10 ROM Routines

Many of the tasks which need to be performed when dealing with the BASIC system are handled by standard routines inside the BASIC ROM. There are standard routines for expression evaluation, checking the syntax of lines, handling the memory allocation, and arithmetic routines. Although some of these will only be of use inside new statements and functions (like the 'Get character at PTRB' routine); many can be used from simple machine code programs, to allow floating point calculations to be performed, or accessing the variables passed by the BASIC 'CALL' statement, perhaps.

Note that these ROM routines can only be used if BASIC is paged in to &8000 to &BFFF. If the machine code program which uses them will be called from BASIC, using either the 'CALL' statement or the 'USR' function, BASIC will be paged in. The programs in chapters 7 to 9 rely on this. However, BASIC will not be paged in if the program is called by using the '*RUN' command in any filing system which itself sits in a paged ROM (like DFS, for example): the filing system ROM will be paged in instead.

To check that the current paged-in ROM is BASIC, the RAM copy of the paged ROM select register (in location &F4) should be compared with the ROM number of the BASIC ROM. This can be found by using OSBYTE &BB (187). For example, this section of code will check that the current ROM is BASIC:

```
LDA #&BB          \Call OSBYTE &BB to read the ROM
LDY #&FF          \ socket number containing BASIC.
LDX #&00          \ X and Y are set to read it without
JSR osbyte        \ modification.
CPX &F4           \If it is not the same as the current
BNE giveup        \ ROM, don't continue.
```

The BASIC ROM does not need to be paged in if the only part of the machine code program which is to be 'RUN' is the initialisation section, and that just needs to check the year of the BASIC ROM (but uses no ROM routines). If this is the case, the BASIC ROM slot number can be found using OSBYTE &BB as

above, and the year byte found by using OSRDRM (&FFB9). For example, the following code will read the year byte of the BASIC ROM:

```
LDA #&BB          \Call OSBYTE &BB to read the ROM
LDY #&FF          \ socket number containing BASIC.
LDX #&00          \ X and Y are set to read it without
JSR osbyte        \ modification.
TXA               \
TAY               \Transfer the ROM number into Y,
LDA #&80          \ and call OSRDRM to read the byte
STA &E7           \ at location &8015 in the BASIC ROM
LDA #&15          \
STA &E6           \
JSR &FFB9         \
```

Note that OSRDRM was implemented for operating the '*ROM' filing system in paged ROMs, so use it with caution (as with most of the rest of the examples in this book!).

Throughout this section, I have used the names of many of the standard BASIC registers, rather than the actual memory they occupy. They are detailed in other parts of this book, but here is a summary of them:

IntA This is the integer accumulator which is held in page zero at &2A to &2D (LSB in &2A, MSB in &2D). It is used in integer calculations, and also to pass integer values between routines.

The low 3 bytes of IntA (&2A to &2C) are also used to hold the variable descriptor block when handling variables. When being used for this, &2A and &2B point to the first byte of the variable value, and &2C contains the variable type (for a description of the variable types, see section 3.1.3). This variable desrcriptor block is sometimes used at &37 to &39 (if IntA is used to hold the value of the variable).

FPA This is the main floating point accumulator, which is held in page zero at &2E to &35 (see section 2.2.2 for the floating point accumulator format). It is used in calculations involving real numbers (together with FPB), and also to pass real values between routines.

FPB This is the secondary floating point accumulator, which is held in page zero at &3B to &42. It is involved in most floating point calculations.

StrA This is the string accumulator, which is held in page 6 (&600 to &6FF). The current length of the string is held in location &36 in page zero. It is used in string manipulations, and to pass string values between routines.

PTRA This is the primary text pointer. The base of the pointer is held in page zero in &B and &C, with the offset in &A. This is used mainly to parse the keyword at the start of a statement.

PTRB This is the secondary text pointer. The base is held in &19 and &IA, with the offset in &1B. This is used mainly for expression evaluation.

STACK This is the BASIC STACK which works downwards in memory from HIMEM. The STACK pointer is held in page zero in &4 and &5. It is used mainly to hold temporary results of calculations, and to save old values of parameters inside FNs and PROCs (see section 5.3).

HEAP This is the BASIC variable HEAP which works upwards in memory from LOMEM. The HEAP pointer is held in page zero in &2 and &3. It is used to hold variables and FN and PROC locations (once found).

Summary

This list is a summary of the routines documented in this section, split into functional groups. Some of the routines have other entry points which are not listed here, but are included with the full description of the routine. For a summary of the ROM in numerical order, see appendix B.

	BASIC1	BASIC2	
10.1 Restarting BASIC			
cstart	8A80	8ADD	Cold start
wstart	8A96	8AF3	Warm start
istart	8A99	8AF6	Enter immediate mode
10.2 Program handling			
tline	88D9	8957	Tokenise a line
inslin	BCAA	BC8D	Insert line in program
dellin	BC4A	BC2D	Delete line in program
schlin	9942	9970	Search for program line
run	BD2C	BD14	Run a program
clear	BD38	BD20	Clear variablesfistacks
clrstk	BD52	BD3A	Reset stacks and restore data
seterl	B3F6	B3C5	Set up ERL to line in error
settop	BE88	BE6F	Set up TOP, check ' Bad program'
10.3 Statement handling			
getcha	8A1E	8A97	Get character at PTRA
getchb	8A13	8A8C	Get character at PTRB
chksda	9810	9857	Check end of statement
cont	8B0C	8B9B	Continue execution
skipln	8AED	8B7D	Skip rest of line

10.4 Expression evaluation

getnsb	9B03	9B29	Get <numeric> or <string>
getfsb	AE1B	ADEC	Get <factor> or <string-factor>
getnmb	A06C	A07B	Get number at PTRB
getina	97AE	97DF	Get a tokenised line number

10.5 VariabIe/FN/PROC management

fndvar	95A9	95DD	Find variable
rdvar	B35B	B32C	Read value of variable
asvar	8BD3	8C21	Assign string variable
asvark	B4E0	B4B4	Assign numeric variable
schvar	9429	9469	Search for variable in list
linkvar	94BC	94FC	Link in new variable
scnvn	951F	9559	Scan variable name
schfnp	941B	945B	Search for FN/PROC in list
lnkfnp	94AD	94ED	Link in new FN/PROC
clrib	94F7	9531	Clear space for new block

10.6 STACK management

pusha	BDA8	BD90	Push IntA, FPA, or StrA
pushi	BDAC	BD94	Push IntA
pushf	BD69	BD51	Push FPA
pushs	BDCA	BDB2	Push StrA
chksp	BE4C	BE34	Check for STACK/HEAP clash
popi	BE02	BDEA	Pop IntA
popi0	BE25	BE0D	Pop integer into page zero
popf	BD96	BD7E	Pop real number; set up (&4B)
pops	BDE3	BDCB	Pop StrA
pshvvd	B33C	B30D	Push value and descriptor
poppar	8C5B	8CC1	Pop parameter value

10.7 Input/output

inputs	BC17	BBFC	Input string to StrA
pchar	B571	B558	Print A as a character
ptoken	B53A	B50E	Print A as a character or token
phex	8570	B545	Print A as a HEX number
plnum0	98F1	991F	Print line number
pnewl	BC42	BC25	Print a CRLF (newline)

10.8 Type conversion

citof	A2AF	A2BE	Convert integer to real
catof	A2DE	A2ED	Convert A to a real number
cftoi	A3F2	A3E4	Convert real to integer
cntos	9ED0	9EDF	Convert number to string
cston	AC5A	AC34	Convert string to number

10.9 Integer routines

lodiay	AF19	AEEA	Load IntA with A, Y
lodi0	AF85	AF56	Load IntA from 00,X−03,X
stori0	BE5C	BE44	Store IntA at 00,X—03,X
negi	ADB5	AD93	Negate IntA
absi	AD94	AD71	Take ABS value of IntA
divi	99C0	99E8	Perform integer division

10.10 Floating point routines

movfab	A20F	A21E	Move FPA into FPB
movfba	A4E4	A4DC	Move FPB into FPA
ldfantl	A691	A686	Set FPA to zero
ldfanl	A6A4	A699	Set FPA to 1
ldfbn0	A463	A453	Set FPB to zero

ldfam	A3A6	A3B5	Load FPA from (&4B)
ldfbm	A33F	A34E	Load FPB from (&4B)
stfam	A37E	A38D	Store FPA at (&4B)
exfam	A4DE	A4D6	Exchange FPA with (&4B)
pntmt1	A7FB	A7F5	Point &4B,&4C at &46C
pntmt2	A7F3	A7ED	Point &4B,&4C at &471
pntmt3	A7F7	A7F1	Point &4B,&4C at &476
pntmt4	A7EF	A7E9	Point &4B,&4C at &47B
tstfa	A1CB	A1DA	Test FPA
nmlfa	A2F4	A303	Normalise FPA
rcofa	A667	A65C	Round FPA & check overflow
negfa	ADA0	AD7E	Negate FPA
addfba	A513	A50B	Add FPB to FPA
mulfab	A61E	A613	Multiply FPA by FPB
mufa10	A1E5	A1F4	Multiply FPA by 10
divfab	A6FC	A6F1	Divide FPA by FPB
dvfa10	A23E	A24D	Divide FPA by 10
series	A889	A897	Perform series evaluation
fixfa	A40C	A3FE	Convert FPA to fixed format
fracfa	A494	A486	Extract fractional part of FPA

10.11 Function entry points

Listed in section 10.11)

10.1 RESTARTING BASIC

These entry points allow BASIC to be re-started, rather than continuing with the execution of the program currently running. This may be necessary if, for example, the program has been altered or corrupted by the statement just executed (like DELETE, for example).

cstart − Cold start

Execution addr

BASIC1 &8A80
BASIC2 &8ADD

Entry conditions:

PAGE points to the program area to be used

HIMEM points to the top of available memory

Exit conditions:

NON−RETURNING

Description

This entry has exactly the same effect as the BASIC ‘NEW’ command. It turns TRACE off, places the sequence &0D &FF in memory at PAGE, and sets TOP to be PAGE+2, before executing a warm start.

