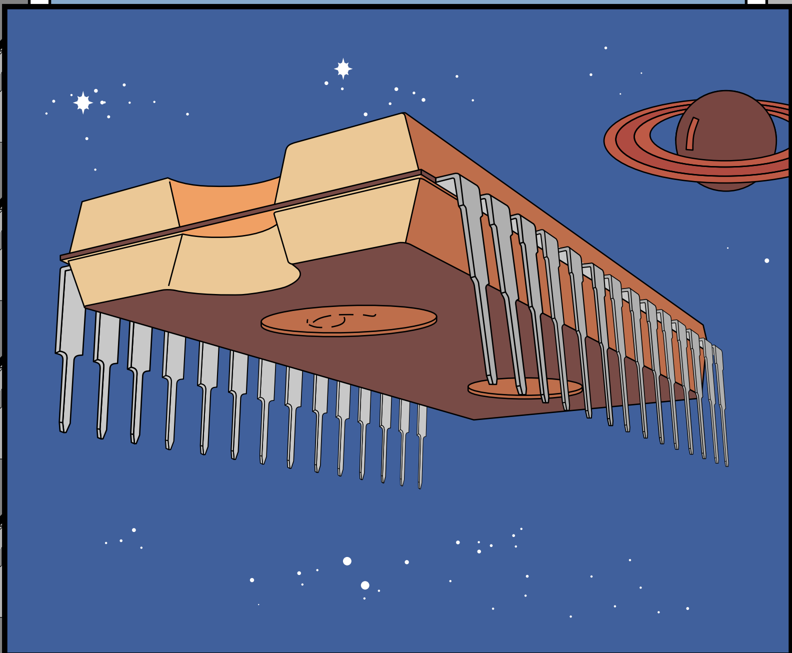
BASIC ROM USER GUIDE

FOR THE BBC MICROCOMPUTER AND ACORN ELECTRON

MARK PLUMBLEY



ADDER

The BASIC ROM User Guide

for the BBC microcomputer and Acorn Electron

Mark D. Plumbley BA,
Churchill College,
Cambridge University

ADDER

Published by Adder Publishing, Cambridge

Published in the United Kingdom by:
Adder Publishing,
PO Box 148,
Cambridge CB1 2EQ

ISBN 0 947929 04 5

First published August 1984
First revision October 2017

The Author would like to thank Adrian Dickens, Nigel Dickens, Tim Gleeson, Ken Vail, Leycester Whewell, Albert Williams and everyone else who helped in the production of this book.

This book was prepared using the Acornsoft VIEW wordprocessor on the BBC Microcomputer and then computer typeset by Parker Typesetting Service, Leicester. Book production by Adder Publishing.

Printed in Great Britain by The Burlington Press Ltd., Foxton, Cambridge.

Contents

Enhancing BASIC

Appendices

Introduction

Many books have been written explaining how to program in BBC BASIC, or how to program in 6502 machine code. Most people therefore know BASIC or machine code without really understanding what BASIC itself is up to. This book fills in that gap by providing a complete description of BASIC as a *system*.

Although BASIC is a very large machine code program, it is essentially very simple, as it is very *structured:* once you can see the overall structure of the system, it is very easy to delve deeper and deeper into its workings, to find out exactly what is happening. This book explains that overall structure: program storage, variable storage, expression evaluation, etc., right down to the mechanisms used by a FOR…NEXT loop or a procedure call. Armed with this knowledge, and the disassembler in chapter 6, you can probe right down to the machine code level of BASIC.

Understanding the operation of a large machine code program such as BBC BASIC has many advantages: not only does it point the way for writing large machine code programs yourself, but it also allows you to write your BASIC programs much more efficiently. Once you know what BASIC has to do to interpret a program, it is possible to write faster programs if you need to, by using resident integer variables wherever possible, using PROCs and FNs rather than GOSUBs, and so on.

The second part of this book describes how to add routines on to BASIC to expand the capabilities of your machine, mainly by trapping the errors that it generates. Adding new commands, overlaying procedures, etc., are all covered, together with how to get back into BASIC to continue afterwards. The examples also show you how to use some of the ROM routines to save space and time in your own machine code programs.

The example programs are complete in that you can type them in and run them, and many of them are useful utilities. However, they also indicate the possibilities available to the adventurous programmer — don't be afraid to chop them about, and use them as a basis to put your own ideas into practice. Chapter 10 provides a comprehensive listing of the BASIC ROM entry points (for both BASIC1 and BASIC2), so that you can experiment with other ideas for new utilities.

Of course, using ROM routines directly will mean that your programs might not work on the Tube, Econet, or with a different BASIC; in fact, the BASIC ROM may not even be 'paged in' when you try to use it. For experimenting with your own machine, however, this doesn't really matter. Commercial programs should *never* use any of these ROM routines; the program might find itself running in a situation you did not allow for. For such programs, or any others which are not restricted to a particular system configuration, only the officially documented facilities should be used.

Note that all Electrons, and the later BBC microcomputers, have BASIC2: the earlier BBC microcomputers have BASIC1. If you are not sure which version of BASIC is in your machine, typing `REPORT` after BASIC has just started up (after a `BREAK` or `*BASIC`), will print the copyright message. If the date is 1981, BASIC1 is fitted; if it is 1982, you have BASIC2. American machines, or those with a second processor, may have US BASIC or HIBASIC: the ROM routines will not be in the same place for these ROMs.

Armed with this book, and plenty of coffee, you should have many happy nights programming. Have fun!

1 The 6502 Microprocessor

At the heart of any microcomputer is the microprocessor. In the BBC micro and Electron this is the 6502, which provides the computer with all its processing power.

By itself, the 6502 is a very simple machine; but it can be made to perform relatively complex tasks (like interpreting programs written in BASIC) by stringing together many of its simple instructions into a machine code program.

This section is not really a tutorial on machine code programming, but more an introduction to the 6502 to give an idea of how the rest of the BASIC system operates around it.

1.1 The 6502 registers

The 6502 has 6 registers altogether: the accumulator A, the index registers X and Y, the program counter PC, the stack register S, and the processor status register P. These are shown in the *programming model*, fig 1.1.

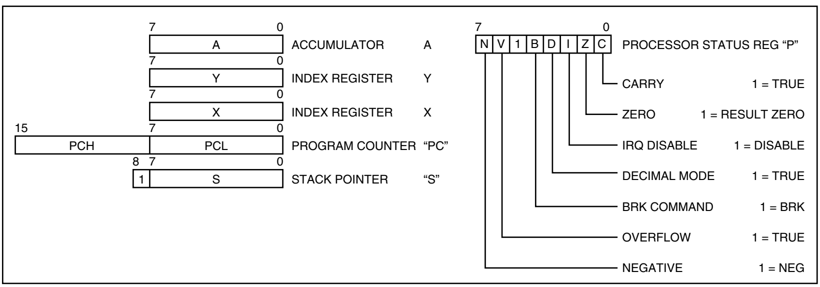


Figure 1.1 – The 6502 programming model.

The accumulator A

The accumulator A is used for all of the arithmetic and logical operations done by the 6502, as well as just loading it from memory and storing it back into memory again. It is the only 6502 register which can be used for adding, subtracting, ANDing, etc. of numbers, and so tends to be used rather a lot. It is 8 bits (1 byte) wide, so it can only hold 256 (&100) different numbers altogether.

As an example, the instruction:

```
AND &80
```

ANDs the 8-bit number in the accumulator with the 8-bit number in location &80 (i.e. ?&80), leaving the result in the accumulator.

The index registers X and Y

Either of these can be used a counter, or as an offset into a table in memory. They can also be loaded from and stored into memory. Again they are only 8 bits wide, so they can only count up to 255 (&FF).

As an example, the instruction:

```
LDA &2000,Y
```

loads the accumulator from the location at &2000+Y. Thus if the Y register contained &2A, the accumulator would be loaded with the contents of location &202A.

The program counter PC

This is the register which tells the 6502 where to get its next instruction from. In a machine code program, the instructions are stored one after another in memory, and the program counter steps through these while they are executed. In practice, you don’t really notice the program counter much (just as you don’t notice the text pointers that BASIC uses to step through *its* program). The program counter is the only 16-bit register that the 6502 has, and allows it to *address* 65536 (&10000) locations.

As an example, the instruction:

```
JMP &8000
```

jumps to location &8000 (in a similar way to the GOTO statement) by loading the number &8000 into the program counter.

The stack pointer S

This register points into a stack in page 1, from &100 to &1FF. Numbers can be *pushed* on the top of the stack, to save them until later, and then *pulled* (or *popped*) again to get back the last number that was *pushed*. This is called a *last in first out* (LIFO) structure, because the first thing that you get out was the last thing that you put in.

When a single byte number is pushed on the stack, it is placed in memory at the location pointed to by the stack pointer (&1F0, say, if the S register contains &F0), and the stack pointer is decremented to point to the location below it in memory. When a byte is pulled, the opposite takes place: the stack pointer is incremented, and the number loaded from the location in page 1 which it points to.

As an example, the instruction:

```
PHA
```

pushes the contents of the accumulator on the 6502 stack.

The processor status register P

This register contains the flags that the 6502 needs for its arithmetic and system operations.

N This is the negative flag. It is set whenever the top bit is set in the 8-bit number just calculated or loaded from memory (see section 1.2 for negative number representation).

V This is set if an overflow occurred the last time an 8-bit signed add or subtract operation was performed (see section 1.2).

B This is the BRK flag. It is set when a BRK instruction is executed (see section 1.3).

D This is the decimal flag. It can be set if any *binary coded decimal* arithmetic is to be performed (see section 1.2).

I This is the interrupt flag. It can be set to prevent the 6502 from being interrupted by a hardware IRQ.

Z This is the zero flag. It is set whenever the 8-bit number just calculated or loaded from memory is zero.

C This is the carry flag. The ADC and SBC instructions use this to indicate whether there was a 'carry over' from the calculation just performed (see section 1.2). It is also used by the shift instructions (section 1.3).

Some of these flags can be tested so that parts of the machine code program are executed conditionally. For example the instruction:

```
BCS carry
```

will branch to the location 'carry' if the carry flag is set: otherwise the program will continue with the instruction after the 'BCS'. The use of these flags is explained more with the instructions in section 1.3.

1.2 Machine code arithmetic

As the 6502 accumulator is only 8 bits wide, it can only represent one of 256 different numbers. Hexadecimal notation is convenient to represent numbers in a byte, because each hexadecimal digit represents 4 bits, so 2 digits represent a whole byte, from &00 to &FF. What the 256 different numbers are used to represent is fairly arbitrary: they can represent positive numbers, negative numbers, or part of a larger number.

1.2.1 Negative numbers

A single byte can be used to represent the positive integers from 0 to 255. This is convenient for counting; but for arithmetic, some way of representing negative numbers is really needed.

If you add the single byte number &04 to &FC, you get &00 (ignoring any carry out of the byte). So, in this case, &FC seems to be behaving as if it was −4 (as '−4' is 'the number which you add to 4 to get 0'). However, it can *also* represent the positive number 252. The answer is that with only 8 bits, you can't tell the difference between '252' or '252 − 256' or '252 + 256' or '252 + any number of 256s'.

So if you want half of the 256 numbers you can represent in a byte to be negative, all you have to do is leave &00 to &7F to be the positive numbers 0 to 127, and let &80 to &FF represent the negative ones. These negative ones will have the same representation as the positive numbers which you get by adding 256 to them, so '−4' will be the same as '−4+256' (252), i.e. &FC.

Choosing the numbers above &80 to be negative is very convenient, because it means that all the numbers with the top bit of the byte set will be negative, while all the numbers with the top bit zero will be positive. Thus the top bit of a signed number like this is the *sign bit* of the number. This is what the N flag in the 6502 is for: it indicates the *sign bit* of the number which has just been operated on.

This representation is often called *2's complement* representation. This is because the negative of a number can be found by changing all the '1's in the binary representation to '0', and all the '0's to '1's (one's complement), and then adding 1 to it. For example, 4 is '00000100', so inverting all the bits we get '11111011', and adding 1 we get '11111100', or &FC. What you're *really* doing when you invert all the bits of a single byte number, is subtracting it from 255 (i.e. '11111111'), so by adding the extra 1 again, you get the number subtracted from 256.

1.2.2 Larger numbers

At first, it may seem a bit restrictive only to be able to represent 256 different numbers in a single byte. However, in decimal, a single digit can only represent one of 10 different numbers (0 to 9), but larger numbers are written down with more than 1 digit, like '59'. In exactly the same way, large numbers can be stored in memory in several bytes, so 1000 (&03E8) can be stored as &03 in one byte (the *most significant byte*, or MSB) and &E8 in the other (the *least significant byte*, or LSB).

When addition is performed in decimal, the least significant digits are added first. Then the next digits are added, together with any *carry* from the first ones, if there was any. The same can be done to add a pair of large numbers in memory: for example, to add 1000 (&03E8) to 25 (&0019) the following operations will take place:

1 Add the LSB of the first number (&E8) to the LSB of the second number (&19). This gives the result &01 with a 1 to carry over to the next byte.

2 Add the MSB of the first number (&03) to the MSB of the second number (&00), with an extra 1 carried over from the last addition. This gives the result &04, with no carry.

The final result of the addition is then &0401, or 1025 in decimal.

The carry over from one byte to the next is done by the C (carry) flag in the 6502 status register. If this is set, the 6502 ADC (add with carry) instruction will automatically add an extra 1 to the addition it is about to do. To add the LSBs together, the carry flag must be cleared first (with the CLC instruction), or an extra 1 may be added where you didn't want one.

Subtraction of larger numbers is done in a very similar way, except the C flag is used as a 'borrow': if it is cleared, the last subtraction needed to borrow 1 from the next byte up, so 1 extra will be subtracted when the next subtraction is performed. To subtract the LSBs, the carry flag must be set first (with the SEC instruction), so the extra 1 is not subtracted.

1.2.3 Overflow

If the single-byte 2's complement number &50, representing 80, is added to the number &33, representing 51, we get &83, which represents −125. Clearly this is not right: the number we should have got was 131. However, 131 is too big to be represented by our single-byte 2's complement number: only the numbers −128 to +127 are allowed. When this happens the result has *overflowed*.

The V (overflow) flag in the 6502 is set if the last add or subtract instruction caused an *overflow*, and the result which was obtained is not a correct 2's complement representation of the answer.

After an addition, the overflow flag will be set if:

(a) a carry occured from bit 6 to bit 7 of the byte, without a carry out of the byte; or

(b) a carry occurred out of the byte without a carry from bit 6 to bit 7.

In other words:

(a) the numbers being added were both positive, but the result is negative; or

(b) the numbers being added were both negative, but the result is positive.

For subtraction, the overflow flag will be set in the corresponding situations, as though you were adding the negative of the number being subtracted.

1.2.4 Binary coded decimal

If the D flag of the 6502 is set it will operate in its binary coded decimal mode, where the 8-bit byte is used to represent two decimal digits, one in each nybble (4 bits). Thus the decimal number 26 will be represented by the hexadecimal number &26. When operating in this mode, all add and subtract operations will automatically adjust the result to ensure that it is in binary coded decimal form again.

This mode is not used very often, although sometimes it is useful for representing decimal numbers exactly.

The decimal flag must never be set when using any operating system or BASIC routines, as they expect to operate in standard binary mode.

1.3 The Instruction Set

The 6502 has 56 different instructions. This section lists them in groups of similar actions, giving the operation of the instruction, and the flags affected by it. Section 1.4 gives the *addressing modes* which can be used with these instructions. Appendix C gives a list of these instructions in alphabetical order.

Load/store operations

LDA The accumulator is loaded with the contents of the specified memory location. Flags affected: N,Z.

LDX The X register is loaded with the contents of the specified memory location. Flags affected: N,Z.

LDY The Y register is loaded with the contents of the specified memory location. Flags affected: N,Z.

STA The contents of the accumulator are stored in memory. The flag bits are unaffected.

STX The contents of the X register are stored in memory. The flag bits are unaffected.

STY The contents of the Y register are stored in memory. The flag bits are unaffected.

Register transfer operations

TAX Copy the contents of the accumulator to the X register. The contents of A are unaffected. Flag bits affected: N,Z.

TAY Copy the contents of the accumulator to the Y register. The contents of A are unaffected. Flag bits affected: N,Z.

TSX Copy the contents of the stack pointer to the X register. The contents of S are unaffected. Flags bits affected: N,Z.

TXA Copy the contents of the X register to the accumulator. The contents of X are unaffected. Flags affected: N,Z.

TXS Copy the contents of the X register to the stack pointer. The contents of X and the status register are unaffected.

TYA Copy the contents of the Y register to the accumulator. The contents of Y are unaffected. Flag bits affected: N,Z.

Stack operations

PHA The contents of the accumulator are pushed on the stack. The stack pointer is updated to point to the next available location. Flag bits are unaffected.

PHP The contents of the processor status register are pushed on the stack, and the stack pointer is updated. Flag bits are unaffected.

PLA The byte on top of the stack is transferred to the accumulator and the stack pointer is updated. Flag bits affected: N,Z.

PLP The byte on top of the stack is transferred to the P register and the stack pointer is updated. All flag bits are affected.

Arithmetic and logical operations

ADC Add the contents of the specified memory location with the carry flag to the accumulator. Result is left in the accumulator. Flags affected: N,V,Z,C.

SBC The specified data is subtracted from the accumulator with a borrow if the carry flag is clear. The result is left in A. C is cleared if a borrow was required else it is set. Flags affected: N,V,Z,C

CMP The contents of the specified memory location are subtracted from the accumulator, setting the flags, but not storing the result. A is unaffected. Flags affected: N is set to bit 7 of the result, Z is set if the result is zero. C is set if the unsigned number in the accumulator is greater than or equal to the data, otherwise cleared (as for the SBC instruction).

CPX The contents of the specified memory location are subtracted from the X register but the result is not stored. The flags are set in the same way as for CMP.

CPY The contents of the specified memory location are subtracted from the Y register but the result is not stored. The flags are set in the same way as for CMP.

AND Performs the bit by bit logical AND of the accumulator and the specified memory location. Result is left in the Accumulator. Flags affected: N,Z.

ORA The bit by bit logical ORing takes place between the accumulator and the memory location, the result is left in A. Flags affected: N,Z.

EOR The contents of the accumulator are exclusive-ored on a bit by bit basis with the specified data, the result is left in A. Flags affected: N,Z.

BIT The logical AND of the accumulator and memory is performed but is not stored. Flag bits affected: Z is set if the result was zero, V and N are set to bits 6 and 7 of the memory location respectively.

Increment/decrement operations

DEC The number in the specified memory location is decremented by 1. Flags affected: N,Z

DEX The number in the X register is decremented by 1. Flags affected: N,Z.

DEY The number in the Y register is decremented by 1. Flags affected: N,Z.

INC The number in the specified memory location is incremented by 1. Flags affected: N,Z.

INX The number in the X register is incremented by 1. Flags affected: N,Z.

INY The number in the Y register is incremented by 1. Flags affected: N,Z.

Shift and rotate operations

ASL The contents of the accumulator or the memory location are shifted one bit to the left. Bit 7 falls in to the carry flag, and bit 0 is set to 0. Flags affected: N,Z,C.

LSR The contents of the accumulator or the memory location are shifted to the right by 1 bit. 0 is placed in bit 7, and bit 0 transferred to C. Flags affected: N is cleared, Z,C.

ROL The contents of the accumulator or the memory location are rotated by one bit to the left. The carry flag is shifted into bit 0, and bit 7 is shifted in to the carry flag. Flags affected: N,Z,C.

ROR The contents of the accumulator or the memory location are rotated by one bit to the right. The carry flag is shifted into bit 7, and bit 0 is shifted in to the carry flag. Flags affected: N,Z,C.

Program control operations

JMP The program counter is loaded with a new address and the program continues from that point. Flags are unaffected.

JSR The contents of the program counter + 2 are pushed on the stack and a new program counter is loaded from the argument. This is called a subroutine call. Flags are unaffected.

RTS The program counter is pulled off the stack and incremented by one, to return from the subroutine. The stack pointer is updated. Flags bits are unaffected.

Conditional branch operations

BCC If the C flag is 0 then branch to the new location, otherwise continue with the next instruction. Flag bits are unaffected.

BCS If the C flag is 1 then branch to the new location, otherwise continue with the next instruction. Flag bits are unaffected.

BEQ If the Z flag is 1 then branch to the new location, otherwise continue with the next instruction. Flag bits are unaffected.

BNE If the Z flag is 0 then branch to the new location, otherwise continue with the next instruction. Flag bits are unaffected.

BMI If the N flag is 1 then branch to the new location, otherwise continue with the next instruction. Flag bits are unaffected.

BPL If the N flag is 0 then branch to the new location, otherwise continue with the next instruction. Flag bits are unaffected.

BVC If the V flag is 0 then branch to the new location, otherwise continue with the next instruction. Flag bits are unaffected.

BVS If the V flag is 1 then branch to the new location, otherwise continue with the next instruction. Flag bits are unaffected.

Flag operations

CLC The Carry flag is cleared, no other flags are affected.

CLD The Decimal flag is cleared, no other flags are affected. This puts the 6502 in binary mode.

CLI The Interrupt flag is cleared, no other flags are affected. This enables interrupts from the IRQ input.

CLV The Overflow bit is cleared, no other flags are affected.

SEC C is set. Other flags remain unaffected.

SED D is set. The ADC and SBC instructions will now operate in the BCD mode. Other flags remain unaffected.

SEI I is set. No IRQs will be acknowledged until it is cleared. Other flag bits are unaffected.

System control operations

BRK This causes an interrupt to be generated and is not maskable. Flags affected: B is set.

NOP The processor does nothing for two cycles.

RTI This pulls the processor status and then the program counter off the stack. The stack pointer is updated. This is used to terminate an interrupt. All flags affected.

1.4 Addressing modes

The *addressing mode* is used to specify how the data needed by an instruction is to be accessed from memory. Most instructions have a single-byte *opcode*, which tells the 6502 which instruction and addressing mode it is, followed by one or two bytes of data to be used by the instruction. Chapter 6 has a table of all the possible opcodes.

Altogether, the 6502 has 13 different addressing modes: these are listed in this section.

Implied addressing

No extra data is required by the instruction. For example:

```
TAX
```

will transfer the contents of the accumulator to the X register, and doesn't need any other information.

Accumulator addressing

No extra data is required by the instruction: it operates on the accumulator. For example:

```
ASL A
```

will shift the accumulator left 1 bit.

Immediate addressing

The single-byte number following the opcode is to be used directly by the instruction. This addressing mode is marked by a '#' in front of the data. For example:

```
ORA #&80
```

will logically OR the contents of the accumulator with the single-byte number '&80' (128).

Absolute addressing

The 2-byte number following the opcode gives the memory location of the data to be used by the instruction. For example:

```
LDY &2000
```

will load the Y register with the contents of memory location &2000.

Zero page addressing

The single-byte number following the opcode gives the memory location in page zero (&0000 to &00FF) of the data to be used by the instruction. This is similar to absolute addressing, except that the MSB of the address is always zero. This is faster than absolute addressing, and takes up only 2 bytes instead of 3 (including the opcode). For example:

```
STA &70
```

will store the contents of the accumulator into the zero page memory location &70.

Absolute indexed addressing

The unsigned contents of the specified index register are added to the 2-byte absolute address following the opcode, to give the location of the data to be used by the instruction. The index register used may be either X or Y, depending on which is allowed with the particular instruction. This addressing mode is marked by a ‘,Y’ or a ‘,X’ following the data. It is useful for accessing tables or reading characters in from a line. For example:

```
DEC &3000,X
```

will decrement the location at &3000+X by 1. If the X register contained &54, the contents of location &3054 will be decremented.

Zero page indexed addressing

The contents of the specified index register are added to the single byte following the opcode, to give the page zero location of the data to be used by the instruction. The carry generated by this addition is ignored: the accessed location is always in page zero. For example:

```
INC &80,X
```

will increment the contents of the location whose LSB is given by &80+X, and whose MSB is &00. Thus if X contains &04, the contents of zero page location &84 will be incremented; if X contains &FE, the contents of zero page location &7E will be incremented.

Relative addressing

The 2’s complement byte following the opcode is added to the program counter to give the location to be used by the instruction. This is only used by the conditional branch instructions. It means that the branch instructions only take up 2 bytes altogether, but the location which is being branched to must be a maximum of −128 to +127 away from the location of the instruction following the branch instruction. For example:

```
.loop  BEQ loop
```

will branch back to the same location if the Z flag is set. The byte following the opcode will be &FE (−2) for this instruction, because the branch instruction is 2 bytes back from the next instruction which would be executed if the branch did not take place.

Indirect addressing

The 2-byte absolute address following the opcode points to two consecutive bytes which contain the LSB and the MSB of the location to be used. The two bytes are stored LSB first, MSB second. This addressing mode is only used by the JMP instruction. For example:

```
JMP (&0200)
```

will jump to the location whose address is contained in &0200 (LSB) and &0201 (MSB).

Pre-indexed indirect addressing

The contents of the X register are added to the single byte following the opcode, to give the zero page location of two consecutive bytes (LSB first) which contain a pointer to the data. For example:

```
LDA (&50,X)
```

will use the number in &50+X (LSB) and &51+X (MSB) as a pointer to the number to be loaded into the accumulator. Thus if X contained &20, location &70 contained the number &34, and location &71 contained the number &12, the number in location &1234 would be loaded into the accumulator.

Post-indexed indirect addressing

The single byte following the opcode gives the zero page location of a 2-byte pointer (LSB first). The unsigned contents of the Y register are added to this pointer, to give the address to be used by the instruction. This instruction mode is very useful for pointing into memory: a pair of page zero locations hold the base of a pointer into memory, and Y holds the offset from that pointer. For example:

```
CMP (&2A),Y
```

will compare the accumulator with the byte pointed to by the base pointer in &2A (LSB) and &2B (MSB), offset by Y. Thus if &2A contains &00, and &2B contains &40, and Y contains &45, the accumulator will be compared with the contents of location &4045.

1.5 Addressing mode groups

A table of allowed addressing modes for each instruction is given on page 508 of the BBC *User Guide*, and the *Electron User Guide* details them in chapter 29. This section summarises the groups of instructions which use the same (or nearly the same) set of addressing modes.

These addressing mode groups are used extensively by the built-in assembler in BASIC. See chapter 6 for more on this.

Implied group

These instructions only use implied addressing. The instructions are:

BRK, CLC, CLD, CLI, CLV, DEX, DEY, INX, INY, NOP, PHA, PHP, PLA, PLP, RTI, RTS, SEC, SED, SEI, TAX, TAY, TSX, TXA, TXS, TYA.

Relative branch group

These instructions only use relative addressing. The instructions are:

BCC, BCS, BEQ, BMI, BNE, BPL, BVC, BVS.

Accumulator operation group

The instructions in this group are:

ADC, SBC, CMP, AND, EOR, ORA, LDA, STA.

These instructions all operate on the accumulator, and allow the following addressing modes:

Immediate (not STA)
Zero page Absolute
Zero page,X
Absolute,X
Absolute,Y
(Indirect,X)
(Indirect),Y

Shift group

The instructions in this group are:

ASL, LSR, ROL, ROR

and they allow the following addressing modes:

Accumulator
Zero page
Absolute
Zero page,X
Absolute,X

Count group

The instructions in this group are:

DEC, INC

and they allow the following addressing modes:

Zero page
Absolute
Zero page,X
Absolute,X

Test group

The instructions in this group are:

BIT, CPX, CPY

and they allow the following addressing modes:

Immediate (not BIT)
Zero page
Absolute

Index load group

The instructions in this group are:

LDX, LDY

and they allow the following addressing modes:

Immediate
Zero page
Absolute
Zero page,X (',Y' for LDX)
Absolute,X (',Y' for LDX)

Index store group

The instructions in this group are:

STX, STY

and they allow the following addressing modes:

Zero page
Absolute
Zero page,X (',Y' for STX)

Jump group

The instructions in this group are:

JMP, JSR

and they allow the following addressing modes:

Absolute
(Indirect) (not JSR)

1.6 The BASIC assembler

The BBC *User Guide* and the *Electron User Guide* give an adequate description of the use of the built-in assembler, so I won't cover it again here. However, BBC micro owners may not be aware of the extra facilities available on the assembler in BASIC 2, over that in BASIC 1 (which is the one described in the *User Guide*). These extra facilities are remote assembly, and the EQU directive.

1.6.1 Remote assembly

The OPT directive controls the action of the assembler while it is in operation. The OPT is followed by a number whose lower 3 bits (only 2 bits in BASIC 1) set the assembler options. These bits are as follows:

Bit	**Option**
0	assembly listing if set
1	errors enabled if set
2	remote assembly if set

Remote assembly allows a machine code program to be assembled to run in one part of memory, but the code put in another. For example, an assembler routine which will be in a paged ROM can be assembled correctly for &8000 onwards, but the code can be placed at &2000 onwards, say, where there is RAM.

If this is being used, P% should be set up to point to the location where the routine will end up (&8000 in the above example), but O% should point to the location where the generated code is to be stored.

1.6.2 The EQU directives

This allows data to be incorporated as part of a machine code program, without having to leave the assembler. The directives available are:

EQUB	equate byte	reserves 1 byte
EQUW	equate word	reserves 2 bytes
EQUD	equate double word	reserves 4 bytes
EQUS	equate string	reserves a string

Note that the EQUS directive only reserves the space for the characters of the string; if a carriage return or CRLF is needed on the end, this must be done separately with an EQUB directive.

For example:

```
EQUB &40
EQUS "HI"
EQUW &1234
```

Will reserve and initialise the following bytes in memory:

```
&40
&48 ("H")
&49 ("I")
&34
&12
```

Using the EQU directive is not only more convenient than using the BASIC equivalent, but it also makes the program much more readable. Many of the programs in this book use the EQU directive, although where it has been used, the alternative BASIC form is available for BASIC 1 users.

2 The BASIC System

The BBC microcomputer system has been designed to allow many different languages (like LISP or FORTH) to be used with it. However, the language that all BBC micros and Electrons start with is BBC BASIC.

2.1 An overview of BASIC

When BASIC is initialised, it takes control of the computer. It prints 'BASIC' on the screen, and prompts for a line to be input. You then type in programs, RUN them, edit and RUN them again until they work, and continue until the power is switched off.

Beneath all of this is 16K of 6502 machine code, in a paged ROM sitting between &8000 and &BFFF, beavering away trying to work out what to do with the line that you just typed in. It is really a whole system all by itself, editing programs, interpreting program statements, evaluating expressions, handling variables; in fact it does everything except actually input and output to the hardware (it leaves that to the Machine Operating System).

Fig 2.1 shows a general overview of BASIC, with its main component parts. The first major section of the BASIC system is the command handler and the statement interpreter. When a line is input at the keyboard, the command handler *tokenises* it, and decides whether to insert it into the program (if it starts with a line number), or to send it to the statement interpreter. The statement interpreter is also used to handle program statements. The action of the command handler and statement interpreter is described in sections 2.3 and 2.4.

The other major section of the BASIC system shown in fig 2.1 is the expression evaluator. This is called by most of the statement handlers (or function handlers) when they want a number or a string to operate on. For example, the MODE statement handler calls the expression evaluator to get the number of the MODE that is to be used. The expression evaluator is described in more detail in chapter 4.

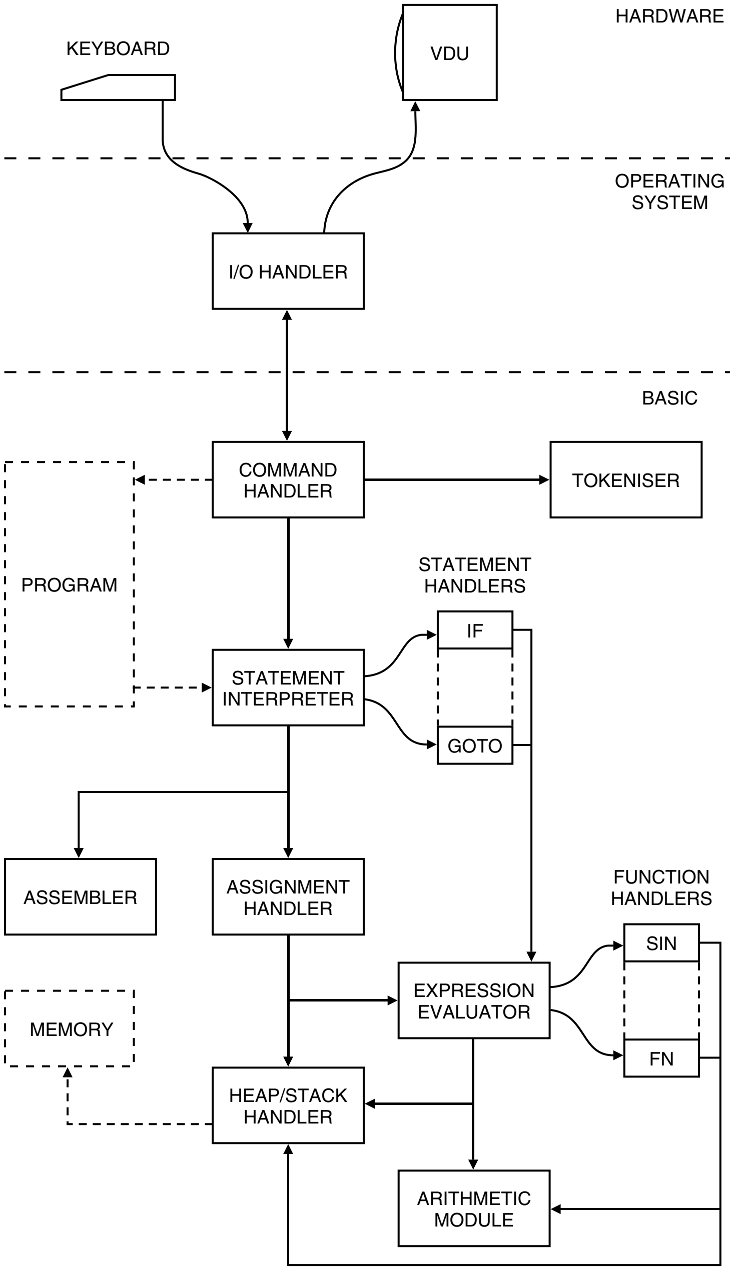


Figure 2.1 – The BASIC system.

The arithmetic module is a collection of routines which is used to perform the calculations required by the expression evaluator (and by the statement and function handlers). Most of these have to be floating point routines, as real numbers are more difficult for the computer to handle than integers or strings. These routines are detailed in chapter 10.

The HEAP/STACK handler is another collection of routines, but these deal with variables and other use of memory by BASIC while the program is running (*dynamic* memory use). Variables, and BASIC's memory use are described in chapter 3.

2.2 The BASIC 'CPU'

The 6502 CPU is a versatile machine, but on its own it is a bit limited. Its 8-bit accumulator, A, can only handle single byte integers; it can't deal with real numbers or strings; it can't allocate space for BASIC variables, and its stack is only 255 bytes deep. To get round this, BASIC has a software 'layer' on top the 6502, to provide a more versatile service.

This new 'layer' has a collection of page 0 locations as 'registers', which are manipulated by the 6502. These registers (together with the routines to handle them) make up the 'Central Processing Unit' of the BASIC system. Fig 2.2 compares the 6502 registers with BASIC's registers.

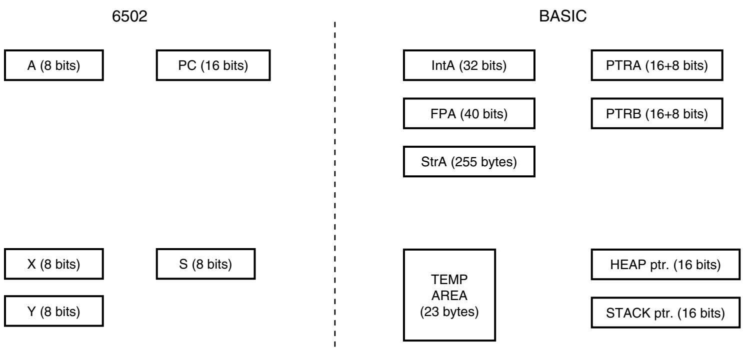


Figure 2.2 – 6502/BASIC registers.

2.2.1 BASIC Integers

Where the 6502 only allows 8-bit integers to be used, most of BASIC's integer work is done with 32-bit (4-byte) integers. For this it has a 4-byte integer accumulator, IntA, stored in page zero at &2A to &2D. The format of the 4-byte integers stored in this accumulator is shown in fig 2.3.

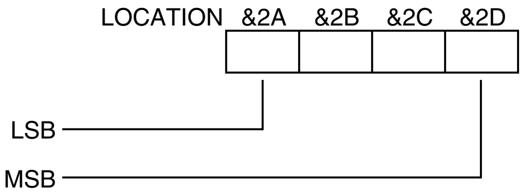


Figure 2.3 – Integer format.

Note that the least significant byte (LSB) is stored *first*, at &2A, with the most significant byte (MSB) at &2D. This means that a single-byte (positive) value at &2A can be converted into a 4-byte integer starting at &2A, by setting the 3 most significant bytes (in &2B, &2C and &2D) to zero.

2.2.2 Real numbers

One of the major advantages of the BASIC 'CPU' over the 6502 equivalent is its ability to deal with real numbers, rather than just integers. For this, it has 2 floating point accumulators, FPA and FPB. For those not familiar with binary floating point representation, here is a brief description.

Decimal integers can be written in binary form, like

9 (decimal)   can be written as: 1001 (binary).

Fractions can be written in decimal by using a decimal point, like '9.6', and binary numbers can be written in a similar form. Thus '0.1' (binary) represents 1/2 (0.5 decimal), '0.01' (binary) represents 1/4 (0.25 decimal), and so on. As an example,

3.625 (decimal)   can be written as: 11.101 (binary)

Using this would give a way to represent numbers on a computer; by holding the integer part as one number, and the fractional part as another. In practice, though, for many applications this is just too limited.

In decimal, for talking about a much wider range of numbers, *scientific form* or *standard form* can be used. For this, the number to be expressed is written down as a number between 1 and 10 (this is the *mantissa*), multiplied by ‘10 to the power of’ another number (this is the *exponent*). Thus 273 can be written as 2.73×10^2 (or 2.73E2).

For the binary representation of real numbers, BASIC uses a similar form to the decimal one: the number to be expressed is written as a number between 1/2 and 1 (not equal to 1), multiplied by ‘2 to the power of’ another number. Thus 11.101 (binary) can be written as 0.11101×2^2 (the exponent is in decimal for clarity). This is often called *floating point* representation, as the actual position of the *binary point* in the number is not fixed to a particular position (in integers, for example, the binary point is always just beneath the least significant bit).

When floating point numbers are stored in variables, they occupy 5 bytes, and are stored as shown in fig 2.4.

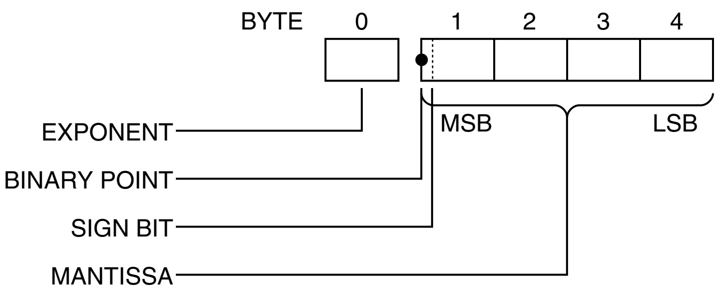


Figure 2.4 – Floating point packed format.

The exponent is stored offset by &80 – i.e. &80 represents 2^0,

&81 represents 2^1, and so on. This allows the number zero to be represented by a floating point number with all its bytes set to 0. Note that zero doesn't fit in to this floating point representation: it is smaller than 2^{-127}, yet it is larger than -2^{-127}. It has to be represented as a special case.

The position of the binary point in the mantissa is just above the most significant bit.

The mantissa is always a number between 1/2 (0.1 binary) and 1 (but not equal to 1), so the top bit of the mantissa is always a '1'. This means that this bit position is not needed for the mantissa (it can always be retrieved by ORing the MSB of the mantissa with &80), so this bit is used to store the sign bit of the number (the top bit of the mantissa will not be a '1' if the number being represented is zero)

The mantissa occupies 4 bytes. This means that 4-byte integers can be converted to floating point format, and back again, without loss of accuracy. The bytes are stored MSB first, LSB last; the opposite order to integers. The mantissa is stored as a positive number, and not in 2's complement format (so the representation for '6' is just the same as the representation for '−6', except the sign bit will be changed).

When a 'packed' floating point number is loaded into one of the floating point accumulators, FPA or FPB, it is unpacked into 8 bytes. The format of these accumulators is shown in fig 2.5.

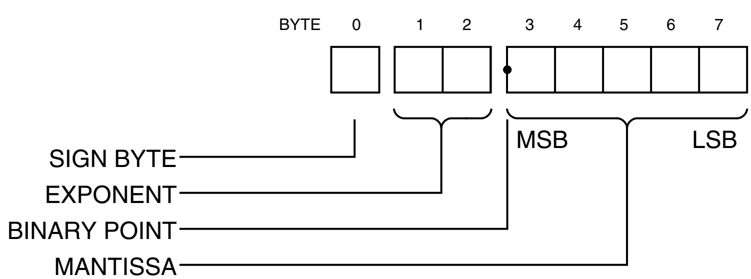


Figure 2.5 – Floating point accumulator format.

The exponent has been expanded into 2 bytes; the high-order byte of the exponent is set to zero when the number is loaded in. This allows results of calculations to temporarily overflow (i.e. the exponent becomes too large for the 5-byte representation to handle), providing that they end up in the correct range before being written out to memory again in the 5-byte packed format. The exponent is still offset by &80.

The mantissa has been expanded to 5 bytes instead of 4. This allows for extra accuracy in the middle of calculations. Before the number is written back out to memory, this extra byte is used to round the rest of the mantissa.

The sign bit has been removed to a whole byte by itself, and the top bit of the mantissa has been restored to '1'. For calculations, this '1' is needed in the top bit where it is supposed to be.

Often during a calculation, the top bit does not stay set (perhaps due to a number almost equal to it being subtracted from it). If this is the case, the value of the number is still given correctly (as the mantissa multiplied by '2 to the power of' the exponent), but the mantissa is now much less than 1/2. Before the number can be written out into memory, the number must be 'normalised' by repeatedly multiplying the mantissa by 2 (i.e. shifting it up by 1 bit), and decrementing the exponent (dividing that part of the representation by 2) to compensate, until the top bit of the mantissa becomes set again.

If this happens, some of the accuracy of the number may have been lost, as some of the bits of the number may have 'fallen off the bottom' before the number was shifted back up again.

Floating point numbers do have certain limitations:

(a) The largest number which can be represented (in the 5-byte format) is just less than 1.0×2^{127} (1.7×10^{38}).

(b) The smallest number (in magnitude) which can be represented (apart from zero) is 1.0×2^{-128} (2.9×10^{-39}).

(c) Because just 32 bits are used to hold the mantissa of the number, the representation is only accurate to 1 part in 2^{32}

(1 part in 4×10^9). This means that if any number stored in this format is printed out in decimal, it will only be accurate to the first 9 decimal digits.

(d) Calculations involving floating point numbers take longer than those involving integers.

The actual format of the floating point accumulators is:

FPA	**FPB**	**USE**
&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 of 5-byte format)
&35	&42	mantissa low-order rounding byte.

2.2.3 Strings

For string handling, BASIC has a string ‘accumulator’, StrA. All of page 6 is allocated to the string accumulator; the characters of StrA are stored from &600 onwards, with location &36 in page zero used to hold the length of the string.

This makes string handling relatively simple, although it does take up a lot of memory.

2.2.4 General workspace

In addition to these accumulators, BASIC has a general workspace area, between &37 and &4E, which it uses for general pointers (instead of the 6502 X and Y registers) and for other different purposes, depending on which part of the system is in operation at the time. FPB is actually in this area, and several routines which do not need to do any floating point calculations may use the same memory that it occupies.

2.2.5 Program pointers

Instead of the Program Counter (PC) of the 6502, BASIC has two pointers, PTRA and PTRB, which it uses to scan through a BASIC program (or a line typed in at the keyboard). Both of these pointers are composed of a 2-byte base pointer, and a single-byte offset from that base. PTRA is mainly used to read the first part of a statement until the statement token is recognised, and PTRB is mainly used for scanning expressions. The format of these pointers is:

&B,&C	PTRA base
&A	PTRA offset
&19,&1A	PTRB base
&1B	PTRB offset

2.2.6 Dynamic memory pointers

The 6502 only has one way of dynamically allocating space during a program: its stack. This works downwards in page 1 with a maximum size of 256 bytes (i.e. from &1FF down to &100).

