled at a lower level (5C06H).

Address... 5C06H
This routine is used to by the circle mainloop to convert a coordinate pair, in register pairs HL and DE, into eight symmetric points on the screen. The Y coordinate is initially negated (5B0BH), reflecting it about the X axis, and the first four points produced by successive clockwise rotations through ninety degrees (5C48H). The Y coordinate is then negated again (5B0BH) and a further four points produced (5C48H). Clockwise rotation is performed by exchanging the X and Y coordinates and negating the new Y coordinate, thus a point (40,10) would become (10,-40). Assuming an aspect ratio of 0.5, for example, the complete sequence of eight points would therefore be:

```
(1)  X,-Y*0.5
(2) -Y,-X*0.5
(3) -X, Y*0.5
(4)  Y, X*0.5
(5)  Y,-X*0.5
(6) -X,-Y*0.5
(7) -Y, X*0.5
(8)  X, Y*0.5
```

It can be seen from the above that, ignoring the sign of the coordinates for the moment, there are only four terms nvolved. Therefore, rather than performing the relatively slow aspect ratio multiplication (5CEBH) for each point, the terms X*0.5 and Y*0.5 can be prepared in advance and the complete sequence generated by interchanging and negating the four terms. With the aspect ratio shown above the initial conditions are set up so that register pair HL=X, register pair DE= -Y*0.5, CXOFF=Y and CYOFF=X*0.5 and successive points are produced by the operations:

```
(1) Exchange HL and CXOFF, negate HL.
(2) Exchange DE and CYOFF, negate DE.
```

In parallel with the computation of each circle coordinate the number of points required to reach the start of the segment containing the point is kept in CPCNT8. This will initially be zero and will increase by 2*RADIUS*SIN(PI/4) as each ninety degree rotation is made. As each of the eight points is produced its Y coordinate value is added to the contents of CPCNT8 and compared to the start and end angles to determine the appropriate course of action. If the point is between the two angles and CPLOTF is zero, or if it is outside the angles and CPLOTF is non-zero, the coordinates are added to the circle centre coordinates (5CDCH) and the point set via the SCALXY, MAPXYC and SETC standard routines. If the point is equal to either of the two angles, and the associated bit is set in CLINEF, the coordinates are added to the circle centre coordinates (5CDCH) and a line drawn to the centre (593CH). If none of these conditions is applicable no action is taken other than to proceed to the next point.

Address... 5CEBH
This routine multiplies the coordinate value supplied in register pair DE by the aspect ratio contained in ASPECT, the result is returned in register pair DE. The standard binary shift and add method is used but the operation is performed as two single byte multiplications to avoid overflow problems.

Address... 5D17H
This routine is used by the “CIRCLE” statement handler to convert an angle operand to the form required by the circle mainloop, the result is returned in register pair DE. While the method used is basically sound, and eliminates one trigonometric computation per angle, the results produced are inaccurate. This is demonstrated by the following example which draws a line to the true thirty degree point on a circle’s perimeter:

```
10 SCREEN 2
20 PI = 4 * ATN(1)
30 CIRCLE(100,100),80,,PI/6
40 LINE(100,100)-(100+80*COS(PI/6),100-80*SIN(PI/6))
50 GOTO 50
```

The result that the routine should produce is the number of points that must be produced by the circle mainloop before the required angle is reached. This can be computed by first noting that there will be INT(ANGLE/(PI/4)) forty-five degree segments prior to the segment containing the required angle. Furthermore each forty-five segment will contain RADIUS*SIN(PI/4) points as this is the value of the terminating Y coordinate. Therefore the number of points required to reach the start of the segment containing the angle is the product of these two numbers. The total count is produced by adding this figure to the number of points required to cover any remaining angle within the final segment, that is RADIUS*SIN(REMAINING ANGLE) points. Unfortunately the routine computes the number of points within a segment by linear approximation from the total segment size on the mistaken assumption that successive points subtend equal angles. Thus in the above example the point count computed for the angle is 30/45*(80*0.707107)=37 instead of

the correct value of forty. The error produced by the routine is therefore at a maximum at the centre of each forty-five degree segment and reduces to zero at the end points.

Address... 5D6EH

This is the “DRAW” statement handler. Register pair DE is set to point to the command table at 5D83H and control transfers to the macro language parser (566CH).

Address... 5D83H

This table contains the valid command letters and associated addresses for the “DRAW” statement commands. Those commands which takes a parameter, and consequently have bit 7 set in the table, are shown with an asterisk:

CMD	TO
U*	5DB1H
D*	5DB4H
L*	5DB9H
R*	5DBCH
M	5DD8H
E*	5DCAH
F*	5DC6H
G*	5DD1H
H*	5DC3H
A*	5E4EH
B	5E46H
N	5E42H
X	5782H
C*	5E87H
S*	5E59H

Address... 5DB1H

This is the “DRAW” statement “U” command handler. The operation of the “D”, “L”, “R”, “E”, “F”, “G” and “H” commands is very similar so no separate description of their handlers is given. The optional numeric parameter is supplied by the macro language parser in register pair DE. This initial parameter is modified by a given handler into a horizontal offset in register pair BC and a vertical offset in register pair DE. For example if leftward or upward movement is required the parameter is negated (5B0BH), if diagonal movement is required the parameter is duplicated so that equal horizontal and vertical offsets are produced. Once the offsets have been prepared control transfers to the line drawing routine (5DFFH).

Address... 5DD8H

This is the “DRAW” statement “M” command handler. The character following the command letter is examined then the two parameters collected from the command string (5719H). If the initial character is “+” or “-“ the parameters are regarded as offsets and are scaled (5E66H), rotated through successive ninety degree steps as determined by DRWANG and then added to the current graphics coordinates (5CDCH) to determine the termination point. If DRWFLG shows the “B” mode to be inactive a line is then drawn (5CCDH) from the current graphics coordinates to the termination point. If DRWFLG shows the “N” mode to be inactive the termination coordinates are placed in GRPACX and GRPACY to become the new current graphics coordinates. Finally DRWFLG is zeroed, turning the “B” and “N” modes off, and the handler terminates.

Address... 5E42H

This is the “DRAW” statement “N” command handler, DRWFLG is simply set to 40H.

Address... 5E46H

This is the “DRAW” statement “B” command handler, DRWFLG is simply set to 80H.

Address... 5E4EH

This is the “DRAW” statement “A” command handler. The parameter is checked for magnitude and placed in DRWANG.

Address... 5E59H

This is the “DRAW” statement “S” command handler. The parameter is checked for magnitude and placed in DRWSCL.

Address... 5E66H

This routine is used by the “DRAW” statement “U”, “D”, “L”, “R”, “E”, “F”, “G”, “H” and “M” (in offset mode) command handlers to scale the offset supplied in register pair DE by the contents of DRWSCL. Unless DRWSCL is

zero, in which case the routine simply terminates, the offset is multiplied using repeated addition and then divided by four (59B4H). To eliminate scaling an “S0” or “S4” command should be used.

Address... 5E87H
This is the “DRAW” statement “C” command handler. The parameter is placed in ATRBYT via the SETATR standard routine. There is no check on the MSB of the parameter so illegal values such as “C265” will be accepted without an error message.

Address... 5E91H
This routine is used by the “PAINT” statement handler to check, via the SCALXY standard routine, that the coordinates in register pairs BC and DE are within the screen. If not an “Illegal function call” error is generated (475AH).

Address... 5E9FH
This is the “DIM” statement handler. A return is set up to 5E9AH, so that multiple Arrays can be processed, DIMFLG is made non-zero and control drops into the Variable search routine.

Address... 5EA4H
This is the Variable search routine. On entry register pair HL points to the first character of the Variable name in the program text. On exit register pair HL points to the character following the name and register pair DE to the first byte of the Variable contents in the Variable Storage Area. The first character of the name is taken from the program text, checked to ensure that it is upper case alphabetic (64A7H) and placed in register C. The optional second character, with a default value of zero, is placed in register B, this character may be alphabetic or numeric. Any further alphanumeric characters are then simply skipped over. If a type suffix character (“%”, “$”, “!” or “#”) follows the name this is converted to the corresponding type code (2, 3, 4 or 8) and placed in VALTYP. Otherwise the Variable’s default type is taken from DEFTBL using the first letter of the name to locate the appropriate entry. SUBFLG is then checked to determine how any parenthesized subscript following the name should be treated. This flag is normally zero but is modified by the “ERASE” (01H), “FOR” (64H), “FN” (80H) or “DEF FN” (80H) statement handlers to force a particular course of action. In the “ERASE” case control transfers straight to the Array search routine (5FE8H), no parenthesized subscript need be present. In the “FOR”, “FN” and “DEF FN” cases control transfers straight to the simple Variable search routine (5F08H), no check is made for a parenthesized subscript. Assuming that the situation is normal the program text is checked for the characters “(“ or “[”. If either is present control transfers to the Array search routine (5FBAH), otherwise control drops into the simple Variable search routine.

Address... 5F08H
This is the simple Variable search routine. There are four types of simple Variable each composed of a header followed by the Variable contents. The first byte of the header contains the type code and the next two bytes the Variable name. The contents of the Variable will be one of the three standard numeric forms or, for the string type, the length and address of the string. Each of the four types is shown below:

02H	“A”	“B”	LSB	MSB						
Integer										
03H	“A”	“B”	LEN	LSB	MSB					
String										
04H	“A”	“B”	EE	DD	DD	DD				
Single Precision										
08H	“A”	“B”	EE	DD	DD	DD	DD	DD	DD	DD
Double Precision										

Figure 46: Simple Variables.

NOFUNS is first checked to determine whether a user defined function is currently being evaluated. If so the search is carried out on the contents of PARM1 first of all, only if this fails will it move onto the main Variable Storage Area. A linear search method is used, the two name characters and type byte of each Variable in the storage area are compared to the reference characters and type until a match is found or the end of the storage area is reached. If the search is successful the routine terminates with the address of the first byte of the Variable contents in register pair DE. If the search is unsuccessful the Array Storage Area is moved upwards and the new Variable is added to the end of the existing ones and initialized to zero. There are two exceptions to this automatic creation of a new Variable. If the search is being carried out by the “VARPTR” function, and this is determined by examining the return address, no Variable will be created. Instead the routine terminates with register pair DE set to zero (5F61H) causing a subsequent “Illegal function call” error. The second exception occurs when the search is being carried out by the Factor Evaluator, that is when the Variable is newly declared inside an expression. In this case DAC is zeroed for numeric types, and loaded

with the address of a dummy zero length descriptor for a string type, thus returning a zero result (5FA7H). These actions are designed to prevent the Expression Evaluator creating a new Variable ("VARPTR") is the only function to take a Variable argument directly rather than via an expression and so requires separate protection). If this were not so then assignment to an Array, via the "LET" statement handler, would fail as any simple Variable created during expression evaluation would change the Array's address.

Address... 5FBAH

This is the Array search routine. There are four types of Array each composed of a header plus a number of elements. The first byte of the header contains the type code, the next two bytes the Array name and the next two the offset to the start of the following Array. This is followed by a single byte containing the dimensionality of the Array and the element count list. Each two byte element count contains the maximum number of elements per dimension. These are stored in reverse order with the first one corresponding to the last subscript. The contents of each Array element are identical to the contents of the corresponding simple Variable. The integer Array AB%(3,4) is shown below with each element identified by its subscripts, high memory is towards the top of the page:

```
(0,4) (1,4) (2,4) (3,4)
(0,3) (1,3) (2,3) (3,3)
(0,2) (1,2) (2,2) (3,2)
(0,1) (1,1) (2,1) (3,1)
(0,0) (1,0) (2,0) (3,0)
```

			Offset	Dim	Count	Count
02H	"A"	"B"	2DH 00H	02H	05H 00H	04H 00H

Figure 47: Integer Array.

Each subscript is evaluated, converted to an integer (4755H) and pushed onto the Z80 stack until a closing parenthesis is found, it need not match the opening one. A linear search is then carried out on the Array Storage Area for a match with the two name characters and the type. If the search is successful DIMFLG is checked and a "Redimensioned array" error generated (405EH) if it shows a "DIM" statement to be active. Unless an "ERASE" statement is active, in which case the routine terminates with register pair BC pointing to the start of the Array (3297H), the dimensionality of the Array is then checked against the subscript count and a "Subscript out of range" error generated if they fail to match. Assuming these tests are passed control transfers to the element address computation point (607DH). If the search is unsuccessful and an "ERASE" statement is active an "Illegal function call" error is generated (475AH), otherwise the new Array is added to the end of the existing Array Storage Area. Initialization of the new Array proceeds by storing the two name characters, the type code and the dimensionality (the subscript count) followed by the element count for each dimension. If DIMFLG shows a "DIM" statement to be active the element counts are determined by the subscripts. If the Array is being created by default, with a statement such as "A(1,2,3)=5" for example, a default value of eleven is used. As each element count is stored the total size of the Array is accumulated in register pair DE by successive multiplications (314AH) of the element counts and the element size (the Array type). After a check that this amount of memory is available (6267H) STREND is increased the new area is zeroed and the Array size is stored, in slightly modified form, immediately after the two name characters. Unless the Array is being created by default, in which case the element address must be computed, the routine then terminates. This is the element address computation point of the Array search routine. The location of a particular element within an Array involves the multiplication (314AH) of subscripts, element counts and element sizes. As there are a variety of ways this could be done the actual method used is best illustrated with an example. The location of element (1,2,3) in a 4*5*6 Array would initially be computed as (((3*5)+2)*4)+1. This is then multiplied by the element size (type) and added to the Array base address to obtain the address of the required element. The computation method is an optimized form which minimizes the number of steps needed, it is equivalent to evaluating (3*(4*5))+(2*4)+(1). The element address is returned in register pair DE.

