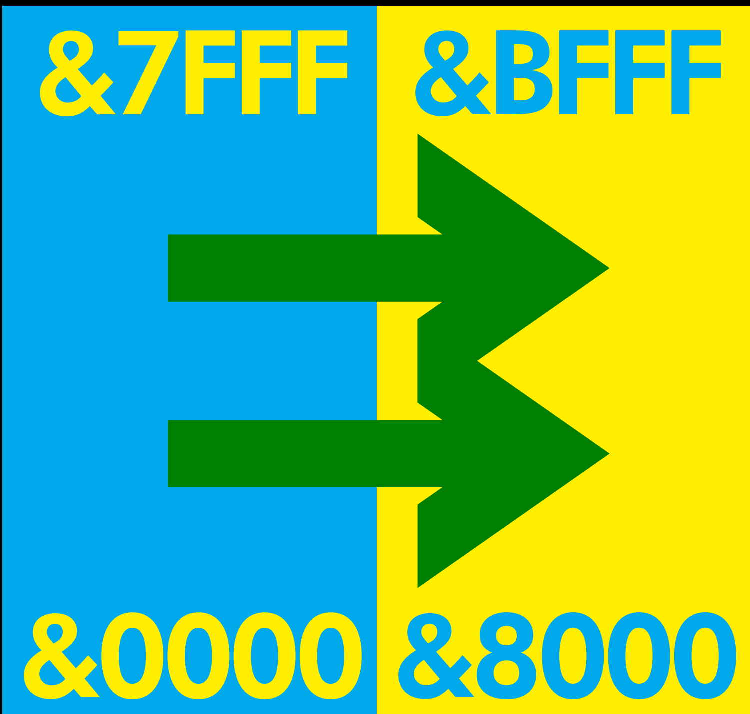
THE ADVANCED BASIC ROM USER GUIDE

FOR THE BBC MICRO



Published by the Cambridge Microcomputer Centre

COLIN PHARO

The Advanced BASIC ROM User Guide

for the BBC Microcomputer

Colin Pharo B.Sc.,MBCS

published by the Cambridge Microcomputer Centre

Published in the United Kingdom by:
The Cambridge Microcomputer Centre,
153–154 East Road,
Cambridge,
England

Telephone (0223) 355404
Tlx 817445
ISBN 0 946827 45 1

First published 1984

Remastered by dv8 in 2017
Second revision March 2023

The latest revision of this book is available at:
https://stardot.org.uk/forums/viewtopic.php?t=13860

The Author would like to thank Dan Nanayakkara, Alex Van Someren and Peter Wederell for their assistance in the production of this book.

This book was computer typeset by Computerset (MFK) Ltd of Saffron Walden.
Book production by CPS of Saffron Walden.

Printed by the Burlington Press (Cambridge) Ltd., Foxton, Cambridge.

Contents

INTRODUCTION

Most people who program microcomputers in BASIC are soon disheartened by its shortcomings. Firstly, there are some applications which can only be sensibly written in assembly language. Secondly, BASIC can be slow. This is not to say that BBC BASIC is badly written. On the contrary, it runs faster than most other dialects of the language. The slowness is simply inherent in the language. Thirdly, programs written in BASIC consume a lot of memory. For the BBC Micro, this problem becomes acute in the graphics modes which leave precious little space for a program. BASIC is an interpreter and not a compiler and many of its shortcomings are attributable to that fact.

A compiler verifies the user program (source code) in a separate compilation step, generating a machine code version (usually) of that program known as the object code. The object code is stripped of all comments. It is the object code which is executed at run time.

An interpreter, however, acts on the source code at run time. It handles the code in a line-by-line fashion, checking syntax and parsing each line every time it is executed. This increases the execution time of the program. It is also wastes space, since the source code has to be present in memory. Thus interpreters use more memory and run more slowly than compiled object code.

The solution to the problem appears to be obvious — buy a compiler! There are, however, problems associated with compilers also. If the compiler is disk or tape based, then it must be loaded into memory for the compilation step (though not for execution). This limits the size of the source code that can be written. A ROM based compiler gets round this problem. Be wary though! Most of the compilers for the BBC micro do not generate machine code object programs, but rather an intermediate code which can be interpreted into machine code at run time. This means that the object code will only run on micros equipped with the same compiler. Another problem to be considered with compilers is their efficiency. A compiler generates a number of machine code instructions for each source language statement. The efficiency of a compiler can be thought of as the ratio of the minimum number of machine code instructions needed to perform a task, to the number generated by the compiler from source code written to perform that same task. As a general rule, the more friendly the language, the less efficient the compiler.

There is a compiler that all BBC micro owners possess. It's called the assembler. The distinction between assemblers and compilers is that the former have a one-to-one relationship with the machine code that they generate; that is to say each assembly language command generates a single machine code instruction. The drawback is that assembly language programming can be difficult, time-consuming and error prone. Moreover, assemblers do not have in-built commands to handle sines, cosines, square roots, random numbers etc.

On the face of it, the user appears to be caught in a cleft stick. The user can either choose BASIC for its ease of use, tolerating speed and size problems, or assembly language, foregoing floating-point arithmetic, trigonometry and so forth.

Fortunately, the BBC BASIC interpreter consists of a large number of small, machine code subroutines, many of which could usefully be invoked from an assembly language program. In this book, 69 such subroutines are described covering 32-bit integer arithmetic, floating-point arithmetic, trigonometry and so forth.

The approach has several advantages:

a) The subroutines are ROM based and occupy no valuable RAM.

b) The subroutines are tried and tested.

c) Functions such as sine, cosine, square root and random numbers can be used in assembly language programs without the need to write this code from scratch.

d) The subroutines frequently incorporate useful error reporting.

e) Object code will run faster and occupy less memory than the equivalent BASIC code.

It is only fair to point out the disadvantages of this approach:

a) This book covers BASIC 1 and BASIC 2 and the technique described will work only on these two ROMs. It will not work with HI BASIC (6502 second processor) or with US BASIC (US BBC micros). Neither will it work with any future releases of BASIC that may be issued. Consequently, it would be most unwise to use the technique directly in any program which is destined for sale to the public, or where the user plans to upgrade the BASIC ROM within the life time of the program. Even in these cases, however, this book has distinct value.

Many of the subroutines in the BASIC ROMs are written with an elegance and tightness which is unlikely to be surpassed by the user. A study of these subroutines will undoubtedly profit the user who is forced to write similar code. There will be many instances, however, when a user may wish to use the technique directly. It would be a pity to deprive all users of an effective and time-saving technique, simply because the technique is not suitable for every occasion.

b) Because assembly language is used, the source code is often more long-winded to write than the equivalent BASIC.

c) The technique is not a cure-all for all problems. Although the user is offered an easy path to sines and cosines, not much more than 10% of the execution time can be saved for these complicated functions. Chapter 9 provides some useful tips in this respect.

d) The user must have some acquaintance with assembly language programming and such knowledge is assumed in this book.

This then is not a primer in assembler. The bulk of this book contains descriptions of BASIC subroutines, their entry points, timings and set-up conditions. This is supported by fully documented and tested examples of code, making it possible for relatively inexperienced assembly language programmers to use the techniques to develop quite sophisticated applications.

The book also contains in places descriptions of the theory necessary for a full understanding of the technique advocated. All programmers inevitably make mistakes and this knowledge is indispensable when debugging those mistakes.

Much of this book is to do with numbers and their representation within the BBC micro. A feel for binary and hexadecimal numbering systems is necessary to understand completely the text presented here. In particular, floating point numbers require that the concept of the binary point be grasped. Therefore, the next chapter is devoted to numbering systems. Experienced programmers will doubtless skip over this chapter. It is specifically intended for the less experienced programmer and for this reason emphasises the ‘why’ as well as the ‘how’ of numbering systems.

1 NUMBERING SYSTEMS

Since it is in every day use, the decimal numbering system is widely understood. It is based on (has a radix of) ten. The radix of a numbering system embodies several important concepts:

a) it is the number of possible values that a single digit can contain. (decimal values are 0 to 9, ten values in all).

b) it is one more than the maximum value of a single digit.

c) each digit in the number will represent some power of the radix. It is the position of the digit within the number which determines the power of the radix by which the digit is multiplied. For example, the decimal number 1932.74 =

10*10*10	10*10	10	1	1/10	1/100
1	9	3	2	7	4

It will be shown that the decimal system is not ideally suited to computers. Fortunately, the alternative numbering systems which suit computers best use the same principles as the decimal system and so they are not difficult to master.

1.1 The Binary System

A computer is an electronic machine. It knows nothing of numbers. It consists of many components, each of which is designed to respond to pulses of electricity, providing the pulses arrive in a particular pattern and at a particular time. At the heart of the computer lies a central processor unit (cpu) which not only performs the arithmetic functions requested by a program, but also exercises control over other component parts of the computer.

At any given instant in time, an electrical pulse in a circuit may either be present or absent. In other words, a circuit can be in one of two states. This is a binary (meaning based on two) system. It is human beings that allocate numbers to these two states. The binary system has only two numbers, 0 and 1. 0 is taken to mean that the pulse is absent, whilst 1 denotes its presence.

As an example of these pulses, each cpu has a repertoire of commands that it can perform, known as its instruction set. At specific intervals of time, the cpu will interpret a pattern of pulses as a command to perform a particular function. This function could be to add or to store etc. The pulses arrive simultaneously down eight wires which collectively are called the data bus. For example:

```
data bus    pulse      binary
--------    -------    ------
  wire 7    present         1   this
  wire 6     absent         0   pattern
  wire 5     absent         0   of
  wire 4     absent         0   pulses
  wire 3    present         1   is
  wire 2     absent         0   the
  wire 1     absent         0   DEY
  wire 0     absent         0   instruction
```

A computer program causes pulses just like this to move from one part of the computer to another (albeit with a good deal of help from both language software and the operating system). In the example above, the individual pulses are known as bits (binary digits) and the collection of 8 bits that were sent down the data bus is called a byte.

Thus a computer, its design constrained by the laws of physics, operates in a binary fashion. It follows that it will be easier to work with computers if the programmer achieves some expertise with the binary system also.

In the binary system, each digit represents some power of two, such that the binary number 01101000, for example, is equivalent to:

```
(0x128) +

(1x64)  +

(1x32)  +

(0x16)  +

(1x8)   +

(0x4)   +

(0x2)   +

(0x1)

= 104 in decimal.
```

1.1.1 Binary Addition

Binary addition follows rules analogous to decimal addition. Each time the sum of a column is two or more (rather than ten or more) there is a carry to the next column. Since there are only two digits in the binary system, it is possible to define very simple rules for binary addition:

```
0+0   = 0

0+1   = 1

1+1   = 0 (carry 1)

1+1+1 = 1 (carry 1)
```

These simple rules can be applied to much larger binary numbers. For example:

```
128 64 32 16 8 4 2 1   decimal
--- -- -- -- - - - -   -------
  0  1  1  0 1 0 0 0       104
  0  1  0  1 1 1 0 1        93 +
--------------------   -------
  1  1  0  0 0 1 0 1       197
--------------------   -------
```

1.1.2 Binary Subtraction

Simple rules can also be defined for binary subtraction:

```
0-0 = 0

1-0 = 1

0-1 = 1 (borrow 1)

1-1 = 0
```

Likewise these simple rules can be applied to the subtraction of larger binary numbers. For example:

```
128 64 32 16 8 4 2 1   decimal
--- -- -- -- - - - -   -------
  0  1  1  0 1 0 0 0       104
  0  1  0  1 1 1 0 1        93 -
--------------------   -------
  0  0  0  0 1 0 1 1        11
--------------------   -------
```

1.1.3 Negative Binary Numbers

So far only positive, binary integers have been considered. Negative binary integers are held in twos complement form, for reasons which will soon become apparent. To convert a binary number to twos complement form, it is only necessary to change all zeroes to ones and all ones to zeroes, adding 1 to the result. In the example above, the binary for decimal 93 was seen to be 01011101.

To obtain the binary for −93:

```
01011101 = +93

10100010 = change 0 to 1 and vice-versa
00000001   add 1
--------
10100011 = -93
--------
```

1.1.4 Binary Subtraction By Addition

The twos complement form for negative numbers makes unnecessary any subtraction circuitry in the cpu. Instead, subtraction can be achieved via the cpu's adder. In the example used for subtraction, 104 − 93 is the same as 104 + (−93). Therefore by converting 93 to its twos complement form, addition can be used to achieve subtraction providing any final carry is ignored:

```
                01101000   104
                10100011   -93 +
                --------   ---
ignore carry 1  00001011    11
                --------   ---
```

1.1.5 Binary Fractions

Binary fractions follow concepts analogous to decimal fractions. In decimal fractions each digit position represents some power of 1/10. Thus the decimal fraction 0.875 =

```
8      7     5
--   + --- + ----
10     100   1000
```

With binary fractions, each digit position represents some power of 1/2 and the point is known as the binary point (rather than decimal point) to emphasise this fact. Thus the binary equivalent of 0.875 is 0.111 =

```
1     1     1
-  +  -  +  -
2     4     8
```

1.2 The Hexadecimal System

Binary numbers have an obvious disadvantage. It takes a large number of zeroes and ones to represent even comparatively small decimal numbers. But computers work in a binary fashion and conversion between the two radices is a laborious process. What is required is some shorthand version of binary.

For a numbering system to relate directly to binary, it must be based on some power of two. The choice of numbering system is mainly determined by the number of bits in a byte. In practice the radix chosen fulfils the following conditions:

a) it is a power of 2
b) it is as close to 10 as possible
c) it subdivides a byte into equal proportions (usually halves but not always).

A BBC micro has 8 bits in a byte. Therefore, a byte subdivides into two equal size quartets. This fixes the numbering system as hexadecimal with a radix of 16 (one more than the maximum decimal number that can be stored in four bits). Had there been 6 bits in a byte, the byte would have been regarded as two triplets and the numbering system would have been octal (based on eight).

There are sixteen digits in the hexadecimal system and a symbol is required to represent each digit. Clearly for the digits 0 to 9, the same symbols (0 to 9) can be used. The single hexadecimal digits that represent decimal numbers 10 to 15 are a bit more of a problem. In fact the symbols A to F are allocated to these numbers. When used in this way, these symbols should not be confused with the ASCII letters 'A' to 'F'. The meaning is always understood because in both BASIC and assembly language, hexadecimal numbers are preceded by an ampersand (&).

Thus in a single byte, the range of hexadecimal numbers that can be stored is from &00 to &FF (0 to 255), with each quartet holding from &0 to &F (0 to 15) as shown below:

```
binary  hex  dec    binary  hex  dec
------  ---  ---    ------  ---  ---
  0000   &0    0      1000   &8    8
  0001   &1    1      1001   &9    9
  0010   &2    2      1010   &A   10
  0011   &3    3      1011   &B   11
  0100   &4    4      1100   &C   12
  0101   &5    5      1101   &D   13
  0110   &6    6      1110   &E   14
  0111   &7    7      1111   &F   15
```

1.2.1 Binary/Hexadecimal Conversion

The most important property of a hexadecimal number is that it can be derived instantly from a binary number. To do this, the binary number is subdivided into quartets starting at the least significant end. If the most significant bit(s) are not a quartet, they should be padded with leading zeroes to make an exact quartet. Hexadecimal can then be substituted for each quartet individually. For example, consider the binary number 11101011010010011. Splitting into quartets gives:

```
           0001 1101 0110 1001 0011

hex =         1    D    6    9    3

    =         &1D693
```

The converse operation from hexadecimal to binary is equally simple. It is impossible to do anything so simple with decimal numbers.

1.2.2 The Roundness of Hexadecimal

There is another bonus obtained by using hexadecimal. Because the design of the computer is based on the number 2, hexadecimal frequently results in an easily remembered, round number where the decimal equivalent does not. One example of this is found in memory addressing:

	hex	dec
total BBC model B memory	10000	65536
1K of memory	400	1024
one page of memory	100	256
PAGE (no disks/Econet)	E00	3584
start of BASIC	8000	32768
start of MOS	C000	49152

Another example can be found in the ASCII character set:

```
From &00 to &1F = control characters     (VDU 0 to VDU 31)
From &30 to &39 = 0 to 9
From &41 to &5A = A to Z
From &61 to &8A = a to z
```

It will be seen that an ASCII number is the number plus &30. The upper-case letters are the number of the letter in the alphabet plus &40, whilst lower-case letters are the number of the letter in the alphabet plus &60. The control characters, representing VDU 0 to VDU 31, can be entered from the keyboard by using CTRL and another key. The action of the CTRL key is to subtract &40 from that other key. Thus VDU 1 = CTRL A, VDU 2 = CTRL B etc.

1.2.3 Hexadecimal Addition

In hexadecimal addition there is a carry to the next more significant digit whenever the sum exceeds decimal 16. Thus the highest number that can exist in any digit is &F (15). Consider the addition of &A and &B, for example. Since &A is equivalent to decimal 10 and &B is equivalent to decimal 11, the decimal sum is 21. As this is greater than decimal 16, 16 must be subtracted and a carry must be generated to the next more significant digit. Thus the hexadecimal sum is &15. The right hand digit (5) is obtained from 21 – 16, whilst the left hand digit is the carry.

1.2.4 Negative Hexadecimal Numbers

The hexadecimal complement form is derived by subtracting the number from a string of &F's and then adding 1. The number of &F's in the string depends on the precision of arithmetic in question. Thus in 16 bit arithmetic, &FFFF would be used, whilst in 32 bit arithmetic &FFFFFFFF would be used. As an example, the 16 bit hexadecimal complement of &ABC is derived as follows:

```
&FFFF
& ABC -
-----
&F543
&   1 +
-----
&F544 result
-----
```

1.2.5 Hexadecimal Fractions

In hexadecimal fractions, each digit position represents some power of 1/16. The point is known as the hexadecimal point (rather than decimal point) to emphasise this fact. Thus the decimal fraction 0.875, which is 14/16, has a hexadecimal equivalent of 0.E.

2 INTEGERS

In BASIC all integer fields occupy four bytes (32 bits). This allows for positive integers from &00000000 to &7FFFFFFF (0 to 2,147,483,647). Negative numbers range from &FFFFFFFF to &80000000 (−1 to −2,147,483,648).

BASIC recognises an integer variable by the % at the end of its name. Some of these variables, known as the resident integer variables, @% and A% to Z%, occupy fixed locations in RAM. Other integer variables are located as referenced in an area of RAM following the program text. The assembler programmer is free to use the resident integer variables. As with BASIC, these memory areas can be used to pass parameters from one program to another. However, O% and P% have special significance as location counters in assembly language and must not be used (O% can safely be used by BASIC 1 users). It is also inadvisable to use @% which controls BASIC print formatting. The assembler program accesses the resident integer variables by address rather than by name. A list of these addresses may be found in Chapter 7.

2.1 Integer Work Areas

Integer numbers are always stored with the least significant byte first and the most significant byte last. Thus the number &12345678 would be held as:

```
byte 0 = &78

byte 1 = &56

byte 2 = &34

byte 3 = &12
```

The BASIC interpreter performs all of its integer arithmetic in four bytes of working storage in Page Zero. The four bytes are &2A, &2B, &2C and &2D. From henceforth, these four bytes will be referred to as the Integer Working Area or IWA. These four bytes are also used by the interpreter for other purposes. When used as the IWA, the normal rules for integer variables are obeyed. &2A contains the least significant byte, whilst &2D contains the most significant byte.

BASIC has its own stack located immediately below HIMEM. Like the processor stack, it runs downwards in memory. The stack is maintained by a set of subroutines within the BASIC ROM. A stack pointer is held in &4 and &5 (lo,hi). Generally, whenever BASIC has two integer fields to process, one will be located in the IWA while the other is in the stack, pointed to by &4 and &5. When using BASIC routines we shall usually load one integer into the IWA and point &4,&5 at the other.

BASIC also has to keep track of whether it is processing integers, floating points or strings. It achieves this in two ways. Firstly subroutines return a value in the A register as follows:

```
A =    0   processing a string
A =  &40   processing an integer
A =  &FF   processing floating point
```

It also stores the type of variable in &27 as follows:

```
&27 =   0   byte
&27 =   4   4 byte integer
&27 =   5   5 byte floating point
&27 = &81   string
&27 = &A4   function
&27 = &F2   procedure
```

From time to time, when using the 32-bit integer subroutines,it will be necessary to set either registers or memory areas to conform with the above.

2.2 Defining Integer Constants

For BASIC 2 owners, defining a 32-bit integer constant to the assembler is a matter of the utmost simplicity. The assembler directive EQUD is provided for this purpose. For example:

```
 10 DIM mc% 100
 20 FOR pass% = 0 TO 2 STEP 2
 30 P% = mc%
 40 [
 50 OPT pass%
 60 .constant EQUD 5000
 70 ]
 80 NEXT pass%
 90 STOP
```

EQUD is an assembler directive. This means that it instructs the assembler to perform a task at assembly time. This is quite different from an instruction mnemonic. Mnemonics are translated into machine code for execution at run time.

The EQUD directive instructs the assembler to reserve four bytes of memory at the current value of the location counter (P%). It automatically reverses the storage of data such that the least significant byte is stored first. Thus, since 5000 is &1388, the directive stores &88 at the current value of the location counter, &13 at the next address, and &00 at the next two addresses. It also steps the location counter by 4.

BASIC 1 users have to improvise to achieve the same effect. Since in BASIC 1 there is only one directive available, namely OPT, this must be pressed into service to provide an equivalent mechanism. Consider the following:

```
 10 DIM mc% 100
 20 FOR pass% = 0 TO 2 STEP 2
 30 P% = mc%
 40 [
 50 OPT pass%
 60 .constant                \ label
 70     OPT FNEQUD(5000)     \ pseudo-directive call
 80 ]
 90 NEXT pass%
100 STOP
110 DEF FNEQUD(A%)
120     !P% = A%
130     P%  = P% + 4
140 = pass%
```

At line 70, OPT is made to call a BASIC function. The value of OPT must not be changed, so the function must return the current value of OPT and indeed line 140 does this. The OPT directive ensures that the call to the function EQUD is performed at assembly time (not execution time). This is not a hybrid program. Once assembled, the BASIC function is no longer required. Within the function itself, statements may be included as required. In the EQUD function, the required constant is stored at the current value of the location counter. Because an integer variable, A%, is used to hold the argument (in this example 5000) passed by the assembler, the constant is automatically aligned with the least significant byte first. The counter is then stepped by 4. The constant thus set up may be referenced by the label on line 60.