Other entry points

NONE

wstart − Warm start

Execution addr

BASIC1 &8A96
BASIC2 &8AF3

Entry conditions:

Resident program at PAGE

TOP points to the next available byte after the program

HIMEM points to the top of available memory

Exit conditions:

NON-RETURNING

Description

LOMEM and HEAP are set to TOP, the variables and FN/ PROC lists are cleared, and STACK is reset to HIMEM. BASIC then enters immediate mode, and waits for a line to be input.

Other entry points

NONE

istart − Enter immediate mode

Execution addr

BASIC1 &8A99
BASIC2 &8AF6

Entry conditions:

Resident program at PAGE

TOP points to the next available byte after the program LOMEM, HIMEM delimit the HEAP/ST ACK memory to be used

Exit conditions:

NON-RETURNING

Description

This entry has the same effect as the BASIC 'END' statement. The 'ON ERROR' pointer is reset, and a line is input into the keyboard buffer. If this starts with a line number, it is inserted into the program; otherwise the line is executed as an immediate command.

Other entry points

NONE

10.2 PROGRAM HANDLING

These are general routines for manipulating the program currently in memory. Note that if the program is altered by inserting or deleting any lines, the HEAP may be corrupted, so a ‘Warm start’ should be executed to return to immediate mode and clear the variables.

tline − Tokenise a line

Execution addr

BASIC1	&88D9
BASIC2	&8957

Entry conditions:

Y	0
(&37)	points to start of line to be tokenised
&3B	start of statement flag: 0 = ‘at start’
&3C	line number flag: 0 = don’t tokenise line numbers

Exit conditions:

Tokenised line starting at original position

&37 − &3D	undefined
A	undefined
X	undefined
Y	undefined
C	undefined

Description

This routine tokenises the line pointed to by the pointer at &37,&38 and terminated by a carriage return. The tokeniser can be in several states initially, and these states are set by the flags in &3B and &3C before entering the routine. &3B tells the tokeniser if it is at the start of a statement (if a ‘*’ is at the start, the rest of

the line is not tokenised); and &3C tells the tokeniser whether to tokenise any numbers it finds, or to leave them as ASCII. The tokeniser follows several rules, and encountering a keyword (or not) may change the state. See section 2.3 for more on tokenising.

Other entry points

1 tline0 − Tokenise start of statement, no line numbers

BASIC1 &88D3
BASIC2 &8951

This entry point sets both of the tokenising flags to zero, and zeros Y, before entering the main routine (i.e. tokenise from the start of a statement, but don't tokenise line numbers).

inslin − Insert line in program

Execution addr

BASIC1 &BCAA
BASIC2 &BC8D

Entry conditions:

Y	offset from &700 of first character of line text
IntA:	line number of line to be inserted
&700−	line to be inserted (keyboard buffer)

Exit conditions:

&37−&3E	undefined
TOP	new top of program
A	&0D
X	undefined
Y	undefined
C	1

Description

This routine inserts a line into the current program. On entry, the line to be inserted should be in the keyboard buffer (at &700 to &7FF), terminated by a carriage return. Y should point to the first character of the line to be inserted into the program (so that the line number itself can be missed out). The low 2 bytes of IntA should contain the line number. The routine will delete the old line if necessary, and then insert the new one if it is not empty. If there is not enough room for the line to be inserted, a ‘LINE space’ error (ERR = 0) will be generated.

Other entry points

NONE

dellin − Delete line in program

Execution addr

BASIC1	&BC4A
BASIC2	&BC2D

Entry conditions:

IntA:	line number of line to be deleted

Exit conditions:

&37,&38	undefined
&3D,&3E	undefined
TOP	new top of program
A	undefined
X	undefined
Y	undefined
C	0=line deleted, 1=line not found

Description

This routine deletes a line from the current program. On entry, the line number of the line to be deleted should be in the low 2 bytes of IntA (at &2A,&2B). If the line could not be found, the routine will exit with C set; otherwise, the line will be deleted, and the routine will exit with C clear.

Other entry points

NONE

schlin − Search for line in program

Execution addr

BASIC1 &9942
BASIC2 &9970

Entry conditions:

IntA: line number of line to be found

Exit conditions:

C 0=line found, 1 =line not found

If C=0, (&3D) points at length byte of line found
If C= 1, (&3D) points at end of last smaller line

A undefined
X preserved
Y 2

Description

This routine searches for a line in the program, given the line number in IntA. If it is found, the pointer at &3D ,&3E is set to point to the length byte of the line (i.e. 1 before the text of the line), and C is cleared. If it is not found, C is set, and the pointer at &3D ,&3E is left pointing at the carriage return on the end of the last line that had a smaller line number than the one being searched for.

Other entry points

NONE

run − Run a program

Execution addr

BASIC1 &BD2C
BASIC2 &BD14

Entry conditions:

Resident program at PAGE

Exit conditions:

NON-RETURNING

Description

This entry point does the same as the BASIC statement ‘RUN’. It clears the variables (apart from the resident integers) and stacks, and starts executing the program from the beginning.

Other entry points

1 gstart − Goto start of program

BASIC1 &BD2F
BASIC2 &BD17

This entry point starts executing the BASIC program in memory at PAGE, but it does not clear the varibles or stacks first.

clear − Clear variables and stacks

Execution addr

BASIC1 &BD38
BASIC2 &BD20

Entry conditions:

Valid PAGE, TOP, HIMEM

Exit conditions:

variables cleared

REPEAT, GOSUB, FOR stacks cleared

DATA pointer restored to PAGE

LOMEM	set to TOP
HEAP	set to TOP
STACK	set to HIMEM

A	0
X	0
Y	preserved
C	preserved

Description

This routine clears all variables and FN/PROC lists (except for the resident integers), and resets the HEAP and all BASIC stacks. It does the same as the BASIC ‘CLEAR’ statement.

Other entry points

NONE

clrstk − Reset stacks, restore data

Execution addr

BASIC1	&BD52
BASIC2	&BD3A

Entry conditions:

Valid PAGE, HIMEM

Exit conditions:

REPEAT, GOSUB, FOR stacks cleared

DATA pointer restored to PAGE

STACK	set to HIMEM

A	0
X	preserved
Y	preserved
C	preserved

Description

This routine resets the BASIC stacks, and restores the DATA pointer to PAGE.

Other entry points

NONE

seterl − Set up ERL

Execution addr

BASIC1 &B3F6
BASIC2 &B3C5

Entry conditions:

PTRA: base points to position of error

Exit conditions:

&8,&9 line number of error (ERL)

A undefined
X undefined
Y undefined
C undefined

Description

This routine searches through the program, keeping track of the current line number, until it finds the line which the base of PTRA points to. It then sets ERL to the number of this line.

Other entry points

NONE

settop − Set up TOP, check ‘Bad program’

Execution addr

BASIC1 &BE88
BASIC2 &BE6F

Entry conditions:

BASIC program at PAGE

Exit conditions:

&12,&13	points to the end of the program (TOP)
A	undefined
X	preserved
Y	1
C	undefined

Description

This routine scans through the current program in memory, and sets TOP to point to the next free memory location after the end of it. If it could not follow the length bytes through to the end of the program, a ‘Bad program’ message will be generated, and a JMP will be made to immediate mode (istart).

Other entry points

NONE

10.3 STATEMENT HANDLING

These routines allow general handling of statements, using the syntax pointers PTRA and PTRB.

PTRA is mostly used for recognising statement keywords, and a few other special uses; it should not be used inside the expression evaluator (i.e. in functions) unless it is saved, and restored before returning. The base of PTRA is stored in &B and &C, with the offset in &A.

PTRB is used for evaluating expressions, and most other general uses. The base ofPTRB is stored in &19 and &1A, with the offset in &1B.

The base of both of these pointers normally points 1 character before the start of the text of the statement currently being executed (i.e. the ':'; or the length byte of the line if it is the first statement on the line). These should not normally be changed during a statement, except at the end, when they will be set up to point to the next one by the 'Check end of statement' routine.

getcha − Get character at PTRA into A

Execution addr

BASIC1	&8A1E
BASIC2	&8A97

Entry conditions:

PTRA: points to the character to be read.

Exit conditions:

PTRA: points to the next character to be read.

A	character read
X	preserved
Y	offset from base of PTRA to character just read
C	undefined

Description

This routine returns the first non-space character found at, or after, PTRA. The offset of PTRA is updated so that it points to the character after the one just read. The character returned by this routine can be re-read if necessary by a ‘LDA (&B), Y’.

Other entry points

NONE

getchb − Get character at PTRB into A

Execution addr

BASIC1	&8A13
BASIC2	&8A8C

Entry conditions:

PTRB: points to the character to be read

Exit conditions:

PTRB: points to the next character to be read.

A	character read
X	preserved
Y	offset from base of PTRA to character just read
C	undefined

Description

This routine returns the first non-space character found at, or after, PTRB. The offset of PTRB is updated so that it points to the character after the one just read. The character returned by this routine can be re-read if necessary by a ‘LDA (&19), Y’.

Other entry points

NONE

chksda − Check for end of statement

Execution address

BASIC1 &9810
BASIC2 &9857

Entry conditions:

PTRA: points at the end of the current statement.

Exit conditions:

PTRA: base points to the statement delimiting character. - offset = 1

A undefined
X preserved
Y 1
C undefined

Description

Starting at PTRA, if the first non-space character found is not a ‘:’ ,a carriage return character, or an ‘ELSE’ token, then a ‘Syntax error’ (ERR = 16) will be generated. If it is one of these, then the base of PTRA will be updated to point to this character, and the offset set to 1. Thus PTRA will point to the first character after the statement delimiter. Finally, the escape flag is tested before returning, and an ‘Escape’ error (ERR = 17) will be generated if an escape condition exists.