Address... 60B1H

This is the "PRINT USING" statement handler. Control transfers here from the general "PRINT" statement handler after the applicable output device has been set up. Upon termination control passes back to the general "PRINT" statement exit point (4AFFH) to restore the normal video output. The format string is evaluated (4C65H) and the address and length of the string body obtained from the descriptor. The program text pointer is then temporarily saved. Each character of the format string is examined until one of the possible template characters is found. If the character does not belong in a template it is simply output via the OUTDO standard routine. Once the start of a template is found this is scanned along until a non-template character is found. Control then passes to the numeric output routine (6192H) or the string output routine (6211H). In either case the program text pointer is restored to register pair HL and the next operand evaluated (4C64H). For numeric output the information gained from the template scan is passed to the numeric conversion routine (3426H) in registers A, B and C and the resulting string displayed (6678H). For string output the required character count is passed to the "LEFT$" statement handler (6868H) in register C and the resulting string displayed (667BH). For either type of output the program text and format string are then examined to determine whether there are any further characters. If no operands exist the handler terminates. If the format string has been exhausted then it is restarted from the beginning (60BFH), otherwise scanning continues from the current position for the next operand (60f6H).

Address... 6250H
This routine is used by the Interpreter Mainloop and the Variable search routine to move a block of memory upwards. A check is first made to ensure that sufficient memory exists (6267H) and then the block of memory is moved. The top source address is supplied in register pair BC and the top destination address in register pair HL. Copying stops when the contents of register pair BC equal those of register pair DE.

Address... 625EH
This routine is used to check that sufficient memory is available between the top of the Array Storage Area and the base of the Z80 stack. On entry register C contains the number of words the caller requires. If this would narrow the gap to less than two hundred bytes an "Out of memory" error is generated.

Address... 6286H
This is the "NEW" statement handler. TRCFLG, AUTFLG and PTRFLG are zeroed and the zero end link is placed at the start of the Program Text Area. VARTAB is set to point to the byte following the end link and control drops into the run-clear routine.

Address... 629AH
This routine is used by the "NEW", "RUN" and "CLEAR" statement handlers to initialize the Interpreter variables. All interrupts are cleared (636EH) and the default Variable types in DEFTBL set to double precision. RNDX is reset (2C24H) and ONEFLG, ONELIN and OLDTXT are zeroed. MEMSIZ is copied to FRETOP to clear the String Storage Area and DATPTR set to the start of the Program Text Area (63C9H). The contents of VARTAB are copied into ARYTAB and STREND, to clear any Variables, all the I/O buffers are closed (6C1CH) and NLONLY is reset. SAVSTK and the Z80 SP are reset from STKTOP and TEMPPT is reset to the start of TEMPST to clear any string descriptors. The printer is shut down (7304H) and output restored to the screen (4AFFH). Finally PRMLEN, NOFUNS, PRMLN2, FUNACT, PRMSTK and SUBFLG are zeroed and the routine terminates.

Address... 631BH
This routine is used by the "DEVICE ON" statement handlers to enable an interrupt source, the address of the relevant device's TRPTBI. status byte is supplied in register pair HL. Interrupts are enabled by setting bit 0 of the status byte. Bits 1 and 2 are then examined and, if the device has been stopped and an interrupt has occurred, ONGSBF is incremented (634FH) so that the Runloop will process it at the end of the statement. Finally bit 1 of the status byte is reset to release any existing stop condition.

Address... 632EH
This routine is used by the "DEVICE OFF" statement handlers to disable an interrupt source, the address of the relevant device's TRPTBL status byte is supplied in register pair HL. Bits 0 and 2 are examined to determine whether an interrupt has occurred since the end of the last statement, if so ONGSBF is decremented (6362H) to prevent the Runloop from picking it up. The status byte is then zeroed.

Address... 6331H
This routine is used by the "DEVICE STOP" statement handlers to suspend processing of interrupts from an interrupt source, the address of the relevant device's TRPTBL status byte is supplied in register pair HL. Bits 0 and 2 are examined to determine whether an interrupt has occurred since the end of the last statement, if so ONGSBF is decremented (6362H) to prevent the Runloop from picking it up. Bit 1 of the status byte is then set.

Address... 633EH
This routine is used by the "RETURN" statement handler to release the temporary stop condition imposed during interrupt driven BASIC subroutines, the address of the relevant device's TRPTBL status byte is supplied in register pair HL. Bits 0, and 2 are examined to determine whether a stopped interrupt has occurred since the subroutine was first activated. If so ONGSBF is incremented (634FH) so that the Runloop will pick it up at the end of the statement. Bit 1 of the status byte is then reset. It should be noted that any "DEVICE STOP" Statement within an interrupt driven subroutine will therefore be ineffective.

Address... 6358H
This routine is used by the Runloop interrupt processor (6389H) to clear an interrupt prior to activating the BASIC subroutine, the address of the relevant device's TRPTBL status byte is supplied in register pair HL. ONGSBF is decremented and bit 2 of the status byte is reset.

Address... 636EH
This routine is used by the run-clear routine (629AH) to clear all interrupts. The seventy-eight bytes of TRPTBL and the ten bytes of FNKFLG are zeroed.

Address... 6389H
This is the Runloop interrupt processor. ONEFLG is first examined to determine whether an error condition currently exists. If so the routine terminates, no interrupts will be processed until the error clears. CURLIN is then examined and, if the Interpreter is in direct mode, the routine terminates. Assuming all is well a search is made of the twenty-six status bytes in TRPTBL to find the first active interrupt. Note that devices near the start of the table will consequently have a higher priority than those lower down. When the first active status byte is found, that is one with bits 0 and 2 set, the associated address is taken from TRPTBL and placed in register pair DE. The interrupt is then cleared (6358H) and the device stopped (6331H) before control transfers to the "GOSUB" handler (47CFH).

Address... 63C9H
This is the "RESTORE" statement handler. If no line number operand exists DATPTR is set to the start of the Program Storage Area. Otherwise the operand is collected (4769H), the program text searched to find the relevant line (4295H) and its address placed in DATPTR.

Address... 63E3H
This is the "STOP" statement handler. If further text exists in the statement control transfers to the "STOP ON/OFF/STOP" statement handler (77A5H). Otherwise register A is set to 01H and control drops into the "END" statement handler.

Address... 63EAH
This is the "END" statement handler. It is also used, with differing entry points, by the "STOP" statement and for CTRL-STOP and end of text program termination. ONEFLG is first zeroed and then, for the "END" statement only, all I/O buffers are closed (6C1CH). The current program text position is placed in SAVTXT and OLDTXT and the current line number in OLDLIN for use by any subsequent "CONT" statement. The printer is shut down (7304H), a CR LF issued to the screen (7323H) and register pair HL set to point to the "Break" message at 3FDCH. For the "END" statement and end of text cases control then transfers to the Mainloop "OK" point (411EH). For the CTRL-STOP case control transfers to the end of the error handler (40FDH) to display the "Break" message.

Address... 6424H
This is the "CONT" statement handler. Unless they are zero, in which case a "Can't CONTINUE" error is generated, the contents of OLDTXT are placed in register pair HL and those of OLDLIN in CURLIN. Control then returns to the Runloop to execute at the old program text position. A program cannot be continued after CTRL-STOP has been used to break from WITHIN a statement, via the CKCNTC standard routine, rather than from between statements.

Address... 6438H
This is the "TRON" statement handler, TRCFLG is simply made non-zero.

Address... 6439H
This is the "TROFF" statement handler, TRCFLG is simply made zero.

Address... 643EH
This is the "SWAP" statement handler. The first Variable is located (5EA4H) and its contents copied to SWPTMP. The location of this Variable and of the end of the Variable Storage Area are temporarily saved. The second Variable is then located (5EA4H) and its type compared with that of the first. If the types fail to match a "Type mismatch" error is generated (406DH). The current end of the Variable Storage Area is then compared with the old end and an "Illegal function call" error generated (475AH) if they differ. Finally the contents of the second Variable are copied to the location of the first Variable (2EF3H) and the contents of SWPTMP to the location of the second Variable (2EF3H). The checks performed by the handler mean that the second Variable, if it is simple and not an Array, must always be in existence before a "SWAP" Statement is encountered or an error will be generated. The reason for this is that, supposing the first Variable was an Array, then the creation of a second (simple) Variable would move the Array Storage Area upwards invalidating its saved location. Note that the perfectly legal case of a simple first Variable and a newly created simple second Variable is also rejected.

Address... 6477H
This is the "ERASE" statement handler. SUBFLG is first set to 01H, to control the Variable search routine, and the Array located (5EA4H). All the following Arrays are moved downward and STREND set to its new, lower value. The program text is then checked and, if a comma follows, control transfers back to the start of the handler.

Address... 64A7H
This routine checks whether the character whose address is supplied in register pair HL is upper case alphabetic, if so it returns Flag NC.

Address... 64AFH
This is the “CLEAR” statement handler. If no operands are present control transfers to the run-clear routine (62A1H) to remove all current Variables. Otherwise the string space operand is evaluated (4756H) followed by the optional top of memory operand (542FH). The top of memory value is checked and an “Illegal function call” error generated (475AH) if it is less than 8000H or greater than F380H. The space required by the I/O buffers (267 bytes each) and the String Storage Area is subtracted from the top of memory value and an “Out of memory” error generated (6275H) if there is less than 160 bytes remaining to the base of the Variable Storage Area. Assuming all is well HIMEM, MEMSIZ and STKTOP are set to their new values and the remaining storage pointers reset via the run-clear routine (62A1H). The I/O buffer storage is re-allocated (7E6BH) and the handler terminates. Unfortunately the computation of MEMSIZ and STKTOP, when a new top of memory is specified, is incorrect resulting in the top of the String Storage Area being set one byte too high. This can be seen with the following where an illegal string is accepted:

```
10 CLEAR 200,&HF380 20 A$=STRING$(201,”A”)
30 PRINT FRE(“”)
```

Because there should be an extra DEC HL instruction at 64EBH the new values of MEMSIZ and STKTOP are initially set one byte too high. When the run-clear routine is called MEMSIZ is copied into FRETOP, the top of the String Storage Area, which results in this being one byte too high as well. Although MEMSIZ and STKTOP are correctly recomputed when the file pointers are reset, FRETOP is left with its incorrect value. When the “FRE” statement is executed in line thirty, and string garbage collection initiated, FRETOP is restored to its correct value but, because the string overflows the String Storage Area by one byte, the amount of free space displayed is -1 byte. To correctly set all the system pointers any alteration of the top of memory should be followed immediately by another “CLEAR” statement with no operands.

Address... 6520H
This routine computes the difference between the contents of register pairs HL and DE. It is a duplicate of the short section of code from 64ECH to 64F1H and is completely unused.

Address... 6527H
This is the “NEXT” statement handler. Assuming further text is present in the statement the loop Variable is located (5EA4H), otherwise a default address of zero is taken. The stack is then searched for the corresponding “FOR” parameter block (3FE2H). If no parameter block is found, or if a “GOSUB” parameter block is found first, a “NEXT without FOR” error is generated (405BH). Assuming the parameter block is found the intervening section of stack, together with any “FOR” blocks it may contain, is discarded. The loop Variable type is then taken from the parameter block and examined to determine the precision required during subsequent operations. The STEP value is taken from the parameter block and added (3172H, 324EH or 2697H) to the current contents of the loop Variable which is then updated. The new value is compared (2F4DH, 2F21H or 2F5CH) with the termination value from the parameter block to determine whether the loop has terminated (65B6H). The loop will terminate for a positive STEP if the new loop value is GREATER than the termination value. The loop will terminate for a negative step if the new loop value is LESS than the termination value. If the loop has not terminated the original program text position and line number are taken from the parameter block and control transfers to the Runloop (45FDH). If the loop has terminated the parameter block is discarded from the stack and, unless further program text is present in which control transfers back to the start of the handler, control transfers to the Runloop to execute the next statement (4601H).

Address... 65C8H
This routine is used by the Expression Evaluator to find the relation (<>=) between two string operands. The address of the first string descriptor is supplied on the Z80 stack and the address of the second in DAC. The result is returned in register A and the flags as for the numeric relation routines:

```
String 1=String 2 ... A=00H, Flag Z,NC
String 1<String 2 ... A=01H, Flag NZ,NC
String 1>String 2 ... A=FFH, Flag NZ,C
```

Comparison commences at the first character of each string and continues until the two characters differ or one of the strings is exhausted. Control then returns to the Expression Evaluator (4F57H) to place the true or false numeric result in DAC.