Rather than using this, BASIC has a STACK which works downwards in memory from HIMEM. It uses this to hold temporary results from calculations, or when a FN or PROC is called. BASIC also has a HEAP which works upwards in memory from LOMEM (usually the TOP of the program), which is where it puts any variables (apart from resident integers). Together, the BASIC STACK and the HEAP can use up all of the memory between the TOP of the program and the bottom of the screen. Chapter 3 describes how variables are stored, and the use of the HEAP and the STACK.

2.3 Tokenising

When a line is typed in at the keyboard, it is inserted into BASIC's keyboard buffer in page 7 (from &700 onwards). From here, the command handler sends the line to the tokeniser, so that the keywords can be *tokenised*. This involves looking through the line and replacing occurrences of keywords (and their abbreviations) in the line by a single byte *token*, with a value between &80 and &FF. This saves memory when the line is put into a program (as, for example, PRINT takes up only 1 byte instead of 5), and it makes it a lot easier (and faster) to recognise the keyword when it is to be *interpreted*.

2.3.1 Keyword tokenising

The keyword table is stored at &806D (BASIC1) or &8071 (BASIC2), in roughly alphabetical order. The format of each entry is:

Keyword
Single-byte token
Flag byte

Table 2.1 gives a list of the keyword tokens, and the address where they JMP to when recognised, in token value order. From this it can be seen that the tokens are divided up into several groups:

&80 to &84	operators
&85 to &8C	auxiliary tokens
&8D	line number token (see section 2.3.2)
&8E	'OPENIN' for BASIC2
&8F to &93	pseudo-variable functions
&94 to &BC	numeric-valued functions
&BD to &C4	string-valued functions
&C5	'EOF'
&C6 to &CD	commands
&CE	(not used)
&CF to &D3	pseudo-variable statements
&D4 to &FF	statements

The tokeniser does not simply tokenise the line: it obeys certain rules, and can be in several states. The flag byte is used to give instructions to the tokeniser about how to continue tokenising the rest of the line, or how to tokenise this keyword. The flags are used as follows:

Bit 0 **C**onditional flag. If this is set, this tells the tokeniser not to tokenise this keyword if it is followed by an alphanumeric character. This means, for example, that 'TIMER' can be used as a variable name, as the 'TIME' part of it will not be tokenised.

Bit 1 **M**iddle flag. If this is set, this tells the tokeniser to go to 'middle of statement' mode after this token.

Bit 2 **S**tart flag. If this flag is set, this tells the tokeniser to go to 'start of statement' mode. The tokeniser must know if it is at the start of a statement or not, because a '*' at the start of a statement will cause tokenising to be abandoned so that the rest of the line can be sent to OSCLI untokenised. If a '*,' is found in the middle of a statement, it will be in the middle of an expression, so the rest of the line should be tokenised. It also needs to know if a pseudo-variable found is a statement or a function.

Bit 3 **F**N/PROC flag. If this flag is set (as it is for FN or PROC), this tells the tokeniser not to tokenise the name immediately following the token. This means, for example, that the 'ERROR' part of 'PROCERROR' will not be tokenised.

Bit 4 **L**ine number flag. If this flag is set, it tells the tokeniser to start tokenising line numbers after this token. This flag is set for keywords like 'GOTO' or 'RENUMBER'. Line number tokenising is usually turned off after any other symbol apart from a ',', a HEX number, or a string.

Bit 5 **R**EM flag. If this is set, it tells the tokeniser to stop tokenising the rest of the line. This flag is used by the 'DATA' and 'REM' tokens.

Bit 6 **P**seudo-variable flag. If this is set, it tells the tokeniser to add &40 to this token if it is found at the start of a statement. This is how the tokeniser decides whether a pseudo-variable is a statement or a function. Note that the pseudo-variable *statement* entry in the token table is not used by the tokeniser; it uses the function entry and converts it to the statement token if it is at the start of a statement. The statement entry is used by 'LIST' when the tokens are being printed out.

Bit 7 (not used)

Other symbols

Special symbols found in the input line which affect tokenising are:

&	scans the following hex number
"	scans the following string constant
:	goes to 'start of statement' state
*****	prevents tokenising if at the start of a statement

2.3.2 Line number tokenising

Line numbers can also be tokenised, as well as keywords. However, they will be left alone unless they are found at the start of a line, or after a token with the 'tokenise line numbers' flag set.

Note that the tokenised line number at the start of the line is not inserted into the program (see section 2.4 for program storage).

Tokenising line numbers speeds up the use of GOTOs or GOSUBs in a program, because the numbers are simpler to decode than an ASCII string of digits; but it does not really save very much memory, as each tokenised line number takes up 4 bytes. Fig 2.6 shows how line numbers are tokenised, once the ASCII digits have been read in and converted to a 16-bit integer (it is actually a 15-bit integer, as line numbers greater than 32767 are not allowed).

The bytes after the &8D line number token *must* be less than &80, or they may look like another token. If this was not the case, one of them may look like an 'ELSE' token, and it may be latched on to by the 'IF' statement as something to do if it got a FALSE result (see section 5.4).

Also, the bytes after the line number token must not be allowed to be a control character (i.e. less than &20). If this was not the case, the byte may look like a &0D (carriage return), which marks the end of a line in a program.

The simplest way to ensure that both of these conditions are met, is to fix the top 2 bits of each byte to '01' so that it is in the range &40 to &7F.

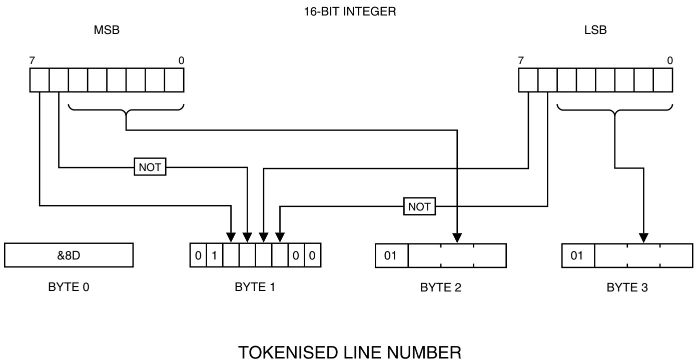


Figure 2.6 – Line number tokenising.

So to convert a 16-bit integer to the tokenised line number format:

1 Set byte 0 to the &8D line number token.

2 Transfer the bottom 6 bits of the LSB of the integer into byte 2 of the tokenised line number, setting bits 7 and 6 to '01'.

3 Transfer the bottom 6 bits of the MSB of the integer into byte 3 of the tokenised line number, setting bits 7 and 6 to '01'.

4 Set byte 1 of the tokenised line number to '01000000' (binary).

5 Transfer bits 7 and 6 of the LSB of the integer into bits 5 and 4 of byte 1 of the tokenised line number, inverting bit 6 before it is inserted into bit 4.

6 Transfer bits 7 and 6 of the MSB of the integer into bits 3 and 2 of byte 1 of the tokenised line number, inverting bit 6 before it is inserted into bit 2.

The line number is now tokenised. It is a bit easier to get the line number out of the tokenised form:

1 Shift byte 1 of the tokenised line number up 2 bits, load it into A, and mask off the bottom 6 bits.

2 EOR this with byte 2 of the tokenised line number. A now contains the LSB of the number.

3 Shift byte 1 of the tokenised line number up by a further 2 bits, and load it into A (the bottom 6 bits are all 0)

4 EOR this with byte 3 of the tokenised line number. A now contains the MSB of the number.

Table 2.1 – Keyword Tokens

Token	**BASIC1**			**BASIC2**		
	Keyword	**Flags**	**Addr**	**Keyword**	**Flags**	**Addr**
80	AND	--------	----	AND	--------	----
81	DIV	--------	----	DIV	--------	----
82	EOR	--------	----	EOR	--------	----
83	MOD	--------	----	MOD	--------	----
84	OR	--------	----	OR	--------	----
85	ERROR	-----S--	----	ERROR	-----S--	----
86	LINE	--------	----	LINE	--------	----
87	OFF	--------	----	OFF	--------	----
88	STEP	--------	----	STEP	--------	----
89	SPC	--------	----	SPC	--------	----
8A	TAB(	--------	----	TAB(	--------	----
8B	ELSE	---L-S--	----	ELSE	---L-S--	----
8C	THEN	---L-S--	----	THEN	---L-S--	----
8D	line no.	--------	----	line no.	--------	----
8E	---	--------	----	OPENIN	--------	BF78
8F	PTR	-P----MC	BF50	PTR	-P----MC	BF47
90	PAGE	-P----MC	AEEF	PAGE	-P----MC	AEC0
91	TIME	-P----MC	AEE3	TIME	-P----MC	AEB4
92	LOMEM	-P----MC	AF2B	LOMEM	-P----MC	AEFC
93	HIMEM	-P----MC	AF32	HIMEM	-P----MC	AF03
94	ABS	--------	AD8D	ABS	--------	AD6A
95	ACS	--------	A8C6	ACS	--------	A8D4
96	ADVAL	--------	AB56	ADVAL	--------	AB33
97	ASC	--------	ACC4	ASC	--------	AC9E

98	ASN	---------	A8CC	ASN	---------	A8DA
99	ATN	---------	A907	ATN	---------	A907
9A	BGET	--------C	BF78	BGET	--------C	BF6F
9B	COS	---------	A989	COS	---------	A98D
9C	COUNT	--------C	AF26	COUNT	--------C	AEF7
9D	DEG	---------	ABE7	DEG	---------	ABC2
9E	ERL	--------C	AFCE	ERL	--------C	AF9F
9F	ERR	--------C	AFD5	ERR	--------C	AFA6
A0	EVAL	---------	AC12	EVAL	---------	ABE9
A1	EXP	---------	AAB4	EXP	---------	AA91
A2	EXT	--------C	BF4F	EXT	--------C	BF46
A3	FALSE	--------C	AEF9	FALSE	--------C	AECA
A4	FN	----F---	B1C4	FN	----F---	B195
A5	GET	---------	AFE8	GET	---------	AFB9
A6	INKEY	---------	ACD3	INKEY	---------	ACAD
A7	INSTR(	---------	AD08	INSTR(	---------	ACE2
A8	INT	---------	AC9E	INT	---------	AC78
A9	LEN	---------	AF00	LEN	---------	AED1
AA	LN	---------	A804	LN	---------	A7FE
AB	LOG	---------	ABCD	LOG	---------	ABA8
AC	NOT	---------	ACF7	NOT	---------	ACD1
AD	OPENIN	---------	BF85	OPENUP	---------	BF80
AE	OPENOUT	---------	BF81	OPENOUT	---------	BF7C
AF	PI	--------C	ABF0	PI	---------C	ABCB
B0	POINT(	---------	AB64	POINT(	---------	AB41
B1	POS	--------C	AB92	POS	---------C	AB6D
B2	RAD	---------	ABD6	RAD	---------	ABB1
B3	RND	--------C	AF78	RND	---------C	AF49
B4	SGN	---------	ABAD	SGN	---------	AB88
B5	SIN	---------	A994	SIN	---------	A998
B6	SQR	---------	A7B4	SQR	---------	A7B4
B7	TAN	---------	A6C9	TAN	---------	A6BE
B8	TO	---------	AF0B	TO	---------	AEDC
B9	TRUE	--------C	ACEA	TRUE	--------C	ACC4
BA	USR	---------	ABFB	USR	---------	ABD2
BB	VAL	---------	AC55	VAL	---------	AC2F
BC	VPOS	--------C	AB9B	VPOS	--------C	AB76
BD	CHR$	---------	B3EE	CHR$	---------	B3BD
BE	GET$	---------	AFEE	GET$	---------	AFBF
BF	INKEY$	---------	B055	INKEY$	---------	B026
C0	LEFT$(	---------	AFFB	LEFTS(	---------	AFCC
C1	MID$(	---------	B068	MID$(	---------	B039
C2	RIGHT$(	---------	B01D	RIGHT$(	---------	AFEE
C3	STR$	---------	B0C3	STR$	---------	B094
C4	STRING$(	---------	B0F1	STRING$(	---------	B0C2
C5	EOF	--------C	ACDE	EOF	--------C	ACB8
C6	AUTO	---L----	905F	AUTO	---L----	90AC
C7	DELETE	---L----	8ECE	DELETE	---L----	8F31
C8	LOAD	------M-	BF2D	LOAD	------M-	BF24
C9	LIST	---L----	B5B5	LIST	---L----	B59C
CA	NEW	--------C	8A7D	NEW	--------C	8ADA
CB	OLD	--------C	8A3D	OLD	--------C	8AB6

CC	RENUMBER	---L----	8F37	RENUMBER	---L----	8FA3
CD	SAVE	------M-	BEFA	SAVE	------M-	BEF3
CE	---	--------	9839	---	--------	982A
CF	PTR	--------	BF39	PTR	--------	BF30
D0	PAGE	--------	9239	PAGE	--------	9283
D1	TIME	--------	927B	TIME	--------	92C9
D2	LOMEM	--------	9224	LOMEM	--------	926F
D3	HIMEM	--------	9212	HIMEM	--------	925D
D4	SOUND	------M-	B461	SOUND	------M-	B44C
D5	BPUT	------MC	BF61	BPUT	------MC	BF58
D6	CALL	------M-	8E6C	CALL	------M-	8ED2
D7	CHAIN	------M-	BF33	CHAIN	------M-	BF2A
D8	CLEAR	-------C	9326	CLEAR	-------C	928D
D9	CLOSE	------MC	BF9E	CLOSE	------MC	BF99
DA	CLG	-------C	8E57	CLG	-------C	8EBD
DB	CLS	-------C	8E5E	CLS	-------C	8EC4
DC	DATA	--R-----	8AED	DATA	--R-----	8B7D
DD	DEF	--------	8AED	DEF	--------	8B7D
DE	DIM	------M-	90DD	DIM	------M-	912F
DF	DRAW	------M-	93A5	DRAW	------M-	93E8
E0	END	-------C	8A50	END	-------C	8AC8
E1	ENDPROC	-------C	9310	ENDPROC	-------C	9356
E2	ENVELOPE	------M-	B49C	ENVELOPE	------M-	B472
E3	FOR	------M-	B7DF	FOR	------M-	B7C4
E4	GOSUB	---L--M-	B8B4	GOSUB	---L--M-	B888
E5	GOTO	---L--M-	B8EB	GOTO	---L--M-	B8CC
E6	GCOL	------M-	932F	GCOL	------M-	937A
E7	IF	------M-	9893	IF	------M-	98C2
E8	INPUT	------M-	BA62	INPUT	------M-	BA44
E9	LET	-----S--	8B57	LET	-----S--	8BE4
EA	LOCAL	------M-	92D5	LOCAL	------M-	9323
EB	MODE	------M-	935A	MODE	------M-	939A
EC	MOVE	------M-	93A1	MOVE	------M-	93E4
ED	NEXT	------M-	B6AE	NEXT	------M-	B695
EE	ON	------M-	B934	ON	------M-	B915
EF	VDU	------M-	93EF	VDU	------M-	942F
F0	PLOT	------M-	93AE	PLOT	------M-	93F1
F1	PRINT	------M-	8D33	PRINT	------M-	8D9A
F2	PROC	----F-M-	92B6	PROC	----F-M-	9304
F3	READ	------M-	BB39	READ	------M-	BB1F
F4	REM	--R-----	8AED	REM	--R-----	8B7D
F5	REPEAT	--------	BBFF	REPEAT	--------	BBE4
F6	REPORT	-------C	BFE6	REPORT	-------C	BFE4
F7	RESTORE	---L--M-	BB00	RESTORE	---L--M-	BAE6
F8	RETURN	-------C	B8D5	RETURN	-------C	B8B6
F9	RUN	-------C	BD29	RUN	-------C	BD11
FA	STOP	-------C	8A59	STOP	-------C	8AD0
FB	COLOUR	------M-	9346	COLOUR	------M-	938E
FC	TRACE	---L--M-	9243	TRACE	---L--M-	9295
FD	UNTIL	------M-	BBCC	UNTIL	------M-	BBB1
FE	WIDTH	------M-	B4CC	WIDTH	------M-	B4A0
FF	---	--------	9839	OSCLI	------M-	BEC2

2.4 Program storage

Once the line has been tokenised, the command handler checks to see if it starts with a line number. If it is, it is inserted into the program (and the old line with the same number, if there is one, is deleted). The format of each line is as follows:

00	MSB of line number
01	LSB of line number
02	length byte (= 'XX')
03	first character of line text
04	etc.
XX−1	&0D (carriage return) line terminator.
XX	start of next line

The length byte is used so that searching for a line number (for a 'GOTO' or 'GOSUB' statement) is much faster. If this length byte is not set up correctly, BASIC will give a 'Bad program' error (see section 9.2 for a salvage routine).

The first character in memory at PAGE is a carriage return character: this gives something to 'latch on to' when BASIC checks for a 'Bad program'. The routine that checks this also sets TOP to point to the next free location after the end of the program.

The end of the program is marked by a byte with the top bit set (i.e. &80 or greater) in the position which would be the MSB of the line number of the next line. This is why line numbers greater than 32767 are not allowed: if one got in, the MSB of its line number would just mark the end of the program.

For example, the program '10PRINT A' would be stored as (if PAGE = &1900).

&1900	&0D	carriage return at start of program
&1901	&00	MSB of line number
&1902	&0A	LSB of line number (10)
&1903	&07	length byte
&1904	&F1	'PRINT' token
&1905	&20	space character

&1906	&41	'A'
&1907	&0D	carriage return end of line marker
&1908	&FF	end of program marker

2.5 Executing statements

If the line input to the command handler did not start with a line number, it passes it on to the statement interpreter to decide what to do with it.

The statement interpreter is also used to RUN programs, as well as just interpreting statements and commands typed in command mode. The command handler has a special entry point to the statement interpreter, so that commands (like 'OLD') can only be executed in command mode, and not in the middle of a program.

The action of the statement interpreter is as follows:

1 It looks at the first character of the statement (skipping any spaces). If it is the token of a BASIC statement keyword (or a command keyword if we came from the command handler), then go to the corresponding statement handler (there is one of these for each statement or command) where the rest of that particular statement will be interpreted.

The *action address* of a particular token (the address to which the statement interpreter jumps when a token is found) is stored in the following format:

BASIC1	**BASIC2**	
&82CB+T	&82DF+T	LSB of action address
&833C+T	&8351+T	MSB of action address

where T is the number of the token (see table 2.1).

2 If the first character of the statement was not a statement keyword token, the statement interpreter checks to see if it is a variable name. If it is, it jumps to the assignment handler. This tries to assign the variable to the expression found after the '=' sign. If there wasn't an '=' after the variable name, it generates a 'Mistake' error (error number 4).

3 If the first character of the statement wasn't a variable name either, the statement interpreter checks to see if it is one of the other special symbols which can be at the start of a line. If it is a '*', it passes the rest of the line to the Operating System Command Line Interpreter (OSCLI) to be acted on. If it is a '[', it jumps into the assembler. If it is an '=', it jumps to the FN return statement handler (as this is the FN return statement).

4 If it wasn't any of those, it checks to see if the first character of the statement actually marks the *end* of the statement – in other words we have an empty statement. If it was, it goes back to stage 1 to interpret the next statement (or go to command mode if we have run out of statements to interpret). Most of the statement handlers jump to here when they have finished, to check that the text pointer is set up to point to the next statement.

5 Finally, if the character wasn't a *statement delimiter* either (a character marking the end of the statement), the statement interpreter gives up, and generates a 'Syntax error' (error number 16).

3 Memory Use

Fig 3.1 shows the memory map as seen by BASIC. The memory that BASIC uses can be split up into 3 major areas: workspace, program storage, and *dynamic storage* (the HEAP and STACK).

The workspace includes most of the general memory used by statements and functions. This is described in more detail in section 3.3.

Program storage has already been described in section 2.4.

Dynamic storage is allocated while a program is actually running; whereas workspace and the program occupy fixed areas while this is going on. Dynamic storage includes the storage of variables on the HEAP, and the use of the STACK for storing temporary results, and saving things during FN or PROC calls. The HEAP and STACK are described in more detail in the next sections.

3.1 Variables and the HEAP

3.1.1 The resident integer variables

The resident integer variables, @% and A% to Z%, are not stored on the HEAP where the rest of the variables are: they occupy the lower half of page 4. Because each one occupies a fixed location, they are very fast to access. They are stored in the following format:

&400 to &403	@%
&404 to &407	A%
etc.	
&468 to &46B	Z%

They are stored in standard 4-byte integer format (i.e. LSB first, MSB last). Here is a short program to list the resident integer variables, and their values (in HEX).

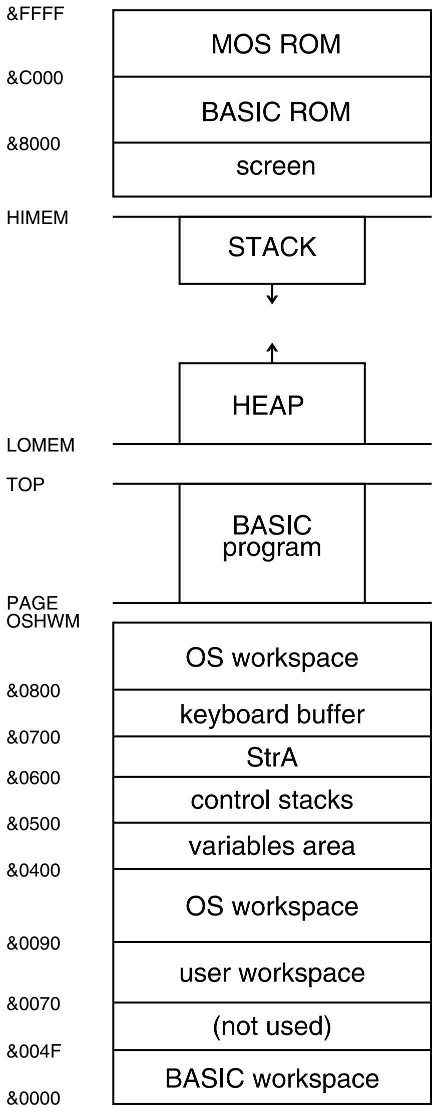


Figure 3.1 – The BASIC memory map.

```
  5 REM Prints out the resident integer variables
 10
 90 vbase = &400
100 FOR char = ASC"@" TO ASC"Z"
110   offset = (char AND &1F)*4
120   value% = vbase!offset
130   PRINT CHR$(char);"% = &";~value%
140   NEXT char
```

3.1.2 Dynamic variables

The rest of the variables used by BASIC are *dynamic* variables, because it allocates space for them when it needs it (i.e. when they are first set). These are stored on the HEAP, which works upwards in memory from LOMEM. To get at the variables once it has put them on the HEAP, BASIC uses a series of *linked lists*.

A linked list starts with a base pointer, which points to the first item in the list. The first item in the list has a pointer which points to the second item in the list, and so on. The end of the list is usually marked by the pointer to the next item being 0. So, if the linked list doesn't contain any items, the base pointer is 0 (a null pointer). Fig 3.2 shows a linked list of three items.

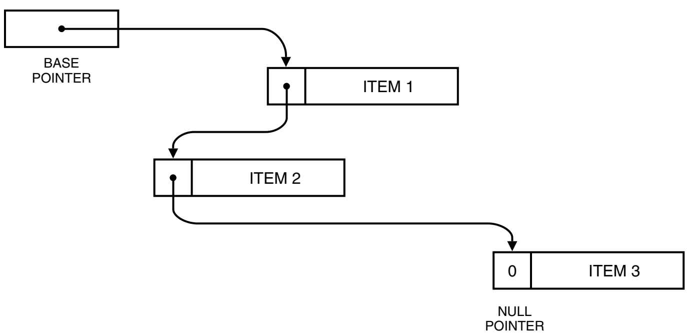


Figure 3.2 – A linked list.

One of the advantages of a linked list is that the items don't need to be in any set pattern in memory, as long as the pointers still point to the next item in the list. This can be very useful for variable storage, as different types of variables occupy a different number of bytes (especially arrays).

In fact, BASIC uses a separate linked list for each possible first letter of a variable name. Although these linked lists are separate, they all use the HEAP in the same way, and the lists link round each other. Using these separate linked lists means that searching for variables is much faster (unless your variable names all start with the same letter!).

The base pointers, which point to the first variable in each particular list, are stored in the upper half of page 4 in the following format:

&482,&483	base pointer for the 'A' list
etc.	
&4B4,&4B5	base pointer for the 'Z' list
etc.	
&4F4,&4F5	base pointer for the 'z' list

A similar linked list is used to store the locations of PROCs and FNs, once they have been called, so that BASIC doesn't have to search through the whole program to find them again. The base pointers for these are:

&4F6,&4F7	base pointer for the PROC list
&4F8,&4F9	base pointer for the FN list

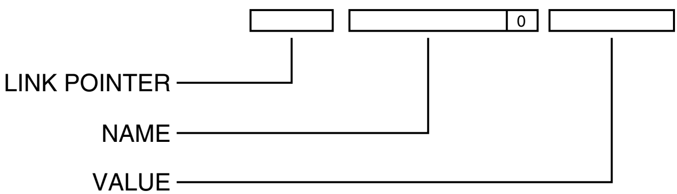


Figure 3.3 – A variable information block.

Each variable (or PROC/FN) on the HEAP is stored as a *Variable Information Block* (fig 3.3). This Variable Information Block is composed of 3 *fields:*

The pointer field (2 bytes).

> This is the pointer which points to the next item in the list (with the same first letter). If this item is at the end of the list, then the MSB of this pointer must be zero (the next item can't be in page zero, so only checking that the MSB is zero saves time).

The name field.

> This holds the name of the variable, with a zero byte to mark the end of the name. For a variable, this name field does not include the first character of the name, because that was used to choose which base pointer to use. It does contain the '$', '%' or '(' characters on the end of the name (if there are any), as this gives the type of the variable.
>
> For a PROC or FN, the first character of the name is included, as there is only one list for all PROCs, and one for all FNs.

The value field.

> This starts with the first byte after the zero byte at the end of the name field. For a variable, the format of this field depends on the type: these are detailed in section 3.1.3.
>
> For a PROC or FN, this field contains a 2-byte pointer to the PROC or FN where it is defined. It points to the first character after the name of the PROC or FN (i.e. to the '(' character if it uses any parameters).

As an illustration of the way variables are stored on the HEAP, the program below will go through the current active variables, printing their names and values. It can be used to print out variables other than those used by the program itself, by setting them up first, and using 'GOTO 90' to start the program (if 'RUN' is used, all variables are cleared first).

The program follows the linked list for each initial letter of variable names, using the variable 'addr' to hold the current pointer.

PROCvar prints out the name and value of the variable whose *Variable Information Block* (VIB) is at 'addr'. The last character of the variable gives its type, and this is used to prevent the program from printing out arrays. To print out the value of the variable, it 'cheats' by giving the name of the variable to EVAL rather than extracting it directly. Section 7.4 gives a machine code version of this routine.

```
   5 REM ****** VRPRINT ******
  10 REM Prints out variables used by the program.
  15 REM If any others are to be printed, use
  20 REM "GOTO 90" so they won't be cleared.
  30
  90 @%=0
 100 PRINT'"Variable"TAB(15)"Value"'
 110 FOR char = ASC("A") TO ASC("z")
 120   addr = &400+2*char    :REM Get pointer address
 130   addr = !addr AND &FFFF
 131                         :REM Get ptr to 1st VIB
 140   IF (addr DIV &100)=0 THEN GOTO190
 141                         :REM Exit if null pointer
 150   REPEAT
 160     PROCvar                 :REM Print variable
 170     addr = !addr AND &FFFF  :REM Get ptr to next VIB
 180     UNTIL (addr DIV &100)=0 :REM Exit if null pointer
 190   NEXT char
 200 END
 990
 998
 999 REM *** Print variable name and value ***
1000 DEFPROCvar
1010 name$ = CHR$(char)      :REM First character of name
1020 nptr = 2                :REM Ptr to name in VIB
1030 IF addr?nptr=0 THEN GOTO1100
1031                         :REM End of name?
1040 REPEAT
1050   name$ = name$+CHR$(addr?nptr)
1051                         :REM Add next char to name
1060   nptr = nptr+1
1070   UNTIL addr?nptr=0     :REM Exit if end of name
1100 PRINT name$,TAB(15);
1105 typ$ = RIGHT$(name$,1)  :REM Get type of variable
1110 IF typ$="(" THENPRINT"<array>" ELSEPRINT EVAL(name$)
1111                         :REM Print value if not array
1130 ENDPROC
```

3.1.3 Variable value formats

When writing programs in BASIC, variables can be one of 3 types: 4-byte integers, floating point numbers, or strings (these are called *dynamic* strings, as BASIC allocates memory for them as it is required). However, the indirection operators (‘?’, ‘!’ and ‘$’) can be used to manipulate 8-bit bytes, 4-byte integers, and *static* strings (i.e. strings at a fixed address in memory).

Once BASIC has found the location of the variable, these bytes and static strings are treated like just like two more variable types (4-byte indirected integers are stored the same as named 4-byte integer variables). To pass variables between routines, a *Variable Descriptor Block* (not to be confused with the Variable Information Block) is used, which is usually left in IntA (the integer accumulator). The format of this is:

&2A,&2B	pointer to the location of the variable value
&2C	type of the variable

This *Variable Descriptor Block* is used, for example, in the *Parameter Block* passed by the BASIC ‘CALL’ statement (when any parameters are passed to it). This means that a user routine can read or set any of the variables passed as parameters to the CALL statement.

The format of the different variable types are:

Type number &00: 8-bit byte

Format:

00	8-bit byte	1 byte

This is just a single byte at the specified location. This type of variable can only be accessed by using the ‘?’ operator; either as ‘?M’ to mean ‘the byte pointed to by M’, or as ‘M?3’ to mean ‘the byte at location M+3’.

Type number &04: 32-bit integer

Format:

00	32-bit integer	4 bytes

This is a 4-byte integer at the specified location. It is stored LSB first, MSB last. This type of variable can be accessed as a named integer variable, like ‘A%’ or ‘integer%’, or by using the ‘!’ operator.

If a named variable is used, the location of the value has to be found first, either by looking it up in the table of resident integer variables, or by searching through one of the linked lists for it. The *name field* of the Variable Information Block in the linked list has the ‘%’ on the end of it, so that it is identifiable as an integer.

If the ‘!’ operator is used, the location of the variable is taken as the number following the ‘!’ (for the unary version); or the sum of the variable before the ‘!’, and the number after it (for the binary version).

Type number &05: 40-bit floating point number

Format:

00	exponent (offset &80)	1 byte
01	mantissa	4 bytes
(bit 7 of byte 01 holds the sign bit)		

This is a floating point number at the specified location. The mantissa is stored MSB first, LSB last (the opposite order to 4-byte integers). The top bit of the mantissa is used to hold the sign bit, as this would always be a ‘1’ (see section 2.2.2 for a description of floating point numbers).

This type of variable can only be accessed as a named variable stored on the HEAP; there is no floating point indirection operator. The location of the variable is found by searching through one of the linked lists for it. There is no symbol on the end of the *name field* of a floating point variable.

Type number &80: static string

Format:

00	ASCII characters of string	nn bytes
nn	&0D terminating character	1 byte

This is a static string at the specified location. It can only be accessed by using the '$' string indirection operator: the location of the string is taken to be the number after the '$'. The carriage return (&0D) terminating character is not counted as one of the characters of the string: it is only used to mark the end.

Space can be allocated for a string of this type, by using the 'reserve space' form of the DIM statement: 'DIM A 20' will allocate space for a string at A of maximum size 20 characters, plus 1 for the terminator.

Type number &81: dynamic string

Format:

00	pointer to string on HEAP	2 bytes
02	space allocated	1 byte
03	current length	1 byte

This is the *String Information Block* of the dynamic string: these 4 bytes will occupy the value field of the Variable Information Block of a string variable. This type of variable can only be accessed as a named variable. The *name field* of the Variable Information Block has the '$' symbol on the end, so it is identifiable as a string.

When a dynamic string is first assigned, the Variable Information Block is created and linked into one of the lists, to hold the name and String Information Block of the string. Then space is allocated on the HEAP for the characters of the string itself, and the String Information Block is set up to point to first character of that string. The string itself does not need a carriage return to mark the end, as the String Information Block holds the length of it.

If the string is empty, no space needs to be allocated for it at all. If the string is a ‘small’ string (less than 8 characters), just the correct number of bytes is allocated on the HEAP for it. If it is a ‘large’ string, an extra 8 bytes are reserved for it, to allow some room for expansion (if this would take the allocated space over 255 characters, 255 bytes are reserved).

Whenever a dynamic string exceeds the space which has been allocated, a new area is reserved for it on the HEAP (using the same rules as above). The ‘gap’ left in the HEAP where the string used to be cannot be recovered (BBC BASIC has no ‘garbage collector’): so if memory is not to be wasted, it is usually a good idea to set strings, at the start of a program, to the largest size that they are likely to become.

The amount of memory wasted in this manner is not usually a great deal, but certain operations tend to use quite a lot (for example, a loop which adds one character on the end of a string each time round). In BASIC2 this has been improved by checking to see if the string is on top of the HEAP: if it is, it can be extended without having to throw away the old area.

3.1.4 Array storage

Arrays are stored in the same kind of Variable Information Block as ordinary variables, but the *value field* of an array is usually much bigger than that of an ordinary variable. The *value field* of an array has to hold the number of dimensions, and the size of each dimension, as well as the the value of each cell in the array.

The Variable Information Block for an array is linked into the list when it is dimensioned: any attempt to read from or write to an array which does not exist will result in the ‘Array’ error (error number 14) being generated.

The *name field* in the Variable Information Block for an array has the ‘(’ symbol on the end, so that it is identifiable as an array. It also has the ‘%’ or ‘$’ symbol before that, if it is an integer array or a string array.

The format of the *value field* of an array with D dimensions is:

00	offset of start of cells (nn)	1 byte
01	size of dimension 1	2 bytes
03	size of dimension 2	2 bytes
05	etc.	
nn−2	size of dimension D	2 bytes
nn	start of cells	

The first byte of the *value field* gives the offset of the start of the cells from the start of the *value field*, rather than the number of dimensions of the array. If the number of dimensions is D, this offset will be 2*D+1 bytes (2 for the size of each dimension, and 1 for the offset byte itself). This will be 3 for single-dimension arrays.

The size of each dimension is stored as the maximum allowed subscript.

Each cell is in the same format as the equivalent variable: if it is an integer array, each cell will contain a 32-bit integer (type number &04); if it is a floating point array, each cell will contain a 40-bit floating point number (type number &05); and if it is a string array, each cell will contain a 4-byte *String Information Block* (type number &81). The actual strings for a string array are stored separately on the HEAP (as for dynamic string variables), as soon as they are first set.

The order of the cells is probably best explained by an example. For the array A(1,1,1) the order of the cells will be:

cell 0	A(0,0,0)
cell 1	A(0,0,1)
cell 2	A(0,1,0)
cell 3	A(0,1,1)
cell 4	A(1,0,0)
cell 5	A(1,0,1)
cell 6	A(1,1,0)
cell 7	A(1,1,1)

The following algorithm can be used to find the required element of an array:

```
C = 0
start at first dimension
REPEAT
        C = (C * size) + subscript
        move on to next dimension
UNTIL no more dimensions left
```

where ‘size’ is one more than the maximum subscript for the dimension of interest (allowing for the subscript 0); and ‘subscript’ is the required subscript of the dimension of interest.

At the end of that algorithm, C will give the cell number of the required element.

Taking the example of the array A(1,1,1) again, if the element required was A(1,1,0), the successive values of C after each iteration of the loop in the algorithm would be:

after 1 pass:	C = 1
after 2 passes:	C = 3
after 3 passes:	C = 6

This means that the element A(1,1,0) is cell number 6 of the array A(1,1,1). This agrees with the list given above.

To get the location of the cell, the cell number must be multiplied by the size of each cell: 4 bytes for an integer or a string, or 5 bytes for a floating point number. This gives the offset (in bytes) of the required cell from the start of the cells.

Once the location of the element has been found, this can be put in the *Variable Descriptor Block*, together with the type of the element (integer, floating point, or string). The array element can now be handled inside BASIC as if it was just another variable in memory.

3.2 The BASIC STACK

The BASIC STACK works downwards from HIMEM. The STACK pointer is held in page zero, at &4,&5. It is used to save temporary results in the middle of calculations, and to save the 6502 stack and parameters when a FN or PROC is called (see section 5.3).

For example, to evaluate the expression:

2 + 5 * 3

the ‘2’ must be saved while the ‘5 * 3’ is being calculated. The 6502 stack *could* be used for this, but it is very small, and would not allow very complex expressions without overflowing (especially when there are FNs to be dealt with).

Before anything is pushed on the STACK, a check is made to ensure that there is enough room for the new item: otherwise there may be a clash with the HEAP which is growing in the opposite direction, upwards from LOMEM (see fig 3.1). If there is not enough room, the ‘No room’ error is generated.

There are routines to push any of BASIC’s accumulators IntA, FPA, and StrA (and pull them again); these are used quite a lot in the expression evaluator. Chapter 4 describes the expression evaluator in more detail.

The other main use of the BASIC STACK is by PROCs and FNs. When one of these is entered, the 6502 stack is transferred onto the BASIC STACK. If this was not done, the small 6502 stack would soon overflow with return addresses for JSRs if the *recursion* of the PROCs or FNs went very deep (i.e. the PROC or FN called itself).

PROCs and FNs also need to make sure that LOCAL variables and parameters used in the PROC or FN are returned to their original values when the call is finished. When the call is started, the values of the parameters in the PROC or FN definition are pushed on the STACK, together with the *Variable Descriptor Block* for the parameter. That gives the location and type of the variable, so it can be restored after the call. Section 5.3 gives more detail on the action of PROCs and FNs.

3.3 Workspace

This section lists the workspace used by BASIC. In many cases, the use of particular locations may be described in more detail elsewhere.

Page Zero

&00 – &01	LOMEM
&02 – &03	HEAP pointer (section 3.1)
&04 – &05	STACK pointer (section 3.2)
&06 – &07	HIMEM
&08 – &09	ERL
&0A	PTRA offset
&0B – &0C	PTRA base (section 2.2.5)
&0D – &11	psuedo-random number for RND
&12 – &13	TOP
&14	PRINT field width
&15	PRINT hex flag (HEX if bit 7 set)
&16 – &17	ON ERROR pointer (section 5.8, chapter 11)
&18	MSB of PAGE (LSB is always zero)
&19 – &1A	PTRB base
&1B	PTRB offset (section 2.2.5)
&1C – &1D	DATA pointer (points before next DATA item)
&1E	COUNT (no of characters printed on line)
&1F	LISTO mask: bit 0: space after line no. bit 1: indent FORs bit 2: indent REPEATs
&20	TRACE flag (&00 = OFF, &FF = ON)
&21 – &22	TRACE maximum line number

&23	WIDTH (or &FF if WIDTH 0 used)
&24	REPEAT stack pointer (section 5.5)
&25	GOSUB stack pointer (section 5.2)
&26	FOR stack pointer (section 5.6)
&27	Temp for expression evaluator
&28	OPT mask: bit 0: produce listing bit 1: give errors bit 2: relocate (BASIC2)
&29	opcode slot for assembler
&2A – &2D	IntA (section 2.2.1)
&2E – &35	FPA (section 2.2.2)
&36	StrA length (characters from &600 on)

Page Zero multi-purpose workspace

&37 – &4E	Main uses are:
&37 – &38	general pointer
&39	name length/variable type
&39 – &40	integer for division and multiplication
&3B – &42	FPB for floating point routines
&43 – &46	floating point multiply/divide workspace
&3F – &47	PRINT hex digit build area
&48	no. of constants for series evaluator
&49	flag for string/number conversion
&4A	exponent for string/number conversion
&4B – &4C	floating point memory pointer
&4D – &4E	pointer for series evaluator
&4F – &8F	(not used)

OS workspace

&90 – &3FF	OS workspace

Page 4 workspace

&400 – &46B	resident integer variables (section 3.1.1)
&46C – &470	floating point temp 1
&471 – &475	floating point temp 2
&476 – &47A	floating point temp 3
&47B – &47F	floating point temp 4
&480 – &4F5	variable list base pointers (section 3.1.2)
&4F6 – &4F7	PROC list base pointer (section 3.1.2)
&4F8 – &4F9	FN list base pointer (section 3.1.2)
&4FA – &4FF	(not used)

Page 5 workspace

&500 – &595	FOR stack (section 5.6)
&596 – &5A3	(not used)
&5A4 – &5CB	REPEAT stack (section 5.5)
&5CC – &5FF	GOSUB stack (section 5.2)

Page 6 workspace

&600 – &6FF	characters of StrA (section 2.2.3)

Page 7 workspace

&700 – &7FF	keyboard input buffer

4 Expression Evaluation

One of the major sections of the BASIC interpreter is the expression evaluator. Virtually every statement uses it to get the number or numbers that it is going to work with. For example the ‘HIMEM’ statement uses it to find the new value that HIMEM is to be set to.

4.1 Operator precedence

When expressions are to be evaluated, some operators take precedence over others. For example, multiplication is always done before addition, unless the addition is surrounded by brackets. This makes expression evaluation somewhat more complex than it would otherwise be, as you can’t just scan along the line, doing every operation as you come across it.

In fact, many old electronic calculators *did* just scan along the line like this. If you pressed:

```
2 + 3 * 5 =
```

you would get the answer ‘25’. This is not particularly satisfactory for an expression evaluator in BASIC, because if ‘2 + 3 * 5’ appears as an expression, it is assumed that the multiplication will be done first, giving the answer ‘17’. Somehow, BASIC must identify that the addition must be done after the multiplication, save the ‘2’ while the ‘3’ and ‘5’ are being multiplied together, and then add the ‘2’ on afterwards.

4.2 Top-down analysis

To get these operator priorities right, BASIC uses a method called *top-down analysis*, where the expression evaluation is divided up into several levels. The top levels deal with the low priority operators, and these call the bottom levels (which deal with the high priority operators) for the items to operate on. This means that the high priority operations will be performed first, by the bottom levels of the expression evaluator, before the results of those operations are passed back to the top levels, for the low priority operations to be performed.

Taking the example of ‘2 + 3 * 5’ again, the top level would deal with the addition, and call the bottom level to get the values for it to add. The bottom level would deal with the multiplication, before passing the result back to the top level.

If we call the top level <expression>, and the bottom level <term>, we can see how this would operate:

1 <expression> calls <term> to get the first item to operate on.

2 <term> gets the number ‘2’ from the line.

3 There is not a ‘*’ or a ‘/’ after the ‘2’, so <term> passes ‘2’ up to <expression>.

4 <expression> finds that there is a ‘+’ after the item that <term> had evaluated, so it saves the ‘2’ and calls <term> again to get the item to add to it.

5 <term> gets the number ‘3’ from the line.

6 There is a ‘*’ following the ‘3’, so <term> saves the ‘3’ and gets the number ‘5’ from the line.

7 The ‘5’ is multiplied by the saved ‘3’, to give the result ‘15’.

8 There is not a ‘*’ or a ‘/’ after the last number just read (the ‘5’), so <term> passes the ‘15’ up to <expression>.

9 <expression> retrieves the ‘2’ that it had saved at stage 4, and adds it to the ‘15’ passed up from <term>, giving the result ‘17’.

10 There is not a ‘+’ or a ‘−’ after the item that <term> had evaluated (the ‘3*5’), so it passes the ‘17’ up as the result of the <expression>.

The levels in this simple expression evaluator can be expressed using *Backus-Naur Form*, or BNF (see appendix A). It is expressed as follows:

```
<expression> ::= <term> {+|- <term>}
<term> ::= <number> {*|/ <number>}
```

::= means ‘is defined as’

{} surround items which can appear zero or more times

| separates alternatives

So an <expression> can consist of just a <term> or any number of <term>s with each one separated by a ‘+’ or a ‘–’. Similarly a <term> can be just a <number>, or it can be any number of <number>s with each one separated by a ‘*’ or a ‘/’.

In the example ‘2 + 3 * 5’:

the <expression> is ‘2 + 3 * 5’

the first <term> is ‘2’
the second <term> is ‘3 * 5’

The BASIC program below shows a simple expression evaluator with the <expression>, <term>, and <number> levels.

FNexpr evaluates an <expression>, calling **FNterm** to get the <term>, and **FNnumber** is used to get the <number>. Spaces are not allowed in expressions evaluated by this program.