Other entry points

1 chksdb − Check end of statement at PTRB

BASIC1 &980B
BASIC2 &9852

This uses the offset of PTRB instead of the offset of PTRA on entry. Providing that the base of PTRA has been copied into PTRB at some time during the statement, this entry point can be used to check for the end of the statement at PTRB.

cont − Continue execution

Execution addr

BASIC1 &8B0C
BASIC2 &8B9B

Entry conditions:

PTRA: base points to the statement delimiting character.
offset = 1

Exit conditions:

NON-RETURNING

Description

This entry will test the statement delimiter at the base of PTRA. If it is an 'ELSE' token, the rest of the line will be skipped, and execution will continue on the next program line. Otherwise, execution will continue with the next statement or program line, giving a TRACE if necessary. If the end of the program has been reached (or the end of the line in immediate mode), a jump will be made to enter immediate mode.

Other entry points

1 contsd − Check end of statement, then continue

BASIC1 &8B09
BASIC2 &8B98

This calls 'check for end of statement' before dropping into the main routine. Entry conditions are as for 'check end of statement'.

skplin − Skip rest of line, then continue execution

Execution addr

BASIC1 &8AED
BASIC2 &8B7D

Entry conditions:

PTRA: points at or before the CR on the end of the line.

Exit conditions:

NON-RETURNING

Description

This entry will skip the rest of the current line, and execution will continue on the next program line, giving a TRACE if necessary. If the end of the program has been reached, or the line was an immediate mode command, a jump will be made to enter immediate mode.

Other entry points

NONE

10.4 EXPRESSION EVALUATION

Expression evaluation is carried out using PTRB to scan the text. At each stage, the result is left in IntA, FPA, or StrA for the code which called the routine. If the type of the result is not what is required by the particular level (for example, an attempt to AND with a string), then a 'Type mismatch' error is generated. See chapter 4 for more on expression evaluation.

getnsb − Get <numeric> or <string> at PTRB

Execution addr

BASIC1 &9B03
BASIC2 &9B29

Entry conditions:

PTRB: points to the next character to be read.

Exit conditions:

PTRB: points to the next character to be read.

If Z=1: result in StrA (string)
If N =1: result in FPA (real)
Otherwise: result in IntA (integer)

&27 result type (&00=string, &40=integer, &FF=real)

&2A−&4E undefined (except where specified above)

A result type
X next character (after the <numeric> or <string>)
Y result type
C undefined

Description

This routine evaluates the <numeric> or <string> at PTRB (leading spaces will be ignored), and sets the 6502 flags according to the type of the result (see chapter 4 for more on expressions). PTRB will be updated to point to the character after the <numeric> or <string>. Nothing should be left in the accumulators (&2A to &36), or in BASIC's temporary workspace (&37 to &4E), as this will be used by the routine. Any temporary results which need to be kept should be saved on the BASIC STACK, or in the 'free for users' zero page area (&70 to &8F). Note also, that because FN's can apppear in a <numeric> or <string>, anything that can be set by a BASIC statement is liable to change. PTRA will be preserved by this routine (it is saved during execution of FNs and PROCs).

Other entry points

1 getnsa − Get <numeric> or <string> at PTRA

```
BASIC1   &9AF7
BASIC2   &9B1D
```

This entry copies PTRA into PTRB before entering the main routine. All other entry and exit conditions are the same.

getfsb − Get <factor> or <string-factor> at PTRB

Execution addr

BASIC1 &AE1B
BASIC2 &ADEC

Entry conditions:

PTRB: points to the next character to be read.

Exit conditions:

PTRB:	points to the next character to be read.
If Z= 1:	result in StrA (string)
If N=1:	result in FPA (real)
Otherwise:	result in IntA (integer)
&27	undefined
&2A−&4E	undefined (except where specified above)
A	result type (&00=string, &40=integer, &FF=real)
X	undefined
Y	undefined
C	undefined

Description

This routine evaluates the <factor> or <string-factor> at PTRB (leading spaces will be ignored), and sets the 6502 flags according to the type of the result (see chapter 4 for more on expressions). PTRB will be updated to point to the first character after the <factor> or <string-factor>. Nothing should be left in the accumulators (&2A to &36), or in BASIC's temporary workspace (&37 to &4E), as this will be used by the routine. Any temporary results which need to be kept should be saved on the BASIC STACK, or in the 'free for users' zero page area (&70 to &8F). Note that FN's can be called inside this routine, so anything that can be set by a BASIC statement is liable to change.

Other entry points

1 getifb − Get integer <factor> at PTRB

```
BASIC1   &92E3
BASIC2   &9292
```

This entry calls the main routine, and then forces the result to be an integer. If the result is a string, a 'Type mismatch' error (ERR = 6) will be generated; if the result is real, it will be converted to an integer. Entry and exit conditions are as for the main routine, except that A and the flags will always indicate an integer result.

2 getrfb − Get real <factor> at PTRB

```
BASIC1   &92AC
BASIC2   &92EB
```

This entry calls the main routine, and then forces the result to be real. If the result is a string, a 'Type mismatch' error (ERR = 6) will be generated; if the result is an integer, it will be converted to a real number. Entry and exit conditions are as for the main routine, except that A and the flags will always indicate a real result.

getnmb − Get number at PTRB

Execution addr

BASIC1 &A06C
BASIC2 &A07B

Entry conditions:

PTRB:	points 1 after the first digit of the number
A	first digit of the number
Y	offset from base of PTRB to first digit of number

Exit conditions:

PTRB:	points to the next character to be read.
C	0=no number found, 1=number found
If N=1:	result in FPA (real)
Otherwise:	result in IntA (integer)
&2A−&35	undefined (except where specified above)
&43	undefined
&48−&4A	undefined
A	result type (&40=integer, &FF=real)
X	undefined
Y	undefined

Description

This routine gets the positive decimal integer at PTRB whose first digit has just been read using the 'Get character at PTRB' routine. If no number was found (i.e. the character in A on entry was not one of '0' to '9'), it will clear C and leave zero in FPA as a real result. If a number was found, it will be left in IntA or FPA, depending on the type ('200000' will be integer, '2E5' or '1.7' will be real).

Other entry points

NONE

getlna − Get a tokenised line number at PTRA

Execution addr

BASIC1 &97AE
BASIC2 &97DF

Entry conditions:

PTRA: points to the next character to be read.

Exit conditions:

If C=0 (no line number found):

PTRA: points to first non-space character found.

A character at PTRA
X preserved
Y PTRA offset

If C= 1 (line number found):

PTRA: points to the next character to be read.

IntA: line number (in &2A,&2B)

A undefined
X preserved
Y PTRA offset

Description

This routine checks for a line number token (&8D) at PTRA (ignoring leading spaces). If it finds one, it gets the 3 bytes of tokenised line number following it into the low-order 2 bytes of IntA, and exits with C set. Otherwise, it exits with C clear. See section 2.3.2 for the format of tokenised line numbers.

Other entry points

NONE

10.5 VARIABLE/FN/PROC MANAGEMENT

Named variables, and the location of FNs and PROCs are stored on the BASIC HEAP, which builds upwards from LOMEM. The HEAP pointer is stored at &2,&3 in page zero, and points to the next available memory location for a variable or FN/PROC information block to be stored in. See section 3. 1 for more on HEAP storage.

Each named variable stored on the HEAP has its own variable information block, which gives the name and value of the variable. These are chained together to form a linked list: one list for each possible first letter (A to z), and one each for FNs and PROCs. The format of the *variable information block* is:

00,01	pointer to start of next block
02−	name of variable
XX	&00 name terminator
XX+1	value starts here

The 'name' field does not include the first letter of the name if it is a variable (but it does if it is a FN or PROC). The name includes any '%', '$', or '(' characters on the end of a variable name: these give the type of the variable.

Much of the variable handling is done using a *variable descriptor block*, which gives the location and type of the variable. This *variable descriptor block* has the following format (when in IntA):

(&2A)	points to the start of the variable value
&2C	holds the type of the variable

Variable types can be:

&00	single byte integer
&04	4-byte integer
&05	5-byte real number
&80	static string terminated by a &0D
&81	dynamic string (stored on the HEAP)

For the format of these variable types, see section 3.1.3.

fndvrb − Find variable at PTRB

Execution addr

BASIC1 &95A9
BASIC2 &95DD

Entry conditions:

PTRB: points to the first character of the variable name.

A first character of the variable name
Y copy of PTRB offset (in &1B)

Exit conditions:

Z=0,C=0: numeric variable found
Z=0,C=1: string variable found
Z=1,C=0: non-existent (but valid) variable name found
Z=1,C=1: no valid variable was found

A undefined
X undefined
Y undefined

If Z=0: (variable exists)

PTRB: points to the character after the variable
IntA: variable descriptor block

&2E−4E undefined

If Z=1,C=0: (non-existent variable)

PTRB: points to the character after the name
&2C variable type
(&37) points 1 before the start of the name
&39 length of name

&3A−&3D undefined

If Z=1,C= 1: (invalid variable)

(&37) points 1 before PTRB

Description

This routine looks for the variable which is at PTRB (this includes indirected variables like ?A or B!5). If the variable exists, it sets up the variable descriptor block in IntA. If it does not exist, but is a valid name, it sets up the pointer at &37 ,&38 with the length of the name in &39, ready to create it if necessary. If a non-existent array name is found, an 'Array' error (ERR = 14) will be generated.

Other entry points

1 fndvra − Find variable at PTRA

BASIC1 &9595
BASIC2 &95C9

This entry first copies PTRA into PTRB, and then skips any leading spaces at PTRB, before entering the main routine. The exit conditions are the same.

2 fncvra − Find variable at PTRA, creating one if necessary

BASIC1 &9548
BASIC2 &9582

This entry calls entry point 1 above, and if a non-existent, but valid, variable name is found, it will create it and clear space for it on the HEAP. Its initial value will be zero (or the empty string). Exit conditions are the same as for the main routine (the variable may still be invalid).

rdvar − Read value of variable

Execution addr

BASIC1 &B35B
BASIC2 &B32C

Entry conditions:

IntA: variable descriptor block

Exit conditions:

If Z=1: result in StrA (string)
If N=1: result in FPA (real)
Otherwise: result in IntA (integer)

A: result type (&00=string, &40=integer, &FF=real)
X: undefined
Y: undefined
C: undefined

Description

This routine gets the value of the variable given by the variable descriptor block in IntA, and transfers it to the relevant accumulator. This can also be used to get the value of parameters passed by the BASIC ‘CALL’ statement.