Address... 65F5H
This routine is used by the Factor Evaluator to apply the “OCT$” function to an operand contained in DAC. The number is first converted to textual form in FBUFFR (371EH) and then the result string is created (6607H).

Address... 65FAH
This routine is used by the Factor Evaluator to apply the "HEX$" function to an operand contained in DAC. The number is first converted to textual form in FBUFFR (3722H) and then the result string is created (6607H).

Address... 65FFH
This routine is used by the Factor Evaluator to apply the "BIN$" function to an operand contained in DAC. The number is first converted to textual form in FBUFFR (371AH) and then the result string is created (6607H).

Address... 6604H
This routine is used by the Factor Evaluator to apply the "STR$" function to an operand contained in DAC. The number is first converted to textual form in FBUFFR (3425H) then analyzed to determine its length and address (6635H). After checking that sufficient space is available (668EH) the string is copied to the String Storage Area (67C7H) and the result descriptor created (6654H).

Address... 6627H
This routine first checks that there is sufficient space in the String Storage Area for the string whose length is supplied in register A (668EH). The string length and the address where the string will be placed in the String Storage Area are then copied to DSCTMP.

Address... 6636H
This routine is used by the Factor Evaluator to analyze the character string whose address is supplied in register pair HL. The character string is scanned until a terminating character (00H or ") is found. The length and starting address are then placed in DSCTMP (662AH) and control drops into the descriptor creation routine.

Address... 6654H
This routine is used by the string functions to create a result descriptor. The descriptor is copied from DSCTMP to the next available position in TEMPST and its address placed in DAC. Unless TEMPST is full, in which case a "String formula too complex" error is generated, TEMPPT is increased by three bytes and the routine terminates.

Address... 6678H
This routine displays the message, or string, whose address is supplied in register pair HL. The string is analyzed (6635H) and its storage freed (67D3H). Successive characters are then taken from the string and displayed, via the OUTDO standard routine, until the string is exhausted.

Address... 668EH
This routine checks that there is room in the String Storage Area to add the string whose length is supplied in register A. On exit register pair DE points to the starting address in the String Storage Area where the string should be placed. The length of the string is first subtracted from the current free location contained in FRETOP. This is then compared with STKTOP, the lowest allowable location for string storage, to determine whether there is space for the string. If so FRETOP is updated with the new position and the routine terminates. If there is insufficient space for the string then garbage collection is initiated (66B6H) to try and eliminate any dead strings. If, after garbage collection, there is still not enough space an "Out of string space" error is generated.

Address... 66B6H
This is the string garbage collector, its function is to eliminate any dead strings from the String Storage Area. The basic problem with string Variables, as opposed to numeric ones, is that their lengths vary. If string bodies were stored with their Variables in the Variable Storage Area even such apparently simple statements as A$=A$+"X" would require the movement of thousands of bytes of memory and slow execution speeds dramatically. The method used by the Interpreter to overcome this problem is to keep the string bodies separate from the Variables. Thus strings are kept in the String Storage Area and each Variable holds a three byte descriptor containing the length and address of the associated string. Whenever a string is assigned to a Variable it is simply added to the heap of existing strings in the String Storage Area and the Variable's descriptor changed. No attempt is made to eliminate any previous string belonging to the Variable, by restructuring the heap, as this would wipe out any throughput gains. If sufficient Variable assignments are made it is inevitable that the String Storage Area will fill up. In a typical program many of these strings will be unused, that is the result of previous assignments. Garbage collection is the process whereby these dead strings are removed. Every string Variable in memory, including Arrays and the local Variables present during evaluation of user defined functions, is examined until the one is found whose string is stored highest in the heap. This string is then moved to the top of the String Storage Area and the Variable contents modified to point to the new location. The owner of the next highest string is then found and the process repeated until every string belonging to a Variable has been compacted. If a large number of Variables are present garbage collection may take an appreciable time. The process can be seen at work with the following program which repeatedly assigns the string "AAAA" to each element of the Array

A$. The program will run at full speed for the first two hundred and fifty assignments and then pause to eliminate the fifty dead strings. A further fifty assignments can then be made before a further garbage collection is required:

```
10 CLEAR 1000 20 DIM A$(200)
30 FOR N=0 TO 200
40 A$(N)=STRING$(4,"A")
50 PRINT".";
60 NEXT N
70 GOTO 30
```

The String Storage Area is also used to hold the intermediate strings produced during expression evaluation. Because so many string functions take multiple arguments, "MID$" takes three for example, the management of intermediate results is a major problem. To deal with it a standardized approach to string results is taken throughout the Interpreter. A producer of a string simply adds the string body to the heap in the String Storage Area, adds the descriptor to the descriptor heap in TEMPST and places the address of the descriptor in DAC. It is up to the user of the result to free this storage (67D0H) once it has processed the string. This rule applies to all parts of the system, from the individual function handlers back through the Expression Evaluator to the statement handlers, with only two exceptions. The first exception occurs when the Factor Evaluator finds an explicitly stated string, such as "SOMETHING" in the program text. In this case it is not necessary to copy the string to the String Storage Area as the original will suffice. The second exception occurs when the Factor Evaluator finds a reference to a Variable. In this case it is not necessary to place a copy of the descriptor in TEMPST as one already exists inside the Variable.

Address... 6787H
This routine is used by the Expression Evaluator to concatenate two string operands. Control transfers here when a "+" token is found following a string operand so the first action taken is to fetch the second string operand via the Factor Evaluator (4DC7H). The lengths are then taken from both string descriptors and added together to check the length of the combined string. If this is greater than two hundred and fifty-five characters a "String too long" error is generated. After checking that space is available in the String Storage Area (6627H) the storage of both operands is freed (67D6H). The first string is then copied to the String Storage Area (67BFH) and followed by the second one (67BFH). The result descriptor is created (6654H) and control transfers back to the Expression Evaluator (4C73H)'

Address... 67D0H
This routine frees any storage occupied by the string whose descriptor address is contained in DAC. The address of the descriptor is taken from DAC and examined to determine whether it is that of the last descriptor in TEMPST (67EEH), if not the routine terminates. Otherwise TEMPPT is reduced by three bytes clearing this descriptor from TEMPST. The address of the string body is then taken from the descriptor and compared with FRETOP to see if this is the lowest string in the String Storage Area, if not the routine terminates. Otherwise the length of the string is added to FRETOP, which is then updated with this new value, freeing the storage occupied by the string body.

Address... 67FFH
This routine is used by the Factor Evaluator to apply the "LEN" function to an operand contained in DAC. The operand's storage is freed (67D0H) and the string length taken from the descriptor and placed in DAC as an integer (4FCFH).

Address... 680BH
This routine is used by the Factor Evaluator to apply the "ASC" function to an operand contained in DAC. The operand's storage is freed and the string length examined (6803H), if it is zero an "Illegal function call" error is generated (475AH). Otherwise the first character is. taken from the string and placed in DAC as an integer (4FCFH).

Address... 681BH
This routine is used by the Factor Evaluator to apply the "CHR$" function to an operand contained in DAC. After checking that sufficient space is available (6625H) the operand is converted to a single byte integer (521FH). This character is then placed in the String Storage Area and the result descriptor created (6654H).

Address... 6829H
This routine is used by the Factor Evaluator to apply the "STRING$" function. After checking for the open parenthesis character the length operand is evaluated and placed in register E (521CH). The second operand is then evaluated (4C64H). If it is numeric it is converted to a single byte integer (521FH) and placed in register A. If it is a string the first character is taken from it and placed in register A (680FH). Control then drops into the "SPACE$" function to create the result string.

Address... 6848H
This routine is used by the Factor Evaluator to apply the "SPACE$" function to an operand contained in DAC. The operand is first converted to a single byte integer in register E (521FH). After checking that sufficient space is available (6627H) the required number of spaces are copied to the String Storage Area and the result descriptor created (6654H).

Address... 6861H
This routine is used by the Factor Evaluator to apply the "LEFT$" function. The first operand's descriptor address and the integer second operand are supplied on the Z80 stack. The slice size is taken from the stack (68E3H) and compared to the source string length. If the source string length is less than the slice size it replaces it as the length to extract. After checking that sufficient space is available (668EH) the required number of characters are copied from the start of the source string to the String Storage Area (67C7H). The source string's storage is then freed (67D7H) and the result descriptor created (6654H).

Address... 6891H
This routine is used by the Factor Evaluator to apply the "RIGHT$" function. The first operand's descriptor address and the integer second operand are supplied on the Z80 stack. The slice size is taken from the stack (68E3H) and subtracted from the source string length to determine the slice starting position. Control then transfers to the "LEFT$" routine to extract the slice (6865H).

Address... 689AH
This routine is used by the Factor Evaluator to apply the "MID$" function. The first operand's descriptor address and the integer second operand are supplied on the Z80 stack. The starting position is taken from the stack (68E6H) and checked, if it is zero an "Illegal function call" error is generated (475AH). The optional slice size is then evaluated (69E4H) and control transfers to the "LEFT$" routine to extract the slice (6869H).

Address... 68BBH
This routine is used by the Factor Evaluator to apply the "VAL" function to an operand contained in DAC. The string length is taken from the descriptor (6803H) and checked, if it is zero it is placed in DAC as an integer (4FCFH). The length is then added to the starting address of the string body to give the location of the character immediately following it. This is temporarily replaced with a zero byte and the string is converted to numeric form in DAC (3299H). The original character is then restored and the routine terminates. The temporary zero byte delimiter is necessary because strings are packed together in the String Storage Area, without it the numeric converter would run on into succeeding strings.

Address... 68E3H
This routine is used by the "LEFT$", "MID$" and "RIGHT$" function handlers to check that the next program text character is ")" and then to pop an operand from the Z80 stack into register pair DE.

Address... 68EBH
This routine is used by the Factor Evaluator to apply the "INSTR" function. The first operand, which may be the starting position or the source string, is evaluated (4C62H) and its type tested. If it is the source string a default starting position of one is taken. If it is the starting position operand its value is checked and the source string operand evaluated (4C64H). The pattern string is then evaluated (4C64H) and the storage of both operands freed (67D0H). The length of the pattern string is checked and, if zero, the starting position is placed in DAC (4FCFH). The pattern string is then checked against successive characters from the source string, commencing at the starting position, until a match is found or the source string is exhausted. With a successful search the character position of the substring is placed in DAC as an integer (4FCFH), otherwise a zero result is returned.

Address... 696EH
This is the "MID$" statement handler. After checking for the open parenthesis character the destination Variable is located (5EA4H) and checked to ensure that it is a string type (3058H). The address of the string body is then taken from the Variable and examined to determine whether it is inside the Program Text Area, as would be the case for an explicitly stated string. If this is the case the string body is copied to the String Storage Area (6611H) and a new descriptor copied to the Variable (2EF3H). This is done to avoid modifying the program text. The starting position is then evaluated (521CH) and checked, if it is zero an "Illegal function call" error is generated (475AH). The optional slice length operand is evaluated (69E4H) followed by the replacement string (4C5FH) whose storage is then freed (67D0H). Characters are then copied from the replacement string to the destination string until either the slice length is completed or the replacement string is exhausted.

Address... 69E4H
This routine is used by various string functions to evaluate an optional operand (521CH) and return the result in register E. If no operand is present a default value of 255 is returned.

Address... 69F2H
This routine is used by the Factor Evaluator to apply the "FRE" function to an operand contained in DAC. If the operand is numeric the single precision difference between the Z80 Stack Pointer and the contents of STREND is placed in DAC (4FC1H). If the operand is a string type its storage is freed (67D3H) and garbage collection initiated (66B6H). The single precision difference between the contents of FRETOP and those of STKTOP is then placed in DAC (4FC1H).

Address... 6A0EH
This routine is used by the file I/O handlers to analyze a filespec such as "A:FILENAME.BAS". The filespec consists of three parts, the device, the filename and the type extension. On entry register pair HL points to the start of the filespec in the program text. On exit register D holds the device code, the filename is in positions zero to seven of FILNAM and the type extension in positions eight to ten. Any unused positions are filled with spaces. The filespec string is evaluated (4C64H) and its storage freed (67D0H), if the string is of zero length a "Bad file name" error is generated (6E6BH). The device name is parsed (6F15H) and successive characters taken from the filespec and placed in FILNAM until the string is exhausted, a "." character is found or FILNAM is full. A "Bad file name" error is generated (6E6BH) if the filespec contains any control characters, that is those whose value is smaller than 20H. If the filespec contains a type extension a "Bad file name" error is generated (6E6BH) if it is longer than three characters or if the filename is longer than eight characters. If no type extension is present the filename may be any length, extra characters are simply ignored.

Address... 6A6DH
This routine is used by the file I/O handlers to locate the I/O buffer FCB whose number is supplied in register A. The buffer number is first checked against MAXFIL and a "Bad file number" error generated (6E7DH) if it is too large. Otherwise the required address is taken from the file pointer block and placed in register pair HL and the buffer's mode taken from byte 0 of the FCB and placed in register A.