The program uses *one character look-ahead*, where the next character is always kept in the variable ‘char$’. This allows the character not recognised by **FNterm**, say, to be passed to **FNexpr** in case it was a ‘+’ or a ‘–’. If this were not done, <expression> would have to re-read the character from the line, before testing it for one of its operators. If a character is recognised, the next one must be read into char$ before another routine is called (for example, on line 1030).

```
  5 REM Simple expression evaluator to demonstrate the
 10 REM "top-down" method of expression analysis
 15 REM (spaces not allowed in expressions)
 20 REM
 90 REM *** Main loop ***
100 REPEAT
110   INPUT"EXPRESSION :"line$
120   lptr = 1
130   PRINT"VALUE IS  :";FNexpr
140   UNTILFALSE
```

```
 990
 1000 DEF FNexpr       :REM Get <expression> from line
 1005 PROCgetchar      :REM Get char into char$
 1010 value = FNterm   :REM Call <term> to get first item
 1015 REPEAT
 1030   IF char$="+" THEN PROCgetchar:value = value+FNterm
 1040   IF char$="-" THEN PROCgetchar:value = value-FNterm
 1045   UNTIL char$<>"+" AND char$<>"-"
 1050 = value          :REM Final result
 1990
 2000 DEF FNterm       :REM Get <term> from line
 2010 value = FNnumber :REM Call <number> to get first item
 2025 REPEAT
 2030   IF char$="*" THEN PROCgetchar:value =value*FNnumber
 2040   IF char$="/" THEN PROCgetchar:value =value/FNnumber
 2042   UNTIL char$<>"*" AND char$<>"/"
 2050 = value          :REM Result of <term>
 2990
 3000 DEF FNnumber     :REM Read in <number> from line
 3020 IF char$>"9" OR char$<"0" PRINT "NO NUMBER":STOP
 3035 number = 0
 3040 REPEAT
 3050   digit = ASC(char$)-&30
 3060   number = number*10 + digit
 3070   PROCgetchar
 3090   UNTIL char$>"9" OR char$<"0"
 3100 = number         :REM Value of <number>
 3990
 4000 DEF PROCgetchar  :REM Get character from line
 4030 char$ = MID$(line$,lptr,1)
 4040 lptr = lptr+1
 4060 ENDPROC
```

The expression evaluator in BASIC has eight levels, rather than just the 2 in the simple model. The levels, and their associated operators, are as follows (lowest priority at the top):

Level	Operators
<testable-condition>	OR, EOR
<logical-expression>	AND
<relnl-expression>	=, <, <=, <>, >, >=
<expression>	+, -
<term>	*, /, MOD, DIV
<sub-term>	^
<factor>	+, - (unary operators)
<primitive>	

Note that `<testable-condition>` is the same as `<numeric>` (see chapter 33 of the BBC *User Guide*, or chapter 25 of the *Electron User Guide*). Numbers, functions and variables appear at the `<primitive>` level. A `<primitive>` could also be a `<testable-condition>` in brackets, causing the expression evaluator to *recurse* down from the top level again. For a more complete definition of the expression evaluator, and the rest of BASIC, see appendix A.

Most functions enter the expression evaluator at the `<factor>` level rather than at the top; this means that variables or numbers can be given to a function without brackets, but an `<expression>` must be included in (round) brackets. So, for example, the expression 'SIN2+5' will be evaluated as '(SIN2)+5'.

When finished, each level of the expression evaluator leaves its result in IntA, FPA, or StrA (depending on the type), with the type in the 6502 accumulator. The type bytes are:

&00	real (floating point) number
&40	integer
&FF	string

Note that these are not the same as the variable types described in section 3.1.

Each level can check this type byte returned to it by a lower level, and do any conversions necessary (or generate an error if a type mismatch has occurred). The particular ROM routines in section 10.4 give more details of the use of these type numbers.

No check is made to see if the expression evaluator is running out of 6502 stack (due to all the subroutines it is calling). This means, for example, that if more that 17 levels of nested brackets are used, the stack will overflow, and the expression will not be evaluated properly (it may even generate an obscure error). In practice, this number of brackets is hardly ever used, so the problem never arises.

5 Program Control Mechanisms

Normally in a BASIC program, the statements are executed one after the other, working through the program. However, several statements are provided which allow this normal flow of control of the program to be changed, either by jumping to another part of the program, or by conditionally executing a series of statements.

BASIC keeps a text pointer, PTRA, which it uses to point to the statement currently being executed, in a similar way to the program counter (PC) in the 6502 (see section 2.2.5). Whenever any of these program control statements, like GOTO, change the flow of control of the program, this pointer is changed to point to the start of the new statement where execution of the program is to continue. When the interpreter continues, it will then start reading in from the statement pointed to by PTRA.

This section details the program control statements in BASIC, and describes the mechanisms that they use to operate.

5.1 GOTO

This is the simplest of the program control statements in BASIC. It just passes control from one part of the program to another.

The action of the BASIC GOTO statement is:

1 Get the line number or <numeric> following the GOTO token.

2 Search the program from the beginning to find a line with that line number; if it is not found, generate a 'No such line' error (error number 41).

3 If the line was found, then point the text pointer PTRA at the start of the first statement on that line. When the BASIC interpreter continues, it will execute statements from there onwards.

5.2 GOSUB…RETURN

The GOSUB statement is similar to the GOTO statement in that it passes control to another part of the program; but it also allows control to RETURN to the statement after the GOSUB statement when the subroutine has finished.

The GOSUB statement has to remember where to RETURN to after the end of the subroutine. A ‘GOSUB stack’ is used to hold the location of the statement following the GOSUB statement, so that the RETURN statement on the end of the subroutine can pass control back to that part of the program. The format of the GOSUB stack is:

&05CC+GSP	LSB of return address
&05E6+GSP	MSB of return address
&25	GOSUB stack pointer (GSP)

The action of the GOSUB statement is:

1 Get the line number or <numeric> following the GOSUB token, and set PTRA to point to the end of the statement.

2 Search the program to find a line with that line number; if it is not found, generate a ‘No such line’ error (error number 41).

3 If the GOSUB stack pointer is more than 25, there are already 26 return addresses (0 to 25) on the stack. In this case, generate a ‘Too many GOSUBs’ error (error number 37), to prevent the GOSUB stack from overflowing (it only has room for 26 entries).

4 If we get here, the GOSUB stack is not full, so push the base of PTRA, which now points to the end of the GOSUB statement, on to the the GOSUB stack. Increment the GOSUB stack pointer (GSP), ready for the next one.

5 Point the text pointer PTRA at the start of the first statement on the line found. When the BASIC interpreter continues, it will execute statements from there onwards.

When a RETURN statement is encountered, it has to retrieve the old value of PTRA, so that it can go back to the statement after the GOSUB which called it.

The action of the RETURN statement is:

1 If the GOSUB stack pointer is 0, the GOSUB stack is empty, and there is no address to return to. In this case, generate the 'No GOSUB' error (error number 38).

2 Pop the return address from the GOSUB stack, decrementing the GOSUB stack pointer to remove it. This return address is then put into PTRA. When the interpreter continues, it will execute statements from there onwards (i.e. starting with the statement after the GOSUB which called the subroutine).

5.3 PROCs and FNs

The ability to call PROCs and FNs is a very powerful feature of BBC BASIC, although as far as the interpreter is concerned it is just a more complex version of the GOSUB statement. With PROC and FN calls, not only does the return address have to be saved, so that control can be returned when the call is finished, but the values of parameters and local variables have to be saved so that they can be restored also.

Once a FN or PROC has been called, its name and location is added to a linked list on the BASIC HEAP, one list for FNs, and one for PROCs. This means that once a FN or PROC has been used, BASIC does not have to search through the whole of the program to find it again (like it does with the line numbers given to a GOTO or GOSUB statement). See section 3.1 for the format of these liked lists.

After the FN or PROC has been found, any parameters which need to be passed are handled. In the description below, *formal parameter* refers to the parameter used in the FN or PROC definition; and *actual parameter* refers to the parameter which is passed to it.

Although PROC is a statement and FN is a function (and hence returns a value), the mechanism which is used when they are called is very similar. To deal with both of them, there is a standard FN/PROC handler which is called by both the FN function and the PROC statement.

The PROC statement has to copy PTRA into PTRB before calling this handler, and then use PTRB (rather than PTRA) to check that it is at the end of the statement when the call has returned. The FN/PROC handler must not alter PTRA, because this is not used in the expression evaluator (and hence the FN function must not change it). The FN function does not need to do any of this (as PTRB will be set up correctly for it), and the FN/PROC handler returns directly to the code which called the FN when it has finished.

The action of the FN/PROC handler is:

1 Save the contents of the 6502 stack on the BASIC stack (with a byte to give the old 6502 stack pointer), and reset the 6502 stack pointer to &1FF. The 6502 stack works downwards in page 1, and the stack pointer points to the next available byte, so it is now empty (fig 5.1 (b)). The 6502 stack is not very big – only 256 bytes – and saving it in this manner allows deep *recursion* of FNs and PROCs without overflowing the small 6502 stack.

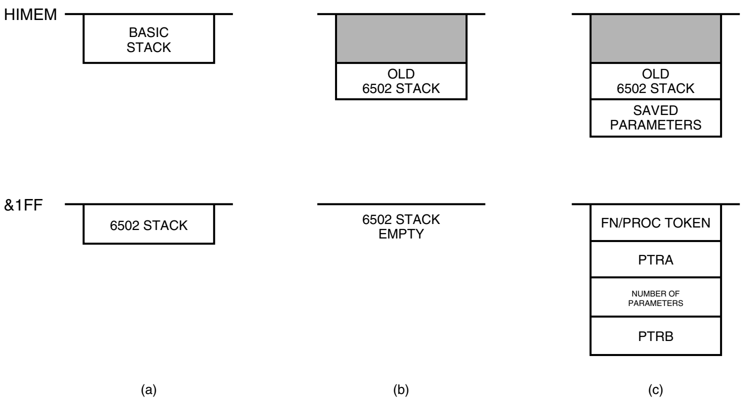


Figure 5.1 – FN/PROC stack use.

2 Save the FN or PROC token as the first item on the 6502 stack, at &1FF. The FN token is &A4, and the PROC token is &F2. This allows the ENDPROC or FN return statement (‘=’) to check that it is inside the correct type of call before it exits.

3 Save PTRA on the 6502 stack.

4 Scan the name of the FN/PROC call. If there is not one immediately following the FN or PROC token, generate a ‘Bad call’ error (error number 30).

5 Search for the name of the FN or PROC in the list of already used calls. If it is found, don’t bother to look through the program for it.

6 If the FN or PROC was not in the list, look through the program from the beginning until a DEF FN or a DEF PROC is found with the correct type and name. This search uses PTRA to look through the program (which is why it was saved at stage 3). If it is found, add it to the list; otherwise, restore the base of PTRA from the 6502 stack (this will tell the error handler on which line the error occurred), and generate a ‘No such FN/PROC’ error.

7 Set PTRA to point to the location found by the search (or found in the list). This will point to the first character following the name after the DEF FN or DEF PROC. If there are any parameters, this character will be an opening bracket, ‘(’.

8 If there are any parameters in the definition, check that they match with those in the call. If they do, push the value and the *variable descriptor block* of each *formal* parameter on the BASIC STACK (i.e. the one in the definition), and assign the new value to it given by the value of the *actual* parameter in the call. Saving the value and *variable descriptor block* allows the formal parameters to be restored to their original values after the call has returned. If the parameters do not match, restore the base of PTRA from the 6502 stack (for the error handler), and generate an ‘Arguments’ error (error number 31).

9 Push the number of parameters on the 6502 stack, so that the correct number can be restored when returning from the call. If there were no parameters, this will be 0.

10 Save PTRB on the 6502 stack. This points to the next part of the line to be interpreted, and will need to be restored after the call has returned. The stacks are now in the state shown in fig 5.1(c).

11 Start off the call by executing a JSR to the statement interpreter, which will start executing statements from PTRA. This leaves this return address on the 6502 stack ready for a FN return statement or an ENDPROC statement (all other statements JMP back to the statement interpreter when they have finished; only the ENDPROC and FN return statements finish by executing an RTS).

12 When we get here, the FN or PROC has finished. If it was a FN, then the result type will be in &27, and the value will be in IntA, StrA, or FPA as appropriate.

13 Restore PTRB from the 6502 stack. This points to the place in the line where interpreting should continue.

14 Pull the number of parameters from the 6502 stack. If there were any, restore the old value of each one by pulling its *variable descriptor block* and value from the BASIC STACK.

15 Restore PTRA from the 6502 stack. The only thing left now on the stack, is the FN or PROC token, which was used to tell the ENDPROC or FN return statement which type of call it was in.

16 Recover the old 6502 stack from the BASIC stack. The stacks are now back to the state that they were when the FN/PROC handler was called (fig 5.1(a)).

17 Retrieve the type of the result from &27 into A, in case this is a FN. If it is a PROC, this stage is not needed, but does no harm.

18 Execute an RTS to return to the code which called the FN/PROC caller. In the case of a FN, this returns to the expression evaluator, with the type of the result of the FN in A, and the result itself in IntA, FPA, or StrA. In the case of a PROC, this returns to the PROC statement handler, which sets PTRA to point to the next statement (using PTRB to find out where the FN/PROC handler had got up to), and jumps back to the statement interpreter to continue execution after the PROC.

By trapping the ‘No such FN/PROC’ error generated if the DEF FN or DEF PROC is not found in stage 6 above, procedures and functions can be overlayed from disc (or tape, but it’s not so useful). There is more on overlaying FNs and PROCs in chapter 8.

The LOCAL statement inside a FN or PROC has to save the old value of variables in a similar way to parameters passed to the call. Each variable in the LOCAL statement has its value pushed on the BASIC STACK, followed by its *variable descriptor block*; and the ‘Number of parameters’ byte on the 6502 stack is incremented. The current value of the variable is then set to zero. Saving it in this manner means that its old value will be restored as if it was just another parameter, when the call returns.

The ENDPROC statement and the ‘=’ (FN return) statement check the state of the stack before they return (just returning could have disastrous results if they didn’t). If they find that there are not at least 4 items on the 6502 stack (there won’t be any if it isn’t in a PROC or a FN), they generate a ‘No FN’ or ‘No PROC’ error. Also, if the token at &1FF (the bottom of the stack) does not match (i.e. a PROC token for ENDPROC, or a FN token for the FN return statement), this error is also generated. Otherwise, if everything is OK, then they execute an RTS (after evaluating the <numeric> in the case of the FN return statement) to return to the FN/PROC handler at stage 12 above.

When executing statements inside a FN or PROC, the 6502 S register contains &F5 (i.e. the next available byte on the stack is at &1F5), and the state of the stack is as follows:

&1F6	RTS addr for FN/PROC handler	2 bytes
&1F8	PTRB base MSB	1 byte
&1F9	PTRB base LSB	1 byte
&1FA	PTRB offset	1 byte
&1FB	number of parameters	1 byte
&1FC	PTRA base MSB	1 byte
&1FD	PTRA base LSB	1 byte
&1FE	PTRA offset	1 byte
&1FF Bottom:	FN/PROC token (&A4/&F2)	1 byte

Note that when the FN/PROC handler gets back at stage 12, the RTS address has been removed from the top.

5.4 IF…THEN…ELSE

This construction allows the statements after the THEN or the ELSE parts to be executed conditionally, depending on the value of the <testable-condition> found after the IF part.

The action of the IF statement is:

1 Evaluate the <testable-condition> following the IF token (i.e. the <numeric> after the IF token: they are just the same).

2 If the <testable-condition> evaluated to be 0 (i.e. false), then scan through the line until an ELSE token or the end of the line is found. If no ELSE was found on the line, then continue execution on the next line. Otherwise, set PTRA to point to the character after the ELSE token, and continue at stage 4.

3 If the <testable-condition> evaluated to be anything other than 0 (i.e. true), check for a THEN token. If there isn't one, JMP to the statement interpreter to continue executing the rest of the line after the <numeric> (you don't have to use a THEN). If there is a THEN token, set PTRA to point to the character after it, and continue at stage 4.

4 Check for a (tokenised) line number following the THEN or ELSE; if there is one, execute a GOTO to that line number. Otherwise, JMP to the statement interpreter to continue executing the rest of the line.

Note that once the IF statement has decided that the THEN section is to be executed, the IF statement does not prevent it from ‘falling into’ the ELSE clause; this is done by the general statement interpreter itself. If it discovers that there is an ELSE token on the end of the statement it has just executed, it will just skip the rest of the line instead (as if it was a REM statement). This means that lines like:

```
PRINT "HELLO" ELSE MISTAKE
```

will not give an error, but the ELSE clause will never be executed.

5.5 REPEAT…UNTIL

This is the simplest of BASIC’s two loop structures, the other being the FOR…NEXT loop. Using this loop, control is repeatedly passed back to the statements following the REPEAT until the UNTIL clause is satisfied.

This loop structure uses a stack in page 5 to save the location of the start of the statement after the REPEAT, so that the UNTIL statement knows where to pass control back to if it is not satisfied. The format of the REPEAT stack is:

&5A4+RSP	LSB of repeat address
&5B8+RSP	MSB of repeat address
&24	REPEAT stack pointer (RSP)

The action of the REPEAT statement is:

1 Check that the REPEAT stack pointer (RSP) is less than 20 (&14). If it isn’t, the REPEAT stack is full, so generate a ‘Too many REPEATs’ error (error number 44).

2 PTRA points to the character after the REPEAT token, so push that address on the REPEAT stack, incrementing the REPEAT stack pointer.

3 JMP to the statement interpreter to continue execution with the statements after the REPEAT token.

The action of the UNTIL statement is:

1 Evaluate the <testable-condition> following the UNTIL token, checking that it is at the end of the statement (if it isn't at the end of the statement, a 'Syntax error' is generated).

2 Check that the REPEAT stack is not empty (i.e. the REPEAT stack pointer is not 0). If it is, generate a 'No REPEAT' error (error number 43).

3 If the <testable-expression> evaluated in stage 1 was zero, get the address of the statement following the REPEAT from the REPEAT stack, leaving it on there for the next time this UNTIL statement is encountered. Set PTRA to this address, and JMP to the statement interpreter to continue execution at the statement after the REPEAT.

4 If the <testable-expression> was not zero, remove the top entry from the REPEAT stack by decrementing the REPEAT stack pointer, and JMP to the statement interpreter to continue execution with the statements following the UNTIL statement.

5.6 FOR…NEXT

This loop structure allows a series of statements to be performed a set number of times, with a different value of the *control variable* each time. This is a more complex loop than the REPEAT…UNTIL loop, as far as the interpreter is concerned, because it takes more time to set up, and there is more to do every time it goes round the loop.

This loop has to save the address and type of the control variable, the STEP size, the TO limit, and the address of the statement after the FOR statement. For this, it has a stack in page 5 in the following format:

&500–50E	First 15-byte FOR entry
&50F–51F	etc.
&587–595	Tenth 15-byte FOR entry
&26	FOR stack pointer (FSP) (multiple of 15)

The FOR stack pointer is an offset from &500 to the next available 15-byte FOR slot. The format of each 15-byte entry is:

&00	Address of control variable	2 bytes
&02	Type of control variable	1 byte
&03	STEP size	5 bytes
&08	TO limit	5 bytes
&0D	Address after FOR statement	2 bytes

If the control variable is an integer, it only uses 4 of the 5 bytes allocated for the STEP size and TO limit.

The action of the FOR statement is:

1 Get the variable following the FOR token; this is going to be the 'control variable'. If it is invalid, or a string variable, generate a 'FOR variable' error (error number 34).

2 Check for an equals sign ('=') following the variable; if there isn't one, generate a 'Mistake' error (error number 4).

3 Evaluate the <numeric> after the equals sign, and set the value of the control variable to this.

4 If the FOR stack pointer is &96 (150) or more, there are already 10 FOR loops in operation and the FOR stack is full. If this is the case, generate a 'Too many FORs' error (error number 35).

5 Save the address and type of the variable (i.e. its *variable descriptor block*) on the FOR stack.

6 If the next character on the line is a TO token, evaluate the <numeric> after it (making sure it is the same type – real or integer – as the control variable), and save that on the

FOR stack. If it isn't a TO token, generate a 'No TO' error (error number 36).

7 If the next character is a STEP token, get the <numeric> following that to use as the step size (making sure it is of the correct type again). If it isn't a STEP token, use 1 as the STEP size instead.

8 Check that we are now at the end of the statement, and set PTRA to point to the next statement.

9 Save PTRA on the FOR stack, to tell NEXT where to return to, and move the FOR stack pointer up by 15 bytes to cover this new FOR entry.

10 Finally, JMP to the statement interpreter to continue execution with the statements after the FOR statement.

The action of the NEXT statement is:

1 Look for a variable name after the NEXT token. If there is one, get its *variable descriptor block* and look down the FOR stack, throwing away the top entry, until the same variable is found. If the FOR stack was empty, generate a 'No FOR' error (error number 32); if the FOR stack wasn't empty, but a FOR loop could not be found with the same control variable, then generate a 'Can't match FOR' error (error number 33).

2 If there was no variable after the NEXT, check that the FOR stack is not empty (generate a 'No FOR' error if it is empty).

3 Get the type and address of the control variable, so that real and integer loop variables can be handled separately. Note, however, that NEXT does not differentiate between single-byte and 4-byte integers (although FOR does), so a single byte variable like '?A%' may give unpredictable results if used as a control variable.

4 Add the STEP size to the control variable.

5 If the new value of the control variable is inside the TO limit (less than or equal if STEP is positive; greater than or equal if STEP is negative) set PTRA to the address of the statement after the FOR statement (from the FOR stack), and JMP to the statement interpreter to continue execution with those statements.

6 If the new value of the control variable is outside the TO limit, move the FOR stack pointer down by 15 bytes to remove the top entry.

7 Set PTRA to point to the next character of the NEXT statement. If it is a comma (‘,’), go back to stage 1 as if it was a new NEXT statement (i.e. we have a multiple NEXT statement). Otherwise, JMP to the statement interpreter to continue execution with the statements following the NEXT statement.

5.7 ON…GOTO/GOSUB

This program control statement allows control to be passed to different parts of the program, depending on the value after the ON.

The action of the ON statement is:

1 If the first character after the ON token is an ERROR token, then go to the ON ERROR handler (section 5.8).

2 Evaluate the <numeric> following the ON token.

3 If the next character is not a GOTO or a GOSUB token, generate an ‘ON syntax’ error (error number 39).

4 Save the GOTO or GOSUB token on the 6502 stack.

5 If the value of the <numeric> was less than zero or greater than 255, give up trying to match it; otherwise, count along the list of line numbers to try to find the entry corresponding to the ON control value. If the entry was found, pop the GOTO or GOSUB token from the 6502 stack, and jump into the GOTO or GOSUB routine (depending on the token) to pass control to that line number.

6 If no match was made, remove the token from the 6502 stack, and look to see if there is an ELSE token on the line. If there is, handle it as if it was an ELSE in an IF statement (i.e. if there is a line number after the ELSE token, GOTO it, otherwise continue execution with the statements after the ELSE token).

7 If there is no ELSE token on the line, generate an ‘ON range’ error (error number 40).

In BASIC1, the token is not popped from the 6502 stack at stage 6; so if an ELSE clause is found and executed, the 6502 stack state has been messed up. If the ON statement was inside a FN or PROC (which keeps its return address on the 6502 stack), this will cause BASIC to crash on the FN or PROC return. The ON statement works correctly without the ELSE clause; and this bug has been cured in BASIC2 anyway.

5.8 ON ERROR

This statement does not directly change control of the program execution like the other program control mechanisms, but it does still involve using the pointers in a similar way. It changes the BASIC statements that the error handler executes when an error is generated.

BASIC keeps an ON ERROR pointer in page zero at &16,&17. This points to the start of a section of BASIC which will be executed when an error occurs.

In BASIC1 the default error handler (stored as 2 lines in the ROM starting at &B443) is:

```
   REPORT:IF ERL<>0 PRINT" at line ";ERL;
 0 PRINT:END
```

In BASIC2 the default error handler (only 1 line at &B433) is:

```
REPORT:IF ERL PRINT" at line ";ERL:END ELSE PRINT:END
```

The action of the ON ERROR statement is:

1 If the first character after the ERROR token is an OFF token, set the ON ERROR pointer to point to the default error handler, and JMP to the statement interpreter to continue with the statements after the ON ERROR OFF statement.

2 If the character was not an OFF token, then set PTRA to point to the first character after the ON ERROR, and set the ON ERROR pointer to point to this. This means that, should an error occur, these statements will be executed as the error handler.

3 Finally, skip the rest of the line as if it was a REM statement (we don't want to execute the error handler yet), and continue execution of the program on the next line.

6 Assembling and Disassembling

6.1 The Assembler

The built-in 6502 assembler in BASIC is a very useful tool, allowing both large and small machine code routines to be written easily. Being a part of BASIC itself, it is very easy to use BASIC variables and functions, conditional assembly (with some sections of the assembly code in IF…THEN statements), or macros (assembly sections in a GOSUB or FN/PROC).

The assembler is written very efficiently, and in total only occupies just over 1K of the 16K BASIC ROM.

The assembler mnemonics in the ROM are stored in a compressed format to save space. Only the least significant 5 bits of each mnemonic character are used, so that the whole mnemonic can be compressed into 15 bits of a 2-byte number. This also means that both upper case or lower case mnemonics will be recognised (or a mixture of the two). Fig 6.1 shows how the characters are packed.

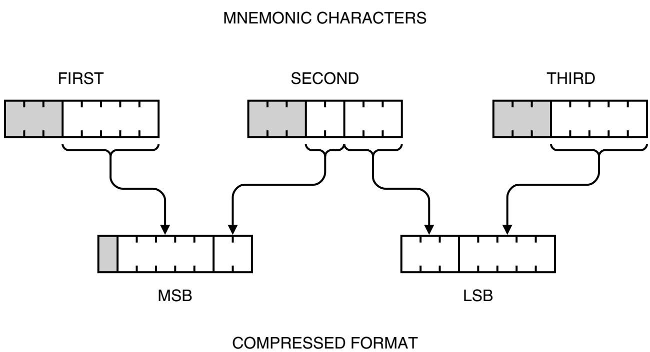


Figure 6.1 – Mnemonic compression.

A further byte is used for each mnemonic, to hold the ‘base value’ of the opcode. For instructions which can only have one addressing mode (such as the instructions which employ implied or relative addressing), this is the actual opcode used; for other instructions, this base value is modified by the actual addressing mode used.

The mnemonic and base opcode are stored as follows:

BASIC1	**BASIC2**	
&843B+M	&8450+M	MSB mnemonic
&8474+M	&848A+M	LSB mnemonic
&84AD+M	&84C4+M	base opcode

where M is the mnemonic number. Table 6.1 shows the mnemonic and base opcode value for each mnemonic number, as stored in the ROM table. Note that the directives OPT and EQU are stored the same as mnemonics, but they need no base opcode. The EQU directive is not implemented in BASIC1.

By comparing this table with fig 6.2, it can be seen that the mnemonics are grouped together with others which allow the same addressing modes. The assembler has a different section of machine code which is used for each of the different groups of mnemonics, to decide which addressing modes to allow. Section 1.5 gives these mnemonic groups.

Table 6.1 – Assembler Mnemonics

No.	Mnemonic	Base	No.	Mnemonic	Base
&01	BRK	&00	&0F	RTI	&40
&02	CLC	&18	&10	RTS	&60
&03	CLD	&D8	&11	SEC	&38
&04	CLI	&58	&12	SED	&F8
&05	CLV	&B8	&13	SEI	&78
&06	DEX	&CA	&14	TAX	&AA
&07	DEY	&88	&15	TAY	&A8
808	INX	&E8	&16	TSX	&BA
809	INY	&C8	&17	TXA	&8A
&0A	NOP	&EA	&18	TXS	&9A
&0B	PHA	&48	&19	TYA	&98
&0C	PHP	&08	&1A	BCC	&90
&0D	PLA	&68	&1B	BCS	&B0
&0E	PLP	&28	&1C	BEQ	&F0

No.	Mnemonic	Base	No.	Mnemonic	Base
&1D	BMI	&30	&2C	ROR	&66
&1E	BNE	&D0	&2D	DEC	&C6
&1F	BPL	&10	&2E	INC	&E6
&20	BVC	&50	&2F	CPX	&E0
&21	BVS	&70	&30	CPY	&C0
&22	AND	&21	&31	BIT	&20
&23	EOR	&41	&32	JMP	&4C
&24	ORA	&01	&33	JSR	&20
&25	ADC	&61	&34	LDX	&A2
&26	CMP	&C1	&35	LDY	&A0
&27	LDA	&A1	&36	STA	&81
&28	SBC	&E1	&37	STX	&86
&29	ASL	&06	&38	STY	&84
&2A	LSR	&46	&39	OPT	---
&2B	ROL	&26	&3A	EQU	---

MSD \ LSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	BRK Implied 1 7	ORA (Ind,X) 2 6				ORA ZP 2 3	ASL ZP 2 5		PHP Implied 1 3	ORA Imm 2 2	ASL Accum 1 2			ORA Abs 3 4	ASL Abs 3 6	
1	BPL Relative 2 2**	ORA (Ind),Y 2 5*				ORA ZP,X 2 4	ASL ZP,X 2 6		CLC Implied 1 2	ORA Abs,Y 3 4*				ORA Abs,X 3 4*	ASL Abs,X 3 7	
2	JSR Absolute 3 6	AND (Ind,X) 2 6			BIT ZP 2 3	AND ZP 2 3	ROL ZP 2 5		PLP Implied 1 4	AND Imm 2 4	ROL Accum 1 2		BIT Abs 3 4	AND Abs 3 4	ROL Abs 3 6	
3	BMI Relative 2 2**	AND (Ind),Y 2 5*				AND ZP,X 2 4	ROL ZP,X 2 6		SEC Implied 1 2	AND Abs,Y 3 4*				AND Abs,X 3 4*	ROL Abs,X 3 7	
4	RTI Implied 1 6	EOR (Ind,X) 2 6				EOR ZP 2 3	LSR ZP 2 5		PHA Implied 1 3	EOR Imm 2 2	LSR Accum 1 2		JMP Abs 3 3	EOR Abs 3 4	LSR Abs 3 6	
5	BVC Relative 2 2**	EOR (Ind),Y 2 5*				EOR ZP,X 2 4	LSR ZP,X 2 6		CLI Implied 1 2	EOR Abs,Y 3 4*				EOR Abs,X 3 4*	LSR Abs,X 3 7	
6	RTS Implied 1 6	ADC (Ind,X) 2 6				ADC ZP 2 3	ROR ZP 2 5		PLA Implied 1 4	ADC Imm 2 2	ROR Accum 1 2		JMP Indirect 3 5	ADC Abs 3 4	ROR Abs 3 6	
7	BVS Relative 2 2**	ADC (Ind),Y 2 5*				ADC ZP,X 2 4	ROR ZP,X 2 6		SEI Implied 1 2	ADC Abs,Y 3 4*				ADC Abs,X 3 4*	ROR Abs,X 3 7	
8		STA (Ind,X) 2 6			STY ZP 2 3	STA ZP 2 3	STX ZP 2 3		DEY Implied 1 2		TXA Implied 1 2		STY Abs 3 4	STA Abs 3 4	STX Abs 3 4	
9	BCC Relative 2 2**	STA (Ind),Y 2 6			STY ZP,X 2 4	STA ZP,X 2 4	STX ZP,Y 2 4		TYA Implied 1 2	STA Abs,Y 3 5	TXS Implied 1 2			STA Abs,X 3 5		
A	LDY Imm 2 2	LDA (Ind,X) 2 6	LDX Imm 2 2		LDY ZP 2 3	LDA ZP 2 3	LDX ZP 2 3		TAY Implied 1 2	LDA Imm 2 2	TAX Implied 1 2		LDY Abs 3 4	LDA Abs 3 4	LDX Abs 3 4	
B	BCS Relative 2 2**	LDA (Ind),Y 2 5*			LDY ZP,X 2 4	LDA ZP,X 2 4	LDX ZP,Y 2 4		CLV Implied 1 2	LDA Abs,Y 3 4*	TSX Implied 1 2		LDY Abs,X 3 4*	LDA Abs,X 3 4*	LDX Abs,Y 3 4*	
C	CPY Imm 2 2	CMP (Ind,X) 2 6			CPY ZP 2 3	CMP ZP 2 3	DEC ZP 2 5		INY Implied 1 2	CMP Imm 2 2	DEX Implied 1 2		CPY Abs 3 4	CMP Abs 3 4	DEC Abs 3 6	
D	BNE Relative 2 2**	CMP (Ind),Y 2 5*				CMP ZP,X 2 4	DEC ZP,X 2 6		CLD Implied 1 2	CMP Abs,Y 3 4*				CMP Abs,X 3 4*	DEC Abs,X 3 7	
E	CPX Imm 2 2	SBC (Ind,X) 2 6			CPX ZP 2 3	SBC ZP 2 3	INC ZP 2 5		INX Implied 1 2	SBC Imm 2 2	NOP Implied 1 2		CPX Abs 3 4	SBC Abs 3 4	INC Abs 3 6	
F	BEQ Relative 2 2**	SBC (Ind),Y 2 5*				SBC ZP,X 2 4	INC ZP,X 2 6		SED Implied 1 2	SBC Abs,Y 3 4*				SBC Abs,X 3 4*	INC Abs,X 3 7	

	0	
0	BRK Implied 1 7	— OP Code — Addressing Mode — Instruction Bytes; Machine Cycles

* Add 1 to N if page boundary is crossed.
** Add 1 to N if branch occurs to same page; add 2 to N if branch occurs to different page.

Figure 6.2 – 6502 op-code matrix.

6.2 The Disassembler

A disassembler is always useful: either for exploring the contents of the ROMs in the machine, or for checking that the machine code that you have just assembled is actually what you wanted (especially if it's got lots of conditional assembly in it).

Most disassemblers take up quite a lot of memory. For a start, they usually use a large table to decode the opcodes, with one entry for each of the 256 possible 1-byte numbers. Each entry of the table contains 3 bytes of mnemonic characters, and a further byte to give the addressing modes allowed with that particular opcode. This means that the disassembler is 1K long already, without any program to decode the instructions. Also, they are usually written in BASIC, which makes them slow, and even larger.

The disassembler described in this section uses the assembler tables in the ROM, and is written in machine code. When assembled, it is less than 500 bytes long, and so will fit in any 2 spare pages of memory (for example, from &B00 to &CFF, which is otherwise used for the user defined characters and function keys).

To use the disassembler, the resident integer variable D% is set to point to the first instruction to be disassembled (similar to the use of P% by the assembler). Typing 'CALL start%' will then disassemble one instruction, and leave D% pointing to the next one to be disassembled. If the variables have been re-set since the program was assembled, 'CALL &B00', or wherever the start of it is, will have to be used instead. This could be built in as a new statement, if required (see chapter 7).

To disassemble a length of code, a loop can be used:

```
        REPEAT:CALL &B00:UNTIL FALSE
or:     REPEAT:CALL &B00:UNTIL D%>&BFFF
```

(page mode will have to be used with a loop like this, as it disassembles at about 150 bytes/second, depending on the screen mode). In fact, a short program could be used to make the use of it very flexible; but the main advantage of it is that other programs can be loaded and run while the disassembler is still resident. If the user defined characters or function keys need to be used while the disassembler is in memory, PAGE could be moved up by 512 bytes, and it could be assembled there.

The ‘EQU’ directive has not been used in the program, so that it will work on either a BASIC1 or BASIC2 machine with no modification. PROCsetup (lines 9000 on) checks which version of BASIC is present, and sets up the correct ROM table labels before it is assembled.

Operation of the disassembler

The disassembler compares the opcode which is to be disassembled against the ‘base opcode’ of each mnemonic, and calculates the difference between them. If this difference can be made up by the offset of a particular addressing mode, and this addressing mode is allowed with the current mnemonic that it is checking, the mnemonic and addressing mode of that particular opcode have been found.

For example, if the value of the opcode was &31, this would be matched with the mnemonic ‘AND’ (base opcode &21) and the addressing mode ‘(IND),Y’ (offset &10). The base opcodes for each mnemonic are stored in the ROM tables, but the disassembler must contain the tables of allowed addressing modes for each group of instructions, and also the extent of each group. These tables are not in the ROM as the assembler does the addressing mode decoding in machine code rather than using tables.

The main opcode matching loop is from lines 1460–1760.

If the opcode is not matched with anything in the table, ‘???’ is printed out (for an unrecognised mnemonic). Note that ‘JMP (IND)’ has to be tested for separately (line 1190) as it does not fit into the pattern with the rest of them.

The allowed addressing mode offsets for each group are:

Addressing mode-grp.		Offset 00	04	08	0C	10	14	18	1C
0	&01–&21	X							
1	&22–&28	0	1	2	3	4	5	6	7
2	&29–&2C	1	A	3		5		7	
3	&2D–&2E	1		3		5		7	
4	&2F–&30	#	1		3				
5	&31		1		3				
6	&32–&33	3							
7	&34–&35	#	1		3		5		7
8	&36	0	1		3	4	5	6	7
9	&37–&38	1		3		5			

These possible offsets are held in the bit table ‘msktab’ in the program (lines 3490–3590). The number of the lowest mnemonic in each group is held in the table ‘grptab’ (lines 3600–3710).

The symbols in the table (X, #, A, 1 to 7) represent the possible addressing modes. Note that they don’t all line up: the addressing mode decode part of the program has to line up all these to get the correct addressing mode. The symbols represent:

X	either relative or implied
#	IMM (same as 2, but different pattern)
0	(IND,X)
1	ZP
2	IMM
3	ABS
4	(IND),Y
5	ZP,X
6	ABS,Y
7	ABS,X (,Y if LDX or STX)

The rest of the program handles the decoding and printing of the addressing mode characters and data. For most of the groups this is not too difficult, as the addressing mode corresponds directly with the offset from the base address; however, some others need to be shifted by an extra offset to ‘line up’ with the others. This shifting is done by lines 1810–2060.