The function, FNEQUD, is an example of a pseudo-directive, a technique extensively used in this book. BASIC 2 also has the following assembler directives:

```
EQUB   for 1 byte
EQUW   for 2 bytes (least significant byte first)
EQUS   for strings
```

BASIC 1 users can use the equivalent pseudo-directive functions below. Note that if the constant supplied to the function is too big, it will be truncated at the most significant end.

Both BASIC 1 and BASIC 2 users will benefit from the RESB pseudo directive which reserves a specified number of bytes (1st argument) and fills them with a specified character (2nd argument).

```
 10 DIM mc% 100
 20 FOR pass% = 0 TO 2 STEP 2
 30 P% = mc%
```

```
 40 [
 50 OPT pass%
 60 .constant1
 70     OPT FNEQUB(50)
 80 .constant2
 90     OPT FNEQUW(500)
100 .string1
110     OPT FNEQUS("Example of a string")
120 .reserve
130     OPT FNRESB(20,0) \ reserve 20 bytes of zero
140 ]
150 NEXT pass%
160 STOP
170 DEF FNEQUB(A%)
180     ?P% = A%
190     P%  = P% + 1
200 = pass%
210 DEF FNEQUW(A%)
220     ?P%     = A% MOD 256
230     ?(P%+1) = A% DIV 256
240     P%      = P% + 2
250 = pass%
260 DEF FNEQUS(A$)
270     $P% = A$
280     P%  = P% + LEN(A$)
290 = pass%
300 DEF FNRESB(A%,B%)
310     FOR I% = 1 TO A%
320     ?P% = B%
330     P%  = P% + 1
340     NEXT
350 = pass%
```

2.3 Integer Routines Summary

The following table summarises 32-bit integer routines available within the BASIC ROM. Conversion routines, e.g. integer to ASCII, are the subject of a later chapter. Unlike the floating point routines covered in the next chapter, the integer routines do not amount to a great deal of code. Moreover, set up procedures are sometimes more laborious than for floating point. In some applications, it may well be preferable to include the integer code within the user program, at the same time changing it slightly to make it more specific to the application. In this event, the user is advised to study the code available within the BASIC ROM, rather than re-invent it.

Name	BASIC 1 address	BASIC 2 address	Function
icomp	&ADB5	&AD93	IWA = -IWA
idiv	&9DE7	&9E0A	IWA = IWA DIV integer variable
iin	&B365	&B336	Copy integer variable to IWA
iminus	&9C9D	&9CC2	IWA = integer variable - IWA
imod	&9DDE	&9E01	IWA = IWA MOD integer variable
imult	&9D4A	&9D6D	IWA = IWA * integer variable
ineg1	&ACEA	&ACC4	IWA = -1
iout	&B4F2	&B4C6	Copy IWA to integer variable
iplus	&9C36	&9C5B	IWA = IWA + integer variable
ipos	&AD94	&AD71	Make IWA positive
ismall	&AF19	&AEEA	IWA = 256*Y + A
itest	&9A85	&9AAD	test integer variable = < > IWA
izero	&AEF9	&AECA	IWA = zero
izpin	&AF85	&AF56	Copy integer variable in zero page to IWA
izpout	&BE5C	&BE44	Copy IWA to integer variable in zero page

2.4 Integer Routines Description

The following pages show how to use each integer routine. Each of the routines handles signs correctly. For example, (−3)*(−9) gives the answer +27.

This assumes that the set up procedures specified for each routine are followed carefully. To ensure that there is no ambiguity, each routine is illustrated by a program. These programs are for demonstration purposes only and are not meaningful applications. They all use BASIC to print results partly because the material necessary to avoid this comes later in the book, and partly to present the technique in a way that is most simple to understand. All of the programs are written in BASIC 2, but conversion to BASIC 1 involves only:

a) replacing all routine addresses by their BASIC 1 equivalents.

b) using pseudo-directives instead of directives.

The use of most of the BASIC ROM's integer routines is natural, since the routines are structured in a way ideally suited to this purpose. However, DIV and MOD are exceptions and the method of using them has had to be contrived.

An important point arises from the interpreter's habit of re-using zero page locations for different purposes. This can cause difficulties in hybrid BASIC/assembly language programs. The following rules should be observed:

a) do not attempt to call these routines from BASIC language.

b) in hybrid programs, make sure that the results of any calculations are safely stored in memory areas within the domain of the program. Do not leave data in BASIC's zero page areas, if this data will be required later.

The following short program illustrates the sort of difficulties which can arise:

```
10 !&2A = 1000
20 PRINT !&2A
30 END

>RUN
    262186
```

These routine addresses, therefore, are provided specifically to assist assembly language programming and this is the recommended way of using them.

subroutine name	:	**icomp**
function	:	IWA = –IWA
BASIC 1 address	:	&ADB5
BASIC 2 address	:	&AD93
entry conditions	:	IWA contains a 32-bit integer
exit status	:	IWA complemented
	:	A destroyed
	:	X unchanged
	:	Y destroyed
	:	P destroyed
typical timing	:	31 microseconds

icomp demonstration (BASIC 2)

```
  0 icomp = &AD93
 10 DIM mc% 200
 20 FOR pass% = 0 TO 2 STEP 2
 30 P% = mc%
 40 [OPT pass%
 50 .constant    EQUD -1234  \ set up value
 60 .result      EQUD 0      \ result
 70 .start
 80   LDX #0                 \ zeroise loop counter
100 .loop1
110   LDA constant,X         \ get next byte of constant
120   STA &2A,X              \ save in IWA
130   INX                    \ bump loop counter
140   CPX #4                 \ end of loop ?
150   BCC loop1              \ no - back
160   JSR icomp              \ complement IWA
170   LDX #0                 \ zeroise loop counter
180 .loop2
190   LDA &2A,X              \ get next byte of IWA
200   STA result,X           \ save in result
210   INX                    \ bump loop counter
220   CPX #4                 \ end of loop ?
230   BCC loop2              \ no - back
240   RTS
250 ]
260 NEXT pass%
270 CALL start
280 PRINT !result
290 END

>RUN
      1234
```

subroutine name	:	**idiv**
function	:	IWA = IWA DIV integer variable
BASIC 1 address	:	&9DE7
BASIC 2 address	:	&9E0A
entry conditions	:	IWA contains dividend
	:	A% contains divisor
	:	&19,&1A (lo,hi) point to a string, ‘ A%’+RETURN
	:	&1B = 0
	:	&04,&05 (lo,hi) = HIMEM
	:	A = #&40
comments	:	DIV and MOD are not easy to extricate from the BASIC ROM and the set up procedures are necessarily somewhat contrived
exit status	:	IWA = quotient
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
error reports	:	division by zero
typical timing	:	794 microseconds

idiv demonstration (BASIC 2)

```
 10 idiv = &9E0A : DIM mc% 200
 20 FOR pass% = 0 TO 2 STEP 2
 30 P% = mc% : [OPT pass%
 40 .dividend EQUD 26
 50 .divisor  EQUD 3
 60 .fudge    EQUD &0D254120     \ " A%" + RETURN
 70 .result   EQUD 0
 80 .start    LDA &6             \ get HIMEM lo
 90           STA &4             \ set &4
100           LDA &7             \ get HIMEM hi
110           STA &5             \ set &5
120           LDA #fudge MOD 256 \ point &19
130           STA &19            \ at fudge lo
140           LDA #fudge DIV 256 \ point &1A
150           STA &1A            \ at fudge hi
160           LDX #0             \ zeroise loop counter
170           STX &1B            \ clear &1B
180 .loop1    LDA dividend,X     \ next byte of dividend
190           STA &2A,X          \ save in IWA
200           LDA divisor,X      \ next byte of divisor
210           STA &404,X         \ save in A%
220           INX                \ bump loop counter
230           CPX #4             \ end of loop ?
240           BCC loop1          \ no - back
250           LDA #&40           \ set integer
260           JSR idiv           \ call idiv
270           LDX #0             \ zeroise loop counter
280 .loop2    LDA &2A,X          \ next byte of IWA
290           STA result,X       \ save in result
300           INX                \ bump loop counter
310           CPX #4             \ end of loop ?
320           BCC loop2          \ no - back
330           RTS:]
340 NEXT pass% : CALL start : PRINT !result : END

>RUN
         8
```

subroutine name : **iin**

function : IWA = integer variable

BASIC 1 address : &B365

BASIC 2 address : &B336

entry conditions : &2A,&2B (lo,hi) points to integer variable

exit status : IWA set up

: A destroyed

: X unchanged

: Y destroyed

: P destroyed

typical timing : 34 microseconds

special note : Note that iin is shown for completeness only. The code represented by iin is trivial. The following code can be substituted:

```
 80 \ CODE TO COPY A 32-BIT INTEGER,
 90 \ intvar, INTO THE IWA.
100 .intvar   EQUD 12345678
110    LDX #0       \ zeroise loop count
120 .loop
130    LDA intvar,X \ next byte of intvar
140    STA &2A,X    \ save in IWA
150    INX          \ bump loop count
160    CPX #4       \ end of loop ?
170    BCC loop     \ no - back
```

iin demonstration (BASIC 2)

```
  0 iin = &B336
 10 DIM mc% 200
 20 FOR pass% = 0 TO 2 STEP 2
 30 P% = mc%
 40 [OPT pass%
 50 .constant    EQUD 1234   \ set up value
 60 .result      EQUD 0      \ result
 70 .start
 80   LDA #constant MOD 256  \ LSB of address of constant
 90   STA &2A                \ set up &2A
100   LDA #constant DIV 256  \ MSB of address of constant
110   STA &2B                \ set up &2B
120   JSR iin                \ call iin
130   LDX #0                 \ zeroise loop counter
140 .loop
150   LDA &2A,X              \ get next byte of IWA
160   STA result,X           \ save in result
170   INX                    \ bump loop counter
180   CPX #4                 \ end of loop ?
190   BCC loop               \ no - back
200   RTS
210 ]
220 NEXT pass%
230 CALL start
240 PRINT !result
250 END

>RUN
      1234
```

subroutine name	:	**iminus**
function	:	IWA = integer variable – IWA
BASIC 1 address	:	&9C9D
BASIC 2 address	:	&9CC2
entry conditions	:	IWA contains a 32 bit integer
	:	&04,&05 point to integer variable
	:	X = #4
exit status	:	IWA set up
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
typical timing	:	52 microseconds

iminus demonstration (BASIC 2)

```
  0 iminus = &9CC2
 10 DIM mc% 200
 20 FOR pass% = 0 TO 2 STEP 2
 30 P% = mc%
 40 [OPT pass%
 50 .constant    EQUD 26     \ set up value
 60 .subtrahend  EQUD 3      \ subtrahend
 70 .result      EQUD 0      \ result
 80 .start
 90   LDX #0                 \ zeroise loop counter
100 .loop1
110   LDA subtrahend,X       \ get next byte of subtrahend
120   STA &2A,X              \ save in IWA
130   INX                    \ bump loop counter
140   CPX #4                 \ end of loop ?
150   BCC loop1              \ no - back
160   LDA #constant MOD 256  \ LSB of address of constant
170   STA &4                 \ set up &4
180   LDA #constant DIV 256  \ MSB of address of constant
190   STA &5                 \ set up &5
200   LDX #4                 \ set up X
210   JSR iminus             \ call iminus
220   LDX #0                 \ zeroise loop counter
230 .loop2
240   LDA &2A,X              \ get next byte of IWA
250   STA result,X           \ save in result
260   INX                    \ bump loop counter
270   CPX #4                 \ end of loop ?
280   BCC loop2              \ no - back
290   RTS
300 ]
310 NEXT pass%
320 CALL start
330 PRINT !result
340 END
```

```
>RUN
        23
```

subroutine name	:	**imod**
function	:	IWA = IWA MOD integer variable
BASIC 1 address	:	&9DDE
BASIC 2 address	:	&9E01
entry conditions	:	IWA contains dividend
	:	A% contains divisor
	:	&19,&1A (lo,hi) point to a string, ‘ A%' + RETURN
	:	&1B = 0
	:	&04,&05 (lo,hi) = HIMEM
	:	A = #&40
comments	:	DIV and MOD are not easy to extricate from the BASIC ROM and the set up procedures are necessarily somewhat contrived
exit status	:	IWA = remainder
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
error reports	:	division by zero
typical timing	:	782 microseconds

imod demonstration (BASIC 2)

```
 10 imod = &9E01 : DIM mc% 200
 20 FOR pass% = 0 TO 2 STEP 2
 30 P% = mc% : [OPT pass%
 40 .dividend EQUD 26
 50 .divisor  EQUD 3
 60 .fudge    EQUD &0D254120      \ " A%" + RETURN
 70 .result   EQUD 0
 80 .start    LDA &6              \ get HIMEM lo
 90           STA &4              \ set &4
100           LDA &7              \ get HIMEM hi
110           STA &5              \ set &5
120           LDA #fudge MOD 256  \ point &19
130           STA &19             \ at fudge lo
140           LDA #fudge DIV 256  \ point &1A
150           STA &1A             \ at fudge hi
160           LDX #0              \ zeroise loop count
170           STX &1B             \ clear &1B
180 .loop1    LDA dividend,X      \ next of dividend
190           STA &2A,X           \ save in IWA
200           LDA divisor,X       \ next of divisor
210           STA &404,X          \ save in A%
220           INX                 \ bump loop counter
230           CPX #4              \ end of loop ?
240           BCC loop1           \ no - back
250           LDA #&40            \ set integer
260           JSR imod            \ call imod
270           LDX #0              \ zeroise loop count
280 .loop2    LDA &2A,X           \ next byte of IWA
290           STA result,X        \ save in result
300           INX                 \ bump loop count
310           CPX #4              \ end of loop ?
320           BCC loop2           \ no - back
330           RTS:]
340 NEXT pass% : CALL start : PRINT !result : END
```

```
>RUN
         2
```

subroutine name	:	**imult**
function	:	IWA = integer variable * IWA
BASIC 1 address	:	&9D4A
BASIC 2 address	:	&9D6D
entry conditions	:	IWA contains a 32 bit integer
	:	&04,&05 point to integer variable
	:	&27 = #4
Comments	:	if the IWA contains an integer larger than &FFFF, it is truncated to 16 significant bits.
exit status	:	IWA set up
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
typical timing	:	164 microseconds

imult demonstration (BASIC 2)

```
 10 imult = &9D6D : DIM mc% 200
 20 FOR pass% = 0 TO 2 STEP 2
 30 P% = mc%
 40 [OPT pass%
 50 .multiplicand EQUD 26        \ set up value
 60 .multiplier   EQUD 3         \ multiplier
 70 .result       EQUD 0         \ result
 80 .start
 90   LDX #0                     \ zeroise loop counter
100 .loop1
110   LDA multiplier,X           \ get next byte of multiplier
120   STA &2A,X                  \ save in IWA
130   INX                        \ bump loop counter
140   CPX #4                     \ end of loop ?
150   BCC loop1                  \ no - back
160   LDA #multiplicand MOD 256  \ LSB of address
170   STA &4                     \ set up &4
180   LDA #multiplicand DIV 256  \ MSB of address
190   STA &5                     \ set up &5
200   LDX #4                     \ set up
210   STX &27                    \ &27
220   JSR imult                  \ call imult
230   LDX #0                     \ zeroise loop counter
240 .loop2
250   LDA &2A,X                  \ get next byte of IWA
260   STA result,X               \ save in result
270   INX                        \ bump loop counter
280   CPX #4                     \ end of loop ?
290   BCC loop2                  \ no - back
300   RTS
310 ]
320 NEXT pass%
330 CALL start
340 PRINT !result
350 END

>RUN
        78
```

subroutine name	:	**ineg1**
function	:	IWA = –1
BASIC 1 address	:	&ACEA
BASIC 2 address	:	&ACC4
entry conditions	:	none
exit status	:	IWA = –1
	:	A destroyed
	:	X unchanged
	:	Y unchanged
	:	P destroyed
typical timing	:	20 microseconds

ineg1 demonstration (BASIC 2)

```
  0 ineg1 = &ACC4
 10 DIM mc% 200
 20 FOR pass% = 0 TO 2 STEP 2
 30 P% = mc%
 40 [OPT pass%
 50 .result EQUD 0          \ result
 60 .start
 70   JSR ineg1             \ call ineg1
 80   LDX #0                \ zeroise loop counter
 90 .loop
100   LDA &2A,X             \ get next byte of IWA
110   STA result,X          \ save in result
120   INX                   \ bump loop counter
130   CPX #4                \ end of loop ?
140   BCC loop              \ no - back
150   RTS
160 ]
170 NEXT pass%
180 CALL start
190 PRINT !result
200 END
```

```
>RUN
        -1
```

subroutine name	:	**iout**
function	:	integer variable = IWA
BASIC 1 address	:	&B4F2
BASIC 2 address	:	&B4C6
entry conditions	:	&37,&38 (lo,hi) points to integer variable
	:	&39 must be non-zero
exit status	:	IWA set up
	:	A destroyed
	:	X unchanged
	:	Y destroyed
	:	P destroyed
typical timing	:	37 microseconds
special note	:	Note that iout is shown for completeness only. The code represented by iout is trivial. The following code can be substituted:

```
 80 \ CODE TO COPY THE IWA INTO A
 90 \ 32-BIT INTEGER VARIABLE, intvar.
100 .intvar EQUD 0
110   LDX #0       \ zeroise loop count
120 .loop
130   LDA &2A,X    \ next byte of IWA
140   STA intvar,X \ save in intvar
150   INX          \ bump loop count
160   CPX #4       \ end of loop ?
170   BCC loop     \ no - back
```

iout demonstration (BASIC 2)

```
  0 iout = &B4C6
 10 DIM mc% 200
 20 FOR pass% = 0 TO 2 STEP 2
 30 P% = mc%
 40 [OPT pass%
 50 .constant    EQUD 1234  \ set up value
 60 .result      EQUD 0     \ result
 70 .start
 80   LDX #0                \ zeroise loop counter
 90 .loop
100   LDA constant,X        \ next byte of constant
110   STA &2A,X             \ set up IWA
120   INX                   \ bump loop counter
130   CPX #4                \ end of loop
140   BCC loop              \ no - back
150   LDA #result MOD 256   \ set up
160   STA &37               \ &37
170   LDA #result DIV 256   \ set up
180   STA &38               \ &38
190   LDA #1                \ set up
200   STA &39               \ &39
210   JSR iout              \ call iout
220   RTS
230 ]
240 NEXT pass%
250 CALL start
260 PRINT !result
270 END

>RUN
      1234
```

subroutine name	:	**iplus**
function	:	IWA = IWA + integer variable
BASIC 1 address	:	&9C36
BASIC 2 address	:	&9C5B
entry conditions	:	IWA contains a 32 bit integer
	:	&04,&05 lo,hi point to integer variable
	:	X = #&4
exit status	:	IWA added
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
typical timing	:	50 microseconds

iplus demonstration (BASIC 2)

```
  0 iplus = &9C5B
 10 DIM mc% 200
 20 FOR pass% = 0 TO 2 STEP 2
 30 P% = mc%
 40 [OPT pass%
 50 .constant    EQUD 26  \ set up value
 60 .adder       EQUD 3   \
 70 .result      EQUD 0   \ result
 80 .start
 90   LDX #0              \ zeroise loop counter
100 .loop1
110   LDA constant,X      \ get next byte of constant
120   STA &2A,X           \ save in IWA
130   INX                 \ bump loop counter
140   CPX #4              \ end of loop ?
150   BCC loop1           \ no - back
160   LDA #adder MOD 256  \ get lo address of adder
170   STA &4              \ save in &4
180   LDA #adder DIV 256  \ get hi address of adder
190   STA &5              \ save in &5
200   LDX #4              \ set X
210   JSR iplus           \ call iplus
220   LDX #0              \ zeroise loop counter
230 .loop2
240   LDA &2A,X           \ get next byte of IWA
250   STA result,X        \ save in result
260   INX                 \ bump loop counter
270   CPX #4              \ end of loop ?
280   BCC loop2           \ no - back
290   RTS
300 ]
310 NEXT pass%
320 CALL start
330 PRINT !result
340 END

>RUN
        29
```

subroutine name	:	**ipos**
function	:	Make IWA positive
BASIC 1 address	:	&AD94
BASIC 2 address	:	&AD71
entry conditions	:	IWA contains a 32 bit integer
comments	:	If the IWA contains a negative integer, it is complemented. Else it is unchanged.
exit status	:	IWA always positive
	:	A destroyed
	:	X unchanged
	:	Y destroyed
	:	P destroyed
typical timing	:	17 or 27 microseconds (17 if already +ve)

ipos demonstration (BASIC 2)

```
  0 ipos = &AD71
 10 DIM mc% 200
 20 FOR pass% = 0 TO 2 STEP 2
 30 P% = mc%
 40 [OPT pass%
 50 .constant    EQUD -1234  \ set up value
 60 .result      EQUD 0      \ result
 70 .start
 80   LDX #0                 \ zeroise loop counter
 90 .loop1
100   LDA constant,X         \ get next byte of constant
110   STA &2A,X              \ save in IWA
120   INX                    \ bump loop counter
130   CPX #4                 \ end of loop ?
140   BCC loop1              \ no - back
150   JSR ipos               \ call ipos
160   LDX #0                 \ zeroise loop counter
170 .loop2
180   LDA &2A,X              \ get next byte of IWA
190   STA result,X           \ save in result
200   INX                    \ bump loop counter
210   CPX #4                 \ end of loop ?
220   BCC loop2              \ no - back
230   RTS
240 ]
250 NEXT pass%
260 CALL start
270 PRINT !result
280 END

>RUN
      1234
```

subroutine name	:	**ismall**
function	:	IWA = 256*Y + A
BASIC 1 address	:	&AF19
BASIC 2 address	:	&AEEA
entry conditions	:	Y and A set up appropriately
comments	:	This is a handy way of initialising the IWA with small numbers up to 16 bits in length
exit status	:	IWA set up
	:	A destroyed
	:	X unchanged
	:	Y destroyed
	:	P destroyed
typical timing	:	20 microseconds