Other entry points

NONE

asvar − Assign string variable

Execution addr

BASIC1&8BD3
BASIC2&8C21

Entry conditions:

IntA: variable descriptor block (MUST be a string)
StrA: value to be assigned

Exit conditions:

Value assigned to variable

HEAP: moved up if necessary

Description

This routine assigns the value in StrA to a static or dynamic string. In the case of a dynamic string, if the space allocated for the string is not large enough, a new space is allocated on the HEAP (see section 3.1.3 for more on string allocation). A static string (one which is to be written into memory using the string indirection operator) will just be stored at the address given, terminated by a carriage return character (&0D). This routine can be used to set the value of string parameters passed by the BASIC ‘CALL’ statement. Both the variable and the value must be a string, as no test is made by this routine for type mismatch.

Other entry points

1 asvark − Assign variable on stack

BASIC1 &8BD0
BASIC2 &8C1E

This entry pulls the variable descriptor block from the STACK into IntA before entering the main routine. It should have previously been pushed on the STACK using the ‘Push IntA’ routine (pushi).

anvark − Assign numeric variable

Execution addr

BASIC1 &B4E0
BASIC2 &B4B4

Entry conditions:

STACK: variable descriptor block

&27: type of value (&00=string, &40=integer, &FF=real)

Real: value in FPA
Integer: value in IntA

Exit conditions:

STACK: variable descriptor block removed (4 bytes)

Value assigned to variable

&37 − &3A undefined

A undefined
X undefined
Y undefined
C undefined

Description

This routine assigns the value in FPA or IntA (type given in &27) to the variable whose variable descriptor block is on the STACK. This should have previously been pushed by the 'Push IntA' routine (pushi). This routine can be used to set the value of numeric parameters passed by the BASIC 'CALL' statement. If the type of the value (in &27) is a string, a 'Type mismatch' error (ERR = 6) will be generated, but the variable type is not checked, and must be numeric.

Other entry points

1 asgtvr − Assign <numeric> to variable on stack

BASIC1&B4DD
BASIC2&B4B1

This entry calls the 'Get <numeric> or <string> at PTRB' routine (getnsb), to set up the value and the type in &27, before entering the main routine. The variable descriptor block should still be on the STACK on entry. All temporary areas (&2A to &4E) will be undefined if this entry is used.

schvar − Search for variable in list

Execution addr

BASIC1 &9429
BASIC2 &9469

Entry conditions:

(&37) points 1 before the start of the variable name
&39 length of name

Exit conditions:

If Z=1: variable not found
If Z=0: variable found

&3A−&3D undefined

A undefined
X preserved
Y undefined
C undefined

If Z=0 (variable found):

(&2A) points to the variable value

Description

This routine searches for a variable name in the linked list. If found, it sets the low 2 bytes of the variable descriptor block in IntA to the address of the value of the variable. This routine is used by the main ‘Find variable at PTRB’ routine (fndvar).

Other entry points

NONE

lnkvar − Link in new variable

Execution addr

BASIC1 &94BC
BASIC2 &94FC

Entry conditions:

(&37) points 1 before the start of the name
&39 length of name

Exit conditions:

New variable information block linked in to HEAP.

(&3A) points to the previous block
HEAP points to the new block

A undefined
X undefined
Y length of name
C undefined

Description

This routine links in a new variable infomation block to the linked list of variables on the HEAP (see section 3.1 for more on the HEAP). The MSB of the new link pointer is zeroed (to mark the end), and the name is transferred to the new block. The routine exits with the pointer at &3A,3B pointing to the previous link pointer (which now points to the new block), so that this pointer can be re-set if there is not enough memory for the new block. This routine does not allocate any memory for the new block; this must be done with a call to the ‘Clear space for information block’ routine (clrib).

Other entry points

NONE

scnvn − Scan variable name

Execution addr

BASIC1 &951F
BASIC2 &9559

Entry conditions:

(&37) points 1 before the start of the name

X (see exit)

Exit conditions:

A first character following variable name
X incremented by the length of the name
Y offset from (&37) of character in A
C undefined

Description

This routine scans the variable name starting one byte after the pointer at (&37). Only the characters A-Z, a-z, @, _, and £ are allowed in variable names (and 0-9 after the first character). The special variable symbols '$' and '%' are not recognised by this routine. This routine is used by the array handler and the FN/PROC handler.

Other entry points

NONE

schfnp − Look for FN/PROC in list

Execution addr

BASIC1 &941B
BASIC2 &945B

Entry conditions:

(&37) points 1 before the FN/PROC token
&39 length of name (including 1 for FN/PROC token)

Exit conditions:

If Z=1: FN/PROC not found in list
If Z=0: FN/PROC found

&3A−&3D undefined

A undefined
X preserved
Y undefined
C undefined

If Z=0 (FN/PROC found):

(&2A) points to the FN/PROC pointer field

Description

This routine searches for a given FN or PROC in the linked list on the HEAP. If found, it leaves the low 2 bytes of IntA pointing to the pointer field of the FN/PROC information block. This pointer field points to the first character after the FN or PROC name definition (i.e. the '(' if it has any parameters). See section 3.1 for HEAP storage.

Other entry points

NONE

lnkfnp − Link in new FN/PROC

Execution addr

BASIC1 &94AD
BASIC2 &94ED

Entry conditions:

(&37)	points 1 before the FN/PROC token
&39	length of name (including FN/PROC token)

Exit conditions:

New FN/PROC information block linked in to the HEAP.

(&3A)	points to the previous block
HEAP	points to the new block

A	undefined
X	undefined
Y	length of name
C	undefined

Description

This routine links in a new FN or PROC infornation block to the linked list of FNs or PROCs on the HEAP (see section 3.1 for more on the HEAP). The MSB of the new link pointer is zeroed (to mark the end), and the name is transferred to the new block. The routine exits with the pointer at &3A,3B pointing to the previous link pointer (which now points to the new block), so that this pointer can be re-set if there is not enough memory for the new block. This routine does not allocate any memory for the new block; this must be done with a call to the ‘Clear space for information block’ routine (clrib).

Other entry points

NONE

clrib − Clear space for new information block

Execution addr

BASIC1	&94F7
BASIC2	&9531

Entry conditions:

X	number of bytes to be cleared (at least 1)
Y	offset of end of name into information block
HEAP	points to start of information block
(&3A)	points to the previous block in the list

Exit conditions:

Bytes cleared in information block given by X on entry

HEAP:	moved up to cover new block
A	LSB of HEAP pointer
X	0
Y	MSB of HEAP pointer
C	0

Description

This routine clears and allocates space on the HEAP for a variable or FN/PROC information block, once the pointer and name have been set up. On entry, Y (as an offset from the HEAP pointer) points to the last character of the name already in the information block, and X contains the number of bytes which need to be zeroed after it (including 1 for the name terminating byte). If the HEAP pointer is above the STACK pointer after the space for the block is allocated, then a 'No room' error is generated (message only in BASIC1, ERR = 0 in BASIC2). Because the bytes are cleared before the space check is made, the top of STACK contents will be destroyed if there is not enough room. This routine is called after the 'Link in new variable' (Inkvar) or 'Link in new FN/PROC' (lnkfnp) routines have set up the name and link pointer.

Other entry points

1 mvheap − Add Y to HEAP pointer

BASIC1 &94FF
BASIC2 &9539

This entry point adds Y to the HEAP pointer. It does not zero any bytes. If the new HEAP pointer is above the STACK pointer, a 'No room' error is generated, otherwise the routine returns.

10.6 ST ACK MANAGEMENT

The BASIC STACK pointer is maintained in page zero in &04,&05 and works downwards from HIMEM. It is used to hold temporary results, and information saved by FNs and PROCs. For more on the use of the STACK, see section 3.2.

pusha − Push IntA, FPA, or StrA on STACK

Execution addr

BASIC1 &BDA8
BASIC2 &BD90

Entry conditions:

If Z=1: string in StrA
IfN=1: real in FPA
Otherwise: integer in IntA

Exit conditions:

Item pushed on ST ACK

STACK: pointer lowered by size of item

A undefined
X preserved
Y undefined
C undefined

Description

This routine tests the 6502 flags on entry to find the type of the item to be pushed on the BASIC STACK. It then pushes the appropriate accumulator (IntA, FPA, or StrA). Note that there is no way to tell the type of an item on the STACK, so this should be saved before this routine is called. If the ST ACK would be lowered below the level of the HEAP by pushing this item, a ‘No room’ error is generated (message only in BASIC1, ERR = 0 in BASIC2), and the item is not pushed.

Other entry points

1 pushi − Push IntA on STACK

BASIC1 &BDAC
BASIC2 &BD94

This routine pushes IntA on the BASIC ST ACK, lowering the STACK pointer by 4 bytes. This can be used to save the variable descriptor block, which is sometimes held in IntA.

2 pushf − Push FPA on STACK

BASIC1 &BD69
BASIC2 &BDB2

This entry pushes FPA on the BASIC STACK, lowering the STACK pointer by 5 bytes.

3 pushs − Push StrA on STACK

BASIC1 &BDCA
BASIC2 &BDB2

This routine pushes StrA on the BASIC STACK, lowering the STACK pointer by one more than the length of the string (the byte on the top gives the length of the string).

chksp − Check for STACK/HEAP clash

Execution addr

BASIC1 &BE4C
BASIC2 &BE34

Entry conditions:

STACK: new value of STACK pointer to be tested

A copy of LSB of new STACK pointer, &4

Exit conditions:

A preserved (LSB of STACK pointer)
X preserved
Y MSB of STACK pointer
C 1

Description

This routine tests the STACK pointer against the HEAP pointer. If the STACK is below the HEAP, a 'No room' error is generated (message only in BASIC1, ERR = 0 in BASIC2). If there is no clash, the routine returns.