The more complex addressing modes are printed using a bit mask table (lines 3800 to 3882) to decide which characters to print. The simpler addressing modes are printed by a separate part of the routine.

```
  10 REM Machine code disassembler
  15 REM using assembler ROM tables
  20 REM
  25 REM      M D Plumbley  1984
  30 REM
  99
 100 PROCsetup            :REM Set up ROM entry points
 590
 595 REM *** Allocate workspace ***
 600 worksp = &0070
 605 grpmsk = worksp      :REM Bit mask of allowed modes
 610 ytemp = worksp+1     :REM Temp for addr mode group
 615 mdstor = worksp+2    :REM Store for addressing mode
 620 opcode = worksp+3    :REM Opcode read in from memory
 625 data  = worksp+4     :REM The 2 bytes after the opcode
 630 addr  = worksp+6     :REM Copy of address in D%
 635 mnem  = worksp+8     :REM Mnemonic construction area
 640 xtemp = worksp+10    :REM Temp for mnemonic number
 645 lastch = worksp+11   :REM Last char of mnemonic
 650 nbytes = worksp+12   :REM Number of instruction bytes
 655 chrmsk = worksp+13   :REM Addr mode character mask
 690
 700 count = &1E
 799
 900 start% = &0B00       :REM User defined char/key area
 905
 910 FOR opt% = 0 TO 3 STEP 3
 920 P% = start%
 950 [OPT opt%
1000 .disass
1010     LDA &410         \Get address from D%, and put it
1020     STA addr         \ in the workspace
1030     LDA &411
1040     STA addr+1
1045
1050     LDY #2           \Transfer the opcode and 2 data
1060 .txbyte              \ bytes to be disassembled
1070     LDA (addr),Y
1080     STA opcode,Y
1090     DEY
1100     BPL txbyte
1105
1110     LDA addr+1       \Print the address and the opcode
1120     JSR phex
1130     LDA addr
```

```
1140     JSR phexsp
1150     JSR pspace
1160     LDA opcode
1170     JSR phexsp
1180
1190     LDA opcode        \If we have a JMP (XXXX), then
1200     CMP #&6C          \ set the mnemonic to "JMP"
1210     BNE mtchop        \ (mnemonic number &32),and the
1220     LDX #&32          \ addressing mode to 8.
1230     STX xtemp         \ Otherwise, attempt to match the
1240     LDA #8            \ opcode with the table
1250     STA mdstor
1260     JMP domode
1270
1280 .nomtch
1290     JSR tbmnem        \If we get here, no match was
1300     LDY #3            \ found, so print a "???",
1310     LDA #ASC"?"       \ and go on to add 1 to D%
1320 .pqloop               \ before finishing
1330     JSR pchar
1340     DEY
1350     BNE pqloop
1360     JMP add1
1370
1380 .tbmnem
1390     LDY #16           \Print spaces until we get to
1400 .tbloop               \ the 16th column. This lines
1410     JSR pspace        \ up all the mnemonics.
1420     CPY count
1430     BCS tbloop
1440     RTS
1450
1451 \ ** Main opcode matching routine **
1452
1460 .mtchop               \Go through all the mnemonics,
1470     LDX #&39          \ and try to match one to the
1480     LDY #&0A          \ opcode.
1485
1490 .nextop
1500     DEX               \If we have tried all the
1510     BEQ nomtch        \ mnemonics, it is invalid.
1515
1520     TXA               \Check to see if we are now in
1530     CMP grptab,Y      \ a new mnemonic group.
1540     BCS samgrp
1550     DEY
1560     LDA msktab,Y
1570     STA grpmsk
1575
1580 .samgrp
1590     LDA opcode        \The opcode can only have this
1600     SEC               \ mnemonic if is a positive
```

```
1610     SBC opbase,X     \ offset from the "base opcode"
1620     BCC nextop       \ of it. Also, the offset must
1630     LSRA             \ be divisible by 4, and must be
1640     BCS nextop       \ &1C or less (&1C=4*7)
1650     LSRA
1660     BCS nextop
1670     CMP #8
1680     BCS nextop
1685
1690     STA mdstor       \Check to see if this addr mode
1700     STY ytemp        \ is allowed with this mnemonic.
1710     TAY              \ If it isn't, go back to check
1720     LDA bittab,Y     \ for another mnemonic.
1730     LDY ytemp        \ "grpmsk" holds the allowed
1740     AND grpmsk       \ addr modes for this mnemonic.
1750     BEQ nextop
1755
1760     STX xtemp        \Success!! - so save the mnemonic
1762                      \ number
1765
1770     LDY ytemp        \If the mode group is 0, it is
1790     TYA              \ either implied or relative
1800     BEQ imprel
1805
1810     LDA #&10         \If the group mask suggests that
1820 .trymsk              \ the mnemonic doesn't allow
1830     BIT grpmsk       \ absolute addressing, we have to
1840     BNE mskok        \ alter the addressing mode until
1850     INC mdstor       \ it does. (The "BPL" will always
1860     LSR grpmsk       \ work after a "LSR".)
1870     BPL trymsk
1875
1880 .mskok               \When we get here, the mask and
1890     LDA grpmsk       \ addr mode offset is OK.
1900     AND #&08         \ However, if the addr mode is 0
1910     BNE modeok       \ and (indir),Y is not allowed,
1920     LDA mdstor       \ then it is really immediate
1930     BNE modeok       \ addressing, which should be
1940     LDA #2           \ addr mode 2
1950     STA mdstor
1955
1960 .modeok              \When we get here, the only thing
1970     CPY #2           \ left to test for is accumulator
1980     BNE domode       \ addressing. If the "allowed
1990     TYA              \ mode" group is 2, and the addr
2000     CMP mdstor       \ mode is also 2, then print the
2010     BNE domode       \ mnemonic, followed by an "A",
2020     JSR pmnem        \ and go to add 1 to D% before
2030     LDA #ASC"A"      \ finishing. Otherwise, go to
2040     JSR pchar        \ "domode".
2050 .jadd1
2060     JMP add1
```

```
2065
2070 .imprel                    \If we get here, the addressing
2080     LDX xtemp              \ mode is either relative or
2090     CPX #&1A               \ implied.
2100     BCS rel
2105
2110     JSR pmnem              \If it is implied, print the
2120     JMP add1               \ mnemonic, and add 1 to D%
2125
2130 .rel                       \If it is relative, we have 1
2140     LDA data               \ extra data byte to print out
2150     JSR phexsp             \ before the mnemonic.
2160     JSR pmnem
2165
2170     LDA #0                 \The absolute addr has to be
2180     STA data+1             \ calculated from the offset.
2190     LDA data               \ First extend the sign of the
2200     BPL nodec              \ offset byte into 2 bytes
2210     DEC data+1
2215
2220 .nodec                     \Then add this 2-byte offset to
2230     SEC                    \ D%, adding another 2 with it.
2240     ADC &410               \ One extra is added by setting
2250     STA data               \ the carry before the addition,
2260     LDA &411               \ the other is added by
2270     ADC data+1             \ incrementing the address
2280     STA data+1             \ afterwards.
2290     INC data
2300     BNE nopage
2310     INC data+1
2315
2320 .nopage                    \Finally, print the absolute
2330     JSR pabs               \ address, and add 2 to D% before
2340     JMP add2               \ leaving.
2350
2355 \ ** Print the mnemonic ***
2360 .pmnem
2370     LDX xtemp              \First, get the number of the
2380     JSR tbmnem             \ mnemonic, and get the LSB and
2390     LDA lsbmn,X            \ MSB of the compressed mnemonic.
2400     ASLA                   \ The shifts are to get the bits
2410     STA mnem               \ ready for the first 5 bits to
2420     LDA msbmn,X            \ be shifted out.
2430     ROLA
2440     STA mnem+1
2445
2450     LDX #3                 \This is the main loop which
2460 .mcloop                    \ shifts 3 characters out of
2470     LDA #0                 \ the 2-byte compressed mnemonic.
2480     LDY #5                 \ 5 bits at a time are shifted
2490 .mbloop                    \ out into the accumulator, and
2500     ASL mnem               \ they are then ORed with &40 to
```

```
2510     ROL mnem+1        \ turn them into upper case
2520     ROLA              \ letters.
2530     DEY
2540     BNE mbloop
2550     ORA #&40
2560     JSR pchar
2570     DEX
2580     BNE mcloop
2585
2590     STA lastch        \Save the last character printed:
2595                       \ it might be an "X".
2600     JMP pspace        \Print a space, and exit.
2605
2606 \ ** Handle the addressing mode stuff **
2610 .domode
2620     LDY mdstor        \First, get the number of bytes
2630     LDX mdbyts,Y      \ used by this addr mode, and
2640     STX nbytes        \ save it.
2645
2650     DEX               \Print the required number of
2660     BEQ nodata        \ data bytes before the mnemonic.
2670     LDA data
2680     JSR phexsp
2690     DEX
2700     BEQ nodata
2710     LDA data+1
2720     JSR phexsp
2725
2730 .nodata
2740     JSR pmnem         \Print the mnemonic.
2745
2750     LSR mdstor        \If the addr mode was odd, it is
2760     BCS smplmd        \ a simple one, so deal with it
2770
2780     LDY mdstor        \If it was not a simple mode, get
2790     LDA chmstb,Y      \ the mask of characters to be
2800     STA chrmsk        \ printed into "chrmsk".
2805
2810     LDY #6            \Starting at the 7th (0..6) char,
2820 .newchr               \ if the bit shifted out of the
2830     ASL chrmsk        \ mask is set, then print it.
2840     BCC nochr
2850     LDA chtab,Y
2860     JSR pchar
2865
2870 .nochr                \If we have got to the 5th char,
2880     CPY #5            \ the data can be printed (i.e.
2890     BNE nodat         \ the "#" or "(" has been printed
2900     JSR pdata         \ if there was one)
2905
2910 .nodat                \Go round for another character
2920     DEY               \ if we haven't printed them all;
```

```
2930     BPL newchr        \ otherwise add "nbytes" to D%
2940     JMP addn          \ and exit.
2950
2960 .smplmd               \If we get here, the addr mode is
2970     JSR pdata         \ either "zero-page", "absolute",
2980     LSR mdstor        \ "zero-page,X" or "absolute,X".
2990     LSR mdstor        \ Shifting out the 2nd bit from
3000     BCC addn          \ "mdstor" gives whether indexed
3010     LDA #ASC","       \ addressing is required.
3020     JSR pchar
3025
3030     LDA #ASC"X"       \If the last character of the
3040     CMP lastch        \ mnemonic was a "X", then use
3050     BNE px            \ "Y" as the index
3060     LDA #ASC"Y"
3070 .px
3080     JSR pchar         \Print the index character, and
3090     JMP addn          \ add "nbytes" to D%.
3095
3096 \ ** Routines to print the data after the mnemonic **
3110 .pabs                 \Print the data as an absolute
3120     LDA #ASC"&"       \ address.
3130     JSR pchar
3140     LDA data+1
3150     JSR phex
3160     LDA data
3170     JMP phex
3175
3180 .pdata                \If the total number of bytes for
3190     LDA nbytes        \ this addressing mode is not 2
3200     CMP #2            \ (i.e. it is 3) then print the
3210     BNE pabs          \ absolute address.
3220 .pzerop
3230     LDA #ASC"&"       \Print the data as a single byte.
3240     JSR pchar
3250     LDA data
3260     JMP phex
3265
3267 \** Exit points; add size to D% and exit ***
3270 .add1                 \Add 1 to D%, and then exit
3280     LDA #1
3290     BNE add
3300 .add2                 \Add 2 to D%, and then exit
3310     LDA #2
3320     BNE add
3360 .addn                 \Add the number of bytes in the
3370     LDA nbytes        \ instruciton to D%, then exit
3375
3380 .add                  \Add A to D%
3390     CLC               \ (The least significant 2 bytes
3400     ADC &410          \ of D%, are stored in &410 and
3410     STA &410          \ &411)
```

```
3420     LDA &411
3430     ADC #0
3440     STA &411
3445
3450     JMP pnewl        \Print a CRLF and exit
3460
3480 \*** Allowed offset table ***
3482 \This table gives the allowed addr mode offset for
3484 \ each group of mnemonics. Bit 7 (the top bit) is set
3486 \ if 0 is allowed; bit 6 set if 4 is allowed; etc.
3490 ]:msktab=P%:P%=P%+10
3500 msktab?0 = &80
3510 msktab?1 = &FF
3520 msktab?2 = &EA
3530 msktab?3 = &AA
3540 msktab?4 = &D0
3550 msktab?5 = &50
3560 msktab?6 = &80
3570 msktab?7 = &D5
3580 msktab?8 = &DF
3590 msktab?9 = &A8
3592
3594 REM ** Addressing mode groups **
3596 REM This table contains the starts of the mnemonics
3598 REM which have the same allowed addressing modes
3600 grptab=P%:P%=P%+11
3610 grptab?&0 = &01
3620 grptab?&1 = &22
3630 grptab?&2 = &29
3640 grptab?&3 = &2D
3650 grptab?&4 = &2F
3660 grptab?&5 = &31
3670 grptab?&6 = &32
3680 grptab?&7 = &34
3690 grptab?&8 = &36
3700 grptab?&9 = &37
3710 grptab?&A = &39
3712
3714 REM *** Bit position table ***
3716 REM This table contains the bit position corresponding
3718 REM to each addressing mode
3720 bittab=P%:P%=P%+8
3730 bittab?0 = &80
3740 bittab?1 = &40
3750 bittab?2 = &20
3760 bittab?3 = &10
3770 bittab?4 = &08
3780 bittab?5 = &04
3790 bittab?6 = &02
3800 bittab?7 = &01
3802
3804 REM *** Addr mode character mask table ***
```

```
3806 REM This table gives the characters to be printed for
3808 REM the non-simple addressing modes
3810 chmstb=P%:P%=P%+5
3820 chmstb?0 = &78 :REM "(,X)"
3830 chmstb?1 = &80 :REM "#"
3840 chmstb?2 = &4E :REM "(),Y"
3850 chmstb?3 = &06 :REM ",Y"
3860 chmstb?4 = &48 :REM "()"
3870 chtab=P%:P%=P%+7
3880 $chtab="Y,)X,(#"
3882
3884 REM *** Addressing mode bytes table ***
3886 REM This table gives the total number of bytes used by
3888 REM a given addressing mode.
3890 mdbyts=P%:P%=P%+9
3900 mdbyts?0 = 2
3910 mdbyts?1 = 2
3920 mdbyts?2 = 2
3930 mdbyts?3 = 3
3940 mdbyts?4 = 2
3950 mdbyts?5 = 2
3960 mdbyts?6 = 3
3970 mdbyts?7 = 3
3980 mdbyts?8 = 3
8000
8010 NEXT
8015 @%=0
8020 PRINT'"Code length =&"~P%-start%
8190
8200 PRINT'''''"** WARNING: Once assembled, the code"
8210 PRINT"generated by this program is not"
8220 PRINT"transferable between different BASICs"
8230 PRINT
8300 PRINT"DO ""CALL &"~disass""" to disassemble 1 line"
8305 PRINT"D% points to code to be disassembled"'
8310 END
8990
8992 REM *** Set up ROM entry points, allowing for ***
8993 REM ***  BASIC 1 and BASIC 2.                 ***
9000 DEFPROCsetup
9010 basic1$ = "BASIC"+CHR$0+"(C)1981 Acorn"+CHR$&A
9020 basic2$ = "BASIC"+CHR$0+"(C)1982 Acorn"+CHR$&A
9030 IF $&8009=basic1$ THEN PROCset1 :ENDPROC
9040 IF $&8009=basic2$ THEN PROCset2 :ENDPROC
9050 PRINT "NOT BASIC 1 OR 2"
9060 END
9290
9292 REM *** Set up BASIC 1 entry points           ***
9300 DEFPROCset1
9310 opbase = &84AD :REM Opcode base value table
9315 lsbmn  = &843B :REM Table of LSB of mnemonic
9320 msbmn  = &8474 :REM Table of MSB of mnemonic
```

```
9325 phex   = &8570 :REM Print A as a HEX byte
9330 phexsp = &856A :REM Print A in HEX, then space
9335 pspace = &B57B :REM Print a space
9340 pnewl  = &BC42 :REM Print a CRLF
9345 pchar  = &B571 :REM Print char in A
9350 ENDPROC
9490
9492 REM *** Set up BASIC 2 entry points           ***
9500 DEFPROCset2
9510 opbase = &84C4 :REM Opcode base value table
9515 lsbmn  = &8450 :REM Table of LSB of mnemonic
9520 msbmn  = &848A :REM Table of MSB of mnemonic
9525 phex   = &B545 :REM Print A as a HEX bytes
9530 phexsp = &B562 :REM Print A in HEX, then space
9535 pspace = &B565 :REM Print a space
9540 pnewl  = &BC25 :REM Print a CRLF
9545 pchar  = &B558 :REM Print char in A
9550 ENDPROC
```

7 Adding New Commands

When the BASIC interpreter discovers anything which it doesn't recognise, it generates an error (usually 'Mistake'), to stop processing of the program or command and go back to command mode. This section describes how new statements and commands can be added to BASIC by intercepting this error.

7.1 Trapping BRK

The method that BASIC uses to generate an error, is to execute a BRK instruction, which is followed by a number of bytes in a standard error format. This format is:

BRK instruction to generate the error
Single byte error number (ERR)
Error message (like 'Mistake')
A zero byte to terminate the message

This is the standard method of generating errors on the Acorn BBC system, and it allows errors to be 'trapped' by intercepting the BRK vector (at &202). By trapping the errors generated by BASIC, it is possible to add new commands, overlay procedures, etc., and continue where it left off. Other errors which are generated by BASIC are described in chapter 11.

When a BRK instruction is executed, the Machine Operating System will JMP to the BRK handler whose address is in the BRK vector at &202,&203. On entry to the BRK handler the following conditions prevail:

(a) The A, X and Y registers are unchanged from when the BRK instruction was executed.

(b) The 6502 stack is prepared ready for an RTI to the instruction following the BRK instruction (i.e. with the 6502 flag byte on the top of the stack, and the return address underneath it). This will return control to the instruction 2 bytes after the BRK instruction.

(c) The pointer in locations &FD,&FE points to the 'error number' byte after the BRK instruction.

Although a return from a BRK instruction is possible (it can be used as a breakpoint in a machine code program), BASIC does not expect such a return; executing an RTI after a BRK instruction has been executed by BASIC (or any other program using it as an error generating mechanism) will probably have fatal results.

The small program below illustrates how the BRK vector can be intercepted, to cause a bleep (CHR$7) each time an error is generated. If you get fed up with this, pressing BREAK or typing '*BASIC' will re-set the BRK vector to point to the default BRK handler in BASIC, missing out this routine.

The code assembles into the user defined character area from &0C00 onwards. If any user defined characters are to be used while the routine is 'linked in' to the BRK vector, it could be assembled somewhere else, by changing line 900. Space could be allocated at PAGE for it by adding 256 to PAGE before the routine is loaded (or typed in), and assembling the code to the old location of PAGE, underneath the BASIC program.

```
  10 REM   Routine to print a bleep on an error
  20 REM
 400 brkv   = &0202           :REM BRK vector location
 410 oldbrk = !brkv AND &FFFF :REM Get default BRK handler
 420
 500 oswrch = &FFEE           :REM OSWRCH (to print bleep)
 505
 900 start% = &0C00           :REM User char area
 905
 910 FOR opt% = 0 TO 3 STEP 3
 915 P%=start%
 920 [OPT opt%
 925
1000 .newbrk
1005     PHA                  \ Save A
1007
1010     LDA #&7              \ Print a bleep
1015     JSR oswrch
1017
1020     PLA                  \ Retrieve A, and continue
1025     JMP oldbrk           \ with default BRK handler.
9000 ]
9010 NEXT
9020 IF newbrk=oldbrk PRINT"Already set up":END
9030 brkv?0 = newbrk MOD &100 :REM Set up BRK vector to
9040 brkv?1 = newbrk DIV &100 :REM point to this routine.
9050 END
```

When the program is assembled, the address of the default BRK handler is retrieved at line 410. This is where the new routine will JMP to when it has printed its bleep. This means that the error message will still be printed by the BASIC BRK handler, as though nothing had happened.

After the program has been assembled, its start address is poked into the BRK vector at lines 9030 and 9040 (the BRK vector is stored low byte first). Line 9020 checks to see if the program has already been set up. If it has, the new BRK handler would jump back to *itself* when it has finished. This means that if any error occurs, it will continue printing bleeps until BREAK is pressed – not very useful (try assembling it twice, and see what happens). This is something to look out for with most error trapping routines; if they fail to clear the error which called them, it will be generated again, and they will be called again in exactly the same situation.

The error trap routine saves A by pushing it on the stack, while it prints the bleep. This is not necessary if the BASIC error handler will be JMPed to immediately afterwards, as it does not use it; but it would be important if a different routine, which relies on A being correct on entry, had intercepted the BRK vector *before* this program was entered. If this other routine had been linked in to the BRK vector in a similar way, the 'JMP oldbrk' on the end of this routine will jump into that routine when it is finished, rather than the BASIC BRK handler.

It is usually a good idea to save any registers you are going to use, if control will be returned to another routine which may need them. If the 'No room' error is being trapped, for example (chapter 11, BASIC2 only), all of the 6502 registers (A, X, Y) must be intact so that the source of the error can be determined.

7.2 The 'Mistake' error

If you type in a word that BASIC doesn't recognise, it generates a 'Mistake' error (error number 4). However, it leaves its statement pointer, PTRA, pointing one character after the start of the name (PTRA was advanced one byte by the action of reading in the first character). This means that the word which caused the error to be generated can be checked, and action taken if it corresponds to a new, 'home-made' statement.

The 'Mistake' error is actually generated when BASIC fails to find an '=' character, often due to a mistyped keyword (such as 'PRIT' instead of 'PRINT'). When this happens, the sequence of actions is as follows:

1 The statement interpreter reads the character at PTRA, advancing PTRA to point to the next character.

2 The character is not a keyword token. It is alphabetic, however, so it looks like the start of a variable name; and the statement interpreter jumps into the variable assignment handler.

3 The assignment handler scans what it thinks is a variable name, using PTRB. This means that PTRA still points one byte after the first character of the name. If the name is of a variable which doesn't already exist, it will create it; but only *after* it has checked that there is an '=' following it.

4 The assignment routine checks for an '=' after the variable name. If it doesn't find one (which it won't, if it was a mistyped keyword), it generates a 'Mistake' error. If it does find one, it continues with the assignment.

In fact there are 5 slightly different causes of a 'Mistake':

(a) A non-existent variable name was found, without an '=' following it. This error is generated before the variable is created, by a sort of 'pre-check' before the main assignment routine is entered.

(b) An existing variable name was found, without an '=' following it. This is not quite the same as (a), above, but the only difference is the return address left on the 6502 stack.

(c) A 'LET' statement, followed by a valid variable, was found, but there was no '=' following the name. If the variable did not exist before this statement, it would have been created before the error was generated (unlike (a) above).

(d) A psuedo-variable name, like ‘HIMEM’, was found, but no ‘=’ followed it.

(e) A ‘FOR’ statement was found, followed by a valid variable, but no ‘=’ followed the name.

All of these leave PTRA pointing 1 byte after the start of the statement, but (c), (d), and (e) leave the 6502 stack in different states. Fortunately, this only happens if the first character of the statement is a keyword token; so if new statements are to be introduced, they should not be allowed to start with one of the tokens mentioned above (so ‘FORAGE’ cannot be a new statement keyword).

Note that new keywords cannot begin with any other tokens either (like the ‘TO’ in ‘TOTAL’) as these will cause a ‘Syntax error’ rather than a ‘Mistake’. However, some of the BASIC keywords are not tokenised if followed by an alphanumeric character (see section 2.3.1), so ‘TIMER’ could be used as a new statement (the ‘TIME’ part would not be tokenised).

For (a) and (b), the prevailing conditions on entry to the BRK handler are:

&FD ,&FE	points to the error number (4)	
Stack contents:	RTI information	3 bytes
	Return address	2 bytes
PTRA:	points 1 after the first byte of the name	

Other conditions are not so important (see chapter 11, error number 4).

When a new statement has been recognised, the 3 bytes of RTI information (pushed by the BRK instruction) and the 2 bytes of return address (the ‘=’ was checked by a subroutine called by the assignment handler) must be pulled from the stack before execution is continued. If this is not done, any FNs or PROCs will not return properly, as they expect their return address to be on the top of the stack (see section 5.3).

7.3 A single character statement

The routine in this section shows a simple example of adding a new statement, by just checking the first character of the statement; the one just before PTRA. If it is a ‘B’, it pulls the 5 bytes to be discarded from the stack, checks that the ‘B’ is the only thing (apart from spaces) in the statement, and produces a bleep. Finally, it JMPs to the BASIC entry point to continue executing the following statements.

Instead of being initialised when the program is assembled, this program links in to the BRK vector when the small routine at the start is CALLed (lines 1000 to 1115). Any programs which are initialised in this way don’t need to be reassembled each time they are used.

Note that the EQUB and EQUS assembler directives are used in this program (lines 1025 to 1040), as they are much clearer than the equivalent in BASIC. However, the EQU directive is not implemented in BASIC 1, and should be replaced with its equivalent using indirection operators.

```
10 REM *** Program to add single character command ***
12 REM
14 REM          M D Plumbley       1984
16 REM
18 REM This program traps the BRK vector. On an error,
20 REM  if ERR (the error number) is 4 ("Mistake")
22 REM  and the unrecognised statement is the single
24 REM  character "B", then a bleep will be produced.
26 REM
28 REM If the error number is not 4, or the first char
30 REM  of the statement is not a "B", then control will
32 REM  be passed to the default error handler.
34 REM
36 REM When setting up, the program tests for BASIC 1
38 REM  or BASIC 2, and uses the corresponding ROM
40 REM  entry points.
42 REM
44 REM Before using on BASIC I, all EQU directives
46 REM  should be replaced with indirections:
48 REM  "EQUB X"  => "]?P%=X:P%=P%+1:[OPTopt%"
50 REM  "EQUS A$" => "]$P%=A$:P%=P%+LEN$P%:[OPTopt%"
52 REM
54 REM The code is assembled into the user defined
56 REM  character space: alternatively, space could
58 REM  be reserved at PAGE for it.
```

```
  60 REM
  99
 100 PROCsetup      :REM Set up correct ROM entry points
 490
 495 REM *** OS routines and vectors ***
 500 OSWRCH = &FFEE
 550 BRKV   = &0202
 799
 900 start% = &0C00 :REM Assemble into user char space
 905
 910 FOR opt% = 0 TO 3 STEP 3
 920 P% = start%
 950 [OPT opt%
1000 .init
1005     LDA &8015             \Test that the correct
1010     CMP #baschr           \ version of BASIC is
1015     BEQ basok             \ in the ROM.
1016
1020     BRK                   \If it isn't, print an
1025     EQUB 60               \ error message.
1030     EQUS "Not BASIC "     \ (baschr set by PROCsetup)
1035     EQUB baschr
1040     EQUB 0
1041
1045 .basok
1050     LDA BRKV              \Load the current BRK vector
1055     LDX BRKV+1            \ into A and X.
1056
1060     CMP #newbrk MOD &100 \If this routine is already
1065     BNE ntsavd            \ set up, don't change BRKV.
1070     CPX #newbrk DIV &100
1075     BEQ saved
1076
1078 .ntsavd
1080     STA svbrkv            \It has not been set up
1085     STX svbrkv+1          \ already, so save old
1090     LDA #newbrk MOD &100 \ BRKV, and set up the new
1095     STA BRKV              \ one.
1100     LDA #newbrk DIV &100
1105     STA BRKV+1
1106
1110 .saved
1115     RTS
1190
1192 \ *** This is the new BRK handling routine ***
1200 .newbrk
1205     PHA            \Save A and Y on 6502 stack
1210     TYA
1215     PHA
1216
1220     LDY #0         \Get error number
1225     LDA (&FD),Y
```

```
1226
1280     CMP #4        \If "Mistake", check for a "B"
1285     BEQ mistak
1286
1400 .giveup
1410     PLA           \Restore A and Y from 6502 stack
1420     TAY
1430     PLA
1431
1440     JMP (svbrkv)  \Go to old BRK handler
1441
1490 \ *** If we get here, an error 4 ("Mistake") has  ***
1492 \ *** ocurred, so see if the charcter is a "B".   ***
1500 .mistak
1510     LDY &A        \Get character at start of statement
1520     DEY
1530     LDA (&B),Y
1531
1540     CMP #ASC"B"   \If it is not a "B", go to the old
1550     BNE giveup    \ BRK handler
1551
1560     PLA           \Discard saved A and Y from stack
1570     PLA
1571
1580     PLA           \Discard RTI information from the
1590     PLA           \ 6502 stack. This is automatically
1600     PLA           \ pushed by the BRK instruction.
1601
1610     PLA           \Discard return addr (of routine
1620     PLA           \ to check for "=") from stack
1621
1630     JSR chksda    \Check for end of statement
1631
1640     LDA #7        \Print a beep
1650     JSR OSWRCH    \ (action at last!)
1651
1660     JMP cont      \Continue execution
1661
6899
6990 \ ***          Routine variables area           ***
6991
7000 .svbrkv EQUW !BRKV   \Space to save old BRK vector
7010
8000 ]
8010 NEXT
8015 @%=0
8020 PRINT'"Code length =&"~P%-start%
8190
8200 PRINT'''''"** WARNING: Once assembled, the code"
8210 PRINT"generated by this program is not"
8220 PRINT"transferable between different BASICS"
8230 PRINT
```

```
8300 PRINT"Execute ""CALL &"~init""" to initialise."'
8310 END
8990
8992 REM *** Set up ROM entry points, allowing for ***
8993 REM *** BASIC 1 and BASIC 2.                  ***
9000 DEFPROCsetup
9010 basic1$ = "BASIC"+CHR$0+"(C)1981 Acorn"+CHR$&A
9020 basic2$ = "BASIC"+CHR$0+"(C)1982 Acorn"+CHR$&A
9030 IF $&8009=basic1$ THEN PROCset1 :ENDPROC
9040 IF $&8009=basic2$ THEN PROCset2 :ENDPROC
9050 PRINT "NOT BASIC I OR II"
9060 END
9290
9292 REM *** Set up BASIC 1 entry points           ***
9300 DEFPROCset1
9310 baschr = ASC"1":REM Used by init routine
9320 chksda = &9810 :REM Check for statement delimiter
9330 cont   = &8B0C :REM Cont execution at next statement
9490
9492 REM *** Set up BASIC 2 entry points           ***
9500 DEFPROCset2
9505 baschr = ASC"2":REM Used by init routine
9530 chksda = &9857 :REM Check for statement delimiter
9540 cont   = &8B9B :REM Cont execution at next statement
9550 ENDPROC
```

The general operation of the program is as follows:

PROCsetup is called to set up the correct ROM entry points required by the routine (‘Check for statement delimiter’ and ‘Continue execution’ in this case). This uses the copyright string to check for the version type, and calls PROCsetl or PROCset2 depending on the year (1981 or 1982). Alternatively, the paged ROM version number, held in location &8008, could be used. This is &00 for BASIC1, and &01 for BASIC2.

When the assembled code is initialised by CALLing the start, the initialisation routine first checks that the year of the ROM is the same as the one it was assembled for; if it isn’t, it won’t link itself in (as the ROM entry points will be wrong). Note that this check will *only* work if the BASIC ROM is paged in when the initialisation routine checks the year; and not if the DFS, say, is paged in (if the routine has just been ‘*RUN’). See chapter 10 for more on this.

If the ROM is correct, the initialisation routine saves the contents of the BRK vector at ‘svbrkv’, and sets the BRK vector to point to the new BRK handling routine.

When an error is generated, and ‘newbrk’ is entered, it checks that the error number pointed to by &FD,&FE is 4, if it isn’t, the error was not a ‘Mistake’, and a JMP is made to the default BRK handler to deal with it.

If the error is a ‘Mistake’, the character before PTRA is tested to see if it is a ‘B’ (the base of PTRA is stored in &B,&C with the offset in &A). If it isn’t the old BRK handler is JMPed to to print the ‘Mistake’ message.

If it is a ‘B’, then the 5 bytes on the 6502 stack are pulled from it (together with the 2 saved registers from the BRK handler). Then the ROM routine is called which checks for the end of the statement at PTRA (which still points just after the ‘B’). This will produce a ‘Syntax error’ (error number 16) if it doesn’t find a ‘:’, an ELSE token, or the end of the line.

Finally, a bleep is printed, and a JMP is made to the ROM routine which continues with the execution of the program. Note that this routine expects the ‘Check for statement delimiter’ routine to be called before it, so that PTRA is set up to actually point 1 byte after the statement terminator. These ROM routines are detailed in chapter 10.

7.4 Recognising keywords

Just using single character statements is not very versatile: most of the time it would be much more useful to give the new statements keywords which reflect the action that they perform, like ‘DUMP’ to dump the variables, or ‘REN’ to renumber a program. The program in this section shows how to implement a command line interpreter to recognise keywords from a table.

The keywords implemented in the program are ‘BEEP’, which beeps (again), and ‘DUMP’, which lists the current active dynamic variables (see section 3.1.2). Neither of them take any arguments.

Note that the EQU assembler directive has been used again (lines 1025 to 1040 as before, and lines 2500 to 2580 in the keyword table).

```
  10 REM *** Program to add new BASIC commands ***
  12 REM
  14 REM     M D Plumbley     1984
  16 REM
  18 REM This program traps the BRK vector. On an error,
  20 REM  if ERR (the error number) is 4 ("Mistake")
  22 REM  then a command line interpreter will test the
  24 REM  statement for a keyword to recognise. If it is
  26 REM  recognised, the keyword's action is performed.
  28 REM  Otherwise, control is passed on to the default
  30 REM  BRK handler.
  32 REM
  34 REM The code is assembled into the user key/char
  36 REM  space: alternatively, space could be reserved
  38 REM  at PAGE for it.
  40 REM
  42 REM Before using with BASIC 1, the EQUs should be
  44 REM  replaced with their equivalent:
  46 REM  "EQUB X"  => "]?P%=X:P%=P%+1:[OPTopt%"
  48 REM  "EQUW X"  => "]!P%=X:P%=P%+2:[OPTopt%"
  50 REM  "EQUS A$" => "]$P%=A$:P%=P%+LEN$P%:[OPTopt%"
  52 REM
  99
 100 PROCsetup      :REM Set up correct ROM entry points
 490
 495 REM *** OS routines and vectors ***
 500 OSWRCH = &FFEE
 550 BRKV   = &0202
 590
 600 svbrkv = &0070 :REM Space to save old BRK vector
 690
 900 start% = &0B00 :REM User key/char area
 905
 910 FOR opt% = 0 TO 3 STEP 3
 920 P% = start%
 950 [OPT opt%
1000 .init
1005     LDA &8015           \Test that the correct
1010     CMP #baschr         \ version of BASIC is
1015     BEQ basok           \ in the ROM.
1016
1020     BRK                 \If it isn't, print an
1025     EQUB 60             \ error message.
1030     EQUS "Not BASIC "   \ (baschr set by PROCsetup)
1035     EQUB baschr
1040     EQUB 0
1041
1045 .basok
1050     LDA BRKV            \Load the current BRK vector
1055     LDX BRKV+1          \ into A and X.
1056
1060     CMP #newbrk MOD &100 \If this routine is already
```

```
1065     BNE ntsavd            \ set up, don't change BRKV.
1070     CPX #newbrk DIV &100
1075     BEQ saved
1076
1078 .ntsavd
1080     STA svbrkv            \It has not been set up
1085     STX svbrkv+1          \ already, so save old
1090     LDA #newbrk MOD &100 \ BRKV, and set up the new
1095     STA BRKV              \ one.
1100     LDA #newbrk DIV &100
1105     STA BRKV+1
1106
1110 .saved
1115     RTS
1190
1192 \ *** This is the new BRK handling routine  ***
1200 .newbrk
1205     PHA            \Save A and Y on 6502 stack
1210     TYA
1215     PHA
1216
1220     LDY #0         \Get error number
1225     LDA (&FD),Y
1226
1280     CMP #4         \If "Mistake", try new keywords
1285     BEQ mistak
1286
1400 .giveup
1410     PLA            \Restore A and Y from 6502 stack
1420     TAY
1430     PLA
1431
1440     JMP (svbrkv)  \Go to old BRK handler
1441
1490 \ *** If we get here, an error 4 ("Mistake") has  ***
1492 \ *** ocurred, so attempt to recognise one of the ***
1494 \ *** command keywords in the table.              ***
1500 .mistak
1510     LDA #keytab MOD &100 \Get start of keyword table
1520     STA &39              \ into (&39)
1530     LDA #keytab DIV &100
1540     STA &3A
1541
1550     LDY &A         \Set (&37) to point to character
1560     DEY            \ before PTRA. It will then point
1570     TYA            \ to the first non-space character
1580     CLC            \ of the statement.
1590     ADC &B
1600     STA &37
1610     LDA &C
1620     ADC #0
1630     STA &38
```

```
1631
1640     JSR nxtwrd    \Call the command line interpreter
1641
1650     BCS giveup    \Exit if no match
1651
1660     DEY           \Adjust the offset of PTRA so that
1665     TYA           \ it points to the first charcter
1670     CLC           \ after the keyword just recognised.
1675     ADC &A
1680     STA &A
1681
1685     PLA           \Discard saved A and Y from stack
1690     PLA
1691
1695     PLA           \Discard RTI information from the
1700     PLA           \ 6502 stack. This is automatically
1705     PLA           \ pushed by the BRK instruction.
1706
1710     PLA           \Discard return addr (of routine
1715     PLA           \ to check for "=") from stack
1716
1720     JMP (&0037)   \Execute the command
1721
1900 \ ***         Command Line Interpreter            ***
1902 \ ***  On entry, (&37) should point to the first  ***
1904 \ ***   char of the word in the program to be     ***
1906 \ ***   recognised. (&39) should point to the     ***
1908 \ ***   start of the keyword table.               ***
1910 \ ***  On exit;                                   ***
1912 \ ***   if C is set, a match was not made         ***
1914 \ ***   if C is clear, the action addr is in      ***
1916 \ ***   &37,38, so that JMP (&37) will call it.   ***
1917 \ ***   Y contains the length of the word.        ***
1918 \ ***                                             ***
1920 \ ***  No abbreviations are allowed.              ***
1922
2135 .nxtwrd
2140     LDY #0        \Beginning of words
2141
2150     LDA (&39),Y   \If no word, this is the end of the
2160     BEQ nomtch    \ table, so no match was made.
2161
2170     CMP (&37),Y   \If the chars do not match,
2180     BNE difrnt    \ try the next keyword.
2181
2190 .nextch
2200     INY           \Get the next character:
2210     LDA (&39),Y   \ if it is the end of the keyword,
2220     BEQ getadr    \ then get its addr, and jump there.
2221
2230     CMP (&37),Y   \If the chars match,
2240     BEQ nextch    \ try the next one.
```

```
2241
2250 .difrnt
2260     INY            \This keyword is not the right one,
2270     LDA (&39),Y    \ so look for the end of it.
2280     BNE difrnt
2281
2290     INY            \Set the base pointer at (&39) to
2300     INY            \ the start of the next keyword in
2310     TYA            \ the table (i.e. 3 bytes past the
2320     SEC            \ end of this keyword, to allow
2330     ADC &39        \ for the address).
2340     STA &39
2350     LDA &3A
2360     ADC #0
2370     STA &3A
2371
2380     JMP nxtwrd     \Try the next keyword in the table
2381
2400 .getadr
2410     INY            \The correct keyword has been
2415     LDA (&39),Y    \ matched, so put its execution
2420     STA &37        \ addr in (&37).
2425     INY
2430     LDA (&39),Y
2435     STA &38
2436
2440     DEY            \Adjust Y so it contains the length
2445     DEY            \ of the recognised word.
2446
2450     CLC            \Flag "Match OK", and exit
2455     RTS
2456
2460 .nomtch
2465     SEC            \Flag "No match", and exit
2470     RTS
2490
2494 \ *** Keyword table. The format of this table ***
2496 \ ***  is; Keyword, zero byte, action addr    ***
2498 \ *** A 0 keyword entry marks end of table.   ***
2499
2500 .keytab
2505     EQUS "BEEP"    \Keyword,
2510     EQUB 0         \ zero byte,
2515     EQUW beep      \ action addr
2516
2520     EQUS "DUMP"
2525     EQUB 0
2530     EQUW dump
2531
2580     EQUB 0         \End of keyword table
2990
2992 \ *** BEEP - This command makes a beep by    ***
```

```
2994 \ ***  printing a BEL character (CHR$7)      ***
3000 .beep
3010     JSR chksda   \Ensure end of statement
3011
3020     LDA #7       \Print a beep
3030     JSR OSWRCH
3031
3035 .alldne
3040     JMP cont     \Continue execution
3090
3092 \ *** DUMP - This command lists the names of ***
3094 \ ***   all of the current active variables. ***
3100 .dump
3105     JSR chksda   \Ensure end of statement
3106
3110     LDA #ASC"A"-1 \Set first initial letter for
3120     STA &39       \ variable (allow for first INC)
3121
3125 .newltr
3130     INC &39       \Use the next initial letter
3131
3140     LDA &39       \If all the letters have been
3150     CMP #ASC"z"+1 \ used up, go to next statement
3160     BCS alldne
3161
3170     ASL A         \Point (&3A) at the right place
3180     STA &3A       \ in the variable link table
3190     LDA #4        \ in the top half of page 4
3200     STA &3B
3201
3205 .newptr
3210     LDY #1        \Get the MSB of the pointer to the
3220     LDA (&3A),Y   \ next variable in the linked list.
3221
3230     BEQ newltr    \If it is 0, we have found the end,
3231                   \ so try another initial letter.
3232
3240     TAX           \Using X as a temp for the MSB,
3245     DEY           \ get the LSB of the pointer to the
3250     LDA (&3A),Y   \ next variable in the list, and
3255     STA &3A       \ set (&3A) to point to this
3260     STX &3B       \ variable.
3261
3262     LDA &39       \Print initial letter of variable
3264     JSR pchar     \ name (not stored in the list)
3265
3266     LDY #2        \Point at 1st stored char
3267
3268 .nxtchr
3270     LDA (&3A),Y   \Get the char in the name. If it
3275     BEQ namend    \ is the end of the name, exit.
3280     JSR pchar     \ Otherwise, print the char, and
```

```
3285     INY            \ go to the next one.
3290     BNE nxtchr     \ (Y never 0 here, so branch always)
3291
3295 .namend
3300     JSR pnewl      \Print a new line after the end of
3305     JMP newptr     \ the name, and try the next link.
8000 ]
8010 NEXT
8015 @%=0
8020 PRINT'"Code length =&"~P%-start%
8190
8200 PRINT'''''"** WARNING: Once assembled, the code"
8210 PRINT"generated by this program is not"
8220 PRINT"transferable between different BASICs"
8230 PRINT
8300 PRINT"Execute ""CALL &"~init""" to initialise."'
8310 END
8990
8992 REM *** Set up ROM entry points, allowing for ***
8993 REM ***  BASIC 1 and BASIC 2.                 ***
9000 DEFPROCsetup
9010 basic1$ = "BASIC"+CHR$0+"(C)1981 Acorn"+CHR$&A
9020 basic2$ = "BASIC"+CHR$0+"(C)1982 Acorn"+CHR$&A
9030 IF $&8009=basic1$ THEN PROCset1 :ENDPROC
9040 IF $&8009=basic2$ THEN PROCset2 :ENDPROC
9050 PRINT "NOT BASIC 1 OR 2"
9060 END
9290
9292 REM *** Set up BASIC 1 entry points           ***
9300 DEFPROCset1
9310 baschr = ASC"1":REM Used by init routine
9320 pchar  = &B571 :REM Print char in A: handle COUNT
9330 pnewl  = &BC42 :REM Print a CRLF, and zero COUNT
9340 chksda = &9810 :REM Check for statement delimiter
9350 cont   = &8B0C :REM Cont execution at next statement
9360 ENDPROC
9490
9492 REM *** Set up BASIC 2 entry points           ***
9500 DEFPROCset2
9505 baschr = ASC"2":REM Used by init routine
9520 pchar  = &B558 :REM Print char in A: handle COUNT
9525 pnewl  = &BC25 :REM Print a CRLF, and zero COUNT
9530 chksda = &9857 :REM Check for statement delimiter
9540 cont   = &8B9B :REM Cont execution at next statenemt
9550 ENDPROC
```

Note that the initialisation and setup routines are substantially the same as for the program in section 7.3 (although there are a few extra ROM routines). The program is longer than the last one, so it destroys the user defined function key area (this means that funny things might happen if you press BREAK, as it is function key 10). The command line interpreter in this program (lines 1500 on) replaces the simple check for a 'B' in the last one.

The keyword recogniser (lines 1900 to 2470) is a subroutine all by itself. It uses a keyword table (lines 2500 to 2580) with each entry in the following format:

keyword characters
a zero byte to terminate the keyword
the action address of the keyword (2 bytes)

The end of the table is marked by the first character of the keyword being a zero byte.

The keyword recogniser is entered with the address of the table in &38,&39 and the address of the keyword to be recognised in &37,&38. If the keyword is recognised, the action address is put into &37,&38, the length of the recognised word is left in Y, and the carry flag cleared. If the keyword is not recognised, the carry flag is set.

Sending the address of the table in this manner allows more than one routine to use the same recogniser, with different tables. This means that it could also be used if new functions are being added as well.

The general operation of the keyword recogniser is as follows:

1 If the first byte of the name is a zero, the end of the table has been reached without a match, so exit with the carry flag set.

2 Compare the keyword in the table against the word in the program. If they both match until the zero at the end of the word in the table is found, get the action address of the keyword.

3 If any characters did not match, move the table pointer up to point to the next entry, and go back to stage 1 to try to match the next one.

When the keyword recogniser has returned, PTRA is updated to point to the first character after the keyword (lines 1660 to 1680).

This allows the routine for the keyword to continue from there, to get anything it needs from the text (or to just check for the end of the statement).

The variable dump routine works in a similar way to the BASIC one in section 3.1.2, but it doesn't print out their values.

7.5 A renumber utility

The RENUMBER command in BASIC is very limited; it only allows you to renumber the whole of your program. This is OK for small programs, but larger programs usually consist of a number of PROC and FN definitions, and it is very easy to loose track of these if they don't start on, say, 1000 boundaries. Using BASIC's blanket renumber on programs such as these will lose this structure completely.

This section describes how to add a new command to allow selected areas of the program to be renumbered. It is less than 512 bytes long, and so will fit in any 2 spare pages in memory (the user defined character and function key pages, perhaps).

Once the program has been assembled, and initialised by CALLing the start address, the new statement 'REN' has been added.

REN L, U; S, I

will renumber the lines in the program between L and U (inclusive) starting at S with an increment of I. All line numbers outside this range will be left unaltered. The GOTO and GOSUB line number references will be dealt with, in the same way as the BASIC RENUMBER command (in fact, the program JMPs into the RENUMBER code to do this!).

For example, if the following program was in memory:

```
  10 REM PROGRAM
 100 A=0
 101 B=0
 110 PROCthing
1000 DEFPROCthing
1010 ENDPROC
```

typing ‘REN 100,110;500,20’ would leave the program as:

```
  10 REM PROGRAM
 500 A=0
 520 B=0
 540 PROCthing
1000 DEFPROCthing
1010 ENDPROC
```

The following errors will be produced if the REN statement is misused:

REN syntax

This error is generated if the REN statement fails to find a comma or a semicolon separating its arguments where expected.

REN space

This error is generated if there is not enough room for the pile of old line numbers the REN statement needs to put on the TOP of the program. This is similar to the ‘RENUMBER space’ error (a fatal error).

REN range

An attempt was made to renumber the program such that the new lines would be out of sequence. In the above example, if ‘REN 1000,1010;1,2’ was typed this error would be generated.

REN type

A string was used as the argument to the REN statement (floating point numbers will be converted to integer if necessary).