Address... 6A9EH
This routine is used by the file I/O handlers to evaluate an I/O buffer number and to locate its FCB. Any "#" character is skipped (4666H) and the buffer number evaluated (521CH). The FCB is located (6A6DH) and a "File not open" error generated (6E77H) if the buffer mode byte is zero. Otherwise the FCB address is placed in PTRFIL to redirect the Interpreter's output.

Address... 6AB7H
This is the "OPEN" statement handler. The filespec is analyzed (6A0EH) and any following mode converted to the corresponding mode byte, these are: "FOR INPUT" (01H), "FOR OUTPUT" (02H) and "FOR APPEND" (08H). If no mode is explicitly stated random mode (04H) is assumed. The "AS'- characters are checked and the buffer number evaluated (521CH), if this is zero a "Bad file number" error is generated (6E7DH). The FCB is then located (6A6DH) and a "File already open" error generated (6E6EH) if the buffer's mode byte is anything other than zero. The device code is placed in byte 4 of the FCB, the open function dispatched (6F8FH) and the Interpreter's output reset to the screen (4AFFH).

Address... 6B24H
This routine is used by the file I/O handlers to close the I/O buffer whose number is supplied in register A. The FCB is located (6A6DH) and, provided the buffer is in use, the close function dispatched (6F8FH) and the buffer filled with zeroes (6CEAH). PTRFIL and the FCB mode byte are then zeroed to reset the Interpreter's output to the screen.

Address... 6B5BH
This is the "LOAD", "MERGE" and "RUN filespec" statement handler. The filespec is analyzed (6A0EH) and then, for "LOAD" and "RUN" only, the program text examined to determine whether the auto-run "R" option is specified. I/O buffer 0 is opened for input (6AFAH) and the first byte of FILNAM set to FFH if auto-run is required. For "LOAD" and "RUN" only any program text is then cleared via the "NEW" statement handler (6287H). As this will reset the Interpreter's output to the screen the buffer FCB is again located and placed in PTRFIL (6AAAH). Control then transfers directly to the Interpreter Mainloop (4134H) for the program text to be loaded as if typed from the keyboard. Note that no error checking of any sort is carried out on the data read.

Address... 6BA3H
This is the "SAVE" statement handler. The filespec is analyzed (6A0EH) and the program text examined to determine whether the ASCII "A" suffix is present. This is only relevant under Disk BASIC, it makes no difference on a standard MSX machine. I/O buffer 0 is opened for output (6AFAH) and control transfers to the "LIST" statement handler (522EH) to output the program text. Note that no error checking information of any sort accompanies the text.

Address... 6BDAH
This routine is used by the file I/O handlers to return the device code for the currently active I/O buffer. The FCB address is taken from PTRFIL then the device code taken from byte 4 of the FCB and placed in register A.

Address... 6BE7H
This routine is used by the file I/O handlers to perform an operation on a number of I/O buffers. The address of the relevant routine is supplied in register pair BC and the buffer count in register A. For example if register pair BC contained 6B24H and register A contained 03H buffers 3, 2, 1 and 0 would be closed. The routine has a slightly different function if it is entered with FLAG NZ. In this case the I/O buffer numbers are taken sequentially from the program text and evaluated (521CH) before the operation is performed, a typical case might be “#1,#2”.

Address... 6C14H
This is the “CLOSE” statement handler. Register pair BC is set to 6B24H, register A is loaded with the contents of MAXFIL and the required number of buffers closed (6BE7H).

Address... 6C1CH
This routine is used by the file I/O handlers to close every I/O buffer. Register pair BC is set to 6B24H, register A is loaded with the contents of MAXFIL and all buffers closed (6BE7H).

Address... 6C2AH
This is the “LFILES” statement handler. PRTFLG is made non-zero, to direct output to the printer, and control drops into the “FILES” statement handler.

Address... 6C2FH
This is the “FILES” statement handler, an “Illegal function call” error is generated (475AH) on a standard MSX machine.

Address... 6C35H
Control transfers here from the general “PUT” and “GET” handlers (7758H) when the program text contains anything other than a “SPRITE” token. A “Sequential I/O only” error is generated (6E86H) on a standard MSX machine.

Address... 6C48H
This routine is used by the file I/O handlers to sequentially output the character supplied in register A. The character is placed in register C and the sequential output function dispatched (6F8FH).

Address... 6C71H
This routine is used by the file I/O handlers to sequentially input a single character. The sequential input function is dispatched (6F8FH) and the character returned in register A, FLAG C indicates an EOF (End Of File) condition.

Address... 6C87H
This routine is used by the Factor Evaluator to apply the “INPUT$” function. The program text is checked for the “$” and “(“ characters and the length operand evaluated (521CH). If an I/O buffer number is present it is evaluated, the FCB located (6A9EH) and the mode byte examined. An “Input past end” error is generated (6E83H) if the buffer is not in input or random mode. After checking that sufficient space is available (6627H) the required number of characters are sequentially input (6C71H), or collected via the CHGET standard routine, and copied to the String Storage Area. Finally the result descriptor is created (6654H).

Address... 6CEAH
This routine is used by the file I/O handlers to fill the buffer whose FCB address is contained in PTRFIL with two hundred and fifty-six zeroes.

Address... 6CFBH
This routine is used by the file I/O handlers to return, in register pair HL, the starting address of the buffer whose FCB address is contained in PTRFIL. This just involves adding nine to the FCB address.

Address... 6D03H
This routine is used by the Factor Evaluator to apply the “LOC” function to the I/O buffer whose number is contained in DAC. The FCB is located (6A6AH) and the LOC function dispatched (6F8FH). An “Illegal function call” error is generated (475AH) on a standard MSX machine.

Address... 6D14H
This routine is used by the Factor Evaluator to apply the “LOF” function to the I/O buffer whose number is contained in DAC. The FCB is located (6A6AH) and the LOF function dispatched (6F8FH). An “Illegal function call” error is generated (475AH) on a standard MSX machine.

Address... 6D25H
This routine is used by the Factor Evaluator to apply the “EOF” function to the I/O buffer whose number is contained in DAC. The FCB is located (6A6AH) and the EOF function dispatched (6F8FH).

Address... 6D39H
This routine is used by the Factor Evaluator to apply the “FPOS” function to the I/O buffer whose number is contained in DAC. The FCB is located (6A6AH) and the FPOS function dispatched (6F8FH). An “Illegal function call” error is generated (475AH) on a standard MSX machine.

Address... 6D48H
Control transfers to this routine when the Interpreter Mainloop encounters a direct statement, that is one with no line number. The ISFLIO standard routine is first used to determine whether a “LOAD” statement is active. If input is coming from the keyboard control transfers to the Runloop execution point (4640H) to execute the statement. If input is coming from the cassette buffer 0 is closed (6B24H) and a “Direct statement in file” error generated (6E71H). This could happen on a standard MSX machine either through a cassette error or by attempting to load a text file with no line numbers.

Address... 6D57H
This routine is used by the “INPUT”, “LINE INPUT” and “PRINT” statement handlers to check for the presence of a “#” character in the program text. If one is found the I/O buffer number is evaluated (521BH), the FCB located and its address placed in PTRFIL (6AAAH). The mode byte of the FCB is then compared with the mode number supplied by the statement handler in register C, if they do not match a “Bad file number” error is generated (6E7DH). With “PRINT” the allowable modes are output, random and append. With “INPUT” and “LINE INPUT” the allowable modes are input and random. Note that on a standard MSX machine not all these modes are supported at lower levels. Some sort of error will consequently be generated at a later stage for illegal modes.

Address... 6D83H
This routine is used by the “INPUT” statement handler to input a string from an I/O buffer. A return is first set up to the “READ/INPUT” statement handler (4BF1H). The characters which delimit the input string, comma and space for a numeric Variable and comma only for a string Variable, are placed in registers D and E and control transfers to the “LINE INPUT” routine (6DA3H).

Address... 6D8FH
This is the “LINE INPUT” statement handler when input is from an I/O buffer. The buffer number is evaluated, the FCB located and the mode checked (6D55H). The Variable to assign to is then located (5EA4H) and its type checked to ensure it is a string type (3058H). A return is set up to the “LET” statement handler (487BH) to perform the assignment and the input string collected. Characters are sequentially input (6C71H) and placed in BUF until the correct delimiter is found, EOF is reached or BUF fills up (6E41H). When the terminating condition is reached and assignment is to a numeric Variable the string is converted to numeric form in DAC (3299H). When assignment is to a string Variable the string is analyzed and the result descriptor created (6638H). For “LINE INPUT” all characters are accepted until a CR code is reached. Note that if this CR code is preceded by a LF code then it will not function as a delimiter but will merely be accepted as part of the string. For “INPUT” to a numeric Variable leading spaces are stripped then characters accepted until a CR code, a space or a comma is reached. Note that as for “LINE INPUT” a CR code will not function as a delimiter when preceded by a LF code. In this case however the CR code will not be placed in BUF but ignored. For “INPUT” to a string Variable leading spaces are stripped then characters accepted until a CR or comma is reached. Note that as for “LINE INPUT” a CR code will not function as a delimiter when preceded by a LF code. In this case however neither code will be placed in BUF both are ignored. An alternative mode is entered when the first character read, after any spaces, is a double quote character. In this case all characters will be accepted, and stored in BUF, until another double quote delimiter is read. Once the input string has been accepted the terminating delimiter is examined to see if any special action is required with respect to trailing characters. If the input string was delimited by a double quote character or a space then any succeeding spaces will be read in and ignored until a non-space character is found. If this character is a comma or CR code then it is accepted and ignored. Otherwise a putback function is dispatched (6F8FH) to return the character to the I/O buffer. If the input string was delimited by a CR code then the next character is read in and checked. If this is a LF code it will be accepted but ignored. If it is not a LF code then a putback function is dispatched (6F8FH) to return the character to the I/O buffer.

Address... 6E6BH
This is a group of ten file I/O related error generators. Register E is loaded with the relevant error code and control transfers to the error handler (406FH):

```
ADDR. ERROR
6E6BH Bad file name
6E6EH File already open
6E71H Direct statement in file
6E74H File not found
6E77H File not open
6E7AH Field overflow
6E7DH Bad file number
6E80H Internal error
6E83H Input past end
6E86H Sequential I/O only
```

Address... 6E92H
This is the “BSAVE” statement handler. The filespec is analyzed (6A0EH) and the start address evaluated (6F0BH). The stop address is then evaluated (6F0BH) and placed in SAVEND followed by the optional entry address (6F0BH) which is placed in SAVENT. If no entry address exists the start address is taken instead. The device code is checked to ensure that it is CAS, if not a “Bad file name” error is generated (6E6BH), and the data written to cassette (6FD7H). Note that no buffering is involved, data is written directly to the cassette, and no error checking information accompanies the data.

Address... 6EC6H
This is the “BLOAD” statement handler. The filespec is analyzed (6A0EH) and RUNBNF made non-zero if the auto-run “R” option is present in the program text. The optional load offset, with a default value of zero, is then evaluated (6F0BH) and the device code checked to ensure that it is CAS, if not a “Bad file name” error is generated (6E6BH). Data is then read directly from cassette (7014H), as with “BSAVE” no buffering or error checking is involved.

Address... 6EF4H
Control transfers to this routine when the “BLOAD” statement handler has completed loading data into memory. If RUNBNF is zero buffer 0 is closed (6B24H) and control returns to the Runloop. Otherwise buffer 0 is closed (6B24H), a return address of 6CF3H is set up (this routine just pops the program text pointer back into register pair HL and returns to the Runloop) and control transfers to the address contained in SAVENT.

Address... 6F0BH
This routine is used by the “BLOAD” and “BSAVE” handlers to evaluate an address operand, the result is returned in register pair DE. The operand is evaluated (4C64H) then converted to an integer (5439H).

Address... 6F15H
This routine is used by the filespec analyzer to parse a device name such as “CAS:”. On entry register pair HL points to the start of the filespec string and register E contains its length. If no device name is present the default device code (CAS=FFH) is returned in register A with FLAG Z. If a legal device name is present its code is returned in register A with FLAG NZ. The filespec is examined until a “:” character is found then the name compared with each of the legal device names in the device table at 6F76H. If a match is found the device code is taken from the table and returned in register A. If no match is found control transfers to the external ROM search routine (55F8H). Note that any lower case characters are turned to upper case for comparison purposes. Thus crt and CRT, for example, are the same device.

Address... 6F76H
This table is used by the device name parser, it contains the four device names and codes available on a standard MSX machine:

```
CAS ... FFH  LPT ... FEH  CRT ... FDH  GRP ... FCH
```

Address... 6F87H
This table is used by the function dispatcher (6F8FH), it contains the address of the function decoding table for each of the four standard MSX devices:

```
CAS ... 71C7H  LPT ... 72A6H  CRT ... 71A2H  GRP ... 7182H
```

Address... 6F8FH
This is the file I/O function dispatcher. In conjunction with the Interpreter’s buffer structure it provides a consistent, device independent method of inputting or outputting data. The required function code is supplied in register A and the address of the buffer FCB in register pair HL. The device code is taken from byte 4 of the FCB and examined to determine whether it is one of the four standard devices, if not control transfers to the external ROM function dispatcher (564AH). Otherwise the address of the device’s function decoding table is taken from the table at 6F87H, the required function’s address taken from it and control transferred to the relevant function handler.