ismall demonstration (BASIC 2)

```
  0 ismall = &AEEA
 10 DIM mc% 200
 20 FOR pass% = 0 TO 2 STEP 2
 30 P% = mc%
 40 [OPT pass%
 50 .result      EQUD 0      \ result
 60 .start
 70   LDY #&13               \ set up constant = #&1388
 80   LDA #&88               \ = 5000
 90   JSR ismall             \ ismall to put 256*Y + A in IWA
100   LDX #0                 \ zeroise loop counter
110 .loop
120   LDA &2A,X              \ get next byte of IWA
130   STA result,X           \ save in result
140   INX                    \ bump loop counter
150   CPX #4                 \ end of loop ?
160   BCC loop               \ no - back
170   RTS
180 ]
190 NEXT pass%
200 CALL start
210 PRINT !result
220 END

>RUN
      5000
```

subroutine name	:	**itest**
function	:	Test integer variable = > or < IWA
BASIC 1 address	:	&9A85
BASIC 2 address	:	&9AAD
entry conditions	:	IWA contains a 32 bit integer
	:	&04,&05 (lo,hi) point to integer variable
	:	&27 must be #4
Comments	:	On exit, the status register is set so that
	:	BEQ will work if variable = IWA
	:	BCC will work if variable < IWA
	:	BCS will work if variable > or = IWA
	:	These tests must be performed immediately, or alternatively PHP can be used to stack the status register for testing later in the program
exit status	:	IWA destroyed
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P see comments above
	:	&4,5 stepped by 4
typical timing	:	58 microseconds

itest demonstration (BASIC 2)

```
 10 itest = &9AAD : DIM mc% 200
 20 FOR pass% = 0 TO 2 STEP 2
 30 P% = mc% : [OPT pass%
 40 .con1        EQUD 1234    \ set up value 1
 50 .con2        EQUD 2345    \ set up value 2
 60 .start  LDX #0            \ zeroise loop counter
 70 .loop1  LDA con2,X        \ get next byte of con2
 80         STA &2A,X         \ save in IWA
 90         INX               \ bump loop counter
100         CPX #4            \ end of loop ?
110         BCC loop1         \ no - back
120         LDA #con1 MOD 256 \ get LSB of con1
130         STA &4            \ save in &4
140         LDA #con1 DIV 256 \ get MSB of con1
150         STA &5            \ save in &5
160         LDX #4            \ set for
170         STX &27           \ integer variable type
180         JSR itest         \ call itest
190         BEQ equals        \ if =
200         BCS more          \ if >
210         LDA #3            \ if <
220         JMP set70         \ set &70
230 .equals LDA #1            \ if =
240         JMP set70         \ set &70
250 .more   LDA #2            \ if >
260 .set70  STA &70           \ &70 set
270         RTS:]
280 NEXT pass% : DIM test$(3)
290 test$(1) = " is equal to "
300 test$(2) = " is greater than "
310 test$(3) = " is less than "
320 CALL start
330 PRINT !con1;test$(?&70);!con2
340 END

>RUN
      1234 is less than 2345
```

subroutine name	:	**izero**
function	:	Make IWA zero
BASIC 1 address	:	&AEF9
BASIC 2 address	:	&AECA
entry conditions	:	none
exit status	:	IWA = 0
	:	A destroyed
	:	X unchanged
	:	Y destroyed
	:	P unchanged
typical timing	:	25 microseconds

izero demonstration (BASIC 2)

```
  0 izero = &AECA
 10 DIM mc% 200
 20 FOR pass% = 0 TO 2 STEP 2
 30 P% = mc%
 40 [OPT pass%
 50 .result      EQUD 0      \ result
 60 .start
 70   JSR izero              \ call izero
 80   LDX #0                 \ zeroise loop counter
 90 .loop
100   LDA &2A,X              \ get next byte of IWA
110   STA result,X           \ save in result
120   INX                    \ bump loop counter
130   CPX #4                 \ end of loop ?
140   BCC loop               \ no - back
150   RTS
160 ]
170 NEXT pass%
180 CALL start
190 PRINT !result
200 END

>RUN
         0
```

subroutine name	:	**izpin**
function	:	Copy integer variable in zero page to IWA
BASIC 1 address	:	&AF85
BASIC 2 address	:	&AF56
entry conditions	:	X points to zero page integer variable
Comments	:	&00,X to &03,X copied into IWA
exit status	:	IWA set up
	:	A destroyed
	:	X unchanged
	:	Y unchanged
	:	P destroyed
typical timing	:	27 microseconds

izpin demonstration (BASIC 2)

```
  0 izpin = &AF56
 10 DIM mc% 200
 20 FOR pass% = 0 TO 2 STEP 2
 30 P% = mc%
 40 [OPT pass%
 50 .result      EQUD 0      \ result
 60 .start
 70   LDX #&70               \ point X to &70
 75                          \ (contains 5000)
 80   JSR izpin              \ call izpin
 90   LDX #0                 \ zeroise loop counter
100 .loop
110   LDA &2A,X              \ get next byte of IWA
120   STA result,X           \ save in result
130   INX                    \ bump loop counter
140   CPX #4                 \ end of loop ?
150   BCC loop               \ no - back
160   RTS
170 ]
180 NEXT pass%
190 !&70 = 5000
200 CALL start
210 PRINT !result
220 END
```

```
>RUN
      5000
```

subroutine name	: **izpout**
function	: Copy IWA to integer variable in zero page
BASIC 1 address	: &BE5C
BASIC 2 address	: &BE44
entry conditions	: X points to zero page integer variable
comments	: IWA copied to &00,X to &03,X
exit status	: IWA unchanged
	: A destroyed
	: X unchanged
	: Y unchanged
	: P destroyed
	: &00,X to &03,X set up
typical timing	: 26 microseconds

izpout demonstration (BASIC 2)

```
  0 izpout = &BE44
 10 DIM mc% 200
 20 FOR pass% = 0 TO 2 STEP 2
 30 P% = mc%
 40 [OPT pass%
 50 .constant    EQUD 5000   \ set up value
 60 .start
 70   LDX #0                 \ zeroise loop counter
 80 .loop
 90   LDA constant,X         \ get next byte of constant
100   STA &2A,X              \ save in IWA
110   INX                    \ bump loop counter
120   CPX #4                 \ end of loop ?
130   BCC loop               \ no - back
140   LDX #&70               \ point izpout to &70
150   JSR izpout             \ call izpout
160   RTS
170 ]
180 NEXT pass%
190 CALL start
210 PRINT !&70
220 END

>RUN
      5000
```

3 FLOATING POINT NUMBERS

Although floating point format can be used to store integers, especially large integers outside of the range of 32-bit integer format, it is essentially designed to handle fractions.

3.1 Floating Point Variables

The BBC BASIC interpreter recognises a floating point variable by the absence of either a '%' or a '$' at the end of its name. Each floating point variable occupies five bytes (40 bits). The number itself is held in the last four bytes (called the mantissa). The first byte (called the exponent) defines the position of the binary point. In other words, it defines the end of the integral part of the number and the start of the fractional part.

Consider the decimal number 7.125. The binary equivalent of this number is 0111.0010 = (0*8) + (1*4) + (1*2) + (1*1) + (0*1/2) + (0*1/4) + (1*1/8) + (0*1/16). To obtain its floating point representation, the mantissa is simply written down as a string of 32 bits, aligned at the most significant end, with the binary point omitted.

```
mantissa
--------
0111 0010  0000 0000  0000 0000  0000 0000
      &72        &00        &00        &00
```

Had the binary point been included, it would have been positioned after the fourth bit in the mantissa. Thus the exponent is 4 in this case.

```
 exponent  mantissa
 --------  --------
0000 0100  0111 0010  0000 0000  0000 0000  0000 0000
       &4        &72        &00        &00        &00
```

It will be seen that the exponent and mantissa above are not the only ones that represent the number 7.125. Consider the following:

```
 exponent  mantissa
 --------  --------
0000 0101  0011 1001  0000 0000  0000 0000  0000 0000
       &5        &39        &00        &00        &00

0000 0110  0001 1100  1000 0000  0000 0000  0000 0000
       &6        &1C        &80        &00        &00
```

When a zero is shifted into the most significant bit of the mantissa, the mantissa is effectively divided by 2. To compensate for this, the exponent is incremented by one, effectively multiplying the number by 2. The decimal analogy is that 7.125 could equally well be expressed as 71.25 tenths or 712.5 hundredths. Clearly it would be difficult to work with floating point numbers if each number could have so many different representations.

Fortunately there is a rule which standardises the format of floating point numbers. The rule is called 'Normalisation'. In a normalised floating point number, the most significant bit of the mantissa is always a 1. This is achieved by shifting the mantissa left, bit-by-bit, until all leading zeroes have disappeared. Since each leftward shift multiplies the mantissa by 2, the exponent must be reduced by one each time. Thus in its normalised form, the floating point representation of 7.125 is:

```
 exponent  mantissa
 --------  --------
0000 0011  1110 0100  0000 0000  0000 0000  0000 0000
       &3       &E4         &00        &00        &00
```

BBC BASIC expects all floating point numbers to be normalised. It makes use of this fact to handle the sign of a floating point number. If the sign of the number is positive, it changes the most significant bit of the mantissa to 0. This is simply a ruse to avoid holding the sign separately and hence to minimise the amount of memory needed to store a floating point number. In fact the number above represents −7.125 and +7.125 is:

```
 exponent  mantissa
 --------  --------
0000 0011  0110 0100  0000 0000  0000 0000  0000 0000
       &3       &64         &00        &00        &00
```

BBC BASIC adds &80 to the exponent. This is purely a device to assist in processing floating point numbers. Thus in BBC BASIC, a floating point variable set to +7.125 actually contains:

```
 exponent  mantissa
 --------  --------
 10000011  0110 0100  0000 0000  0000 0000  0000 0000
      &83       &64         &00        &00        &00
```

It will be seen that, to convert a positive number to negative, it is only necessary to add &80 to the most significant byte of the mantissa.

Thus there are four stages in the process of converting a number to floating point format:

a) Write down the mantissa in binary and set the exponent to the value which fixes the position of the implied binary point.

b) Normalise the number, ensuring that the exponent is adjusted appropriately for each bit shifted.

c) If the sign of the original number was positive, change the most significant bit of the mantissa to zero.

d) Add &80 to the exponent.

It remains now to demonstrate that this process works for purely fractional numbers. Consider the decimal number −0.375. As this represents −3/8 which is the same as −(1/4 + 1/8), the binary equivalent of this number is −.0110. It is easy to write down the mantissa in its unnormalised binary form:

```
mantissa
--------
0110 0000  0000 0000  0000 0000  0000 0000
      &60        &00        &00        &00
```

The exponent of this unnormalised number is zero, since the implied binary point comes immediately in front of the most significant bit of the mantissa. To normalise the mantissa, it must be shifted to the left until all leading zeroes have been removed, reducing the exponent by one each time. Consequently, the exponent is −1, to which must be added &80 as before. In hexadecimal terms:

```
-0.375 = &7F &C0 &00 &00 &00
```

One last contrivance is employed in BBC BASIC. The floating point representation of 0.0 does not follow the rules. It is simply stored as five bytes of zeroes.

Some floating point numbers are tabulated below:

```
  0 = &00 &00 &00 &00 &00
 +1 = &81 &00 &00 &00 &00    -1 = &81 &80 &00 &00 &00
 +2 = &82 &00 &00 &00 &00    -2 = &82 &80 &00 &00 &00
 +3 = &82 &40 &00 &00 &00    -3 = &82 &C0 &00 &00 &00
 +4 = &83 &00 &00 &00 &00    -4 = &83 &80 &00 &00 &00
 +5 = &83 &20 &00 &00 &00    -5 = &83 &A0 &00 &00 &00
 +6 = &83 &40 &00 &00 &00    -6 = &83 &C0 &00 &00 &00
 +7 = &83 &60 &00 &00 &00    -7 = &83 &E0 &00 &00 &00
 +8 = &84 &00 &00 &00 &00    -8 = &84 &80 &00 &00 &00
 +9 = &84 &10 &00 &00 &00    -9 = &84 &90 &00 &00 &00
+10 = &84 &20 &00 &00 &00   -10 = &84 &A0 &00 &00 &00
```

3.2 Integer versus Floating Point

Integer numbers have two big advantages over their floating point counterparts: accuracy and speed.

In all the floating point examples so far presented in this book, the fractional part of the number has always translated into an exact number of halves, quarters, eighths, sixteenths and so on. Numbers like this are referred to as 'machine numbers' because they can be held exactly within a computer. By no means can this be said of all numbers. For example, the fraction 1/5 cannot be held exactly. However many fractional bits are included, the resulting floating point number remains an approximation, albeit a good one, to the original fraction. Numbers that have a large integral part have less bits available for the fractional part and tend to suffer more from fractional inaccuracy. An example of floating point inaccuracy is shown in the following BASIC code:

```
10 J=0
20 FOR I% = 1 TO 40
30 J = J + 0.2
40 NEXT
50 PRINT J
60 END
```

```
>RUN
7.99999999
```

Integer arithmetic is not only accurate, it is faster than floating point. This simply means that it is less complicated and requires less code in the BASIC interpreter.

However, integer numbers have two disadvantages compared to floating point numbers. The first is obvious; they cannot be used to represent fractions. The second is that they cannot handle such a wide range of numbers as floating point (from 1.7*10 to the power –39 up to 1.7*10 to the power 38).

Overall the advantages of using integers are so great, that providing the numbers to be used fall within the range of integer numbers, they should be used if at all possible. This is especially true for financial applications, where loss of accuracy in floating point can render a program useless. Except for very large sums of money, financial data should be held in pence. There is then no fraction to consider (assuming the demise of the halfpenny). The decimal point can always be inserted into an ASCII field when printing reports.

This is an example of a technique known as scaling. Because financial data has only two digits following the decimal point, all numbers are scaled up by a factor of 100 to remove the fractional part altogether. The technique can be applied to other situations.

Floating point comes into its own in mathematical programs where total accuracy is neither expected nor required, such as when drawing curves.

3.3 Floating Point Work Areas

BBC BASIC does all its floating point arithmetic in two working areas in zero page. As with the IWA, these memory areas are not dedicated to floating point and may be re-used for quite different purposes.

Each of the working areas is eight bytes long (not five as might have been expected). The extra 3 bytes are for the following purposes:

a) a sign byte. In the five byte variable format, the sign is contrived by zeroising the most significant bit of the mantissa for positive numbers. In eight byte format, the most significant byte of the mantissa is copied into a sign byte, and the most significant bit of the mantissa is restored to 1.

b) a rounding byte. An extra byte is tacked onto the end to extend the precision of arithmetic. This extra byte is used to round the preceding mantissa when required.

c) an overflow byte. This byte exists to trap errors, such as those which might occur when the result of a multiplication is a number too big to handle.

The five byte format unpacks into the eight byte format as follows:

```
           5 byte        8 byte
           ------        ------
exponent   byte 0 ------> byte 2 exponent
mantissa-1 byte 1 ------> byte 0 sign
mantissa-1 byte 1 OR #&80 byte 3 mantissa-1
mantissa-2 byte 2 ------> byte 4 mantissa-2
mantissa-3 byte 3 ------> byte 5 mantissa-3
mantissa-4 byte 4 ------> byte 6 mantissa-4
                   zero   byte 1 overflow
                   zero   byte 7 rounding
```

For example, +1.0 in the two formats is:

```
           5 byte         8 byte
           ------         ------
exponent    &81            &00 sign
mantissa-1  &00            &00 overflow
mantissa-2  &00            &81 exponent
mantissa-3  &00            &80 mantissa-1
mantissa-4  &00            &00 mantissa-2
                           &00 mantissa-3
                           &00 mantissa-4
                           &00 rounding
```

The two floating point areas used by BASIC are &2E to &35, and &3B to &42 inclusive. These will be referred to as Floating Point Work Area A or FWA, and Floating Point Work Area B or FWB, respectively. They consist of:

```
            FWA  FWB
            ---  ---
sign        &2E  &3B
overflow    &2F  &3C
exponent    &30  &3D
mantissa-1  &31  &3E
mantissa-2  &32  &3F
mantissa-3  &33  &40
mantissa-4  &34  &41
rounding    &35  &42
```

3.4 Defining Floating Point Constants

Neither BASIC 1 nor BASIC 2 has a directive to allow definition of a floating point constant to the assembler. Once again it is necessary to invent a pseudo-directive. This one is called EQUF. It relies on two facts. Firstly it uses a BASIC variable, Z, and the address of the look-up table for variables starting with the letter 'Z' is held in &4B4,&4B5 (lo,hi). Secondly, the pseudo-directive assumes that the actual data in Z will be found 3 bytes into this look-up table. This will be true so long as Z is the first variable which has an entry in this look-up table. To ensure this, make certain that no other BASIC variables start with the letter 'Z'. This caution should be unnecessary, because the techniques advocated discourage hybrid assembly language/BASIC programs.

```
 10 DIM mc% 100 : FOR pass% = 0 TO 2 STEP 2
 20 P% = mc% : [OPT pass%
 30 .constant
 40    OPT FNEQUF(1.0)
 50 ]
 60 NEXT pass%
 70 END
 80 DEF FNEQUF(Z)
 90 I% = 3 + ?&4B4 + 256*?&4B5
100 FOR J% = 1 TO 5
110     ?P% = ?I%
120     P%  = P% + 1
130     I%  = I% + 1
140 NEXT
150 = pass%
```

A particularly useful feature of these pseudo-directives, is that they can be used to evaluate expressions, providing the terms are also literals. For example:

```
30 .constant
40    OPT FNEQUF(3*SIN(PI/4))
```

3.5 Floating Point Routines Summary

The following tables summarise floating point routines available within the BASIC ROM. Conversion routines, e.g. floating point to ASCII, are the subject of a later chapter. So too are routines handling trigonometric functions, square roots, logarithms etc.

Name	BASIC 1 address	BASIC 2 address	Function
aclear	&A691	&A686	FWA = 0
acomp	&AD99	&AD7E	FWA = -FWA
acopyb	&A4E4	&A4DC	FWA = FWB
adiv	&A68B	&A6AD	FWA = fp var / FWA normalised and rounded
adiv10	&A23E	&A24D	FWA = FWA / 10 unnormalised and unrounded
aminus	&A50B	&A4FD	FWA = fp var - FWA normalised and rounded
amult	&A661	&A656	FWA = FWA * fp var normalised and rounded
amult1	&A611	&A606	FWA = FWA * fp var unnormalised and unrounded
amult10	&A1E5	&A1F4	FWA = FWA * 10 unnormalised and unrounded
anorm	&A2F4	&A303	normalise FWA
aone	&A6A4	&A699	FWA = 1
apack	&A37E	&A38D	pack FWA into fp var
apack1	&A376	&A385	pack FWA into &46C to &470
apack2	&A36E	&A37D	pack FWA into &471 to &475
apack3	&A372	&A381	pack FWA into &476 to &47A
aplus	&A50E	&A500	FWA = FWA + fp var normalised and rounded
aplusb	&A513	&A505	FWA = FWA + FWB normalised and rounded
aplus1	-----	&A50B	FWA = FWA + FWB normalised and unrounded
arecip	&A6B0	&A6A5	FWA = 1 / FWA normalised and rounded
around	&A667	&A65C	round FWA
asign	&A1CB	&A1DA	get sign of FWA
aswap	&A4DE	&A4D6	swap fp var and FWA
atest	&9A37	&9A5F	test fp var against FWA
aunp	&A3A6	&A3B5	unpack fp var into FWA
aunp1	&A3A3	&A3B2	unpack &46C to &470 into FWA
bclear	&A463	&A453	FWB = 0
bcopya	&A20F	&A21E	FWB = FWA
bunp	&A33F	&A34E	unpack fp var into FWB

3.6 Floating Point Routines Description

The following pages show how to use each of the floating point routines in the table. Set up procedures are very simple. Routines involving only FWA and/or FWB need no setting up, other than to load FWA/FWB prior to the call. Routines which reference a floating point variable should point &4B,&4C (lo,hi) to that variable.