Other entry points

1 lwrsp − Lower STACK pointer; check for HEAP clash

BASIC1 &BE46
BASIC2 &BE2E

This entry point can be used if up to 255 bytes need to be allocated on the STACK. The LSB of the STACK pointer (in &4) should be loaded into A, and the number of bytes required should be subtracted from this. A call to this entry point will then save A as the LSB of the new STACK pointer, and decrement the MSB (in &5) if the subtraction had cleared the carry flag (i.e. if the number of bytes to be allocated was greater than the LSB of the STACK pointer). The main routine will then be entered to test for a HEAP clash.

popi − Pop IntA from STACK

Execution addr

BASIC1 &BE02
BASIC2 &BDEA

Entry conditions:

STACK: points to the 4-byte integer to be popped

Exit conditions:

IntA: integer popped from STACK

STACK: pointer moved up by 4 bytes

A undefined
X preserved
Y 0
C undefined

Description

This routine pops the 4-byte integer from the top of the ST ACK into IntA, and moves the STACK pointer up by 4 bytes to remove it.

Other entry points

1 rmvi − Remove integer from STACK

BASIC1 &BE17
BASIC2 &BDFF

This entry moves the STACK pointer up by 4 bytes to remove the integer on the STACK. X and Y are preserved.

popi0 − Pop integer from STACK into page zero

Execution addr

BASIC1 &BE25
BASIC2 &BE0D

Entry conditions:

STACK: points to the 4-byte integer to be popped

X points to the destination for the integer

Exit conditions:

00,X to 03,X holds the integer just popped

STACK: pointer moved up by 4 bytes

A undefined
X preserved
Y 0
C undefined

Description

This routine pops the 4-bytes on the top of the STACK into page zero at 00,X to 03,X. It then moves the STACK pointer up by 4 bytes to remove it.

Other entry points

1 popi1 − Pop integer from stack into &37 to &3A

BASIC1 &BE23
BASIC2 &BE0B

This entry sets X to &37 before entering the main routine.

popf − Pop real number from STACK; set up (&4B)

Execution addr

BASIC1 &BD96
BASIC2 &BD7E

Entry conditions:

STACK: points to the 5-byte real number to be popped

Exit conditions:

(&4B) points at real number

STACK: pointer moved up by 5 bytes

A undefined
X preserved
Y preserved
C undefined

Description

This routine pops a real number from the STACK, and moves up the STACK pointer by 5 bytes to remove it. It does not move the number into FPA, but it sets up the floating point memory pointer, (&4B), to point to it. If the number is to be saved, it should be loaded into FPA or FPB after this routine has been called.

Other entry points

NONE

pops − Pop StrA from STACK

Execution addr

BASIC1 &BDE3
BASIC2 &BDCB

Entry conditions:

STACK: points to the string to be popped

Exit conditions:

StrA: string popped from STACK

STACK: pointer moved up to remove string

A undefined
X preserved
Y 0
C undefined

Description

This routine pops a string from the STACK into StrA, and moves the STACK pointer up by one more than the length of the string, to remove it from the stack (the length of the string is the first byte on the stack).

Other entry points

1 rmvs − Remove string from STACK

BASIC1 &BDF4
BASIC2 &BDDC

This entry gets the length of the string from the stack, and moves the STACK pointer up by one more than the length of the string (to allow for the length byte, which was also on the stack).

pshvvd − Push value and descriptor of variable on STACK

Execution addr

BASIC1 &B33C
BASIC2 &B30D

Entry conditions:

IntA: variable descriptor block

Exit conditions:

Value of variable pushed on STACK, followed by descriptor

STACK: lowered by required amount
A undefined
X undefined
Y undefined
C undefined

Description

This routine gets the value of the variable pointed to by the variable descriptor block in IntA, and pushes it on the STACK. It then pushes the variable descriptor block, so the variable can be re-set later. This is used to save the old values of local variables (or parameters) for a FN or a PROC.

Other entry points

NONE

poppar − Pop old parameter value from STACK

Execution addr

BASIC1 &8C5B
BASIC2 &8CC1

Entry conditions:

&37−&39 variable descriptor block

STACK: points to the value to be popped

Exit conditions:

Value assigned to variable

STACK: pointer moved up to remove value

A undefined
X undefined
Y undefined
C undefined

Description

This routine is used to re-assign old values to parameters and local variable which have previously been saved on the STACK. It should NOT be used to assign new variables, because it assumes the allocated space for a string will be large enough (which it will be, if it came from there in the first place). It is used on a return from a procedure or function, to re-set old variable values.

Other entry points

NONE

10.7 INPUT/OUTPUT

These routines are the input and output routines used in BASIC. The output routines all handle COUNT (in &IE) and WIDTH (in &23): COUNT is used by BASIC to keep track of the current cursor column to be used by TAB.

There is no routine to print a number from IntA or FPA: to do this the number can be converted to a string in StrA using the 'Type conversion' routines (section 10.8), and then StrA can be printed (there is not a routine for this either, but it is fairly simple). Input of numbers can also be accomplished by inputting a string, and then converting that to a number.

inputs − Input string from keyboard into StrA

Execution addr

BASIC1 &BC17
BASIC2 &BBFC

Entry conditions:

NONE

Exit conditions:

&600−	string input
&37−&3B	used as the OSWORD parameter block
COUNT	set to zero (in &1E)
A	0
X	undefined
Y	length of string
C	0

Description

This routine calls OSWORD with A=&0 to input a line from the keyboard into StrA at &600 onwards. Maximum line length is 238 bytes; all characters with an ASCII value of less than &20 will not be put in the input line (i.e. the control characters). If the ESCAPE key terminated the input instead of a carriage return, an ‘Escape’ error (ERR = 17) will be generated.

Other entry points

1 inputk − Input string into the keyboard buffer

BASIC1	&BC1D
BASIC2	&BC02

This entry prints the character in A as a prompt, and sets the address for input to be &700 (the keyboard buffer) before joining the main routine. It is used for BASIC’s immediate mode

pchar − Print A as a character

Execution addr

BASIC1 &B571
BASIC2 &B558

Entry conditions:

A character to be printed

Exit conditions:

COUNT updated, allowing for WIDTH if necessary

A preserved
X preserved
Y preserved
C undefined

Description

This routine outputs the character in A using OSWRCH, and increments the value of COUNT. If COUNT has moved past WIDTH, the character will be printed on a new line, and COUNT will be reset.

Other entry points

1 pspace − Print a space

BASIC1 &B57B
BASIC2 &B565

This entry loads A with a space (&20) before entering the main routine.

2 pnewl − Print a newline

BASIC1 &BC42
BASIC2 &BC25

This entry point calls OSNEWL to print a carriage return and a line feed, and then zeros COUNT.

ptoken − Print A as a character or token

Execution addr

BASIC1 &B53A
BASIC2 &B50E

Entry conditions:

A	character or token to be printed

Exit conditions:

COUNT	updated, allowing for WIDTH if necessary
&37−&3A	undefined
A	last character printed
X	preserved
Y	preserved
C	undefined

Description

If the character in A is less than &80, it will be printed out as a character. Otherwise, it will be interpreted as a token, and the corresponding keyword will be printed from the token table. This routine will not handle a line number token, or any other invalid token (which may cause the routine to hang up). This routine is used by the ‘LIST’ and ‘REPORT’ statements.

Other entry points

NONE

phex − Print A as a 2-digit HEX number

Execution addr

BASIC1 &8570
BASIC2 &B545

Entry conditions:

A byte to be printed

Exit conditions:

COUNT updated, allowing for WIDTH if necessary

A last character printed
X preserved
Y preserved
C undefined

Description

This routine prints the byte in A as a 2-digit HEX number (a leading zero will not be suppressed). This routine is used by the assembler, but has been re-located in BASIC2 to save space.

Other entry points

1 phexsp − Print HEX byte, followed by a space

BASIC1 &856A
BASIC2 &B562

This entry calls the main routine to print the 2-digit HEX number in A, and then prints a space after it. This leaves &20 in A on exit.

plnum0 − Print line number

Execution addr

BASIC1 &98F1
BASIC2 &991F

Entry conditions:

IntA: line number to be printed

Exit conditions:

COUNT	updated, allowing for WIDTH if necessary
&14	0 (field width used)
&37	undefined
&3F−&43	undefined
A	last character printed
X	&FF
Y	undefined
C	undefined

Description

This routine prints the line number in the low 2 bytes of IntA as a positive decimal number between 0 and 65535. No leading spaces are printed.

Other entry points

1 pInum5 − Print line number (field 5)

BASIC1 &98F5
BASIC2 &9923

This entry uses a field width of 5 to print the line number: it will be padded with leading spaces if necessary. Location &14 will be set to 5 on exit.

10.8 TYPE CONVERSION

These routines allow conversion between integers, reals, and strings.

The ‘Integer to real’ and ‘Real to integer’ routines are used throughout the expression evaluator in BASIC when the type of the number being dealt with needs to be converted. For example if an integer is being added to a real number, the integer must be converted to real before the addition is carried out.

The ‘String to number’ and ‘Number to string’ routines are used during input and output of numbers, as the I/O routines do not handle numbers directly.

citof − Convert integer to real number

Execution addr

BASIC1 &A2AF
BASIC2 &A2BE

Entry conditions:

IntA: integer to be converted

Exit conditions:

FPA: converted real number (normalised)
IntA: ABS value of original integer

A undefined
X undefined
Y undefined
C undefined

Description

This routine converts the 2's complement (signed) integer in IntA to a real number in FPA.

catof − Convert A to real number

Execution addr

BASIC1 &A2DE
BASIC2 &A2ED

Entry conditions:

A	2's complement signed integer (+127 to −128)

Exit conditions:

FPA:	converted real number (normalised)
A	0 if number is zero, else undefined (non-zero)
X	undefined
Y	undefined
C	undefined
Z	1 if number is zero, else 0

Description

This routine converts the 2's complement (signed) integer in IntA to a real number in FPA.