Address... 6FB7H
This is the “CSAVE” statement handler. The filename is evaluated (7098H) followed by the optional baud rate operand (7A2DH). The identification block is then written to cassette (7125H) with a filetype byte of D3H. The contents of the Program Text Area are written directly to cassette as a single data block (713EH). Note that no error checking information accompanies the data.

Address... 6FD7H
Control transfers to this routine from the “BSAVE” statement handler to write a block of memory to cassette. The identification block is first written to cassette (7125H) with a filetype byte of D0H. The motor is then turned on and a short header written to cassette (72F8H) The starting address is popped from the Z80 stack and written to cassette LSB first, MSB second (7003H). The stop address is taken from SAVEND and written to cassette LSB first, MSB second (7003H). The entry address is taken from SAVENT and written to cassette LSB first, MSB second (7003H). The required area of memory is then written to cassette one byte at a time (72DEH) and the cassette motor turned off via the TAPOOF standard routine. Note that no error checking information accompanies the data.

Address... 7003H
This routine writes the contents of register pair HL to cassette with register L first (72DEH) and register H second (72DEH).

Address... 700BH
This routine reads two bytes from cassette and places the first in register L (72D4H), the second in register H (72D4H).

Address... 7014H
Control transfers to this routine from the “BLOAD” statement handler to load data from the cassette into memory. The cassette is read until an identification block with a file type of D0H and the correct filename is found (70B8H). The data block header is then located on the cassette (72E9H). The offset value is popped from the Z80 stack and added to the start address from the cassette (700BH). The stop address is read from cassette (700BH) and the offset added to this as well. The entry address is read from cassette (700BH) and placed in SAVENT in case auto-run is required. Successive data bytes are then read from cassette (72D4H) and placed in memory, at the start address initially, until the stop address is reached. Finally the motor is turned off via the TAPIOF standard routine and control transfers to the “BLOAD” termination point (6EF4H).

Address... 703FH
This is the “CLOAD” and “CLOAD?” statement handler. The program text is first checked for a trailing “PRINT” token (91H) which is how the “?” character is tokenized. The filename is then evaluated (708CH) and the cassette read until an identification block with a filetype of D3H and the correct filename is found (70B8H). For “CLOAD” a “NEW” operation is then performed (6287H) to erase the current program text. For “CLOAD?” all pointers in the Program Text Area are converted to line numbers (54EAH) to match the cassette data. The data block header is located on the cassette and successive data bytes read from cassette and placed in memory or compared with the current memory contents (715DH). When the data block has been completely read the message “OK” is displayed (6678H) and control transfers directly to the end of the Interpreter Mainloop (4237H) to reset the Variable storage pointers. For “CLOAD?” reading of the data block will terminate if the cassette byte is not the same as the program text byte in memory. If the address where this occurred is above the end of the Program Text Area then the handler terminates with an “OK” message as before. Otherwise a “Verify error” is generated.

Address... 708CH
This routine is used by the “CLOAD” and “CSAVE” statement handlers to evaluate a filename in the program text. The two handlers use different entry points so that a null filename is allowed for “CLOAD” but not for “CSAVE”. The filename string is evaluated (4C64H), its storage freed (680FH) and the first six characters copied to FILNAM. If the filename is longer than six characters the excess is ignored. If the filename is shorter than six characters then FILNAM is padded with spaces.

Address... 70B8H
This routine is used by the “CLOAD” and “BLOAD” statement handlers and for the dispatcher open function (when the device is CAS and the mode is input) to locate an identification block on the cassette. On entry the filename is in FILNAM and the file type in register C, D3H for a tokenized BASIC (CLOAD) file, D0H for a binary (BLOAD) file and EAH for an ASCII (LOAD or data) file. The cassette motor is turned on and the cassette read until a header is found (72E9H). Each identification block is prefixed by ten file type characters so successive characters are read from cassette (72D4H) and compared to the required file type. If the file type characters do not match control transfers back to the start of the routine to find the next header. Otherwise the next six characters are read in (72D4H) and placed in FILNAM. If FILNAM is full of spaces no filename match is attempted and the identification block has been found. Otherwise the contents of FILNAM and FILNM2 are compared to determine whether this is the required file. If the

match is unsuccessful, and the Interpreter is in direct mode, the message “Skip:” is displayed (710DH) followed by the filename. Control then transfers back to the start of the routine to try the next header. If the match is successful, and the Interpreter is in direct mode, the message “Found:” is displayed (710DH) followed by the filename and the routine terminates.

Address... 70FFH
This is the plain text message “Found:” terminated by a zero byte.

Address... 7106H
This is the plain text message “Skip :” terminated by a zero byte.

Address... 710DH
Unless CURLIN shows the Interpreter to be in program mode this routine first displays (6678H) the message whose address is supplied in register pair HL, followed by the six characters contained in FILENAM2.

Address... 7125H
This routine is used by the “CSAVE” and “BSAVE” statement handlers and for the dispatcher open function (when the device is CAS and the mode is output) to write an identification block to cassette. On entry the filename is in FILNAM and the filetype in register A, D3H for a tokenized BASIC (CSAVE) file, D0H for a binary (BSAVE) file and EAH for an ASCII (SAVE or data) file. The cassette motor is turned on and a long header written to cassette (72F8H) The filetype byte is then written to cassette (72DEH) ten times followed by the first six characters from FILNAM (72DEH). The cassette motor is turned off via the TAPOOF standard routine and the routine terminates.

Address... 713EH
This routine is used by the “CSAVE” statement handler to write the Program Text Area to cassette as a single data block. All pointers in the program text are converted back to line numbers (54EAH) to make the text address independent. The cassette motor is turned on and a short header written to cassette (72F8H) The entire Program Text Area is then written to cassette a byte at a time (72DEH) and followed with seven zero bytes (72DEH) as a terminator. The cassette motor is then turned off via the TAPOOF standard routine and the routine terminates.

Address... 715DH
This routine is used by the “CLOAD” and “CLOAD?” statement handlers to read a single data block into the Program Text Area or to compare it with the current contents. On entry register A contains a flag to distinguish between the two statements, 00H for “CLOAD” and FFH for “CLOAD?”. The cassette motor is turned on and the first header located (72E9H). Successive characters are read from cassette (72D4H) and placed in the Program Text Area or compared with the current contents. If the current statement is “CLOAD?” the routine will terminate with FLAG NZ if the cassette character is not the same as the memory character. Otherwise data will be read until ten successive zeroes are found. This sequence of zeroes is composed of the last program line end of line character, the end link and the seven terminator zeroes added by “CSAVE”. Note that the routine will probably terminate during this sequence, when used by “CLOAD?”, as memory comparison is still in progress. This accounts for the rather peculiar coding of the “CLOAD?” handler terminating conditions.

Address... 7182H
This table is used by the dispatcher when decoding function codes for the GRP device. It contains the address of the handler for each of the function codes, most are in fact error generators:

```
TO    FUNCTION
71B6H  0, open
71C2H  2, close
6E86H  4, random
7196H  6, sequential output
475AH  8, sequential input
475AH 10, loc
475AH 12, lof
475AH 14, eof
475AH 16, fpos
475AH 18, putback
```

Address... 7196H
This is the dispatcher sequential output routine for the GRP device. SCRMOD is first checked and an “Illegal function call” error generated (475AH) if the screen is in either text mode. The character to output is taken from register C and control transfers to the GRPPRT standard routine.

Address... 71A2H
This table is used by the DEVICE DISPATCHER when decoding function codes for the CRT device. It contains the address of the handler for each of the function codes, most are in fact error generators:

```
TO    FUNCTION
71B6H  0, open
71C2H  2, close
6E86H  4, random
71C3H  6, sequential output
475AH  8, sequential input
475AH 10, loc
475AH 12, lof
475AH 14, eof
475AH 16, fpos
475AH 18, putback
```

Address... 71B6H
This is the dispatcher open routine for the CRT, LPT and GRP devices. The required mode, in register E, is checked and a “Bad file name” error generated (6E6BH) for input or append. The FCB address is then placed in PTRFIL, the mode in byte 0 of the FCB and the routine terminates. Note that the Z80 RET instruction at the end of this routine (71C2H) is the dispatcher close routine for the CRT, LPT and GRP devices.

Address... 71C3H
This is the dispatcher sequential output routine for the CRT device. The character to output is taken from register C and control transfers to the CHPUT standard routine.

Address... 71C7H
This table is used by the dispatcher when decoding function codes for the CAS device. It contains the address of the handler for each of the function codes, several are error generators:

```
TO    FUNCTION
71DBH  0, open
7205H  2, close
6E86H  4, random
722AH  6, sequential output
723FH  8, sequential input
475AH 10, loc
475AH 12, lof
726DH 14, eof
475AH 16, fpos
727CH 18, putback
```

Address... 71DBH
This is the dispatcher open routine for the CAS device. The current I/O buffer position, held in byte 6 of the FCB, and CASPRV, which holds any putback character are both zeroed. The required mode, supplied in register E, is examined and a “Bad file name” error generated (6E6BH) for append or random modes. For output mode the identification block is then written to cassette (7125H) while for input mode the correct identification block is located on the cassette (70B8H). The FCB address is then placed in PTRFIL, the mode in byte 0 of the FCB and the routine terminates.

Address... 7205H
This is the dispatcher close routine for the CAS device. Byte 0 of the FCB is examined and, if the mode is input, CASPRV is zeroed and the routine terminates. Otherwise the remainder of the I/O buffer is filled with end of file characters (1AH) and the I/O buffer contents written to cassette (722FH). CASPRV is then zeroed and the routine terminates.

Address... 722AH
This is the dispatcher sequential output routine for the CAS device. The character to output is taken from register C and placed in the next free position in the I/O buffer (728BH). Byte 6 of the FCB, the I/O buffer position, is then incremented. If the I/O buffer position has wrapped round to zero this means that there are two hundred and fifty-six characters in the I/O buffer and it has to be written to cassette. The cassette motor is turned on, a short header is written to cassette (72F8H) followed by the I/O buffer contents (72DEH), and the motor is turned off via the TAPOOF standard routine.

Address... 723FH
This is the dispatcher sequential input routine for the CAS device. CASPRV is first checked (72BEH) to determine whether it contains a character which has been putback, in which case its contents will be non-zero. If so the routine terminates with the character in register A. Otherwise the I/O buffer position is checked (729BH) to determine whether it contains any characters. If the I/O buffer is empty the cassette motor is turned on and the header located (72E9H).

Two hundred and fifty-six characters are then read in (72D4H), the cassette motor turned off via the TAPION standard routine and the I/O buffer position reset to zero. The character is then taken from the current I/O buffer position and the position incremented. Finally the character is checked to see if it is the end of file character (1AH). If it is not the routine terminates with the character in register A and FLAG NC. Otherwise the end of file character is placed in CASPRV, so that succeeding sequential input requests will always return the end of file condition, and the routine terminates with FLAG C.

Address... 726DH
This is the dispatcher eof routine for the CAS device. The next character is input (723FH) and placed in CASPRV. It is then tested for the end of file code (1AH) and the result placed in DAC as an integer, zero for false, FFFFH for true.

Address... 727CH
This is the dispatcher putback routine for the CAS device. The character is simply placed in CASPRV to be picked up at the next sequential input request.

Address... 7281H
This routine is used by the dispatcher close function to check if there are any characters in the I/O buffer and then zero the I/O buffer position byte in the FCB.

Address... 728BH
This routine is used by the dispatcher sequential output function to place the character in register A in the I/O buffer at the current I/O buffer position, which is then incremented.

Address... 729BH
This routine is used by the dispatcher sequential input function to collect the character at the current I/O buffer position, which is then incremented.

Address... 72A6H
This table is used by the dispatcher when decoding function codes for the LPT device. It contains the address of the handler for each of the function codes, most are in fact error generators:

```
TO    FUNCTION
71B6H  0, open
71C2H  2, close
6E86H  4, random
72BAH  6, sequential output
475AH  8, sequential input
475AH 10, loc
475AH 12, lof
475AH 14, eof
475AH 16, fpos
475AH 18, putback
```

Address... 72BAH
This is the dispatcher sequential output routine for the LPT device. The character to output is taken from register C and control transfers to the OUTDLP standard routine.

Address... 72BEH
This routine is used by the dispatcher sequential input function to check if a putback character exists in CASPRV, and if not to return Flag Z. Otherwise CASPRV is zeroed and the character tested to see if it is the end of file character (1AH). If not it returns with the character in register A and FLAG NZ,NC. Otherwise the end of file character is placed back in CASPRV and the routine returns with FLAG Z,C.

Address... 72CDH
This routine is used by various dispatcher functions to check if the mode in register E is append, if so a “Bad file name” error is generated (6E6BH).

Address... 72D4H
This routine is used by various dispatcher functions to read a character from the cassette. The character is read via the TAPIN standard routine and a “Device I/O error” generated (73B2H) if FLAG C is returned.

Address... 72DEH
This routine is used by various dispatcher functions to write a character to cassette. The character is written via the TAPOUT standard routine and a “Device I/O error” generated (73B2H) if FLAG C is returned.