For this reason, demonstration programs are not supplied for each routine. In place of these, there is a single program which can be incorporated into a user program, which:

a) provides a standard interface for all floating point arithmetic.

b) can be assembled under BASIC 1 or BASIC 2.

c) the machine code derived will run under either BASIC 1 or BASIC 2.

subroutine name	:	**aclear**
function	:	FWA = zero
BASIC 1 address	:	&A691
BASIC 2 address	:	&A686
entry conditions	:	none
exit status	:	FWA = zero
	:	FWB unchanged
	:	A destroyed
	:	X unchanged
	:	Y unchanged
	:	P destroyed
typical timing	:	25 microseconds

subroutine name	:	**acomp**
function	:	FWA = –FWA
BASIC 1 address	:	&AD99
BASIC 2 address	:	&AD7E
entry conditions	:	none
exit status	:	FWA = –FWA
	:	FWB unchanged
	:	A destroyed
	:	X unchanged
	:	Y unchanged
	:	P destroyed
typical timing	:	34 microseconds

subroutine name	:	**acopyb**
function	:	FWA = FWB
BASIC 1 address	:	&A4E4
BASIC 2 address	:	&A4DC
entry conditions	:	none
exit status	:	FWA = FWB
	:	FWB unchanged
	:	A destroyed
	:	X unchanged
	:	Y unchanged
	:	P destroyed
typical timing	:	36 microseconds

subroutine name	: **adiv**
function	: FWA = fp var / FWA normalised, rounded
BASIC 1 address	: &A68B
BASIC 2 address	: &A6AD
entry conditions	: &4B,&4C (lo,hi) point to fp var
exit status	: FWA = quotient
	: FWB destroyed
	: A destroyed
	: X destroyed
	: Y destroyed
	: P destroyed
error reports	: division by zero
typical timing	: 1545 microseconds

subroutine name	:	**adiv10**
function	:	FWA = FWA / 10 normalised, rounded
BASIC 1 address	:	&A23E
BASIC 2 address	:	&A24D
entry conditions	:	none
exit status	:	FWA = FWA / 10
	:	FWB destroyed
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
typical timing	:	360 microseconds

subroutine name	: **aminus**
function	: FWA = fp var – FWA normalised and rounded
BASIC 1 address	: &A50B
BASIC 2 address	: &A4FD
entry conditions	: &4B,&4C (lo,hi) point to fp var
exit status	: FWA = result
	: FWB destroyed
	: A destroyed
	: X destroyed
	: Y destroyed
	: P destroyed
typical timing	: 254 microseconds

subroutine name	:	**amult**
function	:	FWA = fp var * FWA normalised and rounded
BASIC 1 address	:	&A661
BASIC 2 address	:	&A656
entry conditions	:	&4B,&4C (lo,hi) point to fp var
exit status	:	FWA = result
	:	FWB destroyed
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
error reports	:	too big
typical timing	:	1581 microseconds

subroutine name	:	**amult1**
function	:	FWA = fp var * FWA unnormalised and unrounded
BASIC 1 address	:	&A611
BASIC 2 address	:	&A606
entry conditions	:	&4B,&4C (lo,hi) point to fp var
exit status	:	FWA = result
	:	FWB destroyed
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
error reports	:	too big
typical timing	:	1508 microseconds

subroutine name	:	**amult10**
function	:	FWA = 10 * FWA unnormalised and unrounded
BASIC 1 address	:	&A1E5
BASIC 2 address	:	&A1F4
entry conditions	:	none
exit status	:	FWA = result
	:	FWB destroyed
	:	A destroyed
	:	X unchanged
	:	Y unchanged
	:	P destroyed
typical timing	:	171 microseconds

subroutine name	:	**anorm**
function	:	FWA = FWA normalised
BASIC 1 address	:	&A2F4
BASIC 2 address	:	&A303
entry conditions	:	none
exit status	:	FWA = result
	:	FWB unchanged
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
typical timing	:	27 microseconds

subroutine name	:	**aone**
function	:	FWA = 1
BASIC 1 address	:	&A6A4
BASIC 2 address	:	&A699
entry conditions	:	none
exit status	:	FWA = 1
	:	FWB unchanged
	:	A destroyed
	:	X unchanged
	:	Y destroyed
	:	P destroyed
typical timing	:	37 microseconds

subroutine name	:	**apack**
function	:	fp var = FWA
BASIC 1 address	:	&A37E
BASIC 2 address	:	&A38D
entry conditions	:	&4B,&4C (lo,hi) point to fp var
exit status	:	FWA unchanged
	:	FWB unchanged
	:	A destroyed
	:	X unchanged
	:	Y destroyed
	:	P destroyed
typical timing	:	46 microseconds

subroutine name	:	**apack1**
function	:	&46C to &470 = FWA
BASIC 1 address	:	&A376
BASIC 2 address	:	&A385
entry conditions	:	none
exit status	:	FWA unchanged
	:	FWB unchanged
	:	A destroyed
	:	X unchanged
	:	Y destroyed
	:	P destroyed
	:	&4B,&4C destroyed
typical timing	:	51 microseconds

subroutine name	:	**apack2**
function	:	&471 to &475 = FWA
BASIC 1 address	:	&A36E
BASIC 2 address	:	&A37D
entry conditions	:	none
exit status	:	FWA unchanged
	:	FWB unchanged
	:	A destroyed
	:	X unchanged
	:	Y destroyed
	:	P destroyed
	:	&4B,&4C destroyed
typical timing	:	53 microseconds

subroutine name	:	**apack3**
function	:	&476 to &47A = FWA
BASIC 1 address	:	&A372
BASIC 2 address	:	&A381
entry conditions	:	none
exit status	:	FWA unchanged
	:	FWB unchanged
	:	A destroyed
	:	X unchanged
	:	Y destroyed
	:	P destroyed
	:	&4B,&4C destroyed
typical timing	:	53 microseconds

subroutine name	:	**aplus**
function	:	FWA = fp var + FWA normalised and rounded
BASIC 1 address	:	&A50E
BASIC 2 address	:	&A500
entry conditions	:	&4B,&4C (lo,hi) point to fp var
exit status	:	FWA = result
	:	FWB destroyed
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
error reports	:	too big
typical timing	:	246 microseconds

subroutine name	:	**aplusb**
function	:	FWA = FWA + FWB normalised and rounded
BASIC 1 address	:	&A513
BASIC 2 address	:	&A505
entry conditions	:	none
exit status	:	FWA = result
	:	FWB destroyed
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
error reports	:	too big
typical timing	:	58 microseconds

subroutine name	:	**aplus1**
function	:	FWA = FWA + FWB normalised and unrounded
BASIC 1 address	:	---------
BASIC 2 address	:	&A50B
entry conditions	:	none
exit status	:	FWA = result
	:	FWB destroyed
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
error reports	:	too big
typical timing	:	41 microseconds

subroutine name	:	**arecip**
function	:	FWA = 1 / FWA normalised and rounded
BASIC 1 address	:	&A6B0
BASIC 2 address	:	&A6A5
entry conditions	:	none
exit status	:	FWA = result
	:	FWB destroyed
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
	:	&476 to &47A destroyed
typical timing	:	1619 microseconds

subroutine name	:	**around**
function	:	FWA = FWA rounded
BASIC 1 address	:	&A667
BASIC 2 address	:	&A65C
entry conditions	:	none
exit status	:	FWA = result
	:	FWB unchanged
	:	A destroyed
	:	X unchanged
	:	Y unchanged
	:	P destroyed
typical timing	:	22 microseconds

subroutine name	:	**asign**
function	:	A register denotes sign of FWA
BASIC 1 address	:	&A1CB
BASIC 2 address	:	&A1DA
entry conditions	:	none
comments	:	A register set as follows:
	:	= 0 if FWA is zero
	:	= 1 if FWA is +ve
	:	= –ve if FWA is –ve otherwise
exit status	:	FWA unchanged
	:	FWB unchanged
	:	A see above
	:	X unchanged
	:	Y unchanged
	:	P destroyed
typical timing	:	24 microseconds

subroutine name	:	**aswap**
function	:	FWA = fp var and fp var = FWA
BASIC 1 address	:	&A4DE
BASIC 2 address	:	&A4D6
entry conditions	:	&4B,&4C (lo,hi) point to fp var
exit status	:	FWA = fp var
	:	FWB destroyed
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
typical timing	:	121 microseconds

subroutine name	: **atest**
function	: test fp var against FWA
BASIC 1 address	: &9A37
BASIC 2 address	: &9A5F
entry conditions	: &4B,&4C (lo,hi) point to fp var
comments	: on exit, the P register is set up such that:
	: BEQ works if fp var = FWA
	: BCC works if fp var < FWA
	: BCS works if fp var > or = FWA
exit status	: FWA destroyed
	: FWB destroyed
	: A destroyed
	: X destroyed
	: Y destroyed
	: P see above
typical timing	: 78 microseconds

subroutine name	:	**aunp**
function	:	FWA = fp var
BASIC 1 address	:	&A3A6
BASIC 2 address	:	&A3B5
entry conditions	:	&4B,&4C (lo,hi) point to fp var
exit status	:	FWA = fp var
	:	FWB unchanged
	:	A destroyed
	:	X unchanged
	:	Y destroyed
	:	P destroyed
typical timing	:	49 microseconds

subroutine name	:	**aunp1**
function	:	FWA = &46C to &470
BASIC 1 address	:	&A3A3
BASIC 2 address	:	&A3B2
entry conditions	:	none
exit status	:	FWA = result
	:	FWB unchanged
	:	A destroyed
	:	X unchanged
	:	Y destroyed
	:	P destroyed
typical timing	:	62 microseconds

subroutine name	:	**bclear**
function	:	FWB = 0
BASIC 1 address	:	&A463
BASIC 2 address	:	&A453
entry conditions	:	none
exit status	:	FWA unchanged
	:	FWB = 0
	:	A destroyed
	:	X unchanged
	:	Y unchanged
	:	P destroyed
typical timing	:	25 microseconds

subroutine name	:	**bcopya**
function	:	FWB = FWA
BASIC 1 address	:	&A20F
BASIC 2 address	:	&A21E
entry conditions	:	none
exit status	:	FWA unchanged
	:	FWB = FWA
	:	A destroyed
	:	X unchanged
	:	Y unchanged
	:	P destroyed
typical timing	:	36 microseconds

subroutine name	:	**bunp**
function	:	FWB = fp var
BASIC 1 address	:	&A33F
BASIC 2 address	:	&A34E
entry conditions	:	&4B,&4C (lo,hi) point to fp var
exit status	:	FWA unchanged
	:	FWB = result
	:	A destroyed
	:	X destroyed
	:	Y unchanged
	:	P destroyed
typical timing	:	51 microseconds

3.7 Floating Point Interface Program

This program will perform addition, subtraction, multiplication or division in floating point. It is intended that this code is incorporated into a user program. It may be assembled under either BASIC 1 or BASIC 2. When executed it will detect which BASIC is present and operate accordingly.

On entry to floatsub, the following fields must be set up:

```
&70,&71 (lo,hi) point to argument 1
&72,&73 (lo,hi) point to argument 2
&74,&75 (lo,hi) point to argument 3
&76 = function required
```

```
if &76 = 0 argument 3 = argument 1 + argument 2
if &76 = 1 argument 3 = argument 1 - argument 2
if &76 = 2 argument 3 = argument 1 * argument 2
if &76 = 3 argument 3 = argument 1 / argument 2
```

All mathematical functions return results into argument 3. All results are fully normalised and rounded. Signs are handled correctly throughout. All registers are returned uncorrupted.

Standard Floating Point Interface

```
 10 DIM mc% 500
 20 FOR pass% = 0 TO 2 STEP 2
 30 P% = mc%
 40 [OPT pass%
 50 .basic2              \ BASIC 2
 60 .aplus  JMP &A500
 70 .aminus JMP &A4FD
 80 .amult  JMP &A656
 90 .adiv   JMP &A6AD
100 .apack  JMP &A38D
110 .aunp   JMP &A3B5
120 .basic1              \ BASIC 1
130         JMP &A50E
140         JMP &A50B
150         JMP &A661
160         JMP &A68B
170         JMP &A37E
180         JMP &A3A6
190 .floatsub
200    PHP               \ save P reg
210    PHA               \ save A reg
220    TXA               \ save X reg
230    PHA
240    TYA               \ save Y reg
250    PHA
260    LDA &8015         \ get BASIC year
270    CMP #&32          \ is it 1982 ?
280    BEQ skipmove      \ yes - skipmove
```

```
290    LDX #0              \ zeroise loop counter
300 .move
310    LDA basic1,X        \ next byte of BASIC 1 addresses
320    STA basic2,X        \ overwrite BASIC 2 counterpart
330    INX                 \ bump loop counter
340    CPX #18             \ end of loop ?
350    BCC move            \ no - move
360 .skipmove
370    LDA &72             \ get LSB of argument 2
380    STA &4B             \ save in &4B
390    LDA &73             \ get MSB of argument 2
400    STA &4C             \ save in &4C
410    JSR aunp            \ unpack argument 2 into FWA
420    LDA &70             \ get LSB of argument 1
430    STA &4B             \ save in &4B
440    LDA &71             \ get MSB of argument 1
450    STA &4C             \ save in &4C
460    LDA &76             \ get function
470    BEQ add             \ if add
480    CMP #1              \ test subtract
490    BEQ subtract        \ if subtract
500    CMP #2              \ test multiply
510    BEQ multiply        \ if multiply
520    CMP #3              \ test divide
530    BEQ divide          \ if divide
540    BRK                 \ invalid
550 .add
560    JSR aplus           \ do add
570    JMP result          \ output result
580 .subtract
590    JSR aminus          \ do subtract
600    JMP result          \ output result
610 .multiply
620    JSR amult           \ do multiply
630    JMP result          \ output result
640 .divide
650    JSR adiv            \ do divide
660 .result
670    LDA &74             \ get LSB of result
680    STA &4B             \ save in &4B
690    LDA &75             \ get MSB of result
700    STA &4C             \ save in &4C
710    JSR apack           \ output result
720 .restore
730    PLA                 \ restore
740    TAY                 \ Y reg.
750    PLA                 \ restore
760    TAX                 \ X reg.
770    PLA                 \ restore A reg
780    PLP                 \ restore P reg
790    RTS:]
800 NEXT pass%:STOP
```

3.8 Floating Point Interface Program Tested

The previous program can be tested by changing the BASIC statements. In the changes below, data values are entered into A and B (lines 900 and 910) and the program performs all the calculations.

```
 800 REM  test all functions
 810 NEXT pass%
 820 DIM func$(4)
 830 func$(1) = " + "
 840 func$(2) = " - "
 850 func$(3) = " * "
 860 func$(4) = " / "
 870 ?&76 = -1
 880 REPEAT
 890 ?&76 = ?&76 + 1
 900 A = -20.3
 910 B = -7.258
 920 C = 0
 930 I% = 3 + ?&482 + 256*?&483
 940 ?&70 = I% MOD 256
 950 ?&71 = I% DIV 256
 960 I% = 3 + ?&484 + 256*?&485
 970 ?&72 = I% MOD 256
 980 ?&73 = I% DIV 256
 990 I% = 3 + ?&486 + 256*?&487
1000 ?&74 = I% MOD 256
1010 ?&75 = I% DIV 256
1020 CALL floatsub
1030 PRINT A;func$(1+?&76);B;" = ";C
1040 UNTIL ?&76 = 3
1050 END
```

```
>RUN
     -20.3 + -7.258 = -27.558
     -20.3 - -7.258 = -13.042
     -20.3 * -7.258 = 147.3374
     -20.3 / -7.258 = 2.79691375
```

4 CONVERSIONS

Inevitably, assembly language programs have to deal with the problem of data conversion, particularly from binary to ASCII and vice-versa. The BASIC ROM contains a set of accessible conversion routines which are among the handiest of all the routines in the ROM.

4.1 Conversion Work Areas

The BASIC ROM does all its work with ASCII strings in an area of memory starting at &600. This will be called the String Work Area or SWA. There is another memory area, &36, which contains the length of the current string in the SWA. Another way of looking at this field is that the contents of &36, when added to #&600, point to the next available space in the SWA.

The usual caution is necessary at this point. The area starting at &600 is not reserved for a dedicated purpose. It is also used as a variable parameter block. As for &36, its uses are legion.

Another zero page location which plays an active role in some of these routines is &15. &15 controls the radix when converting a number to an ASCII string. If set to zero, the ASCII string represents a decimal number. If set to −1, it is hexadecimal. BASIC itself sets this field for the PRINT command. The latter setting is used if a tilde (~) appears in the PRINT statement.

During conversions to ASCII, parts of the print format field, @%, are important. Located from &400 to &403, this field controls print formatting as follows:

&403 not applicable (STR$ flag)

&402 format number

format 0 = general
format 1 = exponential
format 2 = fixed decimal

&401 number of digits (exact meaning depends on format)

format 0 = maximum number of digits which can be printed before exponential format is used instead
format 1 = number of digits + 1 that follows the ‘E’
format 2 = number of decimal places

&400 width of print field

&400 does not affect string conversion, but rather is used by the BASIC PRINT command to work further on the converted string. &401 and &402 do affect string conversion. Format 0 is the default, representing typical BASIC formatting. Format 1 specifies that numbers are to be converted to exponential format e.g. 1000 would be converted to 1E3. Format 2 specifies that the number is to be converted to ASCII with a fixed number (to be found in &401) of decimal places. The maximum number of decimal places that can be specified is ten. If a number greater than 10 is placed in &401, the conversion routines default to ten. Similarly, if a format greater than 2 is specified, a default of zero is used. After BREAK, or at power up, BASIC initialises @% to format zero and a width of 10.

4.2 Conversion Routines Summary

Many of the routines below share common entry points. As the set up parameters are quite different in each case, it is easier to present them separately. Integer to floating point conversions (and vice-versa) are so straightforward that no further explanation is required. A demonstration program is supplied in the next section to show the use of ASCII conversion routines.

Name	BASIC 1 address	BASIC 2 address	Function
ascnum	&AC5A	&AC34	ASCII to integer or floating point
fpascdec	&9ED0	&9EDF	floating point to decimal ASCII
fpaschex	&9ED0	&9EDF	floating point to hex ASCII
fpi1	&A3F2	&A3E4	floating point to integer 1
fpi2	&A40C	&A3FE	floating point to integer 2
iascdec	&9ED0	&9EDF	integer to decimal ASCII
iaschex	&9ED0	&9EDF	integer to hex ASCII
ifpa	&A2AF	&A2BE	integer to floating point

4.3 Conversion Routines Description

subroutine name	:	**ascnum**
function	:	the ASCII string in SWA is converted to either an integer placed in IWA or to a floating point number placed in FWA
BASIC 1 address	:	&AC5A
BASIC 2 address	:	&AC34
entry conditions	:	&36 contains length of string in SWA
	:	SWA contains ASCII number
comments	:	the routine places a binary zero at the end of SWA and steps &36 by 1
	:	on exit A and &27 both reflect the type of conversion:
	:	= #&40 for integer
	:	= #&FF for floating point
exit status	:	IWA = result (integer)
	:	FWA = result (floating point)
	:	FWB destroyed
	:	SWA has zero appended
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
	:	&19,&1A,&1B destroyed
	:	&45,&48,&49 destroyed
	:	&27 see above
typical timing	:	1748 microseconds

subroutine name	:	**fpascdec**
function	:	the floating point number in FWA is converted to ASCII decimal and placed in SWA
BASIC 1 address	:	&9ED0
BASIC 2 address	:	&9EDF
entry conditions	:	&15 must be zero
	:	Y must be #&FF
	:	@% must be set as appropriate
comments	:	the routine returns with &36 set to the length of the string
exit status	:	IWA destroyed
	:	FWA destroyed
	:	FWB destroyed
	:	SWA = result
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
	:	&36 = length of string
typical timing	:	4878 microseconds

subroutine name	:	**fpaschex**
function	:	the floating point number in FWA is converted to ASCII hexadecimal and placed in SWA
BASIC 1 address	:	&9ED0
BASIC 2 address	:	&9EDF
entry conditions	:	&15 must be #&FF
	:	Y must be #&FF
	:	@% must be set as appropriate
comments	:	only the integer part of the number is converted
	:	the routine returns with &36 set to the length of the string
exit status	:	IWA destroyed
	:	FWA destroyed
	:	FWB destroyed
	:	SWA = result
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
	:	&36 = length of string
typical timing	:	582 microseconds

subroutine name	:	**fpi1**
function	:	the floating point number in FWA is converted to integer and placed in IWA
BASIC 1 address	:	&A3F2
BASIC 2 address	:	&A3E4
entry conditions	:	none
comments	:	only the integer part of the number is converted
exit status	:	IWA = result
	:	FWA destroyed
	:	FWB destroyed
	:	SWA unchanged
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
typical timing	:	405 microseconds

subroutine name	:	**fpi2**
function	:	the floating point number in FWA is converted to integer and placed in &31 to &34
BASIC 1 address	:	&A40C
BASIC 2 address	:	&A3FE
entry conditions	:	none
comments	:	only the integer part of the number is converted
	:	this routine can be used instead of fpi1 when it is required to preserve IWA
exit status	:	IWA unchanged
	:	FWA destroyed
	:	FWB destroyed
	:	SWA unchanged
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
	:	&31 to &34 = result
typical timing	:	387 microseconds

subroutine name	:	**iascdec**
function	:	the integer number in IWA is converted to ASCII decimal and placed in SWA
BASIC 1 address	:	&9ED0
BASIC 2 address	:	&9EDF
entry conditions	:	&15 must be zero
	:	Y must be #&40
	:	@% must be set as appropriate
comments	:	the routine returns with &36 set to the length of the string
exit status	:	IWA destroyed
	:	FWA destroyed
	:	FWB destroyed
	:	SWA = result
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
	:	&36 = length of string
typical timing	:	5369 microseconds

subroutine name	:	**iaschex**
function	:	the integer number in IWA is converted to ASCII hexadecimal and placed in SWA
BASIC 1 address	:	&9ED0
BASIC 2 address	:	&9EDF
entry conditions	:	&15 must be #&FF
	:	Y must be #&40
	:	@% must be set as appropriate
comments	:	the routine returns with &36 set to the length of the string
exit status	:	IWA destroyed
	:	FWA destroyed
	:	FWB destroyed
	:	SWA = result
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
	:	&36 = length of string
typical timing	:	216 microseconds

subroutine name	:	**ifpa**
function	:	the integer number in IWA is converted to floating point and placed in FWA
BASIC 1 address	:	&A2AF
BASIC 2 address	:	&A2BE
entry conditions	:	none
exit status	:	IWA destroyed
	:	FWA = result
	:	FWB destroyed
	:	SWA unchanged
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
typical timing	:	149 microseconds

4.4 ASCII Conversion Demonstration

The following program is written in BASIC 2. It performs the following tasks:

a) It asks the user to supply a number. This can be either integer or decimal, with or without a sign.

b) It uses OSWORD 0 to read the user's number into the SWA.

c) It converts this number to either binary integer or floating point using 'ascnum'

d) It then converts the number back into fixed format ASCII decimal with 5 decimal places, overwriting the SWA. This result is then printed.

e) note that the conversion process in d) references a routine called 'numasc' which represents either fpascdec or iascdec, this being controlled by the value in Y prior to the call.

```
  0 ascnum = &AC34:numasc = &9EDF
 10 osword = &FFF1:osbyte = &FFF4
 20 osnewl = &FFE7:oswrch = &FFEE
 30 DIM mc% 500
 40 FOR pass% = 0 TO 2 STEP 2
 50 P% = mc%
 60 [OPT  pass%
 70 .msg       EQUB 12             \ clear screen
 80            EQUB 31             \ cursor
 90            EQUB 1              \ x = 1
100            EQUB 12             \ y = 12
110    EQUS "enter your number >"  \ message
120 .msgl      EQUB msgl - msg     \ msg length
130 .osbuff    EQUW &600           \ point to SWA
140            EQUB 20             \ max size
150            EQUB &2A            \ min value
160            EQUB &39            \ max value
170 .osbuffadd EQUW osbuff         \ pointer to osbuff
180 .start
190    LDX  #0                     \ zeroise loop counter
200 .loopmsg
210    LDA  msg,X                  \ get next byte of msg
220    JSR  oswrch                 \ print it
230    INX                         \ bump X
240    CPX  msgl                   \ end of msg ?
250    BNE  loopmsg                \ no - back
260                                \ get a number from keyboard
270 .reply
280    LDX  osbuffadd              \ X = to
290    LDY  osbuffadd+1            \ Y = hi
300    LDA  #0                     \ OSWORD 0
310    JSR  osword                 \ to read reply
```

```
320     BCC  notesc              \ not ESCAPE
330     LDA  #&7E                \ acknowledge
340     JSR  osbyte              \ ESCAPE
350     JMP  reply               \ try again
360 .notesc
370     STY  &36                 \ set up reply length
380     JSR  ascnum              \ convert to binary
390     TAY                      \ save variable type
400     LDA  #0                  \ set decimal
410     STA  &15                 \ print
420     LDA  #2                  \ set format
430     STA  &402                \ = 2
440     LDA  #5                  \ set decimal places
450     STA  &401                \ = 5
460     JSR  numasc              \ convert it back
470     JSR  osnewl              \ new line
480     LDX  #0                  \ zeroise loop counter
490 .ploop
500     LDA  &600,X              \ get next byte of ASCII
510     JSR  oswrch              \ print it
520     INX                      \ bump X
530     CPX  &36                 \ end of print
540     BCC  ploop               \ no - back
550     JSR  osnewl              \ new line
560     RTS
570 ]
580 NEXT pass%
590 CALL start
600 END

>RUN
enter your number >1234.567

1234.56700
```

5 MATHEMATICAL FUNCTIONS

The mathematical functions are surprisingly simple to use. No new ground has to be covered to explain their use. However, the way in which many of them work may be of some interest. It certainly has a bearing on the time they take to run.