EQU has not been used in this program, so it will work without modification with either BASIC 1 or BASIC 2 (although it looks a bit messy).

```
   10 REM ***     Selective renumber utility  ***
   12 REM
   14 REM            M D Plumbley      1984
   16 REM
   18 REM This program traps the BRK vector. If the error
   20 REM  number is 4 ("Mistake") then the command line
   22 REM  interpreter will test for the new command "REN",
```

```
  24 REM  and execute it if it is.
  26 REM
  28 REM   REN L, U; S, I   will renumber lines L to U of a
  30 REM  program, starting at S, with an increment of I.
  32 REM
  34 REM The code is assembled into the user key/char
  36 REM  space. This can be changed by changing line 900
  38 REM
  40 REM The EQU directive is not used in this program, and
  42 REM  it will work without modification on either
  44 REM  BASIC1 or BASIC2 machines.
  46 REM
  99
 100 PROCsetup      :REM Set up correct ROM entry points
 490
 495 REM *** OS routines and vectors ***
 550 BRKV   = &0202
 590
 600 worksp = &0070           :REM Workspace area
 605 svbrkv = worksp          :REM BRK vector save slot
 610 lower  = worksp+&2       :REM Lower renumber limit
 615 upper  = worksp+&4       :REM Upper renumber limit
 620 start  = worksp+&6       :REM Start line number
 625 number = worksp+&8       :REM Next renumber number
 630 line   = worksp+&A       :REM Pointer to line in prog.
 635 pile   = worksp+&C       :REM Ptr. to line no. pile
 640 newnum = worksp+&E       :REM Line no. to be used
 690
 695 REM *** BASIC system variables ***
 700 himem = &0006
 705 top   = &0012
 710 page  = &0018
 715 count = &001E
 720 inta  = &002A            :REM Integer accumulator
 725
 750 renum = 0                :REM To stop "No such var."
 799
 900 start% = &0B00 :REM User key/char
 905
 910 FOR opt% = 0 TO 3 STEP 3
 920 P% = start%
 950 [OPT opt%
1000 .init
1005     LDA &8015            \Test that the correct
1010     CMP #baschr          \ version of BASIC is
1015     BEQ basok            \ in the ROM.
1020
1025     BRK                  \If it isn't, print an
1030 ]?P%=60:P%=P%+1          :REM error message
1035 $P%="Not BASIC ":P%=P%+LEN$P%
1040 ?P%=baschr:P%=P%+1
1045 ?P%=0:P%=P%+1:[OPTopt%
```

```
1050
1055 .basok
1060     LDA BRKV             \Load the current BRK vector
1065     LDX BRKV+1           \ into A and X.
1070
1075     CMP #newbrk MOD &100 \If this routine is already
1080     BNE ntsavd           \ set up, don't change BRKV.
1085     CPX #newbrk DIV &100
1090     BEQ saved
1095
1100 .ntsavd
1105     STA svbrkv           \It has not been set up
1110     STX svbrkv+1         \ already, so save old
1115     LDA #newbrk MOD &100 \ BRKV, and set up the new
1120     STA BRKV             \ one.
1125     LDA #newbrk DIV &100
1130     STA BRKV+1
1135
1140 .saved
1145     RTS
1190
1192 \ *** This is the new BRK handling routine  ***
1200 .newbrk
1205     PHA           \Save A and Y on 6502 stack
1210     TYA
1215     PHA
1220
1225     LDY #0        \Get error number
1230     LDA (&FD),Y
1235
1240     CMP #4        \If "Mistake", try new keywords
1245     BEQ mistak
1250
1400 .giveup
1405     PLA           \Restore A and Y from 6502 stack
1410     TAY
1415     PLA
1420
1425     JMP (svbrkv)  \Go to old BRK handler
1430
1490 \ *** If we get here, an error 4 ("Mistake") has  ***
1492 \ *** ocurred, so attempt to recognise one of the ***
1494 \ *** command keywords in the table.              ***
1500 .mistak
1505     LDA #keytab MOD &100 \Get start of keyword table
1510     STA &39              \ into (&39)
1515     LDA #keytab DIV &100
1520     STA &3A
1525
1530     LDY &A        \Set (&37) to point to character
1535     DEY           \ before PTRA. It will then point
1540     TYA           \ to the first non-space character
```

```
1545     CLC           \ of the statement.
1550     ADC &B
1555     STA &37
1560     LDA &C
1565     ADC #0
1570     STA &38
1575
1580     JSR nxtwrd    \Call the command line interpreter
1585
1590     BCS giveup    \Exit if no match
1595
1600     DEY           \Adjust the offset of PTRA so that
1605     TYA           \ it points to the first charcter
1610     CLC           \ after the keyword just recognised.
1615     ADC &A
1620     STA &A
1625
1630     PLA           \Discard saved A and Y from stack
1635     PLA
1640
1645     PLA           \Discard RTI information from the
1650     PLA           \ 6502 stack. This is automatically
1655     PLA           \ pushed by the BRK instruction.
1660
1665     PLA           \Discard return addr (of routine
1670     PLA           \ to check for "=") from stack
1675
1680     JMP (&0037)   \Execute the command
1685
1690
1990 \ *** This is the command line interpreter bit ***
1992
2000 .nxtwrd
2005     LDY #0        \Beginning of words
2010
2015     LDA (&39),Y   \If no word, this is the end of the
2020     BEQ nomtch    \ table, so no match was made.
2025
2030     CMP (&37),Y   \If the chars do not match,
2035     BNE difrnt    \ try the next keyword.
2040
2045 .nextch
2050     INY           \Get the next character:
2055     LDA (&39),Y   \ if it is the end of the keyword,
2060     BEQ getadr    \ then get its addr, and jump there.
2065
2070     CMP (&37),Y   \If the chars match,
2075     BEQ nextch    \ try the next one.
2080
2085 .difrnt
2090     INY           \This keyword is not the right one,
2095     LDA (&39),Y   \ so look for the end of it.
```

```
2100     BNE difrnt
2105
2110     INY           \Set the base pointer at (&39) to
2115     INY           \ the start of the next keyword in
2120     TYA           \ the table (i.e. 3 bytes past the
2125     SEC           \ end of this keyword, to allow
2130     ADC &39       \ for the address).
2135     STA &39
2140     LDA &3A
2145     ADC #0
2150     STA &3A
2155
2160     JMP nxtwrd    \Try the next keyword in the table
2165
2170 .getadr
2175     INY           \The correct keyword has been
2180     LDA (&39),Y   \ matched, so put its execution
2185     STA &37       \ addr in (&37).
2190     INY
2195     LDA (&39),Y
2200     STA &38
2205
2210     DEY           \Adjust Y so it contains the length
2215     DEY           \ of the recognised word.
2220
2225     CLC           \Flag "Match OK", and exit
2230     RTS
2235
2240 .nomtch
2245     SEC           \Flag "No match", and exit
2250     RTS
2490
2494 \ *** Keyword table. The format of this table ***
2496 \ ***  is; Keyword, zero byte, action addr    ***
2498 \ *** A 0 keyword entry marks end of table.   ***
2499
2500 ]
2505 keytab = P%
2510 $P% = "REN" :P%=P%+LEN$P%
2515 ?P% = 0     :P%=P%+1
2520 !P% = renum :P%=P%+2
2525 ?P% = 0     :P%=P%+1     :REM end of table
2600 [OPT opt%
2790
2792 \ *** This prints a REN syntax error ***
2800 .nocom                  \ If "," missing, or ";"
2805 .noscol                 \ missing, generate a
2810     BRK                 \ "REN syntax" error
2815 ]
2820 ?P%=&60:P%=P%+1
2825 $P%="REN syntax":P%=P%+LEN$P%
2830 ?P%=0:P%=P%+1
```

```
2835 [OPT opt%
2990
2992 \ *** REN - This command renumbers a selected ***
2994 \ ***  part of a program                      ***
3000 .renum
3005     JSR gtinta           \ Get the lower limit line
3010     LDA inta             \  number from the text at
3015     STA lower            \  PTRA, and save it in
3020     LDA inta+1           \  "lower". PTRB points to
3025     STA lower+1          \  the next item.
3030
3035     JSR getchb           \ Check for a comma at PTRB,
3040     CMP #ASC","          \  and error if it isn't.
3045     BNE nocom
3050
3055     JSR gtintb           \ Get the upper limit line
3060     LDA inta             \  number from the text at
3065     STA upper            \  PTRB, and save it in
3070     LDA inta+1           \  "upper".
3075     STA upper+1
3080
3085     JSR getchb           \ Check for a semicolon at
3090     CMP #ASC";"          \  PTRB, and error if it
3095     BNE noscol           \  isn't.
3100
3105     JSR gtintb           \ Get the start number for
3110     LDA inta             \  the renumbered section,
3115     STA start            \  and save it in "start".
3120     LDA inta+1
3125     STA start+1
3130
3135     JSR getchb           \ Check for a comma, and
3140     CMP #ASC","          \  error if it isn't.
3145     BNE nocom
3150
3155     JSR gtintb           \ Get the increment, leaving
3157                          \  leaving it in IntA.
3160
3165     JSR chksdb           \ Check for end of statement
3170
3200     JSR settop           \ Set TOP to the top of the
3202                          \  program, and set up the
3205     JSR setup            \  initial ptrs and numbers
3210
3490 \ ** Go through all the lines, piling up the     ***
3492 \ **  numbers, and checking for range.           ***
3500 .chklns
3505     LDY #0          \ If we're at the end of the
3510     LDA (line),Y    \  program, go on to renumber
3515     BMI renlns      \  the lines
3520
3525     STA (pile),Y    \ Otherwise, add the line
```

```
3530     INY               \  number to the pile on the
3535     LDA (line),Y      \  TOP of the program.
3540     STA (pile),Y
3545
3550     CLC               \ Add 2 to the pile pointer,
3555     LDA #2            \  to cover the new line just
3560     ADC pile          \  added to it. Save the LSB
3565     STA pile          \  of the pile pointer in X,
3570     TAX               \  as it will be needed to
3575     LDA pile+1        \  check against HIMEM.
3580     ADC #0
3585     STA pile+1
3590
3595     CPX himem         \ If the pile pointer is now
3600     SBC himem+1       \  above HIMEM, give a
3605     BCS noroom        \  "REN space" error.
3610
3615     JSR rngchk        \ Check the line range, and
3620     JSR nextln        \  move the pointer to the
3621                       \  next one, and go back to
3625     JMP chklns        \  do another.
3630
3635 .noroom               \ Generate a "REN space"
3640     BRK               \  error.
3645 ]?P%=&61:P%=P%+1
3650 $P%="REN space":P%=P%+LEN$P%
3655 ?P%=0:P%=P%+1
3660 [OPT opt%
3990
3992 \ ** Once the line range has been checked, and the **
3994 \ **  pile set up, come here to renumber the lines **
3996
4000 .renlns                    \ Re-set the line pointer and
4005     JSR setup              \  numbers.
4010
4015 .rnline                    \ If we're at the end of the
4020     LDY #0                 \  program, go on to resolve
4025     LDA (line),Y           \  the GOTO line references.
4030     BMI rsolve
4035
4040     JSR rngchk             \ Set up "newnum" to be the
4045                            \  new line number to be
4050     LDA newnum+1           \  used, and set the line
4055     STA (line),Y           \  number of the current line
4060     INY                    \  to it.
4065     LDA newnum
4070     STA (line),Y
4075
4080     JSR nextln             \ Move the line pointer to
4085                            \  point to the next line,
4090     JMP rnline             \  and jump back to renumber
4095                            \  the next one.
```

```
4100
4500 .rsolve                  \ Jump into RENUMBER to fix
4505     JMP rsvgot           \  the GOTO references.
4510
5989
5990 \ ** Set up current number to first,
5992 \            line pointer to PAGE+1,
5994 \            pile pointer to TOP
6000 .setup
6005     LDA start            \ Set the next number in the
6010     STA number           \  renumbered section to the
6015     LDA start+1          \  start number in the
6020     STA number+1         \  renumbered section.
6025
6030     LDA #1               \ Set the line pointer to
6035     STA line             \  point to the first line
6040     LDA page             \  at PAGE+1
6045     STA line+1
6050
6055     LDA top              \ Set the pile pointer to
6060     STA pile             \  the TOP of the program
6065     LDA top+1
6070     STA pile+1
6075
6080     LDA #0               \ Set the last number used to
6085     STA newnum           \  zero
6090     STA newnum+1
6092
6095     RTS                  \ Exit
6189
6190 \ ** Set "line" to point to next line         **
6200 .nextln
6205     LDY #2               \ Get the length byte of the
6210     LDA (line),Y         \  current line.
6212
6215     CLC                  \ Add the length of the line
6220     ADC line             \  to the line pointer.
6225     STA line
6230     BCC lineok
6235     INC line+1
6240 .lineok
6245     RTS                  \ Exit
6489
6490 \ ** Check range and set up newnum      **
6500 .rngchk
6505     LDY #1               \ Get the current line number
6510     LDA (line),Y         \  into X (LSB) and A (MSB)
6515     TAX
6520     DEY
6525     LDA (line),Y
6530
6535     CPX lower            \ If the current line is not
```

```
6540     SBC lower+1          \  under the lower limit, go
6545     BCS notund           \  to "notund"
6550
6555     LDA (line),Y         \ If it is, check that the
6560     CPX start            \  start line for the REN
6565     SBC start+1          \  section is above this
6570     BCC thisln           \  line. Otherwise, ...
6575
6580 .rngerr                 \ Generate a "REN range"
6585     BRK                  \ error
6590 ]?P%=&62:P%=P%+1
6595 $P%="REN range":P%=P%+LEN$P%
6600 ?P%=0:P%=P%+1
6605 [OPT opt%
6610
6615 .notund                 \ Check to see if the current
6620     LDA (line),Y         \  line number, which is
6625     CMP upper+1          \  not under the lower limit,
6630     BCC notovr           \  is also not over the upper
6635     BNE over             \  limit. If it is inside
6640     CPX upper            \  both these limits, go to
6645     BCC notovr           \  "notovr" to generate a new
6650     BEQ notovr           \  line number.
6655
6660 .over                   \ If the current line number
6665     CMP newnum+1         \  is over the upper limit,
6670     BCC rngerr           \  check that the last line
6675     BNE thisln           \  used was not above this
6680     CPX newnum           \  one. If it was, the last
6685     BCC rngerr           \  renumbered line number was
6690     BEQ rngerr           \  too big, so error.
6695
6700 .thisln                 \ If the current line number
6705     LDA (line),Y         \  is outside the REN limits,
6710     STA newnum+1         \  use the current line
6715     STX newnum           \  number as the new one, and
6720     RTS                  \  exit.
6725
6730 .notovr                 \ If the current line number
6735     CLC                  \  is inside the REN limits,
6740     LDA number           \  use "number" as the new
6745     STA newnum           \  line number, and add the
6750     ADC inta             \  increment to "number".
6755     STA number
6760
6765     LDA number+1         \ The AND is to make sure
6767     AND #&7F             \  that the line number never
6770     STA newnum+1         \  exceeds 32768. If it does,
6775     ADC inta+1           \  it will be lost off the
6780     STA number+1         \  end of the program.
6782
6785     RTS                  \ Exit
```

```
6790
6990 \ ** Get an integer from the text at PTRA **
7000 .gtinta
7005     JSR getnsa           \ Get a <numeric> or <string>
7010     JMP typchk           \  at PTRA, and check type.
7015
7017 \ ** Get an integer from the text at PTRB **
7020 .gtintb                  \ Get a <numeric> or <string>
7025     JSR getnsb           \  at PTRB.
7027
7030 .typchk                  \ If it was a string, give a
7035     BEQ msmtch           \  "REN type" error
7040
7045     BPL noconv           \ If it was real (type -ve),
7050     JSR cftoi            \  convert it to integer.
7052
7055 .noconv
7060     RTS                  \ Exit.
7065
7070 .msmtch                  \ Generate a "REN type"
7075     BRK                  \  error.
7080 ]?P%=&63:P%=P%+1
7085 $P%="REN type":P%=P%+LEN$P%
7090 ?P%=0:P%=P%+1
8000
8010 NEXT
8015 @%=0
8020 PRINT'"Code length =&"~P%-start%
8190
8200 PRINT'''''"** WARNING: Once assembled, the code"
8210 PRINT"generated by this program is not"
8220 PRINT"transferable between different BASICs"
8230 PRINT
8300 PRINT"Execute ""CALL &"~init""" to initialise."'
8310 END
8990
8992 REM *** Set up ROM entry points, allowing for ***
8993 REM ***  BASIC 1 and BASIC 2.                 ***
9000 DEFPROCsetup
9010 basic1$ = "BASIC"+CHR$0+"(C)1981 Acorn"+CHR$&A
9020 basic2$ = "BASIC"+CHR$0+"(C)1982 Acorn"+CHR$&A
9030 IF $&8009=basic1$ THEN PROCset1 :ENDPROC
9040 IF $&8009=basic2$ THEN PROCset2 :ENDPROC
9050 PRINT "NOT BASIC 1 OR 2"
9060 END
9290
9292 REM *** Set up BASIC 1 entry points          ***
9300 DEFPROCset1
9305 baschr = ASC"1":REM Used by init routine
9310 cftoi  = &A3F2 :REM Convert floating point to integer
9315 chksdb = &980B :REM Check statement delimiter at PTRB
9320 getchb = &8A13 :REM Get character at PTRB
```

```
9325 getnsb = &9B03 :REM Get <numeric> or <string> at PTRB
9330 getnsa = &9AF7 :REM Get <numeric> or <string> at PTRA
9535 settop = &BE88 :REM Set up TOP, check "Bad program"
9340 rsvgot = &8FAD :REM Resolve RENUMBERed GOTOs
9345 ENDPROC
9490
9492 REM *** Set up BASIC 2 entry points          ***
9500 DEFPROCset2
9505 baschr = ASC"2":REM Used by init routine
9510 cftoi  = &A3E4 :REM Convert floating point to integer
9515 chksdb = &9852 :REM Check statement delimiter at PTRB
9520 getchb = &8A8C :REM Get character at PTRB
9525 getnsb = &9B29 :REM Get <numeric> or <string> at PTRB
9530 getnsa = &9B1D :REM Get <numeric> or <string> at PTRA
9535 settop = &BE6F :REM Set up TOP, check "Bad program"
9540 rsvgot = &900D :REM Resolve RENUMBERed GOTOs
9545 ENDPROC
```

The initialisation routine, BRK handler, and keyword recogniser used by this program (lines 1000 to 2250) are the same as used in the program in section 7.4. The keyword table (lines 2500 to 2525) contains the single entry ‘REN’.

The general operation of the renumber utility, once recognised, is as follows:

1 The rest of the line after the ‘REN’ is decoded (lines 3000 to 3165). The keyword recogniser leaves PTRA pointing to the first character after the keyword, so this is used to get the first integer. The succeeding characters and integers are read in from PTRB, as this is advanced leaving PTRA still pointing to the first character after the ‘REN’.

2 The old line numbers are piled up above the program, from TOP onwards (lines 3500 to 3625). Also, each line is checked to make sure that the range of the renumbered lines does not overlap with the lines which will not be renumbered. This check is carried out by the routine ‘rngchk’ (which also calculates the new line number, but that is not used at this stage).

3 The lines are then renumbered (lines 4000 to 4095), using the routine ‘rngchk’ to calculate the new line number. This is not done at stage 2, in case there was not enough room for the pile of line numbers; otherwise, the program would be left half-renumbered, with no GOTO references resolved.

4 The GOTO and GOSUB references are resolved. This part is in fact done by the routine in the ROM which is used by the BASIC RENUMBER command. It scans through the program, looking for line number tokens (section 2.3.2). If it finds one, it searches through the pile of old line numbers on top of the program, at the same time keeping track of the corresponding new line number in the program. When it matches the line numbers, it changes the tokenised line number to the new one. If it couldn't match them, it prints the 'Failed at xxx' message, before continuing.

The 'rngchk' routine is used both in stages 2 and 3. It decides whether the current line number is inside the range to be renumbered or not, and generates 'newnum' to be either the current line number, or a new renumbered line number accordingly. If it finds that the renumbering would cause a line number overlap, it generates a 'REN range' error.

The 'getinta' and 'getintb' routines get an integer from the line of text, leaving it in IntA (&2A to &2D). If the argument is in fact a string, a 'REN type' error will be generated. If the argument is a floating point number, it will convert it to an integer. The routine to get a <numeric> or <string> at PTRA will first copy PTRA into PTRB, and then get the <numeric> or <string> at PTRB (thus leaving PTRA unchanged). See chapter 10 for more details of these expression evaluation routines.

With the mechanisms described in this chapter, any number of new statements can be added (provided there is enough memory to keep them all in). The next chapters describe how other errors can be trapped, as well as the 'Mistake' error.

8 Overlaying Procedures

Lack of memory can be a very restrictive and annoying problem with large programs. One way of getting round this is to use several smaller programs, and CHAIN them together (like the ‘Welcome’ cassette). However, this RUNs each program which is loaded in, so all the variables (apart from the resident integers) are lost.

Another method is to ‘overlay’ FNs and PROCs. If the program consists of a number of large sections, which will not be in memory at the same time as one another, these sections can be loaded in on top of each other when one is required. Since only one of the sections will be active at any particular time, the same memory can be used for all of them.

By intercepting the ‘No such FN/PROC’ error, an overlay file can be loaded in, and executed as if it was a normal FN or PROC. When the FN or PROC has finished, the memory that it loaded into is free for another call. This sort of overlaying is more useful on a system with discs, because of its random access ability; but it can be used with cassettes as well if the order in which the overlay files will be required is known (so that they can be saved in that order on the tape).

This chapter describes how to overlay FNs and PROCs, JMPing back in to BASIC to continue when the file has been loaded.

8.1 The ‘No such FN/PROC’ error

This error (error number 29) is generated by the FN/PROC handler when it failed to find the definition of the FN or PROC in the program. See section 5.3 for the operation of the FN/PROC handler. The sequence of actions taken when the FN/PROC handler comes across an undefined call is as follows:

1 The 6502 stack, from &1FF to the item on top of the stack, is saved on the BASIC STACK. The 6502 stack pointer is saved as the byte on top of the BASIC stack, so that the correct number of bytes can be retrieved after the call. After saving, the 6502 stack pointer is re-set to &1FF.

2 The FN or PROC token is saved as the first item on the 6502 stack, at &1FF, so that ENDPROC or the '=' statement know which type of call they are in. The FN token is &A4, and the PROC token is &F2.

3 PTRA is saved on the 6502 stack, from &1FE to &1FC. The stack pointer now points to &1FB (at the next free byte).

4 If there was no name after the FN/PROC token, a 'Bad call' error is generated. Otherwise, the FN/PROC handler searches through the list of already used FNs or PROCs for the name.

5 If it wasn't found in the list (which it won't be, if it is not in the program), the FN/PROC handler searches through the program for the definition. When it doesn't find it, it restores the base of PTRA from the 6502 stack, so that ERL will be set up properly by the BASIC error handler, and generates a 'No such FN/PROC' error.

When this error ocurrs, the prevailing conditions on entry to the BRK handler are:

&FD,&FE	points to the error number (29)		
6502 stack:	&1FB	RTI info.	3 bytes
	&1FE	PTRA offset	1 byte
	&1FF	FN/PROC token	1 byte

BASIC STACK contains old 6502 stack.

&37,&38	points 1 byte before the FN/PROC token
&39	length of name (+1 for token)

The FN/PROC can be re-entered to force it to use an overlayed file as the FN or PROC it was looking for, but first the 6502 stack must be restored to the state immediately before the error was generated. The 3 bytes of RTI information must be pulled from the stack, and the base of PTRA must be pushed back on (&B first, then &C).

At this point the overlay file can be loaded. When the overlay file is in memory, the FN/PROC handler can be re-entered, as if the overlay is a FN or PROC which it has just found.

To re-enter the FN/PROC handler, set the base of PTRA (in &B,&C) to point to the first character which would be after the name of the FN/PROC in the definition, and JMP to &B223 (BASIC1) or &B1F4 (BASIC2).

Jumping to this address will continue with the FN/PROC handler, and the name will not be added to the list of used FNs or PROCs. If the name had been added to the list, difficulties would arise when the overlay had been finished with; the FN/PROC handler would still think that it knew where the overlayed FN or PROC was, but the memory may have already been used by a different overlay file.

8.2 Static overlaying

A very simple method of overlaying a FN or PROC is to load a file into a fixed position in memory (hence 'static') whenever a 'No such FN/PROC' error is generated.

The routine in this section will load the file 'OVERLAY' into memory at &6000 (this can be changed by altering line 600), and then re-enter the FN/PROC handler to use this file as the FN or PROC which could not be found.

The 'OVERLAY' file should be saved as if it is a normal BASIC program: it should *not* contain the 'DEF PROCname' (but it must have the 'ENDPROC' or '=' statement). If parameters are to be passed to it, the '(' should be the first character on the first line of the program. For example, the following overlay file will print the SIN of the number passed to it:

```
10(number)
20PRINT SIN(number)
30ENDPROC
```

If this program is saved as the file 'OVERLAY', any unrecognised FN or PROC call will be passed to it. For example, 'PROCFRED(PI/2)' will print '1'.

This overlay routine cannot tell the difference between FNs and PROCs; it will load the file ‘OVERLAY’ whenever the error is generated. So, if the file is saved as above, ‘X=FNA(3)’ will give a ‘No PROC’ error, when it finds the ‘ENDPROC’ statement on the end of what it thinks is a FN.

If the overlay does not need any parameters, the first character on the first line could be the start of the first statement, or a space.

```
   4 REM This is a simple program to overlay procedures.
   6 REM
   8 REM          M D Plumbley      1984
  10 REM
  12 REM Once this is initialised, if a FN or PROC is not
  14 REM  found in a program, generating the
  16 REM  "No such FN/PROC" error, then the file called
  18 REM  "OVERLAY" will be loaded from disc, and
  20 REM  executed.
  22 REM
  24 REM The overlay file should not contain the name of
  26 REM  the PROC or FN, but any parameters should be
  28 REM  inside brackets on the first line of the file.
  30 REM  If used, the open bracket must be the first
  32 REM  character on the first line of the file.
  90 REM
  95
 100 PROCsetup      :REM Set up correct ROM entry points
 390
 395 REM *** OS vectors ***
 400 brkv   = &0202
 410 oldbrk = !brkv AND &FFFF
 490
 495 REM *** OS routines ***
 500 oscli = &FFF7
 590
 600 ldslot = &6000 :REM Area to load overlay into
 799
 900 start% = &0C00 :REM Assemble into user char space
 905
 910 FOR opt% = 0 TO 3 STEP 3
 920 P% = start%
 950 [OPT opt%
 960
1000 .newbrk
1005     PHA           \Save A and Y on 6502 stack
1010     TYA
1015     PHA
1020
1025     LDY #0        \Get error number
1030     LDA (&FD),Y
```

```
1035
1040     CMP #29       \If "No such FN/PROC", go
1045     BEQ noproc    \ to overlay routine.
1050
1055 .giveup           \Otherwise, restore A and Y and go
1060     PLA           \ to the default BRK handler.
1065     TAY
1070     PLA
1075     JMP oldbrk
1080
2000 .noproc
2005     PLA           \Remove the saved A and Y from the
2010     PLA           \ 6502 stack.
2015
2020     PLA           \Remove the RTI information from the
2025     PLA           \ 6502 stack.
2030     PLA
2035
2040     LDA &B        \Push the base of PTRA, ready for
2045     PHA           \ the return from the FN/PROC.
2050     LDA &C
2055     PHA
2060
2065     LDX #lodtxt MOD &100 \Tell the filing system to
2070     LDY #lodtxt DIV &100 \ load the overlay file
2075     JSR oscli
2080
2085     LDA #ldslot MOD&100+4 \Set PTRA to point to the
2090     STA &B                \ 1st char of the file
2095     LDA #ldslot DIV &100  \ (not CR, line num, or
2100     STA &C                \ length)
2105
2110     JMP prcfnd    \Continue with the FN/PROC handler
2115
2120 .lodtxt           \DFS command to load the overlay
2125 ]$P% = "LOAD OVERLAY ":P%=P%+LEN$P%
2130 $P% = STR$~ldslot :P%=P%+LEN$P%
2135 ?P% = &0D :P%=P%+1
2140
8000 NEXT
8010 @%=0
8020 PRINT'"Code length =&"~P%-start%
8030
8040 REM *** Link new routine in to BRK vector ***
8050 IF newbrk=oldbrk PRINT"Already set up":END
8060 brkv?0 = newbrk MOD &100
8070 brkv?1 = newbrk DIV &100
8080 END
8090
9000 REM *** Set up ROM entry points, allowing for ***
9010 REM ***          BASIC1 and BASIC2           ***
9020 DEFPROCsetup
```

```
9030 IF ?&8015=ASC"1" THEN PROCset1 ELSE PROCset2
9040 ENDPROC
9050
9300 REM *** Set up BASIC1 entry points          ***
9310 DEFPROCset1
9320 prcfnd = &B223 :REM Return to FN/PROC handler
9330 ENDPROC
9340
9500 REM *** Set up BASIC2 entry points          ***
9510 DEFPROCset2
9520 prcfnd = &B1F4 :REM Return to FN/PROC handler
9530 ENDPROC
```

The general operation of the routine is as follows:

1 If the error number is not 29, the default BRK handler is called (lines 1000 to 1080). If the error number is 29, the 3 bytes of RTI information are removed from the stack (as well as the 2 registers saved by the BRK handling routine at 1000 to 1015).

2 The base of PTRA is pushed back on the 6502 stack (lines 2040 to 2055), for the return when the call is finished.

3 The overlay file is loaded by sending the line ‘LOAD OVERLAY 6000’ to the Operating System Command Line Interpreter (OSCLI). This will be interpreted just as if a ‘*LOAD’ had been typed at the keyboard. Note the use of the hexadecimal version of the STR$ function (line 2130). This is in BASIC1 and BASIC2, but is not mentioned in the *User Guide*.

4 The base of PTRA is set to point to the fifth character of the file (at &6004). If the file has been entered as a BASIC program, the first character of the file will be a &0D, followed by a 2-byte line number, followed by the line length byte (see section 2.4 for the program storage format).

5 A JMP is made to re-enter the FN/PROC handler. It will then think that the call definition has been found, and that the base of PTRA points to the first character after the name in the definition. If this character is a ‘(, it will handle any parameters which are listed. It will then start executing statements in the file as if it was a proper FN or PROC.

8.3 Dynamic overlaying

The routine in the last section is a bit limited. It can't tell the difference between different FNs or PROCs, as it doesn't do any name checking; and it always loads into the same area of memory (which must be decided when it is assembled), so only one PROC or FN can operate at a time.

The routine in this section shows how FNs and PROCs can be recognised and loaded onto the BASIC STACK, completely invisible to the main program (except for the amount of memory required to load them). If there is not enough memory to load the FN or PROC, a 'No room' error will be generated. FNs and PROCs loaded like this can call others inside them to be overlayed, and these will also be loaded onto the STACK. The program in section 8.2 would just load the other overlay on top of the first one.

The exit from the FN or PROC is trapped by changing the token byte on the 6502 stack at &1FF, so that a 'No FN' or 'No PROC' error will be generated. This allows the overlayed file to be removed from the STACK when it is finished with, by intercepting these errors.

The overlay files are created in the same manner as the ones in section 8.2, with the '(' as the first character on the first line if necessary. However, the routine will check the name of the FN or PROC, and will load in 'P.fred' if 'PROCfred' is called, and 'F.fred' if 'FNfred' is called. Note that the operating system will treat upper and lower case letters as the same, so 'F.FRED' is the same as 'F.fred' as far is it is concerned.

```
10 REM *** Program to overlay PROCs and FNs   **
12 REM
14 REM     M D Plumbley        1984
16 REM
18 REM Once this is run, if a FN or PROC is not found in
20 REM  a program, generating the "No such FN/PROC"
22 REM  error, then the file with the same name
24 REM  as the FN or PROC will be loaded from disc (or
26 REM  tape). The P directory will be used for PROCs,
28 REM  the F directory for FNs.
30 REM
32 REM The FN or PROC will be loaded on the BASIC
```

```
  34 REM  STACK, and will be removed when it exits.
  36 REM
  38 REM The overlay file should not contain the name of
  40 REM  the PROC or FN, but any parameters should be
  42 REM  inside brackets on the first line of the file.
  44 REM  If used, the open bracket must be the first
  46 REM  character on the first line of the file.
  48 REM
  50 REM Before using with BASIC 1, all EQU directives
  52 REM  should be replaced by indirections:
  54 REM  "EQUB X"  => "]?P%=X:P%=P%+1:[OPTopt%"
  54 REM  "EQUW X"  => "]!P%=X:P%=P%+2:[OPTopt%"
  54 REM  "EQUD X"  => "]!P%=X:P%=P%+4:[OPTopt%"
  54 REM  "EQUS A$" => "]$P%=A$:P%=P%+LEN$P%:[OPTopt%"
  90 REM
  95
 100 PROCsetup   :REM Set up correct ROM entry points
 390
 395 REM *** OS vectors  ***
 400 brkv   = &0202
 410 oldbrk = !brkv AND &FFFF
 490
 495 REM *** OS routines ***
 500 oscli  = &FFF7
 505 osfile = &FFDD
 590
 690 REM *** BASIC registers ***
 700 stack = &0004
 705 inta  = &002A
 799
 800 parms = &0070 :REM Temp for number of parameters
 899
 900 start% = &0B00 :REM User defined character area
 905
 910 FOR opt% = 0 TO 3 STEP 3
 920 P% = start%
 950 [OPT opt%
 960
1000 .newbrk
1005     PHA               \Save A and Y on 6502 stack
1010     TYA
1015     PHA
1020
1025     LDY #0            \Get error number
1030     LDA (&FD),Y
1035
1040     CMP #29           \If "No such FN/PROC", go
1045     BEQ nofnpr        \ to overlay routine.
1047
1050     CMP #7            \If "No FN" see if it is a FN
1055     BEQ jnofn         \ to be thrown away.
1057
```

```
1060     CMP #13         \If "No PROC" see if it is a PROC
1065     BEQ jnoprc      \ to be thrown away.
1070
1075 .ospace
1080 .giveup             \Otherwise, restore A and Y and go
1085     PLA             \ to the default BRK handler.
1090     TAY
1095     PLA
1100     JMP oldbrk
1105
1110 .jnofn              \Jump to the "No FN" handler
1115     JMP nofn
1117
1120 .jnoprc             \Jump to the "No PROC" handler
1125     JMP noproc
1127
1990 \ *** If we get here, a FN or PROC is to be       ***
1992 \ ***  overlayed, after a "No such FN/PROC" error ***
2000 .nofnpr
2005     PLA             \Remove the saved A and Y from the
2010     PLA             \ 6502 stack.
2015
2020     PLA             \Remove the RTI information from the
2025     PLA             \ 6502 stack.
2030     PLA
2035
2040     LDA &B          \Push the base of PTRA, ready for
2045     PHA             \ the return from the FN/PROC.
2050     LDA &C
2055     PHA
2060
2065     LDY &39         \If the length of the name of the
2070     CPY #9          \ FN/PROC, with the token, is > 8,
2075     BCS giveup      \ it is too big to be a filename.
2080
2085     LDA #&0D        \Put a CR on the end of the
2090     STA filnam+1,Y  \ area, ...
2095
2100 .txnmlp             \ and transfer the name from the
2105     LDA (&37),Y     \ text into the filename area.
2110     STA filnam,Y
2115     DEY
2120     BNE txnmlp
2125
2130     LDX #ASC"P"     \If the token on the front of the
2135     CMP #&F2        \ name (the last byte transfered)
2140     BEQ proc        \ was a PROC token, put a "P" on
2145     LDX #ASC"F"     \ the front of the filename;
2150 .proc               \ otherwise use an "F".
2155     STX filnam
2160
2165     LDA #ASC"."     \Put a "." between the P/F and the
```

```
2170      STA filnam+1     \ FN/PROC name.
2175
2180      LDX #pblock MOD &100       \Call OSFILE to find
2185      LDY #pblock DIV &100       \ the length of the
2190      LDA #5                     \ file.
2195      JSR osfile
2200
2205      CMP #1           \If it didn't exist, jump to the
2210      BNE giveup       \ default error handler.
2215
2220      LDA stack        \Save the BASIC STACK pointer in
2225      STA inta         \ IntA, and move the STACK pointer
2230      SEC              \ down ready to load the overlay,
2235      SBC pblock+&0A   \ by subtracting the length of the
2240      STA stack        \ file from it. The file length
2245      STA pblock+2     \ is returned by OSFILE 5 in
2250                       \ pblock+&A and pblock+&B.
2255      LDA stack+1
2260      STA inta+1       \ A copy of the new stack pointer
2265      SBC pblock+&0B   \ is loaded into pblock+2 and
2270      STA stack+1      \ pblock+3, to tell OSFILE &FF
2275      STA pblock+3     \ where to load the file when it
2277                       \ is called.
2280      BCC ospace       \ If the STACK wrapped round,
2282                       \ give an error.
2285
2290      JSR pushi        \Push the old STACK pointer on
2292                       \ the STACK.
2295
2300      LDA #0           \Set the "addr" flag for OSFILE to
2305      STA pblock+6     \ load the file at the given addr
2310
2315      LDX #pblock MOD &100       \Call OSFILE to load
2320      LDY #pblock DIV &100       \ the overlay file into
2325      LDA #&FF                   \ the space allocated
2330      JSR osfile                 \ on the STACK.
2335
2340      LDA stack        \Set the base of PTRA to point to
2345      CLC              \ the first character in the BASIC
2350      ADC #8           \ file (4 up to miss over IntA,
2355      STA &B           \ and another 4 up to miss the
2360      LDA stack+1      \ &0D, line number, and length
2365      ADC #0           \ byte as before).
2370      STA &C
2375
2380      LDA filnam       \Set the FN/PROC identifier byte
2385      STA &1FF         \ on the stack to a "P" or "F"
2390
2395      JMP prcfnd       \Jump into the FN/PROC handler.
2990
3000  .pblock              \OSFILE parameter block
3005      EQUW filnam
```

```
3010     EQUD 0
3015     EQUD 0
3020     EQUD 0
3025     EQUD 0
3030     EQUB 0
3032
3035 .filnam            \Filename area (max 9 characters)
3040     EQUS "123456789"
3045     EQUB &0D
3990
3992 \ ** No FN error   **
4000 .nofn
4005     LDA &1FF         \If the item on the stack was not
4010     CMP #ASC"F"      \ left by the overlay routine,
4015     BNE jgivup       \ there isn't a FN on the STACK.
4017
4020     CPX #&F5         \If the 6502 stack pointer wasn't
4025     BNE jgivup       \ &F5, we're not in a FN.
4027
4030     JSR getnsa       \Get the value of the FN following
4035     JSR chksdb       \ the "=", check end of statement,
4040     JMP doret        \ and jump to do the FN return.
4045
4090 \
4100 .jgivup
4105     JMP giveup       \Jump to the old BRK handler
4110
4990 \ ** No PROC error **
5000 .noproc
5005     LDA &1FF         \If the item on the stack was not
5010     CMP #ASC"P"      \ left by the overlay routine,
5015     BNE jgivup       \ there isn't a PROC on the STACK.
5020
5025     CPX #&F5         \If the 6502 stack pointer wasn't
5030     BNE jgivup       \ &F5, we're not in a PROC.
5032
5035     JSR chksda       \Check end of statement after the
5036                      \ "ENDPROC".
5037
5040 .doret
5045     PLA              \Remove the saved A and Y from the
5050     PLA              \ 6502 stack.
5055
5060     PLA              \Remove the RTI information from
5065     PLA              \ the 6502 stack
5070     PLA
5075
5080     PLA              \Remove the return addr to the
5085     PLA              \ FN/PROC handler.
5090
5095     PLA              \Restore PTRB
5100     STA &1A
```

```
5105     PLA
5110     STA &19
5115     PLA
5120     STA &1B
5125
5130     PLA               \If there were no parameters,
5135     BEQ noparm        \ don't restore any.
5140
5145     STA parms         \Otherwise, restore the saved
5150 .doparm              \ value of each parameter by
5155     JSR popi1         \ popping the variable descriptor
5160     JSR poppar        \ block and value from the BASIC
5165     DEC parms         \ stack.
5170     BNE doparm
5175
5180 .noparm
5185     PLA               \Restore PTRA
5190     STA &C
5195     PLA
5200     STA &B
5205     PLA
5210     STA &A
5215
5220     LDY #0            \Restore the BASIC stack pointer
5225     LDA (stack),Y     \ to the value it was before the
5230     TAX               \ FN or PROC was loaded onto it:
5235     INY               \ this had been pushed on the
5240     LDA (stack),Y     \ STACK when the file was loaded.
5245     STX stack
5250     STA stack+1
5255
5260     LDY #0            \Restore the 6502 stack from the
5265     LDA (stack),Y     \ BASIC STACK. The first byte
5270     TAX               \ gives the old value of the 6502
5275     TXS               \ S register, the rest of the
5280 .txstk               \ bytes are the actual stack
5285     INY               \ contents.
5290     INX
5295     LDA (stack),Y
5300     STA &100,X
5305     CPX #&FF
5310     BNE txstk
5315
5320     TYA               \Move the STACK pointer up to
5325     ADC stack         \ remove the 6502 stack contents
5330     STA stack         \ from it.
5335     BCC stkok
5340     INC stack+1
5345 .stkok
5347
5350     LDA &27           \Set the 6502 flags according to
5352                       \ &27 (in case we're in a FN).
```

```
5253
5355     RTS              \Exit
9000 ]
9010 NEXT
9020 @%=0
9030 PRINT'"Code length =&"~P%-start%
9040
9045 REM *** Link new routine in to BRK vector ***
9050 IF newbrk=oldbrk PRINT"Already set up":END
9060 brkv?0 = newbrk MOD &100
9070 brkv?1 = newbrk DIV &100
9075 END
9080
9500 REM *** Set up ROM entry points, allowing for ***
9510 REM ***           BASIC1 and BASIC2          ***
9520 DEFPROCsetup
9530 IF ?&8015=ASC"1" THEN PROCset1 ELSE PROCset2
9540 ENDPROC
9550
9600 REM *** Set up BASIC1 entry points           ***
9610 DEFPROCset1
9615 prcfnd = &B223 :REM Return to FN/PROC handler
9620 pushi  = &BDAC :REM Push IntA on the BASIC STACK
9625 popi1  = &BE23 :REM Pop &37-&3A from the STACK
9630 poppar = &8C5B :REM Pop parameter value from STACK
9635 getnsa = &9AF7 :REM Get <numeric> or <string>
9640 chksda = &9810 :REM Check end of statement (PTRA)
9645 chksdb = &980B :REM Check end of statement (PTRB)
9650 ENDPROC
9670
9800 REM *** Set up BASIC2 entry points           ***
9810 DEFPROCset2
9815 prcfnd = &B1F4 :REM Return to FN/PROC handler
9820 pushi  = &BD94 :REM Push IntA on the BASIC STACK
9825 popi1  = &BE0B :REM Pop &37-&3A from the STACK
9830 poppar = &8CC1 :REM Pop parameter value from STACK
9835 getnsa = &9B1D :REM Get <numeric> or <string>
9840 chksda = &9857 :REM Check end of statement (PTRA)
9845 chksdb = &9852 :REM Check end of statement (PTRB)
9850 ENDPROC
```

The general operation of the routine is as follows:

1 It creates a filename using the name of the FN or PROC, which is left 1 byte after (&37). If it is a FN, ‘F.’ is put on the front: otherwise ‘P.’ is put on the front.

2 OSFILE is called to find the length of the overlay file, and the BASIC STACK is moved down by a corresponding amount. The old value of the STACK pointer is pushed onto the STACK so that it can be restored to its original value afterwards. This action also checks that the STACK has not gone below the level of the HEAP (and produces a 'No room' error if it has).

3 OSFILE is called again, but this time to load the file into the space created for it on the STACK.

4 A 'P' or an 'F' is put in the token slot on the 6502 stack at &1FF. This will cause a 'No FN' or 'No PROC' error when the FN or PROC exits, so that the STACK can be restored, removing the overlayed file.

5 PTRA is pointed to the first character of the overlay and a JMP is made to continue with the FN/PROC handler.

When a 'No FN' or 'No PROC' error is generated on the return from the overlayed call (caused by the substitution of the call type identifier token at stage 4) the routine must not only do the job normally performed by end of the FN/PROC handler, but also remove the overlayed file from the BASIC STACK.

The action performed when this happens is as follows:

1 If it is the exit from a FN, the value is evaluated, and a check is made for the end of the statement. If it is the exit from a PROC, the end of statement check only is made. These actions were not performed by the FN or PROC return statements before the error was generated.

2 The return address to the FN/PROC handler is pulled from the stack. The rest of this routine will do its job instead.

3 PTRB is restored from the stack.

4 The parameter values, pushed on the BASIC STACK when the FN/PROC call was made, are restored.

5 PTRA is restored from the stack.

6 The BASIC STACK, which is now in the same state which it was just after the overlay file was loaded, is restored to its previous value (which was pushed onto the STACK by the overlaying routine).

7 The 6502 stack is restored from the BASIC STACK.

8 The flags are set according to the byte in &27. If we are returning from a PROC, this has no effect; but if we are returning from a FN, the 6502 flags need to reflect the type of the value of the FN.

9 The routine exits, either to the PROC statement handler, or to the code which asked for the FN value.

For more details on the general operation of PROCs and FNs, see section 5.3. For more details on the 'No FN' (error number 7) and 'No PROC' (error number 13) see chapter 11.

This overlay routine is very much better than the one in section 8.2. However, there are still improvements which could be made to it. For example, if a recursive FN or PROC is used, it will load in another new version each time a call is made. Perhaps a linked list of overlayed files could be used to get round this.

Another way of overlaying may be to shift the STACK down bodily, and load the file between HIMEM and the bottom of the screen. A file loaded in this way could be left in memory until a 'No room' error was generated, and then it could be removed (providing it wasn't being executed at the time). In fact, there are many alternatives and improvements which can be made to this general idea.

9 Trapping Other Errors

Chapters 7 and 8 described how two of the errors generated by BASIC could be trapped, and used to add new commands, or to overlay procedures and functions. This section gives a couple of examples of recovering from other errors.

9.1 Bad MODE recover

If an attempt is made to change mode inside a PROC or a FN, a 'Bad MODE' error (error number 25) is generated. When a PROC or FN is in operation, there will be data on the BASIC STACK, which it will use when it returns (see section 5.3).

A MODE change alters HIMEM and resets the BASIC STACK pointer to this new value of HIMEM. If this was reset inside a PROC or a FN, the BASIC STACK contents would be lost, and BASIC would crash when the call returned.

However, by trapping this error, changing MODE inside a PROC or a FN can be allowed, providing that the bottom of the new MODE is above the current HIMEM. If it is, HIMEM can be left as it is, and the BASIC STACK pointer left unchanged. For example, changing from MODE 3 to MODE 6 would be allowed, as the bottom of screen is higher for MODE 6 than MODE 3.