Address... 72E9H
This routine is used by various dispatcher functions to turn the cassette motor on for input. The motor is turned on via the TAPION standard routine and a “Device I/O error” generated (73B2H) if FLAG C is returned.

Address... 72F8H
This routine is used by various dispatcher functions to turn the cassette motor on for output, control simply transfers to the TAPOON standard routine.

Address... 7304H
This routine is used by the Interpreter Mainloop “OK” point, the “END” statement handler and the run-clear routine to shut down the printer. PRTFLG is first zeroed and then LPTPOS tested to see if any characters have been output but left hanging in the printer’s line buffer. If so a CR,LF sequence is issued to flush the printer and LPTPOS zeroed.

Address... 7323H
This routine issues a CR,LF sequence to the current output device via the OUTDO standard routine. LPTPOS or TTYPOS is then zeroed depending upon whether the printer or the screen is active.

Address... 7347H
This routine is used by the Factor Evaluator to apply the “INKEY$” function. The state of the keyboard buffer is examined via the CHSNS standard routine. If the buffer is empty the address of a dummy null string descriptor is returned in DAC. Otherwise the next character is read from the keyboard buffer via the CHGET standard routine. After checking that sufficient space is available (6625H) the character is copied to the String Storage Area and the result descriptor created (6821H).

Address... 7367H
This routine is used by the “LIST” statement handler to output a character to the current output device via the OUTDO standard routine. If the character is a LF code then a CR code is also issued.

Address... 7374H
This routine is used by the Interpreter Mainloop to collect a line of text when input is from an I/O buffer rather than the keyboard, that is when a “LOAD” statement is active. Characters are sequentially input (6C71H) and placed in BUF until BUF fills up, a CR is detected or the end of file is reached. All characters are accepted apart from LF codes which are filtered out. If BUF fills up or a CR is detected the routine simply returns the line to the Mainloop. If the end of file is reached while some characters are in BUF the line is returned to the Mainloop. When end of file is reached with no characters in BUF then I/O buffer 0 is closed (6D7BH) and FILNAM checked to determine whether auto-run is required. If not control returns to the Interpreter “OK” point (411EH). Otherwise the system is cleared (629AH) and control transfers to the Runloop (4601H) to execute the program.

Address... 73B2H
This is the “Device I/O error” generator.

Address... 73B7H
This is the “MOTOR” statement handler. If no operand is present control transfers to the STMOTR standard routine with FFH in register A. If the “OFF” token (EBH) follows control transfers with 00H in register A. If the “ON” token (95H) follows control transfers with 01H in register A.

Address... 73CAH
This is the “SOUND” statement handler. The register number operand, which must be less than fourteen, is evaluated (521CH) and placed in register A. The data operand is evaluated (521CH) and bit 7 set, bit 6 reset to avoid altering the PSG auxiliary I/O port modes’ The data operand is placed in register E and control transfers to the WRTPSG standard routine.

Address... 73E4H
This is a single ASCII space used by the “PLAY” statement handler to replace a null string operand with a one character blank string.

Address... 73E5H
This is the “PLAY” statement handler. The address of the “PLAY” command table at 752EH is placed in MCLTAB for the macro language parser and PRSCNT zeroed. The first string operand, which is obligatory, is evaluated (4C64H), its storage freed (67D0H) and its length and address placed in VCBA at bytes 2, 3 and 4. The channel’s stack pointer is initialized to VCBA+33 and placed in VCBA at bytes 5 and 6’ If further text is present in the statement this process is repeated for voices B and C until a maximum of three operands have been evaluated, after this a “Syntax error” is

generated (4055H). If there are less than three string operands present an end of queue mark (FFH) is placed in the queue (7507H) of each unused voice. Register A is then zeroed, to select voice A, and control drops into the play mainloop

.

Address... 744DH

This is the play mainloop. The number of free bytes in the current queue is checked (7521H) and, if less than eight bytes remain, the next voice is selected (74D6H) to avoid waiting for the queue to empty. The remaining length of the operand string is then taken from the current voice buffer and, if zero bytes remain to be parsed, the loop again skips to the next voice (74D6H). Otherwise the current string length and address are taken from the voice buffer and placed in MCLLEN and MCLPTR for the macro language parser. The old stack contents are copied from the voice buffer to the Z80 stack (6253H), MCLFLG is made non-zero and control transfers to the macro language parser (56A2H). The macro language parser will normally scan along the string, using the "PLAY" statement command handlers, until the string is exhausted. However, if a music queue fills up during note generation an abnormal termination is forced back to the play mainloop (748EH) so that the next voice can be processed without waiting for the queue to empty. When control returns normally an end of queue mark is placed in the current queue (7507H) and PRSCNT is incremented to show the number of strings completed. If control returns abnormally then anything left on the Z80 stack is copied into the current voice buffer (6253H). Because of the recursive nature of the macro language parser where the "X" command is involved there may be a number of four byte string descriptors, marking the point where the original string was suspended, left on the Z80 stack at termination. Saving the stack contents in the voice buffer means they can be restored when the loop gets around to that voice again. Note that as there are only sixteen bytes available in each voice buffer an "Illegal function call" error is generated (475AH) if too much data remains on the stack. This will occur when a queue fills up and multiple, nested "X" commands exist, for example:

```
10 A$="XB$;"
20 B$="XC$;"
30 C$="XD$;"
40 D$=STRING$(150,"A")
50 PLAY A$
```

There seems to be a slight bug in this section as only fifteen bytes of stack data are allowed, instead of sixteen, before an error is generated. When control returns from the macro language parser register A is incremented to select the next voice for processing. When all three voices have been processed INTFLG is checked and, if CTRL-STOP has been detected by the interrupt handler, control transfers to the GICINI standard routine to halt all music and terminate. Assuming bit 7 of PRSCNT shows this to be the first pass through the mainloop, that is no voice has been temporarily suspended because of a full queue, PLYCNT is incremented and interrupt dequeueing started via the STRTMS standard routine. PRSCNT is then checked to determine the number of strings completed by the macro language parser. If all three operand strings have been completed the handler terminates, otherwise control transfers back to the start of the play mainloop to try each voice again.

Address... 7507H

This routine is used by the "PLAY" statement handler to place an end of queue mark (FFH) in the current queue via the PUTQ standard routine. If the queue is full it waits until space becomes available.

Address... 7521H

This routine is used by the "PLAY" statement handler to check how much space remains in the current queue via the LFTQ standard routine. If less than eight bytes remain (the largest possible music data packet is seven bytes long) FLAG C is returned.

Address... 752EH

This table contains the valid command letters and associated addresses for the "PLAY" statement commands. Those commands which take a parameter, and consequently have bit 7 set in the table, are shown with an asterisk:

CMD	TO
A	763EH
B	763EH
C	763EH
D	763EH
E	763EH
F	763EH
G	763EH
M*	759EH
V*	7586H
S*	75BEH
N*	7621H
O*	75EFH
R*	75FCH
T*	75E2H
L*	75C8H
X	5782H

Address... 755FH
This table is used by the "PLAY" statement "A" to "G" command handler to translate a note number from zero to fourteen to an offset into the tone divider table at 756EH. The note itself, rather than the note number, is shown below with each offset value:

```
16 ... A-
18 ... A
20 ... A+ or B-
22 ... B or C-
00 ... B+
00 ... C
02 ... C+ or D-
04 ... D
06 ... D+ or E-
08 ... E or F-
10 ... E+
10 ... F
12 ... F+ or G-
14 ... G
16 ... G+
```

Address... 756EH
This table contains the twelve PSG divider constants required to produce the tones of octave 1. For each constant the corresponding note and frequency are shown:

```
3421 ... C  32.698 Hz
3228 ... C+ 34.653 Hz
3047 ... D  36.712 Hz
2876 ... D+ 38.895 HZ
2715 ... E  41.201 Hz
2562 ... F  43.662 Hz
2419 ... F+ 46.243 Hz
2283 ... G  48.997 Hz
2155 ... G+ 51.908 Hz
2034 ... A  54.995 Hz
1920 ... A+ 58.261 Hz
1812 ... B  61.773 Hz
```

Address... 7586H
This is the "PLAY" statement "V" command handler. The parameter, with a default value of eight, is placed in byte 18 of the current voice buffer without altering bit 6 of the existing contents. No music data is generated.

Address... 759EH
This is the "PLAY" statement "M" command handler. The parameter, with a default value of two hundred and fifty-five, is compared with the existing modulation period contained in bytes 19 and 20 of the current voice buffer. If they are the same the routine terminates with no action. Otherwise the new modulation period is placed in the voice buffer and bit 6 set in byte 18 of the voice buffer to indicate that the new value must be incorporated into the next music data packet produced. No music data is generated.

Address... 75BEH
This is the "PLAY" statement "S" command handler. The parameter is placed in byte 18 of the current voice buffer and bit 4 of the same byte set to indicate that the new value must be incorporated into the next music data packet produced. No music data is generated. Because of the PSG characteristics the shape and volume parameters are mutually exclusive so the same byte of the voice buffers is used for both.

Address... 75C8H
This is the "PLAY" statement "L" command handler. The parameter, with a default value of four, is placed in byte 16 of the current voice buffer where it is used in the computation of succeeding note durations. No music data is generated.

Address... 75E2H
This is the "PLAY" statement "T" command handler. The parameter, with a default value of one hundred and twenty, is placed in byte 17 of the current voice buffer where it will be used in the computation of succeeding note durations. ho music data is generated.

Address... 75EFH
This is the "PLAY" statement "O" command handler. The parameter, with a default value of four, is placed in byte 15 of the current voice buffer where it is used in the computation of succeeding note frequencies. No music data is generated.

Address... 75FCH
This is the “PLAY” statement “R” command handler. The length parameter, with a default value of four, is left in register pair DE and a zero tone divider value placed in register pair HL. The existing volume value is taken from byte 18 of the current voice buffer, temporarily replaced with a zero value and control transferred to the note generator (769CH).

Address... 7621H
This is the “PLAY” statement “N” command handler. The obligatory parameter is first examined, if it is zero a rest is generated (760BH). If it is greater than ninety-six an “Illegal function call” error is generated (475AH). Otherwise twelve is repeatedly subtracted from the note number until underflow to obtain an octave number from one to nine in register E and a note number from zero to eleven in register C. Control then transfers to the note generator (7673H).

Address... 763EH
This is the “PLAY” statement “A” to “G” command handler. The note letter is first converted into a note number from zero to fourteen, this extended range being necessary because of the redundancy implicit in the notation. The table at 755FH is then used to obtain the offset into the tone divider table and the divider constant for the note placed in register pair DE. The octave value is taken from byte 15 of the current voice buffer and the divider constant halved until the correct octave is reached. The string operand is then examined directly (56EEH) to determine whether a trailing note length parameter exists. If so it is converted (572FH) and placed in register C. If no parameter exists the default length is taken from byte 16 of the current voice buffer. The duration of the note is then computed from:

```
Duration (Interrupt ticks) = 12,000/(LENGTH*TEMPO)
```

With the normal length value (4) and tempo value (120) this gives a note duration of twenty-five interrupt ticks of 20 ms each or 0.5 seconds. The string operand is then examined (56EEH) for trailing “.” characters and, for each one, the duration multiplied by one and a half. Finally the resulting duration is checked and, if it is less than five interrupt ticks, it is replaced with a value of five. Thus the shortest note that can be generated on a UK machine is 0.10 seconds whatever the tempo or note length. The music data packet, which will be three, five or seven bytes long, is then assembled in bytes 8 to 14 of the current voice buffer prior to placing it in the queue. The duration is placed in bytes 8 and 9 of the voice buffer. The volume and flag byte is taken from byte 18 and placed in byte 10 of the voice buffer with bit 7 set to indicate a volume change to the interrupt dequeuing routine. If bit 6 of the volume byte is set then the modulation period is taken from bytes 19 and 20 and added to the data packet at bytes 11 and 12. If the tone divider value is non-zero then it is added to the data packet at bytes 11 and 12 (without a modulation period) or bytes 13 and 14 (with a modulation period). Finally the byte count is mixed into the three highest bits of byte 8 of the voice buffer to complete the preparation of the music data packet. If the tone divider value is zero, indicating a rest, the contents of SAVVOL are restored to byte 18 of the static buffer. The music data packet is then placed in the current queue via the PUTQ standard routine and the number of free bytes remaining checked (7521H). If less than eight bytes remain control transfers directly to the “PLAY” statement handler (748EH), otherwise control returns normally to the macro language parser.

Address... 7754H
This is the single precision constant 12,000 used in the computation of note duration.

Address... 7758H
This is the “PUT” statement handler. Register B is set to 80H and control drops into the “GET” statement handler.

Address... 775BH
This is the “GET” statement handler. Register B is zeroed, to distinguish “GET” from “PUT” and the next program token examined. Control then transfers to the “PUT SPRITE” statement handler (7AAFH) or the Disk BASIC “GET/PUT” statement handler (6C35H).