It might be expected that the BASIC ROM would contain tables of sines, cosines etc. This is not true. Tables would use up far too much memory. Most of the mathematical functions can be expressed as series. For example:

```
           a   a*a   a*a*a   a*a*a*a
exp(a)=1 + - + --- + ----- + ------- + .............. etc.
           1   2*1   3*2*1   4*3*2*1
```

This is an example of a convergent series. Each successive term in the series provides a value smaller than the previous term. Thus if enough terms are taken a good approximation results. The more terms that are taken, the longer it takes to execute; fewer terms reduce the accuracy of the final answer. In practice, therefore, the number of terms considered is a compromise between accuracy and execution time.

In any event, these routines can never be fast to execute. It follows that if they are used in a loop with a large number of iterations, the effect on execution time is significant. The demonstration program in this chapter, which is only intended to show how to use the mathematics routines, is an example of slow circle drawing. The slowness is due to the fact that both sine and cosine functions are used within a loop.

5.1 Mathematical Functions Routines Summary

All of the functions summarised below, bar pi, work in the same way. The floating point number on which the function is to operate is placed in FWA. After the routine has been executed, the required result is to be found in FWA. Note that acs requires two subroutine calls, one immediately after the other. Note also that each function is equivalent to the BASIC function with the same name (but in upper case letters).

Name		BASIC 1 address	BASIC 2 address	Function
acs	1)	&A8CF	&A8DD	FWA = acs (FWA)
	2)	&A929	&A927	
asn		&A8CF	&A8DD	FWA = asn (FWA)
atn		&A90A	&A90A	FWA = atn (FWA)
cos		&A98C	&A990	FWA = cos (FWA)
deg		&ABEA	&ABC5	FWA = deg (FWA)
exp		&AAB7	&AA94	FWA = exp (FWA)
ln		&A807	&A801	FWA = ln (FWA)
log		&ABD0	&ABAB	FWA = log (FWA)
pi		&ABF0	&ABCB	FWA = PI
rad		&ABD9	&ABB4	FWA = rad (FWA)
sin		&A997	&A99B	FWA = sin (FWA)
sqr		&A7B7	&A7B7	FWA = sqr (FWA)
tan		&A6CC	&A6C1	FWA = tan (FWA)

5.2 Mathematical Functions Routines Description

subroutine name	:	**acs**
function	:	FWA = acs (FWA)
BASIC 1 address	:	&A8CF then &A929
BASIC 2 address	:	&A8DD then &A927
entry conditions	:	FWA contains a floating point number between −1 and +1
exit status	:	IWA unchanged
	:	FWA = result
	:	FWB destroyed
	:	SWA unchanged
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
typical timing	:	32567 microseconds

subroutine name	:	**asn**
function	:	FWA = asn (FWA)
BASIC 1 address	:	&A8CF
BASIC 2 address	:	&A8DD
entry conditions	:	FWA contains a floating point number between −1 and +1
exit status	:	IWA unchanged
	:	FWA = result
	:	FWB destroyed
	:	SWA unchanged
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
typical timing	:	31970 microseconds

subroutine name	:	**atn**
function	:	FWA = atn (FWA)
BASIC 1 address	:	&A90A
BASIC 2 address	:	&A90A
entry conditions	:	FWA contains a floating point number between −1E38 and +1E38
exit status	:	IWA unchanged
	:	FWA = result
	:	FWB destroyed
	:	SWA unchanged
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
typical timing	:	19110 microseconds

subroutine name	:	**cos**
function	:	FWA = cos (FWA)
BASIC 1 address	:	&A98C
BASIC 2 address	:	&A990
entry conditions	:	FWA contains number of radians in floating point
exit status	:	IWA unchanged
	:	FWA = result
	:	FWB destroyed
	:	SWA unchanged
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
error reports	:	accuracy lost
typical timing	:	26787 microseconds

subroutine name	:	**deg**
function	:	FWA = deg (FWA)
BASIC 1 address	:	&ABEA
BASIC 2 address	:	&ABC5
entry conditions	:	FWA contains number of radians in floating point
exit status	:	IWA unchanged
	:	FWA = result
	:	FWB destroyed
	:	SWA unchanged
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
typical timing	:	1711 microseconds

subroutine name	:	**exp**
function	:	FWA = exp (FWA)
BASIC 1 address	:	&AAB7
BASIC 2 address	:	&AA94
entry conditions	:	FWA contains a valid floating point argument
exit status	:	IWA unchanged
	:	FWA = result
	:	FWB destroyed
	:	SWA unchanged
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
error reports	:	exp range
typical timing	:	14997 microseconds

subroutine name	:	**ln**
function	:	FWA = ln (FWA)
BASIC 1 address	:	&A807
BASIC 2 address	:	&A801
entry conditions	:	FWA contains a valid floating point argument
exit status	:	IWA unchanged
	:	FWA = result
	:	FWB destroyed
	:	SWA unchanged
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
error reports	:	log range
typical timing	:	17192 microseconds

subroutine name	:	**log**
function	:	FWA = log (FWA)
BASIC 1 address	:	&ABD0
BASIC 2 address	:	&ABAB
entry conditions	:	FWA contains a valid floating point argument
exit status	:	IWA unchanged
	:	FWA = result
	:	FWB destroyed
	:	SWA unchanged
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
error reports	:	log range
typical timing	:	1551 microseconds

subroutine name	:	**pi**
function	:	FWA = PI
BASIC 1 address	:	&ABF0
BASIC 2 address	:	&ABCB
entry conditions	:	none
comments	:	FWA set to 3.14159265
exit status	:	IWA unchanged
	:	FWA = result
	:	FWB unchanged
	:	SWA unchanged
	:	A destroyed
	:	X unchanged
	:	Y destroyed
	:	P destroyed
	:	&4B,&4C destroyed
typical timing	:	87 microseconds

subroutine name	:	**rad**
function	:	FWA = rad (FWA)
BASIC 1 address	:	&ABD9
BASIC 2 address	:	&ABB4
entry conditions	:	FWA contains number of degrees in floating point
exit status	:	IWA unchanged
	:	FWA = result
	:	FWB destroyed
	:	SWA unchanged
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
typical timing	:	1566 microseconds

subroutine name	:	**sin**
function	:	FWA = sin (FWA)
BASIC 1 address	:	&A997
BASIC 2 address	:	&A99B
entry conditions	:	FWA contains number of radians in floating point
exit status	:	IWA unchanged
	:	FWA = result
	:	FWB destroyed
	:	SWA unchanged
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
error reports	:	accuracy lost
typical timing	:	15483 microseconds

subroutine name	:	**sqr**
function	:	FWA = sqr (FWA)
BASIC 1 address	:	&A7B7
BASIC 2 address	:	&A7B7
entry conditions	:	FWA contains a valid floating point argument
exit status	:	IWA unchanged
	:	FWA = result
	:	FWB destroyed
	:	SWA unchanged
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
error reports	:	–ve root
typical timing	:	8783 microseconds

subroutine name	:	**tan**
function	:	FWA = tan (FWA)
BASIC 1 address	:	&A6CC
BASIC 2 address	:	&A6C1
entry conditions	:	FWA contains number of radians in floating point
exit status	:	IWA unchanged
	:	FWA = result
	:	FWB destroyed
	:	SWA unchanged
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
error reports	:	accuracy lost
typical timing	:	41770 microseconds

5.3 Mathematical Functions Demonstration

The following program draws a circle in mode 0, centred at 600,500 and with a radius of 400. It uses the polygon method, with 100 sides in the figure. In BASIC, the program could be written as:

```
 10 MODE 0
 20 xcentre = 600
 30 ycentre = 500
 40 increment = 2*PI/100
 50 stop = 2*PI
 60 radius = 400
 70 MOVE xcentre+radius,ycentre
 80 FOR angle = 0 TO stop STEP increment
 90     DRAW xcentre + radius * COS(angle) ,
        ycentre + radius * SIN(angle)
100     NEXT
110 END
```

It will be seen that the equivalent BASIC 2 assembly language code is somewhat more long-winded to write. Paradoxically, the machine code generated occupies less memory and runs a bit faster than the BASIC code above, in spite of the fact that it was written with clarity as the main objective rather than efficiency.

Perhaps the most striking feature of the assembly language code is that it is all so simple, the clever code being all within the BASIC subroutines called.

```
 10 aunp   = &A3B5:aplus = &A500
 20 apack  = &A38D:atest = &9A5F
 30 amult  = &A656:fpi1  = &A3E4
 40 cos    = &A990:sin   = &A99B
 50 oswrch = &FFEE
 60 DIM mc% 500
 70 FOR pass% = 0 TO 2 STEP 2
 80 P% = mc%
 90 [OPT  pass%
100 .xcentre
110           OPT  FNEQUF(600)
120 .xadd     EQUW xcentre
130 .ycentre
140           OPT  FNEQUF(500)
150 .yadd     EQUW ycentre
160 .radius
170           OPT  FNEQUF(400)
180 .radadd   EQUW radius
190 .angle
200           OPT  FNEQUF(0)
210 .angadd   EQUW angle
220 .increment
230           OPT  FNEQUF(2*PI/100)
```

```
240 .incadd   EQUW increment
250 .stop
260           OPT  FNEQUF(2*PI)
270 .stopadd  EQUW stop
280 .plot     EQUB 25
290 .parm1    EQUB 4
300 .xcoord   EQUW 1000
310 .ycoord   EQUW 500
320 .mode0    EQUW &16
330 \
340 .start
350    LDA    mode0          \ MODE 0
360    JSR    oswrch         \
370    LDA    mode0+1        \
380    JSR    oswrch         \
390    JSR    doplot         \ MOVE
400    LDA    #5             \ reset for
410    STA    parm1          \ PLOT
420 .mainloop
430    JSR    pointincr      \ point &4B,&4C at increment
440    JSR    aunp           \ unpack into FWA
450    JSR    pointangle     \ point &4B,&4C at angle
460    JSR    aplus          \ add them
470    JSR    apack          \ put result in angle
480    JSR    pointstop      \ point &4B,&4C at stop
490    JSR    atest          \ test for end
500    BCC    alldone        \ yes - alldone
510    JSR    doxcoord       \ calculate X coordinate
520    JSR    doycoord       \ calculate Y coordinate
530    JSR    doplot         \ plot it
540    JMP    mainloop       \ and back
550 .alldone
560    RTS                   \ bye bye
570 \
580 .doxcoord                \ CALCULATE X COORD
590    JSR    pointangle     \ point &4B,&4C at angle
600    JSR    aunp           \ unpack into FWA
610    JSR    cos            \ get cosine
620    JSR    pointrad       \ point &4B,&4C at radius
630    JSR    amult          \ multiply
640    JSR    pointx         \ point &4B,&4C at xcentre
650    JSR    aplus          \ add
660    JSR    fpi1           \ convert to integer
670    LDX    #0             \ set loop counter
680 .doxloop
690    LDA    &2A,X          \ get next byte of IWA
700    STA    xcoord,X       \ save it
710    INX                   \ bump X
720    CPX    #4             \ are we done ?
730    BCC    doxloop        \ no - back
740    RTS                   \ back
750 \
```

```
 760 .doycoord                    \ CALCULATE Y COORD
 770    JSR    pointangle         \ point &4B,&4C at angle
 780    JSR    aunp               \ unpack into FWA
 790    JSR    sin                \ get sine
 800    JSR    pointrad           \ point &4B,&4C at radius
 810    JSR    amult              \ multiply
 820    JSR    pointy             \ point &4B,&4C at ycentre
 830    JSR    aplus              \ add
 840    JSR    fpi1               \ convert to integer
 850    LDX    #0                 \ set loop counter
 860 .doyloop
 870    LDA    &2A,X              \ get next byte of IWA
 880    STA    ycoord,X           \ save it
 890    INX                       \ bump X
 900    CPX    #4                 \ are we done ?
 910    BCC    doyloop            \ no - back
 920    RTS                       \ back
 930 \
 940 .doplot                      \ PLOT
 950    LDX    #0                 \ zeroise loop conter
 960 .plotloop
 970    LDA    plot,X             \ get next byte of plot
 980    JSR    oswrch             \ print it
 990    INX                       \ bump X
1000    CPX    #6                 \ end of plot ?
1010    BCC    plotloop           \ no - back
1020    RTS
1030 \
1040 .pointincr                   \ POINT &4B,&4C at increment
1050    LDA    incadd             \ lo address of increment
1060    STA    &4B                \ save
1070    LDA    incadd+1           \ hi address of increment
1080    STA    &4C                \ save
1090    RTS                       \ back
1100 \
1110 .pointangle                  \ POINT &4B,&4C at angle
1120    LDA    angadd             \ lo address of angle
1130    STA    &4B                \ save
1140    LDA    angadd+1           \ hi address of angle
1150    STA    &4C                \ save
1160    RTS
1170 \
1180 .pointstop                   \ POINT &4B,&4C at stop
1190    LDA    stopadd            \ lo address of stop
1200    STA    &4B                \ save
1210    LDA    stopadd+1          \ hi address of stop
1220    STA    &4C                \ save
1230    RTS                       \ back
1240 \
1250 .pointrad                    \ POINT &4B,84C at radius
1260    LDA    radadd             \ lo address of radius
1270    STA    &4B                \ save
```

```
1280    LDA    radadd+1       \ hi address of radius
1290    STA    &4C            \ save
1300    RTS                   \ back
1310 \
1320 .pointx                  \ POINT &4B,&4C at xcentre
1330    LDA    xadd           \ lo address of xcentre
1340    STA    &4B            \ save
1350    LDA    xadd+1         \ hi address of xcentre
1360    STA    &4C            \ save
1370    RTS                   \ back
1380 \
1390 .pointy                  \ POINT &4B,&4C at ycentre
1400    LDA    yadd           \ lo address of ycentre
1410    STA    &4B            \ save
1420    LDA    yadd+1         \ hi address of ycentre
1430    STA    &4C            \ save
1440    RTS                   \ back
1450 ]
1460 NEXT pass%
1470 CALL start
1480 END
1490 DEF FNEQUF(Z)
1500 I% = 3 +?&4B4 +256*?&4B5
1510 FOR J% = 1 TO 5
1520     ?P% = ?I%
1530     P%  = P% + 1
1540     I%  = I% + 1
1550 NEXT
1560 = pass%
```

6 RANDOM NUMBERS

The BBC Micro generates random numbers by applying a pseudo-randomising algorithm to an initial value, known as the seed. After each application of the algorithm, the resulting random number is not only returned to the user, it also becomes the new seed. Naturally, facilities are also provided for the user to supply a value to initialise the seed.

On power-up, the same seed is planted in the random number data field each time. For this reason, the algorithm cannot be completely random. The user can of course alter this by supplying a seed at the start of the program. Providing the user does not previously set TIME in the program, RND(−TIME) will achieve this.

6.1 Random Number Work Area

The area of memory dedicated to random numbers starts at &0D and ends at &11. This five byte area will be called the Random Number Work Area or RWA. It is the source data for all random number operations. In BASIC, there are many varieties of the RND statement.

RND	generates an integer random number between −2147483648 and +2147483647. It executes the algorithm and copies &0D to &10 into the IWA.
RND(−X)	resets the seed to X and returns −X. It copies the IWA into &0D to &10, sets &11 to #&40 and leaves the IWA unchanged at −X.
RND(0)	repeats the last random number returned by RND(1). It simply copies the RWA into the FWA.
RND(1)	generates a floating point random number between 0 and 0.999999. It executes the algorithm and copies the RWA into the FWA.
RND(X)	generates an integer random number between 1 and X. After executing the algorithm, the result is returned in the IWA.

6.2 Random Number Routine Summary

The individual functions supported by the single BASIC statement RND have different entry points. Therefore, in assembly language programming, it is necessary to regard each function as a separate routine. A program is supplied which demonstrates each routine.

Name	BASIC 1 address	BASIC 2 address	Function	
rnd0	&AF9B	&AF6C	RND(0)	repeat last rnd1
rnd1	&AF98	&AF69	RND(1)	FWA = from 0 to 0.999999
rndi	&AF80	&AF51	RND	IWA = random integer number
rndseed	&AF6E	&AF3F	RND(-X)	RWA seeded with X
rndx	&AF53	&AF24	RND(X)	IWA = from 1 to X

6.3 Random Number Routines Description

subroutine name	:	**rnd0**
function	:	FWA = value returned by last rnd1
BASIC 1 address	:	&AF9B
BASIC 2 address	:	&AF6C
entry conditions	:	none
exit status	:	IWA destroyed
	:	FWA = result
	:	FWB unchanged
	:	RWA unchanged
	:	SWA unchanged
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
typical timing	:	82 microseconds

subroutine name	:	**rnd1**
function	:	FWA = random number from 0 to 0.999999
BASIC 1 address	:	&AF98
BASIC 2 address	:	&AF69
entry conditions	:	none
exit status	:	IWA destroyed
	:	FWA = result
	:	FWB unchanged
	:	RWA changed
	:	SWA unchanged
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
typical timing	:	793 microseconds

subroutine name	:	**rndi**
function	:	IWA = a random integer number from −2147483648 to +2147483647
BASIC 1 address	:	&AF80
BASIC 2 address	:	&AF51
entry conditions	:	none
exit status	:	IWA = result
	:	FWA unchanged
	:	FWB unchanged
	:	RWA changed
	:	SWA unchanged
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
typical timing	:	745 microseconds

subroutine name	:	**rndseed**
function	:	RWA = IWA + #&40 in fifth byte
BASIC 1 address	:	&AF6E
BASIC 2 address	:	&AF3F
entry conditions	:	IWA set to value to be seeded.
	:	Unlike in BASIC, it does not have to be negative.
exit status	:	IWA unchanged
	:	FWA destroyed
	:	FWB unchanged
	:	RWA re-seeded
	:	SWA unchanged
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
typical timing	:	42 microseconds

subroutine name	:	**rndx**
function	:	IWA = random number from I to value in IWA
BASIC 1 address	:	&AF53
BASIC 2 address	:	&AF24
entry conditions	:	IWA = maximum value of random number
	:	&4,5 should be pointed to HIMEM
exit status	:	IWA = result
	:	FWA destroyed
	:	FWB unchanged
	:	RWA changed
	:	SWA unchanged
	:	A destroyed
	:	X destroyed
	:	Y destroyed
	:	P destroyed
typical timing	:	2974 microseconds

6.4 Random Number Demonstration

The following program, written in BASIC 2, demonstrates all the random number functions. Note that the last routine to be tested is rndseed. Repeated runs of this program will produce identical results because of this. Delete lines 1030 to 1070 for more random results. Note also that rnd0 does indeed return the same result as the preceding rnd1.

```
  10 rndi = &AF51:rndseed = &AF3F
  20 rnd0 = &AF6C:rnd1    = &AF69
  30 rndx = &AF24:apack   = &A38D
  40 DIM mc% 200
  50 FOR pass% = 0 TO 2 STEP 2
  60 P% = mc%
  70 [OPT pass%
  80 .seedval    EQUD -123456
  90 .rndxval    EQUD 2345678
 100 .result     EQUD 0
 110             EQUB 0
 120 \
 130 .rnditest
 140    JSR rndi             \ do rndi
 150    JSR saveiwa          \ save IWA in result
 170    RTS                  \ bye bye
 180 \
 190 .rndseedtest
 200    LDX #0               \ zeroise loop counter
 210 .rndseedloop
 220    LDA seedval,X        \ get next byte of seed
 230    STA &2A,X            \ save in IWA
 240    INX                  \ bump X
 250    CPX #4               \ end of loop ?
 260    BCC rndseedloop      \ no - back
 270    JSR rndseed          \ seed IWA into RWA
 280    RTS                  \ bye bye
 290 \
 300 .rnd1test
 310    JSR rnd1             \ do rnd1
 320    JSR savefwa          \ save FWA in result
 330    RTS                  \ bye bye
 340 \
 350 .rnd0test
 360    JSR rnd0             \ do rnd0
 370    JSR savefwa          \ save FWA in result
 380    RTS                  \ bye bye
 390 \
 400 .rndxtest
 410    LDA &6               \ HIMEM lo
 420    STA &4               \ save in &4
 430    LDA &7               \ HIMEM hi
 440    STA &5               \ save in &5
```

```
450    LDX #0              \ zeroise loop counter
460 .rndxloop
470    LDA rndxval,X       \ get next byte of rndxval
480    STA &2A,X           \ save in IWA
490    INX                 \ bump X
500    CPX #4              \ end of loop ?
510    BCC rndxloop        \ no - back
520    JSR rndx            \ do rndx
530    JSR saveiwa         \ copy IWA into result
540    RTS                 \ bye bye
550 \
560 .saveiwa
570    LDX #0              \ zeroise loop counter
580 .loopiwa
590    LDA &2A,X           \ get next byte of IWA
600    STA result,X        \ save in result
610    INX                 \ bump X
620    CPX #4              \ end of loop ?
630    BCC loopiwa         \ no - back
640    RTS                 \ back
650 \
660 .savefwa
670    LDA #result MOD 256 \ lo address of result
680    STA &4B             \ save it
690    LDA #result DIV 256 \ hi address of result
700    STA &4C             \ save it
710    JSR apack           \ pack FWA into result
720    RTS                 \ back
730 ]
740 NEXT pass%
750 CLS
760 Z = 0
770 PRINT
780 PRINT "Test of rnd1"
790 CALL rnd1test
800 REM set Z = result
810 I% = 3 + ?&4B4 + 256*?&4B5
820 FOR J% = 0 TO 4
830     ?(I%+J%) = ?(result+J%)
840     NEXT
850 PRINT "result returned = ";Z
860 PRINT
870 PRINT "Test of rnd0"
880 CALL rnd0test
890 REM set Z = result
900 I% = 3 + ?&4B4 + 256*?&4B5
910 FOR J% = 0 TO 4
920     ?(I%+J%) = ?(result+J%)
930     NEXT
940 PRINT "result returned = ";Z
950 PRINT
960 PRINT "Test of rndi"
```

```
 970 CALL rnditest
 980 PRINT "result returned = ";!result
 990 PRINT
1000 PRINT "Test of rndx"
1010 CALL rndxtest
1020 PRINT "result returned = ";!result
1030 PRINT
1040 PRINT "Test of rndseed"
1050 CALL rndseedtest
1060 PRINT "RWA bytes 0-3   = ";!&D
1070 PRINT "RWA byte  4     = ";?&11
1080 END
```

```
>RUN

Test of rnd1
result returned = 0.424982422

Test of rnd0
result returned = 0.424982422

Test of rndi
result returned = -1.34400018E9

Test of rndx
result returned = 2241095

Test of rndseed
RWA bytes 0-3   = -123456
RWA byte  4     = 64
```

7 BASIC MEMORY MAP

Descriptions of many of the memory areas used by the BASIC interpreter have already been given. This memory map is included to allow the reader to explore further into the BASIC ROM if desired. Doubtless there are other subroutines not mentioned in this book which the reader could use in specific applications.