The prevailing conditions on a 'Bad MODE' error are:

Stack contents:	RTI information	3 bytes
	&16 MODE change char.	1 byte

PTRA	points at statement delimiter
&2A	prospective MODE number

If it is possible to change MODE without moving the STACK, this routine will print the MODE change command and continue executing the program. It will not reset HIMEM or the STACK, although the normal MODE change routine will continue to do so whenever the MODE change is made outside a FN or PROC. This means that after this routine has been called, there may be a gap between HIMEM and the bottom of the screen.

```
  10 REM *** Program to allow MODE change inside PROCs ***
  12 REM
  14 REM        M D Plumbley        1984
  16 REM
  18 REM This program traps the "Bad MODE" error (ERR = 25)
  20 REM
  22 REM If there is enough room to change MODE above
  24 REM  HIMEM, without disurbing the BASIC stack, then
  26 REM  MODE can be changed, even if the stack is in use
  28 REM  (i.e. there is a FN or PROC active at the time)
  30 REM
  32 REM "Bad MODE" will still be given if you are changing
  34 REM  to a mode which requires HIMEM to be lower than
  36 REM  the current setting (unless you are not in a
  38 REM  FN/PROC).
  40 REM
  42 REM For BASIC 1, replace EQUs as in chapter 7.
  44 REM
  99
 100 PROCsetup      :REM Set up correct ROM entry points
 490
 495 REM *** OS routines and vectors ***
 500 OSWRCH = &FFEE
 505 OSBYTE = &FFF4
 550 BRKV   = &0202
 590
 595 REM *** Allocate workspace ***
 600 worksp = &0070
 605 svbrkv = worksp
 690
 695 REM *** BASIC system variables ***
 700 Lomem = &0000
 705 Heap  = &0002
 710 Stack = &0004
 715 Himem = &0006
 720 Top   = &0012
 725 Count = &001E
 799
 900 start% = &0C00 :REM Assemble into user char space
 905
 910 FOR opt% = 0 TO 3 STEP 3
 920 P% = start%
 950 [OPT opt%
1000 .init
1005     LDA &8015            \Test that the correct
1010     CMP #baschr          \ version of BASIC is
1015     BEQ basok            \ in the ROM.
1016
1020     BRK                  \If it isn't, print an
1025     EQUB 60              \ error message.
1030     EQUS "Not BASIC "    \ (baschr set by PROCsetup)
1035     EQUB baschr
```

```
1040     EQUB 0
1041
1045 .basok
1050     LDA BRKV             \Load the current BRK vector
1055     LDX BRKV+1           \ into A and X.
1056
1060     CMP #newbrk MOD &100 \If this routine is already
1065     BNE ntsavd           \ set up, don't change BRKV.
1070     CPX #newbrk DIV &100
1075     BEQ saved
1076
1078 .ntsavd
1080     STA svbrkv           \It has not been set up
1085     STX svbrkv+1         \ already, so save old
1090     LDA #newbrk MOD &100 \ BRKV, and set up the new
1095     STA BRKV             \ one.
1100     LDA #newbrk DIV &100
1105     STA BRKV+1
1106
1110 .saved
1115     RTS
1190
1192 \ *** This is the new BRK handling routine ***
1200 .newbrk
1205     PHA           \Save A and Y on 6502 stack
1210     TYA
1215     PHA
1216
1220     LDY #0        \Get error number
1225     LDA (&FD),Y
1226
1230     CMP #25       \If ERR = 25 ("Bad MODE"), then
1235     BEQ badmde    \ try to correct it
1236
1240 .giveup
1245     PLA           \Restore A any Y from 6502 stack
1250     TAY
1255     PLA
1256
1260     JMP (svbrkv)  \Go to old BRK handler
1261
1490 \ *** If we get here, a "Bad MODE" error has      ***
1492 \ ***  occurred. This was either caused by a      ***
1494 \ ***  non-empty BASIC stack, or not enough room. ***
1500 .badmde
1505     LDX &2A       \Get requested mode number from
1510     LDA #&85      \ IntA, and find out what HIMEM
1515     JSR OSBYTE    \ would be in that mode.
1516
1520     CPX Himem     \If new HIMEM would be below the
1525     TYA           \ current HIMEM, then the STACK
1530     SBC Himem+1   \ is in the way.
```

```
1535     BCC giveup
1536
1540     CPX Heap      \If new HIMEM would be below the top
1545     TYA           \ of the variables heap, there is
1550     SBC Heap+1    \ not enough room for the MODE.
1555     BCC giveup
1556
1560     CPX Top       \If HIMEM would be below TOP, there
1565     TYA           \ is not enough room for the MODE.
1570     SBC Top+1     \ This test is in case LOMEM had
1575     BCC giveup    \ not been set to TOP yet.
1576
1580     PLA           \Discard saved values of Y and A
1590     PLA           \ from 6502 stack
1591
1600     PLA           \Discard RTI information from the
1605     PLA           \ 6502 stack. This is pushed by
1610     PLA           \ the BRK instruction.
1611
1615     LDA #0        \Zero COUNT (a MODE change leaves
1620     STA Count     \ the cursor at start of line)
1621
1625     PLA           \Pop "mode change" byte from stack
1630     JSR OSWRCH    \ (pushed by MODE command), and
1631                   \ print it
1632
1635     LDA &2A       \Get mode number from int acc, and
1640     JSR OSWRCH    \ print that
1641
1645     JMP cont      \Command completed, so execute the
1646                   \ next statement.
1647
8000 ]
8010 NEXT
8015 @%=0
8020 PRINT'"Code length =&"~P%-start%
8190
8200 PRINT'''''"** WARNING: Once assembled, the code"
8210 PRINT"generated by this program is not"
8220 PRINT"transferable between different BASICs"
8230 PRINT
8300 PRINT"Execute ""CALL &"~init""" to initialise."'
8310 END
8990
8992 REM *** Set up ROM entry points, allowing for ***
8993 REM ***  BASIC I and BASIC II.                ***
9000 DEFPROCsetup
9010 basic1$ = "BASIC"+CHR$0+"(C)1981 Acorn"+CHR$&A
9020 basic2$ = "BASIC"+CHR$0+"(C)1982 Acorn"+CHR$&A
9030 IF $&8009=basic1$ THEN PROCset1 :ENDPROC
9040 IF $&8009=basic2$ THEN PROCset2 :ENDPROC
9050 PRINT "NOT BASIC 1 OR 2"
```

```
9060 END
9290
9292 REM *** Set up BASIC 1 entry points         ***
9300 DEFPROCset1
9305 baschr = ASC"1":REM Used by init routine
9310 cont   = &8B0C :REM Cont execution at next statement
9320 ENDPROC
9490
9492 REM *** Set up BASIC 2 entry points         ***
9500 DEFPROCset2
9505 baschr = ASC"2":REM Used by init routine
9540 cont   = &8898 :REM Cont execution at next statement
9550 ENDPROC
```

The initialising and BRK handling parts of this routine are very similar to the programs in chapter 7. In fact, there is not really a lot to the program at all.

This routine could be modified to copy the BASIC stack bodily if a MODE change was made which required HIMEM to be lower than its current setting. This could also be used anyway, to ensure that the least amount of memory was being used for each MODE.

Performing a MODE change, and shifting the stack, may be one way of allocating more memory if a 'No room' error is generated. However, this is only possible with BASIC 2, as this error does not use the BRK error generating mechanism in BASIC 1 (see chapter 11 for more on 'No room')

9.2 Bad program salvage

One of the more annoying error messages that BASIC can produce is 'Bad program'. You may have just waited 10 minutes for a long program to load from tape, or spent the last 2 hours typing something in, to be greeted by this message because the program got corrupted somehow. This section describes how the bad program, or as much of it as possible, can be salvaged into an editable form.

Program storage

Program lines are stored in the following format:

00	MSB of line number
01	LSB of line number
02	total length of line (= XX)
03	first character of line text
04	etc.
XX−1	&0D (carriage return) line end marker
XX	MSB of line number of next line
XX+1	etc.

The first byte stored at PAGE is a &0D (carriage return), followed by the MSB of the first line number. The end of the program is marked by an &FF byte after the carriage return on the end of the last line.

The length byte of the line number is used to speed up the search for line numbers in a GOTO or GOSUB. However, if one of these gets corrupted, so that there isn't a &0D where BASIC thinks the end of the line should be, it will give a 'Bad program' error. This could also be caused if the carriage return has been corrupted.

By scanning through the program, re-linking all these length bytes, the program can be salvaged. It may not be completely correct, but at least it will be possible to edit it again.

The salvage routine

This routine can be assembled and the code saved onto disc or cassette by using '*SAVE'. It assembles into the user defined character area, so the code can be loaded in and executed if a 'Bad program' occurs, without disturbing the program to be salvaged.

The program can be loaded and run by typing

```
*LOAD SALVAGE
CALL &C00
```

assuming that it was assembled from &C00 onwards. If the DFS, or any filing system which operates from a paged ROM, is used to load the routine, it should *not* be run by using '*SALVAGE'. If this was used, the DFS ROM, rather than the BASIC ROM, would be paged in while the routine was operating, and the BASIC ROM routines which the are called would not be available. To get round this, the ROM routines required could be duplicated in the salvage routine itself.

```
   4 REM **       Bad program salvage routine       ***
   6 REM
   8 REM              M D Plumbley      1984
  10 REM
  12 REM This routine will scan through the BASIC program
  14 REM  at PAGE and re-set any link pointers which have
  16 REM  been corrupted.
  18 REM
  20 REM Before using with BASIC 1, the EQUS should be
  22 REM  replaced with their equivalents:
  24 REM  "EQUB X"   => "]?P%=X:P%=P%+1:[OPTopt%"
  26 REM  "EQUS A$"  => "]$P%=A$:P%=P%+LEN$P%:[OPTopt%"
  90 REM
  99
 100 PROCsetup      :REM Set up correct ROM entry points
 490
 495 REM *** OS routines and vectors ***
 510 osrdch = &FFE0
 590
 600 worksp = &0070
 605 line   = worksp
 610 ytemp  = worksp+2
 690
 695 REM *** BASIC system variables ***
 700 page   = &0018
 710 inta   = &002A
 799
 900 start% = &0C00 :REM User defined character area
 905
 910 FOR opt% = 0 TO 3 STEP 3
 920 P% = start%
 950 [OPT opt%
 990
 995 \ ** Salvage routine entry point ***
1000 .slvage
1005     LDA page             \Set "line" to point to the
1010     STA line+1           \ first byte of the program
1015     LDY #0               \ at PAGE.
1020     STY line
1025
1030     LDA (line),Y         \If it is a CR, jump to start
```

```
1035     CMP #&0D             \ checking through the lines.
1040     BEQ strtok
1045
1050     JSR pmess                    \Otherwise, print an
1055     EQUS "No CR at start"        \ error message and
1060     NOP                          \ exit.
1065 .end
1070     RTS
1075
1100 .escape                  \This is used to give an
1105     BRK                  \ "Escape" error if the
1110     EQUB 17              \ necessary
1115     EQUS "Escape"
1120     EQUB 0
1125
1195 \ ** Start looking through lines ***
1200 .strtok
1205     JSR pnewl            \Start on a new line
1210
1215     BIT &FF              \If an escape condition is
1220     BMI escape           \ pending, handle it.
1225
1230     LDA line+1           \Print out the address of the
1235     JSR phex             \ current line.
1240     LDA line
1245     JSR phexsp
1250
1255     LDY #1               \If we are at the end of the
1260     LDA (line),Y         \ program, exit.
1265     BMI end
1270
1275     STA inta+1           \Otherwise, print out the
1280     INY                  \ line number.
1285     LDA (line),Y
1290     STA inta
1295     JSR plnum5
1300
1305     LDY #3               \Get the length byte from the
1310     LDA (line),Y         \ line. If it is zero, the
1315     BEQ flink            \ link has failed, so fix it.
1320
1325     TAY                  \Get the byte on the end of
1330     LDA (line),Y         \ the line.
1335
1340     CMP #&0D             \If it is not a CR, the link
1345     BNE flink            \ failed, so fix it.
1350
1355     TYA                  \Transfer the length into A
1360
1365 .newlna
1370     CLC                  \Add the length of the line
1375     ADC line             \ (in A) to the line pointer,
```

```
1380     STA line              \ so it now points to the
1385     BCC strtok            \ line, and go back to
1390     INC line+1            \ "strtok" to handle the next
1395     BCS strtok            \ line.
1400
1990 \ ** If we get here, the link has failed ***
2000 .flink
2005     JSR pmess             \Print a message
2010     EQUS " Failed link"
2015     NOP
2020
2025     LDY #3                \Scan from the start..
2030
2035 .cscan                    \ for control characters
2040     LDA #&1F              \ (i.e. less than &20)
2045     INY
2050
2055 .loop                     \Loop round until a control
2060     CMP (line),Y          \ character is found. If it
2065     BCS fixlnk            \ is, go to fix the link.
2070     INY
2075     BNE loop
2080
2085     DEY                   \If the end wasn't found, set
2090     STY ytemp             \ the "end" to be used at 255
2095
2100     JSR pmess                     \ and print the
2105     EQUS " End not found: F/T"    \ message.
2110     NOP
2115
2120     JSR osrdch            \Read a character, and exit
2125     BCS escape            \ if ESC was pressed.
2130
2135 .notasc                   \Check for a "T".
2140     CMP #ASC"T"
2145     BNE noterm
2150
2155     LDA #&FF              \If it was, set the MSB of
2160     LDY #1                \ the current line to &FF
2165     STA (line),Y          \ to terminate the program,
2170 .nforce                   \ and exit.
2175     RTS
2180
2200 .noterm                   \If it wasn't, check for an
2205     CMP #ASC"F"           \ "F".
2210     BNE nforce
2215
2220     LDY ytemp             \If it was, set the character
2225 .force                    \ where scanning stopped to
2230     LDA #&0D              \ be a CR, and ...
2235     STA (line),Y
2240
```

```
2245     TYA                    \ set the length byte,
2250     LDY #3                 \ and ...
2255     STA (line),Y
2260
2265     JMP newlna             \ go to the next line.
2270
3000 .fixlnk                    \If the control character
3005     LDA (line),Y           \ that was found was a CR,
3010     CMP #&0D               \ force the length byte to
3015     BEQ force              \ point to it.
3020
3025     STY ytemp              \Otherwise, save the offset,
3030
3035     JSR pmess                      \ and print the
3040     EQUS " Control char A/F/T"     \ message.
3045     NOP
3050
3055     JSR osrdch             \Read the character input,
3060     BCS jesc               \ and exit if ESC pressed.
3065
3070     CMP #ASC"A"            \Check for "A".
3075     BNE notasc
3080
3085     LDY ytemp              \If it was, force the
3090     LDA (line),Y           \ control char to be a letter
3095     ORA #&40               \ by ORing it with &40, and
3100     STA (line),Y           \ jump back to continue
3105     JMP cscan              \ scanning the line.
3110
3200 .jesc                      \Jump the the "Escape" error.
3205     JMP escape
8000 ]
8010 NEXT
8015 @%=0
8020 PRINT'"Code length =&"~P%-start%
8190
8200 PRINT'''''"** WARNING: Once assembled, the code"
8210 PRINT"generated by this program is not"
8220 PRINT"transferable between different BASICs"
8230 PRINT
8300 PRINT"Execute ""CALL &"~start%""" to use"'
8310 END
8990
8992 REM *** Set up ROM entry points, allowing for ***
8993 REM ***  BASIC 1 and BASIC 2.                 ***
9000 DEFPROCsetup
9010 basic1$ = "BASIC"+CHR$0+"(C)1981 Acorn"+CHR$&A
9020 basic2$ = "BASIC"+CHR$0+"(C)1982 Acorn"+CHR$&A
9030 IF $88009=basic1$ THEN PROCset1 :ENDPROC
9040 IF $&8009=basic2$ THEN PROCset2 :ENDPROC
9050 PRINT "NOT BASIC 1 OR 2"
9060 END
```

```
9290
9292 REM *** Set up BASIC 1 entry points         ***
9300 DEFPROCset1
9305 plnum5 = &98F5 :REM Print line number (field 5)
9310 pmess  = &BFCB :REM Print message following JSR
9315 pnewl  = &BC42 :REM Print a new line (CRLF)
9320 phex   = &8570 :REM Print A as 2-digit HEX no.
9325 phexsp = &856A :REM Print HEX no. then space
9330 ENDPROC
9490
9492 REM *** Set up BASIC 2 entry points         ***
9500 DEFPROCset2
9505 plnum5 = &9923 :REM Print line number (field 5)
9510 pmess  = &BFCF :REM Print message following JSR
9515 pnewl  = &BC25 :REM Print a new line (CRLF)
9520 phex   = &B545 :REM Print A as 2-digit HEX no.
9525 phexsp = &B562 :REM Print HEX no. then space
9600 ENDPROC
```

The general operation of the routine is as follows:

1 It first checks that there is a carriage return at the start of the program. If there isn't, it prints a message and exits. If this happens, either there was no BASIC program at all, or the routine can be re-started after '?PAGE=13' has been typed.

2 The start address of the current line, and its line number, are printed. If the program is so bad that this salvage routine cannot cope with it properly, this information may help if a hex dump program needs to be used to patch up the program.

3 If the end of the program has been found, the routine exits.

4 If the length byte points correctly to the carriage return on the end of the line, the routine moves on to the next line, and jumps back to stage 2.

5 The message 'Failed link' is printed after the line number, and the line is scanned until a control character is found.

6 If the control character found was a carriage return, the length byte is fixed, and the routine jumps back to continue checking the rest of the program.

7 If the end of the line was not found, or the control character found was not a carriage return, the routine gives the option of forcing the control character to be a letter, forcing the end of the line to be at this point, or marking the end of the program at this line.

The ESC key can be pressed at any time while the salvage operation is underway, and the routine will stop when it is about to do the next line.

The routine may think that it has reached the end of the program before it should have, because it found a negative byte as the MSB of the next line number. It can be forced to continue by typing‘END:?(TOP−1)=0’ to force the end marker to zero before re-starting the salvage routine.

This routine will cope with most things, but if the program is really bad, the following hex dump program maybe useful to examine it by hand. It should be loaded in by setting PAGE above the top of the corrupted program (give plenty of room, just in case), and then just LOADing in as normal.

```
   5 REM **       Hex dump program         **
   6 REM
  10 REM        M D Plumbley    1984
  15 REM
  20 REM Press <space>   to stop listing
  25 REM       <return>  to continue
  30 REM         "Q"     to quit
  35 REM
 100 len% = 8               :REM length of line (bytes)
 200 INPUT"START ADDR :&"input$
 210 start% = EVAL("&"+input$)
 220 INPUT"END   ADDR :&"input$
 230 end% = EVAL("&"+input$)
 400 REPEAT
 410   PROCline(start%)      :REM Hexdump 1 line
 420   start% = start%+len%  :REM Next line
 430   key$ = INKEY$(0)
 440   IF key$=" " THEN PROCwait
 450   IF key$="Q" THEN END
 460   UNTIL start%>end%
 470 END
 998
 999 REM *** Print hexdump of 1 line ***
1000 DEFPROCline(addr%)
1010 @%=4:PRINT~addr%" ";    :REM Addr at start of line
```

```
1015 @%=3
1017 text$ = ""               :REM Clear text string
1020 FOR offset = 0 TO len%-1
1030   byte% = addr%?offset  :REM Get byte
1040   PRINT ~byte%;         :REM Print hex byte
1045   valid = (byte%>=&20 AND byte%<&7F)
1046                          :REM Is it a character?
1050   IF valid THEN chr$=CHR$(byte%) ELSE chr$="."
1060   text$ = text$+chr$    :REM Add char to text string
1070   NEXT offset
1080 PRINT" " text$
1090 ENDPROC
1998
1999 REM *** Wait for <CR> or "Q" to be pressed ***
2000 DEFPROCwait
2010 REPEAT
2020   key$ = GET$
2030   UNTIL key$=CHR$(13) OR key$="Q"
2040 IF key$="Q" THEN END
2050 ENDPROC
```

9.3 Error listing

Sometimes it is not very easy to spot an error in a line of BASIC, especially when it is in the middle of a multi-statement line. The routine in this section will LIST out the line that any error occurred on, together with 2 markers pointing out the possible sources of the error. These represent the positions of the two BASIC text pointers, PTRA and PTRB, at the instant of the error.

For example, if the following line is typed in:

```
>PRINT"HELLO"; REM Should be a ":"
```

the response will be:

```
HELLO
PRINT"HELLO"; REM Should be a ":"
                 ^
                 ^
No such variable
```

The top arrow represents the position of PTRA, and the bottom one represents the position of PTRB. In this case, they both point to the same position (just after the REM token), but in most cases they will be different.

This can also be used to check the position of the pointers, if certain errors are to be intercepted.

```
   5 REM ***        Error listing routine          ***
   7 REM
  10 REM           M D Plumbley       1984
  15 REM
  20 REM When an error occurs, this routine will print out
  25 REM  the offending line, and print the position of
  30 REM  the two BASIC pointers, pointing out the error.
  35 REM
  40 REM This program assembles into user key/character
  42 REM  area at &0B00 onwards.
  44 REM
  46 REM Before using with BASIC 1, the EQUs should be
  48 REM  replaced with their equivalents:
  50 REM  "EQUB X"  => "]?P%=X:P%=P%+1:[OPTopt%
  52 REM  "EQUW X"  => "]!P%=X:P%=P%+2:[OPTopt%"
  54 REM  "EQUS A$" => "]$P%=A$:P%=P%+LEN$P%:[OPTopt%"
  56 REM
  99
 100 PROCsetup      :REM Set up correct ROM entry points
 490
 550 BRKV = &0202
 799
 900 start% = &0B00 :REM User key/char space
 905
 910 FOR opt% = 0 TO 3 STEP 3
 920 P% = start%
 950 [OPT opt%
1000 .init
1005     LDA &8015             \Test that the correct
1010     CMP #baschr           \ version of BASIC is
1015     BEQ basok             \ in the ROM.
1016
1020     BRK                   \If it isn't, print an
1025     EQUB 60               \ error message.
1030     EQUS "Not BASIC "     \ (baschr set by PROCsetup)
1035     EQUB baschr
1040     EQUB 0
1041
1045 .basok
1050     LDA BRKV              \Load the current BRK vector
1055     LDX BRKV+1            \ into A and X.
1056
1060     CMP #newbrk MOD &100 \If this routine is already
1065     BNE ntsavd            \ set up, don't change BRKV.
1070     CPX #newbrk DIV &100
1075     BEQ saved
1076
1078 .ntsavd
```

```
1080     STA svbrkv           \It has not been set up
1085     STX svbrkv+1         \ already, so save old
1090     LDA #newbrk MOD &100 \ BRKV, and set up the new
1095     STA BRKV             \ one.
1100     LDA #newbrk DIV &100
1105     STA BRKV+1
1106
1110 .saved
1115     RTS
1480
1490 \ *** Enter here on BRK ***
1500 .newbrk
1502     PHA             \Save A,Y,X on 6502 stack
1504     TYA
1506     PHA
1508     TXA
1510     PHA
1511
1515     JSR pnewl       \Start a new line
1516
1520     LDA #&FF        \Set up immediate area
1525     STA &3D         \ as default for error area.
1530     LDA #&06        \ (&3D) is used to point to the
1540     STA &3E         \ start of the line in error
1545
1550     LDA &C          \If error occurred in immed mode,
1560     CMP #7          \ don't look for a line
1570     BEQ immed
1575
2010     JSR setERL      \Get ERL, and
2020     LDA &8          \ copy it into the
2030     STA &2A         \ integer accumulator
2040     LDA &9          \ ready for "schlin"
2050     STA &2B
2055
2060     JSR schlin      \Point (&3D) at start of line
2070     BCS noline      \Exit if line not found
2072
2075     JSR pnewl       \Start a new line, followed by
2080     JSR plnum5      \ the line number
2082
2085 .immed
2090     LDA #0          \Reset counters for
2100     STA countA      \ the position of the pointers
2110     STA countB      \ on the line
2115
2120     LDA &A          \Save PTRA in temp area
2130     STA ptrtmp
2140     LDA &B
2150     STA ptrtmp+1
2160     LDA &C
2170     STA ptrtmp+2
```

```
2175
2180     LDA &3D        \Set PTRA to point to start
2190     STA &B         \ of line in error.
2200     LDA &3E        \ (PTRA is used by the line number
2210     STA &C         \ decoding routine)
2220     LDY #1
2230     STY &A
2235
2240     JSR prtlne     \Print out line, setting counters
2245
2250     LDX countA     \Print posn of PTRA
2260     JSR prtptr
2262     JSR pnewl
2265
2270     LDX countB     \Print posn of PTRB
2280     JSR prtptr
2285
2290     LDA ptrtmp     \Restore PTRA from temp area
2300     STA &A
2310     LDA ptrtmp+1
2320     STA &B
2330     LDA ptrtmp+2
2340     STA &C
2342
2345 .noline
2350     PLA            \Restore X,Y,A from 6502 stack
2355     TAX
2360     PLA
2365     TAY
2370     PLA
2371
2375     JMP (svbrkv)   \Continue with default BRK routine
2376
2900 .exit
2910     JMP pnewl      \Print CRLF at end of line
2920
2990 \ *** Print out line at PTRA, setting counters    ***
2991 \ ***  countA and countB to the screen positions  ***
2992 \ ***  of the saved PTRA and PTRB                 ***
3000 .prtlne
3010     LDY &A         \Get next character, and
3020     INC &A         \ increment PTRA
3030     LDA (&B),Y
3035
3040     CMP #&0D       \If end of line,
3050     BEQ exit       \ print CRLF and exit.
3055
3060     CMP #&8D       \If a line number,
3070     BEQ lineno     \ print it
3075
3080     JSR ptoken     \Print char or token in A
3090     JMP counts     \ and skip line number section
```

```
3095
3100 .lineno
3110     JSR getlno      \Get line number after token
3120     JSR plnum0      \ and print it
3130 .counts
3140     CLC             \Move PTRA (position of next
3150     LDA &A          \ char to be printed) into
3160     ADC &B          \ integer accumulator
3170     STA &2A         \ at &2A and &2B
3180     LDA &C
3190     ADC #0
3200     STA &2B
3205
3210     LDA ptrtmp      \Get old PTRA from temp area
3220     ADC ptrtmp+1    \ into X (LSB)
3230     TAX             \
3240     LDA ptrtmp+2    \ and A (MSB)
3250     ADC #0
3255
3260     CPX &2A         \If char at old PTRA has not
3270     SBC &2B         \ been printed yet,
3280     BCC nocntA      \
3290     LDA &1E         \ set countA to COUNT
3300     STA countA      \ (COUNT held in &1E)
3305
3310 .nocntA
3320     CLC             \Get PTRB
3330     LDA &1B         \
3340     ADC &19         \ into X (LSB)
3350     TAX             \
3360     LDA &1A         \ and A (MSB)
3370     ADC #0
3375
3380     CPX &2A         \If char at PTRB has not been
3390     SBC &2B         \ printed yet,
3400     BCC nocntB      \
3410     LDA &1E         \ set countB to COUNT
3420     STA countB
3425
3430 .nocntB
3440     JMP prtlne      \Go back for another char
4990
4991
4992 \ *** Print a "^" in the Xth column ***
4993 \ ***  (entry point is "prtptr")    ***
5006 .loop
5010     LDA #ASC(" ")  \Print a space
5020     JSR pchar
5022
5025 .prtptr
5030     CPX &1E         \If not at the right col,
5040     BNE loop        \ print another space.
```

```
5045
5050     LDA #ASC("^")  \Print a "^"
5060     JSR pchar
5065
5080     RTS            \Exit
7790
7792 \ *** Routine variables area ***
7800 .svbrkv EQUW !BRKV \Space to save BRK vector
7801
7810 .countA EQUB 0     \Screen posn of PTRA
7815 .countB EQUB 0     \Screen posn of PTRB
7816
7820 .ptrtmp EQUW 0     \Temp for PTRA
7825         EQUB 0
8000 ]
8010 NEXT
8015 @%=0
8020 PRINT'"Code length =&"~P%-start%
8190
8200 PRINT'''''"** WARNING: Once assembled, the code"
8210 PRINT"generated by this program is not"
8220 PRINT"transferable between different BASICs"
8230 PRINT
8300 PRINT"Execute ""CALL &"~init""" to initialise."'
8310 END
8990
8992 REM *** Set up ROM entry points, allowing for ***
8993 REM ***  BASIC 1 and BASIC 2.                ***
9000 DEFPROCsetup
9010 basic1$ = "BASIC"+CHR$0+"(C)1981 Acorn"+CHR$&A
9020 basic2$ = "BASIC"+CHR$0+"(C)1982 Acorn"+CHR$&A
9030 IF $&8009=basic1$ THEN PROCset1 :ENDPROC
9040 IF $&8009=basic2$ THEN PROCset2 :ENDPROC
9050 PRINT "NOT BASIC 1 OR 2"
9060 END
9290
9292 REM *** Set up BASIC 1 entry points          ***
9300 DEFPROCset1
9305 baschr = ASC"1":REM Used by init routine
9310 setERL = &B3F6 :REM Get no of line in error into &8,9
9315 schlin = &9942 :REM Find start of line given line no
9320 plnum5 = &98F5 :REM Print &2A,2B in decimal (field 5)
9325 plnum0 = &98F1 :REM Print &2A,2B in decimal (field 0)
9330 ptoken = &B53A :REM Print char, or token if A > &7F
9335 pchar  = &B571 :REM Print char in A, and incr COUNT
9340 pnewl  = &BC42 :REM Print CRLF, and zero COUNT
9345 getlno = &97BA :REM Get tokenised line no at PTRA
9350 ENDPROC
9490
9492 REM *** Set up BASIC 2 entry points          ***
9500 DEFPROCset2
9505 baschr = ASC"2":REM Used by init routine
```

```
9510 setERL = &B3C5 :REM Get no of line in error into &8,9
9515 schlin = &9970 :REM Find start of line given line no
9520 plnum5 = &9923 :REM Print &2A,28 in decimal (field 5)
9525 plnum0 = &991F :REM Print &2A,2B in decimal (field 0)
9530 ptoken = &B50E :REM Print char, or token if A > &7F
9535 pchar  = &B558 :REM Print char in A, and incr COUNT
9540 pnewl  = &BC25 :REM Print CRLF, and zero COUNT
9545 getlno = &97EB :REM Get tokenised line no at PTRA
9550 ENDPROC
```

The general operation of the routine is as follows:

1 The pointer at &3D,&3E is set up to point to the start of the line in error, by searching through the program if necessary.

2 The line is printed out, updating counters which mark the screen position of PTRA and PTRB. Tokens are expanded by the ROM routine ‘ptoken’, but this does not handle line number tokens. These have to be dealt with separately.

3 The markers which point to the positions of PTRA and PTRB are printed out, using the counters set while the error line was being printed.

4 Finally, a JMP is made to the default BRK handler to print out the error message.

The programs in the last few chapters are not really meant to show everything that can be done: they are really just an indication of the way that the BBC BASIC can be enhanced by overlaying procedures, or adding new commands and utilities.

Chapters 10 and 11 detail the routines inside the ROM, and the the other errors generated by BASIC, and these may give ideas for experimenting with more new command and functions, like graphics commands or statistical functions.

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

(187) 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 &F7         \ at location &8015 in the BASIC ROM.
LDA #&15        \
STA &F6         \
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 descriptor 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 &1A, 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 variables/stacks
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
skipin	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
getlna	97AE	97DF	Get a tokenised line number

10.5 Variable/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
ldfan0	A691	A686	Set FPA to zero
ldfan1	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/STACK 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	preserved
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 variables 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 of PTRB 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 appear 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 information 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’ (lnkvar) 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 Stack 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
If N=1:	real in FPA
Otherwise:	integer in IntA

Exit conditions:

Item pushed on STACK

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 STACK 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 STACK, 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 STACK 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 &1E) 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 command input.

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 plnum5 – 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.

Other entry points

NONE

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 A 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 of FPA

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 '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)
If Y= &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 @% (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 @% 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 suppressed. 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=real)
&2A–&35	undefined (except where specified above)
&43	undefined
&48–&4A	undefined
A	number type
X	undefined
Y	undefined
C	undefined
Z	0

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 transferred 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.

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	&40 (result type = integer)
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 popped 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 5-byte packed format. On entry, the pointer at &4B,&4C points at the number to be loaded.

Other entry points

1 ldfat1 – 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, number 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	&A7F5
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	&A1DA

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 (FPA/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. If FPA 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 2^0, 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

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	&ACA1	&AC7B
LEN	string	integer	&AF05	&AED6
LN	real	real	&A807	&A801
LOMEM	– – – –	integer	&AF2B	&AEFC
NOT	integer	integer	&ACFA	&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
SQR	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

11 Errors and Error Recovery

The method that BASIC uses to generate an error is to execute a BRK instruction, which is followed by the error number and error message in the following format:

> BRK instruction to generate the error
> Single byte error number (ERR)
> Error message (like 'Mistake')
> A zero byte to terminate the message

The first section of this chapter describes the default BRK handler in BASIC, and what normally happens when an error is generated. The subsequent sections detail the errors which BASIC can generate, and any recovery from them (if possible), so that they can be intercepted in a similar way to the methods used in chapters 7 to 9.

11.1 The BASIC BRK handler

The Machine Operating System contains a BRK handler, which prints out the error message and restarts the current language. However, BASIC uses its own, so that it can allow errors to be trapped using the 'ON ERROR' statement.

BASIC keeps an 'ON ERROR' pointer in locations &16,&17 in page zero, which is normally set to point to the default error handler (in the ROM). This pointer tells the BASIC BRK handler the location of a set of BASIC statements which will deal with the error.

BASIC resets it to point to the default error handler every time it enters immediate mode (either when it initialises, or when it has finished executing a program), or whenever an 'ON ERROR OFF' statement is executed. When an 'ON ERROR' statement is executed, this pointer will be pointed at the start of the statements on the rest of the line, so that these will be executed when an error occurs.

The other advantage that BASIC gains by using its own error handler, is that the error messages can be tokenised. This means that keywords which appear in error messages (like the ‘RENUMBER’ in ‘RENUMBER space’) only take up 1 byte. The ‘REPORT’ statement, which is used to print out the error message, will convert these tokens into the correct keyword and print them out fully (this uses the ‘ptoken’ ROM routine).

The action of the BASIC 1 BRK handler is:

1 Set up ERL. The base of PTRA will be at the start of the statement which caused the error, so a search is carried out through the program, keeping the line numbers, until the error line is found.

2 Turn TRACE off.

3 Load the ‘ON ERROR’ pointer into PTRA, and start executing the statements making up the error handler by jumping to the ‘Decode and execute command’ entry. This executes the statements as if they had just been typed in as a command.

The default ERROR handler for BASIC1 reads:

```
  REPORT:IF ERL<>0 PRINT" at line ";ERL;
0 PRINT:END
```

The BASIC2 BRK handler has been changed slightly from the BASIC1 version; it will not allow commands to be part of the error handler. This means that you can’t do ‘ON ERROR LIST’ with BASIC2; but it does also stop ‘ON ERROR 10’ (which may have been mistyped for ‘ON ERROR GOTO 10’) which corrupts the program, giving a ‘Bad program’ error.

The action of the BASIC 2 BRK handler is:

1 Set up ERL.

2 Turn TRACE off.

3 If the error number (ERR) is 0, the error is *fatal* (not to be trapped by an ON ERROR statement), so set the 'ON ERROR' pointer to point to the default error handler (i.e. perform 'ON ERROR OFF').

4 Load the 'ON ERROR' pointer into PTRA, ready to execute it later.

5 Clear the BASIC stacks, and restore the DATA pointer. This is done in BASIC1 in the 'Decode and execute command' routine.

6 Abandon the VDU queue (OSBYTE &DA). This is so that the first few characters of the error message to be printed will not be used as part of a multi-character VDU command (like VDU 19 or VDU 23).

7 Acknowledge an ESCAPE condition. In BASIC 1, this is done by the 'Decode and execute command' routine.

8 Set the OPT value to &FF (default).

9 Execute the BASIC statements of the error handler at PTRA, as if they are part of a program.

The default ERROR handler for BASIC2 reads:

```
REPORT:IF ERL PRINT" at line ";ERL:END ELSE PRINT:END
```

Note that the 'REPORT' statement is slightly different for each BASIC: in BASIC1 a VDU 6 command is sent before the error message is printed; in BASIC2 the error message is just printed. This means that if a program turns the screen off using a VDU 21 command, in BASIC1 any error messages will be printed on the screen, but in BASIC2 it will not.

11.2 Numbered errors

The errors detailed in this section have error numbers associated with them, and can be trapped by the BASIC ‘ON ERROR’ statement.

These can be recognised easily by a BRK handler, as &FD,&FE will point at the error number when the BRK handler is entered. Chapters 7 to 9 show how some of these errors can be intercepted.

Error 1 – Out of range

This error is generated by the assembler when the address supplied to a branch instruction is too far away: it should be within −126 to +129 bytes of the branch instruction itself (i.e. within −128 to +127 of the instruction which would be executed if the branch did not take place).

This error (and the ‘No such variable’ error) will be suppressed if ‘OPT 0’ or ‘OPT 1’ is used in the assembler (i.e. bit 1 of OPT is zero). In this case, a displacement of 0 will be used for the branch, and assembly will be allowed to continue. However, due to the way in which the test for this bit is carried out, the ‘Out of range’ error will *only* be suppressed if the OPT setting used is either 0 or 1. In BASIC2, setting bit 2 of the OPT value enables remote assembly (see section 1.6.1); so if this facility is being used, this error will not be suppressed.

This error is recoverable, so that assembly can continue, although recovery should only be attempted if remote assembly is being used (in BASIC2).

Error conditions: (BASIC2 only)

Error number:	1	'Out of range'
Stack contents:	RTI information	3 bytes

&28	current OPT value
A	(current OPT value) DIV 2
X	mnemonic number
Y	undefined

Recovery should only be attempted if:

1 The error number at (&FD) is 1

2 Bit 1 of the current OPT value (bit 0 of A) is 0

To recover from the error:

1 Pull the 3 bytes of RTI information from the stack

2 Set A to zero

3 JMP to &86A5 (BASIC2 only)

This will use a zero displacement for the branch, and assembly will continue.

Error 2 – Byte

This error is generated by the assembler when a 2-byte value is used where only a single byte is allowed (the most significant 2 bytes of the 4-byte integer are ignored). The addressing modes which only allow a single byte are:

LDA #BB	/ Immediate
LDA (BB),Y	/ Post-indexed indirect
LDA (BB,X)	/ Pre-indexed indirect

Recovery should not normally be attempted from this error, as potentially fatal mistakes in an assembler program may not be spotted; however it is possible to recover and just use the LSB of the 2-byte word as the byte if required.

Error conditions:

Error number:	2	'Byte'	
Stack contents:	RTI information		3 bytes

IntA:	value to be used in addressing mode
A	MSB of the 16-bit value in IntA (non-zero)
X	mnemonic number
Y	undefined

Recovery should only be attempted if:

1 The error number at (&FD) is 2

To recover from the error:

1 Pull the 3 bytes of RTI information from the stack

2 JMP to &8669 (BASIC1) or &86A8 (BASIC2)

This will use only the LSB of the 2-byte value as the byte for the instruction, and assembly will continue.

Error 3 – Index

This error is generated by the assembler if it finds an error in the syntax of any of the indexed addressing modes. The main causes of this are:

(a) The absence of an index in one of the indexed indirect modes. For example, 'LDA (&80)' will cause this error.

(b) A comma was found after the data, but no 'X' or 'Y' was found after the comma. For example, 'LDA &80,Z' will cause this error

(c) The wrong index register was used for this particular instruction. For example, 'LDY &80,Y' is not allowed.

Error conditions:

Error number:	3	'Index'
Stack contents:	RTI information	3 bytes

IntA:	value used in the instruction
A	MSB of the 16-bit value in IntA (non-zero)
X	mnemonic number
Y	undefined

This error is not recoverable.

Error 4 – Mistake

This error is generated by BASIC when an equals sign, ‘=’, is not found after the first item of an assignment statement.

The usual cause of this is the mis-typing of a keyword at the start of a statement. When BASIC attempts to interpret the statement, it does not find a keyword, so it assumes that the item is a variable. When it doesn’t find the ‘=’ after it, it generates a ‘Mistake’ error. By trapping this error, it is possible to add in new statements or commands to the language (see chapter 7).

There are, in fact, 5 slightly different causes of a ‘Mistake’:

(a) A non-existent, but valid, variable name was found at the start of a statement, but the first non-space character after it was not a ‘=’.

(b) An existing variable was found at the start of a statement, but the first non-space character after it was not a ‘=’. This looks the same as (a) above, but a slightly different action is taken by the BASIC interpreter.

(c) A ‘LET’ followed by a valid variable name was found, but no ‘=’ was found after the variable.

(d) A pseudo-variable (like ‘HIMEM’) was found at the start of a statement, but no ‘=’ was found after it.

(e) A ‘FOR’ was found, followed by a valid variable name, but no ‘=’ was found after the variable.

Note that if an invalid symbol is found at the start of a statement, and not a valid variable name, then a ‘Syntax error’ (error 16) will be generated instead.

Error conditions:

Error number:	4	'Mistake'	
Stack contents:	RTI information		3 bytes
	Return address		2 bytes
	(Return addr−(d) only		2 bytes)

PTRA: points to the character *after* the first non-space character of the line.

PTRB: points to the character *after* the character which was not an '='.

A	the character which was not an '='
X	undefined
Y	PTRB offset−1 (i.e. points at char in A)

Recovery should only be attempted if:

1 The error number at (&FD) is 4

2 The name at the start of the statement can be recognised as a new command or statement keyword. To attempt this, a pointer could be constructed which points at the character one before PTRA, and recognition attempted from there. See section 7.4 for more on recognising keywords.

To recover from the error:

1 Pull the 3 bytes of RTI information from the stack

2 Pull the 2 bytes of return address from the stack

3 If the first character of the statement was a pseudo-variable token (case (d)), then pull the other 2 bytes of return address from the stack. Normally a statement with a pseudo-variable at the start will not be recognised as a new command (unless one of the new keywords contains the token for it at the front), so this step does not need to be taken.

4 The action of the new statement can now be performed. This should be a call to the 'Check for end of statement' routine at &9810 (BASIC1) or &9857 (BASIC2), to set up the pointers ready to continue with the next statement.

5 Finally, after the action of the new statement has been completed, execution of the rest of the program can be continued with a JMP to &8B0C (BASIC1) or &8B9B (BASIC2). Alternatively, a restart of BASIC may be performed; this may be necessary if the program currently being run has been changed (by deleting a line, perhaps), as the syntax pointers may not point to the correct part of the program.

Note that pseudo-variables are not tokenised if followed by an alphanumeric character (see section 2.3.1). This means that new commands may include these at the start of the new keyword (TIMER', for example).

Error 5 – Missing ,

This error is generated by BASIC if it fails to find a comma where one is required. Most of the functions which expect a comma separating their arguments will give this error if it is missing. For example, 'A=POINT(X)' will cause this error.

Error conditions:

Error number:	5	'Missing ,'
Stack contents:	RTI information (undefined)	3 bytes

A	character which was not a comma
X	undefined
Y	undefined

This error is not recoverable.

Error 6 – Type mismatch

This error is generated by BASIC if a string value was found where a number was expected, or a number was found where a string was expected. There are many ways that this error can be caused, including assigning a string to a number (and vice-versa) or giving the wrong type of argument to a function.

Error conditions:

Error number:	6	'Type mismatch'
Stack contents:	RTI information (undefined)	3 bytes

A	undefined
X	undefined
Y	undefined

This error is not recoverable.

Error 7 – No FN

This error is generated by BASIC when an equals sign is found at the start of a statement (signalling a return from a FN), but a FN is not currently being executed. The FN return routine only decides that a FN is in progress if the 6502 stack pointer is below &FC, and there is a FN token (&A4) as the first item on the stack, at &1FF. See section 5.3 for more on FNs and PROCs.

When inside a FN, the 6502 S register should be &F5 (the next available byte), and the contents of the stack should be:

&1F6	return addr to FN caller	2 bytes
&1F8	PTRB base MSB	
&1F9	PTRB base LSB	
&1FA	PTRB offset	
&1FB	number of parameters	
&1FC	PTRA base MSB	
&1FD	PTRA base LSB	
&1FE	PTRA offset	
&1FF Bottom:	&A4 (FN token)	

Note that the stack is ‘upside down’: the *top of stack* works downwards in page 1. Note also that the parameter values are stored on the BASIC STACK, rather than the 6502 stack.

Section 8.3 illustrates how this error can be used to throw away an overlayed FN when it exits, by substituting a different byte on the bottom of the 6502 stack when the FN is called.