Other entry points

NONE

cftoi − Convert real number to integer

Execution addr

BASIC1 &A3F2
BASIC2 &A3E4

Entry conditions:

FPA: real number to be converted

Exit conditions:

IntA: converted integer

FPA: 2's complement integer part of number in mantissa
FPB: ABS value of fractional part of number in mantissa

A undefined
X undefined
Y undefined
C undefined

Description

This routine converts the floating point number in FPA into an integer in IntA. If the number is too large to be converted to an integer, a 'Too big' error (ERR = 20) will be generated. On conversion, the ABS value of the number will be truncated, and then negated if necessary; this means that '−1.9' will be converted to '−1' (try 'A% = −1.9'). On exit, FPB mantissa contains the ABS value of the fractional part of the number (the top bit of &3E represents 0.5), and the sign of this fraction will be in &2E, so this could be used to round the number properly afterwards, if necessary.

Other entry points

1 int − Take INT ofFPA

BASIC1 &ACA5
BASIC2 &AC7F

This entry performs the equivalent of the BASIC function 'INT': it converts the floating point number to the highest integer which is less than or equal to it (i.e. '−1.9' gets converted to '−2', '1.9' gets converted to '1'). This routine will exit with &40 in A, and the Z and N flags clear, to signal an integer result (as if from the 'Get <factor> or <string-factor>' routine). To round a number to the nearest integer, 0.5 could be added to it before this routine is called.

cntos − Convert number to string

Execution addr

BASIC1 &9ED0
BASIC2 &9EDF

Entry conditions:

Y	type of number (&40=integer, &FF=real)
IfY=&40:	integer in IntA
If Y=&FF:	real in FPA
@%	set as for the BASIC 'PRINT' statement
&15	top bit set if number is to be in HEX

Exit conditions:

StrA:	converted string
IntA:	undefined
FPA:	undefined
FPB:	undefined
&37,&38	undefined
&3B−&46	undefined
&49	undefined
&46C−&470	undefined
A	undefined
X	undefined
Y	undefined
C	undefined

Description

This routine converts the number in either IntA or FPA to a string in StrA. If entered with bit 7 of &15 set, then a HEX number will be produced; otherwise a decimal number will be produced. The format of this number depends on the value of @%o (refer to 'PRINT' in the User Guide). This routine uses most of the page zero temporary area, so any temporary results should be saved out of the way before this routine is called.

Other entry points

1 cntoh − Convert number to HEX string

BASIC1 &9E81
BASIC2 &9E90

This is the routine called if the hex flag (bit 7 of &15) is set on entry to the main routine. This will convert the number to a hex string, ignoring the settings of @%o and &15. Y must still contain the type of the number (if it is real it will be converted to integer before the HEX string is generated). Any leading zeros will be suppress.ed. This entry only uses locations &3F to &46 for the conversion.

cston − Convert string to number

Execution addr

BASIC1 &AC5A
BASIC2 &AC34

Entry conditions:

StrA: string to be converted

Exit conditions:

N 1=real, 0=integer

If N=1: result in FPA (real)
If N=0: result in IntA (integer)

&27 number type (&40=integer, &FF=reaI)

&2A− &35 undefined (except where specified above)
&43 undefined
&48−&4A undefined

A number type
X undefined
Y undefined
C undefined

Description

This routine converts the ASCII decimal number in StrA into either a real number in FPA or an integer in IntA. It uses the ‘Get number at PTRB’ routine (getnmb), pointing PTRB into StrA, and restores PTRB to its original value afterwards. It leaves the 6502 flags indicating the type of the result (either integer or real).

Other entry points

NONE

10.9 INTEGER ROUTINES

Most of the integer arithmetic is performed using the 4-byte integer accumulator, IntA, which is held in page zero at &2A to &2D (LSB in &2A, MSB in &2D). The multiplication and division routines also use two other 4-byte accumulators in the temporary storage area, at &39 to &3C and at &3D to &40.

IntA can be transfered to and from memory by using the variable handling routines in section 10.5, with the variable descriptor block set up as if to point to an integer variable. It can be set to 0 or −1 by using the 'FALSE' and 'TRUE' entry points (section 10.11.

lodiay − Load IntA with A, Y

Execution addr

BASIC1&AF19
BASIC2&AEEA

Entry conditions:

A	LSB of 16-bit positive integer
Y	MSB of 16-bit positive integer

Exit conditions:

IntA: 16-bit positive integer from A, Y

Z=0, N=0 to signal an integer result

A	&40 (result type =integer)
X	preserved
Y	preserved
C	preserved

Description

This routine sets up IntA with the 16-bit positive integer in A and Y. The top 2 bytes of IntA are set to zero.

Other entry points

1 lodia − Load IntA with A

BASIC1 &AF07
BASIC2 &AED8

This entry sets Y to zero before entering the main routine; thus setting IntA to the 8-bit positive integer in A.
&40 (result type = integer)

lodi0 − Load IntA from 00,X to 03,X

Execution addr

BASIC1 &AF85
BASIC2 &AF56

Entry conditions:

X points to 4-byte integer in page zero

Exit conditions:

IntA: 4-byte integer loaded from 00,X to 03,X

Z=0, N=0 to signal an integer result

A preserved
X preserved
Y preserved
C preserved

Description

This routine loads IntA with the 4-byte integer in page zero pointed to by X.

Other entry points

NONE

stori0 − Store IntA at 00,X to 03,X

Execution addr

BASIC1 &BE5C
BASIC2 &BE44

Entry conditions:

X points to 4-byte area in page zero

IntA: number to be transferred

Exit conditions:

00,X to 03,X contains the 4-byte integer in IntA

A MSB of integer
X preserved
Y preserved
C preserved

Description

This routine copies the contents of IntA into a 4-byte area of page zero pointed to by X.

Other entry points

NONE

negi − Negate IntA

Execution addr

BASIC1 &ADB5
BASIC2 &AD93

Entry conditions:

IntA: 4-byte integer to be negated

Exit conditions:

IntA: negated 4-byte integer

Z=0, N =0 to signify an integer result

A &40 (result type = integer)
X preserved
Y 0
C 0

Description

This routine negates the 4-byte integer in IntA.

Other entry points

1 absi − Take ABS value of IntA

BASIC1 &AD94
BASIC2 &AD71

This entry takes the absolute value of IntA. If it is negative, it will be negated; otherwise it will be unaffected. Exit conditions are as for the main routine.

addi − Perform integer addition

Execution addr

BASIC1 &9C36
BASIC2 &9C5B

Entry conditions:

IntA: 4-byte signed integer
STACK: 4-byte signed integer to add to IntA

X anything except '+' or '−'

Exit conditions:

IntA: 4-byte signed integer result

integer popped from STACK

A &40 (type of result = integer)
X preserved
Y 3
C undefined

Description

This routine adds the 4-byte signed integer on the BASIC STACK to the 4-byte signed integer in IntA. No overflow check is made by this routine.

This routine is an integral part of the expression evaluator. The X register must be set to any character other than a '+', or a '−' before the routine is called, or it will attempt to read another part of the expression it expects to be at PTRB. X is its *one character look-ahead* (see section 4.2).

Other entry points

NONE

subi − Perform integer subtraction

Execution addr

BASIC1 &9C9D
BASIC2 &9CC2

Entry conditions:

STACK:	4-byte signed integer
IntA:	integer to subtract from number on STACK
X	anything except ‘+’ or ‘−’

Exit conditions:

IntA:	4-byte signed integer result

integer pqpped from STACK

A	&40 (type of result = integer)
X	preserved
Y	3
C	undefined

Description

This routine subtracts the 4-byte signed integer in IntA from the 4-byte signed integer on the BASIC STACK. No overflow checking is made by this routine.

This routine is an integral part of the expression evaluator. The X register must be set to any character other than a ‘+’, or a ‘−’ before the routine is called, or it will attempt to read another part of the expression it expects to be at PTRB. X is its *one character look-ahead* (see section 4.2).

Other entry points

NONE

muli − Perform integer multiplication

Execution addr

BASIC1 &9D4A
BASIC2 &9D6D

Entry conditions:

IntA: 4-byte signed integer multiplier

STACK: 4-byte signed integer multiplicand

&27 anything except '*', '/', &83 or &81

Exit conditions:

IntA: 4-byte signed integer result
&39−&3C undefined
&3D−&40 ABS value of result

multiplicand popped from STACK

A &40 (type of result = integer)
X copy of &27
Y undefined
C undefined

Description

This routine multiplies the 4-byte signed integer in IntA by the 4-byte signed integer on the BASIC stack. The number in IntA must be between −32768 and +32767, as only the low 2 bytes are used, once its ABS value has been found. The routine does no checking for overflow, so it is a good idea to check for this before calling the routine.

This routine is an integral part of the expression evaluator. Location &27 must be set to any character other than a '*', a '/', a 'MOD' token or a 'DIV' token before the routine is called, or it will attempt to read another part of the expression it expects to be at PTRB. Location &27 is its *one character look-ahead* (see section 4.2).

Other entry points

NONE

divi − Perform integer division

Execution addr

BASIC1	&99C0
BASIC2	&99E8

Entry conditions:

IntA:	4-byte positive integer divisor
&39−&3C	4-byte positive integer dividend
&3D−&40	zero

Exit conditions:

IntA: preserved

&39−&3C	4-byte positive integer quotient
&3D−&40	4-byte positive integer remainder

A	undefined
X	undefined
Y	0
C	undefined

Description

This routine divides the 4-byte integer in page zero at &39 to &3C by the 4-byte positive integer in IntA (&3D to &40 must be set to zero on entry), leaving the result in &39 to &3C, and the remainder in &3D to &40. If IntA is zero on entry to this routine, a 'Division by zero' error (ERR = 18) will be generated.

If a signed division is required, the signed numbers should be converted to positive integers (using the ‘Take ABS value of IntA’ routine above) before this routine is called. The sign of the result can be calculated as the EOR of the signs of the two original operands (which should be saved before their ABS value is used for the division), and the result of the division then negated if necessary.