7.1 Zero Page Dedicated Locations

These addresses are used identically in BASIC 1 and BASIC 2. All two-byte address pointers are low,high.

```
-------------------------------------------------------------
&00 - &01 LOMEM

          pointer to the start of BASIC variables.
-------------------------------------------------------------
&02 - &03 VARTOP

          pointer to the end of BASIC variables.
-------------------------------------------------------------
&04 - &05 BASIC STACK POINTER

          pointer to most recent entry in the BASIC stack.
-------------------------------------------------------------
&06 - &07 HIMEM

          pointer to the start of screen memory-mapped area.
-------------------------------------------------------------
&08 - &09 ERL

          the address of the instruction which errored.
-------------------------------------------------------------
&0A       BASIC TEXT POINTER OFFSET

          the offset with respect to &B,&C of the
          byte of BASIC text currently being processed.
-------------------------------------------------------------
&0B - &0C BASIC TEXT POINTER

          pointer to start of BASIC text line being processed.
-------------------------------------------------------------
&0D - &11 RND WORK AREA

          see Chapter 6.
-------------------------------------------------------------
&12 - &13 TOP

          pointer to the end of BASIC program not
          including variables.
```

&14	PRINT BYTES the number of bytes in a print output field.
&15	PRINT FLAG 0 = decimal -ve = hexadecimal
&16 - &17	ERROR ROUTINE VECTOR pointer to the address of the BASIC error routine.
&18	PAGE DIV 256 page number where BASIC program starts.
&19 - &1A	SECONDARY BASIC TEXT POINTER secondary &B,&C.
&1B	SECONDARY BASIC TEXT OFFSET secondary &A.
&1C - &1D	DATA POINTER pointer to next DATA item.
&1E	COUNT number of bytes printed since last new line.
&1F	LISTO FLAG a number ANDed from the list below: 0 = LISTO off 1 = insert space after line number 2 = indent FOR loops 4 = indent REPEAT loops
&20	TRACE FLAG &00 = trace off &FF = trace on
&21 - &22	MAXIMUM TRACE LINE NUMBER
&23	WIDTH as set by WIDTH command.
&24	REPEAT LEVEL number of nested REPEATs outstanding.

&25	GOSUB LEVEL number of nested GOSUBs outstanding.
&26	15*FOR LEVEL 15 * number of nested FOR loops outstanding.
&27	VARIABLE TYPE &00 = byte &04 = integer &05 = floating point &81 = string &A4 = function name &F2 = procedure name
&28	OPT FLAG bit 0 = list flag bit 1 = errors flag bit 2 = relocate flag (BASIC 2 only)
&29 - &2B	ASSEMBLER CODE BUFFER &29 = opcode &2A = operand low byte &2B = operand high byte

7.2 Zero Page Multiple Use Locations

These addresses are used identically in BASIC 1 and BASIC 2. The main uses only are given.

&2A - &2D	INTEGER WORK AREA
&2E - &35	FLOATING POINT WORK AREA A
&36	LENGTH OF STRING BUFFER
&37 - &3A	GENERAL AREAS
&3B - &42	FLOATING POINT WORK AREA B
&43 - &4F	FLOATING POINT TEMPORARY AREAS
&50 - &6F	not used

7.3 Resident Integer Variables

All are stored with the least significant byte first.

```
&400 - &403  @%
&404 - &407  A%
&408 - 840B  B%
&40C - &40F  C%
&410 - &413  D%
&414 - &417  E%
&418 - &41B  F%
&41C - &41F  G%
&420 - &423  H%
&424 - &427  I%
&428 - &42B  J%
&42C - &42F  K%
&430 - &433  L%
&434 - &437  M%
&438 - &43B  N%
&43C - 843F  O%
&440 - &443  P%
&444 - &447  Q%
&448 - &44B  R%
&44C - &44F  S%
8450 - &453  T%
&454 - &457  U%
&458 - &45B  V%
&45C - &45F  W%
&460 - &463  X%
&464 - 8467  Y%
8468 - &46B  Z%
```

7.4 Floating Point Temporary Areas

All are stored in 5 byte packed floating point format.

```
&46C - &470  TEMP 1
&471 - &475  TEMP 2
&476 - &47A  TEMP 3
&47B - &47F  TEMP 4
```

7.5 Variable Pointer Table

There is a variable look-up table for each character with which a variable name can start. Each pair of addresses is a lo,hi pointer to the variables whose names start with a particular character.

```
&480-&481 = @    &4AA-&4AB = U    &4D2-&4D3 = i
&482-&483 = A    &4AC-&4AD = V    &4D4-&4D5 = j
&484-&485 = B    &4AE-&4AF = W    &4D6-&4D7 = k
&486-&487 = C    &4B0-&4B1 = X    &4D8-&4D9 = l
&488-&489 = D    &4B2-&4B3 = Y    &4DA-&4DB = m
&48A-&48B = E    &4B4-&4B5 = Z    &4DC-&4DD = n
&48C-&48D = F    &4B6-&4B7 = [    &4DE-&4DF = o
&48E-&48F = G    &4B8-&4B9 = \    &4E0-&4E1 = p
&490-&491 = H    &4BA-&4BB = ]    &4E2-&4E3 = q
&492-&493 = I    &4BC-&4BD = ^    &4E4-&4E5 = r
&494-&495 = J    &4BE-&4BF = _    &4E6-&4E7 = s
&496-&497 = K    &4C0-&4C1 = £    &4E8-&4E9 = t
&498-&499 = L    &4C2-&4C3 = a    &4EA-&4EB = u
&49A-&49B = M    &4C4-&4C5 = b    &4EC-&4ED = v
&49C-&49D = N    &4C6-&4C7 = c    &4EE-&4EF = w
&49E-&49F = O    &4C8-&4C9 = d    &4F0-&4F1 = x
&4A0-&4A1 = P    &4CA-&4CB = e    &4F2-&4F3 = y
&4A2-&4A3 = Q    &4CC-&4CD = f    &4F4-&4F5 = z
&4A4-&4A5 = R    &4CE-&4CF = g    &4F6-&4F7 = procedures
&4A6-&4A7 = S    &4D0-&4D1 = h    &4F8-&4F9 = functions
&4A8-&4A9 = T
```

The use of these look-up tables can best be explained by an example. Suppose we run the following program:

```
10 DIM ARRAY%(3)
20 FOR A% = 0 TO 3
30 ARRAY%(A%) = A%
40 NEXT
50 ALPHA$ = "TESTING"
60 AFPNUM = 1.0
70 AINT% = 1
80 END
```

Assume that &482 contains #&0E and &483 contains #&16. Then the look-up table for variables starting with the letter ‘A’ will be found at &160E which will contain the following:

```
&160E = #&2A       pointer to next variable
&160F = #&16       at &162A

&1610 = #&52   R   rest
&1611 = #&52   R   of
&1612 = #&41   A   name
&1613 = #&59   Y   for
&1614 = #&25   %   ARRAY%(
&1615 = #&28   (
```

&1616 = #&00		end of name marker.
&1617 = #&03		2 * number of dimensions + 1
&1618 = #&04		number of elements in 1st dimension (LSB)
&1619 = #&00		number of elements in 1st dimension (MSB)
&161A = #&00		contents
&161B = #&00		of
&161C = #&00		ARRAY%(0)
&161D = #&00		= 0
&161E = #&01		contents
&161F = #&00		of
&1620 = #&00		ARRAY%(1)
&1621 = #&00		= 1
&1622 = #&02		contents
&1623 = #&00		of
&1624 = #&00		ARRAY%(2)
&1625 = #&00		= 2
&1626 = #&03		contents
&1627 = #&00		of
&1628 = #&00		ARRAY%(3)
&1629 = #&00		= 3
&162A = #&3D		pointer to next variable
&162B = #&16		at &163D
&162C = #&4C	L	rest
&162D = #&50	P	of
&162E = #&48	H	variable
&162F = #&41	A	name
&1630 = #&24	$	for ALPHA$
&1631 = #&00		end of name marker
&1632 = #&36		pointer to &1636 which contains
&1633 = #&16		current value of ALPHA$
&1634 = #&07		maximum size allocated to ALPHA$
&1635 = #&07		current size of ALPHA$
&1636 = #&54	T	contents
&1637 = #&45	E	of
&1638 = #&53	S	ALPHA$
&1639 = #&54	T	
&163A = #&49	I	
&163B = #&4E	N	
&163C = #&47	G	
&163D = #&4A		pointer to next variable
&163E = #&16		at &164A

```
&163F = #&46   F   rest
&1640 = #&50   P   of
&1641 = #&4E   N   name
&1642 = #&55   U   for
&1643 = #&4D   M   AFPNUM

&1644 = #&00       end of name marker

&1645 = #&81       contents of
&1646 = #&00       AFPNUM
&1647 = #&00       = +1.0
&1648 = #&00       in 5 byte, packed,
&1649 = #&00       floating point format

&164A = #&00       pointer to next variable = &0000
&164B = #&00       so last entry in table

&164C = #&49   I   rest of
&164D = #&4E   N   variable
&164E = #&54   T   name for
&164F = #&25   %   AINT%

&1650 = #&00       end of name marker

&1651 = #&01       AINT%
&1652 = #&00       contains
&1653 = #&00       +1 in 4byte,
&1654 = #&00       integer format
```

Each entry in the look-up table starts with a 2 byte (lo,hi) pointer to the next entry in the table, except for the last entry which contains #&0000. Following this is the rest of the variable name, excluding its initial letter, and then a zero byte to indicate the end of the name. This is followed by a contents section, the exact format of which depends on the type of variable being stored. Thus integer variables are stored in 4 bytes, whilst floating point variables are stored in 5 bytes. String variables can change in size during the running of a program. Thus the table contains a pointer to the string (rather than the string itself), together with details of its allocated and current size. Arrays can be integer, floating point or string. Integer and floating point array contents are stored in the table, whilst string arrays point to the string. At the start of the contents, there are some additional bytes which define the size of the array.

7.6 BASIC Stacks and Buffers

```
&500 - &5FF  FOR/REPEAT/GOSUB STACK
&600 - &6FF  STRING WORK AREA / CALL PARAMETER BLOCK
&700 - &7FF  BASIC LINE INPUT BUFFER
```

BASIC reads text entered at the screen into the line input buffer. The text is then tokenised (each BASIC command is replaced by a number). If the input line starts with a line number, the tokenised line is inserted into a BASIC program at the proper place. Otherwise the tokenised line is executed immediately.

7.7 BASIC Token and Action Tables

Tokens range from &80 to &FF. BASIC has a token table which for each BASIC command contains:

a) the command name in ASCII
b) the token
c) a flag byte used by the interpreter

The token table is located at:

```
&806D - &8358  BASIC 1
&8071 - &836C  BASIC 2
```

The order of commands in the table is important since it specifies the minimum acceptable abbreviation of a command. A command will be recognised so long as it starts with one or more letters of that command and ends in a full-stop, provided the abbreviation does not match a command earlier in the table. The first command in the table is 'AND' for which 'A.' is sufficient. The second is 'ABS'. 'A.' cannot be used for ABS because the previous command, 'AND', is matched by it. However 'AB.' is quite satisfactory.

&8D is a special token used to prefix a BASIC line number. Commands lower than &8E (&8F in BASIC 1) are processed by the interpreter as and when they occur in a line. The rest of the commands have an associated action address found by looking up a two-part action address table. The first part contains the low byte addresses, whilst the second contains the high bytes. Each table is indexed by the token number less &8E (&8F in BASIC 1).

The action address tables are located at:

```
lo bytes  &835A - &83CA  BASIC 1
          &836D - &83DE  BASIC 2

hi bytes  &83CB - &843C  BASIC 1
          &83DF - &8450  BASIC 2
```

The following tables summarise BASIC commands in alphabetic order.

7.8 BASIC Tables Summary

BASIC COMMAND	BASIC 1 ADDRESS	BASIC 2 ADDRESS	TOKEN HEX
ABS	&AD8D	&AD6A	&94
ACS	&A8C6	&A8D4	&95
ADVAL	&A656	&AB33	&96
AND	-----	-----	&80
ASC	&ACC4	&AC9E	&97
ASN	&A8CC	&A8DA	&98
ATN	&A907	&A907	&99
AUTO	&905F	&90AC	&C6
BGET	&BF78	&BF6F	&9A
BPUT	&BF61	&BF58	&D5
CALL	&8E6C	&8ED2	&D6
CHAIN	&BF33	&BF2A	&D7
CHR$	&B3EE	&B3BD	&BD
CLEAR	&9326	&928D	&D8
CLOSE	&BF9E	&BF99	&D9
CLG	&8E57	&8EBD	&DA
CLS	&8E5E	&8EC4	&DB
COLOUR	&9346	&938E	&FB
COS	&A989	&A98D	&9B
=COUNT	&AF26	&AEF7	&9C
DATA	&8AED	&8B7D	&DC
DEF	&8AED	&8B7D	&DD
DEG	&ABE7	&ABC2	&9D
DELETE	&8ECE	&8F31	&C7
DIM	&90DD	&912F	&DE
DIV	-----	-----	&81
DRAW	&93A5	&93E8	&DF

BASIC COMMAND	BASIC 1 ADDRESS	BASIC 2 ADDRESS	TOKEN HEX
ELSE	-----	-----	&8B
END	&8A50	&8AC8	&E0
ENDPROC	&9310	&9356	&E1
ENVELOPE	&B49C	&B472	&E2
EOF	&ACDE	&ACB8	&C5
EOR	-----	-----	&82
ERL	&AFCE	&AF9F	&9E
ERR	&AFD5	&AFA6	&9F
ERROR	-----	-----	&85
EVAL	&AC12	&ABE9	&A0
EXP	&AAB4	&AA91	&A1
EXT	&BF4F	&BF46	&A2
FALSE	&AEF9	&AECA	&A3
FN	&B1C4	&B195	&A4
FOR	&B7DF	&B7C4	&E3
GCOL	&932F	&937A	&E6
GET	&AFE8	&AFB9	&A5
GET$	&AFEE	&AFBF	&BE
GOSUB	&B8B4	&B888	&E4
GOTO	&B8EB	&B8CC	&E5
=HIMEM	&AF32	&AF03	&93
HIMEM=	&9212	&925D	&D3
IF	&9893	&98C2	&E7
INKEY	&ACD3	&ACAD	&A6
INKEY$	&B055	&B026	&BF
INPUT	&BA62	&BA44	&E8
INSTR(	&AD08	&ACE2	&A7
INT	&AC9E	&AC78	&A8

BASIC COMMAND	BASIC 1 ADDRESS	BASIC 2 ADDRESS	TOKEN HEX	
LEFT$(	&AFFB	&AFCC	&C0	
LEN	&AF00	&AED1	&A9	
LET	&8B57	&8BE4	&E9	
LINE	-----	-----	&86	
line no.	-----	-----	&8D	
LIST	&B5B5	&B59C	&C9	
LN	&A804	&A7FE	&AA	
LOAD	&BF2D	&BF24	&C8	
LOCAL	&92D5	&9323	&EA	
LOG	&ABCD	&ABA8	&AB	
=LOMEM	&AF2B	&AEFC	&92	
LOMEM=	&9224	&926F	&D2	
MID$(	&B068	&B039	&C1	
MOD	-----	-----	&83	
MODE	&935A	&939A	&EB	
MOVE	&93A1	&93E4	&EC	
NEW	&8A7D	&8ADA	&CA	
NEXT	&B6AE	&B695	&ED	
NOT	&ACF7	&ACD1	&AC	
OFF	-----	-----	&87	
OLD	&8A3D	&8AB6	&CB	
ON	&B934	&B915	&EE	
OPENIN	&BF85	&BF78	&8E	&AD in BASIC 1
OPENOUT	&BF81	&BF7C	&AE	
OPENUP	-----	&BF80	&AD	BASIC 2 only
OR	-----	-----	&84	
OSCLI	-----	&BEC2	&FF	BASIC 2 only

BASIC COMMAND	BASIC 1 ADDRESS	BASIC 2 ADDRESS	TOKEN HEX
=PAGE	&AEEF	&AEC0	&90
PAGE=	&9239	&9283	&D0
PI	&ABF0	&ABCB	&AF
PLOT	&93AE	&93F1	&F0
POINT(	&AB64	&AB41	&B0
POS	&AB92	&AB6D	&B1
PRINT	&8D33	&8D9A	&F1
PROC	&92B6	&9304	&F2
=PTR	&BF50	&BF47	&8F
PTR=	&BF39	&BF30	&CF
RAD	&ABD6	&ABB1	&B2
READ	&BB39	&BB1F	&F3
REM	&8AED	&8B7D	&F4
RENUMBER	&8F37	&8FA3	&CC
REPEAT	&BBFF	&BBE4	&F5
REPORT	&BFE6	&BFE4	&F6
RESTORE	&BB00	&BAE6	&F7
RETURN	&B8D5	&B8B6	&F8
RIGHT$(	&B01D	&AFEE	&C2
RND	&AF78	&AF49	&B3
RUN	&BD29	&BD11	&F9

BASIC COMMAND	BASIC 1 ADDRESS	BASIC 2 ADDRESS	TOKEN HEX
SAVE	&BEFA	&BEF3	&CD
SGN	&ABAD	&AB88	&B4
SIN	&A994	&A998	&B5
SOUND	&B461	&B44C	&D4
SPC	-----	-----	&89
SQR	&A7B4	&A7B4	&B6
STEP	-----	-----	&88
STOP	&8A59	&8AD0	&FA
STR$	&B0C3	&B094	&C3
STRING$	&B0F1	&B0C2	&C4
TAB(	-----	-----	&8A
TAN	&A6C9	&A6BE	&B7
THEN	-----	-----	&8C
=TIME	&AEE3	&AEB4	&91
TIME=	&927B	&92C9	&D1
TO	&AF0B	&AEDC	&B8
TRACE	&9243	&9295	&FC
TRUE	&ACEA	&ACC4	&B9
UNTIL	&BBCC	&BBB1	&FD
USR	&ABFB	&ABD2	&BA
VAL	&AC55	&AC2F	&BB
VDU	&93EF	&942F	&EF
VPOS	&AB9B	&AB76	&BC
WIDTH	&B4CC	&B4A0	&FE

8 TIMINGS

Frequently the speed of a program determines its success or failure, especially in graphics applications. Thus, the execution time of a program is an important topic, not least to those who decide to use the technique advocated in this book, and a separate chapter on this topic is justified.

8.1 Units of time

The following units of time are frequently encountered when dealing with computers:

```
                              1
1 picosecond  = -----------------    seconds
                1,000,000,000,000
                                1
1 nanosecond  =     -------------    seconds
                    1,000,000,000
                                1
1 microsecond =         ---------    seconds
                        1,000,000
                                1
1 millisecond =             -----    seconds
                            1,000
                                1
1 centisecond =               ---    seconds
                              100
```

8.2 Computer Processor Speed

The speed of a cpu is determined by its cycle time. A cycle is the interval of time that elapses between successive pulses of the system clock. During a cycle the cpu performs a fundamental cpu operation. This operation could be to fetch a byte of data from RAM or to store a byte of data in RAM, for example. Each machine code instruction takes several cycles. The more complicated instructions take more cycles than the simpler ones. In fact the number of cycles needed for a 6502 instruction is always between 2 and 8.

The BBC micro has a cycle time of 0.5 microseconds, which is just another way of saying that it is clocked at 2 Megahertz (2 million times per second). Thus it can perform its fastest instructions (2 cycles) in 1 microsecond, whilst its slowest instructions (8 cycles) take 4 microseconds.

8.3 Program Speed

Because of the importance of program speed, typical timings have been given for each of BASIC's subroutines. Note that the time spent in a given subroutine is not a constant. The exact number of instructions executed will vary a little depending on the data values being acted upon. Nevertheless, these timings can be used to forecast whether or not a program is going to be fast enough without actually writing the program.

Thus, in the polygonal circle in chapter 5 it was recognised that in addition to some other code, the program would involve 100 sines and cosines. Timings given for sine and cosine predict that these functions alone will take 4.2 seconds. Broad-brush estimating such as this may well be sufficient to discard a technique. The calculation can be refined to give a more precise estimate if required.