Error conditions:

Error number:	7	‘No FN’	
Stack contents:	RTI information		3 bytes
	undefined		

PTRA:	points to the character after the ‘=’
A	undefined
X	copy of S (after TSX)
Y	undefined

Recovery should only be attempted if:

1 The error number at (&FD) is 7

2 The condition of the stack due to which the error occurred can be determined.

To recover from the error:

1 Pull the 3 bytes of RTI information from the stack.

2 Evaluate the <numeric> or <string> following the ‘=’, and check that it is at the end of the statement.

3 If we are in a FN (but it had been ‘hidden’ by changing the token at &1FF, for example) then executing an RTS will exit from the FN. The result of the FN should be in IntA, FPA, or StrA, with the result type stored in &27 (this is done automatically by the ‘Get <numeric> or <string>’ routine).

Note that the recovery performed in section 8.3 is more complex than this, as it also has to throw away the FN from the STACK.

Error 8 – $ range

This error is generated by BASIC if an attempt is made to use the string indirection operator to assign or read from a string in page zero. For example, the statement 'PRINT $80' will cause this error.

It is possible to recover from this error to allow strings to be *assigned* in page zero, but it is not possible to *read* from a page zero string that has 'got through' the $ range check. If the BASIC 'Get value of variable' routine discovers that the address of an indirected string is only a single byte (i.e. in page zero), it will interpret it as 'CHR$' instead. Thus, if this error is being recovered, 'PRINT $&70' will behave the same as 'PRINT CHR$&70' (although '$&70=A$' will place A$ at location &70 onwards). This mechanism does not appear to have any possible use in BBC BASIC, as it should not allow the address of strings to be less than &100. However, the BASIC on the Acorn ATOM used '$' with a single-byte number instead of 'CHR$', so it could be left over from this.

Error conditions:

Error number:	8	'$ range'

Stack contents:	RTI information	3 bytes
	return address	2 bytes

IntA:	address of the defined-address string
A	0
X	undefined
Y	undefined

Recovery should only be attempted if:

1 The error number at (&FD) is 7

To recover from the error:

1 Pull the 3 bytes of RTI information from the stack.

2 Set the type of of the variable to be a defined string, by storing &80 in location &2C (the 'type' byte of the variable descriptor block).

3 Clear the Z flag (this may have been done already), and set the C flag: this indicates that a valid string variable has been found (see 'Find variable' in section 10.5).

4 Execute an RTS instruction, to return to the code which called the 'Find variable routine'.

Error 9 – Missing "

This error is generated by BASIC if the end of the line is found before the closing quote mark of a string. Anything which uses quoted strings (i.e. READ, INPUT, and the 'Get <string-factor>' routine) can cause this error.

Error conditions:

Error number:	9	Missing "
Stack contents:	RTI information undefined	3 bytes

A	&0D
X	undefined
Y	undefined

This error is not recoverable.

Error 10 – Bad DIM

This error is generated by BASIC if an error is encountered in a 'DIM' statement. The possible causes of this are:

(a) An attempt is made to re-dimension an array which already exists

(b) One of the dimensions of the array is either negative, or greater than &3FFF

(c) The total number of bytes required by the array is greater than &FFFF

(d) The size given to a 'reserve bytes' DIM is either less than −1, or greater than &FFFE

(e) An invalid variable name is found as the DIM subject

See also error 11 – 'DIM space'.

Error conditions:

Error number:	10	'Bad DIM'
Stack contents:	RTI information undefined	3 bytes

A	undefined
X	undefined
Y	undefined

This error is not recoverable

Error 11 – DIM space

This error is generated by BASIC if there is not enough memory for the space required by a 'DIM' statement. This can be caused by:

(a) The new value of the HEAP pointer calculated for an array would be above the BASIC STACK, or would have 'wrapped round' the memory map

(b) The new value of the HEAP pointer calculated for a 'reserve bytes' DIM would be above the BASIC STACK; no test for wrap-round is made (so 'DIM A% &FFFE' will move the HEAP pointer down by 1 byte).

If the DIM statement runs out of memory while it is allocating space for the *name* of the array on the HEAP, then a No room' error will be produced instead.

This error can only be recovered if more space can be allocated somehow (by forcing a MODE change and shifting the STACK, perhaps).The two possible causes of this error, (a) and (b), must be recovered differently.

Error conditions:

Error number:	11	'DIM space'
Stack contents:	RTI information	3 bytes

&37,&38	If (a): copy of old HEAP pointer in &2,&3 If (b): undefined (probably lower than (a))
HEAP:	If (a): points at 'offset' byte of array header If (b): old value
A	undefined
X	MSB of new HEAP pointer
Y	LSB of new HEAP pointer
C	set

Recovery should only be attempted if:

1 The error number at (&FD) is 11 (&B)

2 The new HEAP pointer (in A,Y) is above the BASIC STACK pointer. If it is not, the HEAP pointer has wrapped round over the top of the memory, and recovery should be aborted.

3 The BASIC STACK can be shifted up out of the way, so that there is enough room for the new HEAP.

4 The STACK has not already been corrupted by the array header information. In case (a), the 'offset' byte pointed to by the old HEAP pointer gives the number of bytes already written on to the HEAP; if these would be above STACK, then the STACK has been corrupted. In case (b) there is no header information.

To recover from the error:

1 Pull the 3 bytes of RTI information from the 6502 stack.

2 Shift the BASIC STACK so that the STACK pointer is above the required new HEAP pointer (moving the HEAP would be more tricky, due to all the pointers which point into it).

3 Test if the pointer in locations &37 and &38 is equal to the pointer in locations &2 and &3: if it is, then the error is due to (a); otherwise it is due to (b).

4 If the error is due to (a), execute a JMP to &91A0 (BASIC1) or &91EB (BASIC2); if it was due to (b), execute a JMP to &90B5 (BASIC1) or &9108 (BASIC2).

The new HEAP value will be set, and the DIM statement will continue (the DIM'd area will also be cleared if it is for an array).

Error 12 – Not LOCAL

This error is generated by the BASIC 'LOCAL' statement if a FN or PROC is not currently being executed.

BASIC decides that a FN or PROC is not in progress, if the 6502 stack pointer is &FC or above. See section 5.3 for more on PROCs and FNs.

Error conditions

Error number:	12	'Not LOCAL'	
Stack contents:	RTI information		3 bytes

A	undefined
X	copy of S (by 'TSX')
Y	undefined

This error is not recoverable.

Error 13 – No PROC

This error is generated by BASIC when an 'ENDPROC' statement is found, but a PROC is not currently being executed. The ENDPROC handler only decides that a PROC is in progress if the 6502 stack pointer is below &FC, and there is a PROC token (&F2) as the first item on the stack, at &1FF. See section 5.3 for more on FNs and PROCs.

When inside a PROC, the 6502 S register should be &F5 (the next available byte), and the contents of the stack should be:

&1F6	return addr to PROC caller	2 bytes
&1F8	PTRB base MSB	
&1F9	PTRB base LSB	
&1FA	PTRB offset	
&1FB	number of parameters	
&1FC	PTRA base MSB	
&1FD	PTRA base LSB	
&1FE	PTRA offset	
&1FF Bottom:	&F2 (PROC token)	

Note that the stack is 'upside down': the 'top of stack' works downwards in page 1. Note also that the old parameter values are stored on the BASIC STACK, rather than the 6502 stack.

Section 8.3 illustrates interception of this error to remove an overlayed PROC from the STACK when it exits, by changing the token on the bottom of the stack when it is called.

Error conditions:

Error number:	13	'No PROC'	
Stack contents:	RTI information undefined		3 bytes

PTRA: points to the character after the 'ENDPROC'

A	undefined
X	copy of S (after TSX)
Y	undefined

Recovery should only be attempted if:

1 The error number at (&FD) is 13

2 The condition of the stack which caused the error can be determined.

To recover from the error:

1 Pull the 3 bytes of RTI information from the stack.

2 Call the routine to 'Check end of statement at PTRA', at &9810 in BASIC1 or &9857 in BASIC2.

3 If we are in a PROC (but it had been 'hidden' by changing the token at &1FF, for example), executing an RTS will exit from the PROC. This could be done by JMPing to the 'Check end of statement' routine instead.

Error 14 – Array

This error is generated by the BASIC 'Find variable' routine. It will be caused either if an array name is referenced which has not already been dimensioned; or if the array referenced has fewer dimensions than the one in the original DIM statement (if it has more than the one in the DIM statement, a 'Missing)' error will be generated).

Error conditions

Error number:	14	'Array'	
Stack contents:	RTI information undefined		3 bytes

A	undefined
X	undefined
Y	undefined

This error is not recoverable.

Error 15 – Subscript

This error is generated by the BASIC 'Find variable' routine, if the subscript which is used with an array is out of range. This can be caused if the subscript is negative, or if it is larger than the subscript which the array was DIM'd with.

Error conditions

Error number:	15	'Subscript'	
Stack contents:	RTI information undefined		3 bytes

A	undefined
X	undefined
Y	undefined

This error is not recoverable.

Error 16 – Syntax error

This error is generated by the BASIC 'Check for end of statement' routine if the end of a statement was not found. It can also be caused if the first character of the statement is not a statement token, a variable name, or a special symbol (like '*', '=', or '['); as BASIC will assume that it is dealing with an empty statement. For example, 'COUNT' at the start of a statement will generate a 'Syntax error'. It will also be caused if an invalid variable name was found after a 'LET'.

In BASIC1, this error can also be caused if the '#' is missing after a statement or function which expects a file handle. BASIC2 has the new error 'Missing #' (error 45) for this condition.

Error conditions

Error number:	16	'Syntax error'
Stack contents:	RTI information undefined	3 bytes

A	undefined
X	undefined
Y	undefined

This error is not recoverable.

Error 17 – Escape

This error is generated by the BASIC ‘Check for end of statement’ routine (or the last part of it ,which tests the ESCAPE flag in &FF) if an ESCAPE condition is active (i.e. the ESCAPE key has been pressed).

If this error is to be recovered from (ignored), then the ESCAPE condition should be acknowledged with a call to OSBYTE &7E before continuing (or it could be just cleared by OSBYTE &7C). If this is not done, then the escape condition will still be active on return to the BASIC interpreter; and it will generate this error again at its earliest opportunity.

A better way of ‘recovering’ from this error is to disable the ESCAPE key, to prevent the error from being generated in the first place.

Error conditions

Error number:	17	‘Escape’	
Stack contents:	RTI information		3 bytes
	return address		2 bytes

A	undefined
X	undefined
Y	undefined

Recovery should only be attempted if:

1 The error number at (&FD) is 17

To recover from the error:

1 Pull the 3 bytes of RTI information from the stack.

2 Call OSBYTE &7E (or OSBYTE &7C) to acknowledge the ESCAPE condition.

3 Execute an RTS

Error 18 – Division by zero

This error is generated by the BASIC division routines if the divisor of the the attempted division is zero.

Error conditions

Error number:	18	'Division by zero'	
Stack contents:	RTI information undefined		3 bytes

A	undefined
X	undefined
Y	undefined

This error is not recoverable.

Error 19 – String too long

This error is generated by BASIC if an attempt is made to form a string longer than 255 characters. This can either be caused by concatenating 2 long strings together, or by the STRING$ function creating a string which is longer than 255 bytes. Note that only the LSB of the number sent to the STRING$ command is used; so STRING$(260,"*") will produce a string of 4 asterisks, but STRING$(130,"**") will produce an error.

Error conditions

Error number:	19	'String too long'	
Stack contents:	RTI information undefined		3 bytes

A	undefined
X	undefined
Y	undefined

This error is not recoverable.

Error 20 – Too big

This error is generated by BASIC if an overflow occurs. This can be due to:

(a) A floating point number has overflowed after the end of a calculation. This is discovered by the 'Round and check for overflow' routine, before the floating point number is written out to memory (or to one of the temporary stores).

(b) An attempt was made to 'fix' (i.e. convert to integer) a number which would not fit into a 32-bit 2's complement integer.

Note that this error is not generated when two 32-bit integers are added or subtracted: if an overflow happens here, it will go undetected (try 'PRINT 2000000000+2000000000').

Error conditions

Error number:	20	'Too big'
Stack contents:	RTI information undefined	3 bytes

A	undefined
X	undefined
Y	undefined

This error is not recoverable.

Error 21 – -ve root

This error is generated by BASIC if the 'SQR' routine is given a negative argument. ASN and ACS can also generate this error (if the ABS value of their argument is greater than 1), because they are derived from ATN using the SQR routine:

```
ASN(X) = ATN(X/SQR(1-X*X))
ACS(X) = PI/2 - ASN(X)
```

Error conditions

Error number:	21	'-ve root'	
Stack contents:	RTI information undefined		3 bytes

A	undefined
X	undefined
Y	undefined

This error is not recoverable.

Error 22 – Log range

This error is generated by BASIC if the 'LN' routine is given a negative or zero argument. LOG can also generate this error, as it is derived from LN:

```
LOG(X) = LN(X)/LN(10)
```

(BASIC stores 1/LN(10) as a constant, and uses a multiply to convert the LN to a LOG.)

Error conditions

Error number:	22	'Log range'	
Stack contents:	RTI information undefined		3 bytes

A	undefined
X	undefined
Y	undefined

This error is not recoverable.

Error 23 – Accuracy lost

This error is generated by the BASIC SIN, COS, or TAN routines if the binary exponent of the floating point argument is &98 or greater. If it is, then at least 24 of the 32 bits in the mantissa make up the integer part of the number, leaving only 8 bits (or less) for the fractional part. This gives a resolution of worse than 1/256 (0.004) in the result from a SIN or COS (and all of this from the least significant byte).

The angle given to these trigonometric routines is reduced to the range 0 to PI/2 by subtracting a multiple of PI/2 from it. This does not introduce a significant amount of extra inaccuracy, as BASIC stores a more accurate (41 bits) −PI/2 as 2 separate numbers: a 'coarse' −PI/2, and an accurate adjustment to it.

Error conditions

Error number:	23	'Accuracy lost'
Stack contents:	RTI information	3 bytes
	return addr	2 bytes

FPA:	number to find quadrant and offset from
A	binary exponent of FPA
X	undefined
Y	undefined

This error is not recoverable.

Error 24 – Exp range

This error is generated by BASIC if an attempt is made to take the EXP of a number greater than or equal to 89.5. However, using EXP with an argument between 88 and 89.5 will produce a 'Too big' error. This error can also be generated by the exponentiation operator, as it is derived from the EXP and LN functions:

```
A^B = EXP(B*LN(A))
```

Error conditions

Error number:	24	'Exp range'	
Stack contents:	RTI information undefined		3 bytes

A	undefined
X	undefined
Y	undefined

This error is not recoverable.

Error 25 – Bad MODE

This error is generated by the BASIC 'MODE' statement if there is not enough room for the new MODE above the HEAP or the TOP of the BASIC program, or if the BASIC STACK is not empty; i.e. if an attempt is made to change MODE inside a FN or a PROC. HIMEM and the STACK pointer are reset by a MODE change, and if this happened inside a FN or PROC, BASIC would probably crash on exit (like it does if you set 'HIMEM' inside a FN or PROC).

It is possible to recover from this error and perform the MODE change if the BASIC STACK can be preserved. This can be achieved by either shifting it to where the new HIMEM is, or (more simply) by leaving HIMEM where it was, and only allowing MODE changes which leave the bottom of screen memory higher than this. See section 9.1 for a 'Bad MODE' trap program.

Error conditions

Error number: 25 'Bad MODE'

Stack contents:	RTI information	3 bytes
	&16 MODE change character	1 byte

PTRA: points at the statement delimiting character

&2A	Prospective MODE number (LSB of IntA)

A	undefined
X	undefined
Y	undefined

Recovery should only be attempted if:

1 The error number at (&FD) is 25

2 The bottom of the new MODE (found using OSBYTE &85) would not be below the top of the HEAP

3 The bottom of the new MODE would not be below TOP

4 The contents of the BASIC STACK can be preserved

To recover from the error:

1 Check that the bottom of the new MODE would not be below the current HIMEM, and abort the MODE change if it would be.

2 Pull the 3 bytes of RTI information from the stack.

3 Pull the MODE change character from the 6502 stack, and print it (using OSWRCH)

4 Get the new mode number from &2A, and send that to OSWRCH

5 Continue with the execution of the BASIC statements by making a JMP to the ‘Continue execution’ routine at &8B0C (BASIC1) or &8B9B (BASIC2).

This will allow a MODE change inside a FN or PROC, although HIMEM must be brought down below the bottom of the lowest MODE first. It will always allow a MODE change to a smaller mode. It should also be possible to allow mode changes to a larger mode without previously allocating the space, but that would involve shifting the BASIC STACK bodily, and re-pointing the STACK pointer.

Error 26 – No such variable

This error is generated by the BASIC ‘Get <factor> or <string-factor>’ routine if it tries to read the value of a variable which doesn’t exist. If the assembler is being used with an OPT value which has bit 1 cleared (i.e. OPT 0, 1, 4, 5), this error will be suppressed , and the current value of P% will be returned by the ‘Get <factor>’ routine instead. This error is suppressed if OPT 4 or 5 is used (unlike error 1 ‘Out of range’).

By trapping this error it is possible to to add new functions to BASIC. Note, however, that the first character to be found after the name of the function must not be a ‘(’, or BASIC will think that it is an array, and generate the ‘Array’ error instead (this is much more difficult to recover from). Bracketed expressions can be included after a new function, but the first ‘(’ must be separated from the function name by a space.

Error conditions

Error number:	26	‘No such variable’
Stack contents:	RTI information	3 bytes
	return address	2 bytes

PTRB:	points to the character after the end of the name
&2C:	type of the variable (if C=0)
(&37)	points 1 before the start of the name
&39	length of the name (if C=0)
A	undefined
X	undefined
Y	undefined
C	0=non-existent variable; 1=invalid name

Recovery should only be attempted if:

1 The error number at (&FD) is 26

2 The C flag is 0, signalling that a valid (but non-existent) variable name was found (unless you are trying to recognise a special symbol).

3 The name can be matched with the name of a new function. The length of the function name should be the same as that in &39 (if it is not, PTRB will have to be adjusted to point after the function name). Note that the first character of the name can be read by the sequence:

```
LDY #1
LDA (&37),Y
```

To recover from the error:

1 Ensure that the non-existent variable is actually a new function; if it is not, recovery should be aborted.

2 Pull the 3 bytes of RTI information from the stack.

3 Evaluate the function, and place the value in IntA, StrA, or FPA (depending on the type).

4 Load A with a byte which signals the type of the value of the function. This should be the last action performed before returning, as it sets the Z and N flags which will be tested by the code which is returned to. The type bytes are:

String:	&00
Integer:	&40
Real:	&FF

5 Execute an RTS.

This will return the value of the new function to the code which called the ‘Get <factor> or <string-factor>’ routine.

Error 27 – Missing)

This error is generated by BASIC if a closing bracket is expected, but none is found. This can either be caused by leaving off the ‘)’, or by sending too many arguments to a function, or too many dimensions to an array.

Error conditions

Error number:	27	‘Missing)’	
Stack contents:	RTI information undefined		3 bytes

A	undefined
X	undefined
Y	undefined

This error is not recoverable.

Error 28 – Bad HEX

This error is generated by BASIC if the first character after an '&' was not a hexadecimal digit (i.e. 0 to 9, or A to F).

It is possible to recover from this error (if, for example, you want an '&' by itself to mean 0)

Error conditions

Error number:	28	'Bad HEX'	
Stack contents:	RTI information return address		3 bytes

IntA:	0

A	0
X	0
Y	PTRB offset

Recovery should only be attempted if:

1 The error number at (&FD) is 28

To recover from the error:

1 Pull the 3 bytes of RTI information from the stack.

2 Load A with &40, to signal that the type of the result is an integer.

3 Execute an RTS.

This will return 0 to the code which called the 'Get <factor> or <string-factor>' routine, if no HEX character followed the '&'.

Error 29 – No such FN/PROC

This error is generated by BASIC if an attempt is made to access a FN or PROC which is not defined inside the program. First, the FN/PROC handler tries to find it in the list on the HEAP; if it isn't found, it looks through the program for the definition; if it still doesn't find it, this error is generated.

If this error is trapped, it is possible to overlay procedures and functions from disc, for example, and continue execution. Any routine which attempts to recover from this error should be *very* careful with the state of the 6502 stack, as the FN/PROC routine is in the middle of saving the information it needs to enable it to return properly at the end of the PROC or FN. See chapter 8 for more on overlaying FNs and PROCs.

Error conditions

Error number:	29	'No such FN/PROC'

Stack contents:	RTI information	3 bytes
	PTRA offset	1 byte
	FN/PROC token (&A4/&F2)	1 byte

(&37)	points 1 before the calling PROC/FN token

A	copy of &B (PTRA base LSB)
X	undefined
Y	1

Recovery should only be attempted if:

1 The error number at (&FD) is 29

2 The FN or PROC can be overlayed (from disc, for example).

3 The FN or PROC is of the correct type (the token is held in location &1FF)

To recover from the error:

1 Pull the 3 bytes of RTI information from the stack.

2 Save PTRA base on the stack, by pushing the contents of &B followed by the contents of &C.

3 Load the FN or PROC to be overlayed, allocating space for it as necessary.

4 Restart the FN/PROC handler, to execute the FN or PROC.

There are two major alternative ways to re-start the FN/PROC handler:

(a) Set PTRA base (in &B,&C) to point to the first byte of the program section just overlayed (this will be the &0D usually at PAGE). Then JMP to &B149 (BASIC1) or &B11A (BASIC2). This will cause the ‘Look for FN/PROC in program’ routine to search for the FN/PROC again, but this time starting from PTRA base, instead of PAGE. When the FN/PROC is found, it will be added to the list, and the main FN/PROC handler will be re-joined.

(b) Set PTRA base to point to the byte following the name of the defined PROC or FN in the overlayed section (this will be a ‘(’ if any arguments are being used). Then JMP to &B223 (BASIC1) or &B1F4 (BASIC2). This directly rejoins the FN/PROC handler, without adding the name of the overlayed FN/PROC to the list.

Note that if (a) is being used, the same error may be generated again if the name is still not found; if (b) is being used, the name will not be tested (and does not even need to be in the file itself, as long as PTRA can still be set up to point to the character which would be after it).

Error 30 – Bad call

This error is generated by BASIC if no valid name is found after a PROC or FN token. Note that there can be no spaces between the FN or PROC token, and the name.

Error conditions

Error number:	30	'Bad call'	
Stack contents:	RTI information		3 bytes
	PTRA base MSB		1 byte
	PTRA base LSB		1 byte
	PTRA offset		1 byte
	FN/PROC token (&A4/&F2)		1 byte

(&37)	points 1 before the PROC/FN token
A	undefined
X	undefined
Y	2

This error is not recoverable.

Error 31 – Arguments

This error is generated by BASIC if the number of parameters passed to a FN or PROC is not the same as in the definition of the FN or PROC. It can also be caused if the types of the parameters do not match (i.e. a string being passed where a number is expected).

Error conditions

Error number:	31	'Arguments'	
Stack contents:	RTI information		3 bytes
	PTRA offset		1 byte
	FN/PROC token (&A4/&F2)		1 byte

A	undefined
X	undefined
Y	undefined

This error is not recoverable.

Error 32 – No FOR

This error is generated by the BASIC 'NEXT' statement if there is nothing on the FOR stack. See section 5.6 for more on FOR…NEXT loops.

Error conditions

Error number:	32	'No FOR'	
Stack contents:	RTI information		3 bytes

A	undefined
X	0
Y	undefined

This error is not recoverable.

Error 33 – Can't match FOR

This error is generated by the BASIC 'NEXT' statement if the loop variable was specified (as in 'NEXT I'), but it could not find a FOR loop using that variable on the FOR stack. This error will not be generated if the variable specified in the 'NEXT' statement does not exist: a 'Syntax error' (error 16) will be generated instead.

Error conditions

Error number:	33	'Can't match FOR'	
Stack contents:	RTI information		3 bytes

FOR stack: empty

A	0
X	0
Y	undefined

This error is not recoverable.

Error 34 – FOR variable

This error is generated by the BASIC 'FOR' statement if there is no valid numeric variable after the FOR (i.e. either it is invalid, or it is a string variable). This variable can be an indirected variable (like '!X'), although single byte variables should not be used, as NEXT does not deal with them properly.

Error conditions

Error number:	34	'FOR variable'	
Stack contents:	RTI information		3 bytes

A	undefined
X	undefined
Y	undefined

This error is not recoverable.

Error 35 – Too many FORs

This error is generated by the BASIC 'FOR' statement if there are already 10 'FOR' loops on the FOR stack (see section 5.6).

It is possible to recover from this error, to extend the FOR stack into the REPEAT stack area, for example. This should not normally be attempted, as any REPEAT statement will corrupt an extended FOR stack.

Error conditions

Error number: 35 'Too many FORs'

Stack contents: RTI information 3 bytes

FOR stack: full
&26 &96 (or greater if already recovered)

Initial value already assigned to loop variable

A undefined
X undefined
Y copy of FOR stack pointer in &26

Recovery should only be attempted if:

1 The error number at (&FD) is 35

2 No REPEATs will be used in the program (or GOSUBs if the GOSUB stack area will be used as well).

3 The FOR stack pointer (in &26 and Y) is less than &BE (this gives room for 3 more entries). If the GOSUB stack area is to be used as well, the FOR stack pointer should be less than &F2 (this gives a total of 17 entries in the FOR stack).

To recover from the error:

1 Pull the 3 bytes of RTI information from the 6502 stack

2 JMP to &B7F5 (BASIC1) or &B7DA (BASIC2)

This will continue with the FOR statement, as though the FOR stack had not overflowed. The Y register should not be altered by the recovery routine, as it is used on return to the FOR handler.

Error 36 – No TO

This error is generated by the BASIC 'FOR' statement if the first non-space character after the initial value that the loop variable is to be set to, is not a 'TO' token. The initial value must be a <numeric>.

Recovery from this error is not easily possible, although it could be trapped to allow 'FOR lists'; i.e. a line of the form:

```
FOR I=1,3 TO 5,10
```

which would step through the loop with I taking the values 1,3,4,5, and 10. If this was to be implemented, a new 'NEXT' statement would have to be used for this type of 'FOR' (possibly trapped from the 'Mistake' error), as the normal NEXT would not handle it.

Error conditions

Error number:	36	'No TO'	
Stack contents:	RTI information		3 bytes

Initial value already assigned to loop variable

PTRB:	points to the character after that in A
&26	FOR stack pointer
(&37)	address of the loop variable
&39	type of the loop variable
A	character after the initial value (not 'TO')
X	undefined
Y	copy of FOR stack pointer in &26

Recovery should only be attempted if:

1 The error number at (&FD) is 36

2 An alternative form of the 'FOR' statement can be used. Another NEXT should be used for this structure ('ENDFOR' ?), to handle the next value to be assigned to the loop variable.

To recover from the error:

1 Pull the 3 bytes of RTI information from the 6502 stack.

2 Handle the new FOR structure, either using the FOR stack, or by creating a different stack. The address and type of the loop variable (i.e. its *variable descriptor block*) is already on the FOR stack.

3 If a FOR list is being used, the ENDFOR will have to look at the next item on the list; thus the current value of PTRB should be saved for it.

4 If the whole of the FOR list is to be parsed before the loop is entered, the 'Check for end of statement' routine at &9810 (BASIC1) or &9857 (BASIC2) should be called after the FOR list has been checked. Then the statements in the loop can be started with a JMP to the 'Continue execution' routine at &8B0C (BASIC1) or &8B9B (BASIC2).

5 If the FOR list is not to be parsed until the ENDFOR tries to use it, execution can be continued with a JMP to the 'Skip rest of line, and continue' routine at &8AED (BASIC1) or &8B7D (BASIC2). This will continue execution on the next program line (alternatively, the new FOR routine could just search for a ':', and continue from there).

Error 37 – Too many GOSUBs

This error is generated by the BASIC 'GOSUB' statement if there are already 26 GOSUBs on the GOSUB stack. See section 5.2 for more on GOSUBs.

Due to way that the GOSUB stack is stored (as 2 stacks, one after the other), it is not easily possible to recover this error and extend the stack in a similar manner to the FOR stack.

Error conditions

Error number:	37	'Too many GOSUBs'
Stack contents:	RTI information	3 bytes

&25:	&1A (i.e. GOSUB stack pointer = 26)
A	undefined
X	undefined
Y	&1A (copy of location &25)

This error is not recoverable.

Error 38 – No GOSUB

This error is generated by the BASIC 'RETURN' statement if the GOSUB stack is empty.

Error conditions

Error number:	38	'No GOSUB'
Stack contents:	RTI information	3 bytes

&25:	0
A	undefined
X	undefined
Y	0 (copy of GOSUB stack pointer in &25)

This error is not recoverable.

Error 39 – ON syntax

This error is generated by the BASIC ‘ON’ statement if the first non-space character following the <factor> after the ‘ON’ is not a ‘GOTO’ or a ‘GOSUB’ token. This may be caused if the <factor> is mis-formed, as in:

```
ON A#3 GOTO ...
```

Error conditions

Error number:	39	‘ON syntax’
Stack contents:	RTI information	3 bytes

PTRA:	points to the character *after* that in X
A	undefined
X	non-space character after the <factor>
Y	undefined

This error is not recoverable.

Error 40 – ON range

This error is generated by the BASIC ‘ON’ statement if the controlling <factor> is either less than 1, or greater than the number of entries in the ‘GOTO’ or ‘GOSUB’ list.

This error can be avoided by using an ‘ELSE’ clause after the GOTO or GOSUB list (such as ‘ON I GOTO 20,30 ELSE END’), but in BASIC1 the ‘GOTO’ or ‘GOSUB’ token is left on the 6502 stack if the ELSE clause is executed. If this ELSE clause is executed inside a FN or PROC, the return from this FN or PROC will fail, as the return address will no longer be on the top of the stack. In BASIC2, this has been rectified, and the ELSE clause works correctly.

Error conditions

Error number:	40	'ON range'	
Stack contents:	RTI information		3 bytes
	(token – BASIC1 only		1 byte)

PTRA: points to the last part of the statement handled

A	&0D
X	undefined
Y	offset from PTRA base to point end of line

This error is not recoverable.

Error 41 – No such line

This error is generated by the BASIC 'Evaluate and find line number' routine if the line number it is given does not exist. This routine is used by GOTO, GOSUB, and RESTORE, so all of these can generate this error if given a non-existent line number.

This error could be recovered from if, for example, some sort of program overlaying mechanism is being used.

Error conditions

Error number:	41	'No such line'	
Stack contents:	RTI information		3 bytes
	return address		2 bytes

&2A,&2B: line number which was not found

A	undefined
X	undefined
Y	undefined
C	1

Recovery should only be attempted if:

1 The error number at (&FD) is 41

2 The line can be looked for in an alternative area (for example, in an overlayed program section)

To recover from the error:

1 Pull the 3 bytes of RTI information from the stack.

2 Find the line in the alternative program section, and set the pointer at &3D,&3E to point 1 before the first byte of text of the line (i.e. to point to the length byte of the line). Care should be taken not to generate this error again, unless some flag is used to signal that this overlay has already been tried. If the line number is not found in the new section, and the error is generated again, this recovery routine will be called repeatedly, and the machine will 'hang up'.

3 When the line has been found, clear the carry flag (to signal that the line has been found), and execute an RTS.

This will return to the code which called the 'Evaluate and find line number' routine, which will then continue.

Error 42 – Out of DATA

This error is generated by the BASIC 'Find next DATA item' routine of the 'READ' statement if all of the DATA items in the program have been read.

This error could be recovered, either if some sort of overlaying mechanism is being used, or perhaps by forcing a 'RESTORE' on an 'Out of DATA' error.

Error conditions

Error number:	42	'Out of DATA'
Stack contents:	RTI information	3 bytes
	return address	2 bytes

&1C,&1D: point after the last DATA item read

A	undefined
X	undefined
Y	undefined

Recovery should only be attempted if:

1 The error number at (&FD) is 42

2 Either a RESTORE will be forced, or the DATA will be found in an alternative area

3 The DATA pointer in &1C,&1D does not still point at PAGE. If it does, there is no DATA in the program at all, and so forcing a RESTORE would have no effect.

To recover from the error:

1 Pull the 3 bytes of RTI information from the stack.

2 Set PTRB to point to the area where the DATA will be read from. This will be PAGE to force a RESTORE to the start of the program, or it will point to the new area if an overlay has been loaded.

3 Execute a JMP to &BB7A (BASIC1) or &BB60 (BASIC2). This re-starts the 'Find next DATA item' routine looking from PTRB. If PTRB points at a comma or a 'DATA' token when the routine is re-started, then that routine will return to the READ statement handler, with PTRB pointing at the following DATA item.

Care should be taken that this recovery routine is not called again due to a failure to find any DATA in the new area. The DATA pointer could be used as a flag for this, by setting it to PAGE inside this recovery routine. If no DATA is found on return to the READ handler, then this error will be generated again, but with the DATA pointer still set to PAGE.

Error 43 – No REPEAT

This error is generated by the BASIC 'UNTIL' statement if the REPEAT stack is empty.

Error conditions

Error number:	43	'No REPEAT'	
Stack contents:	RTI information		3 bytes

PTRA:	points to the end of the UNTIL statement
&24:	0 (REPEAT stack empty)
A	undefined
X	0 (copy of REPEAT stack pointer in &24)
Y	undefined

This error is not recoverable.

Error 44 – Too many REPEATs

This error is generated by the BASIC 'REPEAT' statement if the REPEAT stack already contains 20 entries.

The REPEAT stack cannot be extended like the FOR stack, as it saves the MSB and LSB of the pointer in 2 stacks, 1 after the other. See section 5.5 for more on REPEAT loops.

Error conditions

Error number:	44	'Too many REPEATs'	
Stack contents:	RTI information		3 bytes

&24:	&14 (REPEAT stack full with 20 entries)
A	undefined
X	&14 (copy of REPEAT stack pointer in &24)
Y	undefined

This error is not recoverable.

Error 45 – Missing

This error is generated by the BASIC file handling routines if the file handle given to a BPUT, BGET, PTR, or EXT is not preceeded by a '#'. This error is only generated by BASIC2; BASIC1 will generate a 'Syntax error' (error 16) instead.

Error conditions (BASIC2 only)

Error number:	45	'Missing #'	
Stack contents:	RTI information		3 bytes

A	character not a '#'
X	undefined
Y	undefined

This error is not recoverable.

11.3 Fatal errors

These errors cannot be trapped by the 'ON ERROR' statement. Some of them are just messages, with a JMP to immediate mode after the message has been printed; others have error number 0, which cannot be trapped (in BASIC 2).

Some of the errors in this section can still be intercepted by a BRK handler, although those that can be intercepted, will all have error number 0. This means that the error message string following the error number byte must be tested if the error is to be identified correctly.

Bad program

This message is printed if the current program in memory has been corrupted when a check is made. After the message has been printed, a JMP is made to restart BASIC in immediate mode: this cannot be trapped.

If the program is OK, the 'Bad program' check routine resets TOP to the top of the program, and returns to the calling routine. The check is made when:

(a) A new program has been loaded (either by 'LOAD' or 'CHAIN').

(b) An 'OLD' statement has been executed.

(c) A 'LIST' statement is about to be executed.

(d) A 'RENUMBER' command is about to be executed.

(e) An 'END' statement is executed. As an END statement is executed at the end of the default BASIC ERROR handler, this check will also be made whenever an error occurs.

See section 9.2 for a 'Bad program' salvage routine.

Failed at xxx

This message is printed by the 'RENUMBER' command if it finds any references to non-existent line numbers. This error cannot be trapped, but it will not abort the RENUMBERing of the program; it will just produce a list of the lines on which it found unresolved line number references.

Line space

This error is generated by the 'Insert line in program' routine if there is not enough room to insert the line into the program (i.e. the length of the line is longer than the gap between TOP and HIMEM).

This error, although 'fatal' to BASIC, could be recovered from if more memory could be allocated (by forcing a MODE change, perhaps).

Error conditions

Error number:	0	'Line space'	
Stack contents:	RTI information		3 bytes
	return address		2 bytes

IntA:	line number of line to be inserted
&700–	line to be inserted (keyboard buffer)
&3B ,&3C	points to the first character to be inserted
A	undefined
X	undefined
Y	undefined

Recovery should only be attempted if:

1 The error number at (&FD) is 0, followed by the string 'Line space', terminated by a zero byte.

2 HIMEM can be moved up from its present position, perhaps by a MODE change. If it can't be moved, then recovery should be aborted.

To recover from the error:

1 Pull the 3 bytes of RTI information from the stack.

2 Change MODE to shift HIMEM to a higher value.

3 Execute a JMP to &BC96 (BASIC1 or BASIC2 – the addresses coincide).

This will re-enter the routine to insert the line in the program. Note that if this recovery is attempted *without* moving HIMEM up, then this error will just be generated again, and the machine will 'hang up'.

No room

This error is generated by BASIC if an attempt is made to extend the HEAP above the STACK, or extend the STACK below the HEAP. In BASIC1, this is a message which is printed before a JMP to immediate mode (so it gives no line number); but in BASIC2 it is an error with error number 0.

In BASIC2 it is possible to trap this error, and recover from it under certain circumstances (providing some more memory can be found from somewhere); but in BASIC1 it does not go through the BRK handler, and so cannot be trapped.

The 'No room' error can be caused in one of 3 ways:

(a) An attempt was made to allocate space for a new *variable information block* on top of the HEAP. If this is the case, then the error is not recoverable, because the 'Allocate new information block' routine clears the space for the block before checking for a clash with the STACK: thus the contents of the STACK will be corrupted.

(b) An attempt was made to allocate space for a dynamic string on the HEAP. This error is recoverable, as a clash with the STACK is tested for before the string is written into the new area.

(c) An attempt was made to allocate space on the BASIC STACK. This error is also recoverable, because a clash with the HEAP is tested for before the item to be pushed is written into the allocated area.

These 3 different causes of a 'No room' must be handled differently, as they require different return conditions, and in the case of (a), recovery should not be attempted at all.

Error conditions (BASIC2 only)

Error number:	0	‘No room’
Stack contents:	RTI information	3 bytes

If (a):

A	0
X	0
Y	1
C	1

If (b):

A	undefined
X	MSB of attempted new HEAP
Y	LSB of attempted new HEAP
C	1

If (c):

A	LSB of attempted new STACK (copy of location &4)
X	undefined
Y	MSB of attempted new STACK (copy of location &5)
C	0

Recovery should only be attempted if:

1 The error number at (&FD) is 0, followed by the string ‘No room’, terminated by a zero byte.

2 The error was not caused by case (a). If the carry flag was clear when the BRK occurred (this should be tested from the RTI information on the 6502 stack) then it was due to case (c), and recovery is possible. Otherwise, if the X register is non-zero it was due to case (b), and recovery is also possible. If the carry flag was set, and the X register is zero, it was due to case (a), and recovery should be aborted.

To recover from the error:

1 Pull the 3 bytes of RTI information from the stack (the top byte was the 6502 status word when the BRK occurred, and the carry can be checked from there)

2 Allocate some more memory. This could either be done by forcing a mode change, or perhaps by throwing away any overlayed program sections which have been placed between HIMEM and the bottom of the screen. Both of these will involve shifting the STACK bodily, and pointing the STACK pointer (in &4,&5) at the bottom of the new STACK.

3 Check that the HEAP/STACK clash does not still exist: it may be that not enough memory could be cleared. If (c) is being dealt with, then the STACK and HEAP will be in the pointers already; but in case (b), the old HEAP pointer is in &2,&3 and the new one is in X (MSB) and Y (LSB).

4 If (c) is being dealt with, then simply executing an RTS will return to the code that called the 'Check for STACK/ HEAP clash' routine.

5 If (b) is being dealt with, then the 'Assign string' routine can be continued with a JMP to &8C6F (BASIC2 only). The new HEAP pointer must be in the X and Y registers as on entry (alternatively, if the new HEAP pointer is already set up by the recovery routine, a JMP can be made to &8C73 instead).

Trapping this routine, together with trapping the 'No such FN/ PROC' error (error 29), would give a very neat method of procedure and function overlaying. When a FN or PROC is not found in the program, the STACK can be shifted down and an overlay loaded from disc between HIMEM and the bottom of the screen; and when the computer runs out of memory and issues a 'No room' error, the overlay can be removed, and the STACK shifted up again.

RENUMBER space

When the RENUMBER statement is used, it creates a list of the old line numbers above TOP so that it can match up the GOTO and GOSUB references after the lines have been renumbered. This error is generated if there is not enough room between the TOP of the program and HIMEM to fit this list.

Error conditions

Error number	0	'Renumber space'	
Stack contents:	RTI information		3 bytes

A undefined
X undefined
Y undefined

This error is not recoverable

Silly

This error is generated by the AUTO or RENUMBER commands if the interval in their call is either 0 or greater than 255.

It is possible to recover from this error (if you *really* want to have all the lines in your program with the same line number).

Error conditions

Error number	0	'Silly'	
Stack contents:	RTI information		3 bytes
	return address		2 bytes

IntA: AUTO/RENUMBER interval

A 0 if the interval = 0, non-zero if interval > 255
X undefined
Y undefined

This error should only be recovered if:

1 The error number at (&FD) is 0, followed by the string 'Silly', terminated by a zero byte.

To recover from the error:

1 Pull the 3 bytes of RTI information from the 6502 stack.

2 Execute a JMP to &8F28 (BASIC1) or &8F8B (BASIC2).

This will continue with the AUTO or RENUMBER command, ignoring any silly restrictions on the size of the interval.

STOP at line xxx

This error is generated by the BASIC 'STOP' statement. In BASIC1, this is just a message which is printed before a JMP to immediate mode; but in BASIC2 it is an error with error number 0. The BASIC2 error message does not use the 'STOP' token (probably because it was converted from the BASIC1 message).

Error conditions (BASIC2 only)

Error number	0	'STOP'
Stack contents:	RTI information	3 bytes

A undefined
X undefined
Y undefined

This error is not recoverable

Appendix A – Syntax definition

This syntax definition is written in Backus-Naur form, or BNF, in a similar manner to the ‘Syntax’ sections in Chapter 33 of the BBC *User Guide*, or chapter 25 of the Electron *User Guide*. As well as the syntax of the keywords, it also includes the expression evaluator, and non-keyword statements. Although this syntax definition is not particularly easy to read at first, it is very useful when trying to understand what BASIC is doing whilst decoding a particular statement or function.

Note that EVAL and FN may be either string or numeric functions (i.e. they may return either a string or numeric value).

OSCLI and OPENUP are not implemented in BASIC1.

Symbols

The following symbols have special meaning in this section:

<> enclose defined items (‘syntactic entities’), like <numeric> or <factor>.

::= should be read as ‘is defined as’.

| should be read as ‘or’: it is used to separate alternative items.

{} denote possible repetition of the enclosed section **zero** or more times.

[] enclose optional items.

Any other symbols are as read (like ‘+’ and ‘MOD’). Note that the ‘<’ and ‘>’ symbols in the definition of <relation operator> do not enclose a syntactic entity, but are ‘less than’ and ‘greater than’ symbols respectively.