Other entry points

NONE

10.10 FLOATING POINT ROUTINES

Most of the floating point arithmetic is done using the main floating point accumulator FPA, at &2E to &35, and the secondary floating point accumulator FPB, at &3B to &42 (in the page zero temporary storage area). The memory area used by FPB may be used for other purposes by routines which do not involve any floating point calculations. See section 2.2.2 for more on floating point number storage.

The format of the accumulators is:

FPA	FPB	
&2E	&3B	sign byte
&2F	&3C	exponent overflow byte
&30	&3D	binary exponent (offset &80)
&31	&3E	mantissa (MSB)
&32	&3F	mantissa
&33	&40	mantissa
&34	&41	mantissa (LSB)
&35	&42	mantissa low order rounding byte

FPA and FPB are transferred to and from memory using a pointer at &4B,&4C. Floating point numbers are packed into 5 bytes when stored out in memory.

movfab − Move FPA to FPB

Execution addr

BASIC1&A20F
BASIC2&A21E

Entry conditions:

FPA: number to be copied

Exit conditions:

FPA: preserved
FPB: copy of FPA

A	undefined
X	preserved
Y	preserved
C	preserved

Description

This routine copies the floating point number in FPA to FPB.

Other entry points

NONE

movfba − Move FPB to FPA

Execution addr

BASIC1	&A4E4
BASIC2	&A4DC

Entry conditions:

FPB:	number to be copied

Exit conditions:

FPB:	preserved
FPA:	copy of FPB

A	undefined
X	preserved
Y	preserved
C	preserved

Description

This routine copies the floating point number in FPB to FPA.

Other entry points

NONE

ldfan0 − Load FPA with zero

Execution addr

BASIC1 &A691
BASIC2 &A686

Entry conditions:

NONE

Exit conditions:

FPA: zero

A 0
X preserved
Y preserved
C preserved
Z 1

Description

This routine sets the floating point accumulator FPA to zero.

Other entry points

NONE

ldfan1 − Load FPA with 1.0

Execution addr

BASIC1 &A6A4
BASIC2 &A699

Entry conditions:

NONE

Exit conditions:

FPA:	1.0
A	&81
X	preserved
Y	&81
C	preserved
Z	0

Description

This routine sets the floating point accumulator FPA to 1.0.

Other entry points

NONE

ldfbn0 − Load FPB with zero

Execution addr

BASIC1&A463
BASIC2&A453

Entry conditions:

NONE

Exit conditions:

FPB:	zero

A	0
X	preserved
Y	preserved
C	preserved
Z	1

Description

This routine sets the floating point accumulator FPB to zero.

Other entry points

NONE

ldfam − Load FPA from (&4B)

Execution addr

BASIC1 &A3A6
BASIC2 &A3B5

Entry conditions:

(&4B) set to point to 5-byte packed real number

Exit conditions:

FPA:	real number unpacked from (&4B)
A	0 if FPA is zero, else undefined (non-zero)
X	preserved
Y	0
C	preserved
Z	set if FPA is zero, else clear

Description

This routine loads the floating point accumulator FPA from memory, unpacking it from its S-byte packed format. On entry, the pointer at &4B,&4C points at the number to be loaded.

Other entry points

1 ldfatl − Load FPA from &46C to &470

BASIC1 &A3A3
BASIC2 &A3B2

This entry pre-sets the memory pointer (&4B) to point to the real number temporary storage slot at &46C before entering the main routine.

ldfbm − Load FPB from (&4B)

Execution addr

BASIC1 &A33F
BASIC2 &A34E

Entry conditions:

(&4B) set to point to 5-byte packed real number

Exit conditions:

FPB:	real number unpacked from (&4B)
A	0 if FPA is zero, else undefined (non-zero)
X	preserved
Y	0
C	preserved
Z	set if FPA is zero, else clear

Description

This routine loads the floating point accumulator FPB from memory, unpacking it from its 5-byte packed format. On entry, the pointer at &4B ,&4C points at the number to be loaded.

Other entry points

NONE

stfam − Store FPA at (&4B)

Execution addr

BASIC1 &A37E
BASIC2 &A38D

Entry conditions:

FPA: real number to be stored

(&4B) points to 5-byte destination

Exit conditions:

Number stored at (&4B)

A undefined
X preserved
Y 4
C preserved

Description

This routine packs FPA into a 5-byte area of memory pointed to by the pointer at &4B,&4C. Note that the-mumber in FPA must be in normalised form (i.e. with the top bit of the MSB of the mantissa set) before this routine is called to store it in memory. FPA and (&4B) are preserved by this operation. There is no corresponding routine to store the contents of FPB into memory.

Other entry points

1 stfatx − Store FPA in floating point temp area

	Temp slot	BASIC1	BASIC2
stfat1	&46C to &470	&A376	&A385
stfat2	&471 to &475	&A36E	&A37D
stfat3	&476 to &47A	&A372	&A381

These entry points pre-set the memory pointer at (&4B) to point to a floating point temporary storage slot (&46C, &471, or &476) before entering the main routine. These slots can be used to hold temporary results in the middle of complex calculations, but they should not be used if the expression evaluator is called, as this may use these areas itself.

exfam − Exchange FPA with number at (&4B)

Execution addr

BASIC1	&A4DE
BASIC2	&A4D6

Entry conditions:

FPA:	real number
(&4B)	real number

Exit conditions:

FPA	real number from (&4B)
FPB	real number from (&4B)
(&4B)	real number from FPA

A	undefined
X	preserved
Y	4
C	preserved

Description

This routine exchanges the (normalised) number in FPA with the number pointed to by (&4B). It loads FPB from (&4B), stores FPA at (&4B), and then copies FPB into FPA.

Other entry points

NONE

pntmtx − Point (&4B) at temp storage slot

Execution addr

	Temp slot	BASIC1	BASIC2
pntmt1	&46C to &470	&A7FB	&AF75
pntmt2	&471 to &475	&A7F3	&A7ED
pntmt3	&476 to &47A	&A7F7	&A7F1
pntmt4	&47B to &47F	&A7EF	&A7E9

Entry conditions:

NONE

Exit conditions:

(&4B)	points to 5-byte temp store slot
A	4
X	preserved
Y	preserved
C	preserved

Description

These routines set the floating point memory pointer in &4B,&4C to point to a temporary storage slot.

Other entry points

NONE

tstfa − Test FPA

Execution addr

BASIC1	&A1CB
BASIC2	&AIDA

Entry conditions:

FPA:	number to be tested

Exit conditions:

If Z= 1,	FPA is zero
If Z=0, N=1	FPA is negative
If Z=0, N=0	FPA is positive

A	zero if Z=0, else undefined (non-zero)
X	preserved
Y	preserved
C	preserved

Description

This routine tests the floating point accumulator FPA, and sets the Z and N flags of the 6502 according to the number.

Other entry points

NONE

nmlfa − Normalise FPA

Execution addr

BASIC1 &A2F4
BASIC2 &A303

Entry conditions:

FPA: number to be normalised

Exit conditions:

FPA: normalised number

A 0 if FPA is zero, else undefined (non-zero)
X undefined
Y undefined
C undefined
Z set if number is zero, else clear

Description

This routine ensures that the number in FPA is in normalised form (i.e. it has the top bit of the MSB of the mantissa set). If it is not already normalised, it will shift up the mantissa of the number (correcting the exponent) until it is.

Other entry points

NONE

rcofa − Round FPA, and check overflow

Execution addr

BASIC1 &A667
BASIC2 &A65C

Entry conditions:

FPA: number to be rounded

Exit conditions:

FPA: number with mantissa rounded into 4 bytes

A 0
X undefined
Y undefined
C undefined
Z 1

Description

This routine tests the low-order rounding byte of FPA mantissa (held in &35), and rounds up the remaining 4 bytes of the mantissa if necessary. The low-order rounding byte is used for more accuracy in the middle of calculations, but must be rounded up into the rest of the mantissa before the number can be stored in memory in its packed format.

The routine then checks the exponent overflow byte (which is used to allow internal calculations to temporarily overflow the normal number limits). If this is zero, no overflow has occurred, and the routine exits; if it is negative, an underflow has occurred, and the number will be set to zero; and if it is positive (non-zero), an overflow has occurred, and a 'Too big' error (ERR = 20) will be generated. This routine (together with normalising) ensures that FPA is ready to be stored in memory in its packed 5-byte, format.

Other entry points

1 nrofa − Normalise, round and check overflow

BASIC1 &A664
BASIC2 &A659

This normalises FPA before entering the main routine above.

negfa − Negate FPA

Execution addr

BASIC1 &ADA0
BASIC2 &AD7E

Entry conditions:

FPA: number to be negated

Exit conditions:

FPA: negative of initial number

Z=0, N=1 to signal a real result

A &FF (to signal a real result)
X preserved
Y preserved
C preserved

Description

This routine negates the real number in FPA, and sets the flags to signal a real result.

Other entry points

NONE

addfba − Add FPB to FPA

Execution addr

BASIC1 &A513
BASIC2 &A50B

Entry conditions:

FPA, FPB contain the numbers to be added

Exit conditions:

FPA:	sum
FPB:	undefined
A	undefined
X	undefined
Y	undefined
C	undefined
Z	undefined

Description

This routine adds the floating point number in FPB to the floating point number in FPA, leaving the result in FPA, and normalises the result. If a subtraction is required, then the number to be subtracted should be negated (using the 'Negate FPA' routine above), and the resulting numbers can added together.

Other entry points

1 addmfa − Add number at (&4B) to FPA

BASIC1 &A50E
BASIC2 &A500

This entry point loads the number at (&4B) into FPB before calling the main routine. On exit, the 'Round FPA and check overflow' routine is called to ensure that it is ready to be stored in memory (a 'Too big' error will be generated if it overflows).

2 subfam − Subtract FPA from number at (&4B)

BASIC1 &A50B
BASIC2 &A4FD

This entry point negates FPA before entering entry point 1 above. The result is left in FPA.