Moreover, timings given in this book can often be used to estimate run times of purely BASIC programs. In general, calling BASIC's subroutines from machine code saves about 10% of the execution time. Thus by adding 10% to the figures in this book it is possible to get a good approximation of the run time of many BASIC program commands also.

8.4 Microsecond Timer

Timings given in this book exclude any set up code required by the subroutine, but include time taken by interrupt routines such as servicing the various clocks maintained by the software. The BASIC TIME facility has a resolution of 1 centisecond which is insufficiently precise for many purposes. The following code uses the User 6522 to obtain 1 microsecond resolution of time and may be used to time another piece of code. The code to be timed is placed at 'test'. Any set up code required, but not to be timed is placed at 'setup'.

```
 10 DIM mc% 500
 20 FOR pass% = 0 TO 2 STEP 2
 30 P% = mc%
 40 [OPT pass%
 50 .save  EQUW 0
 60 .time  EQUD 0
 70 .timer
 80   LDA &206                \ save interrupt
 90   STA save                \ handler address lo
100   LDA &207                \ and save
110   STA save+1              \ hi address too.
120   JSR setup               \ set up.
130   SEI                     \ prevent interrupts.
140   LDA #interrupt MOD 256 \ re-direct IRQ2V to
150   STA &206                \ our own
160   LDA #interrupt DIV 256 \ routine called
170   STA &207                \ "interrupt".
180   CLI                     \ allow interrupts again.
190   LDA #&DF                \ set up ACR of user 6522
200   AND &FE6B               \ to put TIMER 2
210   STA &FE6B               \ in count down mode.
220   LDA #&A0                \ enable TIMER 2
230   STA &FE6D               \ interrupts.
240   STA &FE6E
250   JSR newinterrupt        \ reset TIMER 2.
260   JSR test                \ do code to be timed.
270   LDA &FE68               \ get count down value
280   STA time                \ and save in "time".
290   LDA &FE69               \ also get
300   STA time+1              \ hi byte
310   SEI                     \ prevent interrupts.
320   LDA save                \ restore IRQ2V.
330   STA &206
340   LDA save+1
350   STA &207
360   LDA #&20                \ clear
370   STA &FE6D               \ TIMER 2 interrupts.
380   STA &FE6E
390   CLI                     \ enable interrupts
400   RTS                     \ bye bye
410 .interrupt
420   CLC                     \ bump time by 1
430   LDA time+2              \ every time count
440   ADC #1                  \ down
450   STA time+2              \ expires
460   BCC interruptdone
470   LDA time+3
480   ADC #0
490   STA time+3
500 .interruptdone
510   JSR newinterrupt        \ reset count down timer
520   JMP (save)              \ back to IRQ2V.
```

```
 530 .newinterrupt
 540   LDA #&FD              \ reset
 550   STA &FE68             \ TIMER 2
 560   LDA #&FF              \ as #&FFFD
 570   STA &FE69
 580   RTS                   \ out
1000 .test
1010   \ CODE TO BE TIMED
1020   RTS
2000 .setup
2010   \ SET UP FOR CODE TO BE TIMED
2020   RTS
3000 ]:NEXT pass%
3010 CALL timer
3020 ?(time) = &FF - ?(time)
3030 ?(time+1) = &FF - ?(time+1)
3040 PRINT !time;" microseconds"
3050 END
```

8.5 BASIC Timings

The following tables list all 69 BASIC routines in alphabetic order, together with their typical timings.

subroutine	microseconds
aclear	25
acomp	34
acopyb	36
acs	32,567
adiv	1,545
adiv10	360
aminus	254
amult	1,581
amult1	1,508
amult10	171
anorm	27
aone	37
apack	46
apack1	51
apack2	53
apack3	53
aplus	246
aplusb	58
aplus1	41
arecip	1,619
around	22
ascnum	1,748
asign	24
asn	31,970
atest	70
atn	19,110
aunp	49
aunp1	62
bclear	25
bcopya	36
bunp	51
cos	26,787

subroutine	microseconds	
deg	1,711	
exp	14,997	
fpascdec	4,878	
fpaschex	582	
fpi1	405	
fpi2	387	
iascdec	5,369	
iaschex	216	
icomp	31	
idiv	794	
ifpa	149	
iin	34	
iminus	52	
imod	782	
imult	164	
ineg1	20	
iout	37	
iplus	50	
ipos	17	/ 27
ismall	20	
itest	58	
izero	25	
izpin	27	
izpout	26	
ln	17,192	
log	1,551	
pi	87	
rad	1,566	
rnd0	82	
rnd1	793	
rndi	745	
rndseed	42	
rndx	2,974	
sin	15,483	
sqr	8,783	
tan	41,770	

9 TRIGONOMETRICAL MANIPULATIONS

The previous chapter gave typical timings for all of the BASIC subroutines. Inspection of these timings reveals the fact that trigonometrical functions are especially time consuming. There are a number of different methods which can often be used to get round this problem and this chapter explains many of them. Each method is illustrated by the polygonal circle discussed in Chapter 5, but the methods are applicable to many situations in which trigonometry is used. It should be remembered that the conventional method of drawing circles takes nearly 6 seconds in BASIC and even in machine code requires 5.5 seconds. With a little chicanery, considerable improvements on these times may be achieved. Apart from the first method which must use assembly language, BASIC is used in demonstration programs so that the methods are easier to understand.

9.1 Fixed Shapes Method

The following program illustrates a method that can be used whenever the application draws a geometric shape of fixed dimensions in a fixed position. In this method, all the coordinates to be plotted are stored as constants within the program. In the demonstration program, the two functions, XCOORD and YCOORD respectively, store away all 100 X and Y coordinates of the circle to be plotted. The program itself simply plots these points. All of the BASIC parts of the program are disposable. The generated machine code draws the circle in 0.28 seconds. Of course, this method uses a lot of memory to store coordinates (404 bytes in this example). Moreover, it is not a general purpose routine to draw many circles of different sizes. Nevertheless, it is the quickest method and is useful in many applications.

Fixed Circle Example

```
  0 MODE 0:oswrch = &FFEE
 10 DIM mc% 1000
 20 FOR pass% = 0 TO 2 STEP 2
 30 P% = mc%
 40 [OPT pass%
 50 .xcoords
 60      OPT FNXCOORD
 70 .ycoords
 80      OPT FNYCOORD
 90 .loopcnt EQUB -1
100 .plot    EQUB 25
110 .parms   EQUB 4
120          EQUD 0
130 .circle
140      LDA #&16        \ set mode
150      JSR oswrch      \ =
160      LDA #0          \ zero
170      JSR oswrch
180 .loop
190      LDX loopcnt     \ get loopcnt
200      INX             \ bump it
210      CPX #101        \ test for 101
220      BCS out         \ if > 100 out
230      STX loopcnt     \ save loopcnt
240      TXA             \ put X in A
250      ASL A           \ * 2
260      TAX             \ back in X
270      LDA xcoords,X   \ get xcoord lo
280      STA parms+1
290      LDA ycoords,X   \ get ycoord lo
300      STA parms+3
310      INX             \ bump X
320      LDA xcoords,X   \ get xcoord hi
330      STA parms+2
340      LDA ycoords,X   \ get ycoord hi
350      STA parms+4
360      LDX #0          \ reset plot loop
370 .plotloop
380      LDA plot,X      \ next byte of PLOT
390      JSR oswrch      \ PLOT it
400      INX             \ bump X
410      CPX #6          \ end of plot
420      BCC plotloop    \ no - back
430      LDA #5          \ reset for PLOT
440      STA parms
450      JMP loop        \ next PLOT
460 .out
470      RTS             \ bye bye
480 ]
490 NEXT pass%
```

```
500 TIME=0
510 CALL circle
520 T = TIME / 100
530 PRINT
540 PRINT "time taken = ";T;" seconds"
550 END
560 DEF FNXCOORD
570 FOR J%= 0 TO 100
580     X%=INT(600+400*COS(J%*2*PI/100))
590     ?P% = ?&460
600     P%  = P% + 1
610     ?P% = ?&461
620     P%  = P% + 1
630     NEXT
640 = pass%
650 DEF FNYCOORD
660 FOR J% = 0 TO 100
670     Y%=INT(500+400*SIN(J%*2*PI/100))
680     ?P% = ?&464
690     P%  = P% + 1
700     ?P% = ?&465
710     P%  = P% + 1
720     NEXT
730 = pass%
```

9.2 Reduced Accuracy Method

In this method the number of sides in the polygon is reduced. The following BASIC program has only 32 sides in the polygon. The circle so drawn is quite adequate and is completed in 1.99 seconds:

```
 10 TIME = 0
 20 MODE 0
 30 xcentre = 600
 40 ycentre = 500
 50 increment = 2*PI/32
 60 stop = 2*PI
 70 radius = 400
 80 MOVE xcentre+radius,ycentre
 90 FOR angle = 0 TO stop STEP increment
100     DRAW xcentre + radius * COS(angle) ,
             ycentre + radius * SIN(angle)
110     NEXT
120 T=TIME/100
130 PRINT
140 PRINT "time taken = ";T;" seconds"
150 END
```

9.3 Mathematical Transform Method

In this method, the mathematics of the application are transformed in order to remove the most time-consuming mathematical functions from big loops. The following trigonometrical identities frequently prove useful:

```
SIN(A+B) = SIN(A)*COS(B) + COS(A)*SIN(B)
SIN(A-B) = SIN(A)*COS(B) - COS(A)*SIN(B)
COS(A+B) = COS(A)*COS(B) - SIN(A)*SIN(B)
COS(A-B) = COS(A)*COS(B) + SIN(A)*SIN(B)
```

In the polygonal circle, the radius rotates like the sweep of a radar screen, the angle being incremented by 2*PI/N radians each time (N = number of sides in the polygon). Let us assume that at any given instant the sweeping radius is at an angle A radians. Let us also assume that 'sold' is the sine of A and 'cold' is the cosine of A at this moment. After one more increment, the radius will have swept through A+2*PI/N radians, which we will assume has a sine of 'snew' and a cosine of 'cnew'. But from the identities above:

```
SIN(A+2*PI/N) = SIN(A)*COS(2*PI/N) + COS(A)*SIN(2*PI/N)
```

or

```
snew = sold*COS(2*PI/N) + cold*SIN(2*PI/N)
```

similarly

```
cnew = cold*COS(2*PI/N) - sold*SIN(2*PI/N)
```

It is only necessary to calculate the sine and cosine of the increment, 2*PI/N, which can be done outside the big loop, since cnew and snew can then both be derived without recourse to further trigonometry.

The following program, which draws a 100-sided circle in 2.48 seconds, uses this technique. Note that at the start of the circle when the angle swept out is zero radians, the sine of zero is zero, but the cosine of zero is one (lines 90 and 100). The program is written in BASIC so that the transformation is clear. In assembly language about a further quarter of a second can be saved without any special effort.

```
 10 MODE 0
 20 TIME = 0
 30 xcentre% = 600
 40 ycentre% = 500
 50 radius%  = 400
 60 sides%   = 100
 70 sinc = SIN(2*PI/sides%)
 80 cinc = COS(2*PI/sides%)
 90 sold = 0
100 cold = 1
```

```
110 MOVE xcentre%+radius%,ycentre%
120 FOR I% = 0 TO sides%
130     snew = sold*cinc + cold*sinc
140     cnew = cold*cinc - sold*sinc
150     DRAW xcentre% + radius% * cnew ,
             ycentre% + radius% * snew
160     sold = snew
170     cold = cnew
180     NEXT
190 T=TIME/100
200 PRINT : PRINT "time taken = ";T;" seconds"
210 END
```

9.4 Symmetry Method

This method takes advantage of the symmetry of many geometric shapes. More than one point can be plotted on the basis of a single calculation. The following identities are used:

```
SIN(PI/2   + A) = + COS(A)
SIN(PI     + A) = - SIN(A)
SIN(3*PI/2 + A) = - COS(A)
SIN(2*PI   + A) = + SIN(A)

SIN(PI/2   - A) = + COS(A)
SIN(PI     - A) = + SIN(A)
SIN(3*PI/2 - A) = - COS(A)
SIN(2*PI   - A) = - SIN(A)

COS(PI/2   + A) = - SIN(A)
COS(PI     + A) = - COS(A)
COS(3*PI/2 + A) = + SIN(A)
COS(2*PI   + A) = + COS(A)

COS(PI/2   - A) = + SIN(A)
COS(PI     - A) = - COS(A)
COS(3*PI/2 - A) = - SIN(A)
COS(2*PI   - A) = + COS(A)

TAN(PI/2   + A) = - 1/TAN(A)
TAN(PI     + A) = + TAN(A)
TAN(3*PI/2 + A) = - 1/TAN(A)
TAN(2*PI   + A) = + TAN(A)

TAN(PI/2   - A) = + 1/TAN(A)
TAN(PI     - A) = - TAN(A)
TAN(3*PI/2 - A) = + 1/TAN(A)
TAN(2*PI   - A) = - TAN(A)
```

Thus the polygonal circle can be plotted four points at a time, for example. As the radius sweeps round, not only the point at the current angle, but also points further round by 90 degrees (PI/2 radians), 180 degrees (PI radians) and 270 degrees (3*PI/2 radians) can be plotted without further trigonometry.

At any given time, if the angle swept out by the radius is A radians, the four coordinates will be:

```
X1 = xcentre% + radius%*COS(A)
Y1 = ycentre% + radius%*SIN(A)

X2 = xcentre% + radius%*COS(PI/2+A)
Y2 = ycentre% + radius%*SIN(PI/2+A)

X3 = xcentre% + radius%*COS(PI+A)
Y3 = ycentre% + radius%*SIN(PI+A)

X4 = xcentre% + radius%*COS(3*PI/2+A)
Y4 = ycentre% + radius%*SIN(3*PI/2+A)
```

The identities above give the following:

```
X1 = xcentre% + radius%*COS(A)
Y1 = ycentre% + radius%*SIN(A)

X2 = xcentre% - radius%*SIN(A)
Y2 = ycentre% + radius%*COS(A)

X3 = xcentre% - radius%*COS(A)
Y3 = ycentre% - radius%*SIN(A)

X4 = xcentre% + radius%*SIN(A)
Y4 = ycentre% - radius%*COS(A)
```

Thus it can be seen that four points can be plotted for a single value of A, knowing merely the sine and cosine of A. The four X coordinates are stored in an array called X, while the four Y coordinates are stored in an array called Y. Lines 70 to 160 in this program initialise both arrays with values corresponding to an angle of zero radians. The BASIC program, plotting four points at a time, draws a 100 sided circle in 3.11 seconds:

```
 10 MODE 0
 20 TIME = 0
 30 xcentre% = 600
 40 ycentre% = 500
 50 radius%  = 400
 60 sides%   = 100
 70 DIM X(4)
 80 DIM Y(4)
 90 X(1) = xcentre% + radius%
100 X(2) = xcentre%
110 X(3) = xcentre% - radius%
120 X(4) = xcentre%
130 Y(1) = ycentre%
140 Y(2) = ycentre% + radius%
150 Y(3) = ycentre%
160 Y(4) = ycentre% - radius%
170 FOR I% = 0 TO sides% DIV 4
180     sinA = SIN(2*PI*I%/sides%)
190     cosA = COS(2*PI*I%/sides%)
```

```
200      MOVE X(1),Y(1)
210      X(1) = xcentre%+radius%*cosA
220      Y(1) = ycentre%+radius%*sinA
230      DRAW X(1),Y(1)
240      MOVE X(2),Y(2)
250      X(2) = xcentre%-radius%*sinA
260      Y(2) = ycentre%+radius%*cosA
270      DRAW X(2),Y(2)
280      MOVE X(3),Y(3)
290      X(3) = xcentre%-radius%*cosA
300      Y(3) = ycentre%-radius%*sinA
310      DRAW X(3),Y(3)
320      MOVE X(4),Y(4)
330      X(4) = xcentre%+radius%*sinA
340      Y(4) = ycentre%-radius%*cosA
350      DRAW X(4),Y(4)
360      NEXT
370 T = TIME/100
380 PRINT
390 PRINT "time taken = ";T;" seconds"
400 END
```

This can be further improved. The circle can be drawn in 2.08 seconds by plotting eight points at a time for each of the following angles:

```
       A radians
  PI/2 - A radians
  PI/2 + A radians
  PI   - A radians
  PI   + A radians
3*PI/2 - A radians
3*PI/2 + A radians
2*PI   - A radians
```

Note that the number of sides is increased to 104 simply to make it divisible by 8. Note also that in this example, the eight X and Y coordinates are not saved in an array, but are re-worked each time. This is simply an alternative method.

The eight sets of coordinates to be plotted are:

```
angle A          X1 = xcentre% + radius%*COS(A)
                 Y1 = ycentre% + radius%*SIN(A)

angle PI/2 - A   X2 = xcentre% + radius%*SIN(A)
                 Y2 = ycentre% + radius%*COS(A)

angle PI/2 + A   X3 = xcentre% - radius%*SIN(A)
                 Y3 = ycentre% + radius%*COS(A)

angle PI - A     X4 = xcentre% - radius%*COS(A)
                 Y4 = ycentre% + radius%*SIN(A)

angle PI + A     X5 = xcentre% - radius%*COS(A)
                 Y5 = ycentre% - radius%*SIN(A)
```

```
angle 3*PI/2 - A  X6 = xcentre% - radius%*SIN(A)
                  Y6 = ycentre% - radius%*COS(A)

angle 3*PI/2 + A  X7 = xcentre% + radius%*SIN(A)
                  Y7 = ycentre% - radius%*COS(A)

angle 2*PI - A    X8 = xcentre% + radius%*COS(A)
                  Y8 = ycentre% - radius%*SIN(A)
```

```
 10 TIME=0
 20 MODE 0
 30 xcentre% = 600
 40 ycentre% = 500
 50 radius%  = 400
 60 sides%   = 104
 70 oldsin = 0
 80 oldcos = radius%
 90 FOR I% = 0 TO sides% DIV 8
100     sin = radius%*SIN(2*PI*I%/sides%)
110     cos = radius%*COS(2*PI*I%/sides%)
120     MOVE  xcentre%+oldcos,ycentre%+oldsin
130     DRAW  xcentre%+cos,ycentre%+sin
140     MOVE  xcentre%+oldsin,ycentre%+oldcos
150     DRAW  xcentre%+sin,ycentre%+cos
160     MOVE  xcentre%-oldsin,ycentre%+oldcos
170     DRAW  xcentre%-sin,ycentre%+cos
180     MOVE  xcentre%-oldcos,ycentre%+oldsin
190     DRAW  xcentre%-cos,ycentre%+sin
200     MOVE  xcentre%-oldcos,ycentre%-oldsin
210     DRAW  xcentre%-cos,ycentre%-sin
220     MOVE  xcentre%-oldsin,ycentre%-oldcos
230     DRAW  xcentre%-sin,ycentre%-cos
240     MOVE  xcentre%+oldsin,ycentre%-oldcos
250     DRAW  xcentre%+sin,ycentre%-cos
260     MOVE  xcentre%+oldcos,ycentre%-oldsin
270     DRAW  xcentre%+cos,ycentre%-sin
280     oldsin = sin
290     oldcos = cos
300     NEXT
310 PRINT
320 T = TIME/100
330 PRINT "time taken = ";T;" seconds"
```

9.5 Hybrid Method

This method combines the previous three methods. The following example program will plot a 32-sided circle in 0.7 seconds (under 0.6 seconds if written in assembly language). Eight plots are performed at a time, reducing the size of the main loop to 4. Even so, only 1 sine and 1 cosine are calculated, and these are of course outside the main loop.

```
 10 TIME=0
 20 MODE 0
 30 xcentre% = 600
 40 ycentre% = 500
 50 radius%  = 400
 60 sides%   = 32
 70 oldsin = 0
 80 oldcos = 1
 90 sinc   = SIN(2*PI/sides%)
100 cinc   = COS(2*PI/sides%)
110 FOR I% = 0 TO sides% DIV 8
120     newsin = oldsin*cinc + oldcos*sinc
130     newcos = oldcos*cinc - oldsin*sinc
140     sinold = radius%*oldsin
150     sinnew = radius%*newsin
160     cosold = radius%*oldcos
170     cosnew = radius%*newcos
180     X1%    = xcentre%+cosold
190     X2%    = xcentre%+sinold
200     X3%    = xcentre%-cosold
210     X4%    = xcentre%-sinold
220     X5%    = xcentre%+cosnew
230     X6%    = xcentre%+sinnew
240     X7%    = xcentre%-cosnew
250     X8%    = xcentre%-sinnew
260     Y1%    = ycentre%+cosold
270     Y2%    = ycentre%+sinold
280     Y3%    = ycentre%-cosold
290     Y4%    = ycentre%-sinold
300     Y5%    = ycentre%+cosnew
310     Y6%    = ycentre%+sinnew
320     Y7%    = ycentre%-cosnew
330     Y8%    = ycentre%-sinnew
340     MOVE X1%,Y2%
350     DRAW X5%,Y6%
360     MOVE X2%,Y1%
370     DRAW X6%,Y5%
380     MOVE X4%,Y1%
390     DRAW X8%,Y5%
400     MOVE X3%,Y2%
410     DRAW X7%,Y6%
420     MOVE X3%,Y4%
430     DRAW X7%,Y8%
```

```
440      MOVE X4%,Y3%
450      DRAW X8%,Y7%
460      MOVE X2%,Y3%
470      DRAW X6%,Y7%
480      MOVE X1%,Y4%
490      DRAW X5%,Y8%
500      oldsin = newsin
510      oldcos = newcos
520      NEXT
530 T = TIME/100
540 PRINT
550 PRINT "time taken = ";T;" seconds"
```

10 LARGE MACHINE CODE PROGRAMS

A machine code program can occupy all the memory space between PAGE and HIMEM, and in some cases other areas below PAGE as well. In a typical game program using one of the 20K graphics modes, if the game is disk based, the machine code will occupy from &1900 to &2FFF. This is a little under 6K. Although this size can often be increased by pinching memory areas from &400 to &6FF, and from &A00 to &CFF, the overall machine code size is still small and code must be economic.

However, even though this is undoubtedly a problem, it is not the only one or even the main one. The reason for this is that the BBC BASIC/assembler requires the source code to be present in memory for the assembly step. Thus the amount of machine code that can be written in a single go is considerably less than the amount of memory available. Since the technique described in this book is particularly valuable to those wishing to write large machine code programs, it is pertinent to consider the various ways in which this problem may be overcome.