Example

As an illustration, the definition of the RENUMBER command is:

```
<renumber command> ::= RENUMBER [<line-num> [,<line-num>]]
```

There are two optional sections in this line, so the command can be one of three forms:

1) RENUMBER

2) RENUMBER <line-num> (e.g. RENUMBER 1000)

3) RENUMBER <line-num>,<line-num> (e.g. RENUMBER 100,5 – the second number is not an actual line number, but *syntactically* it is just the same)

Statements

```
<immediate-statement> ::= <line-entry> | <command>
      | <statement>

<line-entry> ::= <line num><line>

<line> ::= {anything}{carriage return}

<command> ::= {a statement starting with a command keyword}

<statement> ::= <keyword-statement> | <assignment-statement>
      | <FN-return-statement> | <OS-statement>
      | <enter-assembler-statement> | <empty-statement>

<keyword-statement> ::= {a statement starting with a keyword}

<assignment-statement> ::= <num-var>=<numeric>
      | <string-var>=<string>

<FN-return-statement> ::= =<string> | =<numeric>

<OS-statement> ::= *<line>

<enter-assembler-statement> ::= [

<empty-statement> ::= {nothing}

<auto command> ::= AUTO [<line-num> [,<line-num>]]

<delete command> ::= DELETE <line-num>, <line-num>
```

```
<load command> ::= LOAD <string>

<list command> ::= LIST <line-num> | [<line-num>],[<line-num>]

<listo command> ::= LISTO <numeric>

<new command> ::= NEW

<old command> ::= OLD

<renumber command> ::= RENUMBER [<line-num> [,<line-num>]]

<save command> ::= SAVE <string>

<ptr statement> ::= PTR# <factor>=<numeric>

<page statement> ::= PAGE =<numeric>

<time statement> ::= TIME =<numeric>

<lomem statement> ::= LOMEM =<numeric>

<himem statement> ::= HIMEM =<numeric>

<bput statement> ::= BPUT# <factor>, <numeric>

<call statement> ::= CALL <numeric> {,<variable>}

<chain statement> ::= CHAIN <string>

<clear statement> ::= CLEAR

<close statement> ::= CLOSE# <factor>

<clg statement> ::= CLG

<cls statement> ::= CLS

<colour statement> ::= COLOUR <numeric>

<data statement> ::= DATA <line>

<def fn statement> ::= DEF FN<variable name> [(<variable>
      {,<variable>})]

<def proc statement> ::= DEF PROC<variable name> [(<variable>
      {,<variable>})]

<dim statement> ::= DIM <dim section> {,<dim section>}

<dim section> ::= <variable>(<numeric> {,<numeric>})
      | <num-var><numeric>
```

```
<draw statement> ::= DRAW <numeric>, <numeric>

<end statement> ::= END

<endproc statement> ::= ENDPROC

<envelope statement> ::= ENVELOPE <numeric>, <numeric>,
      <numeric>, <numeric>, <numeric>, <numeric>,
      <numeric>, <numeric>, <numeric>, <numeric>,
      <numeric>, <numeric>, <numeric>, <numeric>

<for statement> ::= FOR <num-var>=<numeric> TO <numeric>
      [STEP<numeric>]

<gcol statement> ::= GCOL <numeric>, <numeric>

<gosub statement> ::= GOSUB <numeric>

<goto statement> ::= GOTO <numeric>

<if statement> ::= IF <testable-condition> [THEN<statement>
      | THEN<line-num>] {<statement>} [ELSE{<statement>}]

<input statement> ::= INPUT [LINE] {{[<input-message>] ,|;}
      <variable>}

<input message> ::= <string-const> | <format-items>

<input# statement> ::= INPUT# <factor> {,<variable>}

<let statement> ::= LET <string-var>=<string>
      | LET <num-var>=<numeric>

<local statement> ::= LOCAL {<variable>}

<mode statement> ::= MODE <numeric>

<move statement> ::= MOVE <numeric>, <numeric>

<next statement> ::= NEXT [<num-var>]

<on-error statement> ::= ON ERROR <statement>|OFF

<on statement> ::= ON <numeric> GOTO|GOSUB <numeric>
      {,<numeric>} [ELSE <statement>]

<oscli statement> ::= OSCLI <string-factor>

<plot statement> ::= PLOT <numeric>, <numeric>, <numeric>

<print statement> ::= PRINT {~ | , | ; | <format items> |
      <numeric> | <string>}
```

```
<format items> ::= ' | SPC<factor> | TAB(<numeric>[,<numeric>])

<proc statement> ::= PROC <variable name> [(<variable>
      {,<variable>})]

<read statement> ::= READ {[<variable>] [,]}

<rem statement> ::= REM<line>

<repeat statement> ::= REPEAT

<report statement> ::= REPORT

<restore statement> ::= RESTORE

<return statement> ::= RETURN

<run statement> ::= RUN

<sound statement> ::= SOUND <numeric>, <numeric>, <numeric>,
      <numeric>

<stop statement> ::= STOP

<trace statement> ::= TRACE ON|OFF|<numeric>

<until statement> ::= UNTIL <testable condition>

<vdu statement> ::= VDU <numeric> {,|; <numeric>} [,|;]

<width statement> ::= WIDTH <numeric>
```

Expression evaluator

```
<numeric> ::= <testable-condition>

<testable-condition> ::= <logical-expression>
      {OR|EOR <logical-expression>}

<logical-expression> ::= <relnl-expression>
      {AND <relnl-expression>}

<relnl-expression> ::= <expression> |
      <expression><relation-operator><expression> |
      <string><relation-operator><string>

<relation operator> ::= = | < | <= | <> | > | >=

<expression> ::= <term> {+|- <term>}

<term> ::= <sub-term> {<term-operator><sub-term>}
```

```
<term-operator> ::= * | / | MOD | DIV

<sub-term> ::= <factor> {^<factor>}

<factor> ::= <primitive> | -<primitive> | +<primitive>

<primitive> ::= <function> | <num-var> | <num-const> |
      &<hex-number> | (<testable expression>)

<variable> ::= <string-var> | <num-var>

<num-var> ::= <simple-var> | ?<factor> | !<factor> |
      <simple-var>?<factor> | <simple-var>!<factor>

<string> ::= <string-factor> {+ <string-factor>}

<string-factor> ::= <string-function> | <string-var> |
      <string-const> | (<string>)

<string-var> ::= <dynamic-string> | $<factor>

<num-const> ::= {a number like 12 or 1.3E-15}

<line-num> ::= {a positive decimal integer}

<hex-number> ::= {a hexadecimal number like FFE4}

<simple-var> ::= {a numeric variable like A% or FRED(3)}

<dynamic-string> ::= {a string variable like A$ or BBC$(1)}

<string-const> ::= {a string in quotes, "like this string"}
```

Functions

```
<function> ::= {a numeric-valued function}

<string-function> ::= {a string-valued function}

<abs function> ::= ABS<factor>

<acs function> ::= ACS<factor>

<adval function> ::= ADVAL<factor>

<asc function> ::= ASC<string>

<asn function> ::= ASN<factor>

<atn function> ::= ATN<factor>

<bget function> ::= BGET#<factor>
```

```
<cos function> ::= COS<factor>

<count function> ::= COUNT

<deg function> ::= DEG<factor>

<eof function> ::= EOF#<factor>

<erl function> ::= ERL

<err function> ::= ERR

<eval function> ::= EVAL<string-factor>

<exp function> ::= EXP<factor>

<ext function> ::= EXT#<factor>

<false function> ::= FALSE

<fn function> ::= FN<variable name> [(<variable>
       {,<variable>})]

<get function> ::= GET

<himem function> ::= HIMEM

<inkey function> ::= INKEY<factor>

<instr function> ::= INSTR(<string>, <string> [,<numeric>])

<int function> ::= INT<factor>

<len function> ::= LEN<string-factor>

<ln function> ::= LN<factor>

<log function> ::= LOG<factor>

<lomem function> ::= LOMEM

<not function> ::= NOT<factor>

<openin function> ::= OPENIN<string-factor>

<openout function> ::= OPENOUT<string-factor>

<openup function> ::= OPENUP<string-factor>

<page function> ::= PAGE

<pi function> ::= PI
```

```
<point function> ::= POINT(<numeric>, <numeric>)

<pos function> ::= POS

<ptr function> ::= PTR#<factor>

<rad function> ::= RAD<factor>

<rnd function> ::= RND[(<numeric>)]

<sgn function> ::= SGN<factor>

<sin function> ::= SIN<factor>

<sqr function> ::= SQR<factor>

<tan function> ::= TAN<factor>

<time function> ::= TIME

<top function> ::= TOP

<true function> ::= TRUE

<usr function> ::= USR<factor>

<val function> ::= VAL<string-factor>

<vpos function> ::= VPOS


<chr string-func> ::= CHR$<factor>

<eval string-func> ::= EVAL<string-factor>

<fn string-func> ::= FN<variable name> [(<variable>
       {,<variable>})]

<get string-func> ::= GET$

<inkey string-func> ::= INKEY$<factor>

<left string-func> ::= LEFT$(<string>, <numeric>)

<mid string-func> ::= MID$(<string>, <numeric> [,<numeric>])

<right string-func> ::= RIGHT$(<string>, <numeric>)

<str string-func> ::= STR$[~]<factor>

<string string-func> ::= STRING$(<numeric>, <string>)
```

Appendix B – BASIC ROM summary

BASIC1	BASIC2	ROUTINE
8000	8000	BASIC entry point
8006	8006	Paged ROM data
801F	8023	Language initialisation
806D	8071	Keyword table
835A	836D	Keyword action address table
843C	8451	Assembler mnemonic tables
84E6	84FD	‘]’ (Back to BASIC from assembler)
84ED	8504	‘[‘ statement (Assembler entry point)
87E4	8821	Evaluate integer <numeric>
87FD	887C	Substitute token in buffer
8819	8897	Tokenise line number
88AB	8926	Check for alphanumeric char (or ‘.’)
88D3	8951	Tokenise a line
8A13	8A8C	Get character at PTRB
8A1E	8A97	Get character at PTRA
8A3D	8AB6	‘OLD’ statement
8A50	8AC8	‘END’ statement
8A59	8AD0	‘STOP’ statement
8A7D	8ADA	‘NEW’ statement
8A80	8ADD	Cold start
8A96	8AF3	Warm start
8A99	8AF6	Enter immediate mode
8BAA	8B47	‘=’ statement (return FN value)
8BC3	8B73	‘*,’ statement (send line to OSCLI)
8AED	8B7D	‘DATA’, ‘DEF’, ‘REM’ statement (skip line)
8B0C	8B9B	Continue execution at next statement
8B57	8BE4	‘LET’ statement
8BD0	8C1E	Assign string
8C5B	8CC1	Pop parameter value
8CC5	8D2B	‘PRINT#’ statement
8D33	8D9A	‘PRINT’ statement
8DBD	8E2A	‘TAB(X,Y)’ in printable section
8DD9	8E40	‘TAB(’ in printable section
8DF2	8E58	‘SPC’ in printable section
8E57	8EBD	‘CLG’ statement
8E5E	8EC4	‘CLS’ statement
8E6C	8ED2	‘CALL’ statement
8ECE	8F31	‘DELETE’ statement

8F37	8FA3	‘RENUMBER’ statement
905F	90AC	‘AUTO’ statement
90DD	912F	‘DIM’ statement
91EB	9236	Perform ‘space required’ multiplication
9212	925D	‘HIMEM’ statement
9224	926F	‘LOMEM’ statement
9239	9283	‘PAGE’ statement
9326	928D	‘CLEAR’ statement
9243	929F	‘TRACE’ statement
927B	92C9	‘TIME’ statement
9292	92E3	Get integer <factor>
92AC	92EB	Get real <factor>
92B6	9304	‘PROC’ statement
92D5	9323	‘LOCAL’ statement
9310	9356	‘ENDPROC’ statement
932F	937A	‘GCOL’ statement
9346	938E	‘COLOUR’ statement
935A	939A	‘MODE’ statement
93A1	93E4	‘MOVE’ statement
93A5	93E8	‘DRAW’ statement
93AE	93F1	‘PLOT’ statement
93EF	942F	‘VDU’ statement
941B	945B	Look for FN/PROC in list
9429	9469	Look for variable in list
94AD	94ED	Link in new PROC/FN
94BC	94FC	Link in new variable
94F7	9531	Clear space for information block
951F	9559	Scan variable name
9548	9582	Find variable, creating if needed
9595	95C9	Find variable at PTRA
95A9	95DD	Find variable at PTRB
97AC	97DD	Get tokenised line number at PTRA
97D6	9807	Set PTRB to PTRA, then…
97E2	9813	Evaluate <numeric> after ‘=’
980B	9852	Check end of statement at PTRB
9810	9857	Check end of statement at PTRA
9851	9880	Move to start of next statement
9893	98C2	‘IF’ statement
98F1	991F	Print line number in IntA
9942	9970	Look for program line
99C0	99EA	Perform integer division
9A76	9A9E	Perform comparison
9AF7	9B1D	Set PTRB to PTRA, then…

9B03	9B29	Get <numeric> or <string> at PTRB
9B14	9B3A	‘OR’ operator
9B2F	9B55	‘EOR’ operator
9B45	9B72	Get <logical expression>
9B54	9B7A	‘AND’ operator
9B76	9B9C	Get <relnl expression>
9B88	9BAE	‘=’ operator (comparison)
9BA7	9BCD	‘<’ operator
9BAE	9BD4	‘<=’ operator
9BB9	9BDF	‘<>’ operator
9BCB	9BE8	‘>’ operator
9BD4	9BFA	‘>=’ operator
9C1D	9C42	Get <expression>
9C29	9C4E	‘+’ operator
9C90	9CB5	‘−’ operator
9D17	9D3C	‘*’ operator
9DAE	9DD1	Get <term>
9DC2	9DE5	‘/’ operator
9DDE	9E01	‘MOD’ operator
9DE7	9E0A	‘DIV’ operator
9DFD	9E20	Get <sub-term>
9E12	9E35	‘^’ operator (exponentiation)
9E81	9E90	Convert number to HEX string
9ED0	9EDF	Convert number to string
A06C	A07B	Get number at PTRB
A169	A178	Add FPB mantissa to FPA mantissa
A188	A197	Multiply FPA mantissa by 10
A1CB	A1DA	Test FPA
A1E5	A1F4	Multiply FPA by 10
A20F	A21E	Copy FPA into FPB
A23E	A24D	Divide FPA by 10
A295	A2A4	Add A to PFA mantissa
A2AF	A2BE	Convert IntA to FPA
A2DE	A2ED	Convert A to FPA
A2F4	A303	Normalise FPA
A33F	A34E	Load FPB from packed number at (&4B)
A36E	A37D	Store FPA at &471–&475
A372	A381	Store FPA at &476–&47A
A376	A385	Store FPA at &46C–&470
A37E	A38D	Store FPA at (&4B)
A3A3	A3B2	Load FPA from &46C–&470
A3A6	A3B5	Load FPA from (&4B)
A3F2	A3E4	Convert FPA to IntA

A40C	A3FE	Convert FPA to fixed format
A463	A453	Set FPB to zero
A494	A486	Extract fractional part of FPA
A505	A4D0	Subtract number at (&4B) from FPA
A4DE	A4D6	Exchange FPA with number at (&4B)
A4E4	A4DC	Copy FPB into FPA
A50B	A4FD	Subtract FPA from number at (&4B)
A50E	A500	Add number at (&4B) to FPA
A513	A50B	Add FPB to FPA
A611	A606	Multiply FPA by number at (&4B)
A61E	A613	Multiply FPA by FPB
A661	A656	Multiply FPA by (&4B); test for overflow
A691	A686	Set FPA to zero
A6A4	A699	Set FPA to 1
A6B0	A6A5	Invert FPA (set FPA = 1/FPA)
A6B8	A6AD	Divide (&4B) by FPA
A6C9	A6BE	'TAN' function
A6F2	A6E7	Divide FPA by (&4B)
A6FC	A6F1	Divide FPA by FPB
A7B4	A7B4	'SQR' function
A7EF	A7E9	Point &4B,&4C at &47B
A7F3	A7ED	Point &4B,&4C at &471
A7F7	A7F1	Point &4B,&4C at &476
A7FB	A7F5	Point &4B,&4C at &46C
A804	A7FE	'LN' function
A856	A869	Constant: log(e) (i.e. 'LOG EXP 1')
A85B	A86E	Constant: ln(2)
A860	A873	Constant series for 'LN' evaluation
A889	A897	Perform series evaluation
A8C6	A8D4	'ACS' function
A8CC	A8DA	'ASN' function
A907	A907	'ATN' function
A956	A95A	Constant series for 'ATN' evaluation
A989	A98D	'COS' function
A994	A998	'SIN' function
AA5C	AA48	Point &4B,&4C at 'coarse −PI/2'
AA60	AA4C	Point &4B,&4C at adjustment to above
AA69	AA55	Point &4B,&4C at PI/2
AA6D	AA59	Constant: 'coarse −PI/2'
AA73	AA5E	Constant: adjustment to 'coarse −PI/2'
AA77	AA63	Constant: PI/2
AA7C	AA68	Constant: PI/180 (for 'RAD')
AA81	AA6D	Constant: 180/PI (for 'DEG')

AA86	AA72	Constant series for ‘SIN’ evaluation
AAB4	AA91	‘EXP’ function
AB07	AAE4	Constant: e (‘EXP 1’)
AB0C	AAE9	Constant series for ‘EXP’ evaluation
AB56	AB33	‘ADVAL’ function
AB64	AB41	‘POINT’ function
AB92	ABED	‘POS’ function
AB9B	AB76	‘VPOS’ function
ABAD	AB88	‘SGN’ function
ABCD	ABA8	‘LOG’ function
ABD6	ABB1	‘RAD’ function
ABE7	ABC2	‘DEG’ function
ABF0	ABCB	‘PI’ function
ABFB	ABD2	‘USR’ function
AC12	ABE9	‘EVAL’ function
AC55	AC2F	‘VAL’ function
AC9E	AC78	‘INT’ function
ACC4	AC9E	‘ASC’ function
ACD3	ACAD	‘INKEY’ function
ACDE	ACB8	‘EOF’ function
ACEA	ACC4	‘TRUE’ function
ACF7	ACD1	‘NOT’ function
AD08	ACE2	‘INSTR’ function
AD8D	AD6A	‘ABS’ function
ADB5	AD8C	Unary ‘−’ function
AE1B	ADEC	Get <factor> or <string-factor> at PTRB
AE9C	AE6D	Get HEX number
AEE3	AEB4	‘TIME’ function
AEEF	AEC0	‘PAGE’ function
AEF9	AECA	‘FALSE’ function
AF00	AED1	‘LEN’ function
AF0B	AEDC	‘TOP’ function
AF26	AEF7	‘COUNT’ function
AF2B	AEFC	‘LOMEM’ function
AF32	AF03	‘HIMEM’ function
AF78	AF49	‘RND’ function
AF85	AF56	Load IntA from 00,X–03,X
AFB6	AF87	Spin random number generator
AFCE	AF9F	‘ERL’ function
AFD5	AFA6	‘ERR’ function
AFDC	AFAD	Perform INKEY
AFE8	AFB9	‘GET’ function
AFEE	AFBF	‘GET$’ function

AFFB	AFCC	‘LEFT$(’ function
B01D	AFEE	‘RIGHT$(’ function
B055	B026	‘INKEY$’ function
B05D	B02E	Set StrA to empty string
B068	B039	‘MID$(’ function
B0C3	B094	‘STR$’ function
B0F1	B0C2	‘STRING$(’ function
B141	B112	Search for FN/PROC not in list
B1C4	B195	‘FN’ function
B27C	B24D	Handle FN/PROC parameters
B33C	B30D	Push value and descriptor on STACK
B35B	B32C	Read value of variable
B3EE	B3BD	‘CHR$’ function
B3F6	B3C5	Set up ERL
B433	B402	BRK hander
B443	B433	Default BASIC error handling text
B461	B44C	‘SOUND’ statement
B49C	B472	‘ENVELOPE’ statement
B4CC	B4A0	‘WIDTH’ statement
B4E0	B4B1	Assign numeric variable
B53A	B50E	Print A as a character or token
8570	B545	Print A as 2-digit HEX number
B571	B558	Print A as a character (handling COUNT)
856A	B562	Print A as HEX number followed by space
B58D	B577	Print selected LISTO formatting spaces
B5A0	B58A	‘LISTO’ command
B5B5	B59C	‘LIST’ command
B6AE	B695	‘NEXT’ statement
B7DF	B7C4	‘FOR’ statement
B8B4	B888	‘GOSUB’ statement
B8D5	B8B6	‘RETURN’ statement
B8EB	B8CC	‘GOTO’ statement
B903	B8E4	‘ON ERROR OFF’ statement
B911	B8F2	‘ON ERROR’ statement
B934	B915	‘ON’ statement
B9B8	B99A	Get line number, and find it in program
B9ED	B9CF	‘INPUT#’ statement
BA62	BA44	‘INPUT’ statement
BB00	BAE6	‘RESTORE’ statement
BB39	BBF1	‘READ’ statement
BBCC	BBB1	‘UNTIL’ statement
BBFF	BBE4	‘REPEAT’ statement
BC17	BBFC	Input string to StrA

BC1D	BC02	Input string to keyboard buffer
BC42	BC25	Print CRLF (newline)
BC4A	BC2D	Delete line in program
BCAA	BC8D	Insert line into program
BD29	BD11	'RUN' statement
BD38	BD20	Clear variables/stacks
BD52	BD3A	Reset stacks; RESTORE data pointer
BD69	BD51	Push FPA on STACK
BD96	BD7E	Pop real number from STACK
BDA8	BD90	Push IntA, FPA, or StrA on STACK
BDAC	BD94	Push IntA on STACK
BDCA	BDB2	Push StrA on STACK
BDE3	BDCB	Pop StrA from STACK
BDF4	BDDC	Discard string from STACK
BE04	BDEA	Pop IntA from STACK
BE17	BDFF	Discard integer (4 bytes) from STACK
BE23	BE0B	Pop integer from STACK to &37–&3A
BE25	BE0D	Pop integer into page zero
BE46	BE2E	Allocate STACK space; check for 'No room'
BE5C	BE44	Copy IntA into 0,X–3,X
BE6D	BE55	Add Y to pointer at &3D,&3E; Set Y=1
BE7A	BE62	Perform BASIC program load
BE88	BE6F	Test for 'Bad program'
– – – –	BEC2	'OSCLI' statement
BEFA	BEF3	'SAVE' statement
BF2D	BF24	'LOAD' statement
BF33	BF2A	'CHAIN' statement
BF39	BF30	'PTR' statement
BF4F	BF46	'EXT' function
BF50	BF47	'PTR' function
BF61	BF58	'BPUT' statement
BF78	BF6F	'BGET' function
– – – –	BF78	'OPENIN' function
BF81	BF7C	'OPENOUT' function
BF85	BF80	'OPENUP' function ('OPENIN' in BASIC 1)
BF9E	BF99	'CLOSE' statement
BFAE	BFA9	Get file handle at PTRA
BFCB	BFCF	Print text after 'JSR' to this routine
BFE6	BFE4	'REPORT'
– – – –	BFF9	Text: 'Roger'

Appendix C – 6502 Instruction Set Summary

ADC	Add Memory to Accumulator with Carry
AND	‘AND’ Memory with Accumulator
ASL	Shift Left one bit (Memory or Accumulator)
BCC	Branch on Carry Clear
BCS	Branch on Carry Set
BEQ	Branch on result Zero
BIT	Test bits in Memory with Accumulator
BMI	Branch on result Minus
BNE	Branch on result not Zero
BPL	Branch on result Plus
BRK	Force Break
BVC	Branch on Overflow Clear
BVS	Branch on Overflow Set
CLC	Clear Carry flag
CLD	Clear Decimal mode
CLI	Clear Interrupt disable bit
CLV	Clear Overflow flag
CMP	Compare Memory and Accumulator
CPX	Compare Memory and index X
CPY	Compare Memory and index Y
DEC	Decrement Memory by one
DEX	Decrement index X by one
DEY	Decrement index Y by one
EOR	‘Exclusive-OR’ Memory with Accumulator
INC	Increment Memory by one
INX	Increment index X by one
INY	Increment index Y by one
JMP	Jump to new location
JSR	Jump to subroutine
LDA	Load Accumulator with Memory
LDX	Load index X with Memory
LDY	Load index Y with Memory
LSR	Shift one bit right (Memory or Accumulator)

NOP	No operation
ORA	‘OR’ Memory with Accumulator
PHA	Push Accumulator on Stack
PHP	Push Processor Status on Stack
PLA	Pull Accumulator from Stack
PLP	Pull Processor Status from Stack
ROL	Rotate one bit left (Memory or Accumulator)
ROR	Rotate one bit right (Memory or Accumulator)
RTI	Return from Interrupt
RTS	Return from subroutine
SBC	Subtract Memory from Accumulator with Carry
SEC	Set Carry flag
SED	Set Decimal mode
SEI	Set Interrupt disable status
STA	Sore Accumulator in Memory
STX	Store index X in Memory
STY	Store index Y in Memory
TAX	Transfer Accumulator to index X
TAY	Transfer Accumulator to index Y
TSX	Transfer Stack Pointer to index X
TXA	Transfer index X to Accumulator
TXS	Transfer index X to Stack Register
TYA	Transfer index Y to Accumulator

Appendix D – Keyword list

For a list of the keyword tokens, and their associated flags, in token value order, see section 2.3.

94	ABS	A0	EVAL
95	ACS	A1	EXP
96	ADVAL	A2	EXT
80	AND	A3	FALSE
97	ASC	A4	FN
98	ASN	E3	FOR
99	ATN	E6	GCOL
C6	AUTO	A5	GET
9A	BGET	BE	GET$
D5	BPUT	E4	GOSUB
D6	CALL	E5	GOTO
D7	CHAIN	93	HIMEM (left)
BD	CHR$		
D8	CLEAR	D3	HIMEM (right)
D9	CLOSE		
DA	CLG	E7	IF
DB	CLS	A8	INT
9B	COS	BF	INKEY$
FB	COLOUR	A6	INKEY
9C	COUNT	E8	INPUT
DC	DATA	A7	INSTR(
9D	DEG	C0	LEFT$(
DD	DEF	A9	LEN
C7	DELETE	E9	LET
DE	DIM	86	LINE
81	DIV	C9	LIST
DF	DRAW	AA	LN
8B	ELSE	C8	LOAD
E0	END	EA	LOCAL
E1	ENDPROC	AB	LOG
E2	ENVELOPE	92	LOMEM (left)
82	EOR		
C5	EOF	D2	LOMEM (right)
9E	ERL		
9F	ERR	C1	MID$(
85	ERROR	83	MOD

EB	MODE
EC	MOVE
CA	NEW
ED	NEXT
AC	NOT
EE	ON
87	OFF
CB	OLD
8E	OPENIN (BASIC2)
AD	OPENIN (BASIC1)
AD	OPENUP (BASIC2)
AE	OPENOUT
84	OR
FF	OSCLI
90	PAGE (left)
D0	PAGE (right)
AF	PI
F0	PLOT
B0	POINT(
B1	POS
F1	PRINT
F2	PROC
8F	PTR (left)
CF	PTR (right)
B2	RAD
F3	READ
F4	REM
CC	RENUMBER
F5	REPEAT
F6	REPORT
F7	RESTORE
F8	RETURN
C2	RIGHT$(
B3	RND
F9	RUN
CD	SAVE
B5	SIN
B4	SGN
D4	SOUND
89	SPC
B6	SQR
88	STEP
FA	STOP
C3	STR$
C4	STRING$(
8A	TAB(
B7	TAN
8C	THEN
91	TIME (left)
D1	TIME (right)
B8	TO
FC	TRACE
B9	TRUE
FD	UNTIL
BA	USR
BB	VAL
EF	VDU
BC	VPOS
FE	WIDTH

Appendix E – Operating System Calls and Vectors

Routine		Vector		Function
Name	Addr	Name	Addr	
		USERV	200	The user vector
		BRKV	202	The BRK vector
		IRQ1V	204	Primary interrupt vector
		IRQ2V	206	Unrecognised IRQ
OSCLI	FFF7	CLIV	208	Command line interpreter
OSBYTE	FFF4	BYTEV	20A	*FX/OSBYTE call
OSWORD	FFF1	WORDV	20C	OSWORD call
OSWRCH	FFEE	WRCHV	20E	Write character
OSNEWL	FFE7	–	–	Write LF,CR to screen
OSASCI	FFE3	–	–	Write character, &0D=LF,CR
OSRDCH	FFE0	RDCHV	210	Read character
OSFILE	FFDD	FILEV	212	Load/save file
OSARGS	FFDA	ARGSV	214	Load/save file data
OSBGET	FFD7	BGETV	216	Get byte from file
OSBPUT	FFD4	BPUTV	218	Put byte in file
OSGBPB	FFD1	GBPBV	21A	Multiple BPUT/BGET
OSFIND	FFCE	FINDV	21C	Open or close file
		FSCV	21E	File system control
		EVNTV	220	Event vector
		UPTV	222	User print routine
		NETV	224	Econet vector
		VDUV	226	Unrecognised VDU commands
		KEYV	228	Keyboard vector
		INSV	22A	Insert into buffer
		REMV	22C	Remove from buffer
		CNPV	22E	Count/purge buffer
		IND1V	230	Spare vector
		IND2V	232	Spare vector
		IND3V	234	Spare vector
NVWRCH	FFCB	–	–	Non-vectored write char.
NVRDCH	FFC8	–	–	Non-vectored read char.
GSREAD	FFC5	–	–	Read char. from string
GSINIT	FFC2	–	–	String input initialise
OSEVEN	FFBF	–	–	Generate an event
OSRDRM	FFB9	–	–	Read byte in paged ROM

Appendix F – OSBYTE/*FX Call Summary

dec.	hex.	function
0	0	Identify OS version
1	1	Set the user flag
2	2	Select input stream
3	3	Select output stream
4	4	Enable/disable cursor editing
5	5	Select printer destination
6	6	Set character ignored by printer
7	7	Set RS423 baud rate for receiving data
8	8	Set RS423 baud rate for data transmission
9	9	Set flashing colour mark state
10	A	Set flashing colour space state
11	B	Set keyboard auto-repeat delay
12	C	Set keyboard auto-repeat rate
13	D	Disable events
14	E	Enable events
15	F	Flush selected buffer class
16	10	Select ADC channels to be sampled
17	11	Force an ADC conversion
18	12	Reset soft keys
19	13	Wait for vertical sync
20	14	Explode soft character RAM allocation
21	15	Flush specific buffer
22	16	Electron increment ROM polling semaphore
23	17	Electron decrement ROM polling semaphore
24	18	Electron change sound system.
50	32	Econet poll transmit block
51	33	Econet poll receive block
52	34	Econet delete receive block
53	35	Econet sever remote connection
111	6F	Aries RAM board OSBYTE
115	73	Electron blank/restore palette
116	74	Electron reset internal sound system
117	75	Read VDU status
118	76	Reflect keyboard status in LEDs

119	77	Close any SPOOL or EXEC files
120	78	Write current keys pressed information
121	79	Perform keyboard scan
122	7A	Perform keyboard scan from 16 (&10)
123	7B	Inform OS, printer driver going dormant
124	7C	Clear ESCAPE condition
125	7D	Set ESCAPE condition
126	7E	Acknowledge detection of ESCAPE condition
127	7F	Check for EOF on an open file
128	80	Read ADC channel or get buffer status
129	81	Read key with time limit
130	82	Read machine high order address
131	83	Read top of OS RAM address (OSHWM)
132	84	Read bottom of display RAM address (HIMEM)
133	85	Read bottom of display address, given MODE
134	86	Read text cursor position (POS and VPOS)
135	87	Read character at cursor position + MODE
136	88	Perform *CODE
137	89	Perform *MOTOR
138	8A	Insert value into buffer
139	8B	Perform *OPT
140	8C	Perform *TAPE
141	8D	Perform *ROM
142	8E	Enter language ROM
143	8F	Issue paged ROM service call
144	90	Perform *TV
145	91	Get character from buffer
146	92	Read from FRED, 1 MHz bus
147	93	Write to FRED, 1 MHz bus
148	94	Read from JIM, 1 MHz bus
149	95	Write to JIM, 1 MHz bus
150	96	Read from SHEILA, mapped I/O
151	97	Write to SHEILA, mapped I/O
152	98	Examine buffer status
153	99	Insert character into input buffer
154	9A	Write to video ULA control register and copy
155	9B	Write to video ULA palette register and copy
156	9C	Read/write 6850 control register and copy
157	9D	Fast Tube BPUT
158	9E	Read from speech processor
159	9F	Write to speech processor
160	A0	Read VDU variable value

166	A6	Read address of OS variables (low byte)
167	A7	Read address of OS variables (high byte)
168	A8	Read address of ROM pointer table (low byte)
169	A9	Read address of ROM pointer table (high byte)
170	AA	Read address of ROM info table (low byte)
171	AB	Read address of ROM info table (high byte)
172	AC	Read address of key translation table (low byte)
173	AD	Read address of key translation table (high byte)
174	AE	Read address of OS VDU variables (low byte)
175	AF	Read address of OS VDU variables (high byte)
176	B0	Read/write CFS timeout counter
177	B1	Read/write input source
178	B2	Undefined
179	B3	Read/write primary OSHWM
180	B4	Read current OSHWM
181	B5	Read/write RS423 mode
182	B6	Read character definition explosion state
183	B7	Read cassette/ROM filing system switch
184	B8	BBC Read RAM copy of video ULA control register Electron undefined
185	B9	BBC Read RAM copy of video ULA palette register Electron read/write paged ROM service call semaphore
186	BA	Read ROM number active at last BRK
187	BB	Read number of ROM socket containing BASIC
188	BC	Read current ADC channel
189	BD	Read maximum ADC channel number
190	BE	Read ADC conversion type
191	BF	Read/write RS423 use flag
192	C0	Read RS423 control flag
193	C1	Read/write flash counter
194	C2	Read/write space period count
195	C3	Read/write mark period count
196	C4	Read/write keyboard auto-repeat delay
197	C5	Read/write keyboard auto-repeat period
198	C6	Read *EXEC file handle
199	C7	Read/write *SPOOL file handle
200	C8	Read/write ESCAPE, BREAK effect
201	C9	Read/write Econet keyboard disable
202	CA	Read/write keyboard status byte
203	CB	Read/write RS423 handshake extent

204	CC	Read/write RS423 input suppression flag
205	CD	Read/write cassette/RS423 selection flag
206	CE	Read/write Econet OS call interception status
207	CF	Read/write Econet OSRDCH interception status
208	D0	Read/write Econet OSWRCH interception status
209	D1	Read/write speech suppression status
210	D2	Read/write sound suppression status
211	D3	Read/write BELL channel
212	D4	Read/write BELL envelope number/amplitude
213	D5	Read/write BELL frequency
214	D6	Read/write BELL duration
215	D7	Read/write startup message and !BOOT options
216	D8	Read/write length of soft key string
217	D9	Read/write lines printed since last page
218	DA	Read/write number of items in VDU queue
219	DB	Read/write TAB character value
220	DC	Read/write ESCAPE character value
221	DD	Read/write character &C0 to &CF status
222	DE	Read/write character &D0 to &DF status
223	DF	Read/write character &E0 to &EF status
224	E0	Read/write character &F0 to &FF status
225	E1	Read/write function key status
226	E2	Read/write SHIFT+function key status
227	E3	Read/write CTRL+function key status
228	E4	Read/write CTRL+SHIFT+function key status
229	E5	Read/write ESCAPE key status
230	E6	Read/write flags determining ESCAPE effects
231	E7	BBC Read/write IRQ bit mask for user 6522 Electron reserved
232	E8	BBC Read/write IRQ bit mask for 6850 Electron Read/write sound semaphore
233	E9	BBC Read/write IRQ bit mask for system 6522 Electron Read/write soft key pointer
234	EA	Read flag indicating Tube presence
235	EB	Read speech processor presence flag
236	EC	Read/write WRCH destination status
237	ED	Read/write cursor editing status
238	EE	Read/write OS workspace byte
239	EF	Read/write OS workspace byte
240	F0	Read country code
241	F1	Read/write user flag
242	F2	BBC Read RAM copy of serial processor ULA Electron read RAM copy of &FE07

243	F3	Read timer switch state
244	F4	Read/write soft key consistency flag
245	F5	Read/write printer destination flag
246	F6	Read/write character ignored by printer
247	F7	Read/write BREAK intercept code, 1st byte
248	F8	Read/write BREAK intercept code, 2nd byte
249	F9	Read/write BREAK intercept code, 3rd byte
250	FA	Read/write OS workspace byte
251	FB	Read/write OS workspace byte
252	FC	Read/write current language ROM number
253	FD	Read/write last BREAK type
254	FE	Read/write available RAM
255	FF	Read/write start up options

Appendix G – Variable locations

For the format of these variables, see section 3.1.

Resident integers

@%	&0400	I%	&0424	R%	&0448
A%	&0404	J%	&0428	S%	&044C
B%	&0408	K%	&042C	T%	&0450
C%	&040C	L%	&0430	U%	&0454
D%	&0410	M%	&0434	V%	&0458
E%	&0414	N%	&0438	W%	&045C
F%	&0418	O%	&043C	X%	&0460
G%	&041C	P%	&0440	Y%	&0464
H%	&0420	Q%	&0444	Z%	&0468

Variable list base pointers

The pointers marked with a '*' are not available (those characters are not allowed as part of a variable name).

@	&0480*	T	&04A8	h	&04D0
A	&0482	U	&04AA	i	&04D2
B	&0484	V	&04AC	j	&04D4
C	&0486	W	&04AE	k	&04D6
D	&0488	X	&04B0	l	&04D8
E	&048A	Y	&04B2	m	&04DA
F	&048C	Z	&04B4	n	&04DC
G	&048E	[	&04B6*	o	&04DE
H	&0490	\	&04B8*	p	&04E0
I	&0492	]	&04BA*	q	&04E2
J	&0494	^	&04BC*	r	&04E4
K	&0496	_	&04BE	s	&04E6
L	&0498	£	&04C0	t	&04E8
M	&049A	a	&04C2	u	&04EA
N	&049C	b	&04C4	v	&04EC
O	&049E	c	&04C6	w	&04EE
P	&04A0	d	&04C8	x	&04F0
Q	&04A2	e	&04CA	y	&04F2
R	&04A4	f	&04CC	z	&04F4
S	&04A6	g	&04CE		

Bibliography

Acorn User Magazine, published monthly, Redwood Publishing

6502 Assembly Language Programming, L.A.Leventhal, OSBORNE/McGraw Hill, Berkeley, California

Acorn Electron User Guide, Acorn Computers Limited, Cambridge, 1983

Acorn Electron Advanced User Guide, Dickens and Holmes, Adder Publishing/Acornsoft, Cambridge, 1984

The BBC Microcomputer User Guide, John Coll, British Broadcasting Corporation, London, 1982

The Advanced User Guide for the BBC Microcomputer, Bray, Dickens and Holmes, Cambridge Micro Centre, 1983

Beebug Magazine, published every five weeks, BEEBUG, PO Box 109, High Wycombe, Bucks.

Principles of Compiler Design, Aho and Ullman, Addison Wesley, 1979

Programming the 6502, Rodnay Zaks, Sybex, 1980

R650X and R651X Microprocessors (CPU) Data Sheet, Rockwell International, 1984

Understanding and Writing Compilers, Richard Bornat, Macmillan Press, 1979

Glossary

Accumulator – a register used to perform mathematical operations. The 6502 has one accumulator, A, which can deal with 8-bit integers.

Addressing Mode – specifies how any data will be used by a machine code instruction.

ASCII (American Standard Code for Information Interchange) – the ASCII code of a character is the value of the byte which is used to store it in the computer.

Assembler – a program which converts a series of mnemonics into a machine code program.

Bit of memory – this is the fundamental unit of a computer's memory. It may only be in one of two possible states, usually represented by a 0 or 1.

BNF (Backus Naur Form) – a way of writing down the syntax of a computer language.

Buffer – a software buffer is an area of memory set aside for data in the process of being transferred from one device or piece of software to another.

Byte of memory – 8 bits of memory. Data is normally transferred between devices one byte at a time over the data bus.

Chip – derived from the small piece of silicon wafer or chip which has all of the computer logic circuits etched into it. A chip is normally packaged in a black plastic case with small metal leads to connect it to the outside world.

Command – similar to a BASIC statement, but it can only be executed if it is typed in at the keyboard directly (i.e. in *command mode*), rather than as part of a BASIC program. For example, 'AUTO' is a command.

CPU (Central processing unit) – the 6502A in the BBC microcomputer and the Electron. It is this chip which does all of the computing work associated with running programs.

Disassembler – a program which converts a series of bytes in a machine code program into assembler mnemonics.

Field – a space allocated for some data in a register, or in a program listing, or in a storage area. For example, in a Variable Descriptor Block, the first field contains a pointer to the location of the variable, and the second field contains the type of the variable.

Flag – a bit (or byte) which is used to signal a particular condition. For example, the N (negative) flag in the 6502 is set if the number just calculated is negative.

Heap – BASIC uses a HEAP to store the variables used during a program. Data can be added on top of a heap, but once used, the space cannot be recovered until it is completely cleared.

High – sometimes used to designate logic '1'

Indirection – pointing to a variable in memory with the indirection operators ?, ! or $, rather than using a value directly. For example, !&4000 points to the 4-byte integer variable in locations &4000 to &4003.

Interrupt – this signal is produced by peripheral devices and is always directed to the 6502A CPU. Upon receiving an interrupt, the 6502 will normally run a special interrupt routine program before continuing with the task in hand before it was interrupted.

Keyword – a special word (sometimes called a *Reserved Word*) which BASIC uses for a special purpose. For example, PRINT is a keyword which is put before items to be printed out.

Linked list – a list of items in memory, where each item contains a pointer to the next one. The end of the list is usually marked by a null pointer in the last item. A base pointer is used to point to the first item in the list.

Low – sometimes used to designate logic '0'.

Machine code – the programs produced by the 6502 BASIC Assembler are machine code. A machine code program consists of a series of bytes in memory which the 6502 can execute directly.

Mnemonic – the name given to the text string which defines a particular 6502 operation in the BASIC assembler. LDA is a mnemonic which means *load accumulator*.

Opcode – the name given to the binary code of a 6502 instruction. For example, &AD is the opcode which means *load accumulator* (absolute addressing).

Operand – a piece of data on which some operation is performed. This could be a number in a BASIC program, or it could be a byte in the accumulator of the 6502.

Operator – a symbol or device which takes one or two *operands* to produce a single result. If an operator takes one operand, it is a *unary* operator; if it takes two operands, it is a *binary* operator. For example, the '$' operator takes the number following it, and gives as a result the static string at that location.

Overflow – a condition caused when the result of a calculation is too large to be represented properly.

Overlay – a part of a program which is loaded into memory while the main program is running. Large programs can be run in a computer by splitting them up into several overlays, and each one will only be loaded in when they are needed.

Page – a page of memory in the 6502 memory map is &100 (256) bytes long. There are therefore 256 pages in the entire address space. 256 pages of 256 bytes each account for the 65536 bytes of addressable memory.

Page zero – the locations from &0000 to &00FF. These are very useful on the 6502, because any machine code instructions which use them are shorter and faster than those which use any other section of the memory.

Peripheral – any device connected to the 6502 central processor unit, such as the printer port, disc interface etc., but not including the memory.

Program – a BASIC program is a sequence of statements which the BASIC interpreter is to execute one after the other. A machine code program is a sequence of bytes which the 6502 is to execute one after the other as machine code instructions.

RAM (Random Access Memory) – the main memory in the BBC microcomputer and the Electron is RAM because it can be both written to and read from.

Register – a location which can be written to or read from, usually for a special purpose, but which is not necessarily in the main memory map of the computer. The 6502 and peripheral devices contain registers, and BASIC uses a series of page zero locations as if they were its own registers.

ROM (Read only memory) – as the name implies, ROM can only be read from and cannot be modified by being written to.

Stack – the 6502 and BASIC each use a stack for temporary storage of data. Data is pushed onto a stack in sequence, then removed by pulling the data off the stack. The last byte to be pushed is the first byte to be pulled off again. The 6502 stack is used to store return addresses from subroutines; the BASIC stack is used to store temporary results during a calculation, and other data inside a PROC or FN call.

Statement – a sequence of symbols which tells the BASIC interpreter to perform a certain action. For example, the statement ‘A=10’ tells BASIC to assign the value 10 to the variable ‘A’ (this is an *assignment statement*).

Static string – a string whose characters are stored in memory starting at a fixed location. The string is terminated by a &0D byte (carriage return character), which is not counted as one of the characters of the string. For example, $&2000 is the static string whose first character is stored in location &2000.

String Information Block – this block is used to reference the characters of a dynamic string on the BASIC HEAP. It contains a pointer to the start of the string, the amount of memory allocated to the string, and the current length of the string. The *String Information Block* is held in the *value field* of the *Variable Information Block* of a string variable.

Token – a single byte which is used by BASIC to represent a keyword. This saves memory when programs are stored. For example, &80 is the token for 'AND'.

Variable – is used to hold a number or a string (depending on its type). Named variables are stored on the BASIC HEAP (or in page 4 if they are resident integer variables), but indirected variables (accessed using the $, ? and ! operators) can be anywhere in memory.

Variable Descriptor Block – this is passed between routines inside BASIC as a description of a variable, once its location and type has been found. It consists of a pointer to the value of the variable, and a byte which gives the type of the variable.

Variable Information Block – the format used to store variables (and FN/PROC locations) on the BASIC HEAP. It consists of a pointer to the next *Variable Information Block*, the name of the variable, and the value of the variable.

Index

C

D

E

F

G

H

I

K

L

M

N

R

S

W

Y

Z

BASIC ROM USER GUIDE
for the BBC Microcomputer and Acorn Electron

This book contains a detailed description of the BASIC system used on the BBC Microcomputer and Acorn Electron. It covers the operation of BBC BASIC I, BBC BASIC II and Electron BASIC, and enables the serious programmer to considerably enhance the facilities of his machine.

A number of useful examples are provided including a complete disassembler, and various facilities such as listing active variables and overlaying procedures are described.

Extensive reference sections cover the ROM routines and error recovery, including changing MODE inside procedures and salvaging bad programs.

ISBN 0 947929 04 5