3 submfa − Subtract number at (&4B) from FPA

BASIC1 &A505
BASIC2 &A4D0

This entry point calls entry point 2 above, and then negates the result.

mulfab − Multiply FPA by FPB

Execution addr

BASIC1 &A61E
BASIC2 &A613

Entry conditions:

FPA, FPB contain numbers to be multiplied

Exit conditions:

FPA:	product
FPB:	undefined
&43−&47	undefined
A	undefined
X	undefined
Y	0
C	undefined
Z	1

Description

This routine multiplies the real number in FPA by the real number in FPB, leaving the result in FPA. It does not test for either number being zero on entry, but it will still perform the multiplication correctly, even if one of them is (although it will be quicker if it is discovered before this routine is called). The result of the multiplication is not normalised (or tested for overflow), so the normalising routine should be called before it is written out to memory.

Other entry points

1 mulfam − Multiply FPA by number at (&4B)

BASIC1 &A611
BASIC2 &A606

This entry point loads the number at (&4B) into FPB before calling the main routine. If either number is zero, the routine will exit with a zero result immediately.

2 mufamo − Multiply FPA by (&4B); check overflow

BASIC1 &A661
BASIC2 &A656

This entry point calls entry point 1 above, and then normalises the result. Finally, it rounds the low-order byte into the mantissa, and tests for overflow, generating a ‘Too big’ error (ERR = 20) if it is.

mufa10 − Multiply FPA by 10

Execution addr

BASIC1 &A1E5
BASIC2 &A1F4

Entry conditions:

FPA: number to be multiplied by 10

Exit conditions:

FPA: original number multiplied by 10
FPB: undefined

A undefined
X undefined
Y preserved
C undefined
Z undefined

Description

This routine multiplies the number in FPA by 10. It is faster than the general ‘Multiply FPA by FPB’ routine, and does not use as much temporary memory. It does not test for the number being zero on entry, and will produce an invalid number if this is the case (although calling the ‘Test FPA’ routine afterwards will rectify it). If the number overflows, the ‘exponent overflow byte’ (held in &2F) will be incremented, but no error will be generated at this stage.

Other entry points

NONE

divfab − Divide FPA by FPB

Execution addr

BASIC1 &A6FC
BASIC2 &A6F1

Entry conditions:

FPA: dividend
FPB: divisor

Exit conditions:

FPA: quotient (FP A/FPB)
FPB: undefined

&43−&46 undefined
A 0
X undefined
Y undefined
C undefined
Z 1

Description

This routine divides the number in FPA by the number in FPB, leaving the result in FPA. FPA is then normalised, rounded, and checked for overflow. The routine does not test for either number being zero on entry: if the routine is entered with FPB zero, an invalid result will be obtained.

Other entry points

1 divfam − Divide FPA by number at (&4B)

BASIC1 &A6F2
BASIC2 &A6E7

This entry point divides FPA by the number in memory at (&4B), leaving the result in FPA. If the number at (&4B) is zero, then a ‘Divsion by zero’ error (ERR = 18) will be generated.

2 divmfa − Divide number at (&4B) by FPA

BASIC1 &A6B8
BASIC2 &A6AD

This entry divides the number at (&4B) by FPA, leaving the result in FPA. IfFPA is zero on entry, a ‘Division by zero’ error (ERR = 18) will be generated.

3 recfa − Take reciprocal of FPA (set FPA = 1/FPA)

BASIC1 &A6B0
BASIC2 &A6A5

This entry divides FPA into 1, leaving the result in FPA. If FPA is zero on entry, a ‘Division by zero’ error (ERR = 18) will be generated.

dvfa10 − Divide FPA by 10

Execution addr

BASIC1 &A23E
BASIC2 &A24D

Entry conditions:

FPA: number to be divided by 10

Exit conditions:

FPA: original number divided by 10
FPB: undefined

A undefined
X preserved
Y preserved
C undefined
Z undefined

Description

This routine divides the number in FPA by 10, leaving the result in FPA. The ‘Round and check for overflow’ routine should be called if the result of this is to be stored in memory, as an underflow may have resulted from this division. This routine is faster than the general ‘Divide FPA by FPB’ routine, and does not use as much temporary memory.

Other entry points

NONE

series − Perform series evaluation

Execution addr

BASIC1 &A889
BASIC2 &A897

Entry conditions:

FPA:	argument for series evaluation
A	LSB of pointer to constant list
Y	MSB of pointer to constant list

Exit conditions:

FPA:	result of series evaluation
FPB:	undefined
&43−&48	undefined
&4B−&4E	undefined
A	undefined
X	undefined
Y	undefined
C	undefined
Z	1

Description

This routine performs the series evaluation required by some of the BASIC mathematical functions (e.g. SIN, EXP). On entry, the pointer in A (LSB) and Y (MSB) points to a list of constants to be used: the first byte of the list indicates 1 less than the number of 5-byte floating point constants in it. The algorithm that the series evaluator follows is:

```
A = first constant
REPEAT
        A = X/A + next constant
UNTIL no more constants left
```

where X represents the argument passed to the series evaluator in FPA, and A is the eventual result.

Other entry points

NONE

fixfa − Convert FPA to fixed format

Execution addr

BASIC1 &A40C
BASIC2 &A3FE

Entry conditions:

FPA: floating point number to be fixed

Exit conditions:

If ABS(FPA) < 1 on entry:

FPA: zero
FPB: original number

If ABS(FPA) >= 1 on entry:

FPA sign: sign of number
FPA exponent: &A0
FPA mantissa: 2's complement integer part

FPB sign: zero
FPB exponent: zero
FPB mantissa: ABS value of fractional part

A undefined
X preserved
Y preserved
C undefined
Z undefined

Description

This routine converts the floating point number in FPA into its integer and fractional parts. To find the integer part, the conversion truncates the ABS value of the original number, and then negates it if it was negative. This means that the integer part of '− 1.9' found by this routine would be '− 1' (see 'Type conversion routines': section 10.8 for alternative conversion to integer). If the number is too large for an integer, a 'Too big' error (ERR = 20) will be generated. Note that the integer left in FPA mantissa will be in the opposite order to normal integers: the MSB will be in &31, and the LSB will be in &34.

If the ABS value of the original number is less than 1, then the fractional part (i.e. the original number) will be left as a complete real number in FPB. Otherwise, the ABS value of the fractional part will be left in the mantissa of FPB, with no exponent. This requires an exponent of &80 (representing 2 ', positioning the binary point just above the top bit of FPB mantissa) to be given to it, and the sign should also be transferred from the sign of FPA. The exponent should NOT be set if the number in FPB is already complete.

This routine can be used very easily to find the integer part of a number; but if it is to be used to to extract the fractional part, it may be better to test if the ABS value of FPA is less than 1 before calling it (alternatively, the next routine could be used).

Other entry points

NONE

fracfa − Extract fractional part of FPA

Execution addr

BASIC1 &A494
BASIC2 &A486

Entry conditions:

FPA: number to be used (normalised)

Exit conditions:

&4A LSB of 2's complement integer part
FPA fractional part of number (normalised)

A undefined
X undefined
Y preserved
C undefined
Z undefined list

Description

This routine extracts the integer and fractional parts of the number in FPA, leaving the LSB of the (signed) integer part in &4A, and the fractional part as a real number in FPA. The original number will be rounded to the nearest integer, so that the fractional part will be between −0.5 and +0.5. A 'Too big' error (ERR = 20) will be generated if the number is too large to fit in a 4-byte integer, but no test is made to check if it is outside the range of a single byte (the other 3 bytes of the integer part are lost).

Other entry points

NONE

10.11 Function entry points

This is a list of the equivalent entry points for the easily accessible BASIC functions. Some of the other functions require more than one argument, and others cannot be used outside the environment of the expression evaluator.

The 'Argument' column gives the type of the item which will be operated on by the function. The possibilities are:

----	No argument is expected by this function
real	A real number should be in FPA on entry
integer	An integer should be in IntA on entry
string	A string should be in StrA on entry
numeric	Either 'real' or 'integer', with N set if real

Note that if the function expects a numeric, the N and Z flags should specify the type on entry (as if the 'Get <factor> or <string-factor>' routine had just been used).

On exit from these routines, the result will be in IntA, FPA, or StrA, depending on the result. The type of the result will be in A (&00=string, &40=integer, &FF=real).

Function	Argument	Result	BASIC1	BASIC2
ABS	numeric	numeric	&AD90	&AD6D
ADVAL	integer	integer	&AB59	&AB36
ASC	string	integer	&ACC9	&ACA3
ASN	real	real	&A8CF	&A8DD
ATN	real	real	&A90A	&A90A
CHR$	integer	string	&B3F1	&B3C0
COS	real	real	&A98C	&A990
COUNT	----	integer	&AF26	&AEF7
DEG	real	real	&ABEA	&ABC5
ERL	----	integer	&AFCE	&AF9F
ERR	----	integer	&AFD5	&AFA6
EVAL	string	anything	&AC17	&ABEE
EXP	real	real	&AAB7	&AA94
FALSE	----	integer	&AEF9	&AECA
GET	----	integer	&AFE8	&AFB9
GET$	----	string	&AFEE	&AFBF
HIMEM	----	integer	&AF32	&AF03
INT	numeric	integer	&ACAI	&AC7B
LEN	string	integer	&AF05	&AED6
LN	real	real	&A807	&A801
LOMEM	----	integer	&AF2B	&AEFC
NOT	integer	integer	&ACEA	&ACD4
PAGE	----	integer	&AEEF	&AEC0
PI	----	real	&ABF0	&ABCB
POS	----	integer	&AB92	&AB6D
RAD	real	real	&ABD9	&ABB4
RND	----	integer	&AF80	&AF51
RND()	integer	numeric	&AF41	&AF12
SGN	numeric	integer	&ABB2	&AB8D
SIN	real	real	&A997	&A99B
SOR	real	real	&A7B7	&A7B7
TAN	real	real	&A6CC	&A6C1
TIME	----	integer	&AEE3	&AEB4
TOP	----	integer	&AF13	&AEE6
TRUE	----	integer	&ACEA	&ACC4
USR	integer	integer	&ABFE	&ABD5
VAL	string	numeric	&AC5A	&AC34
VPOS	----	integer	&AB9B	&AB76