All the methods require that the overall program be subdivided into a number of smaller modules each of which is small enough to assemble. From each of these smaller modules, the assembled machine code is extracted, whilst the source code itself is kept quite separately in case further amendment is required. There are then two choices. Either the individual machine code extracts are merged together to produce a single, large machine code program. This is the case if the machine code is to be blown on ROM. Alternatively, a BASIC program is written which loads the main machine code module and passes control to the appropriate entry point address. This is the more usual method for tape/disk based games, where the initial BASIC program usually incorporates the manufacturer's logo.

The major problem in all this is addressing. Usually a piece of machine code assembled in one area of memory has to be moved to another area of memory. This is called 'relocation'. But 6502 machine code is not usually relocatable in this way and special steps have to be taken to overcome this fact. The problem subdivides into two distinctly different relocation problems.

Firstly there is the problem of relocation of an individual module. The problems associated with this are called intra-module relocation problems.

Secondly, and much tougher, is the problem of cross-references between individual modules, giving rise to inter-module relocation problems. If Module-A calls a subroutine in Module-B, then whenever Module-B is moved, the address of that subroutine changes and so must all references to it in Module-A. A similar problem arises with any data fields which are referenced in more than one module.

10.1 BASIC 2 Relocation

Those in possession of a BASIC 2 ROM have a tailor-made way of solving intra-module relocation problems in the form of an extended range of OPT codes. When 4 is added to the usual values for OPT, the assembler will assemble the program in an area of memory defined by the contents of O%, but the code will be assembled as though it were located in an area of memory defined by the contents of P%. A simple example should illustrate this well. The module below contains a simple subroutine to divide a number by 2. At line 20, instead of the normal values for pass% of 0 and 3, 4 has been added to each to specify that relocation is wanted. At line 40, O% has been set appropriately so that the machine code will be physically located in the area mc%. At line 30, P% has been set to PAGE, so the assembler will assemble the machine code as though it were located at PAGE. The BASIC lines at 240 to 250 help in saving this machine code.

```
 10DIM mc% 1000
 20FOR pass% = 4 TO 7 STEP 3
 30P%  = PAGE
 40O%  = mc%
 50[OPT pass%
 60\ routine to divide a number by 2
 70\ on entry the number is in A
 80\ on exit  the result is in X and
 90\          remainder  is in Y.
100.div2
110   PHA            \ save A
120   LDY #0         \ zeroise remainder
130   LSR A          \ divide A by 2
140   TAX            \ result in X
150   BCC out        \ no remainder
160   LDY #1         \ set remainder
170.out
180   PLA            \ restore A
190   RTS            \ bye bye
200.finish
210]
220NEXT pass%
230PRINT
```

```
240PRINT " code is at &";~mc%;" and is &";
245PRINT ;~(finish-div2);" bytes long"
250PRINT "relocate at &";~div2
260END
```

The result of running this module on a system with disks (PAGE = &1900) is shown below. Notice that the machine code physically starts at &1B96 and is &B bytes long. The assembly listing shows that the machine code has been assembled to run at &1900, however.

```
1900        OPT pass%
1900        \ routine to divide a number by 2
1900        \ on entry the number is in A
1900        \ on exit  the result is in X and
1900        \          remainder  is in Y.
1900        .div2
1900 48     PHA             \ save A
1901 A0 00  LDY #0          \ zeroise remainder
1903 4A     LSR A           \ divide A by 2
1904 AA     TAX             \ result in X
1905 90 02  BCC out         \ no remainder
1907 A0 01  LDY #1          \ set remainder
1909        .out
1909 68     PLA             \ restore A
190A 60     RTS             \ bye bye
190B        .finish

 code is at &1B96 and is &B bytes long
relocate at &1900
```

To save this machine code we must use:

```
*SAVE "DIV2" 1B96 +B 1900
```

This entire module can be relocated to any valid address that we wish, simply by altering line 30 in the original assembly language program, re-assembling and saving the machine code once again. Notice, though, that only intra-module relocation problems are solved in this way. Any other modules that wish to call ‘div2’ will have to do so by absolute addressing, such as JSR &1900. If we move the start address of ‘div2’ we create an inter-module relocation problem. More will be said of this later, but for the moment let us see how BASIC 1 owners can cope with intra-module relocation problems.

10.2 Intra-Module Relocation Problems

Many 6502 instructions will relocate quite happily without any correction by the programmer. The previous example program, for example, has no problems. It is the machine code instructions which reference absolute addresses within the domain of the overall program which cause problems. The method to be described in this section is often used by BASIC 2 owners in preference to the method outlined in the previous section.

The idea behind the method is to identify those bytes within a module which are not relocatable. Of course, this can be done manually by simple inspection of the program code, but this would be prone to error. It is much safer to let the micro identify the bytes which will not relocate. All that is needed is to assemble the module at two different addresses and compare the two machine code modules byte-for-byte. Those bytes which are different are the bytes which will not relocate. A separate section will later suggest ways in which the number of un-relocatable bytes can be minimised. One of these ways is so important that it needs to be mentioned now. It is a good idea to start each module on a page boundary. In this way, intra-module relocation problems are reduced to high address bytes only.

Consider the previous example program. By changing it slightly, so that the result and remainder are stored in memory areas referenced by labels, we introduce some intra-module relocation problems. To identify where those problems are, we can assemble the program twice at non-overlapping addresses on different page boundaries and test which bytes of the machine code have changed.

```
10MODE7
15REM assemble at &2000
20I% = &2000
30PROCasm
35PRINT : REM assemble at &2100
40I% = &2100
50PROCasm
55PRINT
60FOR I% = 0 TO finish-start
70    IF ?(&2000+I%) = ?(&2100+I%) THEN GOTO 90
80    PRINT "problem at &";~(&2000+I%) ;
85    PRINT "   value = &";~?(&2000+I%)
90    NEXT
100END
110DEF PROCasm
120FOR pass% = 0 TO 3 STEP 3
130P% = I%
140[OPT pass%
150\ routine to divide a number by 2
```

```
160\ on entry the number is in A
170\ on exit  the result is in 'result' and
180\          remainder  is in 'remainder'.
190.start
200.result    EQUB 0
210.remainder EQUB 0
220.div2
230   PHA           \ save A
240   LDA #0        \ zeroise
250   STA remainder \ remainder
260   PLA           \ get A again
270   PHA           \ save A again
280   LSR A         \ divide A by 2
290   STA result    \ save result
300   BCC out       \ no remainder
310   LDA #1        \ set
320   STA remainder \ remainder
330.out
340   PLA           \ restore A
350.finish
360   RTS           \ bye bye
370]
380NEXT pass%
390ENDPROC
```

When the program is run, we obtain a list of the problem areas. Normally this would be done with OPT set at 2, but in this case listings have been obtained for both assemblies so that the reader can verify that the results are correct.

```
2000          OPT pass%
2000          \ routine to divide a number by 2
2000          \ on entry the number is in A
2000          \ on exit  the result is in 'result' and
2000          \          remainder  is in 'remainder'.
2000          .start
2000 00       .result    EQUB 0
2001 00       .remainder EQUB 0
2002          .div2
2002 48       PHA           \ save A
2003 A9 00    LDA #0        \ zeroise
2005 8D 01 20 STA remainder \ remainder
2008 68       PLA           \ get A again
2009 48       PHA           \ save A again
200A 4A       LSR A         \ divide A by 2
200B 8D 00 20 STA result    \ save result
200E 90 05    BCC out       \ no remainder
2010 A9 01    LDA #1        \ set
2012 8D 01 20 STA remainder \ remainder
2015          .out
2015 68       PLA           \ restore A
2016          .finish
2016 60       RTS           \ bye bye
```

```
2100          OPT pass%
2100          \ routine to divide a number by 2
2100          \ on entry the number is in A
2100          \ on exit  the result is in 'result' and
2100          \          remainder  is in 'remainder'.
2100          .start
2100 00       .result    EQUB 0
2101 00       .remainder EQUB 0
2102          .div2
2102 48       PHA           \ save A
2103 A9 00    LDA #0        \ zeroise
2105 8D 01 21 STA remainder \ remainder
2108 68       PLA           \ get A again
2109 48       PHA           \ save A again
210A 4A       LSR A         \ divide A by 2
210B 8D 00 21 STA result    \ save result
210E 90 05    BCC out       \ no remainder
2110 A9 01    LDA #1        \ set
2112 8D 01 21 STA remainder \ remainder
2115          .out
2115 68       PLA           \ restore A
2116          .finish
2116 60       RTS           \ bye bye
```

```
problem at &2007   value = &20
problem at &200D   value = &20
problem at 82014   value = &20
```

Thus we can relocate this program on any page boundary providing we change just 3 bytes. For example, to relocate the machine code at &1500, we would simply enter:

```
*LOAD "DIV2" 1500
?&1507 = &15
?&150D = &15
?&1514 = &15
```

10.3 Intra-Module General Case

The previous section illustrated the way in which intra-module relocation problems can be identified. Generally, however, there will be insufficient memory to assemble the module twice in situ. In these cases it is necessary to break the problem into several steps.

Firstly, assemble the module at one page boundary and *SAVE the machine code. Secondly, assemble the module at a different page boundary and *SAVE the machine code again. Then *LOAD each version separately to non-overlapping addresses. Finally compare each byte-for-byte and report on differences.

step 1.

The example program is assembled at &2000 and *SAVEd as 'DIV2A'. This is done by sandwiching the module between a few BASIC instructions. OSCLI (&FFF7) is used for the *SAVE, but it is written in such a way that it works on BASIC 1 and 2 ROMs.

```
 10mod$ = "DIV2A" : begin% = &2000
 30FOR pass% = 0 TO 2 STEP 2
 40P% = begin%
 50[OPT pass%
 60\ routine to divide a number by 2
 70\ on entry the number is in A
 80\ on exit  the result is in 'result' and
 90\          remainder  is in 'remainder'.
100.result    EQUB 0
110.remainder EQUB 0
120.div2
130   PHA           \ save A
140   LDA #0        \ zeroise
150   STA remainder \ remainder
160   PLA           \ get A again
170   PHA           \ save A again
180   LSR A         \ divide A by 2
190   STA result    \ save result
200   BCC out       \ no remainder
210   LDA #1        \ set
220   STA remainder \ remainder
230.out
240   PLA           \ restore A
250   RTS           \ bye bye
260]
270NEXT pass%
280REM *SAVE this machine code
290DIM X% 256
300$X% = "SAVE " + mod$ + " " + STR$~begin% + " " + STR$~P%
     + " " + STR$~begin%
310Y% = X% DIV 256
320CALL &FFF7
330PRINT CHR$133;"*";$X% : END
```

step 2.

By changing line 10, the module is re-assembled at &2100 and *SAVEd as 'DIV2B'

```
 10mod$ = "DIV2B" : begin% = &2100
 30FOR pass% = 0 TO 2 STEP 2
 40P% = begin%
 50[OPT pass%
 60\ routine to divide a number by 2
 70\ on entry the number is in A
 80\ on exit  the result is in 'result' and
 90\          remainder  is in 'remainder'.
```

```
100.result    EQUB 0
110.remainder EQUB 0
120.div2
130   PHA           \ save A
140   LDA #0        \ zeroise
150   STA remainder \ remainder
160   PLA           \ get A again
170   PHA           \ save A again
180   LSR A         \ divide A by 2
190   STA result    \ save result
200   BCC out       \ no remainder
210   LDA #1        \ set
220   STA remainder \ remainder
230.out
240   PLA           \ restore A
250   RTS           \ bye bye
260]
270NEXT pass%
280REM *SAVE this machine code
290DIM X% 256
300$X% = "SAVE " + mod$ + " " + STR$~begin% + " " + STR$~P%
     + " " + STR$~begin%
310Y% = X% DIV 256
320CALL &FFF7
330PRINT CHR$133;"*";$X% : END
```

step 3.

The following program, called 'INTRA', is a general purpose program for comparing machine code modules up to &1000 bytes in length. It asks the user to provide the first part of the module name (in this case 'DIV2'). It then appends an 'A' to this name and loads the machine code module into an area, load%. It then appends a 'B' to the name and loads this machine code module at load% + &1000. Finally it compares the two modules byte-for-byte and reports all differences.

```
10MODE7
20REM set aside an area to *LOAD up to &1000 bytes twice
30DIM load% &2000
40PRINT TAB(12,2);CHR$141;CHR$131;"INTRA"
50PRINT TAB(12,3);CHR$141;CHR$131;"INTRA"
60PRINT TAB(0,6);CHR$134;"enter module name ";
70INPUT "> " mod$
80A$ = "LOAD " + mod$ + "A " + STR$~load%
90B$ = "LOAD " + mod$ + "B " + STR$~(load%+&1000)
100PROCoscli(A$)
110REM get size of module loaded
120size% = 256*?&2F9 + ?&2F8
130PROCoscli(B$)
140count% = 0
150VDU2 : REM turn print on
160PRINT "Intra-module problem addresses"
```

```
170PRINT
180FOR I% = 0 TO size%-1
190    IF ?(load%+I%) <> ?(load%+I%+&1000)
       THEN PRINT "&";~I%;" from start" :
       count% = count% + 1
200    NEXT
210PRINT
220PRINT "total problems = ";count%
230VDU3 : REM turn printer off
240END
250REM ----------------------------------
260REM routine to *LOAD the machine code
270REM ----------------------------------
280DEFPROCoscli(C$)
290PRINT
300PRINT CHR$133;"*";C$
310DIM X% 256
320$X% = C$
330Y% = X% DIV 256
340CALL &FFF7
350ENDPROC
```

When the program is run on our example modules, it produces the following results:

```
Intra-module problem addresses

&7 from start
&D from start
&14 from start

total problems = 3
```

These are exactly the results that we expected. By inspecting the assembler listing for this module, we can determine what value with respect to the origin page of the module must be inserted into each of these bytes in order to make the module relocatable. In this example the values are all the same as the origin page. If this is the main module, these fixes must be coded into the initial BASIC program. Otherwise they are coded into the main module itself.

10.4 Minimising Intra-Module Problems

Clearly, the less intra-module relocation problems that are present in the machine code, the easier it becomes to amend and relocate that code. A few simple rules can minimise these problems:

a) Start each module on a page boundary.

b) Avoid JMP instructions whenever possible. Use branch instructions instead as these are always relocatable. For example:

```
CLC
BCC label
```

instead of

```
JMP label
```

c) It is frequently necessary to pass parameters from one subroutine to another. Parameters that are passed via zero page locations, the resident integer variables, the 6502 registers or the 6502 stack cause no problems. However, using data fields within the domain of the module will cause problems.

d) Document and test all code thoroughly before saving the machine code. This is an advisable practice, even for single-module programs. For multi-module programs it is more-or-less obligatory because it is so much more difficult to amend program code. It is frequently possible to test individual subroutines by driving data through them from BASIC. The example programs in this book do this quite a lot.

10.5 Inter-Module Relocation Problems

All that has been achieved so far is that individual modules can be made relocatable. The harder problem has now to be addressed; how can one module communicate with another module?

In the discussion that follows, we will assume that only two modules are needed for the application. When more modules are required, the same principles apply. The principles are, however, easier to explain with just two modules.

Once again, the problem becomes easier to manage when we subdivide it into its component parts. The inter-module communications problem can be broken into code and data problems. In the case of code problems we are concerned with how to JMP or JSR from one module to another. This problem becomes really simple if we specify certain design constraints on the two modules. For example, let us specify that there will be a main module, called 'MAIN' and a module of subroutines, called 'SUBS'. In this scheme, MAIN will be located in a memory area that will allow it to co-exist with the initial BASIC program. SUBS will overlay the initial BASIC program at PAGE and must be loaded by MAIN. Let us further specify that:

```
MAIN can call routines in MAIN
MAIN can call routines in SUBS
SUBS can call routines in SUBS BUT not in MAIN
```

Already we have simplified the problem considerably. The key steps can now be taken. Let us assume that SUBS contains three subroutines (sub1, sub2, sub3) and that these are each called in several places from within MAIN. If we use say &70 to specify which subroutine we want, then there need only be one entry point in SUBS.

In this example code, the only entry point into SUBS is at the start, at 'subs'. The first piece of code resolves which subroutine is actually required. In practice, this section of code would probably also save the register set.

```
 10REM SUBS
 15DIM mc% &1000
 20FOR pass% = 0 TO 2 STEP 2
 30P% = mc%
 40[OPT pass%
 50.subs
 60   LDA &70       \ get which sub
 70   CMP #1        \ is it sub1 ?
 80   BEQ sub1      \ yes - sub1
 90   CMP #2        \ is it sub2 ?
100   BEQ sub2      \ yes - sub2
110   CMP #3        \ is it sub3 ?
120   BEQ sub3      \ yes - sub3
130   BRK           \ mistake
140.sub1
150   JMP dosub1    \ DO sub1
160.sub2
170   JMP dosub2    \ DO sub2
180.sub3
190   JMP dosub3    \ DO sub3
200.dosub1
      :
      : assembler
      : code
      :
900]:NEXT pass%
910END
```

Let us assume that our first guess is that MAIN will be located at &2800. We know that MAIN will have to load SUBS at PAGE and also handle all its intra-module relocation fixes. We can also design

MAIN so that it only calls SUBS in one place. The entry point into this program is at ‘main’.

```
 10REM MAIN
 20FOR pass% = 0 TO 2 STEP 2
 30P% = &2800
 40[OPT pass%
 50.main
 60   LDA #&83            \ get PAGE
 70   JSR &FFF4
 80   STY callsub+2       \ initialise callsub
 90   STY osblok+2        \ and osfile parm block
100   LDX #osblok MOD 256 \ get lo-byte
110   LDY #osblok DIV 256 \ get hi-byte
120   LDA #&FF            \ LOAD SUBS
130   JMP (&212)          \ via OSFILE
140.back                  \ return here
      :
      : code
      : to
      : fix
      : SUBS
      : intra-module
      : relocation
      : problems
      :
      :
      : code
      :
400   LDA #1              \ call sub1
410   STA &70
420   JSR callsub
      :
      : code
      :
500   LDA #2              \ call sub2
510   STA &70
520   JSR callsub
      :
      : etc.
      : etc.
      :
700.callsub
710   JMP &0000           \ overwritten by line 80
800.osblok                \ OSFILE parm block
810   EQUW osname         \ point to file name
820   EQUD 0              \ load address (see line 90)
830   EQUD back           \ xqt address (fudge it)
840   EQUD 0
850   EQUD 0
860.osname
870   EQUS "SUBS"         \ file name
```

```
880   EQUB &D          \ CR
900]:NEXT pass%
910END
```

We have now solved all the inter-module code problems. MAIN can freely access all the subroutines in SUBS.

Data problems are even simpler to handle. We simply make sure that there are no data problems. Firstly, all data fields are defined in MAIN except for any specific work areas needed by SUBS but of which MAIN needs no knowledge. Secondly, only a routine in MAIN is allowed to access a data field in MAIN. Thirdly, parameters passed between MAIN and SUBS and vice-versa are always in zero page memory, resident integer variables, registers or the processor stack. An example should clarify matters.

Let us assume that 'sub1' is a routine in SUBS that updates the player's score. The data field 'score' is of course in MAIN. Let us also assume that 'score' is a 16 bit integer, and that &80 and &81 are used to communicate the score between MAIN and SUBS. Then the code in MAIN might well be:

```
LDA #1      \ specify sub1
STA &70     \
LDA score   \ get lo-byte
STA &80     \ save it
LDA score+1 \ get hi-byte
STA &81     \ save it
JSR callsub \ get it updated
LDA &80     \ get lo-byte
STA score   \ put it back
LDA &81     \ get hi-byte
STA score+1 \ put it back
```

In this way, 'sub1' has no knowledge of the field 'score' but can still function as a subroutine.

By defining design constraints on the individual modules, we have ensured that there are no inter-module relocation problems. The intra-module relocation problems are identified by program.

In practice, each application has to be treated on its own merits. Sometimes more modules are needed. In any event, an overall strategy such as outlined here will greatly simplify relocation problems. The strategy will ensure that the machine code programs will work correctly on both tape and disk based systems, providing all references to relocation problem fixes are relative to PAGE.

10.6 Initial BASIC Program

Let us assume that we have two modules, MAIN and SUBS designed along the lines suggested before. Let us assume that we have now discovered that MAIN should be loaded at PAGE+&800 and SUBS should be loaded at PAGE as was always intended.

We will assume also that 'INTRA' identified the following intramodule problems :–

```
MAIN (originM = page at which MAIN starts)

&15 from start should be originM
&74 from start should be originM+2
&FC from start should be originM+1

SUBS (originS = page at which SUBS starts)

&83 from start should be originS+1
```

The fix for SUBS has to be coded into MAIN. The fixes for MAIN are coded into the initial BASIC program which would look something like this:

```
 10MODE7
 20A$ = "LOAD MAIN " + STR$~(PAGE+&800)
 30DIM X% 256
 40$X% = A$
 50Y%  = X% DIV 256
 60CALL &FFF7
 70?(PAGE+&815) = (PAGE DIV 256) + 8
 80?(PAGE+&874) = (PAGE DIV 256) + 10
 90?(PAGE+&8FC) = (PAGE DIV 256) + 9
100CALL PAGE+&800
110END
```

INDEX

THE ADVANCED BASIC ROM USER GUIDE FOR THE BBC MICROCOMPUTER

This book delves deep into the BBC microcomputer BASIC 1 and BASIC 2 ROMS and comes up with 69 useful subroutines that can be called from an assembly language program. The routines cover:

- 32 bit integer arithmetic
- floating point arithmetic
- maths. functions such as sine, cosine, log, square root
- data conversions
- random numbers

The author has programmed commercially for 18 years on a wide variety of computers including in more recent years the BBC microcomputer. The book attempts to fill some of the important gaps in the microcomputer literature and covers in addition:

- making trigonometry faster
- writing large, relocatable, machine code programs
- useful pseudo-directives

There are many program examples in this book and much more besides. The serious programmer of the BBC micro will find this book to be a valuable aid.

Published by
Cambridge Microcomputer Centre
153–154 East Road, Cambridge, England.