Address... 7766H
This is the “LOCATE” statement handler. If a column coordinate is present it is evaluated (521CH) and placed in register D, otherwise the current column is taken from CSRX. If a row coordinate is present it is evaluated (521CH) and placed in register E, otherwise the current row is taken from CSRY. If a cursor switch operand exists it is evaluated (521CH) and register A loaded with 78H for a zero operand (OFF) and 79H for any non-zero operand (ON). The cursor is then switched by outputting ESC, 78H/79H, “5” via the OUTDO standard routine. The row and column coordinates are placed in register pair HL and the cursor position set via the POSIT standard routine.

Address... 77A5H
This is the “STOP ON/OFF/STOP” statement handler. The address of the device’s TRPTBL status byte is placed in register pair HL and control transfers to the “ON/OFF/STOP” routine (77CFH).

Address... 77ABH
This is the “SPRITE ON/OFF/STOP” statement handler. The address of the device’s TRPTBL status byte is placed in register pair HL and control transfers to the “ON/OFF/STOP” routine (77CFH).

Address... 77B1H
This is the “INTERVAL ON/OFF/STOP” statement handler. As there is no specific “INTERVAL” token (control transfers here when an “INT” token is found) a check is first made on the program text for the characters “E” and “R” then the “VAL” token (94H). The address of the device’s TRPTBL status byte is placed in register pair HL and control transfers to the “ON/OFF/STOP” routine (77CFH).

Address... 77BFH
This is the “STRIG ON/OFF/STOP” statement handler. The trigger number, from zero to four, is evaluated (7C08H) and the address of the device’s TRPTBL status byte placed in register pair HL. The “ON/OFF/STOP” token is examined and the TRPTBL status byte modified accordingly (77FEH). Control then transfers directly to the Runloop (4612H) to avoid testing for pending interrupts until the end of the next statement.

Address... 77D4H
This is the “KEY(n) ON/OFF/STOP” statement handler. The key number, from one to ten, is evaluated (521CH) and the address of the devices’ TRPTBL status byte placed in register pair HL. The “ON/OFF/STOP” token is examined and the TRPTBL status byte modified accordingly (77FEH). Bit 0 of the TRPTBL status byte, the ON bit, is then copied into the corresponding entry in FNKFLG for use during the interrupt keyscan and control transfers directly to the Runloop (4612H).

Address... 77FEH
This routine checks for the presence of one of the interrupt switching tokens and transfers control to the appropriate routine: “ON” (631BH), “OFF” (632BH) or “STOP” (6331H). If no token is present a “Syntax error” is generated (4055H).

Address... 7810H
This routine is used by the “ON DEVICE GOSUB” statement handler (490DH) to check the program text for a device token. Unless none of the device tokens is present, in which case Flag C is returned, the device’s TRPTBL entry number is returned in register B and the maximum allowable line number operand count in register C:

DEVICE	TRPTBL#	LINE NUMBERS
KEY	00	10
STOP	10	01
SPRITE	11	01
STRIG	12	05
INTERVAL	17	01

Additionally, for “INTERVAL” only, the interval operand is evaluated (542FH) and placed in INTVAL and INTCNT.

Address... 785CH
This routine is used by the “ON DEVICE GOSUB” statement handler (490DH) to place the address of a program line in TRPTBL. The TRPTBL entry number, supplied in register B, is multiplied by three and added to the table base to point to the relevant entry. The address, supplied in register pair DE, is then placed there LSB first, MSB second.

Address... 786CH
This is the “KEY” statement handler. If the following character is anything other than the “LIST” token (93H) control transfers to the “KEY n” statement handler (78AEH). Each of the ten function key strings is then taken from FNKSTR and displayed via the OUTDO standard routine with a CR,LF (7328H) after each one. The DEL character (7FH) or any control character smaller than 20H is replaced with a space.

Address... 78AEH
This is the “KEY n”, “KEY(n) ON/OFF/STOP”, “KEY ON” and “KEY OFF” statement handler. If the next program text character is “(“ control transfers to the “KEY(n) ON/OFF/STOP” statement handler (77D4H). If it is an “ON” token (95H) control transfers to the DSPFNK standard routine and if it is an “OFF” token (EBH) to the ERAFNK standard routine. Otherwise the function key number is evaluated (521CH) and the key’s FNKSTR address placed in register pair DE’ The string operand is evaluated (4C64H) and its storage freed (67D0H)’ Up to fifteen characters are copied from the string to FNKSTR and unused positions padded with zero bytes. If a zero byte is found in the operand string an “Illegal function call” error is generated (475AH). Control then transfers to the FNKSB standard routine to update the function key display if it is enabled.

Address... 7900H
This routine is used by the Factor Evaluator to apply the “TIME” function. The contents of JIFFY are placed in DAC as a single precision number (3236H).

Address... 790AH
This routine is used by the Factor Evaluator to apply the “CSRLIN” function. The contents of CSRY are decremented and placed in DAC as an integer (2E9AH).

Address... 7911H
This is the “TIME” statement handler. The operand is evaluated (542FH) and placed in JIFFY.

Address... 791BH
This routine is used by the Factor Evaluator to apply the “PLAY” function. The numeric channel selection operand is evaluated (7C08H). If this is zero the contents of MUSICF are placed in DAC as an integer of value zero or FFFFH. Otherwise the channel number is used to select the appropriate bit of MUSICF and this is then converted to an integer as before.

Address... 7940H
This routine is used by the Factor Evaluator to apply the “STICK” function to an operand contained in DAC. The stick number is checked (521FH) and passed to the GTSTCK standard routine in register A. The result is placed in DAC as an integer (4FCFH) .

Address... 794CH
This routine is used by the Factor Evaluator to apply the “STRIG” function to an operand contained in DAC. The trigger number is checked (521FH) and passed to the GTTRIG standard routine in register A. The result is placed in DAC as an integer of value zero or FFFFH.

Address... 795AH
This routine is used by the Factor Evaluator to apply the “PDL” function to an operand contained in DAC. The paddle number is checked (521FH) and passed to the GTPDL standard routine in register A. The result is placed in DAC as an integer (4FCFH).

Address... 7969H
This routine is used by the Factor Evaluator to apply the “PAD” function to an operand contained in DAC. The pad number is checked (521F) and passed to the GTPAD standard routine in register A. The result is placed in DAC as an integer for pads 1, 2, 5 or 6. For pads 0, 3, 4 or 7 the result is placed in DAC as an integer of value zero or FFFFH.

Address... 7980H
This is the “COLOR” statement handler. If a foreground colour operand exists it is evaluated (521CH) and placed in register E, otherwise the current foreground colour is taken from FORCLR. If a background colour operand exists it is evaluated (521CH) and placed in register D, otherwise the current background colour is taken from BAKCLR. If a border colour operand exists it is evaluated (521CH) and placed in BDRCLR. The foreground colour is placed in FORCLR and ATRBYT, the background colour in BAKCLR and control transfers to the CHGCLR standard routine to modify the VDP.

Address... 79CCH
This is the “SCREEN” statement handler. If a mode operand exists it is evaluated (521CH) and passed to the CHGMOD standard routine in register A. If a sprite size operand exists it is evaluated (521CH) and placed in bits 0 and 1 of RG1SAV, the Workspace Area copy of VDP Mode Register 1. The VDP sprite parameters are then cleared via the CLRSPR standard routine. If a key click operand exists it is evaluated (521CH) and placed in CLIKSW, zero to disable the click and non-zero to enable it. If a baud rate operand exists it is evaluated and the baud rate set (7A2DH). If a printer mode operand exists it is evaluated (521CH) and placed in NTMSXP, zero for an MSX printer and non-zero for a general purpose printer.

Address... 7A2DH
This routine is used to set the cassette baud rate. The operand is evaluated (521CH) and five bytes copied from CS1200 or CS2400 to LOW as appropriate.

Address... 7A48H
This is the “SPRITE” statement handler. If the next character is anything other than a “$” control transfers to the “SPRITE ON/OFF/STOP” statement handler (77ABH). SCRMOD is then checked and an “Illegal function call” error generated (475AH) if the screen is in 40x24 Text Mode. The sprite pattern number is evaluated and its location in the

VRAM Sprite Pattern Table obtained (7AA0H). The string operand is then evaluated (4C5FH) and its storage freed (67D0H). The sprite size, obtained via the GSPSIZ standard routine, is compared with the string length and, if the string is shorter than the sprite, the Sprite Pattern Table entry is first filled with zeroes via the FILVRM standard routine. Characters are then copied from the string body to the Sprite Pattern Table via the LDIRVM standard routine until the string is exhausted or the sprite is full. If the string is longer than the sprite size any excess characters are ignored.

Address... 7A84H
This routine is used by the Factor Evaluator to apply the "SPRITE$" function. The sprite pattern number is evaluated and its location in the VRAM Sprite Pattern Table obtained (7A9FH). The sprite size, obtained via the GSPSIZ standard routine, is then placed in register pair BC to control the number of bytes copied. After checking that sufficient space is available in the String Storage Area (6627H) the sprite pattern is copied from VRAM via the LDIRMV standard routine and the result descriptor created (6654H). Note that as no check is made on the screen mode during this function some interesting side effects can be found, see below.

Address... 7A9FH
This routine is used by the "SPRITE$" statement and function to locate a sprite pattern in the VRAM Sprite Pattern Table. The pattern number operand is evaluated (7C08H) and passed to the CALPAT standard routine in register A. The pattern address is placed in register pair DE and the routine terminates. Note that no check is made on the pattern number magnitude for differing sprite sizes. Pattern numbers up to two hundred and fifty-five are accepted even in 16x16 sprite mode when the maximum pattern number should be sixty-three. As a result VRAM addresses greater than 3FFFH will be produced which will wrap around into low VRAM. With the "SPRITE$" statement this will corrupt the Character Generator Table, for example:

```
10 SCREEN 3,2
20 SPRITE$(0)=STRING$(32,255)
30 PUT SPRITE 0,(0,0), ,0
40 SPRITE$(65)=STRING$(32,255)
50 GOTO 50
```

The above puts a real sprite in the top left of the screen and then uses an illegal statement in line 40 to corrupt the VRAM just to the right of it. The "SPRITE$" function can also be manipulated in this way and, as there is no screen mode check, up to thirty-two bytes of the Name Table can be read in 40x24 Text Mode, for example:

```
10 SCREEN 0,2
20 PRINT"something"
30 A$=SPRITE$(64)
40 PRINT A$
```

Address... 7AAFH
This is the "GET/PUT SPRITE" statement handler, control is transferred here from the general "GET/PUT" statement handler (775BH). Register B is first checked to make sure that the statement is "PUT" and an "Illegal function call" error generated (475AH) if otherwise. SCRMOD is then checked and an "Illegal function call" error generated (475AH) if the screen is in 40x24 Text Mode. The sprite number operand, from zero to thirty-one, is evaluated (521CH) and passed to the CALATR standard routine to locate the four byte attribute block in the Sprite Attribute Table. If a coordinate operand exists it is evaluated and the X coordinate placed in register pair BC, the Y coordinate in register pair DE (579CH). The Y coordinate LSB is written to byte 0 of the attribute block in VRAM via the WRTVRM standard routine. Bit 7 of the X coordinate is then examined to determine whether it is negative, that is off the left hand side of the screen. If so thirty two is added to the X coordinate and register B is set to 80H to set the early clock bit in the attribute block. For example an X coordinate of -1 (FFFFH) would be changed to +31 with an early clock. The X coordinate LSB is then written to byte 1 of the attribute block via the WRTVRM standard routine. Byte 3 of the attribute block is read in via the RDVRM standard routine, the new early clock bit is mixed in and it is then written back to VRAM via the WRTVRM standard routine. If a colour operand is present it is evaluated (521CH), byte 3 of the attribute block is read in via the RDVRM standard routine the new colour code is mixed into the lowest four bits and it is written back to VRAM via the WRTVRM standard routine. If a pattern number operand exists it is evaluated (521CH) and checked for magnitude against the current sprite size provided by the GSPSIZ standard routine. The maximum allowable pattern number is two hundred and fifty-five for 8x8 sprites and sixty-three for 16x16 sprites. The pattern number is written to byte 2 of the attribute block via the WRTVRM standard routine and the handler terminates.

Address... 7B37H
This is the "VDP" statement handler. The register number operand, from zero to seven, is evaluated (7C08H) followed by the data operand (521CH). The register number is placed in register C, the data value in register B and control transferred to the WRTVDP standard routine.

Address... 7B47H
This routine is used by the Factor Evaluator to apply the "VDP" function. The register number operand, from zero to eight, is evaluated (7C08H) and added to RGOSAV to locate the corresponding register image in the Workspace Area. The VDP register image is then read and placed in DAC as an integer (4FCFH).

Address... 7B5AH
This is the "BASE" statement handler. The VDP table number operand, from zero to nineteen, is evaluated (7C08H) followed by the base address operand (4C64H). After checking that the base address is less than 4000H (7BFEH) the VDP table number is used to locate the associated entry in the masking table at 7BA3H. The base address is ANDed with the mask and an "Illegal function call" error generated (475AH) if any illegal bits are set. The VDP table number is then added to TXTNAM to locate the current base address in the Workspace Area and the new base address placed there. The VDP table number is divided by five to determine which of the four screen modes the table belongs to. If this is the same as the current screen mode the new base address is also written to the VDP (7B99H).

Address... 7B99H
This routine is used by the "BASE" statement handler to update the VDP base addresses. The current screen mode, in register A, is examined and control transfers to the SETTXT, SETT32, SETGRP or SETMLT standard routine as appropriate. Note that this is not a full VDP initialization and that the four current table addresses (NAMBAS, CGPBAS, PATBAS and ATRBAS) which are the ones actually used by the screen routines, are not updated. This can be demonstrated with the following, where the Interpreter carries on outputting to the old VRAM Name Table:

```
10 SCREEN 0
20 BASE(0)=&H400
30 PRINT"something"
40 FOR N=1 TO 2000:NEXT
50 BASE(0)=0
```

Note also that this routine contains a bug. While SETTXT is correctly used for 40x24 Text Mode, SETGRP is used for 32x24 Text Mode and SETMLT for Graphics Mode and Multicolour Mode. Any "BASE" statement should therefore be immediately followed by a "SCREEN" statement to perform a full initialization.

Address... 7BA3H
This masking table is used by the "BASE" statement handler to ensure that only legal VDP base addresses are accepted. The table number and corresponding Workspace Area variable are shown with each mask:

MASK	TABLE
03FFH	00, TXTNAM
003FH	01, TXTCOL
07FFH	02, TXTCGP
007FH	03, TXTATR
07FFH	04, TXTPAT
03FFH	05, T32NAM
003FH	06, T32COL
07FFH	07, T32CGP
007FH	08, T32ATR
07FFH	09, T32PAT
03FFH	10, GRPNAM
1FFFH	11, GRPCOL
1FFFH	12, GRPCGP
007FH	13, GRPATR
07FFH	14, GRPPAT
03FFH	15, MLTNAM
003FH	16, MLTCOL
07FFH	17, MLTCGP
007FH	18, MLTATR
07FFH	19, MLTPAT

Address... 7BCBH
This routine is used by the Factor Evaluator to apply the "BASE" function. The VDP table number operand, from zero to nineteen, is evaluated (7C08H) and added to TXTNAM to locate the required Workspace Area base address. This is then placed in DAC as a single precision number (3236H).

Address... 7BE2H
This is the "VPOKE" statement handler. The VRAM address operand is evaluated (4C64H) and checked to ensure that it is less than 4000H (7BFEH). The data operand is then evaluated (521CH) and passed to the WRTVRM standard routine in register A to write to the required address.

Address... 7BF5H
This routine is used by the Factor Evaluator to apply the “VPEEK” function to an operand contained in DAC. The VRAM address operand is checked to ensure it is less than 4000H (7BFEH). VRAM is then read via the RDVRM standard routine and the result placed in DAC as an integer (4FCFH).

Address... 7BFEH
This routine converts a numeric operand in DAC to an integer (2F8AH) and places it in register pair HL. If the operand is equal to or greater than 4000H, and thus outside the allowable VRAM range, an “Illegal function call” error is generated (475AH).

Address... 7C08H
This routine evaluates (521CH) a parenthesized numeric operand and returns it as an integer in register A. If the operand is greater than the maximum allowable value initially supplied in register A an “Illegal function call” error is generated (475AH).

Address... 7C16H
This is the “DSKO$” statement handler. An “Illegal function call” error is generated (475AH) on a standard MSX machine.

Address... 7C1BH
This is the “SET” statement handler. An “Illegal function call” error is generated (475AH) on a standard MSX machine.

Address... 7C20H
This is the “NAME” statement handler. An “Illegal function call” error is generated (475AH) on a standard MSX machine.

Address... 7C25H
This is the “KILL” statement handler. An “Illegal function call” error is generated (475AH) on a standard MSX machine.

Address... 7C2AH
This is the “IPL” statement handler. An “Illegal function call” error is generated (475AH) on a standard MSX machine.

Address... 7C2FH
This is the “COPY” statement handler. An “Illegal function call” error is generated (475AH) on a standard MSX machine.

Address... 7C34H
This is the “CMD” statement handler. An “Illegal function call” error is generated (475AH) on a standard MSX machine.

Address... 7C39H
This routine is used by the Factor Evaluator to apply the “DSKF” function to an operand contained in DAC. An “Illegal function call” error is generated (475AH) on a standard MSX machine.

Address... 7C3EH
This routine is used by the Factor Evaluator to apply the “DSKI$” function. An “Illegal function call” error is generated (475AH) on a standard MSX machine.

Address... 7C43H
This routine is used by the Factor Evaluator to apply the “ATTR$” function. An “Illegal function call” error is generated (475AH) on a standard MSX machine.

Address... 7C48H
This is the “LSET” statement handler. An “Illegal function call” error is generated (475AH) on a standard MSX machine.

Address... 7C4DH
This is the “RSET” statement handler. An “Illegal function call” error is generated (475AH) on a standard MSX machine.

Address... 7C52H
This is the “FIELD” statement handler. An “Illegal function call” error is generated (475AH) on a standard MSX machine.

Address... 7C57H
This routine is used by the Factor Evaluator to apply the “MKI$” function to an operand contained in DAC. An “Illegal function call” error is generated (475AH) on a standard MSX machine.

Address... 7C5CH
This routine is used by the Factor Evaluator to apply the “MKS$” function to an operand contained in DAC. An “Illegal function call” error is generated (475AH) on a standard MSX machine.

Address... 7C61H
This routine is used by the Factor Evaluator to apply the “MKD$” function to an operand contained in DAC. An “Illegal function call” error is generated (475AH) on a standard MSX machine.

Address... 7C66H
This routine is used by the Factor Evaluator to apply the “CVI” function to an operand contained in DAC. An “Illegal function call” error is generated (475AH) on a standard MSX machine.

Address... 7C6BH
This routine is used by the Factor Evaluator to apply the “CVS” function to an operand contained in DAC. An “Illegal function call” error is generated (475AH) on a standard MSX machine.

Address... 7C70H
This routine is used by the Factor Evaluator to apply the “CVD” function to an operand contained in DAC. An “Illegal function call” error is generated (475AH) on a standard MSX machine.

Address... 7C76H
This routine completes the power-up initialization. At this point the entire Workspace Area is zeroed and only EXPTBL and SLTTBL have been initialized. A temporary stack is set at F376H and all one hundred and twelve hooks (560 bytes) filled with Z80 RET opcodes (C9H). HIMEM is set to F380H and the lowest RAM location found (7D5DH) and placed in BOTTOM. The one hundred and forty-four bytes of data commencing at 7F27H are copied to the Workspace Area from F380H to F40FH The function key strings are initialized via the INIFNK standard routine, ENDBUF and NLONLY are zeroed and a comma is placed in BUFMIN and a colon in KBFMIN. The address of the MSX ROM character set is taken from locations 0004H and 0005H and placed in CGPNT+1 and PRMPRV is set to point to PRMSTK. Dummy values are placed in STKTOP, MEMSIZ and VARTAB (their correct values are not known yet), one I/O buffer is allocated (7E6BH) and the Z80 SP set (62E5H). A zero byte is placed at the base of RAM, TXTTAB is set to the following location and a “NEW” executed (6287H). The VDP is then initialized via the INITIO, INIT32 and CLRSPR standard routines, the cursor coordinates are set to row 11, column 10 and the sign on message “MSX system etc.” is displayed (6678H). After a three second delay a search is carried out for any extension ROMs (7D75H) and a further “NEW” executed (6287H) in case a BASIC program has been run from ROM. Finally the identification message “MSX BASIC etc.” is displayed (7D29H) and control transfers to the Interpreter Mainloop “OK” point 411FH.

Address... 7D29H
This routine is used during power-up to enable the function key display, place the screen in 40x24 Text Mode via the INITXT standard routine, and display (6678H) the identification message “MSX BASIC etc.”. The amount of free memory is then computed by subtracting the contents of VARTAB from the contents of STKTOP and displayed (3412H) followed by the “Bytes free” message.

Address... 7D5DH
This routine is used during power-up to find the lowest RAM location. Starting at EF00H each byte is tested until one is found that cannot be written to or an address of 8000H is reached. The base address, rounded upwards to the nearest 256 byte boundary, is returned in register pair HL.

Address... 7D75H
This routine is used during power-up to perform an extension ROM search. Pages 1 and 2 (4000H to BFFFH) of each slot are examined and the results placed in SLTATR. An extension ROM has the two identification characters “AB” in the first two bytes to distinguish it from RAM. Information about its properties is also present in the first sixteen bytes as follows:

Reserved	Byte 10-15
BASIC Text Address MSB	Byte 9
BASIC Text Address LSB	Byte 8
DEVICE Address MSB	Byte 7
DEVICE Address LSB	Byte 6
STATEMENT Address MSB	Byte 5
STATEMENT Address LSB	Byte 4
INITIALIZE Address MSB	Byte 3
INITIALIZE Address LSB	Byte 2
42H ('B')	Byte 1
41H ('A')	Byte 0

Figure 48: ROM Header.

Each page in a given slot is examined by reading the first two bytes (7E1AH) and checking for the "AB" characters. If a ROM is present the initialization address is read (7E1AH) and control passed to it via the CALSLT standard routine. With a games ROM there may be no return to BASIC from this point. The "CALL" extended statement handler address is then read (7E1AH) and bit 5 of register B set if it is valid, that is non-zero. The extended device handler address is read (7E1AH) and bit 6 of register B set if it is valid. Finally the BASIC program text address is read (7E1AH) and bit 7 of register B set if it is valid. Register B is then copied to the relevant position in SLTATR and the search continued until no more slots remain. SLTATR is then examined for any extension ROM flagged as containing BASIC program text. If one is found its position in SLTATR is converted to a Slot ID (7E2AH) and the ROM permanently switched in via the ENASLT standard routine. VARTAB is set to C000H, as it is not known how large the Program Text Area is, TXTTAB is set to 8008H and BASROM made non-zero to disable the CTRL-STOP key. The system is cleared (629AH) and control transfers to the Runloop (4601H) to execute the BASIC program.

Address... 7E1AH
This routine is used to read two bytes from successive locations in an extension ROM. The initial address is supplied in register pair HL and the Slot ID in register C. The bytes are read via the RDSLT standard routine and returned in register pair DE. If both are zero FLAG Z is returned.

Address... 7E2AH
This routine converts the SLTATR position supplied in register B into the corresponding Slot ID in register C and ROM base address in register H. The position is first modified so that it runs from 0 to 63 rather than from 64 to 1, so that the required information is present in the form:

7	6	5	4	3	2	1	0
0	0	PSLOT #		SSLOT #		PAGE #	

Figure 49

Bits 0 and 1 are shifted into the highest two bits of register H to form the address. Bits 4 and 5 are shifted to bits 0 and 1 of register C to form the Primary Slot number. Bits 2 and 3 are shifted to bits 2 and 3 of register C to form the Secondary Slot number and bit 7 of the corresponding EXPTBL entry copied to bit 7 of register C.

Address... 7E4BH
This is the "MAXFILES" statement handler. As control transfers here when a "MAX" token (CDH) is detected the program text is first checked for a trailing "FILES" token (B7H). The buffer count operand, from zero to fifteen, is then evaluated (521CH) and any existing buffers closed (6C1CH). The required number of I/O buffers are allocated (7E6BH), the system is cleared (62A7H) and control transfers directly to the Runloop (4601H).

Address... 7E6BH
This is the I/O buffer allocation routine. It is used during power-up and by the "MAXFILES" and "CLEAR" statement handlers to allocate storage for the number of I/O buffers supplied in register A. Two hundred and sixty-seven bytes are subtracted from the contents of HIMEM for every buffer to produce a new MEMSIZ value. The size of the existing String Storage Area (initially two hundred bytes) is computed by subtracting the old contents of STKTOP from the old contents of MEMSIZ, this is then subtracted from the new MEMSIZ value to produce the new STKTOP value. A further one hundred and forty bytes are subtracted for the Z80 stack and an "Out of memory" error generated (6275H) if this address is lower than the start of the Variable Storage Area. Otherwise the buffer count is placed in MAXFIL and MEMSIZ and STKTOP set to their new values. The caller's return address is popped, the Z80 SP set to the new position and the return address pushed back onto the stack. FILTAB is then set to the start of the I/O buffer pointer block and each pointer set to point to the associated FCB. Finally the address of I/O buffer 0, the Interpreter's "LOAD" and "SAVE" buffer, is placed in NULBUF and the routine terminates.

Address... 7ED8H
This is the plain text message "MSX system" terminated by a zero byte.

Address... 7EE4H
This is the plain text message “version 1.0” CR,LF terminated by a zero byte.
Address... 7EF2H
This is the plain text message “MSX BASIC “ terminated by a zero byte.

Address... 7EFDH
This is the plain text message “Copyright 1983 by Microsoft” CR,LF terminated by a zero byte.

Address... 7F1BH
This is the plain text message “ Bytes free” terminated by a zero byte.

Address... 7F27H
This block of one hundred and forty-four data bytes is used to initialize the Workspace Area from F380H to F40FH.

Address... 7FB7H
This seven byte patch fixes a bug in the external device parsing routine (55F8H). It checks for a zero length device name in register A and changes it to one if necessary.

Address... 7FBEH
This section of the ROM is unused and filled with zero bytes.